Abstract

We study the inherent space requirements of shared storage algorithms in asynchronous fault-prone systems. Previous works use codes to achieve a better storage cost than the well-known replication approach. However, a closer look reveals that they incur extra costs somewhere else: Some use unbounded storage in communication links, while others assume bounded concurrency or synchronous periods. We prove here that this is inherent, and indeed, if there is no bound on the concurrency level, then the storage cost of any reliable storage algorithm is at least $f + 1$ times the data size, where $f$ is the number of tolerated failures. We further present a technique for combining erasure-codes with full replication so as to obtain the best of both. We present a storage algorithm whose storage cost is close to the lower bound in the worst case, and adapts to the concurrency level.
1 Introduction

We reason about the storage space required for emulating reliable shared storage over fault-prone nodes. The traditional approach to building such storage stores full replicas of the data in each node [4]. This approach entails a fixed storage cost equal to the size of the data times the number of nodes, regardless of the level of concurrency.

Recently, there is an active area of research of employing codes, and in particular erasure codes, in distributed algorithms with the goal of reducing the storage cost [3, 5, 8, 6, 12, 7]. But when we look at these works closely, we find that in all asynchronous solutions, extra costs are hidden somewhere. Some keep an unbounded number of versions [8], or as many as the allowed level of concurrency [6]. Others keep unbounded information in channels [7, 5]. While others assume periods of synchrony [3] or allow returning obsolete values [12].

To provide intuition about erasure-coded reliable storage algorithms, we give in Section 3 a simple space-efficient solution that only guarantees safe semantics [10], which are too weak to be of practical use. We use this example to illustrate the challenges that have led algorithms that provide stronger semantics to store many versions of the coded data.

Then, in Section 4 we prove that this is inherent: any lock-free algorithm that simulates reliable storage in an asynchronous system where \( f \) storage nodes can fail must sometimes store \( f + 1 \) full replicas of written data, or its storage cost can grow without bound. Specifically, our bound applies to any fault-tolerant implementation of a multi-writer multi-reader (MWMR) register that satisfies at least weak regularity, a safety notion weaker than linearizability.

We prove our result for the fault-prone shared memory model [2, 11] in order to avoid reasoning explicitly about channels. The same bound applies to message passing systems if we limit the capacity of communication channels. For the sake of our proof, we define a specific adversary behavior, which makes the proof fairly compact.

Understanding the inherent storage cost limitation that stems from our lower bound, and in particular, the fact that, under high concurrency, nodes have to keep full replicas, leads us to develop an adaptive approach that combines the advantages of full replication and coding. We present in Section 5 an algorithm that simulates an FW-Terminating [1] strongly regular [11] MWMR register, whose storage requirement is close to the storage limitation in the worst case, and uses less storage in runs with low concurrency. The algorithm does not assume any a priori bound on concurrency; rather, it uses erasure codes when concurrency is low and switches to replication when it is high.

Finally, we believe that our work is only a first effort to combine erasure coding with replication in order to achieve adaptive storage costs. We conclude in Section 6 with some thoughts about directions for future work.

2 Preliminaries

2.1 Model

We consider an asynchronous fault-prone shared memory system [2, 11] consisting of set \( N = \{b_0, \ldots, b_n\} \) of base objects supporting arbitrary atomic read-modify-write (RMW) access by clients from some finite set \( \Pi \). Any \( f \) base objects and any number of clients may fail by crashing, for some predefined \( f < n/2 \). We study algorithms that emulate a shared object to a set of clients.

Clients interact with the emulated object via high-level operations. 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 consists of 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 \( b_i \in N \) and receives responses in return. We model the state of each \( b_i \in N \) as changing, 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.
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 $A$. We use the notion of time $t$ during a run $r$ to refer to the configuration incurred after the $t^{th}$ action in $r$. A run fragment is a contiguous subsequence of a run.

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. We again use different terminology to distinguish incomplete invocations to the high-level service from incomplete RMWs triggered on base objects and refer to the former as outstanding operations and to the latter as pending RMWs.

Operation $op_i$ precedes operation $op_j$ in a run $r$, denoted $op_i \prec_r op_j$, if $op_i$’s response occurs before $op_j$’s invoke in $r$. Operations $op_i$ and $op_j$ are concurrent in a run $r$, if neither precedes the other. A run with no concurrent operations is sequential.

### 2.2 Storage service definitions

We study emulations of an MWMR register, which stores a value $v$ from a domain $\mathbb{V}$, and offers an interface for invoking read and write operations. Initially, the register holds some distinguished initial value $v_0 \in \mathbb{V}$. The sequential specification for this service is as follows: A read returns the latest written value, or $v_0$ if none was written.

The storage resources consumed by the MWMR register emulations discussed herein are measured in units of bits. For constructive algorithmic results, bits are stored in base objects following writes triggered by clients, and correctness lies upon the existence of a decoding algorithm that can recover $v \in \mathbb{V}$ from the bits available to the reader. The common examples for such decoding algorithms are 1) the trivial decoder mapping $D = \log_2 |\mathbb{V}|$ bits to the value $v$ using the standard binary representation, as in the case of replication; and 2) an erasure-code decoder mapping a set of $D$ or more code bits to $v$. For the impossibility proof we use a fundamental information theoretic argument that any representation, either coded or uncoded, cannot guarantee to recover $v$ precisely from fewer than $D = \log_2 |\mathbb{V}|$ bits. This argument excludes common storage-reduction techniques like compression and de-duplication, which only work in probabilistic setups and with assumptions on the written data.

We now proceed to detail the properties describing the MWMR register.

**Liveness** There is a range of possible liveness conditions, which need to be satisfied in fair runs of a storage algorithm. 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** 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 execution that satisfies $r$’s operation precedence relation and the object’s sequential specification. A write $w$ in a run $r$ is relevant to a read $rd$ in $r$ [11] 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 [10], we consider a hierarchy of safety notions. Lamport [10] defines regular and safe single-writer registers. Shao et al. [11] 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 [11]), if for every run \( r \) and read \( rd \) that returns in \( r \), there exists a linearization \( L_{rd} \) of the subsequence of \( r \) consisting of the write operations in \( r \) and \( rd \). A MWMR register is strongly regular, (called MWRegWO in [11]), 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 \( \text{rel}-\text{writes}(r, rd_1) \cap \text{rel}-\text{writes}(r, rd_2) \), 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 \( \sigma_w \) 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 \( \sigma_w \) so as to obtain a linearization of the subsequence of \( r \) consisting of the write operations in \( r \) and \( rd \).

