Lightweight writeback for Byzantine storage systems

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

[†]*UT Austin*
[‡]*Arizona State University*

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 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. Timestamps generated by our solution are non-skipping and messages sent to readers and writers contain only a finite number of values, operation identifiers, and timestamps. We introduce *lightweight write back*, a new mechanism which enables readers to write back only the (non skipping) timestamp of the value read and not the value itself. This is particularly important since Byzantine readers that write back a value could force the servers to process infinitely large messages, whereas a non-skipping timestamp is practically finite in size. With a novel use of secret sharing techniques combined with writeback throttling, we manage to tolerate Byzantine readers without the use of any unproven cryptographic 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, 6, 8, 9, 10]. 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 [7] read and write semantics are possible, but stronger semantics are not []. We consider implementations with atomic semantics in this paper.

We consider solutions in a system of n servers that do not communicate with each other 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 solution 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 [10, 5].

Of particular interest are solutions that bound the bandwidth consumed by client-server communication. Bazzi and Ding [4] present a solution that (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. Further, the solution requires at least $4f + 1$ servers. In [4], a distinction is made between *amortized bounded* solutions and *bounded* solution(See section 3.3). It is natural to ask whether it is possible to provide similar (or even better) boundedness properties when $n \leq 4f$.

In this paper, we show that it is possible to have a wait-free f-resilient atomic solution in the non-communicating server model that requires only

$3f + 1$ servers, satisfies all the amortized boundedness guarantees of Bazzi and Ding's solution, and provides boundedness guarantees also on the size of messages sent to writers. In [4], a distinction is made between *amortized bounded* solutions and *bounded* solution. For write operations, we show only amortized boundednes. For read operations, we show amortized boundedness but, using the techniques of [4], we can transform this solution into a bounded solution. Our solution tolerates Byzantine readers without the use of any unproven cryptographic assumptions, such as hardness of factoring or of computing discrete logarithms, on which public-key cryptography depends. Further, it differs from all previous solutions that achieve atomic semantics in the non-communicating server model (bounded or otherwise) in requiring readers to write back only the timestamps of the values they read, rather than the values themselves. Such *lightweight write back*, while intriguing in itss own right, is of particular interest when considering Byzantine readers: if the values written by writers are not of fixed size (large files for example), a Byzantine reader can write back a very large value (potentially unbounded) and force the servers to check whether the written back value is valid—this is true even for solutions that use unproven cryptographic assumptions. By writing back only the the timestamp, a read operation always costs a server a bounded amount of work to process. To our knowledge, this issue was not considered in previous work.

To achieve our results, we build on existing work and introduce some new techniques. We use the concurrent-reader-detection and the write back-throttling ideas from 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 reduce increase the resiliency of our solution by estimating the timestamp of the value to be read in a new way. Instead of choosing the $f + 1$'st largest timestamp as the potential timestamp of a value to read, we choose the $2f+1$'st smallest timestamp amongst the timestamps received. While the $f + 1$'st timestamp worked well for $n = 4f + 1$ it does not work well for $n = 3f + 1$. By choosing the $2f + 1$'st smallest timestamp, we are, in a sense, assuming that all the servers that are not heard from have higher timestamps and then choosing the $f + 1$'st largest amongst the timestamps of *all* servers, whether they have been heard from or not. What makes this approach work is the continuous update of the $2f+1$'st smallest timestamp as responses are received from new servers.

Writing back timestamp We do away with the requirement of writing back values by adding an extra communication round to the write operation which enables the reader to ascertain, when it writes back a timestamp, that the writer's message containing the corresponding value will be eventually received by all correct servers.

Bounding message sizes to writers We bound the sizes of messages sent to servers using three rounds of communication betwwen 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 writeback throttling to tolerate Byzantine readers without using unproven cryptographic assumptions. The idea is for the writer to associate with each value it writes both a primary secret (a polynomial of degree f), and a set of secondary secrets (random strings). Each server receives from the writer a share of the primary secret, a unique set of secondary secrets, plus a copy of one of the secondary secrets sent to each of the other servers (we call these copies *proofs*). A correct server will not divulge to a reader its share of the primary secret or its proofs for a particular value unless it can ascertain that the write operation for that value has made sufficient progress. Further, a correct server will not allow a reader to write back a value (or, in our case, the timestamp of a value) unless the reader can present (i) in the first round, sufficiently many shares that are consistent with the share stored at the server, and (ii) in the second round, sufficiently many proofs matching the stored secrets at the server.

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. The number of read operations in the system is exponential in the size of the operation identifiers. Similarly write operations are identified by the writer identifier and the timestamp of the value being written. Since timestamps are non-skipping, write operation can also be represented by finite strings in practice.

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 be Byzantine faulty and may deviate arbitrarily from the specified protocol. The remaining $(n - f)$ servers are correct and follow the specified protocol. We require that the total number of servers n be at least $3f + 1$.

