MASIN: Multi-version codes for Atomic Storage with INplace-update

Marwen Zorgui
Center for Pervasive Communications and Computing, University of California, Irvine, CA

Robert Mateescu
Western Digital Research, Milpitas, CA

Filip Blagojevic
Western Digital Research, Milpitas, CA

Cyril Guyot
Western Digital Research, Milpitas, CA

Zhiying Wang
Center for Pervasive Communications and Computing, University of California, Irvine, CA

Abstract
We study the design of storage-efficient algorithms for emulating atomic shared memory over an asynchronous, distributed message-passing system. Our first algorithm is an atomic single-writer multi-reader algorithm based on a novel erasure-coding technique, termed multi-version code. Next, we propose an extension of our single-writer algorithm to a multi-writer multi-reader environment, which combines replication and multi-version code. Moreover, when the number of concurrent writes is bounded, we propose a simplified variant of the second algorithm that has a simple structure similar to the single-writer algorithm.

Let \( N \) be the number of servers, and the shared memory variable be of size 1 unit. Our algorithms have the following properties: (i) The write operation is wait-free if the number of server failures is bounded by a parameter \( f \). The algorithms also guarantee the wait-freedom of the read as long as the number of writes concurrent with the read is smaller than a design parameter \( \nu \), and the number of server failures is bounded by \( f \). (ii) The overall storage size for the first algorithm, and the steady-state storage size for the second algorithm, are all \( N/\lceil N-2f \rceil \) units. Moreover, our simplified variant of the second algorithm achieves the worst-case storage cost of \( N/\lceil N-2f \rceil \), asymptotically matching a lower bound by Cadambe et al. for \( N \gg f, 1 \ll \nu \leq f+2 \). (iii) The write and read operations only consist of a small number (2 to 3) of communication phases, with bounded write and read communication costs. (iv) Similar to replication-based algorithms, all of our algorithms possess the in-place update property: a server only needs to store information associated with the latest value it observes.

2012 ACM Subject Classification Dummy classification

Keywords and phrases Distributed systems, shared memory emulation, erasure codes.

Digital Object Identifier 10.4230/LIPIcs.CVIT.2016.23
1 Introduction

The emulation of a consistent, fault-tolerant, read-write shared memory in a distributed, asynchronous message-passing network has been an active area of research in distributed computing theory. Several applications demand concurrent and consistent access to the stored value by multiple writers and readers. In their celebrated paper [3], Attiya, Bar-Noy, and Dolev proposed a fault-tolerant algorithm (ABD algorithm) for emulating a shared memory that achieves atomic consistency or linearizability [14, 19]. ABD uses a replication-based storage scheme at the servers to attain fault tolerance. Variations of replication-based system algorithms appear in practical systems [18, 20].

Following [3], several papers developed algorithms that use erasure coding instead of replication for fault tolerance, with the goal of improving upon the storage efficiency. In erasure coding, each server stores a function of the value called a coded symbol. A decoder can recover the value by accessing a number (called the coding parameter) of coded symbols. The number of bits used to represent a coded symbol is typically much smaller than the number of bits used to represent the value. Erasure coding is well known to lead to smaller storage costs as compared to replication [23]. Erasure-code based implementations of consistent data storage appear in [5, 11, 15, 16, 24] for crash failures. In [4, 10, 13] erasure codes are used in algorithms for implementing atomic memory that tolerate Byzantine failures. In [11, 12, 17], the authors provide algorithms that permit repair of crashed servers, while implementing consistent storage. Bounds on the performance costs for erasure-code based implementations appear in [6, 8, 24].

Motivation: We study the emulation of a shared atomic memory in an asynchronous message-passing network. A shared atomic storage can be emulated by composing individual atomic objects [14, 21]. Thus, we seek to implement only one atomic read/write memory object. We consider a network with fixed nodes and reliable channels. Nodes can have crash failures. An arbitrary number of client nodes can fail. Every new invocation at a client waits for a response of a preceding invocation at the same client (called well-formedness). The stored value is of size 1 unit. We assume fixed system parameters $N, f, \nu$, where $N$ is the number of servers, up to $f$ server failures can be tolerated, $f \leq (N - 1)/2$, and a read operation should terminate if it is concurrent with less than $\nu$ write operations. Here $\nu$ is called the liveness parameter.

