Amnesic Distributed Storage

Gregory Chockler[1], Rachid Guerraoui[2⋆], and Idit Keidar[3]

[1] IBM Haifa Research Lab, Haifa, Israel,
chockler@il.ibm.com
[2] School of Computer and Communication Sciences,
EPFL, CH-1015, Lausanne, Switzerland
rachid.guerraoui@epfl.ch
[3] Department of Electrical Engineering,
The Technion – Israel Institute of Technology

Abstract. Distributed storage algorithms implement the abstraction of a shared register over distributed base objects. We study a specific class of storage algorithms, which we call *amnesic*: these have the pragmatic property that old values written in the implemented register might be eventually forgotten, i.e., they are not permanently kept in the storage and might be overwritten in the base objects by more recent values. This paper precisely captures this property and argues that most storage algorithms are amnesic. We establish a fundamental impossibility of an amnesic storage algorithm to implement a robust register abstraction over a set of base objects of which at least one can fail arbitrarily, even if only in a responsive manner, unless readers are allowed to write to the base objects. Our impossibility helps justify the assumptions made by practical robust storage algorithms. We also derive from this impossibility the first sharp distinction between *safe* and *regular* registers. Namely, we show that, if readers do not write, then no amnesic algorithm can implement a regular register using safe registers.

1 Introduction

Storage is a critical aspect of modern computing systems. Today, there is strong interest in distributed storage architectures, either server-based or in the form of storage area networks *(SANs)*, which leverage the technological advances in networks of attached commodity disks to provide increased storage space, availability, and disaster recovery. At the heart of a distributed storage architecture lies an algorithm that implements the *read* and *write* operations of a register abstraction over several underlying base objects, sometimes called servers. Such distributed storage algorithms constitute an active area of research. A major challenge addressed by these algorithms is to ensure that (high-level) *read* and *write* implementations tolerate asynchrony, contention, and failures.

We study in this paper the fundamental limitations of a specific class of storage algorithms, which we define precisely and call *amnesic*. As we explain

⋆ Part of this work was conducted when the author was on sabbatical at MIT CSAIL.

later (Section 5), most previously suggested storage algorithms are amnesic, e.g., [15,6,13,17,2,16,8,11,12,3], although the notion has never been specifically highlighted. Roughly speaking, an *amnesic* storage algorithm is one that eventually forgets old values previously stored in the implemented register after some sequence of new values is written. For instance, an algorithm that stores in base objects the last k values written in the implemented register, for some $k \geq 0$, e.g., [6,13,2,3], is amnesic, because a sequence of k new writes erases all previously stored values. On the other hand, an algorithm that stores the entire history of values written in the base objects, (where values are drawn from an unbounded domain), e.g., [10], is not amnesic. In this sense, amnesia can be seen as a restriction on an algorithm's space consumption, although it is not explicitly formulated this way. Instead, we capture the notion of amnesia in an abstract way in terms of *reachable configurations* of a distributed storage algorithm (Section 3).

Our motivation for refraining from an explicit space restriction is twofold. First, we are interested in algorithms that manipulate potentially unbounded value domains, such as integers, or files. Although in every execution, such an algorithm's space consumption is finite (and depends on the sizes of the written values), it is inherently unbounded. Second, many practical algorithms employ monotonically increasing timestamps [17,14,18,2,10,3,12], which are considered pretty cheap in practice. Thus, the classical concept of *bounded memory* is inadequate for reasoning about many interesting algorithms. We further note that the concept of bounded memory, by itself, does not capture "reasonable" space restrictions. For example, it does not preclude an algorithm that manipulates a large but finite domain (e.g., files of size 1KB), from storing all the (2^{8192}) values in its domain if they were all written at some point. Focusing on amnesic algorithms with unbounded domains provides an abstract way to rule out such algorithms, without precluding the use of increasing timestamps.

