Efficient Robust Storage using Secret Tokens*

Technical Report TR-TUD-DEEDS-07-01-2009

Dan Dobre, Matthias Majuntke, Marco Serafini and Neeraj Suri
{dan,majuntke,marco,suri}@cs.tu-darmstadt.de

August 3, 2009

Abstract

We present algorithms that reduce the time complexity and improve the scalability of robust storage for unauthenticated data. Robust storage ensures progress under every condition (wait-freedom) and never returns an outdated value (regularity) nor a forged value (Byzantine fault tolerance). The algorithms use *secret tokens*, which are values randomly selected by the clients and attached to the data written into the storage. Tokens are secret because they cannot be predicted by the attacker before they are used, and thus revealed, by the clients. Our algorithms do not rely on unproven cryptographic assumptions as algorithms based on self-verifying data. They are optimally-resilient, and ensure that reads complete in two communication rounds if readers do not write into the storage, or in one communication round otherwise.

1 Introduction

We study the problem of efficiently implementing a robust storage for unauthenticated data from Byzantine storage components. Robust storage is an abstraction which supports read and write operations that are always live (wait-freedom) and read operations that never return an outdated or a spurious value (regularity). A robust algorithm uses passive storage components called *base objects* (or *objects*) that may suffer Nonresponsive-arbitrary faults [1]. Robust storage implementations for unauthenticated data are attractive because they do not incur the overhead of cryptography and they are invulnerable to cryptographic attacks. Existing unauthenticated algorithms with optimal resilience and optimal time-complexity [2–6] have a much higher (worst-case) read latency compared to algorithms storing self-verifying data, using digital signatures [7–9]. This is critical because many practical workloads are dominated by read operations. Therefore, the natural question arises if it is also possible to achieve minimal read latency without fundamentally strengthening the assumptions of the system model.

In this paper we propose two robust storage implementations for unauthenticated data with optimal resilience and optimal time complexity. The first algorithm supports unbounded readers and features constant read complexity. The second algorithm features fast reads, i.e., every read operation terminates after one round of communication with the base objects. Our algorithms circumvent the lower bounds established in [2, 4] by using *secret tokens*. A secret token (briefly token) is a value randomly selected by the client and attached to the messages sent to the base objects. The secrecy property of a token selected by a correct client is that the adversary can not generate its value before the client actually uses the token.

Secret tokens are useful because they prevent faulty base objects from simulating client operations (read or write) that have not yet been invoked but will ac-

*Research funded in part by EC Inspire, Microsoft Research and DFG GRK 1362 (TUD GKmM)

tually occur at some later point. However, tokens are weaker than signatures, because they cannot prevent a faulty base object from successfully forging a value that is never written. Consider for instance the lower bound of reading from a safe storage with optimal resilience [4]. It states that with t faulty objects, a read that does not modify the base objects takes at least $t + 1$ communication rounds before it can read a value. In each read round, a different malicious object simulates a concurrency with the same write, thereby triggering a new read round. With secret tokens, the second read round definitely reveals which value can be returned and the read terminates.

The assumption that tokens are secret can be violated with some probability. However, this probability can be arbitrarily reduced, for example, by uniformly and independently generating random tokens of k bits and by increasing the value of k. Note that in practice, assumptions in general hold only with a certain probability, e.g., the assumption that no more than t base objects fail.

Our first algorithm does not require readers to modify the base objects. As a consequence, it supports an unbounded number of possibly malicious readers. Every read completes after two communication rounds, which we show to be a tight bound. Thus, the algorithm improves on the read complexity of $t + 1$ rounds established for unauthenticated storage with optimal resilience when readers do not write [4]. Our second algorithm guarantees that every read is *fast*, i.e. it terminates after a one communication round by allowing readers to modify the base objects. The general lower bound of two rounds for reading from a robust storage with optimal resilience [2] is circumvented using tokens which are written by readers into the storage.

An alternative approach to the use of secret tokens to reduce the time-complexity is the use of cryptography, namely digital signatures [7–9]. Digital signatures generally require the generation of a secret (e.g. private) key, which entails the generation of a random bit string. Secret tokens have the following advantages over signatures: (1) no certification and key pre-distribution/sharing is needed, eliminating the need for a PKI and/or a trusted dealer; (2) no unproven assumptions such as the hardness of factorization or of discrete logarithm computation are needed; (3) the assumption of a computationally bounded adversary is not needed; (4) sampling of secret tokens can be done offline or asynchronously, without imposing an overhead in the critical execution path of the algorithm as done if signatures are used. Our algorithms are also designed to gracefully degrade their properties if the secrecy of the tokens is violated, whereas existing authenticated protocols do not discuss the system behaviors if signatures can be forged by the adversary.

1.1 Previous and Related Work

The different types of read/write storage *safe*, *regular* and *atomic* (in increasing strength) have been introduced by Lamport [10]. The study of reliable distributed storage using faulty storage components was initiated in [11] for the crash model and was extended to the Byzantine model in [7]. Since then, several Byzantine resilient distributed storage algorithms have been developed. However, only a few of them exhibit the features of robustness and optimal resilience. For instance, some implementations do not ensure wait-freedom [12] but weaker termination guarantees, such as obstruction-freedom [13] introduced in [14], or finite-writes [4]. Other works implement only weaker safe storage semantics [1,2,4,7]. Safe storage may return arbitrary values under concurrency.