We assume that clients cannot spoof each other's or servers messages and that servers cannot spoof each other's or client messages. While such an assumption can be enforced in practice using cryptographic techniques with unproven assumptions, such techniques are not required to enforce this assumption. We assume FIFO point-to-point asynchronous channels between clients and servers. Servers do not communicate with other servers.

Writers can fail by crashing and follow their protocol before they crash. In section 3 we assume that the readers can fail only by crashing. Later, we relax this assumption and we consider Byzantine readers in section 4. 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 $2^{-k'}$.

3 Lightweight Write Back

We now present a protocol that implements a wait-free atomic register using $3f + 1$ replicas and bounded number of messages, where the reader does not have to write back the value. The reader instead only writes back the timestamp associated with the value, which, in practice, can be thought of as bounded since our protocol implements non-skipping timestamps [3].

Figures 1 - 3 present a single-writer-multiple-reader version of the protocol that assumes benign readers. In Section 4 we extend this protocol to handle Byzantine readers. This protocol can also be easily extended to support multiple-writers, using ideas from [4], as described in Section 5.

3.1 Protocol Overview

In our protocol, the writer writes in three phases. In the first phase, the writer writes a value/timestamp pair and waits for replies from $n - f$ servers. The second and third phase are more subtle. In the second phase, the writer sends a message to each server indicating that the first phase is finished. In the third phase, the writer sends a message to each server indicating that the second phase is finished. When a correct server receives a second phase message it can conclude that, if the writer executed no further writes, at least $f + 1$ correct servers have received the (value,timestamp) pairs sent in the first phase. Similarly, if a server receives a third phase message, it can conclude that at least $f + 1$ correct servers have received the second phase message. The reason for the second and third phases will become clearer when we describe the reader's protocol.

To understand the reader's protocol, we consider a simple scenario. The reader starts by requesting third phase information from the servers. Each server replies with the most current timestamp for which it knows that the corresponding write operation reached its third phase. Now, assume that the reader receives replies from all correct servers in

response to its request for third 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 operation during that time. Of special interest is the largest third 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 third phase response with $t_{largest}$, it can simply request all second phase and 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. Also, having received $f + 1$ second phase messages for $t_{largest}$, the reader knows that the writer completed the first phase of the write (the one in which the value is written): the reader therefore does not need to write back the value (by redoing the first phase of the writer) and can limit itself to completing the second and the third phase of the write. Hence, at least in this simple scenario, it is possible for the reader to maintain atomic semantics by writing back only the timestamp of the value read and not the value itself.

While this scenario is instructive, it is too simple—a number of serious complications can occur. 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 the 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. This is the same approach taken in [4], but our protocol for detecting concurrent read operations consumes bounded bandwidth, whereas that of [4] is unbounded. Our solution guarantees that if the writer completes the write of a value whose timestamp is larger than $t_{largest} + 1$, then it will be able to detect any concurrent reader and such readers will be able to terminate. But if the writer writes values with timestamps $t_{largest} + 1$ to all correct servers and $t_{largest} + 2$ to only some correct servers and those servers replace the values they store with these newer values, then the reader might not be able to receive enough values whose timestamp is $t_{largest}$ and therefore be unable to terminate. To avoid having the values whose timestamps are equal to $t_{largest}$ erased, we require servers to keep the three most up to date written values. This way, either the reader is independently able to decide which value to read (when the writer is not fast) or the writer detects the reader and helps the reader to decide on a value by asking the servers to forward the latest value written.

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 third 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 second and 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 not able to decide, then there are other correct servers whose replies are not amongst the $n - f$ replies, and, waiting long enough to decide, the reader will eventually receive some message from one of the remaining correct servers. When it receives more messages while it is waiting to decide, the reader recalculates $t_{largest}$ with the assumption that, with the new messages it received, it must finally have replies from all correct servers: therefore, the reader re-requests first and second 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 largest third phase timestamp. We will show that this is suf-

ficient to guarantee that the reader can decide and that its decision is valid.

3.2 Protocol Description

We assume that a writer will not write a new value until it finishes writing the previous value. We also assume that each message is tagged with the id of the operation to which it pertains and messages that do not pertain to the current operation are ignored. For simplicity, we do not show these tags in the code.

The Write operation: The main write operation is performed in three phases (rounds of message exchanges). The writer also runs the GetConcurrentReaders sub-protocol in parallel with the first two phases. To proceed to the third phase the writer waits for the completion of the second phase and for the GetConcurrentReaders sub-protocol to terminate.

The GetConcurrentReaders sub-protocol is explained in Section 3.4. We now present the main write operation assuming that GetConcurrentReaders (i) always terminates and consumes bounded bandwidth, and (ii) returns to the writer the identifiers of all concurrent read operations that are known to all correct servers and have not terminated before GetConcurrentReaders ends.

In the first phase of a write operation (W1), the writer sends the value-timestamp pair that it intends to write to all the servers after starting the GetConcurrentReaders sub-protocol in parallel. On receiving the value-timestamp pair, the servers store this information in *RNextVal* (line 147) and acknowledges the writer. The writer waits to collect at least $(n-f)$ acknowledgements before proceeding to the next phase.