We establish in this paper (Section 4) a fundamental limitation on amnesic storage algorithms. We prove that it is impossible for an amnesic storage algorithm to *robustly* implement a register abstraction using a set of distributed (failure-prone) base objects, when readers do not write to the base objects. Underlying our impossibility lies the notion of *robustness*. In short, we consider as *robust* an algorithm that implements a live regular register [15] in the presence of contention, asynchrony, and arbitrary (Byzantine) failures of base objects [13,5,4]. Our impossibility holds if at least one base object can fail in a responsive yet arbitrary manner (R-Arbitrary failure [13]) among an arbitrarily large set of atomic base objects. (A fortiori, the result also holds if a base object may fail in a non-responsive arbitrary (NR-arbitrary) manner). Such arbitrary failures capture software bugs or malicious intrusions, which cannot be ruled out when the service is geographically disperse. We do not require that the algorithm tolerate process failures, which also strengthens our impossibility result.

The assumption that readers do not write is important for large systems with many readers. Whereas it is reasonable for storage servers to communicate using an authenticated channel with a single trusted writer and to assume it

not to be malicious[4], a storage that is accessible to a large population of readers cannot typically trust all of them (authenticating all readers to prevent storage corruption might be infeasible). Hence, a more feasible alternative is to disallow the readers to modify the base object states, which is the assumption under which our impossibility is proven.

Given the vast amount of work on practical robust storage, our impossibility result may come as a surprise. In fact, our result does not imply that practical robust storage is unattainable, but rather justifies *why* previous solutions have had to employ authentication [8,17,19], store unbounded histories [10], have servers actively push updates to clients [18], give up on liveness in some situations [2], allow readers to write [3,11], or implement safe registers instead of regular ones [13,17,2,12] (see Section 5).

Our impossibility indeed holds if the implemented register needs to be regular but not if it needs to be safe. In a sense, regularity conveys an important aspect of robustness in the face of concurrency: no value can be returned if it was not written. Since many amnesic robust storage solutions implement safe registers [13,17], we can use our impossibility to derive the first sharp line between safe and regular semantics. We prove that there is no amnesic implementation of a regular register with an unbounded domain from safe ones, if readers do not write (Section 6). The line we draw between safety and regularity is analogous to the celebrated sharp line drawn by Lamport between atomicity and regularity [15], which states that no *bounded memory* algorithm can implement an *atomic* register using *regular* ones, if the readers do not write [15]. No such separation between safety and regularity has ever been established. We identify such a separation by replacing the notion of *bounded* algorithm in Lamport's formulation with our alternative notion of *amnesic* algorithm.
To summarize, this paper makes the following contributions:

- We define the notion of *amnesia*, capturing a pragmatic property of many storage algorithms.
- We prove the *impossibility* of devising a storage algorithm that is *robust* and *amnesic* without allowing readers to write.
- We derive the first sharp distinction between *safe* and *regular* register semantics.

2 Model

2.1 Shared memory model

We consider an asynchronous shared memory system consisting of a finite collection of processes interacting through n base objects $O_1, \ldots, O_n$. The term *base* objects is used to distinguish these from the higher level object abstraction implemented by the shared memory algorithm.

[4] With a compromised writer, the stored information is rendered meaningless anyway, regardless of any distributed storage algorithm's actions [17,19].

We consider storage algorithms that tolerate at least one arbitrary failure of a base objects. We focus on a weak form of such failures, called *responsive arbitrary (R-Arbitrary)* [13]. This means that the base objects always respond to an invocation, but may respond with an arbitrary value. This assumption strengthens our impossibility result, which directly applies to the more severe *non-responsive arbitrary (NR-Arbitrary)* [13] failures.

Processes are sequential in their ways of invoking high-level operations. That is, after invoking a high-level operation, and until it obtains a response, a process does not invoke a new high-level operation. After invoking a high-level operation, a process might invoke a sequence of low-level operations. We do not assume these low-level invocations to be sequential. That is, a process might invoke several operations on low-level base objects concurrently.

