

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

Amitanand S. Aiyer †, Lorenzo Alvisi †, Rida A. Bazzi ‡

† Department of Computer Sciences

The University of Texas at Austin

‡ Department of Computer Sciences

Arizona State University

{anand, lorenzo}@cs.utexas.edu, bazzi@asu.edu

Abstract

We present the first optimally resilient, bounded, wait-free implementation of a replicated register providing atomic semantics in a system in which readers can be Byzantine, up to f servers ($n \geq (3f + 1)$) are subject to Byzantine failures and servers do not communicate with each other. 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 the subject of much study [1, 2, 3, 4, 5, 7, 10, 11, 12]. 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 [9] read and write semantics are possible, but stronger semantics are not [8]. 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 subject to Byzantine failures. Systems in which servers do not communicate with each other are interesting because such solutions can be relatively easily translated into solutions in a shared object model. Also, solutions that depend on communication between servers tend to have high message complexity, quadratic in the number of servers [12, 5].

In the non-communicating servers model, the best previous solution that provides wait-free atomic semantics requires $4f + 1$ servers [4]. 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 [7] (we discuss that solution further in the related work).

None of these solutions consider Byzantine readers. Considering Byzantine behavior of readers is relevant because wait-free atomic solutions require that readers write to servers [6]. All existing work that

considers Byzantine readers uses cryptographic techniques based on unproved number-theoretic assumptions [10, 5].

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 [4].

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 [4]. 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*. We show that this new way of selecting the timestamp is necessary to guarantee safety, i.e. atomic semantics. Indeed, 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 [3], 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 such an assumption can be enforced in practice using cryptographic techniques that rely on unproven assumptions, such techniques are not needed to enforce our requirement. We assume FIFO point-to-point asynchronous channels between clients and servers. Servers do not communicate with other servers.

We assume that writers can only fail by crashing. In Section 3 we also assume that the readers can fail only by crashing, but we relax this assumption in Sec-

tion 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. 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 now 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 extend this protocol to handle Byzantine readers. These protocols can also be easily extended to support multiple-writers, using ideas from [4], as described in Section 6.

3.1 Protocol Overview

3.1.1 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.

3.1.2 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 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 can be certain as to $t_{largest}$ really is—in reality, in our protocol the reader can only estimate $t_{largest}$ by using the $2f + 1$ ’st smallest second phase timestamp. We will show that this is sufficient to guarantee that the reader can decide and that its decision is valid.

A more detailed description of the pseudocode can be found in Appendix ??

```

write() {
101:   inc(ts)

      // Phases W1
102:   cobegin {
           writeVal();
           CR = GetConcurrentReaders()
        } coend

      // Phase W2
103:   send (WRITE_TS, ts, CR) to all
104:   wait for (n - f) acks.
}

writeVal() {
105:   send (WRITE_VAL, ⟨v, ts⟩) to all
106:   wait for (n - f) acks.
}

```

Figure 1: The Writer’s Protocol

3.2 Protocol Correctness

We show that the protocol implements atomic semantics, that the value read by readers are valid, and that the operations always terminate even if some client crash in the middle of their operations. We also show that the protocol is bounded assuming that the GetConcurrentReaders sub-protocol terminates and is bounded, which is shown in Section 3.3.

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() {
  ∀s: last_comp[s] = ⊥
  ∀s: fwd[s] = ⊥
  ∀s: Values[s] = ∅

  // Phase R1
  send (GET_TS) to all

  repeat
201:   on receive (TS, s, ts) from server s
202:     last_comp[s] = ts

203:   on receive (FWD, s, ⟨v, ts⟩, Vals) from server s
204:     fwd[s] = ⟨v, ts⟩
205:     Values[s] = Values[s] ∪ Vals

206:   until (|{x : last_comp[x] ≠ ⊥}| ≥ n - f)

  // Phase R2
207:   send (GET_VAL) to all

  repeat
208:   on receive (TS, s, ts) from server s
209:     last_comp[s] = ts
210:     send (GET_VAL) to all

211:   on receive (VALS, s, Vals) from server s
212:     Values[s] = Values[s] ∪ Vals

213:   on receive (FWD, s, ⟨v, ts⟩, Vals) from server s
214:     fwd[s] = ⟨v, ts⟩
215:     Values[s] = Values[s] ∪ Vals

  until (∃⟨v_c, ts_c⟩ : fwded(⟨v_c, ts_c⟩)
216:         ∨ (notOld(⟨v_c, ts_c⟩) ∧ valid(⟨v_c, ts_c⟩)))

  // Phase R3
217:  WriteBack(ts_c)
218:  return ⟨v_c, ts_c⟩
}

WriteBack( ts ) {
  // Round 1
219:  send (WBACK_VAL, ⟨v, ts⟩) to all
220:  wait for (n - f) acks.

  // Round 2
221:  send (WBACK_TS, ts) to all
222:  wait for (n - f) acks. }

```

Figure 2: Reader's protocol

Atomicity We order all operations according to their timestamps. The timestamp of a write operation is the timestamp of the value being written and the timestamp of a read operation is the timestamp of the returned value. When two operations have the same timestamp, we order the write before the read. Given

Initialization:

$$\text{READERS} := \emptyset$$

$$\text{RNextVal} := \perp$$

```

server() {
  // Write Protocol messages

301:   on receive (WRITE_VAL, ⟨v, ts⟩) from writer
302:     if (RVal.ts < ts)
303:       RPrev2 := RPrev
304:       RPrev := RVal
305:       RVal := ⟨v, ts⟩

306:   send WRITE-ACK1 to the writer

307:   on receive (WRITE_TS, ts, CR) from writer
308:     if (Rcts < ts)
309:       Rcts := ts

310:   for each r ∈ CR:
311:     send (FWD, s, RVal, { RVal, RPrev, RPrev2 })
      to r
312:   READERS = READERS \ CR
313:   send WRITE-ACK2 to the writer

  // Read Protocol messages
314:   on receive (GET_TS) from reader r:
315:     READERS.enqueue(r)
316:     send (TS, s, Rcts) to r

317:   on receive (GET_VAL) from reader r
318:     send (VALS, s, { RVal, RPrev }) to r

319:   on receive (WBACK_VAL, ⟨v, ts⟩) from reader r
320:     wait for ( Rcts ≥ ts - 1 )
321:     if (RVal.ts < ts)
322:       RPrev2 := RPrev
323:       RPrev := RVal
324:       RVal := ⟨v, ts⟩

325:   send WBACK-ACK1 to r

326:   on receive (WBACK_TS, ts) from reader r
327:     wait for ( RVal.ts ≥ ts )
328:     if (Rcts < ts)
329:       Rcts := ts

