

Bounded Wait-Free Implementation of Optimally Resilient Byzantine Storage without (Unproven) Cryptographic Assumptions

Amitanand S. Aiyer¹, Lorenzo Alvisi¹ *, and Rida A. Bazzi²

¹ Department of Computer Sciences,
University of Texas at Austin

² Department of Computer Sciences,
Arizona State University

Abstract. We present the first optimally resilient, bounded, wait-free implementation of a distributed atomic register, tolerating Byzantine readers and (up to one-third of) Byzantine servers, without the use of unproven cryptographic primitives or requiring communication among servers. Unlike previous (non-optimal) solutions, the sizes of messages sent to writers depend only on the actual number of active readers and not on the total number of readers in the system. With a novel use of secret sharing techniques combined with write back throttling we present the first solution to tolerate Byzantine readers information theoretically, without the use of cryptographic techniques based on unproven number-theoretic assumptions.

1 Introduction

Distributed storage systems in which servers are subject to Byzantine failures have been widely studied. Results vary in the assumptions made about both the system model and the semantics of the storage implementation. The system parameters include the number of clients (readers and writers), the synchrony assumptions, the level of concurrency, the fraction of faulty servers, and the faulty behavior of clients. In the absence of synchrony assumptions, atomic [8] read and write semantics are possible, but stronger semantics are not [7]. We consider implementations with atomic semantics in this paper.

We consider solutions in an asynchronous system of n servers that do not communicate with each other (non-communicating servers) and in which up to f servers are subject to Byzantine failures (f -resilient), any number of clients can fail by crashing (wait-free), and readers can be

* This work was supported in part by NSF awards CSR—PDOS 0509338 and CyberTrust 043051.

subject to Byzantine failures. Systems in which servers do not communicate with each other are interesting because solutions that depend on communication between servers tend to have high message complexity, quadratic in the number of servers [10, 4].

In the non-communicating servers model, the best previous solution that provides wait-free atomic semantics requires $4f + 1$ servers [3]. That solution (i) requires clients and servers to exchange a finite number of messages and (ii) limits the size of the messages sent by the servers to the readers: the size of these messages is bound by a constant times the logarithm of the number of write operations performed in the system—or, equivalently, by a constant times the size of a timestamp. Unfortunately, this solution allows messages sent to writers to be as large as the maximum number of potential readers in the system, even during times when the number of *actual* readers is small. Recently, and concurrently with our work, a wait-free atomic solution that requires not more than $3f + 1$ servers was proposed, but that solution requires unbounded storage, message of unbounded size, and an unbounded number of messages per read operation [6].

None of these solutions consider Byzantine readers. Byzantine behavior of readers is relevant because wait-free atomic solutions require that readers write to servers [5]. All existing work that considers Byzantine readers uses cryptographic techniques based on unproven number-theoretic assumptions [4, 9].

So, the existing results leave open two fundamental questions:

- Is the additional cost of f replicas over the optimal for unbounded solutions required to achieve a bounded wait-free solution?
- Is the use of cryptographic techniques required to tolerate Byzantine readers?

We answer both questions in the negative. We show that tolerating Byzantine readers can be achieved with information-theoretic guarantees and without the use of unproven number-theoretic assumptions. We also show that a bounded wait-free implementation of a distributed storage with atomic semantics is possible for $n = 3f + 1$ (which is optimal). Our solution also bounds the size of messages sent to writers—a significant improvement over Bazzi and Ding’s non-optimal solution [3].

To achieve our results, we refine existing techniques and introduce some new techniques. The ideas we refine include *concurrent-reader detection* and *write-back throttling*, originally proposed in the atomic wait-free solution of Bazzi and Ding [3]. In what follows we give a high level overview of the new techniques we introduce.

Increasing resiliency We increase the resiliency of our solution by introducing a new way by which a reader selects the timestamp of the value it will try to read. Instead of choosing the $f + 1$ 'st largest among the received timestamps, in our protocol the reader chooses the $2f + 1$ 'st *smallest*. In fact, we realized that the $f + 1$ largest timestamp worked well for $n = 4f + 1$ simply because, for that value of n , the $f + 1$ largest received timestamp coincides with the $2f + 1$ smallest. We guarantee the liveness of our new selection process by having the reader continuously update the value of the $2f + 1$ 'st smallest timestamp as it receives responses from new servers.