2.3 Erasure codes

A \( k \)-of-\( n \) erasure code takes a value from domain \( V \) and produces a set \( S \) of \( n \) pieces from some domain \( E \) s.t. the value can be restored from any subset of \( S \) that contains no less than \( k \) different pieces. We assume that the size of each piece is \( D/k \), and two functions \( \text{encode} \) and \( \text{decode} \) are given: \( \text{encode} \) gets a value \( v \in V \) and returns a set of \( n \) ordered elements \( W = \{\langle v_1, 1 \rangle, \ldots, \langle v_n, n \rangle\} \), where \( v_1, \ldots, v_n \in E \), and \( \text{decode} \) gets a set \( W' \subseteq E \times \mathbb{N} \) and returns \( v' \in V \) s.t. if \( |W'| \geq k \) and \( W' \subseteq W \), then \( v = v' \). In this paper we use \( k = n - 2f \). Note that when \( k = 1 \), we get full replication.

3 A Simple Algorithm

In order to develop intuition for the structure and limitations of distributed storage algorithms, we present in Section 3.1 a simple storage-efficient algorithm that ensures safe semantics, but not regularity. Although this algorithm has no practical use, it shows that the impossibility result of Section 4 does not apply to a weaker safety property. In Section 3.2 we then illustrate how this simple algorithm can be extended to ensure regularity using unbounded storage (similarly to some previous works), as proven to be inherent by our main result in the next section.

3.1 Safe and wait-free algorithm

This algorithm simulates a wait-free and strongly safe MWMR register using erasure codes. It stores exactly \( n \) pieces of the data, one in each base object. The algorithm’s definitions are presented in Algorithm 2 and the algorithm of client \( c_j \) can be found in Algorithm 2.

We define \( \text{Timestamps} \) to be the set of timestamps \( \langle \text{num}, c \rangle \), s.t. \( \text{num} \in \mathbb{N} \) and \( c \in \Pi \), ordered lexicographically. We define \( \text{Pieces} \) to be the set of pairs consisting of an element from \( E \) (possible outputs of the \( \text{encode} \) function) and a number, and \( \text{Chunks} = \text{Pieces} \times \text{Timestamps} \). Each base object \( bo_i \) stores exactly one value from \( \text{Chunks} \), initially \( \langle \langle v_{0i}, i \rangle, \langle 0, 0 \rangle \rangle \), where \( v_{0i} \) is the \( i \)th piece of \( v_0 \).

Since memory is fault-prone, actions are triggered in parallel on all base objects. This parallelism is denoted using \( \text{ifor} \) in the code. Operations then wait for \( n - f \) base objects to respond. Recall that \( n = 2f + k \), so every two sets of \( n - f \) base objects have at least \( k \) pieces in common. Thus, if a write completes after storing pieces on \( n - f \) base objects, a subsequent read accessing any \( n - f \) base objects finds \( k \) pieces of the written value (as needed for restoring the value), provided that they are not over-written by later writes.

A \( \text{write}(v) \) operation (lines 1-9) first produces \( n \) pieces from \( v \) using \( \text{encode} \), then reads from \( n - f \) base objects to obtain a new timestamp, and finally, tries to store every piece together with the timestamp at a different base object. For every base object \( bo_i \), \( c_j \) triggers the \( \text{update} \) RMW function, which overwrites \( bo_i \) only if \( c_j \)’s timestamp is bigger than the timestamp stored in \( bo_i \).

A \( \text{read} \) (lines 13-19) reads the values stored in \( n - f \) base objects, and then tries to restore valid data as follows. If \( c_j \) reads at least \( k \) values with the same timestamp, it uses the \( \text{decode} \) function, and returns the restored value. Otherwise, it returns \( v_0 \). The latter occurs only if there are outstanding \( \text{writes} \), that had updated fewer than \( n - f \) base objects before the reader has accessed them. Therefore, these \( \text{writes} \)
are concurrent with \( c_j \)'s read, and by the safety property, any value can be returned in this case. The algorithm’s correctness is formally proven in Appendix A.1.

Algorithm 1 Definitions.

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

Algorithm 2 Safe register emulation. Algorithm for client \( c_j \).

1: operation \( \text{write}(v) \)
2: \( W \leftarrow \text{encode}(v) \)
3: \( R \leftarrow \text{readValue()} \)
4: \( ts \leftarrow \langle \max(|ts|(|ts, *, *) \in R|) + 1, j \rangle \)
5: \( \| \) for all \( (v, i) \in W \)
6: \( \text{update}(bo_i, (v, i), ts) \rightarrow \text{trigger RMW on bo}_i \)
7: \( \text{wait for } n - f \text{ responses} \)
8: \( \text{return } \text{“ok”} \)
9: end
10: \( \text{update}(bo, w, ts) \triangleq \)
11: \( \text{if } ts > bo_.ts \)
12: \( bo \leftarrow \langle w, ts \rangle \)
13: operation \( \text{read()} \)
14: \( R \leftarrow \text{readValue()} \)
15: \( \text{if } \exists ts \text{ s.t. } |\{v \mid (ts, v) \in R\}| \geq k \)
16: \( ts' \leftarrow ts \text{ s.t. } \{v \mid (ts', v) \in R\} \geq k \)
17: \( \text{return } \text{decode}\{v \mid (ts', v) \in R\} \)
18: \( \text{return } v_0 \)
19: end
20: \( \text{procedure } \text{readValue()} \)
21: \( R \leftarrow \{\} \)
22: \( \| \) for \( i=1 \) to \( n \)
23: \( \text{if } (ts, v) \in \text{read}(bo_i) \)
24: \( \text{wait until } |R| \geq n - f \)
25: \( \text{return } R \)
26: end procedure

3.2 Achieving regularity with unbounded storage

We now give intuition why extending this approach to satisfy regularity requires unbounded storage. Note that a read from a regular register must return a valid value even if it has concurrent writes, and that a write may remain outstanding indefinitely in case the writer fails.

Consider a system with \( n = 4, f = 1, k = 2 \), where \( b_1 \) is faulty and clients \( c_1 \) and \( c_2 \) invoking \( \text{write}(v_1) \) and \( \text{write}(v_2) \) respectively, as illustrated in Figure 1a.

Since base objects may fail, clients \( c_1 \) and \( c_2 \) try to store their pieces in all the base objects in parallel (as in Algorithm 2). Assume that \( c_1 \)'s first RMW on \( b_2 \) and \( c_2 \)'s RMW on \( b_1 \) take effect. If these RMWs would overwrite the pieces in \( b_1 \) and \( b_2 \), and \( c_1 \) and \( c_2 \) would then immediately fail, the storage will remain with no restorable value. In this case, no later read can return a value satisfying regularity (note that since the two outstanding writes are concurrent with any future read, a safe register may return an arbitrary value). Therefore, \( c_1 \) and \( c_2 \) cannot overwrite the existed value in the base objects.

Consider next a client \( c_3 \) attempting to write \( v_3 \) as in Figure 1b. Even if \( c_3 \) reads the base objects, it cannot learn of any complete write. Moreover, when its RMW takes effect on \( b_4 \), it cannot distinguish between a scenario in which \( c_2 \) and \( c_3 \) have failed (thus, their pieces can be overwritten), and the scenario in which one of \( c_2 \) and \( c_3 \) is slow and will eventually be the only client to complete a writes (in which case overwriting its value may leave the storage with no restorable value). Thus, \( c_3 \) cannot overwrite any piece.