Distributed storage has also been studied in a model where active base objects are able to push messages to subscribed clients and that are able to communicate with each other [15–17]. In our work we consider passive base objects which are only able to respond to client requests and do not communicate. Different works assume a stronger model where data is authenticated (called self-verifying data) [7–9], typically using digital signatures. As discussed, such solutions entail a certification and a key pre-distribution phase, they are often based on unproven assumptions, they are not secure against computationally unbounded adversaries and they entail a noticeable computation overhead.

Lower bounds have shown that protocols using the optimal number of $3t + 1$ base objects [15] require at

least *two rounds* to implement both read and write operations [2,4]. If readers are not allowed to modify the state of the base objects, the read latency is linear in the number of the base objects [4]. Our algorithms have a write complexity of two rounds, which is optimal. The read lower bounds mentioned above are circumvented using secret tokens.

The authors of [3,5] study the best case complexity of robust atomic storage. However, reads are not bounded wait-free, requiring an unbounded number of rounds in the worst case. Recent works have studied amnesic distributed storage [5,6,18,19]. Amnesic storage algorithms do not store the entire history of written values in the base objects [19].

1.2 Summary of Contributions

We now briefly summarize our contributions.

(1) We show that secret tokens can be used to reduce the read complexity of unauthenticated storage with optimal resilience from $O(n)$ rounds [4], where n is the number of base objects, to just two communication rounds. The resulting algorithm supports a possibly unbounded number of malicious readers. Our implementation is gracefully degrading. Even if the secrecy of tokens is violated, the algorithm preserves the safety properties of regular storage.

(2) We show that if readers do not write, then the cost of two communication rounds of the read operation is a lower bound for every unauthenticated storage algorithm with optimal resilience. The lower bound of [4] does not hold in a model that allows the use of secret tokens. Therefore, the time complexity of our first algorithm is optimal.

(3) Under the assumption that readers can modify the base objects, we exhibit an implementation in which every read completes after one communication round. The read lower bound of two communication rounds [2] is circumvented also in this case by using secret tokens. This algorithm is also gracefully degrading. It preserves wait-freedom and never returns a forged value. It may however return an outdated value if the secrecy of tokens is violated.

2 System Model and Definitions

We consider an asynchronous distributed system consisting of a collection of clients, interacting with a finite collection of n storage elements (called base objects). Clients are divided into a singleton writer process and a (possibly infinite) set of reader processes. When needed, the number of readers is denoted by r. Up to $t < \lfloor n/3 \rfloor$ base objects can be nonresponsive-arbitrary [1]. Any number of reader processes can suffer Byzantine failures and the writer may fail by crashing. Clients communicate with the base objects by message-passing using point-to-point reliable communication channels. Base objects do not communicate with each other and do not push messages to clients.

We assume the existence of a function GetToken used by clients that takes no arguments and outputs a value in $\{0,1\}^*$ and has the following property:
Secrecy: The adversary cannot generate the i^{th} output of function GetToken before the i^{th} invocation of GetToken.

This assumption can be implemented by sampling a value (called token) randomly, uniformly and independently from $\{0,1\}^k$. With 2^k different tokens and large k (in practice a few bytes suffice), the probability of creating a token before learning it is negligibly small.

A storage abstraction is a data structure with an initial value v_0 and two operations: WRITE(v), which stores $v \neq v_0$ in the storage and READ, which returns the value from the storage. We say that an operation *op* is *complete* in a run if the run contains a response step for *op*. For any two operations *op* and *op*′, when the response step of *op* precedes the invocation step of *op*′, we say *op precedes op*′. If neither *op* nor *op*′ precedes the other then they are *concurrent*.

A regular storage returns the value of the last complete WRITE preceding READ, or of some concurrent WRITE. A safe storage behaves like a regular one only if no WRITE overlaps the READ. Else, it may return arbitrary values.

The time-complexity (or latency) of a distributed storage algorithm is defined as the number of com-

munication round-trips from the clients to the base objects and back.

3 An Implementation Supporting Unbounded Readers

Our first algorithm uses $n \geq 3t + 1$ base objects to implement a multi-reader single-writer (MRSW) regular storage and features optimal time complexity for both operations (see Section 3.4). In the following we give a detailed description of the algorithm.

3.1 Overview

Both READ and WRITE operations take at most two rounds. In each round, the client sends a message to all objects. Each round terminates at the latest after receiving matching replies from $n-t$ correct objects. A value is written in two consecutive phases, called *pre-write* and *write* phase. In the first READ round, the reader samples a set of candidates such that the value returned after the second round is among them. In the second round, the reader collects from the objects copies of the values in the candidate set, until it finds a value to return.

The base objects maintain the array $history[0\dots]$ used by the base objects to keep track of the values written. The entry $history[ts].pw$ stores a timestamp-value pair $tsval$ of the form $\langle ts, v\rangle$ and $history[ts].w$ the pair $\langle tsval, token\rangle$. The initial token value is the empty token denoted ϵ. Variable ts stores the timestamp of the last written value. The variables of an object are collectively called *fields*.

In the pre-write phase, of WRITE(v), the writer: (1) increases its timestamp ts, (2) assigns the timestamp-value pair $\langle ts, v\rangle$ to its variable pw and (3) writes pw to $n-t$ objects' $history[ts].pw$ fields (short pw field). In the write phase, the writer (1) saves the previously written value w in the variable w_p, (2) invokes GetToken and assigns its output to variable $w.token$, (3) assigns pw to $w.tsval$ and (4) writes both w and w_p to $n-t$ correct objects' $history[ts].w$ and $history[ts-1].w$ fields (short w fields). The WRITE implementation and the algorithm of the base objects are given in Figures 1 and 2 respectively.

In the following we detail the READ implementation since it is more involved and constitutes the main focus of this paper.