The solution must be *live* in the sense that every high-level operation must eventually complete. We do not require the algorithm to terminate in the presence of process failures, i.e., it does not have to be *wait-free*.

2.2 Registers

We study more specifically storage algorithms that deterministically implement the abstraction of a *register*, which is accessed using *Read()* and *Write()* operations. If the base objects are also registers, we denote their (low-level) operations as *read()* and *write()*, to avoid confusion with the high-level operations. To strengthen our impossibility result, we restrict our attention to storage algorithms that emulate a *single-writer single-reader (SWSR)* register: that is, the emulated register is only writable by a single process (the *writer*), and is read by a single process (the *reader*). The result a fortiori holds for multi-writer multi-reader registers. We assume an infinite value domain V from which the parameters of the *Write()* operation can be arbitrarily chosen by the writer. When it completes, *Write()* simply returns an *Ok* indication. A *Read()* operation does not have any input parameter, and returns a value from V upon completion.

The sequential specification of a register stipulates that a read should return the last value written. When read and write operations may overlap, several semantics have been defined [15]: A register is called *safe* if every read operation that does not overlap any write operation returns the register's value when the read was invoked, i.e., the latest written value or the initial value of the register if no value was written. A register is *regular* if it is safe and every read operation that overlaps some write operations returns either one of the values written by overlapping writes or the register's value before the first overlapping write is invoked. A register is *atomic* if it is regular and if, for any two write operations W and W' with respective input values v and v' such that W' is invoked after W returns, and any two read operations R and R' such that R' is invoked after R returns, if R' returns v, then R does not return v'. To strengthen our impossibility result, we will allow base objects to be atomic.

2.3 Configurations

In this paper, we are only interested in the states of base objects, and not of processes. Therefore, by slight abuse of terminology, we define a *configuration* to be a set of states of all the correct base objects (and not the processes). Basically, the system starts from an initial configuration and each atomic step of the algorithm, e.g., a low-level *write()* on a base object, leads the system to a new configuration. The execution of a high-level operation involves several atomic steps that lead the system from a configuration C to another configuration C'. The assumption that the reader does not write means that $C' = C$ in case no *Write()* is invoked.

We say that a configuration C' is *write-reachable* from a configuration C if there is a sequence S of *Write()* operations that, when executed without overlapping with any other high-level operation, leads the system to C'. If all input parameters of the *Write()* operations of sequence S are from a set $V' \subset V$, we say that C' is *write-reachable* from C *using* V'.

3 Amnesic Storage Algorithms

We introduce in this section the notion of an *amnesic storage algorithm*. We characterize this notion in terms of configurations and write-reachability.

Intuitively, a storage algorithm is amnesic if all but a finite number of configurations reached by the algorithm can be eventually *erased* if a sufficient number of different values are written after them. In short, erasing a configuration C', itself obtained from some configuration C, means reaching a new configuration C'' (after writing a sufficient number of different values) that makes it impossible to tell whether C' was indeed reached. That is, C'' could be reached directly from C without going through C'. The sequence that reaches C'' from C' is in a sense an *eraser* sequence. An observer of C'' does not know whether C' occurred or not.

To preclude the trivial case where erasing a configuration is always performed by the very same sequence, i.e., some sort or *reset* sequence, we require that the configuration C'' could be obtained from C by using values from *any* infinite subset of values. Notice that we do not require that *any* sufficiently long sequence erases every configuration. Yet, our definition is rather weak because we simply require that every configuration has an eraser sequence using any infinite subset of values: such a weak definition strengthens our impossibility result. Formally,

Definition 1 (Amnesic Storage). *A storage algorithm A is amnesic if in every execution of A in which infinitely many different values are written, there is a point t, so that for every configuration C reached from point t onward, every configuration $C1$ write-reachable from C, and every infinite subset of values $V' \subset V$, there is a configuration $C2$ that is write-reachable from both C and $C1$ using V'.*

We say that (see Figure 1) t is the *amnesia point*, $C2$ is an *eraser configuration* of $C1$ from C using V'; the sequence of *Writes()* used to reach $C2$ from $C1$ is

an *eraser sequence* of $C1$ from C using V'; the sequence used to reach $C2$ from C is a *bypass sequence* of $C1$ from C using V'.

