Space Bounds for Reliable Storage: Fundamental Limits of Coding*

Alexander Spiegelman†
Viterbi EE Department
Technion, Haifa, Israel
sashas@tx.technion.ac.il

Yuval Cassuto
Viterbi EE Department
Technion, Haifa, Israel
ycassuto@ee.technion.ac.il

Gregory Chockler
CS Department
Royal Holloway, London, UK
gregory.chockler@rhul.ac.uk

Idit Keidar
Viterbi EE Department
Technion, Haifa, Israel
idish@ee.technion.ac.il

ABSTRACT

We study the inherent space requirements of reliable storage algorithms in asynchronous distributed systems. A number of recent works have used codes in order to achieve a better storage cost than the well-known replication approach. However, a closer look reveals that they incur extra costs in certain scenarios. Specifically, if multiple clients access the storage concurrently, then existing asynchronous code-based algorithms may store a number of copies of the data that grows linearly with the number of concurrent clients. We prove here that this is inherent. Given three parameters, (1) the data size – D bits, (2) the concurrency level – c, and (3) the number of storage node failures that need to be tolerated – f, we show a lower bound of $\Omega(min(f, c) \cdot D)$ bits on the space complexity of asynchronous distributed storage algorithms. Intuitively, this implies that the asymptotic storage cost is either as high as with replication, namely $O(fD)$, or as high under concurrency as with the aforementioned code-based algorithms, i.e., $O(cD)$.

We further present a technique for combining erasure codes with replication so as to obtain the best of both. We present an adaptive $f - tolerant$ storage algorithm whose storage cost is $O(min(f, c) \cdot D)$. Together, our results show that the space complexity of providing reliable storage in asynchronous distributed systems is $\Theta(min(f, c) \cdot D)$.

*This work was supported in part by the Israel Science Foundation and Royal Society International Exchanges Grant IE130802.

†Alexander Spiegelman is grateful to the Azrieli Foundation for the award of an Azrieli Fellowship.

PODC'16, July 25 - 28, 2016, Chicago, IL, USA

ISBN 978-1-4503-3964-3/16/07…$15.00

DOI: http://dx.doi.org/10.1145/2933057.2933104

1. INTRODUCTION

In recent years we have seen an exponential increase in storage capacity demands, creating a need for big data storage solutions. In this era, distributed storage plays a key role. Data is typically stored on a collection of nodes accessed asynchronously by clients over a network. By storing redundant information, data remains available following failures. The most common approach to achieve this is via replication [4]; in asynchronous settings, $2f + 1$ replicas are needed in order to tolerate f failures [4]. Given the immense size of data, the storage cost of replication is significant. Previous works have attempted to mitigate this cost via the use of erasure codes [3, 5, 11, 7, 18, 10].

Indeed, codes can reduce the storage cost as long as data is not accessed concurrently by multiple clients. For example, if the data size is D bits and a single failure needs to be tolerated, erasure-coded storage ideally requires $(k + 2)D/k$ bits for some parameter $k > 1$ instead of the $3D$ bits needed for replication. But as concurrency grows, the cost of erasure-coded storage grows with it: when c clients access the storage concurrently, existing asynchronous code-based algorithms [5, 11, 7, 10] store $O(cD)$ bits in storage nodes or communication channels. Intuitively, this occurs because coded data cannot be reconstructed from a single storage node. Therefore, writing coded data requires coordination – old values cannot be deleted before ensuring that sufficiently many blocks of the new value are in place. This is in contrast to replication, where written values can always be read coherently from a single copy, and so old values may be safely overwritten without coordination.

In this work we prove that this extra cost is inherent: Given three problem parameters: f, c, and D, where f is the number of storage node failures tolerated (client failures are unrestricted), c is the concurrency allowed by the algorithm, and D is the data size, we prove that the storage complexity is $\Theta(min(f, c) \cdot D)$. Asymptotically, this means either a storage cost as high as that of replication, or as high as keeping as many versions of the data as the concurrency level.

Lower bound.

Our results are proven for emulations of a lock-free multi-reader multi-writer regular register [14, 16]; see Section 2

for definitions. (Interestingly, the lower bound does not hold for the weaker safe register semantics; see our technical report [17] for details). We consider algorithms that use (arbitrary) *black-box* encoding schemes, i.e., produce and manipulate code blocks of a given value independently of other values and meta-data; as formalized in Section 3. The storage consists of such code blocks, in addition to possibly unbounded data-independent meta-data, (e.g., timestamps), which we do not count as part of the storage cost. Our black-box assumption excludes storage-reduction techniques like de-duplication, which do require data-dependent meta-data. In Section 4 we survey how this assumption holds in related work on popular storage algorithms [3, 5, 11, 7, 10, 12], and compare it with assumptions made in proving other lower bounds [6]. Yet, the question whether there is a more storage-efficient algorithm that circumvents our result by taking stored values into consideration remains open; see further discussion in Section 7.

We prove the bound in Section 5: we first use a fundamental pigeonhole argument to show that as long as no ongoing write operation contributes code blocks consisting of D or more bits to the storage, no write operation can complete. We then define a parameter $0 < \ell \leq D$. For a given ℓ, we devise a particular adversary behavior, which we prove drives the storage to a state where either (1) $f + 1$ storage nodes hold at least ℓ bits each, or (2) the storage holds more than $D - \ell + 1$ bits in distinct code blocks for each of c different operations. Now, picking $\ell = D/2$ implies our lower bound.

Algorithm.

To prove our bound tight, we present in Section 6 a reliable storage algorithm whose storage cost is $O(min(f, c) \cdot D)$. We achieve this by combining the advantages of replication and erasure coding. Our algorithm does not assume any a priori bound on concurrency; rather, it uses erasure codes when concurrency is low and adaptively switches to replication when it is high.

2. COMPUTATION MODEL

We consider an asynchronous fault-prone shared memory system [2, 1, 13] consisting of set $B = \{bo_i, \ldots, bo_n\}$ of n base objects (typically residing at distinct storage nodes) supporting arbitrary atomic *read-modify-write* (RMW) access by clients from some infinite set Π (see Figure 1). Any f out of n base objects and any number of clients may fail by crashing, for some predefined $f < n/2$.

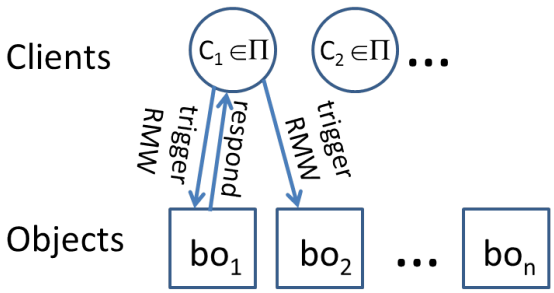


Figure 1: Clients and base objects.

We study algorithms that emulate a shared *register* [14], which stores a value v from some domain $\mathbb{V}$, where $D = \log_2 |\mathbb{V}|$. Initially, the register holds some initial value $v_0 \in \mathbb{V}$. Clients interact with the emulated register via high-level *read* and *write operations*. A client that performs a write operation is called a *writer*, and a client performing a read is a *reader*.