In the second phase (W2), the writer sends a message to all the servers to update the current value-timestamp pair at the servers. On receiving this message, the servers replace *RVal* with *RNextVal* , and update *RPrev* and *RPrev2* accordingly, before acknowledging the writer. The writer again has to wait to receive at least $(n-f)$ acknowledgements.

Before beginning the third phase (W3), the writer waits for the GetConcurrentReaders sub-protocol to end. The writer then sends a message to each server asking it to (i) forward the values stored at the server to the concurrent readers detected by the writer, and (ii) update the value of *Rts* . After forwarding the messages and updating the set of active readers, the server acknowledges the writer.

The write operation completes when the writer receives $(n-f)$ acknowledgements in the third phase.

The Read operation: The read operation can also be considered to be taking place in three phases. In the first two phases, the reader contacts the servers to gather a value that satisfies both the validity (*valid*) and freshness (*notOld*) criterion. In these phases, the reader also collects responses from servers, that are not solicited by the reader, but have been forwarded by the servers because of the writer (FWD messages in lines 12 and 44). To proceed to the third phase, the reader waits either (i) until it finds a value that satisfies both the validity and freshness conditions, or (ii) until it knows that it has a value that has been forwarded by the writer (*fwded* is true). On finding such a value, the reader writes back the timestamp in phase 3 to complete the read.

In the first phase (R1), the reader asks for the timestamp of the latest completed write at each server. This information is received by the servers during the third phase of the write (W3) and is stored in *Rts* . On receiving this request, the servers respond with the timestamp stored in *Rts* . The reader waits to collect at least $(n-f)$ responses before it begins the next phase.

In the second phase (R2), the reader requests the servers for the values stored at the server and collects them. In this phase the reader will re-request the servers for the values, if the reader receives a timestamp response from one of the f servers that did not respond in the previous phase (R1). The reader keeps collecting responses and re-requesting values, until it finds a value-timestamp pair that satisfies both the validity and freshness criterion (i.e *notOld* and *valid* hold true in line 126).

In the final phase, the reader writes back the timestamp corresponding to the value-timestamp pair that either satisfies both the validity and freshness conditions or has been forwarded by the writer. The write back is done in two rounds, one after an-

other, and correspond to the writer's operation for phases W2 and W3.

For the first write back round, the reader only sends the timestamp of the value that is decided upon to all the servers. On receiving this timestamp, the servers wait until they directly receive the writer's message from phase W1, to update the value, before acknowledging the reader. The reader has to wait for $(n - f)$ acknowledgements before starting the next round. In the second round of the write back, the reader asks all the servers to update their timestamp for the latest completed write. On receiving $(n - f)$ acknowledgements for the second round, the read operation completes.

```
    write( ) {
1:      inc(ts)

        // Phases W1–W2
2:      cobegin {
            writeVal();
            CR = GetConcurrentReaders()
        } coend

        // Phase W3
3:      send (WRITE_TS, ts, CR) to all
4:          wait for (n − f) acks.
    }

    writeVal( ) {
        // Phase W1
5:      send (NEXT_VAL, ⟨v, ts⟩) to all
6:          wait for (n − f) acks.

        // Phase W2
7:      send (WRITE_VAL) to all
8:          wait for (n − f) acks. }
```

Figure 1: The Writer's Protocol

3.3 Protocol Correctness

We show that the protocol implement 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.

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 this ordering, to prove atomicity, it is sufficient to prove the following lemmas (whose proofs can be found in the Appendix).

Lemma 1. *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 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 $x \leq t$. □

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 *Rts* 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 $x \leq t$. □

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 *Rts* 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 $x \leq t - 1$. □

3.3.1 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. □

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. □

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 RequestValue() phase initiated 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 RequestValue() phase, 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. □

3.4 Bounded Detection of Readers

The protocol requires that the writer be able to detect the presence of ongoing read operations—more specifically, 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). We recall here that a read operation is uniquely determined by a reader identifier and an operation tag. We also recall that a reader does not issue a read operation before starting finishing a previous operation. 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]. Unfortunately, when it receives a list of allegedly active reader operations, the writer has no way, in this implementation, to determine whether the received sets contain operations that are indeed active or are just the fabrication of a faulty server: 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 (and thus the size of a reader's identifier) 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.

3.4.1 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 a 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.

3.4.2 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. □

Theorem 2 (Detection). *If a read operation r never terminates and the writer starts executing the 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. □

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. □

Theorem 4 (Boundedness). *The total number of messages exchanged between the server and the writer during the GetConcurrentReaders subprotocol 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. □

4 Byzantine Readers

We now explain how the protocol presented in Section 3 can be modified to tolerate Byzantine readers without the use of unproven cryptographic assumptions. In the following, we call the protocol of Section 3 the non-reader-tolerant protocol, or NRT, protocol.