330:   READERS.remove(r)
331:   send WBACK-ACK2 to r

  // GetConcurrentReaders Protocol messages
332:   on receive (GET_ACT_RD_CNT) from writer
333:     send (RDRS_CNT, s, READERS.size()) to writer

334:   on receive (GET_ACT_RDS, count) from writer
335:     send (READERS, s, READERS[1:count]) to writer

336:   on receive (GET_ACT_RDS_INS, A) from writer
337:     send (RDRS_INS, s, READERS ∩ A) to writer }

```

Figure 3: Protocol for server s

this ordering, to prove atomicity, it is sufficient to prove the following lemmas .

Lemma 1. *If a writer completes a write for timestamp t , no further reader will satisfy $\text{fwded}[\langle v, x \rangle]$ for any*

$x \leq t$.

Proof: Since the writer has already completed the write operation for timestamp t , no correct server will forward a message saying that the latest timestamp is x for any $x \leq t$. Thus $fwded[\langle v, x \rangle]$ cannot become true for any $x \leq t$. \square

Lemma 2 (W-R atomicity). *Once a writer completes a write operation for a value with timestamp t , no read operation that starts after the write operation terminates will return a value with a timestamp smaller than t*

Proof: When a writer completes write for timestamp t , all but f processes would have set their value of timestamp $Rcst$ to t . Thus a later read cannot receive more than $2f$ timestamps (with the TS message) that are $< t$. Thus $notOld[x]$ will always be false for any value $< t$

Also, from lemma 9, $fwded[\langle v, x \rangle]$ cannot become true for any $x \leq t$. \square

Lemma 3 (R-R atomicity). *Once a read operation returns a value with timestamp t , no future read will return a value with a smaller timestamp.*

Proof: When a reader completes the write back for timestamp t , all but f processes would have set their value of timestamp $Rcst$ to t . Thus a later read cannot receive more than $2f$ timestamps (with the TS message) that are $< t$. Thus, for any later reader, $notOld[x]$ will always be false for any value $< t$

Also, since the reader has returned t , the writer has already completed the write operation for timestamp $t - 1$. Thus, from lemma 9, $fwded[\langle v, x \rangle]$ cannot become true for any $x \leq t - 1$. \square

3.2.1 Validity and Wait-freedom

The following lemmas have simple proofs that can be found in the Appendix.

Lemma 4 (Correctness). *A read only returns a value that is written by a writer.*

Proof: A value is set as the chosen_value has to satisfy either $fwded$ or $valid$. Since at least $f + 1$ servers are required to return the same value for either of these conditions, at least one of them must be correct. Correct clients only accept values received from the writer. \square

Lemma 5 (Write Liveness). *A write operation always*

terminates and is wait-free.

Proof: Theorem 1 shows that the GetConcurrentReaders sub-protocol always terminates.

In the remainder of a write operation, the writer only waits for $n - f$ responses at any stage. Since at most f servers are faulty, it follows that a write operation always terminates and is wait-free. \square

Lemma 6 (Read Liveness). *A read operation always terminates and is wait-free.*

Proof: Proof by Contradiction. Assume that a read operation never terminates.

Eventually the reader should receive acknowledgements for the GET_TS message from all the correct processes. Let the global time at that instance be gts_0 .

Let t be the timestamp of the last write to have completed before gts_0 , and ts_{max} be the timestamp of the $(2f + 1)$ -th smallest timestamp calculated by the reader (at gts_0).

The writer could be writing timestamp $t + 1$ but, it would not have started writing timestamp $t + 2$. Thus,

$$ts_{max} \leq t_{largest} \leq t + 1$$

where $t_{largest}$ is the largest (third phase, W3) timestamp value received from the correct servers.

If the writer has already detected the reader during a previous write, then eventually the reader will eventually receive $f + 1$ forwarded messages from the correct servers and will be able to decide on the forwarded value-timestamp pair.

If not, by theorem 2, if the writer writes timestamp $t + 2$, the writer will detect the read operation during the GetConcurrentReaders sub-protocol for write $t + 2$. On receiving the third phase message (W3) from the writer, all correct servers (in W3) will forward values for timestamp t , $t + 1$ and $t + 2$ to the reader.

For correct servers that never receive the final message for write $t + 2$, consider the following two cases:

Case1: $ts_{max} \leq t$. At most f correct servers may not have received the value for timestamp t . When the reader receives responses from all the correct servers for the GET_VAL message sent after gts_0 , the reader will have at least $f + 1$ matching value timestamp pairs $\langle v, ts \rangle$ for $ts = t$. Thus, the reader would decide on the value with timestamp t after writing it back to the servers.

Case 2: $ts_{max} == t + 1$. Since at least one correct server has updated the value of `last_comp` to be $t + 1$ it must be the case that the client (the writer, or the reader doing a write-back) must have received $n - f$ responses from servers that have updated `RVal` with timestamp $t + 1$.

Thus, when the correct servers among these $n - f$ servers respond to the `GET_VAL` message, the reader will have $f + 1$ matching responses for timestamp $t + 1$ to decide on. □

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.

In this Section we show that our solution is amortized bounded. The solution 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.

To make the solution bounded for the reader techniques such as [4] can be used. We omit these details from the presentation here.

Lemma 7 (Boundedness). *The total number of messages exchanged between the server and the reader for each read operation is bounded*

Proof: For each read operation, the reader sends to each server a maximum of 1 `GET_TS` message, $f + 1$ `GET_VAL` messages, 1 `WBACK_VAL` message, and 1 `WBACK_TS` message. Also, a reader will only receive from each server 1 `TS` message, up to $(f + 1)$ `VALS` messages sent in response to the `GET_VAL` message, up to 1 `VALS` messages forwarded in response to a concurrent write, and 2 acknowledgements in response to the `WBACK_VAL` and `WBACK_TS` messages. □

Lemma 8 (Write Boundedness). *The total number of messages exchanged between the server and the writer during a write operation is bounded.*

Proofs for the above lemmas are in the Appendix.

Proof: During each write operation, the writer sends three `WRITE` messages and receives three acknowledgements from each server. The sizes of the replies from servers are bounded. In addition, by Theorem 4, the messages exchanged in the `GetConcurrentReaders` sub-protocol are bounded. □

3.3 Bounded Detection of Readers

The protocol requires that the writer be able to detect the presence of ongoing read operations. A writer that invokes `GetConcurrentReaders()` after all correct servers have begun 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 [4]. 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.

We would like a solution that maintains the desired functionality but somehow bounds the size of the responses that a writer can receive, so that servers that send longer messages would be immediately unmasked as faulty. Clearly, no bounded solution is possible if the number of readers is infinite, because it would not be possible to bound the size of a reader identifier. We therefore assume that the set of readers is finite. Under these assumptions, a simple way to bound the implementation outlined above would be to prohibit servers from sending lists of active reader operations that are larger than the maximum number r_{max} of potential operations in the system. However, this solution is profoundly unsatisfactory because the number of active reader operations can be very small when compared to r_{max} . For example, if an operation can be represented with 100 bits (50 for reader identifiers and 50 to differentiate operations), then server responses could still contain up to 2^{100} operations even when only a handful of readers are active. Ideally, the response size should be proportional to the number of active reader operations: only solutions that match this ideal can be called bounded in any practical sense. Luckily, this match is exactly what the solution that we are about to present guarantees. In our solution, the size of messages sent by any server cannot be larger

than the size of r_{max} plus the size of the list of identifiers of the active reader operations (readers with ongoing operations). Note that the size of r_{max} is logarithmic in the number of different operations (this includes the reader identifier and the operation tag) and therefore is of the same order of magnitude as the size of an operation identifier (assuming identifier are of fixed size). The GetConcurrentReaders protocol is shown in Figure 4.