To distinguish the high-level emulated operations from low-level base object access, we refer to the latter as *RMWs*. We say that RMWs are *triggered* and *respond*, whereas operations are *invoked* and *return*. A (high-level) operation is emulated via a series of trigger and respond *actions* on base objects, starting with the operation's invocation and ending with its return. In the course of an operation, a client triggers RMWs separately on each $bo_i \in B$. The state of each $bo_i \in B$ changes atomically, according to the RMW triggered on it, at some point after the time when the RMW is triggered but no later than the time when the matching response occurs. To distinguish incomplete invocations to the emulated register from incomplete RMWs triggered on base objects, we refer to the former as *outstanding* operations and to the latter as *pending* RMWs.

A parameter c defines the write concurrency level, that is, at most c write operations are outstanding at a given time. We assume that $c < |\mathbb{V}|/2 = 2^{D-1}$. We use standard definitions of algorithms, runs, etc, which, for completeness, appear in Appendix A. The emulated register must satisfy the following two properties:

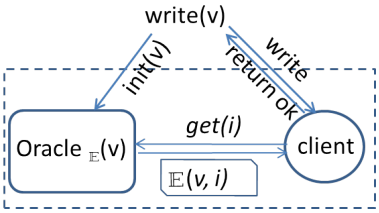
Lock-freedom If at some point in a fair run there is an outstanding operation of a correct client, then *some* operation eventually returns.

Regularity Our safety requirement is regularity, which is weaker than atomicity. There are a number of ways to extend Lamport's notion of regularity [14] to multi-writer registers [16]; we use the weakest one for our lower bound and the strongest for our algorithm, (called MWRegWeak and MWRegWO in [16], resp.), as defined in Appendix A. Intuitively, regularity means that a read rd returns a value written by either (1) the last write w that completes before rd is invoked, or (2) some write that is concurrent to rd or to w, or (3) v_0 if no value is written before rd.

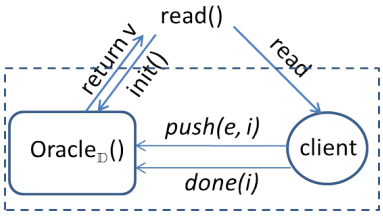
3. STORAGE ALGORITHM MODEL AND ASSUMPTIONS

We first give a formal model for coded storage algorithms, then define the notion of storage cost in this model, and finally state our assumptions that the encoding is symmetric and algorithms use it as a black-box.

We consider algorithms that use (arbitrary) encoding schemes, which produce code blocks in some domain $\mathcal{E}$, so that each value is coded independently of other values. The coding scheme is based on two functions: The encoding function $\mathbb{E} : \mathbb{V} \times \mathbb{N} \rightarrow \mathcal{E}$ maps value/natural number pairs to code blocks. We denote the number of bits in block $e \in \mathcal{E}$ as $|e|$. The decoding function $\mathbb{D} : 2^{\mathcal{E}} \rightarrow \mathbb{V} \cup \{\perp\}$ takes as a parameter a set of code blocks and returns a value in $\mathbb{V}$, or $\perp$ in case no value can be decoded. For example, in a replication approach, each block e can be a full value v, so $\mathbb{D}(\{e\})$ simply returns v. Another example is *k-of-n* erasure codes, where for any value v and any subset S of size k of the set $\{e_i \mid e_i = E(v, i),\ 1 \leq i \leq n\}$, $\mathbb{D}(S) = v$. We capture rateless codes [15], in which an encoder can generate a



(a) A writer and its oracle.



(b) A reader and its oracle.

Figure 2: A model for code-based storage. Encoding and decoding are captured by oracles.

limit-less sequence of blocks, by using $\mathbb{N}$ as the domain for block numbers.

We encapsulate the encoder and decoder into two oracles, $oracle_{\mathbb{E}}$ and $oracle_{\mathbb{D}}$ as illustrated in Figure 2. The interaction with these oracles is as follows:

Definition 1 (Encoding/Decoding Oracles)**.** A *w=write(v)* (*read()*) invocation at a client c_k initializes an $oracle_{\mathbb{E}}(c_k, w)$ ($oracle_{\mathbb{D}}(c_k, w)$, resp.), which expires when w completes. $oracle_{\mathbb{E}}(c_k, w)$ exposes a $get(i)$ operation, which returns $\mathbb{E}(v, i)$ for $i \in \mathbb{N}$; and $oracle_{\mathbb{D}}(c_k, w)$ exposes two operations, $push(e, i)$ and $done(i)$, such that for all $i \in \mathbb{N}$, if c_i calls $done(i)$, then its read operation completes and returns $\mathbb{D}(\{e \mid push(e, i)$ previously occurred$\})$. We omit the parameters c_k, w when they are clear from the context.

Writers produce code blocks via $oracle_{\mathbb{E}}$ and store them in the storage, whereas readers try to obtain enough blocks to decode legal values via $oracle_{\mathbb{D}}$. In addition to code blocks, clients and base objects can store unbounded meta-data, e.g., program counters and timestamps. But to avoid trivializing the problem, the meta-data must be data-independent, as formally defined below.

Information is represented as list of code blocks and meta-data, $\langle e_1, e_2, \ldots, e_k; m\rangle$, where $\forall i$, $e_i \in \mathcal{E}$ and the meta-data m is from some arbitrary domain. The *state of a client* that has an outstanding operation consists of the information stored at the client as well the parameters of its pending RMWs that have not yet taken effect. The state of a client with no outstanding operation is empty. A *base object's state* consists of the information stored at the base object and all the responses of pending RMWs that took effect on it. For a base object bo_i (client c_i), we denote the list of code blocks in bo_i's (c_i's) state at time t in run r as $bo_i^r(t)$ (resp. $c_i^r(t)$).

Let $\mathcal{S}$ be an ordered set including all base objects and clients, i.e., $B \cup \Pi$ ordered in some arbitrary way. For $S = \{bo_1, \ldots, bo_k, c_1, \ldots\} \subseteq \mathcal{S}$, $S^r(t)$ is the list of lists $bo_1^r(t), \ldots, bo_k^r(t), c_1^r(t), \ldots$ sorted according to their order in $\mathcal{S}$. A *block instance* $b \in S^r(t)$ is a triple $\langle i, j, e\rangle$ so that e is stored in the j^{th} position in the i^{th} list in S. We refer to the block contents as $b.e$.

Storage cost.

We count the number of bits stored in blocks in base objects as well as in clients, and neglect meta-data size. Note that oracle states are not counted as part of the storage cost, since we wish to measure the additional space required for making the data available for shared access, beyond its (trivial) existence at its sources and readers.

Definition 2 (Storage Cost)**.** The *storage cost at time t in a run r* is $\Sigma_{b \in \mathcal{S}^r(t)} |b.e|$. The *storage cost of an algorithm* A is the maximum storage cost at any point t in any run r of A.

Assumptions.

To make sure that the encoding does not leak information using block sizes, we assume *symmetry*, in the sense that output block sizes do not depend on input values. (Otherwise, we could for example, represent three values 0, 1, and 10 using a single coded block e_1 of size at most 1 bit by having $|e_1| = 0$ encode 10). Formally:

Definition 3 (Symmetric Encoding)**.** An encoding function $\mathbb{E}$ is *symmetric* if for every $v, v' \in \mathbb{V}$ and for all $i \in \mathbb{N}$, $|\mathbb{E}(v, i)| = |\mathbb{E}(v', i)|$. We denote $size(i) \triangleq |\mathbb{E}(v, i)|$.