Bounding message sizes to writers We bound the sizes of messages sent to servers using three rounds of communication between writers and servers. These rounds occur in parallel with the first two rounds of the write protocol and no server receives a total of more than two messages across the three rounds. In the first round, the writer estimates the number of concurrent readers; in the second and third rounds it determines their identities.

Tolerating Byzantine readers We use write back throttling combined with secret sharing to tolerate Byzantine readers. The idea is to associate a random secret with each write and share the secret among the servers in such a way that it can only be reconstructed if enough servers reveal their shares. By requiring that a correct server only divulge its share if the write has made sufficient progress, we use a reader's ability to reconstruct the secret as a proof that the reader is allowed to write back. By using secret sharing, we avoid relying on unproven number theoretic assumptions and achieve instead information-theoretic guarantees.

2 Model/Assumptions

The system consists of a set of n replicas (servers), a set of m writers and a set of readers. Readers and writers are collectively referred to as clients. Clients have unique identifiers that are totally ordered. When considering boundedness of the sizes of messages, we assume that a read operation in the system can be uniquely identified with a finite bit string (otherwise any message sent by a reader can be unbounded in size). The identifier consists of a reader identifier and a read operation tag. Similarly write operations are identified by the writer identifier and the timestamp of the value being written. Since timestamps are non-skipping [2], writes can also be represented by finite strings.

Clients execute protocols that specify how *read* and *write* operations are implemented. We assume that clients do not start a new operation before finishing a previous operation. We assume that up to f servers may deviate arbitrarily from the specified protocol (Byzantine) and that the remaining $(n - f)$ servers are correct. We require that the total number of servers n be at least $3f + 1$.

We assume that messages cannot be spoofed. While this is typically enforced in practice using digital signatures, based on public key cryptography, such techniques are not necessarily required to enforce our requirement. We assume FIFO point-to-point asynchronous channels between clients and servers. Servers do not communicate with other servers.

Writers are benign and can only fail by crashing. In Section 3 we also assume that the readers are benign; we relax this assumption in Section 5 where we consider Byzantine readers. When considering Byzantine readers, we make the additional assumption that the channels between the servers and the writers are private i.e. messages sent over these channels cannot be eaves-dropped by the adversary.

For our implementation, the probability that a given read operation by a Byzantine reader improperly writes back a value is 2^{-k} where k is a security parameter. We choose k to be sufficiently large so that the probability of failure for all operations is small. If $k = o + k'$ bits, where o is the number of bits required to represent one operation, then the system failure probability is $2^{-k'}$.

Schemes based on public key cryptography, in the best case, also suffer from this negligible small probability of error. If the unproven assumptions that they are based upon do not hold, their probability of error can be significantly larger.

3 Bounded Atomic Register

We present a single-writer protocol that implements a wait-free atomic register using $3f + 1$ replicas where the size and the number of messages exchanged per operation is bounded.

Figures 1–3 present a single-writer-multiple-reader version of the protocol that assumes that the readers are benign. In Section 5 we show how to extend this protocol to handle Byzantine readers. These protocols can also be easily extended to support multiple-writers, using ideas from [3]. We refer the reader to [1] for proofs and a more detailed discussion.

3.1 Protocol Overview

The write operation The write operation is performed in two phases.

In phase 1, the writer sends the value to all the servers and waits for $(n - f)$ acknowledgements. The writer also initiates, in parallel, the GetConcurrentReaders protocol to detect concurrent readers. The GetConcurrentReaders is a bounded protocol, described in Section 3.3, that detects all read operations which are considered to be active at all the non-faulty servers, when the protocol is executed.

In phase 2, the writer asks all the servers to update their *current timestamp*, and to forward the values that they have to all the concurrent readers detected in phase 1. On receiving $(n - f)$ acknowledgements, the write operation completes.

This two-phase mechanism guarantees that if a non-faulty server updates its current timestamp, then at least $f + 1$ non-faulty servers must have already received the value.