```

Definitions:
notLarge( $s$ )  $\triangleq$   $|\{x : count[x] \geq count[s]\}| \geq f + 1$ 

GetConcurrentReaders() {
   $\forall s$ : readers[ $s$ ] :=  $\perp$ 
   $\forall s$ : count[ $s$ ] :=  $\perp$ 
   $\forall s$ : sent[ $s$ ] := false
  union_set :=  $\perp$ 

  send (GET_ACT_RD_CNT) to all servers

  repeat
401:   on receive (RDRS_CNT,  $s$ ,  $count$ ) from server  $s$ 
402:     count[ $s$ ] =  $count$ 
403:      $\forall p$ : if (notLarge( $p$ )  $\wedge$  sent[ $p$ ] = false)
404:       send (GET_ACT_RDS) to server  $p$ 
405:       sent[ $p$ ] := true

406:   on receive (READERS,  $s$ ,  $R$ ) from server  $s$ 
407:     if ( $\neg$  sent[ $s$ ]  $\vee$  (sent[ $s$ ]  $\wedge$  count[ $s$ ]  $\neq$  | $R$ |))
408:       detect failure of  $s$ 
409:     else
410:       readers[ $s$ ] :=  $R$ 
411:   until ( $|\{s : readers[s] \neq \perp\}| \geq f + 1$ )

412:   union_set :=  $\cup_s$  readers[ $s$ ]
413:   for each ( $s : sent[s] \neq true$ )
414:     send (GET_ACT_RDS_INS, union_set) to server  $s$ 

  repeat
415:   on receive (READERS,  $s$ ,  $R$ ) from server  $s$ 
416:     if ( $\neg$  sent[ $s$ ]  $\vee$  (sent[ $s$ ]  $\wedge$  count[ $s$ ]  $\neq$  | $R$ |))
417:       detect failure of  $s$ 
418:     else
419:       readers[ $s$ ] :=  $R$ 

420:   on receive (RDRS_INS,  $s$ ,  $R$ ) from server  $s$ 
421:     if ( $R \not\subset$  union_set)
422:       detect failure of  $s$ 
423:     else
424:       readers[ $s$ ] :=  $R$ 
425:   until ( $|\{s : readers[s] \neq \perp\}| \geq n - f$ )

426:   CR =  $\{x : |\{s : x \in readers[s]\}| \geq (f + 1)\}$ 
427:   return CR
}

```

Figure 4: Bounded detection of readers: Writer code

3.3.1 Protocol Description

The idea of the protocol is to first estimate the number of active reader 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 collects active reader operations sets whose overall size is bounded by the sum of the valid active list counts determined in the first phase. The protocol guarantees that the sets collected in the second phase detect all ongoing read operations. We describe the two phases in more details in what follows.

(1) The first phase involves two communication steps. In the first communication step, the writer prompts the servers for their active list count and determines which servers return a *valid* active list count $count$, where a count $count$ is valid if there are at least $f + 1$ servers that return a count equal to or greater than $count$. This means that there is at least one correct server whose count is equal to or greater than $count$. In the second communication step, the writer requests the actual list of active reader operations from every server that returned a valid count. Here there is a slight technicality, as the number of active read operations that are active at the servers with valid counts might have grown since the writer prompted them for their active list counts. We handle this by requiring the servers to save the list of active readers when they receive a count request: when the servers receive a request for the actual list of active readers, they return the one they have previously saved. If a server replies with a list that is longer than the count it has send previously, the server is declared faulty. The first phase ends when the writer collects list of active readers, with valid counts, from $f + 1$ servers. At this point, the writer can be certain that at least one these $f + 1$ servers is correct and its list contains all issuers of ongoing read operations whose read requests reached all correct servers before the execution of the detection protocol. However, there is no way for the writer to determine which specific server(s), among the $f + 1$ servers with valid counts, are correct. So, the writer constructs the union of all the active list sets received by the end of phase 1. As we just indicated, the union set must contain all active reader operations whose request messages have reached all correct servers before the start of the detection protocol.

(2) In the second phase, the writer sends the union set

to all servers from which it has *not* requested an active list. On receipt of the union set, a server s is required to send the intersection of the union set with its own active list. The writer collects replies from the servers from which it did not receive a list of active readers in the first phase. There are two types of such servers. First, the servers that received a request for their list of active reader operations in phase 1, but did not return their response in time, before the end of phase 1. Second, the servers that were not sent a request in phase 1 and are replying to intersection requests made in phase 2. The second phase ends when the total number of servers that send lists either in the first or the second phase is greater than or equal to $n - f$. After the end of the second phase, the writer includes every reader that appears in $f + 1$ active reader sets in the set CR of active readers.

3.3.2 Proof of Correctness

Theorem 1 (Termination). *The `GetConcurrentReaders()` protocol always terminates if the writer never crashes.*

Proof. The writer only waits for up to $n - f$ responses from both the first and second phases to terminate. Since servers do not have to wait to respond, all correct servers will eventually respond to the writer and the writer will always finish the execution of the protocol. \square

Theorem 2 (Detection). *If a read operation r never terminates and the writer starts executing the detection protocol after all correct servers receive the read request, then the writer will include r in the set CR at the end of the detection protocol if it has not already detected the read operation r during a previous write.*

Proof. Since the read operation r does not terminate, and the writer has not detected r during a previous write operation, r will be present in the active reader's list for all the correct servers for the whole duration of the execution of the detection protocol. Since the writer waits for $f + 1$ responses to compute the union, and r is present in the active reader's list for every correct server, r should belong to the union_set. Thus if r is present in the active reader's set for any correct server it will be reported to the writer.

The writer collects the information about the active readers from at least $n - f \geq 2f + 1$ servers. Since at least $f + 1$ of them are correct and contain r , they will

report r to the writer either initially in phase 1 or in response to the union set in phase 2 and r will belong to CR at the end of the protocol. \square

Theorem 3 (Validity). *Every read operation r that is in the CR set at the end of `GetConcurrentReaders()` is an operation that was active during the `GetConcurrentReaders()` protocol.*

Proof. A read operation r is added to set CR only if it is reported to be present in the set of active readers by at least $f + 1$ servers. Since at least one of these servers must be correct, the read operation r must have been active when the server responded to the writer's message. \square

Theorem 4 (Boundedness). *The total number of messages exchanged between the server and the writer during the `GetConcurrentReaders` sub-protocol is bounded both in number and size.*

Proof: The writer only sends 2 messages and receives 2 messages from each server.

Let *count* be the actual number of concurrent read operations in the system. Thus the size of active readers at all correct servers will be at most *count*. The messages exchanged in the first round only contain the size of the set of active reader operations. This size requires can be encoded with r_{max} and is of the same order of magnitude as the reader's identifiers. In second round, the writer collects the active readers set from those servers whose count is at most the $f + 1$ -th largest. Thus the sets in these messages are no larger than *count*. Also, the size of the union_set and the responses containing the intersection is at most $(n - 2f) \times count$, which is bounded by our definition. \square

4 Tolerating Byzantine Readers

In a wait-free atomic implementation of replicated storage, readers must write to servers to ensure read-read atomicity [6]. To tolerate Byzantine readers, servers need guarantees that the values written by readers are valid i.e. have originated at the writer. 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 [10]. Assuming servers have public keys, servers can sign messages they send to readers and the reader’s *proof for a value* consists of signatures for that value from $f + 1$ different servers (typically the value is actually a value/timestamp pair).

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.

Our approach assumes that the writer is honest and