We can repeat this process by allowing an unbounded number of clients to invoke writes and store exactly one piece each, without allowing any piece to be overwritten. While this example only shows that a direct extension of Algorithm 2 consumes unbounded storage, in the next section we prove a lower bound on the storage required by any protocol.
4 Storage Lower Bound

We now show a lower bound on the required storage of any lock-free algorithm that simulates weakly regular MWMR register. Our bound stipulates that if the number of clients that can invoke write operations is unbounded, then either (1) there is a time during which there exist \( f + 1 \) base objects each of which stores at least \( D \) bits of some write, or (2) the storage can grow without bound.

**Information theoretic storage model** The storage lower bound presented in this section is obtained under a precise and natural information theoretic model of storage cost. We model the general behavior of a base object in a distributed protocol as follows. Upon each RMW operation triggered on it, the base object implements some function \( E \), whose inputs are the values currently stored in the base object and the data provided with the write. After the RMW operation, the bits output from \( E \) are everything that is stored in the base object. Upon a read operation triggered on the base object, the bits currently stored in it are input to some function \( D_i \), whose output is the value returned to the reader. To justify this model, let us observe that the role of base objects in the distributed register emulation is to store sufficient information to guarantee successful information reconstruction by a client following some future read. In the next lemma we give a more formal definition of the functions \( E \) and \( D_i \), and prove an elementary lower bound on the number of bits that \( E \) needs to output.

**Lemma 1.** Let \( E \) be a function on \( s \) arguments \( u_1, \ldots, u_s \) taking values from sets \( U_1, \ldots, U_s \), respectively. Let the output of \( E \) be a binary vector \( \{0, 1\}^\ell \). If there exist \( s \) functions \( \{D_i\}_{i=1}^s \) such that \( D_i(E(u_1, \ldots, u_s)) = u_i \) for every assignment to \( u_1, \ldots, u_s \), then necessarily \( \ell \geq \lceil \log_2(|U_1| \cdot \ldots \cdot |U_s|) \rceil \).

**Proof.** By a simple pigeonhole argument. For simplicity we assume that the sizes \( |U_i| \) are powers of 2 for every \( i \). Suppose the theorem statement is not true, that is, the output of \( E \) has fewer than \( \log_2(|U_1| \cdot \ldots \cdot |U_s|) \) bits. Then there exist at least two assignments to \( u_1, \ldots, u_s \) that map to the same output of \( E \). Hence the outputs of the functions \( \{D_i\}_{i=1}^s \) will be the same on both assignments, which is a violation because at least one \( u_i \) differs between the two assignments.

We next show how Lemma 1 implies lower bounds on the storage used in base objects. Since the information reconstruction algorithm is run by the client on inputs from base objects, we may regard each RMW operation \( i \) as requiring the base object to store a value \( u_i \) from some set \( U_i \). The size of the set \( U_i \) may change arbitrarily between writes and base objects. The particular choices of set sizes are immaterial for the current discussion, but in general they satisfy the necessary condition that globally on all surviving base objects the product of set sizes is at least \( |V| \). In the next lemma we prove that the most general function implemented by a base object upon RMW is a function \( E \) as specified in Lemma 1.
Lemma 2. Without loss of generality, a function $E$ used by a base object is a fixed (“hard coded”) function that does not depend on the instantaneous values $u_1, \ldots, u_s$.

Proof. Suppose the base object has a family of functions $E^1, \ldots, E^m$ that each maps values $u_1, \ldots, u_s$ to bits. Then, in order to allow recovering the $u_i$ values, we must store additional $\log_2(m)$ bits to inform the functions $D_i$ about which $E^j$ function was used. Therefore, this scenario is equivalent to having $E(u_1, \ldots, u_s) = [E^j(u_1, \ldots, u_s); j]$, where $;$ represents concatenation, and $E$ is a fixed function.

Lemma 2 addresses the possibility of base objects to reduce the amount of storage by adapting their functions to the instantaneous stored values. The lemma proves that without prior knowledge on the written data it is not possible to adaptively reduce the storage requirement mandated by Lemma 1. Now we are ready to prove the main property needed for our storage model. The next theorem shows that each write to a base object must add a number of bits depending on the required set size for that write, irrespective of the information presently stored from prior writes.

Theorem 1. Any write triggered on a base object with value $u_s \in U_s$ adds at least $\log_2(|U_s|)$ bits.

Proof. We prove by induction on $s$. By the induction hypothesis after $s - 1$ writes the base object stores $\log_2(|U_1| \cdot \cdots \cdot |U_{s-1}|)$ bits. Then following write $s$ triggered on the base object, we know from Lemmas 12 that any function implemented in the base object that will allow recovering $u_1, \ldots, u_s$ needs at least $\log_2(|U_1| \cdot \cdots \cdot |U_s|)$ bits. By simple subtraction we get that the new write adds at least $\log_2(|U_s|)$ bits.

The outcome from Theorem 1 is that the base object storage cost in bits is obtained as the sum of the storage requirements of individual writes. Hence in the sequel we can assume without loss of generality that each stored bit is associated with a particular write.

With the storage model in place, we now organize the proof as follows: First, in Observation 2 we observe a necessary condition for a write operation to complete. Next, we define an (unfair) adversary, and in Lemma 3 we show that under this adversary’s behavior, no write operation can complete as long as the number of base objects that store at least $D$ bits that are associated with some written value is less than $f$. Finally, in Lemma 4 and Theorem 2 we show that for every size $S$, for any algorithm that uses less storage than $S$ and with which the number of base objects that store at least $D$ bits of some written value is less than $f$ at a given time we can build a fair run in which no write operation completes.

For any time $t$ in a run $r$ of an algorithm $A$ we define the following sets, as illustrated in Figure 2:

- $C(t)$: the set of all clients that have outstanding write operations at time $t$.
- $C^+(t) \subseteq C(t)$: the set of clients that have outstanding write operations $write_i(v_i)$ s.t. at least one bit associated with $v_i$ is stored in one of the base objects or in one of the other correct clients at time $t$.
- $C^-(t) = C(t) \setminus C^+(t)$. Clients in $C^-(t)$ may have attempted to store a bit via an RMW that did not respond, or may have stored information that was subsequently erased, or may have not attempted to store anything yet.
- $F(t) = \{b_i \in N \mid b_i \text{ stores } D \text{ bits of some write at time } t \}$.

From the definition of $C^+(t)$ we get the following:

Observation 1. At any time $t$ in a run $r$, the storage size is at least $|C^+(t)|$ bits.

