On the Time-Complexity of Robust and Amnesic Storage[⋆]

Dan Dobre, Matthias Majuntke and Neeraj Suri

TU Darmstadt, Hochschulstr. 10, 64289 Darmstadt, Germany
{dan,majuntke,suri}@cs.tu-darmstadt.de

Abstract. We consider wait-free implementations of a regular read/write register for unauthenticated data using a collection of $3t + k$ base objects, t of which can be subject to Byzantine failures. We focus on *amnesic* algorithms that store only a limited number of values in the base objects. In contrast, non-amnesic algorithms store an unbounded number of values, which can eventually lead to problems of space exhaustion. Lower bounds on the time-complexity of read and write operations are currently met only by non-amnesic algorithms. In this paper, we show for the first time that amnesic algorithms can also meet these lower bounds. We do this by giving two amnesic constructions: for $k = 1$, we show that the lower bound of *two* communication rounds is also sufficient for *every* read operation to complete and for $k = t + 1$ we show that the lower bound of *one* round is also sufficient for *every* operation to complete.

Keywords: distributed storage, Byzantine failures, wait-free algorithms

1 Introduction

Motivated by recent advances in the Storage-Area Network (SAN) technology, and also by the availability of cheap commodity disks, distributed storage has become a popular method to provide increased storage space, high availability and disaster tolerance. We address the problem of implementing a reliable read/write distributed storage service from unreliable storage units (e.g. disks), a threshold of which might fail in a malicious manner. Fault-tolerant access to replicated remote data can easily become a performance bottleneck, especially for data-centric applications usually requiring frequent data access. Therefore, minimizing the time-complexity of read and write operations is essential. In this paper, we show how optimal time-complexity can be achieved using algorithms that are also space-efficient.

An essential building block of a distributed storage system is the abstraction of a read/write register, which provides two primitives: a *write* operation, which writes a value into the register, and a *read* operation which returns a value previously written [1]. Much recent work, and this paper as well, focuses on *regular*

⋆ Research funded in part by DFG GRK 1362 (TUD GKmM), EC NoE ReSIST and Microsoft Research via the European PhD Fellowship.

registers where read operations never return outdated values. A regular register is deemed to return the last value written before the read was invoked, or one written concurrently with the read (see [1] for a formal definition). Regular registers are attractive because even under concurrency, they never return spurious values as sometimes done by the weaker class of *safe* registers [1]. Furthermore, they can be used, for instance, together with a failure detector to implement consensus [2].

The abstraction of a reliable storage is typically built by replicating the data over multiple unreliable distributed storage units called *base objects*. These can range from simple (low-level) read/write registers to more powerful base objects like *active disks* [3] that can perform some more sophisticated operations (e.g. an atomic read-modify-write). Taken to the extreme, base objects can also be implemented by full-fledged servers that execute more complex protocols and actively push data [4]. We consider Byzantine-fault tolerant register constructions where a threshold $t < n/3$ of the base objects can fail by being *non-responsive* or by returning *arbitrary* values, a failure model called NR-arbitrary [5]. Furthermore, we consider *wait-free* implementations where concurrent access to the base objects and client failures must not hamper the liveness of the algorithm. Wait-freedom is the strongest possible liveness property, stating that each client completes its operations independent of the progress and activity of other clients [6]. Algorithms that wait-free implement a regular register from Byzantine components are called *robust* [7]. An implementation of a reliable register requires the (client) processes accessing the register via a high-level operation to invoke multiple low-level operations on the base objects. In a distributed setting, each invocation of a low-level operation results in one *round* of communication from the client to the base object and back. The number of rounds needed to complete the high-level operation is used as a measure for the time-complexity of the algorithm.

Robust algorithms are particularly difficult to design when the base objects store only a limited number of written values. Algorithms that satisfy this property are called *amnesic*. With amnesic algorithms, values previously stored are not permanently kept in storage but are eventually erased by a sequence of values written after them. Amnesic algorithms eliminate the problem of space exhaustion raised by (existing) non-amnesic algorithms, which take the approach of storing the entire version history. Therefore, the amnesic property captures an important aspect of the space requirements of a distributed storage implementation. The notion of amnesic storage was introduced in [7] and defined in terms of *write-reachable configurations*. A configuration captures the state of the correct base objects. Starting from an initial configuration, any low-level read/write operation (i.e., one changing the state of a base object) leads the system to a new configuration. A configuration C' is write-reachable from a configuration C when there is a sequence consisting only of (high-level) write operations that starting from C, leads the system to C'. Intuitively, a storage algorithm is amnesic if, except a finite number of configurations, all configurations reached by the algorithm are eventually *erased* by a sufficient number of values written after them. Erasing a configuration C', which itself was obtained from a configuration C, means to reach a configuration C'' that could have been obtained directly from C without going through C'. This means that once in C'', the system cannot tell whether it has ever been in configuration C'. For instace, an algorithm that stores the entire history of written values in the base objects is not amnesic. In contrast, an algorithm that stores in the base objects only the last l written values is amnesic because after writing the $l + 1^{\mathrm{st}}$ value, the algorithm cannot recall the first written value anymore.

1.1 Previous and Related Work

Despite the importance of *amnesic and robust* distributed storage, most implementations to date are either *not* robust or *not* amnesic. While some relax wait-freedom and provide weaker termination guarantees instead [2, 8], others relax consistency and implement only the weaker safe semantics [2,5,9,10]. Generally, when it comes to robustly accessing (unauthenticated) data, most algorithms store an unlimited number of values in the base objects [10–12]. Also in systems where base objects push messages to subscribed clients [4, 13, 14], the servers store every update until the corresponding message has been received by every non-faulty subscriber. Therefore, when the system is asynchronous, the servers might store an unbounded number of updates. A different approach is to assume a stronger model where data is self-verifying [9, 15, 16], typically based on digital signatures. For unauthenticated data, the only existing robust and amnesic storage algorithms [17, 18] do not achieve the same time-complexity as non-amnesic ones. Time-complexity lower bounds have shown that protocols using the optimal number of $3t + 1$ base objects [4] require at least *two rounds* to implement both read/write operations [2, 10]. So far these bounds are met only by non-amnesic algorithms [12]. In fact, the only robust and amnesic algorithm with optimal resilience [17] requires an unbounded number of read rounds in the worst case. For the $4t + 1$ case, the trivial lower bound of *one round* for both operations is *not* reached by the only other existing amnesic implementation [18] that albeit elegant, requires at least *three* rounds for reading and *two* for writing.