```
Write(value, ts)
  send (value, ts) to all
  wait for  $n - f$  acks
```

Figure 5: Simple client write protocol

```
Read(ts)
  send READ_REQUEST to all
  wait for  $t$  identical replies with value  $v$  and timestamp  $ts$ 
  send  $v$  to all
  wait for  $n - f$  acks
```

Figure 6: Simple client read protocol

that it uses private channels to communicate with the servers.

4.1 Provably correct proofs using secret sharing

Consider the read and write protocols in Figures 5 and 6, which are typically part of larger 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. For example, in the protocol of Section 3 the reader attempts to read a value with specific timestamps and the reader tries different timestamps until it is guaranteed to decide or to get enough forwarded values. We will not concern ourselves with termination in the remainder of this section. We assume that the remainder of the protocol guarantees that, even if all the Byzantine servers do not respond or respond with arbitrary messages, eventually the reader will be able to receive enough messages from the non-faulty servers to terminate.

We would like to transform the two protocols so that correct readers are not affected and faulty readers cannot writeback 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. Until it receives the FINISHED_SENDING_VALUES message a server does not process the value received

```

Write(value, ts)
  generate secret  $s$  and shares  $s_1[1 \dots k + 1], \dots, s_n[1 \dots k + 1]$ 
  send (value, ts,  $s_i[1 \dots k + 1]$ ) to  $server_i, 1 \leq i \leq n$ 
  send FINISHED_SENDING_VALUES to all
  wait for  $n - f$  replies

```

Figure 7: Modified client write protocol

from the writer. In particular, the server would not send the value to reader before it receives FINISHED_SENDING_VALUES. In other words, the values received in the first part are hidden and are not processed 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 writeback 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 detailed of the secret sharing scheme are given below.

The modified writer and reader protocols are shown in Figures 7 and 8.

Each secret is split using the techniques of [13] such that $k.t$ shares are required to reconstruct the secret, where $k > f$. Each server is given $k + 1$ 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 FIN-