Note that different block numbers (of all values) may have different sizes.

We next state our assumption that the storage treats the coding as a black-box. First, we define the notion of a source function, which we shall use to prohibit generation of code blocks by any source other than $oracle_{\mathbb{E}}$:

Definition 4 (Source Function)**.** A function is a *source function* for a run r if it maps every (b, t) s.t. $b \in \mathcal{S}^r(t)$ to a pair $\langle w, i\rangle$ s.t. $b.e$ was returned by $get(i)$ in $oracle_{\mathbb{E}}(w)$.

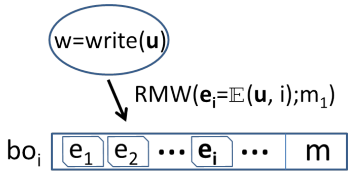
We use a source function to trace blocks in the storage to operations that produced them. To capture the restriction that the algorithm's decision what to store does not rely on block contents, we stipulate that we can replace the value written by a write operation w in a run r by an arbitrary value v, yielding the same sequence of states and actions, except that all stored block instances whose source is $\langle w, i\rangle$ are replaced with $\mathbb{E}(v, i)$. For clarity, we refer to the operation as w in both runs (see Figure 3).

Definition 5 (Black-Box Coding)**.** An algorithm A is *black-box coding* if for every run r there is a source function $source^r$ s.t. for every $w = write(u)$ operation in r, $\forall u \in \mathbb{V}$, there is run r_v satisfying the following:

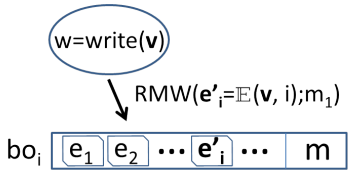
1. r_v has the same sequences of invocations and returns as r except that $w = write(v)$ (possibly with no change) and return values of read operations may be different; and

2. client and base object states at every time t in r_v are the same as at time t in r except that the contents of every $b \in \mathcal{S}^r(t)$ s.t. $source^r(b, t) = \langle w, i\rangle$ for some i is replaced by $e' = \mathbb{E}(v, i)$.

In the following, we will only consider source functions satisfying Definition 5. In case multiple such source functions for r exist, we fix an arbitrary one and refer to it as $source^r$.



(a) Run r



(b) Run r_v

Figure 3: Black-box coding. Runs r and r_v have the same trace except that write w is invoked with u in r and with v in r_v; and each base object bo_i's state (blocks and meta-data) is identical at all times in both runs, except that blocks produced by w's oracle in r are replaced in r_v by the corresponding blocks of v.

4. RELATED WORK

Our model captures numerous existing distributed storage algorithms, including ones that use replication [4], and erasure codes [3, 5, 11, 7, 10, 12]. We note that some of them report a storage cost below $O(cD)$ [3, 5, 18, 10]. This is sometimes achieved by assuming periods of synchrony [3]. Other works shift the cost from storage nodes to the network and keep unbounded information in channels [10, 5]. However, since we define parameters and responses of pending RMWs to be part of clients' and base objects' states, information in channels is counted in our storage cost model and hence these algorithms are subject to our bound. The only non-black-box storage algorithm we are aware of is [18], where multiple values are encoded jointly, saving space, but also forfeiting regular register semantics. It is as of now unclear whether lifting the black-box assumption suffices in order to circumvent our result.

An earlier version of this work [17] showed a special case of the result in this paper for infinite concurrency. Cadambe et al. [6] prove closely related lower bounds for coded storage algorithms. First, they show that asynchronous fault-tolerant storage algorithms require strictly more storage than synchronous erasure-coded algorithms. Second, similarly to this paper, they extend the result given in our earlier version to show that the storage cost must grow linearly with $\min(f, c)$, but their result is proven under a different set of assumptions than ours. In particular, while both papers make certain "black box" assumptions about the storage, [6] does not rule out joint coding as we do, but instead restricts protocol actions in a way that forbids them from depending on a written value in more than one communication round; this affords the protocol more freedom than our model in one communication round, and less freedom in all other rounds. On the face of it, the two sets of assumptions appear to be incomparable, though they both achieve the same end result, as we discuss in Section 7 below. Additionally, our bound allows algorithms to use unbounded (data-independent) meta-data and is proven for lock-free register emulations, whereas the bound in [6] includes meta-data and is shown for wait-free registers.

Another tradeoff between the number of writers and space complexity of reliable register emulations has been recently studied in [8]. Inspired by our adversary structure, they show that the number of fault-prone read/write registers needed to emulate a reliable multi-writer register grows linearly with the number of clients that can write to the register (even in sequential runs). Here, on the other hand, we consider storage nodes supporting fully general read-modify-write, for which that lower bound does not apply.

The challenge of providing a lower bound on stored data when meta-data is potentially unbounded was also previously addressed in the context of byzantine storage [9]. That paper has shown that certain storage algorithms cannot be "amnesic", i.e., cannot "forget" values written to them. Like our black-box assumption, the notion of amnesia was defined in terms of runs. However, it did not yield explicit bound on storage cost.

5. STORAGE LOWER BOUND

We now show a lower bound of $O(min(f, c) \cdot D)$ bits on the storage cost of any lock-free algorithm that uses symmetric black-box coding to simulate a regular register:

Theorem 1. *Consider a lock-free algorithm A that uses symmetric black-box coding to simulate a regular register. The storage cost of A is $\Omega(min(f, c) \cdot D)$.*

For the sake of our proof, we quantify the number of bits in blocks contributed by client c_i's operation w to base objects and clients other than c_i.

Definition 6. Let $S \subset \mathcal{S}$, and consider a time t and an operation w by client c_j in a run r. We define $S^r(t, w) \triangleq \{i \in \mathbb{N} \mid \exists b \in (S \setminus \{c_j\})^r(t)\colon\ source^r(b, t) = \langle w, i\rangle\}$, and $||S^r(t, w)|| \triangleq \Sigma_{i \in S^r(t,w)} size(i)$.

For $I \subseteq \mathbb{N}$, we say that two values $v' \neq v''$ in $\mathbb{V}$ are *I-colliding* if $\forall i \in I$, $\mathbb{E}(v', i) = \mathbb{E}(v'', i)$. We next use the pigeonhole argument and the assumption of symmetric black-box coding in order to show that write operations cannot return until some write stores enough bits in different blocks in every set of $n - f$ base objects.