Overview: We start by recalling some properties of NRT protocol. When a correct server receives a third phase W3 message from a writer, it believes that at least $f + 1$ correct servers have received and updated their latest value with the value/timestamp pair for which a timestamp is being written in W3. This belief is true because the writer has completed phases W1 and W2 and received $n - f$ acknowledgements from servers, at least $f + 1$ of which are correct. If a reader is benign, (i) write back throttling during the first round of write back, and (ii) the requirement that a *benign* reader finishes the first round of the write back before proceeding to the second, ensure that this belief holds true when a correct server accepts a second round write back (to update the value of *Rts*). However, if the reader is Byzantine, the reader may not wait for the completion of W1 by the writer before starting its first round of write back. Also, a reader might proceed to its second round of write back without finishing its first round of write back. Thus accepting the write back without any checks can violate the correctness of the protocol. We need mechanisms that allow readers to *prove* that the writer finished its W1 phase for a given timestamp in order for servers to accept their first round writeback message for that timestamp. Similarly, readers must be able to prove that enough servers accepted their first round writeback messages in order to proceed to the second round of writeback.

These mechanisms can be realized as follows. In the first phase of a write operation, the writer generates a random secret polynomial P of degree f and distributes random secret shares to the servers so that any $f + 1$ shares can be used to reconstruct the secret. The share of a server s is simply $P(s)$ assuming server identifiers are the integers 1 though n. The writer does not send the secret polynomial to any server. A server is not supposed to divulge a

share unless it has received a W2 message from the writer for the write operation for which it received a share (or, alternatively, if it can be convinced that the writer has sent a W2 message to a correct server). A reader can *prove* its first writeback to server q by providing $f+1$ shares that can be used to construct a polynomial P' such that $P'(q)$ is equal to share of q (we show that with very low probability a Byzantine reader can fabricate $f+1$ shares that can be used to match the share of a correct server). By providing these shares, a reader is essentially providing a proof with very high probability that a correct server received a W2 message from the writer for the timestamp being written back. After receiving a *proof* of a writeback, a server is convinced that the writer has progressed to phase W2 and from that time on it will be willing to divulge its share.

A complication with the protocol occurs if a reader that gets $f+1$ matching timestamps (and values), say ts, with purported proofs is not able to *convince* enough servers to accept its first round write back message (for instance, if the reader receives f responses from faulty servers that provide bogus shares). In that case, the reader might not be able to terminate. In this situation there are two possibilities: either the writer sent $W2$ messages for ts to $f+1$ correct servers or the writer stopped before sending these messages. In the first case, the reader will eventually get from $f+1$ correct servers matching timestamps and shares that enable it to proceed with its first round of write back. In the second case, the writer clearly did not progress to phase W3 for timestamp ts, which means that the $2f+1$'st smallest timestamp on which the reader is basing its write back is not correct (too large). In fact, it must be the case that eventually the reader will receive more messages from correct servers which enable it to calculate a smaller $2f+1$'st smallest timestamp. When the reader receives *all* messages from correct servers, the $2f+1$'st timestamp it calculates will be a timestamp for which the writer started the W3 phase and finished the W2 phase and the reader will be able to collect $f+1$ timestamps and corresponding shares to finish its first round of writeback.

One issue has to do with the number messages that the reader sends before it is guaranteed to finish its first write back round. This number is f^2. In fact, if messages that enable the reader to update its $2f+1$'st smallest timestamp are slow in arriving, the reader would have to try to redo its first round of write back each time it receives a message from a server that did not reply to its earlier attempt at write back. In fact, every time a reader receives a late reply to its round one write back message, it collects all the shares it has received so far and it sends those shares to servers. Servers try to find $f+1$ shares amongst these shares that they can use to reconstruct their own share (note that the server might need to consider and exponential number of possibility (exponential in f) when looking for $f+1$ shares that can reconstruct the secret. The security parameter has to be chosen large enough to take into consideration that the probability increases with the increase in the number of combinations considered). The set of shares can increase at most f times after which the reader must have received messages from all servers and can give up on writing back a value with timestamp ts. Also, the reader can update the $2f+1$'st smallest timestamp f times, For each one of these updates the reader would have to go through the process just described in f rounds for a total of f^2 rounds.

Once a reader manages to finish its first round of write back it has to convince servers to allow it to go for a second round of write back. To that end, it needs to provide proof that enough correct servers must have the value written by the writer and accepted the first round of write back from the reader. Our mechanism for helping a reader in its second round of write back does not use secret sharing and instead uses *matching* secrets. (A complication might occur because of write back messages that arrive late at the servers. We will assume in the discussion that there are no late writebacks and we address that problem later.) In the first round of a write, the writer sends to each server p n *writeback secrets*, which are random k-bit strings where k is a security parameter. The writer sends wb_secret_{pq}, one for each server q, Also, in the same round, the writer provides every server q with a *writeback proof* string $wb\_proof_{qp} = secret_{pq}$, one for each server in the system. A server responds with a write-