The read operation To understand the reader's protocol, we consider a simple scenario. The reader starts by requesting second phase information from the servers. Each server replies with the most current timestamp for which it knows that the corresponding write operation reached its second phase. Now, assume that the reader receives replies from all correct servers in response to its request for second phase information. The timestamps returned by these correct servers can be quite different because the reader's requests could reach them at different times and the writer could have executed many write operations during that time. Of special interest is the largest second phase timestamp returned by a correct server. Let us call that timestamp $t_{largest}$. If the writer executes no write operation after its write of $t_{largest}$, then, when the reader receives the second phase response with $t_{largest}$, it can simply request all first phase messages and be guaranteed to receive $f + 1$ replies with identical value v and timestamp $t_{largest}$; at that time, the reader would be able to determine that, by reading v , it would not violate atomic semantics. The reader then writes back the value and then the timestamp in two phases to complete the read operation.

While this scenario is instructive, it overlooks some complications. For instance, a fast writer might write many values with timestamps larger than $t_{largest}$. Also, the reader does not know when it has received replies from all correct servers. If we assume, for now, that the reader *can* tell when it has received values from all correct servers, then we can solve the problems caused by a fast writer by having the fast writer help the

```

write() {
  inc(ts)
  // Phases W1
  cobegin {
    writeVal();
    CR = GetConcurrentReaders()
  } coend

  // Phase W2
  send (WRITE_TS, ts, CR) to all
  wait for (n - f) acks.
}
writeVal() {
  send (WRITE_VAL, <v, ts>) to all
  wait for (n - f) acks.
}

```

Fig. 1. The Writer’s Protocol

reader to terminate. This is done by having the writer detect concurrent read operations and then have the writer request from the server to *flush out* the written value to concurrent readers. Our solution requires that servers keep the 3 most up to date written values because the detection of concurrent readers is only guaranteed when the writer completely writes a value whose timestamp is larger than $t_{largest} + 1$.

There remains the problem of the reader not knowing when it has received replies from all correct servers. In fact, in response to its request for second phase information, the reader can receive replies only from $n - f$ servers— f of which may be faulty—and it might not be able to terminate based on these responses. We handle this situation by simply *assuming* that these $n - f$ messages are all from correct servers. If they indeed are, then the reader will for sure be able to decide on $t_{largest}$ by requesting the first phase information (it is possible that the reader will be able to decide even if they are not correct). If, however, the reader is unable to decide, then there are other correct servers whose replies are not amongst the $n - f$ replies, and, by waiting long enough, the reader will eventually receive some message from one of those servers. When an undecided reader receives a new message, it recalculates $t_{largest}$ assuming that, with the new messages it received, it must finally have replies from all correct servers: therefore, the reader re-requests the first phase information from all servers. This process continues until the reader indeed receives replies from all correct servers, in which case, it is guaranteed to decide.

Finally, in the above discussion we have assumed that the reader knows what $t_{largest}$ is—in reality, in our protocol the reader can only estimate $t_{largest}$ by using the $2f + 1$ ’st smallest second phase timestamp. We can show that this is sufficient to guarantee that the reader can decide and that its decision is valid [1].

3.2 Protocol Guarantees

The protocol presented provides atomic semantics. The reader and the writer protocols always terminate and are wait-free.