Observation 2. Consider a run $r$ of an algorithm that simulates a weakly regular lock-free MWMR register, and a write operation $w$ in $r$. Operation $w$ cannot return until there is time $t$ s.t. for every $B \subseteq N$ s.t. $|B| = n - f$, there is some client in $C(t)$ whose pending write's value can be restored from $B$. 

6
Proof. Assume that some write completes when there is a set $B \subset N$ s.t. $|B| = N - f$ and there is no client in $C(t)$ whose write’s value can be restored from $B$. Now, let all the base objects in $N \setminus B$ and all the clients in $C(t)$ fail, and invoke a read operation $rd$. By lock-freedom, $rd$ completes, although no value satisfying weak regularity can be returned. A contradiction.

For our lower bound, we define a particular environment behavior that schedules actions in a way that prevents progress:

**Definition 1.** (Ad) At any time $t$, Ad schedules an action as follows:

1. If there is a pending RMW on a base object in $N \setminus F(t)$ by a client 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 round robin order a client $c_i \in \Pi$ that wants to trigger an RMW, and schedule $c_i$’s action without allowing it to affect the base object yet.

In other words, Ad delays RMWs triggered by clients in $C^{-}(t)$ as well as RMWs on base objects in $F(t)$, and fairly schedules all other actions. Thus, though this behavior may be unfair, in every infinite run of Ad, every correct client gets infinitely many opportunities to trigger RMWs. We demonstrate Ad’s behavior in Figure 2. (a) Clients $c_2$ and $c_4$ are in $C^{-}(t)$ at time $t$, where $c_4$ has no pending RMWs and $c_2$ 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 $c_2$ on $b_3$. (b) In this example $c_2$ overwrites $b_3$ and so $c_3$ moves from $C^{+}$ to $C^{-}$. Since $c_3$ is the only client that has a pending RMW on a base object not in $F(t+1)$, Ad schedules the response on the RMW triggered by $c_3$ on $b_2$ at time $t + 1$. Now notice that at time $t + 2$ there is no client in $C^{-}(t + 2)$ with a pending RMW on a base object in $N \setminus F(t + 2)$, and thus, by the second rule, Ad chooses in round robin a client in $\Pi$ and allows it to trigger an RMW.

The following observation immediately follows from the adversary’s behavior.

**Observation 3.** Assume an infinite run $r$ in which the environment behaves like Ad. For each base object $b_0$, if $b_0 \in F(t)$ at some time $t$, then $b_0 \in F(t')$ for all $t' > t$.

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

**Lemma 3.** As long as the environment behaves like Ad, for any time $t$ when $|F(t)| \leq f$, there is a set $B$ of $n - f$ base objects s.t. there is no client in $C(t)$ whose value can be restored from $B$ at time $t$. 

7
Proof. As soon as a client $c_i$ stores a piece of data in a base object, $c_i$ joins $C^+$, and from that point on, as long as its data remains in the system, $c_i$ is prevented by Ad from storing any further values. Therefore, unless $c_i$ stores all $D$ bits of its value in some base object, it is impossible to reconstruct this value from the bits that were stored. Since the number of base objects storing all $D$ bits of some client value at some time $t$ is no more than $|F(t)|$, and since $|F(t)| \leq f$, the lemma follows.

From Observation 2 and Lemma 3 we conclude:

**Corollary 1.** Consider a run $r$ of an algorithm that simulates a weakly regular lock-free MWMR register. If the adversary behaves like $Ad$, and $|F(t)| \leq f$ for all $t$ in $r$, then no write completes in $r$.

Having shown that adversary $Ad$ can prevent progress in algorithms that store $D$ bits of information in too few base objects, we turn to show that we can prevent progress also in fair runs, leading to violation of lock-freedom.

**Lemma 4.** Consider a finite run $r$ with $t$ steps of an algorithm that simulates a lock-free MWMR register, where the environment behaves like adversary $Ad$. If $C^-(t) \neq \emptyset$ and $|F(t)| \leq f$, then it is possible to extend $r$ by allowing the environment to continue to behave like $Ad$ up to a time $t' \geq t$ when either $|F(t')| > f$ or some client $c_i \in C(t')$ either returns (i.e., completes the write) or receives a response from some base object.

**Proof.** Consider a client $c_i \in C^-(t)$, and denote by $T_{c_i}(t)$ the set of base objects on which $c_i$ has pending RMWs at time $t$. We first show that if $c_i$ neither receives a response from any base object nor returns, we can extend $r$ to some time $t''$ s.t. $|T_{c_i}(t'')| > f$ at time $t''$.

We extend $r$ by allowing the environment to continue to behave like $Ad$ until the first time $t'$ in which $c_i$ is the next client chosen by the adversary to trigger an RMW. If $c_i$ receives a response from some base object by time $t'$, we are done. Else, by definition of $Ad$, $T_{c_i}(t') \subseteq F(t')$. Now consider a fair run $r'$ that is identical to $r$ till time $t'$, and at time $t'$ all the clients except $c_i$ fail. Notice that $|T_{c_i}(t')| \leq |F(t')| \leq f$, so $c_i$ cannot wait for responses from base objects in $T_{c_i}(t')$, and therefore, by lock-freedom, $c_i$ either returns, or triggers an RMW on some base object in $N \setminus T_{c_i}(t')$ at time $t'$ in $r'$. The runs $r$ and $r'$ are indistinguishable to $c_i$, hence, $c_i$ either returns or triggers an RMW on some base object in $N \setminus T_{c_i}(t')$ at time $t'$ in $r$. If $c_i$ returns we are done.

We repeat this extension several times until, (after at most $f+1$ times), at some time $t''$, $|T_{c_i}(t'')| > f$. If $|F(t'')| > f$, we are done. Otherwise, $T_{c_i}(t'') \not\subseteq F(t'')$, and therefore, $Ad$ schedules a response to one of the pending RMWs of $c_i$ at time $t''$.

**Theorem 2.** For any $S$, there is no algorithm that simulates a weakly regular lock-free MWMR register with less storage than $S$ s.t. at every time $t$, $|F(t)| \leq f$.

**Proof.** Assume by way of contradiction that there is such an algorithm, $A$. We build a run of $A$ in which the environment behaves like adversary $Ad$.

We iteratively build a run $r$ with infinitely many responses, starting by invoking $S$ write operations and allowing the run to proceed according to $Ad$ until some time $t$. By the assumption, the storage is less than $S$, so by Observation 1 $|C^+(t)| < S$, and since $|C(t)| = S$, $C^-(t) \neq \emptyset$. Now since $|F(t)| \leq f$, by Lemma 1 we can extend $r$ to a time $t'$, where the environment behaves like $Ad$ until time $t'$ and some client $c_i \in C^-(t')$ either returns or receives a response from some base object at time $t'$. By Corollary 1 $c_i$ does not return, and thus, it receives a response.