Fig. 1. Amnesia.

Clearly, an algorithm that recalls the entire history of written values in the base registers is not amnesic. On the other hand, an algorithm that stores in all base registers the last k values written in the high-level register is amnesic. The eraser sequence $S2$ simply needs to be of size k and from V'. The amnesia point captures a situation where an algorithm initially stores some finite number of values forever, but eventually, (at the amnesia point), its storage is "exhausted" and it cannot store additional values forever. To be non-amnesic, an algorithm with an infinite domain needs to be able to recall (in the sense that they cannot be erased) infinitely many configurations that it visited.

It is important to notice here that storage algorithms can also record timestamps while being amnesic. Consider for instance an algorithm that stores in the base registers the last k values written, as well as the total number of values written. Consider a configuration $C1$ obtained after writing i values, starting from an initial configuration C. An eraser sequence using some infinite subset V' consists of k new different *Write()* invocations with parameters from V', and a bypass sequence consists of $i+k$ different *Write()* invocations with parameters from V', the latter k being the same as for the eraser sequence.

Note that violating amnesia does not directly translate to excessive storage requirements. An algorithm may be able to represent some property of an unbounded history in a bounded way, much like a finite-state automaton can recall that an unbounded string belongs to some regular language[5]. Nevertheless, we are unaware of any previous *storage* algorithm that employs such a succinct representation, and so our impossibility result has broad applicability.

4 Impossibility of Amnesic Robust Storage

In this section, we establish the impossibility of devising a storage algorithm that is at the same time *amnesic* and *robust* when readers do not write. Recall that in our context, a storage algorithm is *robust* if (a) at least one base object

[5] We are grateful to Prasad Jayanti for pointing this out.

can suffer an (R-arbitrary) failure; (b) the implemented register is *regular*, i.e., tolerates contention; (c) every invoked high-level operation terminates, including in the presence of contention.

4.1 Simplifications

As we assume that only the writer of the implemented register can modify the base objects, these are also, without loss of generality, single-writer registers, with the same writer as that of the implemented register. Still without loss of generality, since we preclude the reader from writing, we can assume that:

1. Every high level *Read()* invocation translates into a finite series of concurrent read invocations of all base objects O_1, O_2, .., O_n.
2. The set of configurations obtained after performing any sequence S of *Write()* operations without any overlapping *Read()* is the same as if this sequence was invoked concurrently with this *Read()*. Recall that our definition of configurations only includes base object states.

4.2 Overview of the impossibility proof

To prove our impossibility, we proceed by contradiction. More specifically, we assume that there is an amnesic robust storage algorithm A that implements a register over an infinite domain V and we exhibit a scenario where A violates the regularity of the register.

We show that A violates regularity by having the reader return a value that was never written to the implemented register. Not surprisingly, this value is obtained from a low-lever read of the faulty base register. Our scenario has the reader unable to distinguish the response of the faulty base register from the response of a correct one, precisely because A is amnesic, and readers do not write. The proof then goes through three steps:

- *Step 1.* (Using the amnesia assumption.) We construct an execution E, which we call an *amnesic* execution, where sequences of *Write()* operations erase each other in turn, using n different subsets of value domains, $V_1, \ldots, V_n$. We will argue that every amnesic storage algorithm can generate such an execution.
- *Step 2.* (Using the assumption that readers do not write.) We next construct a slight modification E' of E, where the reader samples one base register after each sequence above has erased the previous configuration. No matter how many samples are taken, the reader still cannot obtain evidence of any written value from more than one base register, because the evidence is continuously erased. Finally, the reader returns some value v_i from some subset V_i, for which it saw evidence in some base register O_i.
- *Step 3.* (Using robustness.) Finally, we construct an execution E'_i in which no value from V_i is written, bypassing the configurations where values from V_i are stored. In E'_i, O_i incurs an R-arbitrary failure, and returns the same values as in E'. Since the reader cannot distinguish E'_i from E', it returns v_i, which was never written.