Claim 1. *Let w be a write operation invoked in a run r of A, and t be a point in r. Consider a set of values $U \subset \mathbb{V}$, $|U| < 2^{D-1}$, and a set of base objects $S \subset \mathcal{S}$. If $||S^r(t, w)|| < D$, then there are two $S^r(t, w)$-colliding values $u \neq u'$ in $\mathbb{V} \setminus U$.*

Proof. Since $|\mathbb{V} \setminus U| > 2^{D-1}$ and $||S^r(t, w)|| < D$, the claim follows from the pigeonhole argument. □

Lemma 1. *Consider a run r of algorithm A that begins with the invocation of c concurrent write operations. Let S be a set of at least $n - f$ base objects and assume that at every time t in r for every operation w in r, $||S^r(t, w)|| < D$. Then no write operation returns in r.*

Proof. Let $W_{ops} = \{w_1, \ldots, w_c\}$ be the set of c concurrent writes invoked in r. Assume by contradiction that there exists a complete write in W_{ops}. Let w be the first such write, and t be the time when it returns. Next we inductively build a sequence of sets of values $U_0, U_1, \ldots, U_c$, where $|U_i| = i$:

- $U_0 = \{\}$
- $\forall i \in \{0, \ldots, c-1\}$, we use U_i to build U_{i+1}. By the lemma premise, $||S^r(t, w_{i+1})|| < D$. Now since $|U_i| < c < 2^{D-1}$, by Claim 1, there are two $S^r(t, w_{i+1})$-*colliding* values $u_{w_{i+1}} \neq u'_{w_{i+1}}$ in $\mathbb{V} \setminus U_i$. We let $U_{i+1} = U_i \cup \{u_{w_{i+1}}\}$.

The set U_c contains exactly c (different) values s.t. for every operation $w_i \in W_{ops}$ there is a value $u_{w_i} \in U_c$ that has a $S^r(t, w_i)$-*colliding* value $u'_{w_i} \in \mathbb{V}$. By applying Definition 5 (c times), there is a run r' that begins with the invocation of c concurrent write operations, in which every operation $w_i \in W_{ops}$ writes u_{w_i} s.t. w returns at time t, and for every operation $w_i \in W_{ops}$, $S^r(t, w_i) = S^{r'}(t, w_i)$. Next, let clients with outstanding operations and all base objects in $B \setminus S$ fail at time t in r' (note that by assumption $|S| \geq n - f$, so $|B \setminus S| \leq f$), and let some client c_j invoke a solo read operation at time $t+1$. By lock-freedom, c_j's read operation completes, and by regularity, it returns a value $u \in U_c$ at some time $t' > t$.

Let w' be the operation that writes u in r'. Since u has a $S^r(t, w')$-*colliding* value u' and since $S^r(t, w') = S^{r'}(t, w')$, u and u' are $S^{r'}(t, w')$-*colliding*. By Definition 5, there is a run r'' with the same operations as in r' except that w' writes u' (instead of u) s.t. every client's and base object's state at time t in r' is identical to its state at time t in r'' (note that clients with outstanding operations and all base objects in $B \setminus S$ fail at time t) except that for every block instance $b \in \mathcal{S}^{r'}(t)$ s.t. $source^{r'}(b, t) = \langle w', i \rangle$, $b.e$ is replaced with a block $\mathbb{E}(u', i)$. In particular, states of base objects in S at time t are identical to their states at time t in r' except that for every block instance $b \in S^{r'}(t)$ s.t. $source^{r'}(b, t) = \langle w', i \rangle$, $b.e$ is replaced with a block $\mathbb{E}(u', i)$.

Now since u and u' are $S_{r'}(t, w')$-*colliding*, states of base objects in S at time t in r'' are identical to their states at time t in r'. In addition, since clients with outstanding operations and all base objects in $B \setminus S$ fail at time t, the solo reader c_j cannot distinguish between r' and r'', and thus, it pushes the same blocks to its oracle and calls *done* with the same number in r'' as in r', and therefore, its read operation returns u at time t'' in run r''. However, since the clients invoke write operations with different values in r', u is not written in r''. A contradiction to regularity.

□

Having shown a condition under which write operations cannot complete, we define an (unfair) adversary behavior that takes advantage of this in order to prevent progress. We introduce some notation, and then use it in order to define the adversary. We define a parameter $0 < \ell \leq D$, and for any time t in a run r of algorithm A we define the following sets, as illustrated in Figure 4. For convenience, from now on we omit the superscript r.

- $C(t)$: the set of outstanding write operations at time t.
- $C^-_\ell(t) = \{w \in C(t) \mid ||\mathcal{S}(t, w)|| \leq D - \ell\}$: The set of write operations each of which has at most $D-\ell$ bits in blocks, produced by its oracle with different numbers, in the storage (excluding the client performing it) at time t.
- $C^+_\ell(t) = C(t) \setminus C^-_\ell(t)$.
- $F_\ell(t) = \{bo_i \in B \mid \Sigma_{b \in \{bo_i\}(t)} |b.e| \geq \ell\}$. Base objects that store blocks that consist (together) of more than ℓ bits at time t. These are base objects that we will "freeze" in our counter-example because they are already "full", i.e., consume enough space for our lower bound.

We fix the parameter ℓ throughout the proof and omit subscript ℓ from the notation. The next observation on storage cost immediately follows from the definitions.

Observation 1. *At any point t in every run r of A, the storage cost is at least $|C^+(t)|(D - \ell + 1)$.*

We next define a particular adversary behavior that schedules actions in a way that prevents progress. Note that the adversary controls the scheduling of client actions and RMW responses.

Definition 7. (Ad) At any time t, Ad schedules an action as follows:

1. If there is a pending RMW on a base object in $B \setminus F(t)$ by a client performing an operation in $C^-(t)$, then choose the longest pending of these RMWs, allow it to take effect on the corresponding base object, and schedule its response.
2. Else, choose in a fair order an operation by a client $c_i \in \Pi$ and schedule its action (trigger RMW, call its oracle, get response from its oracle, or return), without allowing it to affect the base object yet. By fair order we mean any order in which every client is chosen infinitely often (e.g., $c_1, c_1, c_2, c_1, c_2, c_3 \ldots$).

In other words, Ad delays RMWs triggered by operations in $C^+(t)$ (for which the storage already holds $D - \ell$ bits) as well as RMWs on "frozen" base objects in $F(t)$ (which store at least ℓ bits), and fairly schedules all other actions. We demonstrate Ad's behavior in Figure 4. Though this behavior may be unfair, in every infinite run of Ad, every correct client gets infinitely many opportunities to take steps. We use Ad to build an unfair run with no progress (no write returns), and then build an indistinguishable fair run to contradict lock-freedom. The following observation immediately follows from the adversary's freezing of base objects in F.

Observation 2. *Assume run r of algorithm A in which the environment behaves like Ad. For each base object bo, if $bo \in F(t)$ at some time t, then $bo \in F(t')$ for all $t' > t$ in r.*

Another consequence of Ad's behavior is captured by the following:

Lemma 2. *Consider a run r of algorithm A. If the adversary behaves like Ad, then for every time t and for every write operation w in r, $||(\mathcal{S} \setminus F(t))(t, w)|| < D$.*

Proof. Assume by way of contradiction that there is time t and write operation w performed by client c_j s.t. $||(\mathcal{S} \setminus F(t))(t, w)|| \geq D$. The definition of $(\mathcal{S} \setminus F(t))(t, w)$ takes into account only blocks returned by w's oracle that are stored outside of $c_j(t)$. Thus, w triggered at least one RMW that has a matching response before time t in r. Let $t' \leq t$ be the