3.2 READ Implementation

The full algorithm of the readers can be found in Figure 3. As mentioned earlier, READ performs in two rounds. In the first round, the reader collects from $n-t$ base objects the latest and the second latest values written w and w_p and adds them to the set of return candidates C. For this purpose the reader sends a message *rd1* to all objects (line 18) and awaits $n-t$ matching responses of type *rd1_ack* (line 20).

In the second round, the reader gathers copies of the candidate values in C from the history of pw and the w fields of the base objects until it finds a candidate it can safely return. For this purpose, in the second round (1) the reader adds the timestamps of the candidates in C to a set TS (line 21) and (2) sends a message *rd2* to all objects (line 22). Upon reception of a *rd2* message, each correct object constructs two sets PW and W, and for each timestamp $ts \in TS$ it adds to PW and W the corresponding value from the $history[ts].pw$ and $history[ts].w$ fields, if present. Finally, it sends a *rd2_ack* message containing PW and W back to the reader. When the reader receives a matching *rd2_ack* message from base object i for the first time, it records PW and W in its variables $PW[i]$ and $W[i]$, and removes all candidates from C which are incomplete (lines 23–25). If a value c is incomplete then it is missing from $n-t$ objects' history of w fields. In this case, the WRITE of c does not precede READ and thus c can be disregarded without violating regularity. The reader keeps waiting for additional *rd2_ack* messages until there is a candidate $c \in C$ such that no candidate in C has a higher timestamp (i.e., predicate highCand(c) holds) and c is stored at $t+1$ base objects in the pw or w field (i.e., predicate safe(c) holds).

Our implementation guarantees that the condition in line 26 is eventually satisfied in every READ. In the following we give a rough intuition of why this is true (the detailed proof can be found in Section 3.3).

Observe that $C \neq \emptyset$ because the second-last writ-

```
   Initialization:
1      ts ← 0; w ← ⟨⟨0, v_0⟩, ϵ⟩
   WRITE(v)
       /* Pre-write Phase */
2      inc(ts)
3      pw ← ⟨ts, v⟩
4      send pw⟨ts, pw⟩ to all objects
5      wait for reception of pw_ack⟨ts⟩ from n − t objects
       /* Write Phase */
6      w_p ← w
7      w.token ← GetToken()
8      w.tsval ← pw
9      send wr⟨ts, w, w_p⟩ to all objects
10     wait for reception of wr_ack⟨ts⟩ from n − t objects
11     return ack
```

Figure 1: Algorithm of the writer.

ten value reported by a correct object is never incomplete. Assume by contradiction that READ never completes, i.e. there is a candidate $c \in C$ such that c is never eliminated from C and c is never safe. Consider the following two cases. Case (1): c is reported in the first READ round *after* the pre-write phase of $c.tsval$ has completed. In this case, $c.tsval$ is pre-written to $t+1$ correct objects before any of them is accessed by the second READ round. Hence $t+1$ correct objects eventually report $c.tsval$ from their pw history and c becomes safe. Case (2): c is reported during the first READ round *before* the pre-write phase of $c.tsval$ has completed. Clearly, c is reported by a malicious object. By the Secrecy assumption, the token used by the adversary is different from the token which is indeed written together with $c.tsval$. Hence, no correct object reports c and c is eliminated from C. Therefore, each value either becomes safe or is removed from the set of candidates.

It is important to note that the algorithm implements a regular storage even if the Secrecy assumption does not hold. Specifically, the proof of regularity below does not rely on the inability of the adversary to guess the token.

3.3 Correctness

Lemma 1 (Regularity)**.** *The* READ *operation either returns the latest value written before* READ *is invoked or one that is written concurrently with* READ*.*

Proof. Note that if READ returns a value $c.tsval.val$, then safe(c) holds. This implies that $t+1$ objects respond with $c.tsval$ and some of these is correct. Hence, either $c.tsval$ has been written or is $\langle 0, v_0 \rangle$. We now show that READ does not return values older that the latest WRITE preceding READ.

If no WRITE completes before READ then we are done. Else, let R be a READ invocation and W = WRITE(v) be the last WRITE that completes before R is invoked. Let ts be the timestamp associated with v. We need to show that if $c.tsval.val$ is returned, then $c.tsval.ts \geq ts$.

We assume by contradiction that $c.tsval.ts < ts$. Since W precedes R, the write phase of $\langle ts, v \rangle$ completes at $t+1$ correct objects before any of them is accessed by R. Therefore, these $t+1$ objects report to the first round of R values with timestamp ts or higher. Since READ waits for $n-t$ responses, it receives a response from one of these $t+1$ correct objects. Let i denote this object and let c' be the value with the lowest timestamp of the two values reported

```
   Initialization:
1      ts ← 0; history[0].pw ← ⟨0, v_0⟩; history[0].w ← ⟨pw, ϵ⟩
2  upon reception of pw⟨ts', pw⟩ from writer
3      history[ts'].pw ← pw
4      send pw_ack⟨ts'⟩ to writer
5  upon reception of wr⟨ts', w, w_p⟩ from writer
6      if ts' > ts then ts ← ts'
7      history[ts'].w ← w; history[ts' − 1].w ← w_p
8      send wr_ack⟨ts'⟩ to writer
9  upon reception of rd1⟨tsr⟩ from reader j
10     send rd1_ack⟨tsr, history[ts].w, history[ts − 1].w⟩ to reader j
11 upon reception of rd2⟨tsr, TS⟩ from reader j
12     PW ← {history[ts'].pw : ts' ∈ TS}
13     W ← {history[ts'].w : ts' ∈ TS}
14     send rd2_ack⟨tsr, PW, W⟩ to reader j
```