By repeating this process, we get a run $r$ with infinitely many responses. By Observation 3 and by the assumption that $|F(t)| \leq f$, there is a time $t_1$ in $r$ s.t. for any time $t_2 > t_1$, $F(t_1) = F(t_2)$. Notice that by the adversary’s behavior, each correct client gets infinitely many opportunities to trigger RMWs. In addition, since $Ad$ picks responses from base objects not in $F(t)$ in the order they are triggered, every client that receives infinitely many responses, receives a response to every RMW it triggers on a
base object in $N \setminus F(t_1)$. Therefore, we can build a fair run $r'$ that is identical to $r$ but every base object $bo \in F(t_1)$ fails at time $t_1$, and every client that receives finitely many responses fails after its last response. Since there are infinitely many responses in $r'$ and the number of clients invoking operations in this run is finite, there is at least one client that receives infinitely many responses in $r'$, and thus is correct in $r$. Therefore, by lock-freedom, some client eventually completes its write operation in $r$. A contradiction to Corollary 1.

From Theorem 2, it follows that if the storage is bounded, then there is a time in which $f + 1$ base objects store $D$ bits of some write. This yields the following bound:

**Corollary 2.** There is no algorithm that simulates a weakly regular lock-free MWMR register and stores less than $(f + 1)D$ bits in the worst case.

5 Strongly Regular MWMR Register Emulation

We present a storage algorithm that combines full replication with erasure coding in order to achieve the advantages of both. The main idea behind our algorithm is to have base objects store pieces from at most $k$ different writes, and then turn to store full replicas. In Appendix A.2, we prove the following about our algorithm:

**Theorem 3.** There is an FW-terminating algorithm that simulates a strongly regular register, whose storage is bounded by $(2f + k)2D$ bits, and in runs with at most $c < k$ concurrent writes, the storage is bounded by $(c + 1)D/k$ 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.

**Data structure**

The algorithm uses the same definitions as the safe one (Section 3), given in Algorithm 1, and its pseudocode appears in Algorithms 3 and 4. The algorithm relies on a set of $n$ shared base objects $bo_1, \ldots, bo_n$ each of which consists of three fields $V_p$, $V_f$, and $storedTS$:

$$bo_i = \langle storedTS, V_p, V_f \rangle \text{ s.t. } V_f, V_p \subset Chunks, \text{ and } storedTS \in TimeStamps,$$

initially $\langle\langle 0, 0 \rangle, \{\langle\langle 0, 0 \rangle, \langle v_0, i \rangle \rangle\}, \{\\}\rangle$.

The $V_p$ field holds a set of timestamped coded pieces of values so that the $i^{th}$ piece of any value can only be stored in the $V_p$ field of object $bo_i$. The $V_f$ field stores a timestamped replica of a single value, (which for simplicity is represented as a set of $k$ coded pieces). And $storedTS$ holds the highest timestamp of a write that is known to this object to have completed the update round on $n - f$ base objects (see below).

**Write operation and storage efficiency**

The write operation (lines 3–15) 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 23–31) to collect the contents of the $V_p$ and $V_f$ fields stored at $n - f$ base objects as well as to determine the highest $storedTS$ timestamp known to these objects. The implementations of the update and garbage collection rounds are given by the update (lines 32–39) and GC (lines 40–45) routines, respectively.

The write implementation starts by breaking the supplied value $v$ into $k$ erasure-coded pieces (line 4). This is followed by 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 $v$ on
the base object. This timestamp is set to be higher than any other timestamp that has been returned (line 6) 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 \( i^{th} \) coded piece \( \langle e, i \rangle \) of \( v \) in \( bo_i.V_p \) if the size of \( bo_i.V_p \) is less than \( k \) (lines 36), 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 38). Note that storing \( \langle e, i \rangle \) in \( bo_i.V_p \) coincides with an attempt to reduce its size by removing stale coded pieces of values whose timestamps are smaller than \( storedTS \) (line 36). This guarantees that the size of \( V_p \) never exceeds the number \( c < k \) 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 pieces into this base object.

In the write’s garbage collection round, the client attempts to further reduce the storage usage by (1) removing all coded pieces associated with timestamps lower than \( ts \) from both \( bo_i.V_p \) and \( bo_i.V_f \) (lines 41–42), and (2) replacing a full replica (if it exists) of its written value \( v \) in \( bo_i.V_f \) with its \( i^{th} \) coded piece \( \langle e, v \rangle \) (line 44). 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 coded pieces 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 \( ts \), reflecting the fact that a write with \( ts' \geq ts \) that the update round.

**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 \( k \) coded pieces 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 pieces of this value. However, a read round may sample different base objects at different times (that is, it does not necessarily obtain a snapshot of all base objects), and the number of pieces stored in \( V_p \) is bounded. Thus, the read may be unable to see \( k \) matching pieces of any single new value for indefinitely long, as long as new values continue to be written concurrently with the read.

To cope with such situations, the reads are only required to return in runs where a finite number of writes are invoked, thus only guaranteeing FW-Termination. Our implementation of read (lines 16–22) proceeds by invoking multiple consecutive rounds of RMWs on the base objects via the readValue routine. After each round, the reader examines the collection of the values and timestamps returned by the base objects to determine if any of the values having \( k \) matching coded pieces are associated with timestamps that are at least as high as \( storedTS \) (line 18). If any such value is found, the one associated with the highest timestamp is returned (line 21). 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.
We proved a lower bound on the required storage of any lock-free algorithm that simulates a weakly regular MWMR register. Our bound stipulates that if write concurrency is unbounded, then either (1)
there is a time during which there exist \( f + 1 \) base objects each of which stores a full replica of some written value, or (2) the storage can grow without bound.

We showed that our lower bound does not hold for safe register emulation. And finally, by understanding these inherent limitations, we introduced a new technique for emulating shared storage by combining full replication with erasure codes. We presented an implementation of an FW-Terminating strongly regular MWMR register, whose storage cost is adaptive to the concurrency level of write operations up to certain point, and then turns to store full replicas. In periods during which there are no outstanding writes, our algorithm’s storage cost is reduced to a minimum.

Our work leaves some questions open for future work. First, we conjecture that a wait-free implementation with similar storage costs requires readers to write. Second, our algorithm requires more storage than the bound. We believe that our technique can be used for implementing additional adaptive algorithms, with storage costs closer to the lower bound. Another interesting question that remains open is whether the liveness condition of the lower bound is tight. In other words, is there an algorithm that emulates an obstruction-free weakly regular register with a better storage cost.
A Correctness Proofs

A.1 Wait-Free and Safe Algorithm

Here we prove the algorithm in Section 3.

**Lemma 5.** The storage of the algorithm is $nD/k$.

*Proof.* The size of each piece is $D/k$. We have $n$ base objects, and each base object stores exactly one piece.

**Lemma 6.** The algorithm is wait-free.

*Proof.* There are no loops in the algorithm, and the only blocking instructions are the waits in lines 7 and 24. In both cases, clients wait for no more than $n - f$ responses, and since no more than $f$ base objects can fail, clients eventually continue. Therefore, a client that gets the opportunity to perform infinitely many actions completes its operations.