4.3 Impossibility

Theorem 1. *No storage algorithm can be amnesic and robust if a single base register can suffer a responsive arbitrary failure and readers do not write.*

Proof (Proof of Theorem 1.). We assume a storage algorithm A that deterministically implements the abstraction of a register with an infinite value domain V, using a collection of n base objects $O_1, \ldots, O_n$. We assume by contradiction that A is robust and amnesic.

We partition V into $n+1$ infinite subsets $V_0, V_1, \ldots, V_n$. (The intersection of every two subsets is empty, and the union of all subsets is V).

Step 1. We construct an infinite execution of A, E, which we call an *amnesic execution* (see Figure 2). E goes through the infinite sequence of configurations:

$$C_{1,1}, \ldots, C_{1,n}, C_{2,1}, \ldots, C_{2,n}, \ldots, C_{j,1}, \ldots, C_{j,n}, \ldots$$

such that for every $i \in \{1, .., n\}$, there is an execution E_i such that (a) E_i is the subsequence of E obtained by omitting all the configurations $C_{k,i}$, $k > 0$. (That is, configurations $C_{k,i}$ for all k are skipped.) And (b) no value from V_i is ever written in E_i.

Fig. 2. Amnesic execution E; execution E2 (bypassing V_2) highlighted.

We construct E recursively as follows. We perform a series of *Writes()* of different values from V_0 until the algorithm reaches an amnesia point at some configuration C. Then we apply a *Write()* of a single value from V_1. We denote the resulting configuration $C_{1,1}$. Then we use the assumption that the storage algorithm is amnesic and apply to $C_{1,1}$ a sequence that erases $C_{1,1}$ from C using V_2. We denote the resulting configuration by $C_{1,2}$. Then we use again the assumption that the storage algorithm is amnesic and apply to $C_{1,2}$ a sequence that erases $C_{1,2}$ from $C_{1,1}$ using V_3. We denote the resulting configuration by $C_{1,3}$. And so forth recursively. We apply to $C_{j,k}$ a sequence that erases $C_{j,k}$ from $C_{j,k-1}$ using V_{k+1}, where $C_{j,n+1} = C_{j+1,1}$ and $V_{n+1} = V_1$. The resulting execution E is infinite and can be generated from every amnesic storage algorithm.

In addition, for every $1 \leq i \leq n$, execution E_i is constructed, also recursively, as follows (see Figure 2). Up to $C_{1,i-1}$, E_i is exactly like E and hence no *Write()* uses any value from V_i. Configuration $C_{1,i+1}$ is then (directly) reached from $C_{1,i-1}$ via the bypassing sequence of $C_{1,i}$ from $C_{1,i-1}$ using values from V_{i+1}. Then we continue as in E until $C_{2,i-1}$, at which point we execute the bypass sequence of $C_{2,i}$ from $C_{2,i-1}$. And so forth: we apply the same sequence as in E to reach $C_{j,k}$ from $C_{j,k-1}$ for $k \neq i$, and for i, we apply the bypass sequence of $C_{j,i}$ from $C_{j,i-1}$ to reach $C_{j,i+1}$ from $C_{j,i-1}$. It is easy to see that properties (a) and (b) above hold for every E_i, for $i \in \{1, .., n\}$.