Definitions:
 $\text{valid}(\langle v, ts \rangle) \triangleq |\{s : \langle v, ts \rangle \in \text{Values}[s]\}| \geq f + 1$
 $\text{notOld}(\langle v, ts \rangle) \triangleq |\{s : \text{last_comp}[s] \leq ts\}| \geq 2f + 1$
 $\text{fwded}(\langle v, ts \rangle) \triangleq |\{s : \text{fwd}[s] = \langle v, ts \rangle\}| \geq f + 1$

```

read() {
  Vs: last_comp[s] =  $\perp$ ; fwd[s] =  $\perp$ ; Values[s] =  $\emptyset$ 
  // Phase R1
  send (GET-TS) to all

  repeat
    on receive (TS, s, ts) from server s
      last_comp[s] = ts
    on receive (FWD, s,  $\langle v, ts \rangle$ , Vals) from server s
      fwd[s] =  $\langle v, ts \rangle$ 
      Values[s] = Values[s]  $\cup$  Vals
  until ( $|\{x : \text{last\_comp}[x] \neq \perp\}| \geq n - f$ )

  // Phase R2
  send (GET-VAL) to all
  repeat
    on receive (TS, s, ts) from server s
      last_comp[s] = ts
      send (GET-VAL) to all
    on receive (VALS, s, Vals) from server s
      Values[s] = Values[s]  $\cup$  Vals
    on receive (FWD, s,  $\langle v, ts \rangle$ , Vals) from server s
      fwd[s] =  $\langle v, ts \rangle$ 
      Values[s] = Values[s]  $\cup$  Vals
  until ( $\exists (v_c, ts_c) : \text{fwded}(\langle v_c, ts_c \rangle) \wedge (\text{notOld}(\langle v_c, ts_c \rangle) \wedge \text{valid}(\langle v_c, ts_c \rangle))$ )

  // Phase R3
  WriteBack( $ts_c$ )
  return  $\langle v_c, ts_c \rangle$ 
}

WriteBack( ts ) {
  // Round 1
  send (WBACK-VAL,  $\langle v, ts \rangle$ ) to all
  wait for (n - f) acks.
  // Round 2
  send (WBACK-TS, ts) to all
  wait for (n - f) acks.
}

```

Fig. 2. Reader's protocol

Initialization:
 $\text{READERS} := \emptyset$
 $\text{RNextVal} := \perp$

```

server() {
  // Write Protocol messages
  on receive (WRITE-VAL,  $\langle v, ts \rangle$ ) from writer
    if (RVal.ts < ts)
      (RPrev2, RPrev, RVal) := (RPrev, RVal,  $\langle v, ts \rangle$ )
      send WRITE-ACK1 to the writer
  on receive (WRITE-TS, ts, CR) from writer
    if (Rcts < ts)
      Rcts := ts
    for each  $r \in \text{CR}$ :
      send (FWD, s, RVal, { RVal, RPrev, RPrev2 })
        to r
      READERS = READERS  $\setminus$  CR
      send WRITE-ACK2 to the writer
  // Read Protocol messages
  on receive (GET-TS) from reader r:
    READERS.enqueue(r)
    send (TS, s, Rcts) to r
  on receive (GET-VAL) from reader r
    send (VALS, s, { RVal, RPrev }) to r
  // Write back Messages
  on receive (WBACK-VAL,  $\langle v, ts \rangle$ ) from reader r
    wait for ( Rcts  $\geq$  ts - 1 )
    if (RVal.ts < ts)
      (RPrev2, RPrev, RVal) := (RPrev, RVal,  $\langle v, ts \rangle$ )
      send WBACK-ACK1 to r
  on receive (WBACK-TS, ts) from reader r
    wait for ( RVal.ts  $\geq$  ts )
    if (Rcts < ts)
      Rcts := ts
      READERS.remove(r)
      send WBACK-ACK2 to r
  // GetConcurrentReaders Protocol messages
  on receive (GET-ACT-RD-CNT) from writer
    send (RDRS_CNT, s, READERS.size())
      to writer
  on receive (GET-ACT-RDS, count) from writer
    send (READERS, s, READERS[1:count])
      to writer
  on receive (GET-ACT-RDS-INS, A) from writer
    send (RDRS-INS, s, READERS  $\cap$  A) to writer
}

```

Fig. 3. Protocol for server s

Boundedness A solution is amortized bounded if m operations do not generate more than $m \times k$ messages, for some constant k without some servers being detected as faulty. In an amortized bounded solution, a client executing a particular operation might have to handle an unbounded number of *late* messages. In a bounded solution a client operation will always handle no more than k messages for some constant k and if more than k messages are received, the faulty behavior of some servers will be detected.

Our solution is amortized bounded. This does not rule out the possibility that a reader receives many unsolicited messages from a server. All we can do in that case is to declare the server faulty and our proof of boundedness does not apply to such rogue servers that are detected to be faulty.