back proof if it accepts the first round writeback message sent by the reader. When a reader finishes its first write back round, it knows that $f + 1$ correct servers accepted the first round write back. This means that the first round write back message will eventually be accepted by all correct servers because the shares provided by these $f + 1$ servers will be enough to convince all correct servers. So, eventually the reader will receive $2f + 1$ *writeback proofs* and the reader presents these proofs when attempting its second round of write back. A server accepts a second phase writeback only if the reader provides $2f + 1$ matching proofs (which guarantees that $f + 1$ correct servers must have accepted the first phase writeback). It is possible that some of the collected proofs are from faulty servers and the second phase writeback will not be accepted (but not rejected either). While waiting for an acknowledgment of its second phase writeback, the reader will send any new acknowledgment of its first phase writeback to all servers. So, eventually, the reader will get $2f + 1$ acknowledgments from the correct readers because, as we have argued, all the correct readers will get the first phase message from the writer. This will guarantee that the reader will be able to collect $2f + 1$ proofs that will be accepted by $2f + 1$ correct servers.

Late writebacks: There is another complication due to late writebacks. If a writeback arrives late at a server, the server might not have the proof to give the reader because the old proofs might have been replaced with newer ones due to subsequent writes. If a server that receives a writeback message, and has a current timestamp that is larger than the timestamp being written back, it simply sends a writeback acknowledgment, but without a proof.

The meaning of a writeback without proof is that the writer started the second phase of the write of a value with a higher timestamp. So, when the reader finishes its first round of write back, it will collect $2f + 1$ acknowledgments, some with proofs and some without proofs and send these along with its second phase writeback. If one of the acknowledgments without proofs is from a correct server, then this means that the writer must have started writing a new value and finished the third phase of the write operation for which the reader is sending a second phase writeback, and therefore all correct servers will eventually receive the third phase message from the writer and can accept the writeback. If none of the acknowledgments without proofs is from a correct server, then the reader will eventually receive either enough proofs (as we argued in the previous paragraph) or one acknowledgment without proof from a correct server; in either case, the correct reader will be able to finish its second phase writeback.

It should be clear from the description that a correct server will not accept a second phase writeback unless $f + 1$ correct servers accepted the first phase write back. Thus if a correct server accepts a second phase write back then its belief holds true. Also, the read operation initiated by a correct reader always terminates.

Figures 5 - 9 present the pseudocode for handling Byzantine readers. The protocol for detection of concurrent readers is same as the one presented in Figure 4.

4.1 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.

4.1.1 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 $i \in \{1 \ldots, n\}$, 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 $2^{-(k-\log(f(2f+1))-(f+1)n)}$. □

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.

4.1.2 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. □

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 *Rts* 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 $x \leq t$. □

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 *Rts* 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 $x \leq t-1$. □

4.1.3 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. □

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. □

Lemma 14 (Belief 1). *If a correct server sets its value of Rts to t in response to a third phase message (W3) 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 W3 for timestamp t after receiving $n-f$ responses from W2. Correct servers respond to W2 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. □

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

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 1 of the write(W1).*

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 $f+1$ shares that are consistent with the share it holds. Thus at least one correct server must have revealed its share to the reader.

Since correct servers only reveal their share on receiving the second phase write message (or on accepting a first phase write-back message) for timestamp t, it follows that the writer must have completed first phase write (W1) before sending the second phase write message to the server. □

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 W3 phase of the write, it follows that the writer must have completed phase W1.

Thus, on receiving the write back message from the reader with the $f+1$ correct shares, and receiving the W1 message from the writer, all correct servers will accept the first round of write back message and will respond with either the wb_proof (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 $2f+1$ matching proofs that can convince all the correct servers to accept a second round write back.

If a correct server responds with a $\perp$, then the writer must have sent a W3 message for timestamp $t+1$. Thus, after receiving the W2 message for timestamp $t+1$ (line 198) 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. □

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 (third phase, W3) 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 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:

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 RequestValue() phase initiated after gts_0, the reader will have at least $f + 1$ matching value timestamp pairs $\langle v, ts \rangle$ for timestamp $ts = t (= ts_{max})$.

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 RequestValue phase initiated 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. □

4.1.4 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,
- $(f + 1) \times (f + 1)$ messages for the first round of the write back.

  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.

  Also, the value of the $(2f + 1)$-th smallest timestamp can be updated up to f times after receiving $n - f$ responses. This can cause the write back to be initiated for up to $f + 1$ different timestamps. (for a less tighter bound note that *acceptable* only holds for up to n different values).

  Thus totally the first round of the write back message can be sent up to $(f + 1) \times (f + 1)$ times.

- and up to $(f + 1) \times (f + 1)$ 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 in 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 $(f + 1)$ different timestamp values that the reader may try to decide on, the maximum number of second round messages sent by the reader is $(f+1) \times (f + 1)$.