Figure 2: Algorithm of the base objects.

```
   Predicates:
       safe(c) ≜ |{i ∈ Q : c.tsval ∈ PW[i] ∨ c ∈ W[i]}| ≥ t + 1
       incomplete(c) ≜ |{i ∈ Q : c ∉ W[i]}| ≥ n − t
       highCand(c) ≜ c ∈ C : (∀c' ∈ C : c.tsval.ts ≥ c'.tsval.ts)
   READ()
15     C ← TS ← Q ← ∅
16     PW[i] ← W[i] ← ∅, 1 ≤ i ≤ n
       /* Round 1 */
17     inc(tsr)
18     send rd1⟨tsr⟩ to all objects
       repeat
19         if received rd1_ack⟨tsr, w, w_p⟩ then C ← C ∪ {w, w_p}
20     until received rd1_ack⟨tsr, ∗⟩ from n − t objects
21     TS ← {c.tsval.ts : c ∈ C}
       /* Round 2 */
22     send rd2⟨tsr, TS⟩ to all objects
       repeat
23         if received rd2_ack⟨tsr, PW, W⟩ from object i then
24             Q ← Q ∪ {i}; PW[i] ← PW; W[i] ← W
25             C ← C \ {c ∈ C : incomplete(c)}
26     until (received rd2_ack⟨tsr, ∗⟩ from n − t objects) ∧
               (∃c ∈ C : safe(c) ∧ highCand(c))
27     return c.tsval.val
```

Figure 3: Algorithm of the readers.

by i such that $c'.tsval.ts \geq ts$. We show that c' is not incomplete. Assume the contrary.

By definition of incomplete, c' is missing from the history of $n-t$ objects. There are two cases to consider. If c' is reported in w, then by the choice of c', it holds that $c'.tsval = \langle ts, v \rangle$. Otherwise, c' is reported in w_p, which implies that WRITE($c'.tsval.val$) precedes the second round of R. In both cases c' has been stored in the history of w fields of $t+1$ correct objects before the second read round starts. Hence, c' is missing from the history of w fields of at most $n-t-1$ objects, a contradiction. Consequently, c' is not incomplete and is never removed from the set C of candidates. As $c'.tsval.ts \geq ts > c.tsval.ts$, c is not highCand, contradicting the assumption that R returns $c.tsval.val$. □

Lemma 2 (Wait-freedom)**.** READ *and* WRITE *operations are wait-free.*

Proof. As the WRITE operation waits for at most $n-t$ objects to respond and by assumption there are $n-t$ correct objects, it never blocks. We now show that the READ operation does not block.

We assume by contradiction that READ blocks in line 23. We consider the time after which all correct objects (at least $n-t$) have responded. We first show that $C \neq \emptyset$. Let c be the second-last value written to a correct object and reported in w_p (line 19). Observe that WRITE($c.tsval.val$) is complete before the second round of R starts. Therefore c is missing from the history of at most $n-t-1$ objects and thus, c is never eliminated from C.

We now show that for all $c \in C$, safe(c) holds. Assume by contradiction that there exists $c \in C$ and c is not safe. We distinguish the following two cases: Case (1): c is reported in the first round by some correct object. This implies that $c.tsval$ is pre-written to $t+1$ correct objects before any of them is read in the second round. Therefore, these $t+1$ correct objects respond with $c.tsval$ in PW and c is safe. Case (2): only malicious objects respond with c in the first read round. If no correct object reports c in the second read round, then c is incomplete and hence $c \notin C$. Else, if some correct object reports $c' = c$, then $c'.token = c.token$. By the Secrecy property, the malicious base objects report c only after the WRITE of c' has invoked GetToken. As the pre-write phase precedes the invocation of GetToken, $c.tsval$ is pre-written to $t+1$ correct objects before the second READ round starts and therefore c is safe. □

Theorem 1. *The Algorithm appearing in figures 1, 2 and 3 wait-free implements a MRSW regular storage.*

Proof. Follows directly from Lemma 1 and Lemma 2. □

Efficiency After having proved the correctness, we now discuss the efficiency of the algorithm. As the algorithm stores the history of written values in the base objects, the storage requirements depend on the number of write operations. Note that, if readers do not write, storing less values is an open problem [19]. The messages used are of constant size except the second read round messages which are $O(n)$. Observe that neither the storage requirements of the base objects nor the communication complexity (i.e. message size) depends on the number of readers in the system. Thus, the algorithm is scalable, supporting a possibly unbounded number of malicious clients. As announced, the time-complexity of both READs and WRITEs is of two rounds in the worst case.

In the following we show that the round-complexity of the algorithm is tight.

3.4 Optimality: Fast Reads Must Write

In this section we give a rough intuition of why the presented algorithm has optimal time-complexity. Due to space limitations, we make only a statement of the result. A detailed proof can be found in appendix A.1.

Theorem 2. There is no fast READ implementation of a single-reader single-writer ($SRSW$) safe storage from $4t$ base objects if the reader does not modify the base objects' state.

This result, together with the lower bound of two rounds for the WRITE [4], imply that our first algorithm exhibits optimal time-complexity.

Our proof derives from three indistinguishable runs. In the first run, READ is concurrent with WRITE, all correct base objects have responded and the faulty objects have crashed. In the second run, WRITE precedes READ but the faulty objects are malicious and hide the written value from the reader, simulating the concurrency of the first run. In the third run, no value is written and the malicious base objects forge the value of the writer. The reader finds itself in a situation in which it cannot distinguish between the second and the third run. If the reader returns a value, then it returns the same value in both runs, which violates safety either in the second or the third run. Else if the reader waits for more base objects, then it would block in the first run, which violates liveness.