Since $f$-tolerant shared memory emulation is impossible with less than $2f + 1$ servers [21], ABD algorithm uses the lowest number of servers, given by $2f + 1$, to emulate an atomic shared object, with a total storage cost of $2f + 1$ units. We assume throughout the paper that ABD uses $2f + 1$ servers. On the other hand, erasure-code-based emulations typically use a higher number of servers $N > 2f + 1$, such that the overall storage is reduced to less than that of ABD. References [6, 24] showed that overall storage cost of emulation algorithms fundamentally grows with the server failures $f$ and the write concurrency $\nu - 1$.

In particular, when $\nu$ is large compared to $f$, it is shown that the storage cost of ABD is optimal. On the other hand, a result that applies to most emulation algorithms is that the worst-case overall storage cost is lower bounded by $\frac{(\nu - 1)N}{N - f + \nu - 2}$ for $\nu \leq f + 2$ [6], but its achievability remains an open problem\(^1\). A main question of interest in this paper is:

**Question 1:** Can coding across $\nu$ versions reduce the worst-case storage to $\frac{(\nu - 1)N}{N - f + \nu - 2}$?

\(^1\) In [6, 24], the write concurrency refers to the number of concurrent write operations that are invoked but not completed at any point. This is different from our liveness parameter $\nu$, but the lower bounds in [6, 24] still applies to our setting under certain conditions. See Section 4.5 and Section 5 for details.
Recently, multi-version code [25], using information-theoretic techniques, answered this question in the affirmative for a coding-theoretic model. While the model in [25] differs from the model of interest in a number of ways (e.g., it lacks formal notions of write and read protocols, and decoding requirement is loosely based on the concept of atomicity), a key result is that one can jointly code \( \nu \) versions using a simple yet asymptotically optimal scheme, which encodes each version separately and stores enough data of only the latest version observed in each server. In this work, we leverage the insights from [25] in the design of storage-efficient emulation algorithms, and analyze the merits of using multi-version codes.

**Contributions:** We present our algorithms, termed MASIN (Multi-version codes for Atomic Storage with Inplace-update). We first propose a single-writer multi-reader (SWMR) atomic shared memory emulation algorithm, Algorithm 1. Then, we propose a multi-writer multi-reader (MWMR) algorithm, Algorithm 2, along with a simpler variant, Algorithm 2-A, under a specific setting to be specified.

Our algorithms have a simple structure reminiscent of the ABD algorithm as servers only need to store information associated with the latest value they observe, without any logs or history. We call this in-place update. The write and read operations only consist of 2 or 3 communication phases.

We require the following safety and liveness properties, irrespective of the number of client failures.

- **Atomicity:** The algorithm must emulate a shared atomic read-write object that supports concurrent access by the clients in the system, where the observed global external behaviors “look like” the object is being accessed sequentially [19].

- **\( \nu \)-concurrency wait-freedom:** We require a write operation to terminate if the number of server failures in the execution is bounded by \( f \), and a read operation to terminate if the number of server failures is bounded by \( f \) and the number of concurrent writes with the read is less than \( \nu \). We call such liveness property \( \nu \)-concurrency wait-freedom. In general the number of concurrent writes at any point in an execution is allowed to be arbitrarily large, and not limited by the liveness parameter \( \nu \).

Trading-off between consistency and liveness has been an active research and engineering topic, with different algorithms and implementations proposed. For instance, BigTable, a distributed storage system by Google, prioritizes safety over liveness [7], whereas, Amazon’s Dynamo does not compromise liveness at the expense providing a weaker eventual consistency [9]. The algorithms in [5, 17] provide \( \nu \)-concurrency wait-freedom. In practice, our algorithms do not need to know the exact worst-case concurrency level over all executions. Instead, it can use \( \nu \) as an estimate of the concurrency, say, for 90% of the read operations. If a reader is not able to return the value, it can re-try and complete the read once the number of concurrent writes reduces to less than \( \nu \).

We consider storage and communication costs defined below.

- **Storage cost:** The storage cost is defined to be the total amount of data stored across all servers, at a point during the execution of an algorithm. We assume that metadata (e.g., timestamp) is of negligible size compared to the stored value, and is hence ignored. The steady-state storage cost corresponds to a steady-state point, for which there is no ongoing write, and the completed writes have delivered their messages to all live servers. The worst-case storage cost is the largest storage cost among all points in all executions.