We now prove that the algorithm satisfies strongly safety. We relay on the following single observation.

**Observation 4.** The timestamps in the base objects are monotonically increasing.

**Definition 2.** For every run $r$, we define the sequential run $\sigma_{w_r}$ as follows: All the completed write operations in $r$ are ordered in $\sigma_{w_r}$ by their timestamp.

**Lemma 7.** For every run $r$, the sequential run $\sigma_{w_r}$ is a linearization of $r$.

*Proof.* Since $\sigma_{w_r}$ has no read operations, the sequential specification is preserved in $\sigma_{w_r}$. Thus, we left to show the real time order: For every two completed writes $w_i, w_j$ in $r$, we need to show that if $w_i \prec_r w_j$, then $w_i \prec_{\sigma_{w_r}} w_j$.

Denote $w_i$’s timestamp by $ts$. By Observation 4 at any point after $w_i$’s return, at least $n - f$ base objects store timestamps bigger than or equal to $ts$. When $w_j$ picks a timestamp, it chooses a timestamp bigger than those it reads from $n - f$ base objects. Since, $n > 2f$, $w_j$ picks a timestamp bigger than $ts$, and therefore $w_j$ is ordered after $w_i$ in $\sigma_{rd}$.

**Definition 3.** For every run $r$, for every read $rd$ that has no concurrent write operations in $r$, we define the sequential run $\sigma_{rd}$ by adding $rd$ to $\sigma_{w_r}$ after all the writes that precede it in $r$.

In order to show that the algorithm simulates a safe register, we proof in Lemmas 8 and 9 that the real time order and sequential specification respectively, are preserved in $\sigma_{rd}$.

**Lemma 8.** For every run $r$, for every read $rd$ that has no concurrent write operations in $r$, $\sigma_{rd}$ preserves $r$’s operation precedence relation (real time order).

*Proof.* By Lemma 7 the order between the writes in $\sigma_{rd}$ are preserved, and by construction of $\sigma_{rd}$ the order between $rd$ and write operations is also preserved.

**Lemma 9.** Consider a run $r$ and any read $rd$ that has no concurrent writes in $r$. Then $rd$ returns the value written by the write with the biggest timestamp that precedes $rd$ in $r$, or $v_0$ if there is no such write.
Proof. In case there is no write before \( rd \) in \( r \), since there are also no writes concurrent with \( rd \), \( rd \) reads pieces with timestamp \((0, 0)\) from all base objects, and thus, returns \( v_0 \). Otherwise, let \( w \) be the write\((v)\) associated with the biggest timestamp \( ts \) among all the writes invoked before \( rd \) in \( r \). Let \( t \) be the time when \( rd \) is invoked. Recall that \( rd \) has no concurrent writes, so all the writes invoked before time \( t \) complete before time \( t \) and store there pieces in \( n - f \) base objects unless the base objects already hold a higher timestamp. By Observation 4 and the fact that \( w \) has the highest timestamp by time \( t \), we get that at time \( t \) there are at least \( n - f \) base objects that store a piece of \( v \). Since \( n = 2f + k \), every two sets of \( n - f \) base objects have at least \( k \) base objects in common. Therefore, \( rd \) reads at least \( k \) pieces of \( v \), and thus, restores and returns \( v \).

\[ \square \]

Corollary 3. There exists an algorithm that simulates a safe wait-free MWMR register with a worst-case storage cost of \( nD/k = (2f/k + 1)D \).

A.2 Strongly Regular Algorithm

Here we prove the algorithm in Section 5. We start by proving the storage cost.

Observation 5. For every run of the algorithm, for every base object \( bo_i \), \( bo_i.ts \) monotonically increasing.

Lemma 10. Consider a run \( r \) of the algorithm, and two writes \( w_1, w_2 \), where \( w_1 \) writes with timestamp \( ts_1 \). If \( w_1 \prec_r w_2 \), then \( w_2 \) sets its \( ts \), to a timestamp that is not smaller than \( ts_1 \).

Proof. By Observation 5 for each base object \( bo \), \( bo.ts \) is monotonically increasing. Therefore, after \( w_1 \) finishes the garbage collection phase, there is a set \( S \) consisting of \( n - f \) base objects s.t. for each \( bo_i \in S \), \( bo_i.ts \geq ts \). Recall that \( n = 2f + k \), thus every two sets of \( n - f \) base objects have at least one base object in common. Therefore, \( w_2 \) gets a response from at least one base object in \( S \) in its first phase, and thus sets \( ts = ts' \) s.t. \( ts' \geq ts \).

\[ \square \]

Lemma 11. For any run \( r \) of the algorithm, for any base object \( bo \) at any time \( t \) in \( r \), \( bo.V_p \) does not store more than one piece of the same write.

Proof. The writes perform the second phase at most one time on each base object \( bo \), and in each update they store at least one piece in \( bo.V_p \). And since they does not store in \( bo.V_p \) during the third phase, the lemma follows.

\[ \square \]

Lemma 12. Consider a run \( r \) of the algorithm in which the maximum number of concurrent writes is \( c < k - 1 \). Then the storage at any time in \( r \) is not bigger than \( (2f + k)(c + 1)D/k \) bits.

Proof. Recall that we assume that \( n = 2f + k \) and the size of each piece is \( D/k \). Thus it suffices to show that there is no time \( t \) in \( r \) s.t. some base object stores more than \( c + 1 \) pieces at time \( t \).

Assume by way of contradiction that the claim is false. Consider the time \( t \) when some \( bo \in N \) stores \( c + 2 \) pieces for the first time. Notice that \( |bo.V_p| \leq c + 1 < k \) till time \( t \), and therefore, \( bo.V_p \) does not contain more then one piece from the same write, and \( bo.V_f = \bot \) till time \( t' \). Now consider the write \( w \) that was invoked last among all the writes that store pieces in \( bo.V_p \) at time \( t \), denote its piece by \( p \). Since \( bo \) stores \( c + 2 \) pieces at time \( t' \), by Lemma 12 there must be two writes \( w_1 \) and \( w_2 \) whose pieces \( p_1, p_2 \) are stored at time \( t \) in \( bo.V_p \), and both returns before \( w \) is invoked. Denote their timestamps \( ts_1 \) and \( ts_2 \), and assume without loss of generality that \( ts_1 > ts_2 \). By Lemma 10 \( w \) sets its \( ts \) to \( ts' \) s.t. \( ts' \geq ts_1 \). Now consider two cases. First, if \( p \) was added before \( p_2 \), then \( bo.ts > ts_2 \) when \( p_2 \) was added. A contradiction. Otherwise, \( p \) was added after \( p_2 \). Thus, \( p_2 \) was deleted in line 36 of the update when \( p \) was added. A contradiction.

\[ \square \]
Lemma 13. The storage is never more than \((2f + k)2D\) bits at any time \(t\) in any run \(r\) of the algorithm.