Also, a reader will only receive from each server

- 1 TS message,
- up to $(f + 1)$ VALUE messages, sent in response to the GET_VALUE message,
- up to 1 FWD message, forwarded in response to a concurrent write,
- up to $(f + 1) \times (f + 1)$ messages in response to the messages send by the client for the first round of the write back, and
- up to $(f+1) \times (f+1)$ 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. □

5 Multiple Writers

The protocol presented in Section 3 can eazily 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 server. To perform a read operation, the reader preforms 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 preforms 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.

6 Related Work

Distributed storage systems have been widely studied in [1, 2, 3, 4, 5, 6, 8, 9, 10]. 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 [9]. 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 or readers and writers using the optimal $3f + 1$ replicas [10]. 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 [6]. 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 boundedness 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, 8, 10]. 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.

currsize

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, London, UK, 2004. Springer-Verlag.

[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, London, UK, 2006. Springer-Verlag.

[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. Guerraoui and M. Vukolic. Refined Quorum Systems. Technical Report LPD-REPORT-2007-001, EPFL, 2007.

[7] L. Lamport. On interprocess communication. part i: Basic formalism. *Distributed Computing*, 1(2):77–101, 1986.

[8] B. Liskov and R. Rodrigues. Byzantine clients rendered harmless. In *DISC 2005*, pages 311–325, London, UK, 2005. Springer-Verlag.

[9] D. Malkhi and M. K. Reiter. Secure and scalable replication in phalanx. In *Proc. 17th SRDS*, pages 51–58, 1998.

[10] J.-P. Martin, L. Alvisi, and M. Dahlin. Minimal byzantine storage. In *DISC '02*, pages 311–325, London, UK, 2002. Springer-Verlag.

Definitions:

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

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

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

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

   // Phase R1
 9:  RequestTimestamp()

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

12:     on receive (FWD, s, ⟨v, ts⟩, Vals) from server s
13:        fwd[s] = ⟨v, ts⟩
14:        Values[s] = Values[s] ∪ Vals

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

   // Phase R2
16: RequestValue()

   repeat
17:     on receive (TS, s, ts) from server s
18:        last_comp[s] = ts
19:        RequestValue()

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

22:     on receive (FWD, s, ⟨v, ts⟩, Vals) from server s
23:        fwd[s] = ⟨v, ts⟩
24:        Values[s] = Values[s] ∪ Vals

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

   // Phase R3
26: WriteBack(ts_c)
27: return ⟨v_c, ts_c⟩
}

RequestTimestamp( ) {
   send (GET_TS) to all
}

RequestValue( ) {
   send (GET_VAL) to all
}

WriteBack( ts ) {
   // Round 1
28: send (WBACK_VAL, ts) to all
29:    wait for (n − f) acks.

   // Round 2
30: send (WBACK_TS, ts) to all
31:    wait for (n − f) acks.   }
```

Figure 2: Reader's protocol

Initialization:

READERS := ∅

RNextVal := ⊥

```
server( ) {
      // Write Protocol messages
32:   on receive (NEXT_VAL, ⟨v, ts⟩) from writer
33:      RNextVal := ⟨v, ts⟩

34:   on receive (WRITE_VAL) from writer
35:      if ( RVal .ts < RNextVal .ts)
36:         RPrev2 := RPrev
37:         RPrev  := RVal
38:         RVal   := RNextVal

39:      send WRITE-ACK1 to the writer

40:   on receive (WRITE_TS, ts, CR) from writer
41:      if ( Rts < ts)
42:         Rts := ts

43:      for each r ∈ CR:
44:         send (FWD, s, RVal , { RVal , RPrev , RPrev2 }) to r
45:      READERS = READERS \ CR
46:      send WRITE-ACK2 to the writer

      // Read Protocol messages
47:   on receive (GET_TS) from reader r:
48:      READERS .enqueue(r)
49:      send (TS, s, Rts ) to r

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

52:   on receive (WBACK_VAL, ts) from reader r
53:      wait for ( RNextVal .ts ≥ ts )
54:      if ( RVal .ts < ts)
55:         RPrev2 := RPrev
56:         RPrev  := RVal
57:         RVal   := RNextVal

58:      send WBACK-ACK1 to r

59:   on receive (WBACK_TS, ts) from reader r
60:      wait for ( RVal .ts ≥ ts )
61:      if ( Rts < ts)
62:         Rts := ts

63:      READERS .remove(r)
64:      send WBACK-ACK2 to r

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

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

69:   on receive (GET_ACT_RDS_INS, A) from writer
70:      send (RDRS_INS, s, READERS ∩ A) to writer   }
```

Figure 3: Protocol for server s

```
    Definitions:
    notLarge( s ) ≜ |{x : count[x] ≥ count[s]}| ≥ f + 1

    GetConcurrentReaders() {
        ∀s: readers[s]  :=  ⊥
        ∀s: count[s]    :=  ⊥
        ∀s: sent[s]     :=  false
        union_set       :=  ⊥

        send (GET_ACT_RD_CNT) to all servers

        repeat
71:         on receive (RDRS_CNT, s, count) from server s
72:             count[s] = count
73:         ∀p: if (notLarge(p) ∧ sent[p] = false)
74:                 send (GET_ACT_RDS) to server p
75:                 sent[p] := true