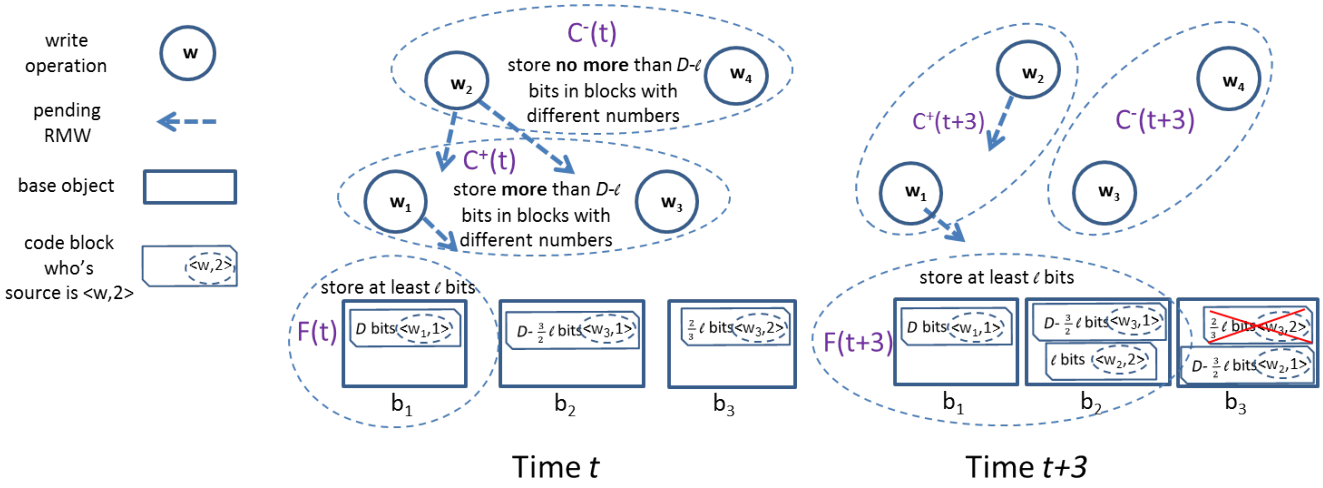


Figure 4: Example scenario in run of a storage algorithm with adversary Ad. In this example, $2D/5 < \ell < D$. At time t, only w_2 and w_4 are in $C^-(t)$, where w_4 has no pending RMWs and w_2 has one triggered RMW on $b_1 \in F(t)$ and one triggered RMW on $b_3 \notin F(t)$. Therefore, by the first rule, Ad schedules the response on the RMW triggered by w_2 on b_3. In this example w_2 overwrites w_3's block in b_3, thus w_3 moves from C^+ to C^-. Then, at time $t+1$, no response can be scheduled by rule 1 (no operation in $C^-(t+1)$ has a pending RMW on a base object in $N \setminus F(t+1)$), so by rule 2, Ad chooses w_2 and lets it trigger an RMW on base object b_2. Now since w_2 is the only operation that has a pending RMW on a base object not in $F(t+2)$, Ad schedules the response on the RMW triggered by w_2 on b_2 at time $t+2$. In this example w_2 adds a block with ℓ bits to b_2. Thus, c_2 is included in $C^+(t+3)$. In addition, b_2 stores more than ℓ bits at time $t+3$, so it belongs to $F(t+3)$.

time when the last RMW triggered by w responded, and denote this RMW by rmw and the base object on which rmw was triggered by bo. By Ad, $w \in C^-(t'-1)$, and therefore, by definition, $||(\mathcal{S} \setminus F(t'-1))(t'-1, w)|| \leq D - \ell$. Now consider two cases:

- First, rmw adds blocks (possibly overwriting other blocks) with less than ℓ bits to bo. In this case, since bo is the only storage component that changed at time t', $|(\mathcal{S} \setminus F(t'))(t', w)| < D$.
- Second, rmw adds blocks (possibly overwriting other blocks) with at least ℓ bits to bo. In this case, $bo \in F(t')$. Now since $||(\mathcal{S} \setminus F(t'-1))(t'-1, w)|| \leq D$, by Observation 2, $F(t'-1) \subseteq F(t')$, and given $bo \in F(t')$ and it is the only storage component that changed at time t', we get $||(\mathcal{S} \setminus F(t'))(t', w)|| \leq ||(\mathcal{S} \setminus F(t'-1))(t'-1, w)|| < D$.

So far we showed that $||(\mathcal{S} \setminus F(t'))(t', w)|| < D$. By Observation 2, and since no RMW by w takes effect after time t', $(\mathcal{S} \setminus F(t''))(t'', w) \subseteq (\mathcal{S} \setminus F(t'))(t', w)$, $\forall t'' \geq t'$. Therefore, we get $||(\mathcal{S} \setminus F(t))(t, w)|| < D$. A contradiction. □

The next corollary uses Lemmas 1 and 2 in order to conclude that Ad can prevent progress of write operations.

Corollary 1. *Consider a run r of algorithm A that begins with the invocation of c concurrent write operations. If the adversary behaves like Ad and $|F(t)| \leq f$ for all t in r, then no write operation returns in r.*

Proof. By Lemma 2, for every time t for every write operation w in r, $||(\mathcal{S} \setminus F(t))(t, w)|| < D$. And since $B \subset \mathcal{S}$, for every time t for every write operation w in r, $||(B \setminus F(t))(t, w)|| < D$. Now since $|F(t)| \leq f$ for every time t in r, $|B \setminus F(t)| \geq n - f$. Therefore, by Lemma 1, no write operation returns in r. □

We have shown that Ad can prevent completion of write operations in algorithms that store ℓ bits in less than $f+1$ base objects. However, this does not directly imply a storage bound, since Ad is not fair. In the next lemma we close this gap by showing a fair run in which lock-freedom must be satisfied , i.e., operations invoked by correct clients must eventually complete, in order to blow up the storage. We show that for every algorithm, we can build a run where at some point the algorithm either stores ℓ bits in each of $f+1$ base objects (namely, $\exists t : |F(t)| > f$), or there are c concurrent operations each of which adds at least $D - \ell + 1$ bits to the storage cost (i.e., $|C^+(t)| = c$).

Lemma 3. *There is a run r of A and a time t in r when $|C^+(t)| = c$ or $|F(t)| > f$.*

Proof. Assume by way of contradiction that there is no such run of algorithm A. We first build a run r of A with c clients that concurrently write different values, in which the environment behaves like adversary Ad. By the contradiction assumption, $|C^+(t)| < c$ and $|F(t)| \leq f$ for all t in r. We start with the invocation of c concurrent write operations, and allow the run to proceed indefinitely according to Ad. We say that a client c, which performs write operation w, is *stuck* in r if there is a time t in r s.t. for all $t' \geq t$, $w \in C^+(t')$ (and so no RMWs triggered by c take effect after time t). By Observation 2 and the assumption that $|F(t)| \leq f$ for all t, there is a time t_1 in r s.t. for every time $t_2 \geq t_1$, $F(t_1) = F(t_2)$.

Now we build a run r' that is identical to r but every base object $bo \in F(t_1)$ fails at time t_1 ($|F(t_1)| \leq f$), and every

stuck client fails after its last RMW takes effect. Since by *Ad*, RMWs do not take effect on base objects in $F(t_1)$ after time t_1, runs r and r' are indistinguishable to all correct clients and base objects. Now notice that by *Ad*'s behavior, each correct client in r' gets infinitely many opportunities to trigger RMWs. In addition, since (1) for every correct client c_i in r' there are infinitely many times t when $c_i \in C^-(t)$, (2) *Ad* picks responses from base objects not in $F(t)$ in the order they are triggered, and (3) there are no correct base objects in $F(t')$ for all $t' > t_1$, every RMW triggered by a correct client on a correct base object has a matching response in r'. Therefore, run r' is fair.