4 An Implementation of Fast READs

The second algorithm we present in this paper also uses $n \geq 3t+1$ base objects and implements a MRSW regular storage. The main difference to the previous algorithm is that every READ operation completes after one communication round.

4.1 Overview

In each round the client (reader or writer) sends a message to all objects and waits until it has received matching replies from at most $n-t$ correct objects. Like in the previous algorithm, a value is written in two phases, a pre-write and a subsequent write phase. Unlike in the previous algorithm, in the pre-write phase, in addition to writing data, the writer also reads control data from the base objects. Readers write control data and read data written by the writer.

The base objects maintain in addition to the history of written values an array $tsrtoken[1...r]$ which is updated by the readers. The entry $tsrtoken[j]$ stores a timestamp-token pair of the form $\langle tsr, token \rangle$, where tsr is the most recent timestamp of reader j and $token$ the corresponding token value.

In the pre-write phase, of WRITE(v), the writer: (1) increases its timestamp ts, (2) stores the last pre-written value in pw_p (3) assigns the timestamp-value pair $\langle ts, v \rangle$ to its variable pw, (4) writes pw and w to $t+1$ correct objects' $history[ts].pw$ and $history[ts-1].w$ fields, (5) reads the objects' $tsrtoken[*]$ fields written by the readers and (6) for each reader j adds $tsrtoken[j]$ to the set $Tsrtokens[j]$. In the write phase, the writer (1) assigns $\langle pw, Tsrtokens \rangle$ to variable w and (2) writes both w and pw_p to $t+1$ correct objects' $history[ts].w$ and $history[ts-1].pw$ fields. The algorithm of the writer appears in Figure 4.

In the following we detail the READ implementation and the interaction with the base objects, which is slightly more involved.

4.2 READ Implementation

The full algorithm of the base objects is given in Figure 5 and that of the readers in Figure 6. As mentioned earlier, READ completes in one communication round. The reader (1) increments its timestamp tsr, (2) selects a secret token $token$ and (3) sends a message rd containing tsr and $token$ to all objects. Upon reception of rd from reader j, each correct object (1) stores $\langle tsr, token \rangle$ in $tsrtoken[j]$, (2) computes a timestamp ts_{max} such that any higher timestamped value stored has been written concurrently with READ and (3) sends a message rd_ack containing three values with timestamps $ts_{max}-1$, ts_{max} and $ts_{max}+1$ (if available) back to the reader. When the reader receives a rd_ack message from object i for the first time, it stores the value with timestamp ts_{max} in $w[i]$ and adds $w[i]$ to the set of candidates C. The other two values are added to $PW[i]$. In addition it removes all incomplete candidates from C. A candidate is incomplete when $n-t$ objects have reported candidates with lower timestamps. Observe that the choice of ts_{max} as candidate is crucial: (a) values with higher timestamps can be safely disregarded without violating regularity and (b) the value corresponding to ts_{max} is stored in $t+1$ correct objects' pw field before any of them is read. The latter property is critical because otherwise, a candidate might never become safe. The termination condition is the existence of a candidate which is both highCand and safe.

```
   Initialization:
1      Inittsrtokens[j] ← ∅, 1 ≤ j ≤ r
2      ts ← 0; pw ← ⟨0, v_0⟩; w ← ⟨pw, Inittsrtokens⟩

   WRITE(v)
       /* Pre-Write Phase */
3      Tsrtokens ← Inittsrtokens
4      inc(ts)
5      pw_p ← pw
6      pw ← ⟨ts, v⟩
7      send pw⟨ts, pw, w⟩ to all objects
       repeat
8          if received pw_ack⟨ts, tsrtoken⟩ from object i then
9              Tsrtokens[j] ← Tsrtokens[j] ∪ {tsrtoken[j]}, 1 ≤ j ≤ r
10     until received pw_ack⟨ts, ∗⟩ from n − t objects
       /* Write Phase */
11     w ← ⟨pw, Tsrtokens⟩
12     send wr⟨ts, pw_p, w⟩ to all objects
13     wait for reception of wr_ack⟨ts⟩ from n − t objects
14     return ack
```

Figure 4: Algorithm of the writer.

Our implementation guarantees that this condition is eventually satisfied in every READ. We now give an intuition of why this is true.

Recall that, for every candidate c it holds that c is pre-written to $t+1$ correct objects before any of them is read. We now explain why. The negation thereof implies that at least $t+1$ correct objects store the timestamp-token pair of READ *before* c is pre-written to them. At least one of them reports the token in the pre-write phase, such that c and all higher timestamped values are stored together with the token in the write phase. Consequently, all correct objects (at least $n-t$) report to READ only values with lower timestamps and c is eliminated from C. It is not difficult to see that if the correct base objects report the entire pw history, then every candidate would eventually become safe. Our approach simulates this behaviour, but the correct objects send at most three values, with consecutive timestamps centered around ts_{max}. The reasoning behind it is the following: if some candidate is lower than the first, then it is not highCand. Else, if it is higher than the third, then it is removed from C.

4.3 Correctness

Lemma 3 (Regularity)**.** *The* READ *operation either returns the latest value written before* READ *is invoked or one that is written concurrently with* READ.

Proof. Observe that if READ returns a value $c.val$, then safe(c) holds. This implies that $t+1$ objects respond with c and some of these is correct. Hence, either c has been written or is $\langle 0, v_0 \rangle$. We now show that READ does not return values older than the latest WRITE preceding READ.