```

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 value  $v$  and timestamp  $ts$ 
      fork {
        send (v, collectdSharesFor(v)) to all
        wait for  $n - f$  acks
        value_read = v
      }
  until (value_read !=  $\perp$ )

```

Figure 8: Modified client read protocol

ISHED_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 writeback 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 to 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. Only the servers can check if the reconstructed proof is correct. But this may cause the write backs to not succeed if there are not enough correct shares. So receiving enough identical values is not enough for the reader to finish its read operation. 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 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 write backs from a correct reader 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 [6]. 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.

5.2.1 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 writeback 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 writeback 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 writeback. 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 writeback. 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 writeback.

5.2.2 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.

Detailed pseudocode and correctness proofs can be found in Appendix A.

6 Multiple Writers

The protocol presented in Section 3 can easily be extended to support multiple writers using standard techniques. We assume that each writer has a unique identifier w_i , and that the set of writer identifiers is totally ordered.

To implement a m writer atomic register, each server maintains m copies of its data structures – one for each writer. To perform a read operation, the reader performs a read to get the latest value from each of the m writers and chooses the one with the highest timestamp. If there are values from different writers with the same timestamp, ties are broken based on the ordering of the writer’s id. The writer operation is mostly similar to the writer operation for the single writer presented in Section 3. The only challenge is in the way timestamps are incremented to implement non-skipping timestamps.

In order to implement non-skipping timestamps, the writer performs a (multi-writer) read operation to get the value-timestamp information for the latest completed write. The writer then chooses the next higher timestamp for its current write.

7 Related Work

Distributed storage systems have been widely studied in [1, 2, 3, 4, 5, 7, 10, 11, 12]. These works vary widely in terms of the consistency semantics provided, resilience to faults and client failures, and the assumptions about the environment.

Atomic semantics: Malkhi and Reiter first used quorum systems to build a scalable distributed storage system [11]. Their system uses self-verifying data to achieve atomic semantics with $n \geq 3f + 1$ replicas. Martin et al. were the first to implement an atomic register for generic data in an asynchronous system with unbounded number of readers and writers using the optimal $3f + 1$ replicas [12]. They achieve atomic semantics without reader write-back, so they can trivially handle Byzantine readers. However their protocol is not wait-free, may require an unbounded number of messages during a read operation, and it is vulnerable to faulty servers causing the timestamps to grow infinitely large.

Non-skipping timestamps: Bazzi and Ding [3] introduced non-skipping timestamps to counter the rapid exhaustion of the timestamp space: they require $4f + 1$ replicas. Cachin and Tesaro [5] achieve non-skipping timestamps using $3f + 1$ replicas using threshold cryptography. Their solution can tolerate both Byzantine readers and writers but requires servers to communicate among themselves and needs cryptography.

Wait-freedom: Abraham et al. show that constructing a wait-free register in a shared memory model with $< (4f + 1)$ replicas requires a two-round write operation for at least one server [1]. In [1], they show a wait-free construction of a safe register that uses only $3f + 1$ replicas. In [2] they also develop a wait-free regular register but require $n \geq 4f + 1$ replicas.

Recently, Guerraoui and Vukolic have proposed the novel abstraction of *refined quorum systems* to capture both (i) the worst case conditions with asynchrony, contention and failures (ii) and also, the best case conditions involving synchrony, no contention, and no failures [7]. Using this abstraction, they provide a distributed storage implementation that guarantees wait-free atomic semantics in a shared memory model for generic data without any authentication primitives with optimal (best-case) complexity (number of rounds). This solution is optimally resilient (it requires $n \geq 3f + 1$), but it does not address worst case boundness as we do. Under adversarial contention and

asynchrony assumptions, the solution allows a read operation to send an unbounded number of messages. Our solution guarantees boundedness in all executions, but in the absence of adversarial conditions, read and write operations in our solution require a larger number of rounds than those of their solution. Their solution does not tolerate Byzantine readers as we do.

Finiteness: Bounded Wait-free registers were first introduced in [4], but required $4f + 1$ replicas and were only partially bounded: messages between the writer and the faulty servers could be infinitely large. Also, [4] only considers benign clients.

Byzantine Readers: Handling faulty clients (readers and writers) has been considered by [5, 10, 12]. All these approaches are based on cryptographic primitives that rely on the unproven assumption about the computation hardness of problems such as factoring and discrete logarithms, and involve communication between servers.

References

- [1] I. Abraham, G. V. Chockler, I. Keidar, and D. Malkhi. Byzantine disk paxos: optimal resilience with byzantine shared memory. In *Distributed Computing*, pages 387–408. Springer-Verlag, April 2006.
- [2] I. Abraham, G. V. Chockler, I. Keidar, and D. Malkhi. Wait-free regular storage from byzantine components. *IPL*, July 2006.
- [3] R. A. Bazzi and Y. Ding. Non-skipping timestamps for byzantine data storage systems. In *DISC '04*, pages 405–419. Springer-Verlag, 2004.
- [4] 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.
- [5] 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.
- [6] R. Fan and N. Lynch. Efficient replication of large data objects. In *DISC '03*, volume 2848 of *LNCS*, pages 75–91, Oct. 2003.
- [7] R. Guerraoui and M. Vukolic. Refined Quorum Systems. Technical Report LPD-REPORT-2007-001, EPFL, 2007.
- [8] M. Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124–149, January 1991.
- [9] L. Lamport. On interprocess communication. part i: Basic formalism. *Distributed Computing*, 1(2):77–101, 1986.
- [10] B. Liskov and R. Rodrigues. Byzantine clients rendered harmless. In *DISC 2005*, pages 311–325. Springer-Verlag, 2005.
- [11] D. Malkhi and M. K. Reiter. Secure and scalable replication in phalanx. In *Proc. 17th SRDS*, pages 51–58, 1998.
- [12] J.-P. Martin, L. Alvisi, and M. Dahlin. Minimal byzantine storage. In *DISC '02*, pages 311–325. Springer-Verlag, 2002.
- [13] M. Tompa and H. Woll. How to share a secret with cheaters. *J. Cryptol.*, 1(2):133–138, 1988.

A Detailed Protocol Description for Handling Byzantine Readers

Section 5 describes how the protocol in section 3 can be modified to handle Byzantine readers.

Figures 9 - 13 present the pseudocode for handling Byzantine readers. The protocol for detection of concurrent readers is same as the one presented in Figure 4.

A.1 The write operation

To perform a write operation, the writer first installs the shares and values at all the servers. This is done by sending the SECRET message. The data received by this message is used purely for verifying the values written back, and to furnish proofs to enable the reader write back. The servers do not divulge the secrets or values, unless they receive the WRITE_VAL/WBACK_VAL messages from the writer/reader during the protocol.

A.1.1 Generating secrets

The function generateSecretsAndProofs() generates two secrets for each write operation, $SecretOne^{ts}$ and $SecretTwo^{ts}$.

The first secret, $SecretOne^{ts}$, is intended to be reconstructed only when at least $f + 1$ servers receive the WRITE_VAL message. Thus, this secret is split with $t = f + 1$. The secret is generated as a polynomial of degree $(f + 1)k$, where $k > f$. Each server i is given a total of $k + 1$ shares (random points on the polynomial). k of these shares form the secret part, $ShOne_i^{ts}[1 \dots k]$ and will be revealed by the server only when it accepts a WRITE_VAL or a WBACK_VAL message and installs the value. The $ShOne_i^{ts}[k + 1]$ is used to verify if the secret reconstructed is correct. A write back from a reader is accepted only if among the shares given by the reader, there are enough shares to reconstruct the correct secret.

We note that the server may have to try out various combinations of secret shares to try and reconstruct the actual secret. We believe that one can use error correcting codes to perform this operation efficiently. We intend to explore this in the future and currently assume that the server tries out all the possible combinations to check if any of them can regenerate the correct secret.

The second secret, $SecretTwo^{ts}$ is intended to be reconstructed only when at least $(n - f)$ servers have accepted a write back message for the first round write back. Thus, this secret is split with $t = (n - f)$. $SecretTwo^{ts}$ is a polynomial of degree $(n - f)k$, where $k > f$. Each server gets $k + 1$ shares.

The server reveals k out of $k + 1$ shares, $ShTwo_i^{ts}[1 \dots k]$, only after it has accepted the first round write back message for timestamp ts . To accept the second round write back, the server uses $ShTwo_i^{ts}[k + 1]$ to verify that the $SecretTwo^{ts}$ can be correctly reconstructed. This ensures that at least $(n - f)$ servers have accepted the first round write back before any non-faulty server accepts the second round write back.

After sending the shares along with the data about what value and timestamp is going to be written, the writer performs the write protocol as in Section 3.

```

write() {
501:   inc(ts)
502:   generateSecretsAndProofs()
      // Phase W0
503:   sendShares()

      // Phases W1
504:   cobegin {
        writeVal();
        CR = GetConcurrentReaders()
      } coend

      // Phase W3
505:   send (WRITE_TS, ts, CR) to all
506:   wait for (n - f) acks.
}

sendShares() {
  // Phase W0
507:    $\forall i$ : send (SECRET,  $ShOne_i^{ts}[1 \dots k + 1]$ ,  $ShTwo_i^{ts}[1 \dots k + 1]$ ,  $\langle v, ts \rangle$ ) to server  $i$ 
}

writeVal() {
  // Phase W1
508:   send (WRITE_VAL,  $\langle v, ts \rangle$ ) to all
509:   wait for (n - f) acks.
}

```

Figure 9: The Writer's Protocol for tolerating malicious readers

A.2 Protocol Correctness

We show that the protocol implements atomic semantics, that the value read by readers are valid, and that the operations always terminate even if some client crash in the middle of their operations. We also show that the protocol is bounded.

Definitions:

$$\text{valid}(\langle v, ts \rangle) \triangleq |\{s : \langle v, ts, * \rangle \in \text{Values}[s]\}| \geq f + 1$$

$$\text{notOld}(\langle v, ts \rangle) \triangleq |\{s : \text{last_comp}[s] \leq ts\}| \geq 2f + 1$$

$$\text{fwded}(\langle v, ts, * \rangle) \triangleq |\{s : \text{fwd}[s] = \langle v, ts \rangle\}| \geq f + 1$$

$$\text{GetShares}(\langle v_c, ts_c \rangle) \triangleq \{x | \exists s : \langle v_c, ts_c, x \rangle \in \text{Values}[s]\}$$