By the contradiction assumption $|C^+(t)| < c$ for all t in r. Therefore, there is at least one client that is not stuck in r, and thus, there is at least one client that is correct in r'. Hence, by lock-freedom, some client eventually completes its write operation in r'. Now since r and r' are indistinguishable to all clients that are correct in both, the same is true in r. However, by Corollary 1, no write operation completes in r. A contradiction.

□

So far we have shown that every algorithm has a run where at some point either ℓ bits are stored in $f+1$ base objects, or there are c concurrent operations each of which adds at least $D - \ell + 1$ bits to the storage cost. We now combine this result with Observation 1 to conclude our lower bound:

Proof (Theorem 1). Let $\ell = D/2$. By Lemma 3, there is a run r of A and a time t in r when $|C^+(t)| = c$ or $|F(t)| > f$. If $|F(t)| > f$, then the storage cost at time t in r is $(f+1)\ell = (f+1)D/2 = \Omega(fD)$. Otherwise, $|C^+(t)| = c$, and so by Observation 1, the storage cost at time t in r is at least $c(D-\ell) = cD/2 = \Omega(cD)$. The theorem follows.

□

By picking $\ell = D$, we get a second conclusion from Lemma 3 and Observation 1. The following corollary proves that any coding scheme short of full replication must exhibit storage growth linear in the concurrency.

Corollary 2. *The storage cost of any algorithm that uses a black-box coding scheme to simulate a regular lock-free register, and does not store D bits (enough to represent a full replica) in $f+1$ base objects, grows linearly with the concurrency.*

6. ADAPTIVE REGULAR REGISTER

We present a storage algorithm that combines full replication with erasure coding in order to achieve the advantages of both. A k-of-n erasure code takes a value from $\mathbb{V}$ and produces a set S of n blocks from $\mathcal{E}$ s.t. the value can be restored from any subset of S that contains no less than k different blocks. We assume that the size of each block is D/k. $Oracle_{\mathbb{E}}$ and $Oracle_{\mathbb{D}}$ are encapsulated by two functions *encode* and *decode*, respectively: *encode* gets a value $v \in \mathbb{V}$ and returns a set of n ordered elements $W = \{\langle e_1, 1\rangle, \ldots, \langle e_n, n\rangle\}$, where $e_1, \ldots, e_n \in \mathcal{E}$, and *decode* gets a set $W' \subset \mathcal{E} \times \mathbb{N}$ and returns $v' \in \mathbb{V}$ s.t. if $|W'| \geq k$ and $W' \subseteq W$, then $v = v'$. We use $k = n - 2f$. Note that when $k = 1$, we get full replication.

The main idea behind our algorithm is to have base objects store blocks from at most k different *writes*, and then turn to store full replicas. Our algorithm satisfies strong regularity and FW-termination, which is a stronger liveness property than lock-freedom (see Appendix A). In the technical report [17], we prove the following:

Theorem 2. *There is an FW-terminating algorithm that simulates a regular register, whose storage cost is $min((c+1)(2f+k)D/k, (2f+k)2D)$ bits. Moreover, in a run with a finite number of writes, if all the writers are correct, the storage is eventually reduced to $(2f+k)D/k$ bits.*

Notice that k is a parameter of the algorithm, and if we pick $k = f$, then asymptotically the storage cost of our algorithm is $O(min(cD, fD)) = O(min(c, f) \cdot D)$.

The algorithm's pseudocode appears in Algorithms 1-3. The algorithm uses a set of n shared base objects $bo_1, \ldots, bo_n$ each of which holds three fields V_p, V_f, and $storedTS$.

Algorithm 1 Definitions.

1: $TimeStamps = \mathbb{N} \times \Pi$, with selectors num and c, ordered lexicographically.
2: $Pieces = (\mathcal{E} \times \mathbb{N})$
3: $Chunks = Pieces \times TimeStamps$, with selectors val, ts
4: $encode : \mathbb{V} \to 2^{\mathcal{E} \times \{1,2,\ldots,n\}}$, $decode : 2^{\mathcal{E} \times \{1,2,\ldots,n\}} \to \mathbb{V}$ s.t. $\forall v \in \mathbb{V}$, $encode(v) = \{\langle *, 1\rangle, \ldots, \langle *, n\rangle\} \wedge \forall W \in 2^{\mathcal{E} \times \mathbb{N}}$, if $W \subseteq encode(v) \wedge |W| \geq k$, then $decode(W) = v$

5: **base objects:**
6: $\forall i \in \{1, \ldots, n\}$, $bo_i = \langle storedTS, V_p, V_f\rangle$ s.t. $V_f, V_p \subset Chunks$, and $storedTS \in TimeStamps$, initially $\langle\langle 0, 0\rangle, \{\langle\langle 0, 0\rangle, \langle v_{0_i}, i\rangle\rangle\}, \{\}\rangle$.

The V_p field holds a set of timestamped code blocks so that the i^{th} block of a value can be stored in the V_p field of object bo_i. The V_f field stores a timestamped replica of a *single* value, (represented as a set of k code blocks). And $storedTS$ holds a timestamp, as explained below.

Write operation and storage efficiency.

The write operation (lines 3–14) consists of 3 sequentially executed rounds: *read timestamp*, *update*, and *garbage collection*; and, the read consists of one or more sequentially executed *read* rounds. At each round, the client invokes RMWs on all base objects in parallel, and awaits responses from at least $n - f$ base objects. The read rounds of both write and read rely on the readValue routine (lines 21–28) to collect the contents of the V_p and V_f, fields from $n - f$ base objects, as well as to determine the highest $storedTS$ known to these objects. The implementations of the update and garbage collection rounds are given by the update (lines 29–36) and GC (lines 37–42) routines, respectively.

The write implementation starts by encoding v into k code blocks (line 4) and invoking the read round where the client uses the combined contents of the V_p, V_f and $storedTS$ fields returned by readValue to determine the timestamp ts to be stored alongside v's code blocks on the base object; ts is set to be higher than all returned timestamps thus ensuring that the order of the timestamps associated with the stored values is compatible with the order of their corresponding writes, (which is essential for regularity).

The client then proceeds to the update round where it attempts to store the i^{th} code block $\langle e, i\rangle$ of v in $bo_i.V_p$ if the size of $bo_i.V_p$ is less than k (lines 33), or its full replica in $bo_i.V_f$ if ts is higher than the timestamp associated with the

value currently stored in $bo_i.V_f$ (line 35). Storing $\langle e, i\rangle$ in $bo_i.V_p$ coincides with an attempt to reduce its size by removing stale code blocks of values whose timestamps are smaller than $storedTS$ (line 33). This guarantees that the size of V_p never exceeds the number of concurrent writes, which is a key for achieving our adaptive storage bound. Lastly, the client updates $bo_i.storedTS$ so as its new value is at least as high as the one returned by the readValue routine. This allows the timestamp associated with the latest complete update to propagate to the base object being written, in order to prevent future writes of old blocks into this base object.