If no WRITE completes before READ then we are done. Else, let R be a READ invocation of reader j and W = WRITE(v) be the last WRITE that completes before R is invoked. Let ts be the timestamp associated with v. We need to show that if $c.val$ is returned, then $c.ts \geq ts$.

We assume by contradiction that $c.ts < ts$.

Initialization:

1 $Inittsrtokens[j] \leftarrow \emptyset$; $tsrtoken[j] \leftarrow \langle 0, \epsilon \rangle$, $1 \leq j \leq r$

2 $history[0].pw \leftarrow \langle 0, v_0 \rangle$; $history[0].w \leftarrow \langle \langle 0, v_0 \rangle, Inittsrtokens \rangle$

3 **upon** reception of $pw\langle ts, pw, w \rangle$ from writer

4 $history[ts].pw \leftarrow pw$; $history[ts-1].w \leftarrow w$

5 send $pw\_ack\langle ts, tsrtoken \rangle$ to writer

6 **upon** reception of $wr\langle ts, pw_p, w \rangle$ from writer

7 $history[ts-1].pw \leftarrow pw_p$; $history[ts].w \leftarrow w$

8 send $wr\_ack\langle ts' \rangle$ to writer

9 **upon** reception of $rd\langle tsr, token \rangle$ from reader j

10 **if** $tsr > tsrtoken[j].tsr$ **then** $tsrtoken[j] \leftarrow \langle tsr, token \rangle$

11 $ts_{max} \leftarrow \max\{ts : tsrtoken[j] \notin history[ts].w.Tsrtokens[j]\}$

12 $w \leftarrow history[ts_{max}].w.tsval$

13 $PW \leftarrow \{history[ts_{max}-1].pw, history[ts_{max}+1].pw\}$

14 send $rd\_ack\langle tsr, PW, w \rangle$ to reader j

Figure 5: Algorithm of the base objects.

Predicates:

$\mathsf{safe}(c) \triangleq |\{i \in Q : c \in PW[i] \cup \{w[i]\}\}| \geq t+1$

$\mathsf{incomplete}(c) \triangleq |\{i \in Q : w[i].ts < c.ts\}| \geq n-t$

$\mathsf{highCand}(c) \triangleq \forall c' \in C : c.ts \geq c'.ts$

READ()

15 $C \leftarrow Q \leftarrow \emptyset$

16 $PW[i] \leftarrow \emptyset$; $w[i] \leftarrow \perp, 1 \leq i \leq n$

17 $inc(tsr)$

18 $token \leftarrow \mathsf{GetToken}()$

19 send $rd\langle tsr, token \rangle$ to all objects

repeat

20 **if** received $rd\_ack\langle tsr, PW, w \rangle$ from object i **then**

21 $Q \leftarrow Q \cup \{i\}$; $PW[i] \leftarrow PW$; $w[i] \leftarrow w$; $C \leftarrow C \cup \{w\}$

22 $C \leftarrow C \setminus \{c \in C : \mathsf{incomplete}(c)\}$

23 **until** (received $rd\_ack\langle tsr, * \rangle$ from $n-t$ objects) $\wedge$ $(\exists c \in C : \mathsf{safe}(c) \wedge \mathsf{highCand}(c))$

24 **return** $c.val$

Figure 6: Algorithm of the readers.

Let $\langle tsr, token \rangle$ be the timestamp-token pair of R. Since W precedes R, GetToken is invoked by R after W is complete. If some malicious object reports $\langle tsr, token' \rangle$ to W, then the Secrecy assumption implies that $token \neq token'$. Therefore, W does not include $\langle tsr, token \rangle$ in the set $Tsrtokens[j]$ corresponding to $\langle ts, v \rangle$. Furthermore, the write phase of $\langle ts, v \rangle$ completes at $t+1$ correct base objects before any of them is accessed by R. As $\langle tsr, token \rangle \notin Tsrtokens[j]$, these $t+1$ correct objects report values with timestamp ts or higher from their w field.

Let c' be the value with the lowest timestamp received from the w field of any of the $t+1$ objects. As R waits for at least $n-t$ objects to respond, such a value exists. We show that c' is not incomplete. Assume the contrary. By definition of incomplete, $n-t$ objects must report values with timestamps lower than $c'.ts$ from their w field. At least one of these is a correct object i among the $t+1$ updated by W. By the choice of c', $w[i].ts \geq c'.ts$. Therefore, c' is not incomplete and is never removed from the set C of candidates. As $c'.ts \geq ts > c.ts$, c is not highCand, contradicting the assumption that c is returned by R. □

Lemma 4 (Wait-freedom)**.** READ *and* WRITE *operations are wait-free.*

Proof. As the WRITE operation waits for at most $n-t$ objects to respond and by assumption there are $n-t$ correct objects, it never blocks. We now show that the READ operation does not block.

We assume by contradiction that READ blocks in line 23. We consider the time after which all correct objects (at least $n-t$) have responded. We first show that $C \neq \emptyset$. Let c be the $t+1$st highest value reported in the w field of a correct object. Clearly, c is not incomplete and thus it is not removed from C.

Let $c \in C$ be the highest value reported in the w field of a correct object. We show that (1) highCand(c) holds and (2) safe(c) holds.

Step (1): If c is not highCand, then there exists $c' \in C$ and $c'.ts > c.ts$. By the choice of c, there are $n-t$ correct objects i that report values $w[i]$ such that $w[i].ts < c'.ts$. This implies that $c' \notin C$, a contradiction.