Proof. Each base object stores no more than \(2k\) pieces at any time \(t\) in \(r\). The lemma follows.

Lemma 14. Consider a run \(r\) of the algorithm with finite number of writes, in which all writes correct. Then the storage is eventually reduced to \((2f + k)D/k\) bits.

Proof. Consider a write \(w\) with the biggest timestamp \(ts\) in \(r\). Since \(w\) is correct, and since writes are wait-free, \(w\) returns, and eventually performs free on every base object. Consider a base object \(bo\) s.t. \(w\) performs free on \(bo\) at time \(t\). Notice that \(w\) deletes all pieces with smaller timestamps than \(ts\) and set \(bo.ts = ts\) at time \(t\). Now recall that \(bo\) ignore all updates with timestamp less than \(bo.ts\), and therefore, \(bo\) store only \(w\)'s piece at any time after time \(t\). The lemma follows.

From Lemmas \([12, 13, 14]\) we get:

Corollary 4. The storage of the algorithm is bounded by \((2f + k)2D\) bits, and in runs with at most \(c < k\) concurrent writes the storage is bounded by \((c + 1)D/k\) bits. Moreover, in a run with a finite number of writes, if all the writes are correct, the storage is eventually reduced to \((2f + k)D/k\) bits.

We no prove the liveness property.

Lemma 15. Consider a fair run \(r\) of the algorithm. Then every write \(w\) invoked by a correct client \(c_i\) eventually completes.

Proof. Consider a correct client \(c_i\). The write \(w\) is divided into three phase s.t. in each phase, \(c_i\) invokes operations on all the base objects, and waits for \(n - f\) responses. The run \(r\) is fair, so every action invoked by \(c_i\) on a correct base object eventually returns, and no more than \(f\) base objects fail in \(r\). Therefore, eventually \(c_i\) receives \(n - f\) responses in each of the phases and returns.

Observation 6. When a piece from \(bo.V_p\) is deleted, \(bo.ts\) is increased.

Lemma 16. If at time \(t\), \(c_i\) completes the second phase of write with timestamp \(ts\), then for every \(t' > t\) for every \(S \subseteq N\) s.t. \(|S| \geq n - f\), exist write \(w\) with \(ts' \geq ts\) s.t. at least \(k\) pieces of \(w\) are stored in \(S\).

Proof. Consider time \(t'\). Let \(ts\) be the highest timestamp written by a write \(w\) that completed the second phase by time \(t\). It is sufficient to show the lemma hold for \(ts\).

First note that \(\forall bo, bo.ts \leq ts\) before time \(t\), because no write with a larger timestamp than \(ts\) started the third phase. This means that \(w\)'s update left at last one piece in which \(bo\) it occurred. Now consider a set \(S\) of \(n - f\) base objects, and since \(n = 2f + k\), \(w\)'s update occurred in set \(S'\) that contains at least \(k\) base objects in \(S\).

If \(w\) wrote to \(V_p\), it was not overwritten by time \(t\), because (1) no other write began free with timestamp bigger than \(ts\), and (2) since there is no base object \(bo\) s.t. \(bo.ts \geq ts\), no write delete \(w\)'s piece in the second phase. Therefore if \(w\) wrote to \(V_p\) in all base objects in \(S'\), the lemma holds.

Otherwise, \(w\) wrote \(k\) pieces to \(V_f\) in base objects in some set \(S'' \subseteq S'\). Consider two cases: First, there is base object \(bo'\) in \(S''\) s.t. some write overwritten \(w\)'s pieces in \(bo'.V_f\) before time \(t\). Since there is no write with timestamp bigger than \(ts\) that started the third phase before time \(t\), it is guarantee that \(k\) pieces with timestamp \(ts' > ts\) stored in \(bo'.V_f\) at time \(t\), and the lemma holds. Else, since \(w\)'s pieces stored in \(S'\) \(S''\) does not overwritten before time \(t\), the lemma holds (no matter if \(w\) performed the third phase or not).

Invariant 1. For any run \(r\) of the algorithm, for any time \(t\) in \(r\), for any set \(S\) of \(n - f\) base objects. Let \(ts_s = \max\{bo.ts \mid bo \in S\}\). Then there is a timestamp \(ts' \geq ts_s\) s.t. there are at least \(k\) different pieces associated with \(ts'\) in \(S\).
Proof. We prove by induction. **Base:** the invariant holds at time 0.  **Induction:** Assume that the induction holds before the \( t \)th action is scheduled, we show that it holds also at time \( t \). Assume that the \( t \)th action is RMW on a base object \( bo \), and consider any set \( S \) of \( n-f \) base objects. If \( bo \not\in S \) then the invariant holds. Else, consider the two possible RMW actions:

- The \( t \)th action is **update**. If no pieces are deleted, the invariant holds. If \( bo.ts \) is increased, then consider the write with timestamp \( ts \) that is the the biggest timestamp among all writes that complete the second phase before time \( t \). Notice that \( bo.ts \leq ts \) at time \( t \), and by Lemma 16 the invariant holds. The third option is that a piece \( p \) with timestamp \( ts' > bo.ts \) is deleted and \( bo.ts \) is not increased. Note that by Observation 6 such piece can be deleted only from \( bo.V_f \), and since \( p \) is overwritten by \( k \) pieces with bigger timestamp, the invariant holds.

- The \( t \)th action is **free**. If \( bo.ts \) is not changes, then the invariant holds. Else, Consider the write with the biggest timestamp \( ts \) among all writes that complete the second phase before time \( t \). Note that \( bo.ts \) is set to a timestamp \( ts' \leq ts \), so by Lemma 16 the invariant holds.

\[ \square \]

**Lemma 17.** Consider a fair run \( r \) of the algorithm. If there is a finite number of write invocations in \( r \), then every read operation \( rd \) invoked by a client \( c_i \) eventually returns.

Proof. Assume by way of contradiction that \( rd \) does not return in \( r \). By Lemma 15 the writes are wait-free, and since the number of write invocations in \( r \) is finite, there is a time \( t \) in \( r \) s.t. no write performs actions after time \( t \). Therefore, any read that invokes \( readValue() \) procedure after time \( t \) receives a set \( S \) of values that is stored in a set of \( n-f \) base objects at time \( t \). By invariant 1 there is a timestamp \( ts \) s.t. there is at least \( k \) different pieces in \( S \) associated with \( ts \), and \( ts > bo.ts \) for all \( bo \in S \). Now since the every correct read \( rd \) invokes \( readValue() \) infinitely many times in \( r \), \( rd \) returns. A contradiction.

\[ \square \]

The next corollary follows from Lemmas 15, 17.

**Corollary 5.** The algorithm satisfies the WF-termination property.

We now prove that the algorithm satisfies strong regularity.