```

read() {
  ∀s: last_comp[s] = ⊥
  ∀s: fwd[s] = ⊥
  ∀s: Values[s] = ∅
  ∀ts: writingBack[ts] = false
  written_back_value = ⊥

  // Phase R1
601: send (GET-TS) to all
602: repeat
603:   on receive (TS, s, ts) from server s
604:     last_comp[s] = ts

605:   on receive (FWD, s, ⟨v, ts, sh⟩, Vals) from server s
606:     fwd[s] = ⟨v, ts, sh⟩
607:     Values[s] = Values[s] ∪ Vals

608:   on receive (VALS, s, Vals) from server s
609:     Values[s] = Values[s] ∪ Vals
610: until (|{x : last_comp[x] ≠ ⊥}| ≥ n - f)

  // Phase R2
611: send (GET-VAL) to all
612: repeat
613:   on receive (TS, s, ts) from server s
614:     last_comp[s] = ts
615:     send (GET-VAL) to all

616:   on receive (VALS, s, Vals) from server s
617:     Values[s] = Values[s] ∪ Vals

618:   on receive (FWD, s, ⟨v, ts, sh⟩, Vals) from server s
619:     fwd[s] = ⟨v, ts, sh⟩
620:     Values[s] = Values[s] ∪ Vals

  if (∃⟨v_c, ts_c⟩: fwded(⟨v_c, ts_c⟩)
621:     ∨ (notOld(⟨v_c, ts_c⟩) ∧ valid(⟨v_c, ts_c⟩)))
622:   writingBack[ts_c] = true
623:   fork WriteBack(⟨v_c, ts_c⟩)
624: until (written_back_value ≠ ⊥)

625: return written_back_value
}

```

Figure 10: Reader's protocol, tolerating malicious readers (part 1 of 2)

A.2.1 Secrets are hard to guess

Theorem 5. Given a f -degree polynomial P over Z_p , where p is a prime number and $\log p > k$. For a given $i \in \{1 \dots, n\}$, the probability that a random variable x over Z_p is equal to $P(i)$ is less than 2^{-k} .

Proof. The probability that x equals any specific value is $1/p < 2^{-k}$. □

Theorem 6. Given a f -degree polynomial P over Z_p , where p is a prime number and $\log p > k$. The probability that a Byzantine reader finishes a first round write back for which the writer has not finished its W1 phase is at most $2^{-(k - \log(f(2f+1)) - (f+1)n)}$.

Proof. We give a generous upper bound. For a given timestamp, then reader sends at most f sets of shares of increasing size. For each set, a server will consider at most n choose $f + 1$ possibilities. The number of these possibilities is less than $2^{(f+1)n}$. Each possibility can succeed with probability 2^{-k} . The maximum number of ways in which a reader can succeed in writing to a correct in one of the f tries is $f(2f + 1)2^{(f+1)n}$. The probability that the reader succeeds in writing to one correct server is at $2^{-(k - \log(f(2f+1)) - (f+1)n)}$. □

```

WriteBack( $\langle v_c, ts_c \rangle$ ) {
  // Round 1
626:  Shares = GetShares( $\langle v_c, ts_c \rangle$ )
627:  send (WBACK_VAL,  $ts_c$ , Shares) to all
628:  repeat
629:    on receive (WBACK_VAL_ACK, s,  $proofs\_or\_bottom_s$ , ts) from server s
      ACKS1 = ACKS1  $\cup$  { $proofs\_or\_bottom_s$ }

630:    if (Shares  $\neq$  GetShares( $\langle v_c, ts_c \rangle$ ))
631:      goto line 626

632:  until WBACK_VAL_ACK's are received from  $n - f$  different servers
  // Round 2
633:  local wb2_cnt=0;
634:  ACKS1 = set of WBACK_VAL_ACK messages received
635:  send (WBACK_TS,  $ts$ , ACKS1) to all
636:  repeat
637:    on receive (WBACK_VAL_ACK, s,  $proofs\_or\_bottom_s$ , ts) from server s
      ACKS1 = ACKS1  $\cup$  { $proofs\_or\_bottom_s$ }
638:    goto line 634

639:  on receive (WBACK_TS_ACK, s, ts) from server s
640:    wb2_cnt++
  until  $wb2\_cnt \geq n - f$ 
  written_back_value =  $\langle v_c, ts_c \rangle$ 
}

```

Figure 11: Reader's protocol, tolerating malicious readers(part 2 of 2)

It follows that the probability that a reader succeeds in writing back a bogus timestamp in the first round can be made arbitrarily small. Similarly, we can show that the probability that a reader can succeed in writing back a bogus timestamp in the second round can be made arbitrarily small.

A.2.2 Atomicity

To show atomicity, we should show that there is a global ordering of operations that is consistent with real time ordering and such that the resulting execution is a valid sequential execution. We order all operations according to their timestamps. The timestamp of a write operation is the timestamp of the value being written and the timestamp of a read operation is the timestamp of the returned value. When two operations have the same timestamp, we order the write before the read. Given this ordering, to prove atomicity, it is sufficient to prove the following.

W-R Once a writer completes a write operation for a value with timestamp t , no read operation that starts after the write operation terminates will return a value with a timestamp smaller than t .

R-R Once a read operation returns a value with timestamp t , no future read will return a value with a smaller timestamp.

Lemma 9. *If a writer completes a write for timestamp t , no further reader will satisfy $fwded[\langle v, x \rangle]$ for any $x \leq t$.*

Proof: Since the writer has already completed the write operation for timestamp t , no correct server will send a FWD message saying that the latest value-timestamp is $\langle v, x \rangle$ for any $x \leq t$. Thus $fwded[\langle v, x \rangle]$ cannot become true for any $x \leq t$ since this requires at least one response from a correct server. \square

Lemma 10 (W-R atomicity). *Once a writer completes a write operation for a value with timestamp t , no further read will return an older value.*

Proof: When a writer completes write for timestamp t , all but f processes would have set their value of timestamp $Rctsto t$. Thus a later read cannot receive more than $2f$ timestamps (with the TS message) that are $< t$. Thus $notOld[x]$ will always be false for any value $< t$

Also, from lemma 9, $fwded[\langle v, x \rangle]$ cannot become true for any $x \leq t$. \square

Lemma 11 (R-R atomicity). *Once a read operation returns a value with timestamp t , no future read will return an older value.*

Proof: When a reader completes the write back for timestamp t , all but f processes would have set their value of timestamp $Rctsto t$. Thus a later read cannot receive more than $2f$ timestamps (with the TS message) that are $< t$. Thus, for any later reader, $notOld[x]$ will always be false for any value $< t$

Also, since the reader has returned t , the writer has already completed the write operation for timestamp $t - 1$. Thus, from lemma 9, $fwded[\langle v, x \rangle]$ cannot become true for any $x \leq t - 1$. \square

Definitions:

consistent(Shares, ver_part) \triangleq Shares contain enough shares to reconstruct the correct secret. i.e. the polynomial reconstructed \mathcal{F} passes through the verification part ver_part.

Initialization:

READERS := \emptyset
 RNextVal := \perp