In the write's garbage collection round, the client attempts to further reduce the storage usage by (1) removing all code blocks associated with timestamps lower than ts from both $bo_i.V_p$ and $bo_i.V_f$ (lines 38–39), and (2) replacing a full replica (if it exists) of its written value v in $bo_i.V_f$ with its i^{th} code block $\langle e, i\rangle$ (line 41). It is safe to remove the full replica and values with older timestamps at this point, since once the update round has completed, it is ensured that the written value or a newer written value is restoreable from any $n - f$ base objects. This mechanism ensures that all code blocks except the ones comprising the value written with the highest timestamp are eventually removed from all objects' V_p and V_f sets, which reduces the storage to a minimum in runs with finitely many writes, which all complete. The garbage collection round also updates the $bo_i.storedTS$ field to ensure its value is at least as high as ts.

Algorithm 2 regular register emulation. Algorithm for client c_j.

```
 1: local variables:
 2:     storedTS, ts ∈ TimeStamp, WriteSet ∈ Pieces

 3: operation Write(v) do
 4:     WriteSet ← encode(v)
        ▷ round 1: read timestamps
 5:     ⟨storedTS, ReadSet⟩ ← readValue()
 6:     tmp ← max(storedTS.num,
        max{tmp′ | ⟨⟨tmp′, ∗⟩, ∗⟩ ∈ ReadSet})
 7:     ts ← ⟨n + 1, j⟩
        ▷ round 2: update
 8:     || for i = 1 to n
 9:         update(bo_i, WriteSet, ts, storedTS, i)
10:     wait for n − f responses
        ▷ round 3: garbage collect
11:     || for i = 1 to n
12:         GC(bo_i, WriteSet, ts, i)
13:     wait for n − f responses
14:     return "ok"

15: operation Read() do
16:     ⟨storedTS, ReadSet⟩ ← readValue()
17:     while ∄ts ≥ storedTS s.t.
        |{⟨ts, v⟩ | ⟨ts, v⟩ ∈ ReadSet}| ≥ k do
18:         ⟨storedTS, ReadSet⟩ ← readValue()
19:     ts′ ← max_{ts≥storedTS} (|{⟨ts, v⟩ | ⟨ts, v⟩ ∈ ReadSet}| ≥ k)
20:     return decode({v | ⟨ts′, v⟩ ∈ ReadSet})
```

Key Invariant and read operation.

The write implementation described above guarantees the following key invariant: at all times, a value written by either the latest complete write or a newer write is available from every set consisting of at least $n-f$ base objects (either in the form of k code blocks in the objects' V_p fields, or in full from one of their V_f fields). Therefore, a read will always be able to reconstruct the latest completely written or a newer value provided it can successfully retrieve k matching blocks of this value. However, a read round may sample different base objects at different times (that is, it does not necessarily obtain an atomic snapshot of the base objects), and the number of blocks stored in V_p is bounded. Thus, the read may be unable to see k matching blocks of any single new value, as long as new values continue to be written concurrently with the read.

Nevertheless, for FW-Termination, the reads are only required to return in runs where a finite number of writes are invoked. Our implementation of read (lines 15–20) proceeds by invoking consecutive rounds of RMWs on the base objects via the readValue routine. After each round, the reader examines the collection of returned values and timestamps to determine if any value has k code blocks and is also associated with a timestamp that is at least as high as $storedTS$ (line 17). If any such value is found, the one associated with the highest timestamp is returned (line 20). Otherwise, the reader proceeds to invoke another round of base object accesses. Note that returning values associated with older timestamps may violate regularity, since they may have been written earlier than the write with timestamp $storedTS$, which in turn may have completed before the read was invoked.

Algorithm 3 Functions used in regular register emulation.

```
21: procedure readValue()
22:     ReadSet ← {}, T ← {}
23:     || for i=1 to n
24:         tmp ← read(bo_i)
25:         ReadSet ← ReadSet ∪ tmp.V_f ∪ tmp.V_p
26:         T ← T ∪ {tmp.storedTS}
27:     wait for n − f responses
28:     return ⟨max(T), ReadSet⟩

29: update(bo, WriteSet, ts, storedTS, i) ≜
30:     if ts ≤ bo.storedTS
31:         return
32:     if |bo.V_p| < k
            ▷ write a piece and remove old pieces
33:         bo.V_p ← bo.V_p \ {⟨ts′, v⟩ ∈ bo.V_p | ts′ < storedTS}
            ∪{⟨ts, ⟨e, i⟩⟩ | ⟨e, i⟩ ∈ WriteSet}
34:     else if bo.V_f = {} ∨ ∃ts′ < ts : ⟨ts′, ∗⟩ ∈ bo.V_f
            ▷ write a piece and remove old pieces
35:         bo.V_f ← {⟨ts, ⟨e, j⟩⟩ | ⟨e, j⟩ ∈ WriteSet
            ∧j ∈ {1, . . . , k}}
36:     bo.storedTS ← max(bo.storedTS, storedTS)

37: GC(bo, WriteSet, ts, i) ≜
        ▷ keep only new pieces
38:     bo.V_p ← {⟨ts′, v⟩ ∈ bo.V_p | ts′ ≥ ts}
39:     bo.V_f ← {⟨ts′, v⟩ ∈ bo.V_f | ts′ ≥ ts}
40:     if ⟨ts, ∗⟩ ∈ bo.V_f
            ▷ V_f holds a full replica of my write
            ▷ V_f keep only one piece of it
41:         bo.V_f ← {⟨ts, ⟨e, i⟩⟩ | ⟨e, i⟩ ∈ WriteSet}
42:     bo.storedTS ← max(bo.storedTS, ts)
```

7. DISCUSSION

We studied the inherent space requirements of reliable storage in asynchronous distributed settings. We proved an asymptotic bound of $\Omega(min(f, c) \cdot D)$ for any storage al-

gorithm using a symmetric black-box coding scheme, which produces code blocks of values independently of other values. We then presented an algorithm that combines replication and erasure codes, whose storage cost is $O(min(f, c) \cdot D)$.

Our work leaves open questions for future work. First, it is unclear whether the same lower bound still applies when stored bits are allowed to depend on multiple concurrent write values. The main requirement for extending our proof to general coding is a model that correctly accounts for the information stored in the base objects and clients when the clients code jointly. Our black-box assumption rules out such joint coding. Whereas in principle, [6] allows stored information to depend on multiple input values, their assumption that only one round of the protocol depends on the written value essentially forces clients to "forget" the value they used in such joint coding. For example, if the algorithm stores $v_1 + v_2$ instead of either v_1 or v_2, it cannot reproduce the original values, rendering such joint coding useless. Second, while asymptotically optimal, the constants in our bound are not tight, and it could be interesting to close this gap. Finally, we believe that our model and adversary definitions can yield additional lower bounds.

Acknowledgements

We thank Dahlia Malkhi, Yoram Moses, and Rotem Oshman for insightful comments.

References

[1] Ittai Abraham, Gregory Chockler, Idit Keidar, and Dahlia Malkhi. Byzantine disk paxos: optimal resilience with byzantine shared memory. *Distributed Computing*, 18(5), 2006.