<p>Definitions:</p> <pre> notLarge(s) \triangleq $\{x : \text{count}[x] \geq \text{count}[s]\} \geq f + 1$ GetConcurrentReaders() { $\forall s$: readers[s] := \perp $\forall s$: count[s] := \perp $\forall s$: sent[s] := false union_set := \perp // Get Active reader count send (GET_ACT_RD_CNT) to all servers // Get Active reader lists from servers with valid count repeat on receive (RDRS_CNT, s, R) from server s count[s] = R $\forall p$: if (notLarge(p) \wedge sent[p] = false) send (GET_ACT_RDS) to server p sent[p] := true on receive (READERS, s, R) from server s if (\neg sent[s] \vee (sent[s] \wedge count[s] \neq R)) detect failure of s else readers[s] := R until ($\{\text{readers}[s] : \text{readers}[s] \neq \perp\} \geq f + 1$) union_set := \cup_s readers[s] </pre>	<pre> // Get union_set \cap Active reader lists from the rest for each (s : sent[s] \neq true) send (GET_ACT_RDS_INS, union_set) to server s repeat on receive (READERS, s, R) from server s if (\neg sent[s] \vee (sent[s] \wedge count[s] \neq R)) detect failure of s else readers[s] := R on receive (RDRS_INS, s, R) from server s if (R $\not\subseteq$ union_set) detect failure of s else readers[s] := R until ($\{s : \text{readers}[s] \neq \perp\} \geq n - f$) CR = $\{x : \{s : x \in \text{readers}[s]\} \geq (f + 1)\}$ return CR } </pre>
---	---

Fig. 4. Bounded detection of readers: Writer code

To make the solution bounded for the reader techniques such as [3] can be used.

3.3 Bounded Detection of Readers

The protocol requires that the writer be able to detect ongoing read operations. A writer that invokes `GetConcurrentReaders()` after all correct servers have started processing a read request r issued by client c_r must be able to identify r (assuming r does not terminate before the end of the execution of the detection protocol).

A simple way to implement the required functionality is for the writer to collect, from all servers, the sets of ongoing read operations (the *active reader operations*) and to identify those among them that appear in at least $f + 1$ sets: this is the approach taken in [3]. Because it is possible that some servers may have begun processing read requests that have not yet reached the other servers, faulty servers can send arbitrarily long lists of bogus active operations without being detected as faulty. Our protocol rectifies this problem, and is shown in Figure 4.

Protocol Description The idea of the protocol is to first estimate the number of active read operations in the system and then accept lists of active reader operations whose size is bounded by this estimate. The difficulty is in ensuring that all genuinely active operations, and only those, are detected. The protocol has two phases. In the first phase, the writer determines a set of servers who are returning a *valid* active list

count, i.e. a count of active reader operations that does not exceed the count returned by at least *some* correct server. In the second phase, the writer requests these servers for their active lists, which are known not to be too large.

For servers, whose count is not known to be *valid*, the writer cannot request the active list since it could be too large. However, once the writer has collected $f + 1$ active lists from servers with a *valid* count, the writer sends the union of these lists to the remaining servers and only requests for the elements in the server's active list that is present in the union.

On receiving the active sets (or their intersection) from at least $n - f$ servers, the writer computes the set of concurrent readers.

Protocol guarantees Since the writer only needs to wait for $(n - f)$ responses, this sub-protocol always terminates. The number of messages exchanged in this sub-protocol is bounded as the writer does not send or receive more than two messages to any server. The messages sent in the first two phases are bounded in size because the writer only requests active lists from servers with a *valid* count. The messages in the third step is also bounded in size, because the size of the computed union set is bounded.

4 Tolerating Byzantine Readers

In a wait-free atomic implementation of replicated storage, readers must write to servers to ensure read-read atomicity [5]. With Byzantine readers, servers need guarantees that the values written by readers are valid. This can be satisfied by having the reader present a proof that a correct server vouches for the value and such a proof can be satisfied by having the reader present evidence that $f + 1$ servers vouch for the value it wants to write back. Traditionally, such vouchers or proofs rely on public key cryptography, which depend on unproven assumptions such as the hardness of factoring, or the hardness of computing discrete logarithms [9].