Step (2): Observe that $t+1$ correct objects have stored c in their pw field before any of them replies to READ. Else, no correct object would reply with c in the w field (line 11). Let i be any of these correct objects. We assume by contradiction that $c \notin PW[i] \cup \{w[i]\}$. Let ts_{max} be the timestamp computed by object i in line 11. If $c.ts - 1 \leq ts_{max} \leq c.ts + 1$, then c is reported either from $PW[i]$ or $w[i]$ and we are done. Observe that, since c is pre-written to i together with the last written value (with timestamp $c.ts-1$), it holds that $ts_{max} \geq c.ts - 1$. Therefore, the only remaining case is $ts_{max} > c.ts+1$. This implies that c' exists such that $ts_{max} > c'.ts > c.ts$. Since the value with timestamp ts_{max} is pre-written to $t+1$ correct objects before they are read, c' is written to $t+1$ correct objects before they are read. Hence, c' or a higher timestamped value is not incomplete, contradicting the assumption that c is highCand. □

Theorem 3. *The Algorithm appearing in figures 4, 5 and 6 wait-free implements a MRSW regular storage.*

Proof. Follows directly from Lemma 3 and Lemma 4. □

Efficiency We now discuss the efficiency of the algorithm. Like in the previous algorithm, the storage requirements depend on the number of write operations. In addition, the base objects store up to $n \cdot r$ timestamp-token pairs together with each value written to them. Messages exchanged between the reader and the base objects are of constant size. The WRITE messages *pw_ack* (respectively *wr*) contain r (respectively $n \cdot r$) timestamp-token pairs. The time-complexity of the READ is one communication round in the worst case, which is clearly optimal. Every WRITE completes after two rounds which is also optimal [4].

5 Conclusion

The algorithms presented effectively circumvent lower bounds established for unauthenticated storage by using secret tokens. The first algorithm sup-

ports unbounded readers and features constant read complexity. The second algorithm features fast reads, i.e., every read terminates after one round of communication with the base objects. Even if the secrecy assumption of the token is violated both algorithms are gracefully degrading. The first algorithm fully preserves regularity and the second algorithm never blocks and never returns a forged value. However, the probability of property violation is negligibly small if the token space is large enough. The algorithms are secure against a computationally unbounded adversary because tokens are purely random and therefore they cannot be computed.

Both algorithms require base objects to store all the values they receive from the writers. If readers do not write, storing less values is an open problem [19]. Concerning the second algorithm, a sophisticated arbitration mechanism as shown in [6] is needed to overcome this problem, which goes beyond the scope of the paper. Although some very practical storage systems [20] rely on the same assumption this might raise issues of storage exhaustion and needs careful garbage collection.

References

[1] Jayanti, P., Chandra, T.D., Toueg, S.: Fault-tolerant wait-free shared objects. J. ACM **45**(3) (1998) 451–500

[2] Guerraoui, R., Vukolić, M.: How fast can a very robust read be? In: PODC '06: Proceedings of the twenty-fifth annual ACM symposium on Principles of distributed computing, New York, NY, USA, ACM (2006) 248–257

[3] Guerraoui, R., Levy, R.R., Vukolić, M.: Lucky read/write access to robust atomic storage. In: DSN '06: Proceedings of the International Conference on Dependable Systems and Networks (DSN'06). (2006) 125–136

[4] Abraham, I., Chockler, G., Keidar, I., Malkhi, D.: Byzantine disk paxos: optimal resilience with byzantine shared memory. Distributed Computing **18**(5) (2006) 387–408

[5] Guerraoui, R., Vukolić, M.: Refined quorum systems. In: PODC '07: Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing. (2007) 119–128

[6] Dobre, D., Majuntke, M., Suri, N.: On the time-complexity of robust and amnesic storage. In: OPODIS '08: Proceedings of the 12th International Conference on Principles of Distributed Systems, Berlin, Heidelberg, Springer-Verlag (2008) 197–216

[7] Malkhi, D., Reiter, M.: Byzantine quorum systems. Distrib. Comput. **11**(4) (1998) 203–213

[8] Cachin, C., Tessaro, S.: Optimal resilience for erasure-coded byzantine distributed storage. In: DSN '06: Proceedings of the International Conference on Dependable Systems and Networks (DSN'06), Washington, DC, USA, IEEE Computer Society (2006) 115–124

[9] Liskov, B., Rodrigues, R.: Tolerating byzantine faulty clients in a quorum system. In: ICDCS '06: Proceedings of the 26th IEEE International Conference on Distributed Computing Systems, Washington, DC, USA, IEEE Computer Society (2006) 34

[10] Lamport, L.: On interprocess communication. part II: Algorithms. Distributed Computing **1**(2) (1986) 86–101

[11] Attiya, H., Bar-Noy, A., Dolev, D.: Sharing memory robustly in message-passing systems. J. ACM **42**(1) (1995) 124–142

[12] Herlihy, M.: Wait-free synchronization. ACM Trans. Program. Lang. Syst. **13**(1) (1991) 124–149

[13] Hendricks, J., Ganger, G.R., Reiter, M.K.: Low-overhead byzantine fault-tolerant storage. In: SOSP '07: Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles, New York, NY, USA, ACM (2007) 73–86

[14] Herlihy, M., Luchangco, V., Moir, M.: Obstruction-free synchronization: Double-ended

queues as an example. In: ICDCS '03: Proceedings of the 23rd International Conference on Distributed Computing Systems, Washington, DC, USA, IEEE Computer Society (2003) 522

[15] Martin, J.P., Alvisi, L., Dahlin, M.: Minimal Byzantine Storage. In: Proceedings of the 16th International Symposium on Distributed Computing (DISC 2002), LNCS 2508. (2002) 311–325