```

server() {
  // Write Protocol messages
701:  on receive (SECRET,  $ShOne_i^{ts}[1 \dots k + 1]$ ,  $ShTwo_i^{ts}[1 \dots k + 1]$ ,  $\langle v, ts \rangle$ ) from writer
      RNextVal :=  $\langle v, ts, ShOne_i^{ts}[1 \dots k] \rangle$ 
702:  RNextSecShOne :=  $ShOne_i^{ts}[1 \dots k]$ 
703:  RNextVerShOne :=  $ShOne_i^{ts}[k + 1]$ 
704:  RNextSecShTwo :=  $ShTwo_i^{ts}[1 \dots k]$ 
705:  RNextVerShTwo :=  $ShTwo_i^{ts}[k + 1]$ 
706:

707:  on receive (WRITE_VAL,  $\langle v, ts \rangle$ ) from writer
708:  if (RVal.ts < RNextVal.ts)
709:    RPrev2 := RPrev
710:    RPrev := RVal
711:    RVal := RNextVal
712:    RSecShOne := RNextSecShOne
713:    RVerShOne := RNextVerShOne
714:    RSecShTwo := RNextSecShTwo
715:    RVerShTwo := RNextVerShTwo

716:  send WRITE-ACK1 to the writer

717:  on receive (WRITE_TS, ts, CR) from writer
718:  if (Rcts < ts)
719:    Rcts := ts

720:  for each  $r \in CR$ :
721:    send (FWD, s, RVal, { RVal, RPrev, RPrev2 }) to r
722:  READERS = READERS  $\setminus$  CR
723:  send WRITE-ACK2 to the writer

  // Read Protocol messages
724:  on receive (GET_TS) from reader r:
725:    READERS.enqueue(r)
726:    send (TS, s, Rcts) to r

727:  on receive (GET_VAL) from reader r
728:    send (VALS, s, { RVal, RPrev }) to r

```

Figure 12: Protocol for server s , tolerating malicious readers (part 1 of 2)

A.2.3 Validity and Wait-freedom

Lemma 12 (Correctness). *A read only returns a value that is written by a writer.*

Proof: A value is set as the chosen_value has to satisfy either *fwded* or *valid*. Since at least $f + 1$ servers are required to return the same value for either of these conditions, at least one of them must be correct. Correct clients only accept values received from the writer. \square

Lemma 13 (Write Liveness). *A write operation always terminates and is wait-free.*

Proof: Theorem ?? shows that the GetConcurrentReaders sub-protocol always terminates.

In the remainder of a write operation, the writer only waits for $n - f$ responses at any stage. Since at most f servers are faulty, it follows that a write operation always terminates and is wait-free. \square

Lemma 14 (Belief 1). *If a correct server sets its value of Rctsto t in response to a second phase message (W2) from the writer, then at least $f + 1$ correct servers should have received and updated the value-timestamp for timestamp t*

Proof: Since the writer is benign, the writer only starts phase W2 for timestamp t after receiving $n - f$ responses from W1. Correct servers respond to W1 only after they have received and updated RVal with the latest value-timestamp pair. Since $n - f \geq 2f + 1$ at least $f + 1$ correct servers should have received and updated the value-timestamp for timestamp t . \square

Lemma 15 (Belief 2). *If a correct server sets its value of Rctsto t in response to a write back message (second round) from a*

```

729:   on receive (WBACK_VAL, ts, shares) from reader r
730:     wait for ( RNextVal.ts ≥ ts )
731:     if (RVal.ts > ts)
732:       send (WBACK_VAL_ACK, s, ⊥, ts) to r
733:     else if (RVal.ts = ts)
734:       send (WBACK_VAL_ACK, s, RValproofs, ts) to r
735:     else if (consistent(shares, RNextVerShOne))
736:       RPrev2 := RPrev
737:       RPrev := RVal
738:       RVal := RNextVal
739:       RSecShOne := RNextSecShOne
740:       RVerShOne := RNextVerShOne
741:       RSecShTwo := RNextSecShTwo
742:       RVerShTwo := RNextVerShTwo
743:       send (WBACK_VAL_ACK, s, RSecShTwo, ts) to r
744:     else if (⊥ ∈ shares)
745:       wait for ( RVal.ts ≥ ts )
746:       if (RVal.ts > ts)
747:         send (WBACK_VAL_ACK, s, ⊥, ts) to r
748:       else if (RVal.ts = ts)
749:         send (WBACK_VAL_ACK, s, RSecShTwo, ts) to r
750:     else
751:       // ignore

751:   on receive (WBACK_TS, ts, Shares) from reader r
752:     wait for ( RVal.ts ≥ ts )
753:     if (Rcts ≥ ts)
754:       READERS.remove(r)
755:       send (WBACK_TS_ACK, s, ts) to r
756:     else if (Rcts < ts ∧ consistent(Shares, RVerShTwo))
757:       Rcts := ts
758:       READERS.remove(r)
759:       send (WBACK_TS_ACK, s, ts) to r
760:     else if (⊥ ∈ PROOFS)
761:       wait for ( Rcts ≥ ts )
762:       READERS.remove(r)
763:       send (WBACK_TS_ACK, s, ts) to r
764:     else
765:       // ignore

// GetConcurrentReaders Protocol messages
765:   on receive (GET_ACT_RD_CNT) from writer
766:     send (RDRS_CNT, s, READERS.size()) to writer

767:   on receive (GET_ACT_RDS, count) from writer
768:     send (READERS, s, READERS[1:count]) to writer