In general, it is not sufficient for a reader to prove that the value it is writing originated from the writer. For instance, if the reader is expected to write more than one value in some order, then the reader should not write a later value without having completed writing the previous value in the order. Omitting to write some values can in general lead to violations of the protocol's requirements.

With respect to the protocol presented in Section 3, the servers should verify two things:

1. A reader is allowed to write back a value only if it proves that it received the value from a correct server (i.e. by having $f + 1$ servers vouch for the value).
2. A reader is allowed to perform a second round write back of the timestamp only after $f + 1$ correct servers have accepted the first round write back message (i.e. $2f + 1$ servers accepted the first round write back).

With public key cryptography, these proofs can be easily implemented. A server signs messages it sends to a reader and a reader can provide as proof the requisite number of signed messages.

An important observation is that these (signed) messages indicate the progress in terms of the server state and are not specific to the particular read operation. The protocol remains correct even if these proofs are put together from signed messages sent by servers in response to different read operations.

We present a secret-sharing-based approach that can be used to implement these types of proofs. This shows that the (strong) assumption of computationally one-way functions, used by PKIs, is not required for these applications. We believe that our approach can be used not just with the protocol presented in Section 3, but also with other protocols that have a similar structure—however, characterizing such protocols and developing a general framework to replace cryptographic-based techniques with our techniques are left for future work.

4.1 Provably correct proofs using secret sharing

Consider the read and write protocols in Figure 5, which are typically part of larger protocols.

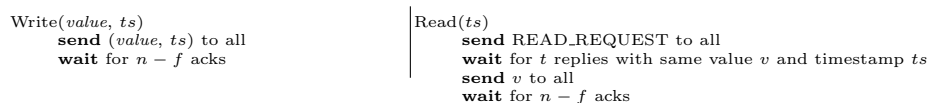


Fig. 5. Simple client protocols

In the protocol, the reader is attempting to read a value whose timestamp is ts and the writer writes a value whose timestamp is ts (not necessarily the same). The server code corresponding to these two protocols is the obvious one.

As presented, the reader protocol is not guaranteed to terminate and typically, it will be part of a larger protocol that ensures termination.

We will not concern ourselves with termination in the remainder of this section.

We would like to transform the two protocols so that correct readers are not affected and faulty readers cannot write back a value that was not written by the writer.

We achieve the transformation by splitting the write operation into two parts. In a first part, the writer sends the value to be written along with some *other information* that we will explain shortly. In the second part of a write operation, the writer sends a message indicating that the first part has finished. The servers process the message with the values, exactly as in the simple protocol, but only when it receives the FINISHED_SENDING_VALUES message. In other words, the values received in the first part are hidden and are not processed (or sent to readers) unless the server knows that the writer finishes the first part. This knowledge can be obtained directly from the writer or indirectly from the reader.

So, we need a way for a reader that received t identical values to convince a correct server that only received a first message from the writer to open the value that the server received from the writer. By doing this, the reader is effectively writing back a value but without having to sign the value. The write back consists of making a hidden value non-hidden. A reader knows that it can write back to all correct servers if it knows that the writer finished the first part and started the second part at some correct server because, in that case, all correct servers will eventually receive the value from the writer which is sent in the first part.

So, the question is how can we provide a proof that the writer made enough progress. This is where the *other information* enter the picture. The main idea is to have the writer associate a randomly generated secret with the value it wants to write. The writer generates this secret, creates shares, and sends these shares to servers before (roughly speaking) starting the actual write operation.

A server which *knows* that the writer has completed the first part and started the second part, is willing to provide the value as well as shares of the secret. The secret is shared such that if (and only if) enough messages are received, the reader will be able to reconstruct the secret. Thus, reconstructing the secret can be used as a proof that the writer made enough progress in its write operation. The details of the secret sharing scheme are given below.

The modified writer and reader protocols are shown in Figure 6.