76:         on receive (READERS, s, R) from server s
77:             if ( ¬ sent[s] ∨ (sent[s] ∧ count[s] ≠ |R| ))
78:                 detect failure of s
79:             else
80:                 readers[s] := R
81:     until (|{readers[s] : readers[s] ≠⊥}| ≥ f + 1)

82:     union_set := ∪s readers[s]
83:     for each (s : sent[s] ≠ true)
84:         send (GET_ACT_RDS_INS, union_set) to server s

        repeat
85:         on receive (READERS, s, R) from server s
86:             if ( ¬ sent[s] ∨ (sent[s] ∧ count[s] ≠ |R|))
87:                 detect failure of s
88:             else
89:                 readers[s] := R

90:         on receive (RDRS_INS, s, R) from server s
91:             if ( R ⊄ union_set )
92:                 detect failure of s
93:             else
94:                 readers[s] := R
95:     until (|{s : readers[s] ≠⊥}| ≥ n − f)

96:     CR = { x : |{s : x ∈ readers[s]}| ≥ (f + 1) }
97:     return CR
    }
```

Figure 4: Bounded detection of readers: Writer code

```
    write( ) {
98:         inc(ts)
99:         generateSecretsAndProofs()

            // Phases W1–W2
100:        cobegin {
              writeVal();
              CR = GetConcurrentReaders()
            } coend

            // Phase W3
101:        send (WRITE_TS, ts, CR) to all
102:          wait for (n − f) acks.
    }

    writeVal( ) {
            // Phase W1
103:        ∀s: send ( NEXT_VAL, ⟨v, ts, share_s, wb_secrets_s, proofs_s⟩, )
                    to server s

            // Phase W2
104:        send (WRITE_VAL) to all
105:          wait for (n − f) acks.
    }
```

Figure 5: The Writer's Protocol for handling Byzantine Readers

Definitions:

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

acceptable($\langle v, ts \rangle$) $\triangleq$ $\exists s : last\_comp[s] = ts \wedge writingBack(ts) = false$

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

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

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

read( ) {

$\forall s$: last_comp[s] = $\perp$

$\forall s$: fwd[s] = $\perp$

$\forall s$: Values[s] = $\emptyset$

$\forall ts$: writingBack[ts] = false

written_back_value = $\perp$

// Phase R1

106: **send** (GET_TS) **to all**

107: **repeat**

108: **on receive** (TS, s, ts) from server s

109: last_comp[s] = ts

110: **on receive** (FWD, s, $\langle v, ts, sh \rangle$, $Vals$) from server s

111: fwd[s] = $\langle v, ts, sh \rangle$

112: Values[s] = Values[s] $\cup$ $Vals$

113: **on receive** (VALS, s, $Vals$) from server s

114: Values[s] = Values[s] $\cup$ $Vals$

115: **until** ($|\{x : last\_comp[x] \neq \perp\}| \geq n - f$)

// Phase R2

116: **send** (GET_VAL) **to all**

117: **repeat**

118: **on receive** (TS, s, ts) from server s

119: last_comp[s] = ts

120: **send** (GET_VAL) **to all**

121: **on receive** (VALS, s, $Vals$) from server s

122: Values[s] = Values[s] $\cup$ $Vals$

123: **on receive** (FWD, s, $\langle v, ts, sh \rangle$, $Vals$) from server s

124: fwd[s] = $\langle v, ts, sh \rangle$

125: Values[s] = Values[s] $\cup$ $Vals$

if $\Big($ $\exists \langle v_c, ts_c \rangle$: fwded($\langle v_c, ts_c \rangle$)

126: $\vee$ (notOld($\langle v_c, ts_c \rangle$) $\wedge$ acceptable($\langle v_c, ts_c \rangle$) $\wedge$ valid($\langle v_c, ts_c \rangle$)) $\Big)$

127: writingBack[ts_c] = true

128: **fork** WriteBack($\langle v_c, ts_c \rangle$)

129: **until** (written_back_value $\neq \perp$)

130: **return** written_back_value

}

Figure 6: Reader's protocol (part 1 of 2)

```
    WriteBack(⟨v_c, ts_c⟩) {
        // Round 1
131:    Shares = GetShares(⟨v_c, ts_c⟩)
132:    send (WBACK_VAL, ts_c, Shares) to all
133:    repeat
134:        on receive (WBACK_VAL_ACK, s, proofs_or_bottom_s, ts) from server s
                ACKS1 = ACKS1 ∪ {proofs_or_bottom_s}