1.2 Paper Contributions

Current state of the art protocols leave the following question open: *Do amnesic algorithms inherently have a non-optimal time complexity?* This paper addresses this question and shows, for the first time, that amnesic algorithms can achieve optimal time complexity in both the $3t + 1$ and $4t + 1$ cases. Justified by the impossibility of amnesic and robust register constructions when readers do not write [7], one of the key principles shared by our algorithms is having the readers change the base objects' state. The developed algorithms are based on a novel concurrency detection mechanism and a helping procedure, by which a writer detects overlapping reads and helps them to complete. Specifically, the paper makes the following two main contributions:

– A first algorithm, termed DMS, which uses $4t + 1$ base objects, described in Section 3. With DMS, *every* (high-level) read and write operation is *fast*, i.e.,

it completes after only *one round* of communication with the base objects. This is the first robust and amnesic register construction (for unauthenticated data) with optimal time-complexity.
– A second algorithm, termed DMS3, which uses the optimal number of $3t+1$ base objects, presented in Section 4. With DMS3, every (high-level) read operation completes after only *two rounds*, while *write* operations complete after *three rounds*. This is the first amnesic and robust register construction (for unauthenticated data) with optimal read complexity. Note also that, compared to the optimal write complexity, it needs only one additional communication round.

Table 1 below summarizes our contributions and compares DMS and DMS3 with recent distributed storage solutions for unauthenticated data.

Table 1. Distributed storage for unauthenticated data

Protocol	Resilience	Worst-Case Time-complexity		Amnesic	Robust
		Read	Write		
Abraham et al. [18]	$4t+1$	3	2	$\surd$	$\surd$
DMS	$4t+1$	1	1	$\surd$	$\surd$
Guerraoui and Vukolić [10]	$3t+1$	2	2	$\times$	$\surd$
Byzantine Disk Paxos [2]	$3t+1$	$t+1$	2	$\surd$	$\times$
Guerraoui et al. [17]	$3t+1$	unbounded	3	$\surd$	$\surd$
DMS3	$3t+1$	2	3	$\surd$	$\surd$

2 System Model and Preliminaries

2.1 System Model

We consider an asynchronous shared memory system consisting of a collection of processes interacting with a finite collection of n base objects. Up to t out of n base objects can suffer NR-arbitrary failures [5] and any number of processes may fail by crashing. Each object implements one or more *registers*. A register is an object type with value domains *Val*, an initial value v_0 and two invocations: *read*, whose response is $v \in \mathit{Vals}$ and $\mathit{write}(v)$, $v \in \mathit{Vals}$, whose response is *ack*. A read/write register is single-reader single-writer (SRSW) if only one process can read it and only one can write to it; a register is multi-reader single-writer (MRSW) if multiple processes can read it. Sometimes processes need to perform two operations on the same base object, a write (of a register) followed by a read (of a different register). To reduce the number of rounds, we collapse consecutive write/read operations accessing the same base object to a single low-level operation called *write&read*. The *write&read* operation can be implemented in a single round, for instance using active disks [3] as base objects[1].

[1] Note that since *write&read* is not an atomic operation, it can be implemented from simple read/write registers and thus the model is not strengthened.

2.2 Preliminaries

In order to distinguish between the target register's interface and that of the base registers, throughout the paper we denote the high-level read (resp. write) operation as READ (resp. WRITE). Each of the developed protocols uses an underlying layer that invokes operations on different base objects in separate threads in parallel. We use the notation from [2] and write **invoke** $write(X_i,v)$ (resp. **invoke** $x[i] \leftarrow read(X_i)$) to denote that a write(v) operation on register X_i (resp. a read of register X_i whose response will be stored in a local variable $x[i]$) is invoked in a separate thread by the underlying layer. The notation **invoke** $x[i] \leftarrow write\&read(\langle Y_i, v\rangle, X_i)$ denotes the invocation of an operation *write&read* on base object i, consisting of a write(v) on register Y_i followed by a read of register X_i (whose response will be stored in $x[i]$).

As base objects may be non-responsive, high-level operations can return while there are still pending invocations to the base objects. The underlying layer keeps track of which invocations are pending to ensure well-formedness, i.e., that a process does not invoke an operation on a base object while invocations of the same process and on the same base object are pending. Instead, the operation is denoted enabled. If an operation is enabled when a pending one responds, the response is discarded and the enabled operation is invoked. See e.g. [2] for a detailed implementation of such layers.

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_1 and op_2, when the response step of op_1 precedes the invocation step of op_2, we say op_1 *precedes* op_2. If neither $op1$ nor $op2$ precedes the other then the two operations are said to be *concurrent*.

In order to better convey the insight behind the protocols, we simplify the presentation in two ways. We introduce a shared object termed *safe counter* and describe both algorithms in terms of this abstraction. Although easy to follow, the resulting implementations require more rounds than the optimal number. Thus, for each of the protocols we explain how with small changes these rather didactic versions can be "condensed" to achieve the announced time-complexity. The full details of the optimizations can be found in our publicly available technical report [19]. Secondly, for presentation simplicity we implement a SRSW register. Conceptually, a MRSW register for m readers can be constructed using m copies of this register, one for each reader. In a distributed storage setting, the writer accesses all m copies in parallel, whereas the reader accesses a single copy. It is worth noting that this approach is heavy and that in practice, cheaper solutions are needed to reduce the communication complexity and the amount of memory needed in the base objects.

We now introduce the *safe counter* abstraction used in our algorithms. A safe counter has two *wait-free operations* INC and GET. INC modifies the counter by incrementing its value (initially 0) and returns the new value. Specifically, the k^{th} INC operation denoted INC^k returns k. GET returns the current value of the counter without modifying it. The counter provides the following guarantees:

Validity: If GET returns k then GET does not precede INC^k.

Safety: If INC^k precedes GET and for all $l > k$ GET precedes INC^l, then GET returns k.

Note that under concurrency, a safe counter might return an outdated value, but never a forged value. In the absence of concurrency, the newest value is returned.

We now explain the intuition behind our algorithms. Both algorithms use the safe counter introduced above to arbitrate between writer and reader. During each READ (resp. WRITE) operation, the reader (resp. writer) executes INC to advance the counter (resp. GET to read the counter). The values returned by the counter's operations are termed *views*. By incrementing its current view, a READ announces its intent to read from the base objects. A subsequent invocation of GET by the writer returns the updated view. When the writer detects a concurrent READ, indicated by a view change, it *freezes* the most recent value previously written. Freezing a value v means that v may be overwritten *only if* the READ operation that attempts to read v has completed. We note that the READ operation that caused a value v to be frozen does not violate regularity by returning v because all newer values were written concurrently with the READ. However, READs must not return old values previously frozen. This is necessary to ensure regularity and it is done by freezing a value v together with the view of the READ due to which v is frozen. A READ whose view is higher than the one associated with v knows that it must pick a newer value. A READ operation completes when it finds a value v to return such that (a) v is reported by a correct base object and (b) v is not older than the latest value written before the READ is invoked.