- **Communication cost:** The read (resp. write) communication cost is the largest amount of the total transmitted data, among all read (resp. write) operations of all executions. Metadata cost is again neglected.

The storage and communication costs of our algorithms, ABD, and two previous coding-
Related work: Assume that a read operation is concurrent with several writes, including failed writes, i.e., writes invoked by failed writers. Then, in erasure coding, it is possible that the reader obtains information of different values, but does not have sufficient number of coded symbols to decode and return any value. In order to handle the difficulty brought by concurrent writes, several techniques and liveness guarantees have been proposed, described below.

Algorithms in [5, 10, 13] store history of received coded symbols, and hide ongoing writes from a read until enough number of coded symbols have been propagated to the servers. However, the worst-case storage cost grows unbounded with the number of concurrent writes. Algorithms in [11, 15, 16] propagate full replicas at a first phase before performing erasure coding at a second phase. In SCCK [24] a writer communicates full replicas, and each server, upon receipt of a full replica, either stores a coded symbol or the full replica depending on its state, leading to a worst-case storage of $2N$ units. ORCAS-A [11] is similar to our algorithms in that the server stores only the latest version. However, the read operation uses reader registration to be explained later. The algorithm in [15] achieves the lowest overall storage in the steady state at the expense of costly write communication on the order of $N^2$ units. In [16], replicas of all ongoing writes are stored in an edge layer of servers, and coded symbols are stored in a back-end layer. The total worst-case storage can be unbounded even with garbage collection in the edge layer.

The strongest liveness guarantee for read operations is wait-freedom, which guarantees that all operations invoked by correct clients eventually terminate despite the concurrent invocations of other clients, implemented by reader registration in [4, 11, 15, 16]. That is, a read operation registers itself at the servers it contacts, and keeps receiving symbols from them until successful recovery of a value. However, the amount of communication of a read operation can be unbounded and depends on concurrent writes with the read.
In contrast, our algorithms guarantee liveness if the number of concurrent writes with a read is smaller than $\nu$, and only uses 2 or 3 phases of communications, with bounded write and read communication costs. A similar liveness setting is found in CASGC [5], but we will demonstrate the advantage of our algorithms in Section 5 in terms of the storage cost and protocol simplicity. SCCK [24] satisfies the finite-write termination liveness, namely, in every execution with finitely many writes, every read operation invoked by a non-failed reader terminates [2]. In HGR [13], read operations satisfy obstruction-freedom, that is, a read returns if there is a sufficiently long period during the read when no other operation takes steps.

**Organization:** In Section 2, we introduce useful definitions and lemmas. The SWMR and MWMR MASIN algorithms are presented and analyzed in Sections 3 and 4. Detailed comparisons with previous algorithms and conclusions are drawn in Section 5.

## 2 Preliminaries

In this section, we introduce necessary definitions, the principles of erasure codes and multi-version codes, and a lemma that is used to prove atomicity throughout the paper.

We consider algorithms that tolerate $f$ failures out of $N$ servers, with a liveness parameter $\nu$. We define a quorum set $Q$ to be a subset of the server nodes, such that its size satisfies $|Q| \geq N - f$. It follows that for any two quorums $Q_1, Q_2$, we have $|Q_1 \cap Q_2| \geq N - 2f$. We assume that every data value comes from a finite set $\mathcal{V}$. In this paper we refer to $\log_2 |\mathcal{V}|$ as 1 unit. We also arbitrarily choose $v_0$ from $\mathcal{V}$ to be a default value. Different versions of the data value are associated with different tags.

**Erasure codes:** Let $\Phi$ be an $(N,k)$ maximum distance separable code (e.g. Reed-Solomon code) that takes a value in $\mathcal{V}$ as input and outputs $N$ coded symbols in $\mathcal{W}$, where $\log_2 |\mathcal{W}| = \frac{1}{k} \log_2 |\mathcal{V}|$, corresponding to $\frac{1}{k}$ unit. Any $k$ of the $N$ coded symbols suffice to decode the value. We say that a tag $t$ is decodable if the read operation collects at least $k$ coded symbols with tag $t$.