769:   on receive (GET_ACT_RDS_INS, A) from writer
770:     send (RDRS_INS, s, READERS ∩ A) to writer
}

```

Figure 13: Protocol for server s , tolerating malicious readers (part 2 of 2)

reader, then at least $f + 1$ correct servers should have received and updated the value-timestamp for timestamp t

Proof: A correct server accepts a second round write back from a reader only if the reader provides at least $2f + 1$ proofs that match the *secrets* received from the writer.

At least $f + 1$ of these servers must be correct. Correct servers only divulge the proofs, in response to the first round write back message, after receiving the value-timestamp from the writer and updating it. \square

Lemma 16 (Belief WB1). *A correct server accepts a (first round) write back message from a reader, and updates the value-timestamp for timestamp t only if the writer has completed phase 0 of the write(W_0).*

Proof: Consider the first correct server s to accept a first round write back message for timestamp t .

A correct server s accepts a first round write back only if it receives $f + 1$ secret parts that can correctly reconstruct the secret. Thus at least one correct server must have revealed its secret part shares to the reader.

Since correct servers only reveal their shares on receiving the first phase write message (or on accepting a first phase write-back message) for timestamp t , it follows that the writer must have completed phase 0 of the write (W_0) before sending the first phase write message to the server. \square

Lemma 17 (Write back termination). *If a correct reader tries to write back a timestamp t for which (i) it has received from*

a correct server in the first phase of the read (in response to GET_TS) and (ii) for which it has received (ii) $f + 1$ matching responses from correct servers, then the write back eventually terminates.

Proof: Since a correct server has returned a timestamp t from the W2 phase of the write, it follows that the writer must have completed phase W0.

Thus, on receiving the write back message from the reader with the $f + 1$ correct secret parts $ShOne_i^t[1 \dots k]$, and receiving the SECRET message from the writer (during W0), all correct servers will accept the first round of write back message and will respond with $ShTwo_i^t[1 \dots k]$ (or \perp if the writer has overwritten the value).

If no correct server responds with a \perp , on receiving the responses from all the correct servers, the reader will have $(n - f)$ secret parts $ShTwo_i^t[1 \dots k]$ that can correctly reconstruct $SecretTwo^t$. Thus convincing all correct servers to accept a second round write back.

If a correct server responds with a \perp , then the writer must have sent a W2 message for timestamp $t + 1$. Thus, after receiving the W1 message for timestamp $t + 1$ (line 761) all correct servers will accept the second round of the write back.

Thus eventually, in either case, the reader will be able to convince all the correct servers and receive $(n - f)$ acknowledgements from them. Thus terminating the write back phase. \square

Lemma 18 (Read Liveness). *A read operation always terminates and is wait-free.*

Proof: Proof by Contradiction. Assume that a read operation never terminates.

Eventually the reader should receive acknowledgements for the GET_TS message from all the correct processes. Let the global time at that instance be gts_0 .

Let t be the timestamp of the last write to have completed before gts_0 , and ts_{max} be the timestamp of the $(2f + 1)$ -th smallest timestamp calculated by the reader (at gts_0).

The writer could be writing timestamp $t + 1$ but, it would not have started writing timestamp $t + 2$. Thus,

$$ts_{max} \leq t_{largest} \leq t + 1$$

where $t_{largest}$ is the largest (second phase, W2) timestamp value received from the correct servers.

If the writer has already detected the reader during a previous write, then the reader will eventually receive $f + 1$ forwarded messages from the correct servers and will be able to decide on the forwarded value-timestamp pair.

If not, by theorem ?? if the writer writes timestamp $t + 2$, the writer will detect the read operation during the GetConcurrentReaders sub-protocol for write $t + 2$. On receiving the second phase message (W2) from the writer, all correct servers (in W2) will forward values for timestamp t , $t + 1$ and $t + 2$ to the reader.

For correct servers that never receive the final message for write $t + 2$, consider the following two cases:

case 1 $ts_{max} \leq t$

At most f correct servers may not have received the value for timestamp t . When the reader receives responses from all the correct servers for the GET_VAL message sent after gts_0 , the reader will have at least $f + 1$ matching value timestamp pairs $\langle v, ts \rangle$ for timestamp $ts = t$.

Thus the reader would decide on the value with timestamp t after writing it back to the servers.

case 2 $ts_{max} = t + 1$

Since at least one correct server has updated the value of last_comp to be $t + 1$ it must be the case that at least $f + 1$ correct servers have received and updated their value-timestamp for $t + 1$ (by Lemmas 14 and 15).

Thus, when all the correct servers respond to the GET_VAL messages sent after gts_0 , the reader will have $f + 1$ matching responses for timestamp $t + 1 = ts_{max}$ to decide on.

By Lemma 17 the reader's write back phase is guaranteed to terminate because the reader is writing back ts_{max} which was received from a correct server in response to GET_TS. \square

A.2.4 Boundedness

In this section, we show that for each operation, a bounded number of messages of bounded size will be generated. The solution 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.

Lemma 19 (Boundedness). *The total number of messages exchanged between the server and the reader for each read operation is bounded*

Proof: For each read operation, the reader sends to each server a maximum of

- 1 GET_TS message,
- $f + 1$ GET_VAL messages,
- 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.

During each write back attempt, the client sends only one message. However since the shares gathered by the client may contain a few from faulty servers the reader can retry on receiving more shares. A client asks the first time when it has $f + 1$ shares, it can retry until it gets f more shares. If it gets $2f + 1$ shares, then at least $f + 1$ of those will be correct and it will be accepted. Making a total of $f + 1$ attempts.

Thus totally the first round of the write back message can be sent up to $n(2f + 5)$ times.

- and up to $n(2f + 5)$ messages for the second round of the write back.

The reader starts initiating the second round when it gets $(n - f)$ responses. It retries each time it receives a new response from the previous round. Since the total number of responses can only go up to n , the reader will have to retry only up to $f + 1$ times.

Also, since there are at most $2(f + 1)$ different timestamp values that the reader may try to decide on, the maximum number of second round messages sent by the reader is $2(f + 1) \times (f + 1)$.

Also, a reader will only receive from each server

- 1 TS message,
- up to $(f + 1)$ VALUE messages, sent in response to the GET_VAL message,
- up to 1 FWD message, forwarded in response to a concurrent write,
- up to $n(2f + 5)$ messages in response to the messages sent by the client for the first round of the write back, and
- up to $n(2f + 5)$ messages in response to the messages sent by the client in the second round.

□

Any reader or server that sends more than the maximum number of messages specified by Lemma 19 in a particular operation can be detected as faulty and ignored (by the receiver). For simplicity, we do not explicitly show this in the pseudocode provided.

Lemma 20 (Write Boundedness). *The total number of messages exchanged between the server and the writer during a write operation is bounded.*

Proof: During each write operation, the writer sends three WRITE messages and receives three acknowledgements from each server. The sizes of the replies from servers are bounded. In addition, by Theorem 4, the messages exchanged in the GetConcurrentReaders sub-protocol are bounded. □