3 A Fast Robust and Amnesic Algorithm

We start by describing an initial version of protocol DMS that uses the safe counter abstraction. It is worth noting that the algorithm requires more rounds than the optimum, but it conveys the main idea. Next, we explain the changes applied to DMS to obtain an algorithm with optimal time-complexity.

3.1 Protocol Description

We present a robust and amnesic SRSW register construction using a safe counter and $4t + 1$ regular base registers, out of which t can incur NR-arbitrary failures. Figure 1 illustrates a simple construction of the safe counter used. The description of the counter is omitted for the sake of brevity. The shared objects used by DMS are detailed in Figure 2 and the algorithm appears in Figure 3.

The WRITE performs in two phases, (1) a write phase where it first writes a timestamp-value pair to $n - t$ registers and (2) a subsequent read phase, where it executes GET to read the current view. In case a view change occurs between two successive WRITEs, the value of the first WRITE is frozen. Recall that once frozen, a value is not erased before the next view change. Similarly, the READ consists of (1) a write phase, where it first executes INC to increment the current

Fig. 1. Safe counter from $4t + 1$ safe registers $Y_i \in \mathit{Integers}$.

```
Predicates:
    $\mathsf{safe}(c) \triangleq$
    $|\{i : c' \in y[i] \wedge c' \geq c\}| \geq t + 1$

GET()
    for $1 \leq i \leq n$ do $y[i] \leftarrow \perp$
    for $1 \leq i \leq n$ do
        invoke $y[i] \leftarrow \mathit{read}(Y_i)$
    wait for $n - t$ responses
    return $\max\{c \in \mathit{Integers} : \mathsf{safe}(c)\}$

Local variables:
    $y[1 \ldots n] \in \mathit{Integers}$
    $k \in \mathit{Integers}$, initially 0

INC()
    $k \leftarrow k + 1$
    for $1 \leq i \leq n$ do
        invoke $\mathit{write}(Y_i, k)$
    wait for $n - t$ responses
    return $k$
```

view and (2) a subsequent read phase, where it reads at least $n - t$ registers. To ensure that READ never returns a corrupted value, the returned value must be read from $t+1$ registers, a condition captured by the predicate safe. Moreover, to ensure regularity, READ must not return old values written before the last WRITE preceding the READ. This condition is captured by the predicate highestCand.

We now give a more detailed description of the algorithm. As depicted in Figure 2, each base register consists of three value fields *current*, *prev* and *frozen* holding timestamp-value pairs, and an integer field *view*. The writer holds a variable x of the same type and uses x to overwrite the base registers. Each WRITE operation saves the timestamp-value pair previously written in *x.prev*. Then, it chooses an increasing timestamp, stores the value together with the timestamp in *x.curr* and overwrites $n - t$ registers with x. Subsequently, the writer executes GET. If the view returned by GET is higher than the current view (indicating a concurrent READ), then $x.\mathit{view}$ is updated and the most recent value previously written is frozen, i.e., the content of *x.prev* is stored in *x.frozen* (line 14, Figure 3). Finally, WRITE returns *ack* and completes. It is important to note that the algorithm is amnesic because each correct base object stores at most three values (*curr*, *prev* and *frozen*).

The READ first executes INC to increment the current view, and then it reads at least $n-t$ registers into the array $x[1...n]$, where element i stores the content of register X_i. If necessary, it waits for additional responses until there is a *candidate* for returning, i.e., a read timestamp-value pair that satisfies both predicates safe and highestCand. A timestamp-value pair c is safe when it appears in some field *curr*, *prev* or *frozen* of $t+1$ elements of x, ensuring that c was reported by at least one correct register. Enforcing regularity is more subtle. Simply waiting until the highest timestamped value read becomes safe might violate liveness because it may be reported by a faulty register. To solve this problem, we introduce the predicate highestCand. A value c is highestCand when $2t+1$ base registers report values that were written *not after* c, which implies that newer values are missing from $t + 1$ correct registers. As any complete WRITE skips at most t correct registers, all values newer than c were written *not* before READ is invoked and consequently, they can be discarded from the set of possible return candidates.

We now explain with help of Figure 4 why READs are wait-free. We consider the critical situation when multiple WRITEs are concurrent with a READ.

Fig. 2. Shared objects used by DMS.

Types:

$TSVals \triangleq Integers \times Vals$, with selectors ts and val

Shared objects:

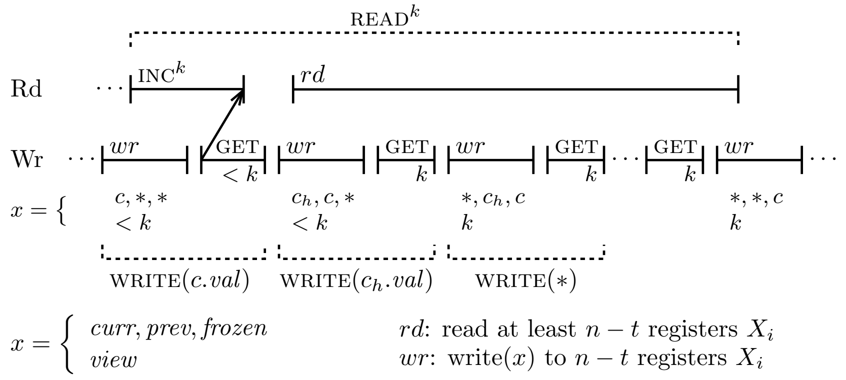
- regular registers $X_i \in TSVals^3 \times Integers$ with selectors $curr$, $prev$, $frozen$ and $view$, initially $\langle\langle 0, v_0\rangle, \langle 0, v_0\rangle, \langle 0, v_0\rangle, 0\rangle$
- safe counter object $Y \in Integers$, initially $Y = 0$