**Definition 4.** For every run \( r \), \( \sigma_r \) is a sequential run s.t. the writes in \( r \) are ordered in \( \sigma_r \) by their timestamp, and every read in \( r \) that returns a value associate with timestamp \( ts \), is ordered in \( \sigma_r \) immediately after the write that is associate with timestamp \( ts \).

For simplicity we say the that \( v_0 \) was written by \( write w_0 \) that associated to timestamp 0 at time 0.

**Lemma 18.** Consider a run \( r \), and a read \( rd \) that returns a value \( v \). Consider also the timestamp \( ts' \) that \( rd \) obtains in line 20 (Algorithm 3). Then \( v \) is the value written by a write associated with timestamp \( ts' \) or \( v_0 \) if \( ts' = 0 \).

Proof. By the code, if \( ts' = 0 \), then \( rd \) returns \( v_0 \). Now notice that \( rd \) obtains at least \( k \) different pieces associated with timestamp \( ts' \), thus by decode definition, \( rd \) returns \( v \).

\[ \square \]

**Corollary 6.** For every run \( r \), \( \sigma_r \) satisfies the sequential specification.

**Observation 7.** Consider a write \( w \) that obtains \( ts \) and \( ts^+ \) in the first phase, then \( ts > ts^+ \).

**Lemma 19.** For every run \( r \), for every two writes \( w_1, w_2 \) with timestamp \( ts_1, ts_2 \). If \( w_2 \) was invoked after \( w_1 \) finished the second phase, then \( ts_1 < ts_2 \).
Proof. First notice that for every base object \( bo \), if a write \( w \) overwrites pieces of a write \( w' \) in \( bo.V_f \), that \( w' \) timestamp is bigger than \( w' \)'s. And by Observation \( 7 \) if \( w \) deletes \( w' \)'s piece from \( bo.V_p \), then it stores a piece with bigger timestamp than \( w' \)'s timestamp. Therefore, the maximal timestamp in each base object is monotonically increasing. Now recall that in the second phase \( w_1 \) performed update on \( n - f \) base object, and notice that after \( w_1 \) performs update on base object \( bo \) the maximal timestamp in \( bo \) is at lest as big as \( ts_1 \). Now since two sets of \( n - f \) base object have at least one base object in common, \( w_2 \) picks \( ts \) > \( ts_1 \).

\[ \square \]

Lemma 20. For every run \( r \), for every two writes \( w_1, w_2 \) in \( r \), if \( w_1 \prec_r w_2 \), then \( w_2 \) is not ordered before \( w_1 \) in \( \sigma_r \).

Proof. Follows immediately from Lemma \[19 \]

\[ \square \]

Lemma 21. For every run \( r \), for every read \( rd \) and write \( w_1 \), if \( rd \prec_r w_1 \), then \( w_1 \) is not ordered before \( rd \) in \( \sigma_r \).

Proof. Assume that \( rd \) returns value that is associated with timestamp \( ts \) belonging to some write \( w \), and \( w_1 \) is associated with timestamp \( ts_1 \). Since \( rd \) returns \( w \)'s value, \( w \) begins the third phase before \( rd \) returns. And since \( w_1 \) was invoked after \( rd \) returns, \( w_1 \) was invoked after \( w \)'s second phase. Therefore, by Lemma \[19 \] \( ts_1 > ts \), and thus \( w_1 \) is ordered after \( w \) in \( \sigma_r \). Recall that by the construction of \( \sigma_r \), \( rd \) is ordered immediately after \( w \) in \( \sigma_r \), hence, \( rd \) is ordered before \( w_1 \) in \( \sigma_r \).

\[ \square \]

Lemma 22. For every run \( r \), for every read \( rd \) and write \( w_1 \), if \( w_1 \prec_r rd \), then \( rd \) is not ordered before \( w_1 \) in \( \sigma_r \).

Proof. Consider a write \( w_1 \) with timestamp \( ts_1 \) and a read \( rd \) s.t. \( w_1 \prec_r rd \). Assume by way of contradiction that \( rd \) is ordered before \( w_1 \) in \( \sigma_r \). Then \( rd \) returns a value with a timestamp \( ts \) that is associated with a write \( w \) that is ordered before \( w_1 \) in \( \sigma_r \). By the construction of \( \sigma_r \), \( ts_1 > ts \). Now since \( w_1 \) completed the third phase before \( rd \) invoked, and since by Observation \[7 \] for each \( bo \), \( bo.ts \) is monotonically increasing, when \( rd \) invoked, for every set \( S \) of \( n - f \) base objects, the maximal \( bo.ts \) of all \( bo \in S \) is bigger than or equal to \( ts_1 \), and thus bigger than \( ts \). Therefore \( rd \) set \( ts \), in the first phase, to timestamp bigger than \( ts \), and thus does not return \( w \)'s value. A contradiction.

\[ \square \]

The next corollary follows from Corollary \[6 \] and Lemmas \[20 \] \[21 \] \[22 \]

Corollary 7. The algorithm simulates a strongly regular register.

The following theorem stems from Corollaries \[4 \] \[5 \] and \[7 \]

Theorem 3. There is a FW-terminating algorithm that simulates a strongly regular register, which storage is bounded by \((2f + k)2D \) bits, and in runs with at most \( c < k \) concurrent writes, the storage is bounded by \((c + 1)D/k \) bits. Moreover, in a run with a finite number of writes, if all the writes are correct, the storage is eventually reduced to \((2f + k)D/k \) bits.

References

[1] Ittai Abraham, Gregory Chockler, Idit Keidar, and Dahlia Malkhi. Byzantine disk paxos: optimal resilience with byzantine shared memory. Distributed Computing, 18(5):387–408, 2006.

[2] Yehuda Afek, Michael Merritt, and Gadi Taubenfeld. Benign failure models for shared memory. In Distributed Algorithms, pages 69–83. 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, pages 336–345. 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):124–142, 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, pages 115–124. IEEE, 2006.

[6] 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, pages 253–260. IEEE, 2014.

[7] 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, pages 182–196, Berlin, Heidelberg, 2008. Springer-Verlag.

[8] 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, pages 135–144. IEEE, 2004.

[9] Prasad Jayanti, Tushar Deepak Chandra, and Sam Toueg. Fault-tolerant wait-free shared objects. Journal of the ACM (JACM), 45(3):451–500, 1998.

[10] Leslie Lamport. On interprocess communication. Distributed computing, 1(2):86–101, 1986.

[11] Cheng Shao, Jennifer L Welch, Evelyn Pierce, and Hyunyoung Lee. Multiwriter consistency conditions for shared memory registers. SIAM Journal on Computing, 40(1):28–62, 2011.

[12] Zhiying Wang and Viveck Cadambe. Multi-version coding in distributed storage. In Information Theory (ISIT), 2014 IEEE International Symposium on, pages 871–875. IEEE, 2014.