135:        if ( Shares ≠ GetShares(⟨v_c, ts_c⟩))
136:            goto line 131

137:    until WBACK_VAL_ACK's are received from n − f different servers
        // Round 2
138:    local wb2_cnt=0;
139:    ACKS1 = set of WBACK_VAL_ACK messages received
140:    send (WBACK_TS, ts, ACKS1) to all
141:    repeat
142:        on receive (WBACK_VAL_ACK, s, proofs_or_bottom_s, ts) from server s
                ACKS1 = ACKS1 ∪ {proofs_or_bottom_s}
143:            goto line 139

144:        on receive (WBACK_TS_ACK, s, ts) from server s
145:            wb2_cnt++
        until wb2_cnt ≥ n − f
        written_back_value = ⟨v_c, ts_c⟩
    }
```

Figure 7: Reader's protocol (part 2 of 2)

Definitions:

consistent(Shares, given_share) $\triangleq$ Shares contains at least $f+1$ shares, such that the secret polynomial $\mathcal{F}$ generated by these $f+1$ shares agrees with the given_share (i.e. given_share = $\mathcal{F}$[s]).

secretsMatch(Proofs, mySecrets) $\triangleq$ At least $2f+1$ proof values in Proofs match the corresponding values in mySecrets i.e. $|\{x|\exists y : mySecrets[x] = Proofs[y][x]\}| \geq 2f+1$

Initialization:

READERS := $\emptyset$
RNextVal := $\perp$

server( ) {

// Write Protocol messages

on receive (NEXT_VAL,
146: $\langle v, ts, share_s, (wb\_secret_{s1}, wb\_secret_{s2}, \ldots, wb\_secret_{sn}), (proof_{s1}, proof_{s2}, \ldots, proof_{sn}), \rangle$) from writer
RNextVal := $\langle v, ts, share_s \rangle$
RNextWBproof := $(proof_{s1}, proof_{s2}, \ldots, proof_{sn})$
RNextWBsecret := $(wb\_secret_{s1}, wb\_secret_{s2}, \ldots, wb\_secret_{sn})$
147:

148: **on receive** (WRITE_VAL) from writer
149: **if** (*RVal* .ts < *RNextVal* .ts)
150: *RPrev2* := *RPrev*
151: *RPrev* := *RVal*
152: *RVal* := *RNextVal*
153: *RValsecrets* := *RNextWBsecret*
154: *RValproofs* := *RNextWBproof*

155: **send** WRITE-ACK1 to the writer

156: **on receive** (WRITE_TS, ts, CR) from writer
157: **if** (*Rts* < ts)
158: *Rts* := ts

159: **for each** $r \in CR$:
160: **send** (FWD, s, *RVal* , { *RVal* , *RPrev* , *RPrev2* }) to r
161: *READERS* = *READERS* $\setminus$ CR
162: **send** WRITE-ACK2 to the writer

// Read Protocol messages

163: **on receive** (GET_TS) from reader r:
164: *READERS* .enqueue(r)
165: **send** (TS, s, *Rts*) to r

166: **on receive** (GET_VAL) from reader r
167: **send** (VALS, s, { *RVal* , *RPrev* }) to r

Figure 8: Protocol for server s (part 1 of 2)

```
168:    on receive (WBACK_VAL, ts, shares) from reader r
169:        wait for ( RNextVal .ts ≥ ts )
170:        if ( RVal .ts > ts)
171:            send (WBACK_VAL_ACK, s, ⊥, ts) to r
172:        else if ( RVal .ts = ts)
173:            send (WBACK_VAL_ACK, s, RValproofs , ts) to r
174:        else if (consistent(shares, RNextVal .share))
175:            RPrev2 := RPrev
176:            RPrev  := RVal
177:            RVal   := RNextVal
178:            RValsecrets    := RNextWBsecret
179:            RValproofs     := RNextWBproof
180:            send (WBACK_VAL_ACK, s, RValproofs , ts) to r
181:        else if ((⊥∈ shares ))
182:            wait for ( RVal .ts ≥ ts )
183:            if ( RVal .ts > ts)
184:                send (WBACK_VAL_ACK, s, ⊥, ts) to r
185:            else if ( RVal .ts = ts)
186:                send (WBACK_VAL_ACK, s, RValproofs , ts) to r
            else
187:            // ignore


188:    on receive (WBACK_TS, ts, PROOFS) from reader r
189:        wait for ( RVal .ts ≥ ts )
190:        if ( Rts ≥ ts)
191:            READERS .remove(r)
192:            send (WBACK_TS_ACK, s, ts) to r
193:        else if ( Rts < ts ∧ secretsMatch(PROOFS, RValsecrets ))
194:            Rts := ts
195:            READERS .remove(r)
196:            send (WBACK_TS_ACK, s, ts) to r
197:        else if ( ⊥∈ PROOFS )
198:            wait for ( Rts ≥ ts )
199:            READERS .remove(r)
200:            send (WBACK_TS_ACK, s, ts) to r
            else
201:            // ignore

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

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

206:    on receive (GET_ACT_RDS_INS, A) from writer
207:        send (RDRS_INS, s, READERS ∩ A) to writer
}
```

Figure 9: Protocol for server s, (part 2 of 2)