Fig. 3. Robust and amnesic storage algorithm DMS $(4t + 1)$

```
Predicates (reader):
  readFrom($c$, $i$) $\triangleq$ ($c = x[i].curr \wedge x[i].view < view$) $\vee$
                              ($c = x[i].frozen \wedge x[i].view = view$)
  safe($c$) $\triangleq |\{i : c \in \{x[i].curr, x[i].prev, x[i].frozen\}\}| \geq t + 1$
  highestCand($c$) $\triangleq |\{i :$ readFrom($c', i$) $\wedge\ c'.ts \leq c.ts\}| \geq 2t + 1$

Local variables (reader):
  $view \in Integers$, initially 0
  $x[1 \ldots n] \in TSVals^3 \times Integers$

READ()
1   for $1 \leq i \leq n$ do  $x[i] \leftarrow \perp$
2   $view \leftarrow$ INC($Y$)
3   for $1 \leq i \leq n$ do invoke $x[i] \leftarrow read(X_i)$
4   wait until $n - t$ responded $\wedge\ \exists c \in TSVals$: safe($c$) $\wedge$ highestCand($c$)
5   return $c.val$

Local variables (writer):
  $newView$, $ts \in Integers$, initially 0
  $x \in TSVals^3 \times Integers$, initially $\langle\langle 0, v_0\rangle, \langle 0, v_0\rangle, \langle 0, v_0\rangle, 0\rangle$

WRITE($v$)
6   $ts \leftarrow ts+1$
7   $x.prev \leftarrow x.curr$
8   $x.curr \leftarrow \langle ts, v\rangle$
9   for $1 \leq i \leq n$ do invoke $write(X_i, x)$
10  wait for $n - t$ responses
11  $newView \leftarrow$ GET($Y$)
12  if $newView > x.view$ then
13     $x.view \leftarrow newView$
14     $x.frozen \leftarrow x.prev$
15  return $ack$
```

Specifically, we consider the k^{th} READ (henceforth READ^k), whose INC results in k (henceforth INC^k), and the *last* WRITE that still reads a view lower than k, i.e., the corresponding GET returns a view lower than k. Note that by the safety property of the counter, INC^k does not precede GET and thus c is stored in $2t+1$ correct registers *before* any of them is read. A key aspect of the algorithm is to ensure that no matter how many WRITEs are subsequently invoked, c never disappears from all fields of those $2t+1$ correct registers, as long as READ^k is still in progress. Essentially this holds because the subsequent WRITE re-writes c to all registers and it also freezes c to ensure that future WRITEs do the same. In this process, c migrates from *curr* to *prev* and from *prev* to *frozen* where it stays until the next view change. Therefore, c eventually becomes safe. But what if c is not highestCand? In this situation, at least $t+1$ correct registers report timestamp-value pairs higher than c. We note that if any of them had stored c in its *frozen* field, then it would report c. This implies that none of these registers has stored c in its *frozen* field and thus, also none of these registers has stored a timestamp-value pair higher than c_h in its *curr* field. Therefore, c_h is reported by $t+1$ correct registers, and hence it is safe. Note that c_h is also highestCand because only faulty registers report values with higher timestamps.

Fig. 4. Correctness argument of the READ operation in DMS



We now explain how the fast algorithm is derived from DMS. The principle underlying the optimization is to condense one round of write to the base objects and a subsequent round of read of the base objects into a single round of *write&read*. For this purpose we disregard the safe counter abstraction and directly weave INC and GET (Fig. 1) into READ and WRITE (Fig. 3) respectively. As a result, the reader advances the view *and* reads the base registers in one round. Likewise, the writer stores a value in the base registers *and* reads the view in a single round. The reader code (Fig. 3) is modified as follows: variable *view* is incremented locally, and line 3 is replaced with the statement **for** $1 \leq i \leq n$ **do invoke** $x[i] \leftarrow$ *write&read*$(\langle Y_i, \textit{view}\rangle, X_i)$. Similarly, in the writer code (Fig. 3), line 9 is replaced with the statement **for** $1 \leq i \leq n$ **do invoke** $y[i] \leftarrow \mathit{write\&read}(\langle X_i, x\rangle, Y_i)$. Additionally in line 11, instead of executing GET, the writer picks the $t+1^{\text{st}}$ highest element of y.

We now informally argue that the optimization is correctness preserving. As in the above example, we consider READ^k and the last WRITE that reads a view lower than k. Recall that the WRITE operation stores c in $2t+1$ correct base objects and each of them responds with the current view it has stored. The writer then picks the $t+1^{\text{st}}$ highest view reported. We argue that $t+1$ correct base objects have stored c before any of them respond to READ^k. This would imply that c is safe. As the WRITE operation reads a view lower than k, out of the $2t+1$ correct base objects accessed by it, at most t report k. Thus, the remaining $t+1$ objects are accessed by READ^k only after c was written to them. Applying the above arguments, it is not difficult to see that c is never erased from $t+1$ correct registers before READ^k completes, and thus it eventually becomes safe. Regarding regularity, again, arguments similar to above can be used. A formal proof of the optimized algorithm can be found in the full paper [19]. The remainder of this section is concerned with the correctness of DMS.

3.2 Protocol Correctness

Lemma 1 (Regularity). *Algorithm DMS in Figure 3 implements a regular register.*

Proof. We show that the READ operation always returns the value of the latest WRITE preceding the READ, or a newer written value. Suppose that $c.val$ is the value returned by READ^k. We assume by contradiction that there exists a value $c_h.val$ such that $c_h.ts > c.ts$ and WRITE$(c_h.val)$ precedes READ^k. As WRITE$(c_h.val)$ is complete, $n-2t$ correct registers have stored c_h or a higher timestamp-value pair before any of them is read. The fact that $c.val$ is returned implies that c is highestCand. Thus, there are at least $2t+1$ registers X_i and values c' with timestamp $c'.ts \leq c.ts$ such that readFrom(c',i) is true. Note that one of them is a correct register X_i updated with c_h. As values are written with monotonically increasing timestamps, by definition of readFrom, necessarily c' is read from $x[i].frozen$ and $x[i].view = k$. However, because the counter is valid, the first time a WRITE operation reads view k is only after the WRITE of $c_h.val$. Thus, in view k only timestamp-value pairs c_h or higher are frozen, a contradiction. □

Lemma 2 (Wait-freedom). *Algorithm DMS in Figure 3 implements wait-free* READ *and* WRITE *operations.*

Proof. The WRITE operation is nonblocking because it never waits for more than $n-t$ responses. Showing that READs are also live is more subtle. To derive a contradiction, we assume that READ^k blocks at line 4 and show that there exists a candidate for returning. We consider the time after which all correct base objects (at least $3t+1$) have responded. We choose c as the $2t+1^{\text{st}}$ lowest timestamp-value pair readFrom a correct register. Note that c is highestCand by construction because values with timestamps $\leq c.ts$ are readFrom $2t+1$ correct registers (set L). Also, we note that values with timestamps $\geq c.ts$ are readFrom $t+1$ correct registers (set R). In the following, we distinguish the cases where the WRITE of $c.val$ reads a view equal to k (case 1), or lower than k (case 2). Note that by the validity of the counter, only views $\leq k$ are returned. Case 1 implies that (a) only timestamp-value pairs lower than c are frozen, and (b) c is the highest timestamp-value pair readFrom the *curr* field of a correct register. Together (a) and (b) imply that c is the highest timestamp-value pair readFrom a correct register. Thus, for all registers $X_i \in R$ $(\geq t+1)$, readFrom(c',i) implies that $c' = c$ and hence, c is safe. We now consider case 2 where WRITE$(c.val)$ reads a view lower than k. This implies that c or a higher timestamp-value pair is frozen in view k. If $t+1$ registers in L were updated with c before they are read, then they would report c either from their *curr* or their *frozen* field, and clearly c would be safe. Therefore, c is missing from $t+1$ correct registers. Thus, WRITE$(c.val)$'s write phase (lines 9–10) does not precede READ^k's read phase (lines 3–4). By the transitivity of the precedence relation, INC^k (line 2) precedes GET (line 11). By the safety of the counter, WRITE$(c.val)$ reads view k, a contradiction. □

Theorem 1 (Robustness). *The algorithm in Figure 3 wait-free implements a regular register.*

Proof. Immediately follows from Lemma 1 and 2.