[2] Yehuda Afek, Michael Merritt, and Gadi Taubenfeld. Benign failure models for shared memory. In *Distributed Algorithms*. Springer, 1993.

[3] Marcos Kawazoe Aguilera, Ramaprabhu Janakiraman, and Lihao Xu. Using erasure codes efficiently for storage in a distributed system. In *Dependable Systems and Networks, 2005. DSN 2005. Proceedings. International Conference on*. IEEE, 2005.

[4] Hagit Attiya, Amotz Bar-Noy, and Danny Dolev. Sharing memory robustly in message-passing systems. *Journal of the ACM (JACM)*, 42(1), 1995.

[5] Christian Cachin and Stefano Tessaro. Optimal resilience for erasure-coded byzantine distributed storage. In *Dependable Systems and Networks, 2006. DSN 2006. International Conference on*. IEEE, 2006.

[6] Viveck Cadambe, Zhiying Wang, and Nancy Lynch. Information-theoretic lower bounds on the storage cost of shared memory emulation. In *PODC*, 2016.

[7] Viveck R Cadambe, Nancy Lynch, Muriel Medard, and Peter Musial. A coded shared atomic memory algorithm for message passing architectures. In *Network Computing and Applications (NCA), 2014 IEEE 13th International Symposium on*. IEEE, 2014.

[8] Gregory Chockler, Dan Dobre, Alexander Shraer, and Alexander Spiegelman. Space bounds for reliable multi-writer data store: Inherent cost of read/write primitives. *arXiv preprint arXiv:1508.03762*, 2015.

[9] Gregory Chockler, Rachid Guerraoui, and Idit Keidar. Amnesic distributed storage. In *Distributed Computing*. Springer, 2007.

[10] Partha Dutta, Rachid Guerraoui, and Ron R. Levy. Optimistic erasure-coded distributed storage. In *Proceedings of the 22Nd International Symposium on Distributed Computing*, DISC '08, Berlin, Heidelberg, 2008. Springer-Verlag.

[11] Garth R Goodson, Jay J Wylie, Gregory R Ganger, and Michael K Reiter. Efficient byzantine-tolerant erasure-coded storage. In *Dependable Systems and Networks, 2004 International Conference on*. IEEE, 2004.

[12] James Hendricks, Gregory R Ganger, and Michael K Reiter. Low-overhead byzantine fault-tolerant storage. In *ACM SIGOPS Operating Systems Review*, volume 41. ACM, 2007.

[13] Prasad Jayanti, Tushar Deepak Chandra, and Sam Toueg. Fault-tolerant wait-free shared objects. *Journal of the ACM (JACM)*, 45(3), 1998.

[14] Leslie Lamport. On interprocess communication. *Distributed computing*, 1(2), 1986.

[15] Heverson Borba Ribeiro and Emmanuelle Anceaume. Datacube: A p2p persistent data storage architecture based on hybrid redundancy schema. In *Parallel, Distributed and Network-Based Processing (PDP), 2010 18th Euromicro International Conference on*. IEEE, 2010.

[16] Cheng Shao, Jennifer L Welch, Evelyn Pierce, and Hyunyoung Lee. Multiwriter consistency conditions for shared memory registers. *SIAM Journal on Computing*, 40(1), 2011.

[17] Alexander Spiegelman, Yuval Cassuto, Gregory Chockler, and Idit Keidar. Space bounds for reliable storage: Fundamental limits of coding. *arXiv preprint arXiv:1507.05169*, 2015.

[18] Zhiying Wang and Viveck Cadambe. Multi-version coding in distributed storage. In *Information Theory (ISIT), 2014 IEEE International Symposium on*. IEEE, 2014.

APPENDIX

A. FORMAL DEFINITIONS

An *algorithm* defines the behavior of clients as deterministic state machines, where state transitions are associated with actions such as RMW trigger/response. A *configuration* is a mapping to states from system components, i.e., clients and base objects. An *initial configuration* is one where all components are in their initial states.

A *run* of algorithm A is a (finite or infinite) alternating sequence of configurations and actions, beginning with some initial configuration, such that configuration transitions occur according to A. For a run r, *trace(r)* is the subsequence of r consisting of all the operation invocation and returns in r. We use the notion of time t during a run r to refer to the configuration reached after the t^{th} action in r. A *run fragment* is a contiguous subsequence of a run starting and ending with a configuration. We assume that runs are *well-formed*, in that each client's first action is an invocation, and a client has at most one outstanding operation at any time.

We say that a base object or client is *faulty* in a run r if it fails any time in r, and otherwise, it is *correct*. A run is

fair if (1) for every RMW triggered by a correct client on a correct base object, there is eventually a matching response, (2) every correct client gets infinitely many opportunities to trigger RMWs.

Liveness There is a range of possible liveness conditions, which need to be satisfied in fair runs. A *wait-free* object is one that guarantees that every correct client's operation completes, regardless of the actions of other clients. A *lock-free* object guarantees progress: if at some point in a run there is an outstanding operation of a correct client, then *some* operation eventually completes. An *FW-terminating* [1] register is one that has wait-free *write* operations, and in addition, if there are finitely many *write* invocations in a run, then every *read* operation completes.

Safety In order to define regularity, we first introduce some terminology: Operation op_i *precedes* operation op_j in a run r, denoted $op_i \prec_r op_j$, if op_i's return occurs before op_j's invoke in r. Operations op_i and op_j are *concurrent* in a run r if neither one precedes the other. A run with no concurrent operations is *sequential*. Two runs are *equivalent* if every client performs the same sequence of operations in both, where operations that are outstanding in one can either be included in or excluded from the other. A *linearization* of a run r is an equivalent sequential run that preservers r's operation precedence relation and the object's sequential specification. The sequential specification for a register is as follows: A read returns the latest written value, or v_0 if none was written. A *write* w in a run r is *relevant* to a *read* rd in r [16] if $rd \not\prec_r w$; *rel-writes*(r, rd) is the set of all *writes* in r that are relevant to rd.

Following Lamport [14], we consider a hierarchy of safety notions. Lamport [14] defines *regular* and *safe* single-writer registers. Shao et al. [16] extend Lamport's notion of regularity to MWMR registers, and give four possible definitions. Here we use two of them. The first is the weakest definition, and we use it in our lower bound proof. The second, which we use for our algorithm, is the strongest definition that is satisfied by ABD [4] in case readers do not change the storage (no *write-back*):

A MWMR register is *weakly regular*, (called $MWRegWeak$ in [16]), if for every run r and *read* rd that returns in r, there exists a linearization of the subsequence of r consisting of rd and the writes in r. A MWMR register is *strongly regular*, (called $MWRegWO$ in [16]), if it satisfies weak regularity and the following condition: For all *reads* rd_1 and rd_2 that return in r, for all writes w_1 and w_2 in *rel-writes*$(r, rd_1) \cap$ *rel-writes*(r, rd_2), it holds that $w_1 \prec_{L_{rd_1}} w_2$ if and only if $w_1 \prec_{L_{rd_2}} w_2$.

We extend the safe register definition and say that a MWMR register is *strongly safe* if there exists a linearization σ_w of the subsequence of r consisting of the *write* operations in r, and for every *read* operation rd that has no concurrent *writes* in r, it is possible to add rd at some point in σ_w so as to obtain a linearization of the subsequence of r consisting