Step 2. We now construct an execution E' interleaving a single *Read()* with the sequences of *Write()* operations involved in E. Remember that, without loss of generality, we assume that every *Read()* implementation consists of a sequence of concurrent invocations of all base objects. The interleaving in execution E' is constructed as follows. *Read()* is invoked when the base objects are in configuration $C_{1,1}$. The reader returns from the kth *read()* of base object O_j when the system is in configuration $C_{k,j}$. For instance, the reader returns from the first *read()* of the first base object, O_1, when the system is in configuration $C_{1,1}$, then from the first *read()* of the second base object, O_2, when the system is in configuration $C_{1,2}$.

By the assumption that the reader does not write, execution E' can also be generated by every amnesic storage algorithm. By our liveness assumption, the *Read()* eventually returns a value. Since the *Read()* is invoked after the first write from V_1, by regularity, it returns a value v_i from some V_i for $0 < i \leq n$.

Step 3. We now make use of our assumption of robustness to derive a contradiction. We construct execution E'_i, which is the same as E_i, with two exceptions:

1. For every $j \neq i$, as in E', we apply the kth *read()* of base object O_j when the system is in configuration $C_{k,j}$.
2. The kth *read()* of base object O_i occurs during the kth bypass.
3. O_i returns the same response to its kth *read* invocation in E'_i as in in execution E'.

Execution E'_i can also be generated by every amnesic storage algorithm with base object O_i failing in an arbitrary way. Executions E' and E'_i look the same to the reader (by construction), which then returns a value v_i from V_i in E'_i. But no value from V_i is written in E'_i, contradicting regularity.

5 Amnesic Algorithms and Circumventing the Impossibility

Our impossibility justifies certain assumptions and design decisions made by existing storage algorithms. In this section, we illustrate the importance of the notion of amnesia, by showing that the majority of reliable storage algorithms in the literature are amnesic, and discuss how existing algorithms circumvent our impossibility result.

First note that every bounded memory algorithm is by definition trivially amnesic, because no infinite sub-domains of its domain exist. Since our impossibility result only applies to algorithms that can store values from unbounded domains, it is more interesting to consider algorithms that can manipulate such domains. Interestingly, most bounded memory algorithms in the literature can be easily extended to support unbounded domains. For example, Jayanti et al. [13] present an emulation of a safe register from ones that can suffer NR-Arbitrary faults. Although originally described as a bounded memory algorithm, it does not make any use of the domain size, and only stores values from the domain. Hence, this algorithm can easily work with an unbounded domain, where in each execution, it consumes storage as required for representing the values written in that execution. This algorithm circumvents our impossibility result by implementing only safe storage.

Lamport [15] presents a bounded memory algorithm for implementing a wait-free regular register from safe bits, with readers that do not write. Given the existence of robust safe register emulations [13,17,2], had this algorithm manipulated unbounded domains, it would have contradicted our impossibility. However, this algorithm is aware of its value domain and makes heavy use of this knowledge– it stores one bit for each value in the value domain of the register. The algorithm works as follows: a write operation of the ith value in the domain changes the ith bit to 1, and subsequently writes 0 in bits $i-1, i-2, \ldots, 1$. The reader reads the bits $1, 2, 3, \ldots$ until it encounters a 1 in some bit i, at which point it stops reading and returns i. We observe that neither the reader nor the writer ever accesses a bit higher than the one pertaining to the largest written value. Therefore, this algorithm too can be extended to unbounded domains, e.g., integers, by allocating the ith bit the first time a value greater or equal to i is written. Despite its exponential storage requirements, the resulting algorithm is also amnesic, since writing a single value larger than all previously written ones is an eraser sequence. So how does this algorithm circumvent our impossibility result? We observe that the extended algorithm no longer ensures liveness, because if the writer writes an infinite monotonically increasing sequence of values (an *amnesic* sequence), the reader can "trail" the writer, and never encounter a 1 in any register. Thus, even such exponential storage does not save us from the impossibility.