4 A Robust and Amnesic Algorithm with Optimal READ-Complexity and Resilience

Similar to the previous section, we describe an initial version of DMS3 that uses a safe counter. The algorithm requires more rounds than the optimum but it is easier to understand because most of its complexity is hidden in the counter implementation. Then, we overview the changes necessary to obtain the optimal algorithm. The full details of the optimized DMS3 such as the pseudocode and proofs can be found in our technical report [19]. We proceed in a bottom-up fashion and describe the counter implementation first.

4.1 A Safe Counter with Optimal Resilience

We present a safe counter with operations INC and GET using $3t+1$ base objects $i \in \{1 \ldots n\}$, where t base objects can be subject to NR-arbitrary failures. The types and shared objects used by the counter are depicted in Figure 5 and the algorithm appears in Figure 6. Each base object i implements two regular registers: a register T_i holding a timestamp written by GET and read by INC, and a second register Y_i consisting of two fields *pw* and *w*, modified by INC and read by GET. While the *pw* field stores only the counter value, the *w* field stores the counter value together with a *high-resolution timestamp* [20]. A high-resolution timestamp is a timestamp-array with n entries, one for each base object.

Fig. 5. Shared objects used by the safe counter $(3t+1)$

Additional Types:

TSs $\triangleq$ *Integers* array of size n, *Integers*$[n]$

TSsInt $\triangleq$ *TSs* $\times$ *Integers* with selectors *hrts* (high-resolution timestamp) and *cnt*

Shared objects:

- regular registers $Y_i \in$ *Integers* $\times$ *TSsInt* with selectors *pw* and *w*, initially $Y_i = \langle 0, \langle [0, \ldots, 0], 0\rangle\rangle$
- regular registers $T_i \in$ *Integers*, initially 0

The GET operation performs in two phases. The first phase reads from the base objects until $n-t$ registers Y_i have responded and all responses are *non-conflicting*. This condition is captured by the predicate conflict. When two base objects i and j are in conflict, then at least one of them is malicious. In this situation, the GET operation can wait for more than $n-t$ responses without blocking, effectively filtering out responses from malicious base objects. Next, the GET operation uses the responses to build a candidate set from values appearing in the w field of Y_i. In the second phase, the GET operation chooses an increasing timestamp ts and overwrites $n-t$ registers T_i with ts; at the same time it re-reads the registers Y_i until $n-t$ of them have responded and there exists a *candidate* to return. This condition is captured by the predicates safe and highCand. If no candidate can be returned (because of overlapping INC operations), GET returns the initial counter value 0.

Similarly, the INC operation performs in two phases, a pre-write and a write phase. The pre-write phase accesses $n-t$ base objects i, overwriting the pw field of Y_i with an increasing counter value and reading the individual timestamps stored in T_i into a single high-resolution timestamp. Subsequently, in the write phase, INC stores the counter value together with the high-resolution timestamp in the w field of $n-t$ registers Y_i and returns.

We now show that the algorithm in Figure 6 wait-free implements a safe counter. We do this by showing that the two following properties are satisfied:

Validity: If GET returns k then GET does not precede INC^k.
Safety: If INC^k precedes GET and for all $l > k$ GET precedes INC^l, then GET returns k.

Lemma 3 (Validity). *The counter object implemented in Figure 6 is valid.*

Proof. If the initial value is returned then we are done. Else only a value $c.cnt = k$ is returned such that c is safe. This implies that $t+1$ base objects report values k or higher either from their pw or w fields. As not all of them are faulty, there exists a correct object Y_i and a value $l \geq k$ such that l was indeed written to Y_i. As INC^k precedes INC^l (or it is the same operation) and GET does not precede INC^l, it follows that GET does not precede INC^k. $\square$

Lemma 4 (Safety). *The counter object implemented in Figure 6 is safe.*

Fig. 6. Safe counter algorithm $(3t+1)$

```
Local variables (INC):
    $y \in Integers \times TSsInt$, initially $\langle 0, \langle [0,\ldots,0], 0\rangle\rangle$
    $cnt \in Integers$, initially 0   //counter value
    $hrts[1\ldots n] \in Integers$, initially $[0,\ldots,0]$   //high-resolution timestamp
INC()
1     $cnt \leftarrow cnt + 1$
2     $y.pw \leftarrow cnt$
3     for $1 \le i \le n$ do invoke $hrts[i] \leftarrow write\&read(\langle Y_i, y\rangle, T_i)$
4     wait for $n-t$ responses
5     $y.w.hrts \leftarrow hrts$
6     $y.w.cnt \leftarrow cnt$
7     for $1 \le i \le n$ do invoke $write(Y_i, y)$
8     wait for $n-t$ responses
9     return $ack$

Predicates (GET):
    $\mathsf{conflict}(i,j) \triangleq y[i].w.hrts[j] \ge ts$
    $\mathsf{safe}(c) \triangleq |\{i : \max\{PW[i]\} \ge c.cnt \vee (\exists c' \in W[i] \wedge c'.cnt \ge c.cnt)\}| > t$
    $\mathsf{highCand}(c) \triangleq c \in C \wedge (c.cnt = \max\{c'.cnt : c' \in C\})$
Local variables (GET):
    $PW[1\ldots n] \in 2^{Integers}$, $W[1\ldots n] \in 2^{TSsInt}$, $C \in 2^{TSsInt}$
    $y[1\ldots n] \in Integers \times TSsInt \cup \{\bot\}$
    $ts \in Integers$, initially 0
GET()
10    for $1 \le i \le n$ do $y[i] \leftarrow \bot$; $PW[i] \leftarrow W[i] \leftarrow \emptyset$
11    $C \leftarrow \emptyset$
12    $ts \leftarrow ts + 1$
13    for $1 \le i \le n$ do invoke $y[i] \leftarrow read(Y_i)$
      repeat
14        CHECK
15    until a set $S$ of $n-t$ objects responded $\wedge\ \forall i,j \in S : \neg\mathsf{conflict}(i,j)$
16    $C \leftarrow \{y[i].w : |\{j : y[j].w \ne y[i].w\}| \le 2t\}$
17    for $1 \le i \le n$ do invoke $y[i] \leftarrow write\&read(\langle T_i, ts\rangle, Y_i)$
18    repeat
19        CHECK
20        $C \leftarrow C \setminus \{c \in C : |\{i : \exists c' \in W[i] \wedge c' \ne c\}| \ge 2t+1\}$
21    until $n-t$ responded $\wedge\ \exists c \in C$: $(\mathsf{safe}(c) \wedge \mathsf{highCand}(c)) \vee C = \emptyset$
22    if $C \ne \emptyset$ then return $c.cnt$ else return 0

CHECK
    if $Y_i$ responded then
        $PW[i] \leftarrow PW[i] \cup \{y[i].pw\}$
        $W[i] \leftarrow W[i] \cup \{y[i].w\}$
```