[16] Bazzi, R.A., Ding, Y.: Non-skipping timestamps for byzantine data storage systems. In: DISC. (2004) 405–419

[17] Aiyer, A., Alvisi, L., Bazzi, R.A.: Bounded wait-free implementation of optimally resilient byzantine storage without (unproven) cryptographic assumptions. In: DISC. (2007)

[18] Abraham, I., Chockler, G., Keidar, I., Malkhi, D.: Wait-free regular storage from byzantine components. Inf. Process. Lett. **101**(2) (2007)

[19] Chockler, G., Guerraoui, R., Keidar, I.: Amnesic Distributed Storage. In: Proceedings of the 21st International Symposium on Distributed Computing (DISC'07). (2007)

[20] Goodson, G.R., Wylie, J.J., Ganger, G.R., Reiter, M.K.: Efficient byzantine-tolerant erasure-coded storage. In: DSN '04: Proceedings of the 2004 International Conference on Dependable Systems and Networks (DSN'04), Washington, DC, USA, IEEE Computer Society (2004) 135

A Appendix

A.1 Fast Reads Must Write

In this section we prove that fast read implementations require the reader to modify the state of the base objects. The proof uses similar arguments as the lower bound proof in [2] and is illustrated in Figure 7. We partition the set of base objects into four distinct subsets T_1, T_2, T_3, T_4 each of size t. The initial state of every correct base object is denoted as σ_0. A round rnd of an operation is depicted by a line of rectangles. A rectangle in a line corresponding to some round rnd of operation op means that all base objects in the corresponding block have received the message from the client in round rnd of operation op and have sent a reply message.

Theorem 4. There is no fast READ implementation I of a $SRSW$ safe storage from $4t$ base objects if the reader does not modify the base objects' state.

Proof. To exhibit a contradiction, we construct a run of the safe implementation I that violates safety. We exhibit a run in which some READ returns a value that was never written.

- Let R_1 be the run in which all objects are correct except T_4 that crashes at a later point. Furthermore, let rd be the READ operation by reader r. After T_1 sends rd_ack to r, object T_1 is still in the initial state σ_0. Before rd reads from another object, a WRITE operation wr is invoked by the correct writer to write a value $v \neq v_0$ to the storage. By the assumption that I is wait-free, wr completes in R_1, say at time t_1 after invoking a finite number m of rounds. Due to asynchrony all messages sent by the writer to T_3 during w remain in transit. We refer to the state of base object T_2 at time t_1 as σ_1. At some time after t_1, object T_4 crashes. Due to asynchrony, all messages exchanged between r and T_2 and T_3 are delayed until after t_1. By our assumption on wait-freedom of I, r completes after receiving $read\_ack$ messages from correct objects T_1, T_2, and T_3 and returns some value v_r skipping T_4.
- Let R_2 be the run similar to run R_1, except that in R_2: (1) READ rd is invoked only after wr completes (after t_1) and (2) object T_1 is malicious and at t_1 before replying to r, forges its state to σ_0, the initial state of correct objects. Other messages are delivered as in R_1. Note that wr cannot distinguish run R_2 from R_1 and therefore wr completes in R_2 at t_1. Note also that, rd is invoked after wr completes, so safety implies that rd must return v. However, note that

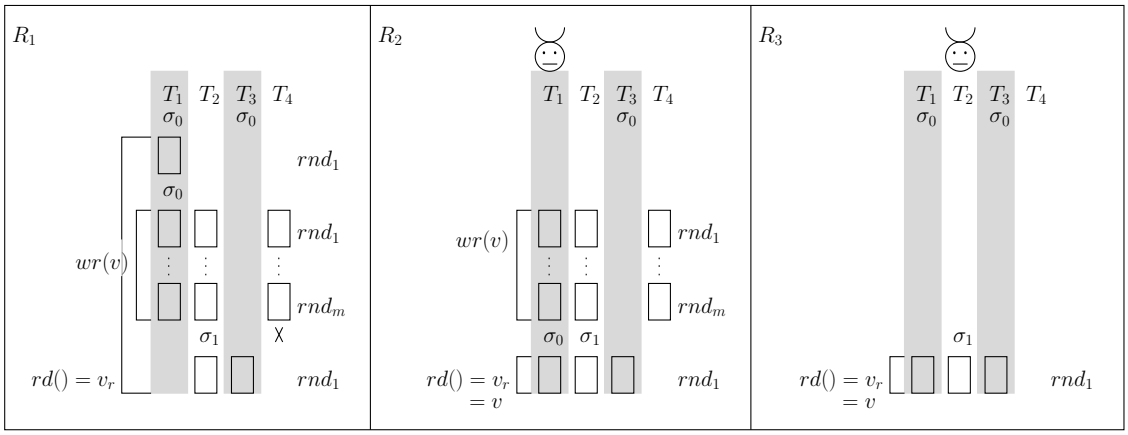

Figure 7: Illustration of the proof of Theorem 4

in R_1 and R_2 the reader receives in rd the identical messages and, since the processes do not have access to a global clock, r cannot distinguish R_2 from R_1. Therefore, in R_1 and R_2, rd returns the same value, i.e. by safety v_r must equal v.

- Finally, we consider the run R_3 in which wr is never invoked, but T_2 is malicious and forges its state to σ_1 at the beginning of the run. READ rd is invoked in R_3 as in R_2. Since, upon receiving *read_ack* messages from T_1, T_2, and T_3, the reader receives identical information as in run R_2, the reader cannot distinguish R_2 from R_3, and rd completes in R_3 and returns $v_r = v$. However, by safety, in R_3, rd must return v_0. Since $v \neq v_0$, safety is violated in R_3.

□