Each secret is split using the techniques of [11] such that $k.t$ shares are required to reconstruct the secret, where $k > f$. Each server is given $k + 1$

<pre> Write(value, ts) generate secret s $\forall 1 \leq i \leq n$ generate shares $s_i[1 \dots k + 1]$ $\forall 1 \leq i \leq n$ send (value, ts, $s_i[1 \dots k + 1]$) to $server_i$ send FINISHED_SENDING_VALUES to all wait for $n - f$ replies </pre>	<pre> Read(ts) send READ_REQUEST to all value_read = \perp repeat receive message (v, ts, $s_i[1 \dots k]$) from server i if received $\geq t$ messages with the same v and ts fork { send (v, collectedSharesFor(v)) to all wait for $n - f$ acks value_read = v } until (value_read != \perp) </pre>
--	---

Fig. 6. Modified client protocols

shares, along with the message that was going to be sent in the simple protocol. After sending these messages to all the servers, the writer sends the FINISHED_SENDING_VALUES message.

If a read request reaches a server after the server received the FINISHED_SENDING_VALUES message from the writer, the server sends the value to the reader along with k of the shares the server received from the writer. One share is never revealed and is used by servers only for verifying that a reconstructed secret is correct. Secret sharing ensures that the secret can be reconstructed if and only if t number of servers give away their shares.

The server now accepts a write back from a reader only if the reader can provide enough shares that can reconstruct the correct secret at the server. The server can accept a write back even before receiving the FINISHED_SENDING_VALUES message. If the server decides to accept the write back message, the server unhides the message and acts as if it has received a FINISHED_SENDING_VALUES message directly from the writer.

Note that unlike the case with cryptographic solution in which the reader could determine locally whether the received signatures are valid, in our protocol the reader needs the cooperation of the servers in order to determine if the received shares can reconstruct the correct secret. Here, the reader cannot determine if the proof is going to be valid because it does not have any shares to verify against. This may cause the write backs to not succeed if there are not enough correct shares. Also, the shares that enable the reader to write back might not arrive all at once. The reader might first receive $n - f$ replies that do not include the crucial shares of some slow correct servers. So, the reader's protocol would have the reader request verification from servers every time the reader receives a new share. The reader can finish when it has received enough identical values and it knows that it can write back.

5 Protocol tolerating Byzantine readers

To convert the protocol presented in Section 3 to a Byzantine reader tolerant protocol, the writer has to perform an extra phase. This extra phase contains the secrets required for both the phases of the write protocol. So phase 1 and Phase 2 of the original protocol are replaced with (Phase 1' || Phase 2'), Phase 1'', and Phase 2'' where the primes and double primes are used to indicate the transformed phases and the || indicates that the first transformed phases are executed concurrently.

5.1 The write protocol

Before beginning the two phases of the write, the writer generates two random secrets. The writer sends the shares for these secrets, along with the value and timestamp information that it is going to write, to all the servers before initiating the write phases.

The first secret is used to prove that the reader has received $f + 1$ identical values and is split such that $t = f + 1$. The second secret is used to prove that the reader has received $(n - f)$ acknowledgements to the first phase write back and is split such that $t = (n - f)$.

On receiving these shares and information regarding the value and timestamp that is going to be written, the servers hold them separately and do not update any values or timestamps that are used in the original protocol.

After sending these shares and values to all the servers, the writer begins the original write protocol, asking the servers to update the value and then the timestamp. Only on receiving the message from writer to update the value, or on accepting a write back message, will a server update its value and reveal its secret shares. The same holds true for updating the timestamp.

5.2 The read protocol

The read protocol is similar to the original read protocol in Section 3. The only difference being that the reader needs to include the collected shares as a proof to be allowed to perform the write back.

As in Section 3, the reader waits to collect $f + 1$ matching responses for an acceptable timestamp. It then tries to write back the value providing as a proof the set of shares collected so far. Retrying each time it receives more shares, or when it can try to write back a different value.

The server will only accept the write back if value being written back matches the value that was initially received from the writer, and the shares can reconstruct the correct secret. If at least one non-faulty server has revealed the value, then all servers will eventually receive the value and the shares sent by the writer to be able to verify the information provided by the reader. Thus valid write backs from correct readers will be eventually accepted.