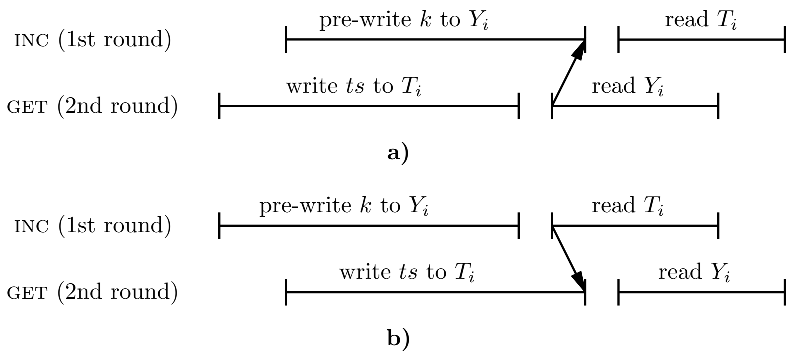
Proof. Let INC^k be the last operation preceding the invocation of GET. Furthermore, for all $l > k$, GET precedes INC^l. By assumption, $c.cnt = k$ was written to the w field of $t+1$ correct objects before GET is invoked. Therefore, c is added to the candidate set C (line 16) and because at most $2t$ objects respond without c, it is never removed. Furthermore, $t+1$ correct objects eventually report c in the second GET round and c becomes safe. As there are no concurrent INC operations, eventually $2t+1$ correct objects report values k or lower from their w field and hence all c_h where $c_h.cnt > k$ are removed from C. Thus, c eventually becomes both safe and highCand and $c.cnt = k$ is returned. □

Lemma 5 (Wait-freedom). *The counter object implemented in Figure 6 is wait-free.*

Proof. As the INC operation never waits for more than $n-t$ responses, clearly it never blocks. In the following we prove that the GET operation does not block (1) at line 15 and (2) at line 21. We assume by contradition that the GET operation blocks. Case (1): as the GET operation never updates a correct base object with ts before the second round, correct base objects are never in conflict with each other and thus the GET operation does not block at line 15. Case (2): The GET operation blocks at line 21. Therefore, there exists $c \in C$ and c is not safe. Let $c.cnt = k$. If some correct base object has reported c in its w field in the first round of GET, then $t+1$ correct base objects report k or higher in their pw field in the second round and thus c is safe. Therefore, we assume that no correct base object reports c in w in the first round. If no correct object reports c in w in the second round, then $2t+1$ correct base objects respond with $c' \neq c$ in their w field and c is removed from C. In the following we assume that some correct object reports c in w in the second round. Let F ($|F| > 0$) denote the set of faulty objects that report c in their w field in the first round. Let X ($|X| \geq 0$) be the set of correct base objects i such that Y_i reports to the second GET round a value lower than k in both fields pw and w. This implies that the pre-write phase of INC at Y_i does not precede the second GET round reading Y_i (see Fig. 7 (a)). By the semantics of *write&read*, the second GET round has updated T_i with ts *before* reading Y_i (line 17). Similarly, the first round of INC has pre-written k to Y_i *before* reading T_i (line 3). By transitivity, the second GET round has completed the update of T_i *before* the first INC round has read T_i, and thus T_i reports ts (Fig. 7 (a)). Let $X' = \{j \in X : c.hrts[j] = ts\}$, that is, the objects in X that have actually responded to the first INC round. Note that for all $i \in F$ and for all $j \in X'$, conflict(i, j) is true. Hence, the $2t+1-|F|$ objects that have responded without c in their w field in the first round of GET do not include any object in X'. Overall, after the second GET round, $2t+1-|F|+|X'|$ base objects have responded without c in their w field. If $|F| \leq |X'|$ then c is removed from the set of candidates C (line 20), a contradiction. Therefore, we consider the case $|F| > |X'|$. Out of the $t+1$ correct base objects updated by the pre-write phase of INC, $t+1-|X'|$ respond with a timestamp lower than ts. Consequently, for every such base object i, GET has completed updating T_i with ts *not before* INC reads T_i (see Figure 7 (b)). By the semantics of *write&read*

and by the transitivity of the precedence relation, register Y_i has stored k in its *pw* field before the second GET round reads Y_i. Hence, at least $t+1-|X'|+|F|$ base objects report values k or higher. As $|F|>|X'|$, $t+1$ base objects report k or a higher value, and thus c is safe, a contradiction. □

Fig. 7. Safe counter correctness argument



Theorem 2. *The Algorithm in Figure 6 wait-free implements a safe counter.*

Proof. Follows directly from Lemma 3, 4 and 5. □

4.2 The DMS3 Protocol

Protocol Description

In this section we present a robust and amnesic SRSW register construction from a safe counter and $3t+1$ regular base registers, out of which t can be subject to NR-arbitrary failures. We now describe the WRITE and READ operations of the DMS3 algorithm illustrated in Figure 8.

The WRITE operation performs in three phases, (1) a pre-write phase (lines 7–9) where it stores a timestamp-value pair c in the *pw* field of $n-t$ registers, (2) a read phase (line 10), where it calls GET to read the current view and (3) a write phase (lines 14–16), where it overwrites the w field of $n-t$ registers with c. If the read phase results in a view change, the most recent value previously written is frozen together with the new view. This is done by updating the *view* field and copying the value stored in w to the *frozen* field (lines 11–13). The reader performs exactly the same steps as in DMS (see Section 3).