Many algorithms that store unbounded timestamps are also amnesic, including the classical ABD [6] algorithm, which tolerates only crash failures of base objects, and the safe register emulations of [17,2,12]. Other amnesic algorithms provide atomic semantics (which are stronger than regular) either by assuming a stronger model where data is self-verifying (and hence cannot be forged by base objects), or by having readers write [17,16,8,11,3].

It is also possible to circumvent our impossibility with amnesic algorithms by providing weaker (non-terminating in case of contention) termination guarantees [2]. Specifically, Abraham et al. [2] propose a termination condition called *finite writes (FW)*, which guarantees progress only in executions with a finite

number of writes, and present a amnesic storage algorithm implementing a regular FW-terminating register.

A few algorithms circumvent our impossibility by forgoing amnesia. These include the Pasis system [10], which achieves atomic semantics. It circumvents our impossibility by having authenticated readers that are allowed to modify the data stored at the base registers (storage nodes). The storage nodes also keep all versions of data that have been written in the execution, and are therefore, not amnesic. To prevent storage exhaustion, the system implements a garbage collection mechanism, which works well in practice, but, as the authors point out in [9], might fail to terminate in some scenarios.

Martin et al. [18], as well as Bazzi and Ding [7] also provide atomic semantics. They assume storage servers (instead of base registers) that communicate with each other, and a subscription model whereby storage servers push writer updates to subscribed clients. Since theoretically, there is no bound on the number of messages in the reliable push channel in an asynchronous system, these algorithms are also not amnesic. This approach nevertheless, is a viable design alternative in the settings where servers are available and the number of clients is limited.

6 Sharp Separation Between Regularity and Safety

Many amnesic robust storage solutions implement safe registers [13,17,2]. This maybe surprising, as safe and regular semantics are commonly believed to be "equivalent", justified by the existence of known bounded memory reductions from regular registers to safe ones. In particular, Lamport [15] presents a bounded memory algorithm for emulating a regular register from safe ones, in which readers do not write. The algorithm assumes a bounded value domain and its storage requirements, as well as the number of memory accesses in the algorithm, are notoriously high (proportional to the number of possible values the regular register can hold).

The following theorem is an immediate corollary of Theorem 1 and the existence of amnesic storage algorithms that implement a t-tolerant wait-free safe register (with unbounded value domains) from a collection of n base registers up to t of which can suffer arbitrary failures [13,17,2].

Theorem 2. *If readers do not write, it is impossible to implement a live regular register from safe ones with an amnesic algorithm and an infinite domain.*

Interestingly, Lamport has proved the following [15] :

> If readers do not write, it is impossible to implement an atomic register from regular ones with a bounded algorithm.

Thus, Lamport has shown that in *bounded memory* implementations, disallowing readers to write draws a sharp line between regularity and atomicity, but not between safety and regularity.

Hence, our result shows that, when one considers amnesic with infinite value domain instead of bounded memory, the same sharp line does exist between regularity and safety.

7 Concluding Remarks

The observation that no existing storage algorithm with reasonable space requirements that is regular, live in the presence of contention, and does not require readers to write, or preclude arbitrary faults of base registers was made by Abraham et al. [1]. They conjectured that, roughly speaking, if readers do not write, then the storage size grows linearly with the number of values written over the execution's time span. The difficulty in proving this conjecture stems from the lack of appropriate definitions, since the classical notion of bounded memory cannot capture "linear growth" in storage requirements. Our notion of *amnesic* memory is an attempt to capture practical limitations on the information an algorithm recalls about its history, and gives an explanation to the observation that led to this conjecture. An interesting direction for future work may be providing a concrete lower bound on the space requirements of robust storage algorithms that are not amnesic. In addition, we believe that more impossibilities and fine grained distinctions could be obtained if one reconsiders bounded memory restrictions with our amnesic notion in mind.