Moreover, since the servers receive the value from the writer directly, the reader need not send the value along with the write back. It is sufficient to use a lightweight tag, or the timestamp to identify the write [5]. Thus, preventing the servers from having to process large messages from bad readers. For simplicity, we assume that the protocol does not have this optimization.

On accepting the first write back for the value, the server responds with its shares of the second secret. On receiving $(n-f)$ of these responses, the reader proceeds to write back the timestamp in the second phase sending the shares it received in these responses as proof that $(n-f)$ servers have accepted the first phase write back. The reader retries writing back whenever it receives additional shares. When all the correct servers accept the write back value and respond with their shares of the second secret, the reader will have enough correct shares to reconstruct the secret correctly and complete the write back.

Late write backs One complication is that if a first round write back arrives late at a server, the server might not have the shares to give the reader because the old shares might have been replaced with newer ones due to subsequent writes. If a server that receives a write back message has a current timestamp that is larger than the timestamp being written back, it simply sends a write back acknowledgment, but without the shares (sending \perp for the shares).

The meaning of a write back without shares is that the writer has started the second phase of the write of a value with a higher timestamp. When the reader finishes its first round of write back it will collect $(n-f)$ acknowledgments, some with shares and some without shares, and send these along with its second phase write back. If one of the acknowledgments without shares is from a correct server, then this means that the writer must have started writing a new value and finished the second phase of the write operation for which the reader is sending a second phase write back, and therefore all correct servers will eventually receive the second phase message from the writer and can accept the write back.

If none of the acknowledgments without secrets is from a correct server, then the reader will eventually receive either enough secrets (as we argued in the previous paragraph) or one acknowledgment without secrets from a correct server; in either case, the correct reader will be able to finish its second phase write back.

Bounding number of retries The reader only tries to write back values that have been received from at least $f + 1$ different servers. Since the reader queries the servers for values up to $f + 1$ times and gets up to 2 values from each server in addition to 3 values in the forwarded message, the reader can get up to $2(f + 1) + 3$ different values from each server. Thus a correct reader may only retry writing back a maximum of $\frac{n(2f+5)}{f+1}$ different values. Also, since each value will only be retried $f + 1$ times, the number of messages exchanged due to the retries is still bounded.

Acknowledgements We thank Allen Clement and Harry Li for their helpful discussions on secret verification.

References

1. A. Aiyer, L. Alvisi, and R. A. Bazzi. Bounded Wait-free Implementation of Optimally Resilient Byzantine Storage without (Unproven) Cryptographic Assumptions. Technical Report TR-07-32, University of Texas at Austin, Department of Computer Sciences, July 2007.
2. R. A. Bazzi and Y. Ding. Non-skipping timestamps for byzantine data storage systems. In *DISC '04*, pages 405–419. Springer-Verlag, 2004.
3. R. A. Bazzi and Y. Ding. Bounded wait-free f -resilient atomic byzantine data storage systems for an unbounded number of clients. In *DISC '06*, pages 299–313. Springer-Verlag, 2006.
4. C. Cachin and S. Tessaro. Optimal resilience for erasure-coded byzantine distributed storage. In *DSN*, pages 115–124, Washington, DC, USA, 2006. IEEE Computer Society.
5. R. Fan and N. Lynch. Efficient replication of large data objects. In *DISC '03*, volume 2848 of *LNCS*, pages 75–91, Oct. 2003.
6. R. Guerraoui and M. Vukolic. Refined Quorum Systems. Technical Report LPD-REPORT-2007-001, EPFL, 2007.
7. M. Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124–149, January 1991.
8. L. Lamport. On interprocess communication. part i: Basic formalism. *Distributed Computing*, 1(2):77–101, 1986.
9. B. Liskov and R. Rodrigues. Byzantine clients rendered harmless. In *DISC 2005*, pages 311–325. Springer-Verlag, 2005.
10. J.-P. Martin, L. Alvisi, and M. Dahlin. Minimal byzantine storage. In *DISC '02*, pages 311–325. Springer-Verlag, 2002.
11. M. Tompa and H. Woll. How to share a secret with cheaters. *J. Cryptol.*, 1(2):133–138, 1988.