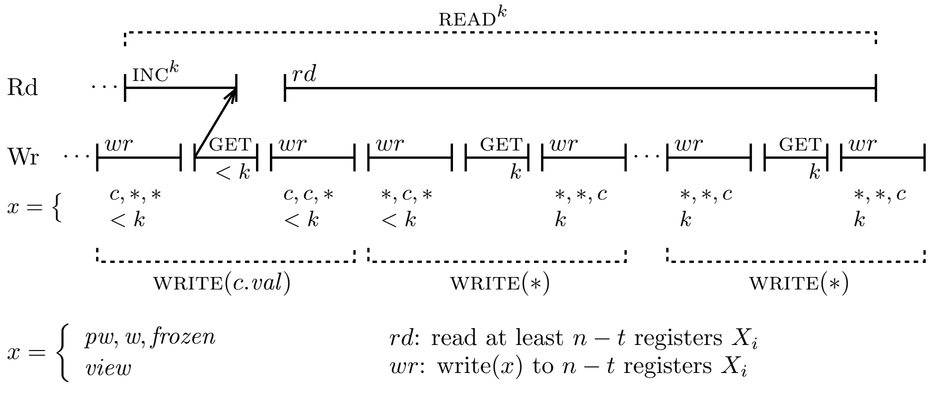
We now explain with help of Figure 9 why READs are wait free. Similar to the description of DMS in Section 3, we consider READ^k and the last WRITE that reads a view lower than k. Note that INC^k does not precede GET and thus, c is stored in the *pw* field of $t+1$ correct registers before they are read. Also, the w field of $t+1$ correct registers is updated with c. As the subsequent WRITE

Fig. 8. Robust and amnesic storage algorithm DMS3 $(3t+1)$

```
Shared objects:
    regular registers $X_i \in \mathit{TSVals}^3 \times \mathit{Integers}$, with selectors $\mathit{pw}$, $w$, $\mathit{frozen}$
    and $\mathit{view}$, initially $X_i = \langle\langle 0, v_0\rangle, \langle 0, v_0\rangle, \langle 0, v_0\rangle, 0\rangle$

Predicates (reader):
    $\mathsf{readFrom}(c, i) \triangleq (c = x[i].w \wedge x[i].\mathit{view} < \mathit{view}) \vee$
                        $(c = x[i].\mathit{frozen} \wedge x[i].\mathit{view} = \mathit{view})$
    $\mathsf{safe}(c) \triangleq |\{i : c \in \{x[i].\mathit{pw}, x[i].w, x[i].\mathit{frozen}\}\}| \geq t+1$
    $\mathsf{highestCand}(c) \triangleq |\{i : \mathsf{readFrom}(c', i) \wedge c'.\mathit{ts} \leq c.\mathit{ts}\}| \geq 2t+1$

Local variables (reader):
    $\mathit{view} \in \mathit{Integers}$, initially 0
    $x[1 \ldots n] \in \mathit{TSVals}^3 \times \mathit{Integers} \cup \{\perp\}$

READ()
1       **for** $1 \leq i \leq n$ **do** $x[i] \leftarrow \perp$
2       $\mathit{view} \leftarrow \mathrm{INC}(Y)$
3       **for** $1 \leq i \leq n$ **do invoke** $x[i] \leftarrow \mathit{read}(X_i)$
4       **wait** until $n - t$ responded $\wedge\ \exists c \in \mathit{TSVals}$: $\mathsf{safe}(c) \wedge \mathsf{highestCand}(c)$
5       return $c.\mathit{val}$

Local variables (writer):
    $\mathit{ts}$, $\mathit{newView} \in \mathit{Integers}$, initially 0
    $x \in \mathit{TSVals}^3 \times \mathit{Integers}$, initially $\langle\langle 0, v_0\rangle, \langle 0, v_0\rangle, \langle 0, v_0\rangle, 0\rangle$

WRITE($v$)
6       $\mathit{ts} \leftarrow \mathit{ts}+1$
7       $x.\mathit{pw} \leftarrow \langle \mathit{ts}, v\rangle$
8       **for** $1 \leq i \leq n$ **do invoke** $\mathit{write}(X_i, x)$
9       **wait** for $n - t$ responses
10      $\mathit{newView} \leftarrow \mathrm{GET}(Y)$
11      **if** $\mathit{newView} > x.\mathit{view}$ **then**
12          $x.\mathit{view} \leftarrow \mathit{newView}$
13          $x.\mathit{frozen} \leftarrow x.w$
14      $x.w \leftarrow \langle \mathit{ts}, v\rangle$
15      **for** $1 \leq i \leq n$ **do invoke** $\mathit{write}(X_i, x)$
16      **wait** for $n - t$ responses
17      return $\mathit{ack}$
```

encounters a view change, c is written to the *frozen* field of $t+1$ correct registers, where it stays until READ^k completes. Hence, c is sampled from $t+1$ correct registers' *pw, w* or *frozen* field and thus it is safe. Note that c is also highestCand because only faulty registers report newer values.

Fig. 9. Correctness argument of the READ operation in DMS3


READ^k
Rd
INC^k
rd
Wr
wr
GET
$< k$
k
$x = \{$
$c, *, *$
$c, c, *$
$*, c, *$
$*, *, c$
WRITE($c.val$)
WRITE($*$)
$x = \{$ $pw, w, frozen$ / $view$
rd: read at least $n - t$ registers X_i
wr: write(x) to $n - t$ registers X_i

With DMS3, the high-level operations have a non-optimal time-complexity. We now explain how the optimized version is obtained by collapsing individual low-level operations. More precisely, a write operation and a consecutive read operation are merged together to a *write&read* operation. The safe counter abstraction is disregarded and the counter operations INC and GET are weaved into READ and WRITE respectively. Recall that the counter operations consist of two rounds each. In the WRITE implementation, the pre-write phase and the first round of GET are collapsed. Note that the three-phase structure of the WRITE is preserved in that the writer reads the current view *before* it moves to the write phase. Similarly, in the READ implementation, the second INC round and the read phase are merged together. Overall, this results in a time-complexity of *three* rounds for the WRITE and *two* rounds for the READ.

We now informally argue that the optimization is correctness preserving. As above, we consider READ^k and the last WRITE that reads a view lower than k. We argue that $t+1$ correct base registers have stored c in their *pw* field before any of them is read. This would imply that c is safe. The fact that the WRITE of $c.val$ reads a view lower than k implies that k is missing from at least $2t+1$ base objects. We know from the safe counter algorithm in the previous section that if only $2t$ base objects respond without k, then k is never removed from the set of candidates. As the safe counter implementation is wait-free, k is eventually read, contradicting the initial assumption. Therefore, $2t+1$ base objects respond without k, and thus there are $t+1$ correct base objects among them that are accessed by (the read phase of) READ^k only after c was pre-written to them. By applying similar arguments as above, it is not difficult to see that c does not disappear from any of the $t+1$ correct base objects before READ^k completes. This would imply that c eventually becomes safe. For a formal treatment we refer the interested reader to our full paper [19]. The remainder of this section is concerned with the correctness of DMS3.