Acknowledgments

We thank Ittai Abraham, Lorenzo Alvisi, Faith Ellen, Eli Gafni, Prasad Jayanti, Nancy Lynch, Dahlia Malkhi, Jean-Philippe Martin, Michel Raynal, and Marko Vukolić for many fruitful discussions on robust storage algorithms.

References

1. I. Abraham, G. Chockler, I. Keidar, and D. Malkhi, 2002. Private communication.
2. I. Abraham, G. Chockler, I. Keidar, and D. Malkhi. Byzantine Disk Paxos: Optimal resilience with byzantine shared memory. *Distributed Computing*, 18(5):387–408, Apr. 2006. Earlier version in PODC 2004.
3. I. Abraham, G. Chockler, I. Keidar, and D. Malkhi. Wait-free regular storage from byzantine components. *Information Processing Letters (IPL)*, 101(2):60–65, Jan. 2007.
4. Y. Afek, D. S. Greenberg, M. Merritt, and G. Taubenfeld. Computing with faulty shared objects. *Journal of the ACM*, 42(6):1231–1274, November 1995.
5. Y. Afek, M. Merritt, and G. Taubenfeld. Benign failures models for shared memory. In *7th Intl. Workshop on Distributed Algorithms*, pages 69–83. Springer Verlag, September 1993. *LNCS 725*.
6. H. Attiya, A. Bar-Noy, and D. Dolev. Sharing memory robustly in message-passing systems. *Journal of the ACM*, 42(1):124–142, January 1995.
7. R. A. Bazzi and Y. Ding. Non-skipping Timestamps for Byzantine Data Storage Systems. In *18th International Symposium on Distributed Computing (DISC'04), LNCS 3274*, pages 405–419, Oct. 2004.

8. C. Cachin and S. Tessaro. Optimal resilience for erasure-coded Byzantine distributed storage. In *Intl. Conference on Dependable Systems and Networks (DSN 2006)*, pages 115–124, June 2006.
9. G. Goodson, J. Wylie, G.Ganger, and M. Reiter. Efficient byzantine-tolerant erasure-coded storage. Technical Report CMU-PDL-03-104, Parallel Data Laboratory, CMU, December 2003.
10. G. Goodson, J. Wylie, G.Ganger, and M. Reiter. Efficient byzantine-tolerant erasure-coded storage. In *The International Conference on Dependable Systems and Networks (DSN-2004)*, June 2004.
11. R. Guerraoui, R. R. Levy, and M. Vukolic. Lucky read/write access to robust atomic storage. In *DSN*, pages 125–136. IEEE Computer Society, 2006.
12. R. Guerraoui and M. Vukolić. How Fast Can a Very Robust Read Be? In *25th ACM Symposium on Principles of Distributed Computing (PODC'06)*, 2006.
13. P. Jayanti, T. Chandra, , and S. Toueg. Fault-tolerant wait-free shared objects. *Journal of the ACM*, 45(3):451–500, 1998.
14. S. Lakshmanan, M. Ahamad, and H. Venkateswaran. Responsive security for stored data. *IEEE Trans. on Parallel and Distributed Systems*, 14(19):818–828, Sept. 2003.
15. L. Lamport. On interprocess communication – Part II: Algorithms. *Distributed Computing*, 1(2):86–101, 1986.
16. B. Liskov and R. Rodrigues. Byzantine clients rendered harmless. In *19th International Symposium on Distributed Computing (DISC 2005)*, LNCS. Springer, September 2005.
17. D. Malkhi and M. Reiter. Byzantine quorum systems. *Distributed Computing*, 11(4):203–213, 1998.
18. J.-P. Martin, L. Alvisi, and M. Dahlin. Minimal byzantine storage. In *Proceedings of the 16th International Symposium on Distributed Computing (DISC)*, October 2002.
19. R. Rodrigues and B. Liskov. Rosebud: A Scalable Byzantine-Fault-Tolerant Storage Architecture. Technical Report MIT-LCS-TR-932, MIT Laboratory for Computer Science, 2004.