Protocol Correctness

Lemma 6 (Regularity). *Algorithm DMS3 in Figure 8 implements a regular register.*

Proof. Identical to the proof of Lemma 1. □

Lemma 7 (Wait-freedom). *Algorithm DMS3 in Figure 8 implements wait-free* READ *and* WRITE *operations.*

Proof. The WRITE operation is nonblocking because it never waits for more than $n-t$ responses. To derive a contradiction we assume that READ^k blocks at line 4 and show that there exists a candidate for returning. We consider the time after which all correct base objects (at least $2t+1$) have responded. We choose c as the highest timestamp-value pair readFrom a correct register. Note that c is highestCand by construction because values with timestamps $\leq c.ts$ are readFrom $2t+1$ correct registers. In the following, we distinguish the cases where the view read by the WRITE of $c.val$ is equal to k (case 1) or it is lower than k (case 2). Note that by the validity of the counter, only views $\leq k$ are returned. Case 1: Let X_i be a correct register such that readFrom(c, i). Since by assumption $x[i].view = k$, c is readFrom the *frozen* field of X_i. However, in view k only timestamp-value pairs lower than c are frozen, a contradiction. Now we consider case 2, where the WRITE($c.val$) reads a view lower than k. This implies that INC^k does not precede GET. As the pre-write phase (lines 8–9) precedes GET (line 10), and INC^k (line 2) precedes the read phase (lines 3–4), by transitivity, the pre-write phase also precedes the read phase (see Figure 9). Thus, $t+1$ correct registers have stored c in their pw field *before* they are read. What is left to show is that no subsequent WRITE erases c from all fields of those $t+1$ correct registers. Note that in view k, only timestamp-value pairs c or higher a frozen. Thus, if c was stored in the w field of $t+1$ correct registers before they are read, then c would be safe. Hence, c is missing from $t+1$ correct registers' w field. Consequently, WRITE($c.val$)'s write phase (lines 15–16) does not precede READ^k's read phase (lines 3–4). By transitivity, the subsequent WRITE reads view k and freezes c. Note that c is erased from pw only after c was previously stored in w (line 14). Furthermore, c is erased from w only after it was stored in *frozen* (line 13). As k is the last view, by the validity of the safe counter, c is never erased from *frozen.* □

Theorem 3 (Robustness). *Algorithm DMS3 in Figure 8 implements a robust register.*

Proof. Immediately follows from Lemma 6 and 7.

5 Concluding Remarks

We have presented amnesic algorithms that robustly implement a shared register from a collection of n base objects, of which up to $t < n/3$ can be subject to NR-arbitrary failures. For $n \geq 3t + 1$ we have shown that *two* rounds of communication with the base objects are sufficient for every READ operation to complete. This is the first robust and amnesic register construction that matches the *two*-round lower bound proved in [10]. For the $n \geq 4t + 1$ case, we have presented the first robust and amnesic register construction that matches the (trivial) *one*-round lower bound for *every* operation. Note that our construction is tight because with less than $4t+1$ base objects, both the READ and the WRITE operations require at least *two* communication rounds [2, 10].

The main result of this paper, that robust access to amnesic storage is possible in optimal time is somewhat surprising given the large body of literature on non-amnesic [4, 10–14] and non-robust [5, 8, 9, 18] algorithms. Moreover, our result is counter-intuitive because so far, only non-amnesic algorithms match the time-complexity lower bounds. As a corollary, our result suggests that the intuition of amnesic algorithms being inherently less efficient than non-amnesic ones is largely unjustified.

Some of the prior amnesic (but not robust) register implementations assume that the readers cannot modify the base objects (see e.g. [2]). This assumption in fact results in implementations that possess several properties that could be valuable in practice, for instance the ability to tolerate any number of malicious readers while using only $\mathcal{O}(1)$ memory at the base objects. We are not aware of any robust implementation supporting that as well, and in fact, our algorithms are not an exception. We leave as an open problem the question whether robust and amnesic register implementations exist, that would support any number of readers while using only $\mathcal{O}(1)$ memory at the base objects.

Acknowledgments

We thank Gregory Chockler, Felix Freiling, Marco Serafini and Jay Wylie for many useful comments on an earlier version of this paper.

References

1. Lamport, L.: On interprocess communication. part II: Algorithms. Distributed Computing **1**(2) (1986) 86–101
2. Abraham, I., Chockler, G., Keidar, I., Malkhi, D.: Byzantine disk paxos: optimal resilience with byzantine shared memory. Distributed Computing **18**(5) (2006) 387–408
3. Chockler, G., Malkhi, D.: Active disk paxos with infinitely many processes. Distributed Computing **18**(1) (2005) 73–84
4. 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

5. Jayanti, P., Chandra, T.D., Toueg, S.: Fault-tolerant wait-free shared objects. J. ACM **45**(3) (1998) 451–500
6. Herlihy, M.: Wait-free synchronization. ACM Trans. Program. Lang. Syst. **13**(1) (1991) 124–149
7. Chockler, G., Guerraoui, R., Keidar, I.: Amnesic Distributed Storage. In: Proceedings of the 21st International Symposium on Distributed Computing (DISC'07). (2007) 139–151
8. 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
9. Malkhi, D., Reiter, M.: Byzantine quorum systems. Distrib. Comput. **11**(4) (1998) 203–213
10. 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
11. 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) 135–144
12. 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
13. Bazzi, R.A., Ding, Y.: Non-skipping timestamps for byzantine data storage systems. In: DISC. (2004) 405–419
14. Aiyer, A., Alvisi, L., Bazzi, R.A.: Bounded wait-free implementation of optimally resilient byzantine storage without (unproven) cryptographic assumptions. In: DISC. (2007) 7–19
15. 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
16. 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) 34–43
17. 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
18. Abraham, I., Chockler, G., Keidar, I., Malkhi, D.: Wait-free regular storage from byzantine components. Inf. Process. Lett. **101**(2) (2007)
19. Dobre, D., Majuntke, M., Suri, N.: On the time-complexity of robust and amnesic storage. Technical Report TR-TUD-DEEDS-04-01-2008, Technische Universität Darmstadt (2008) http://www.deeds.informatik.tu-darmstadt.de/dan/amnesicTR.pdf.
20. Gregory Chockler, Rachid Guerraoui, I.K., Vukolic, M.: Reliable distributed storage. IEEE Computer (2008) To appear.