The intuition of our coding strategy is explained informally as below. A naive code can encode each version separately with $k$ being $N - 2f$, which is the size of the intersection of two quorums. It needs to store all the $\nu$ concurrent versions at the servers to ensure a correct read. The benefit of erasure code is that the worst-case storage $\frac{\nu N}{N - 2f}$ is smaller than that of ABD, $2f + 1$, when, for example, $\nu < 2f + 1$, $N \gg f$.

We extend to jointly code $\nu$ concurrent versions and further reduce the storage. Motivated by multi-version code [25], this is achieved by encoding each version separately using $k = \lceil \frac{N - 2f}{\nu} \rceil$, and storing only the version with the highest tag at the servers, resulting in a worst-case storage of $N/\lceil \frac{N - 2f}{\nu} \rceil$ (see details in Algorithm 2-A). We will show by the Pigeonhole principle that this simple modification of $k$ guarantees correctness if a read is concurrent with less than $\nu$ writes. For fixed $N, f, \nu$, compared to naive code, our code reduces the storage size by a factor of up to 2.

**Example 1.** Let $N = 11$, $f = 2$, $\nu = 3$. ABD uses the minimum number of servers, $2f + 1 = 5$, and the storage is 5. The naive code and our Algorithm 2-A both use $N = 11$ servers, and have the worst-case storage of $\frac{\nu N}{N - 2f} = 4.71$ and $N/\lceil \frac{N - 2f}{\nu} \rceil = 3.67$, respectively.

**Remark.** For fixed system parameters $N, f, \nu$, the coding parameter is $k = \lceil \frac{N - 2f}{\nu} \rceil$. In fact, we can use only $\tilde{N} = (k - 1)\nu + 2f + 1 \leq N$ servers and do not use the remaining, while keeping the same the coding parameter $\lceil \frac{N - 2f}{\nu} \rceil$. Throughout the paper, we will...
use the reduced number of servers, and assume the integer $k$ satisfies

$$k = \left\lceil \frac{N - 2f}{\nu} \right\rceil = 1 + \frac{N - (2f + 1)}{\nu}. \quad (1)$$

Next we state a lemma of a sufficient condition for atomicity, which will be used to prove correctness for our algorithms.

Lemma 2 (Lemma 13.16 of [21]). Let $\beta$ denote a sequence of actions of the external interface of a read/write object. Suppose $\beta$ is well formed for each client and contains no incomplete operations. Let $\Pi$ be the set of all operations in $\beta$. A sufficient condition for atomicity of $\beta$ is: there exists a partial ordering $\prec$ of all the operations in $\Pi$, satisfying the following properties:

1. If the response for $\pi_1$ precedes the invocation for $\pi_2$ in $\Pi$, then it cannot be the case that $\pi_2 \prec \pi_1$.
2. If $\pi_1$ is a write operation in $\Pi$ and $\pi_2$ is any operation in $\Pi$, then either $\pi_1 \prec \pi_2$ or $\pi_2 \prec \pi_1$.
3. The value returned by each read operation is the value written by the last preceding write operation according to $\prec$ (or the default value, if there is no such write).

We now define the partial ordering that we use in conjunction with Lemma 2 in the correctness proofs. We define tags of operations for each algorithm in its corresponding section.

Definition 3 (Partial Ordering $\prec$). Consider an execution $\alpha$ and consider two operations $\pi_1, \pi_2$ that complete in $\alpha$. Let $T(\pi_1)$ and $T(\pi_2)$ respectively denote the tags of operations $\pi_1$ and $\pi_2$. Then we define the partial ordering on the operations as: $\pi_1 \prec \pi_2$ if (1) $T(\pi_1) < T(\pi_2)$; or (2) $T(\pi_1) = T(\pi_2)$ if $\pi_1$ is a write and $\pi_2$ is a read.

3 Single-Writer Multi-Reader Algorithm

3.1 Algorithm Description

In this section, we describe our SWMR MASIN algorithm (See Algorithm 1). The different phases of the write and read protocols are executed sequentially. In each phase, a client sends messages to servers to which the non-failed servers respond. Termination of each phase depends on getting responses from at least one quorum.

The write protocol has one phase, where a tag is incremented and the associated value is encoded using an $(N,k)$ erasure code and propagated to at least a quorum. For a tag-value pair $r = (t,v)$, we write $tag(r) = t$.

The read protocol is carried out in two phases, one for getting values, and one for writing back. The write-back phase is triggered whenever the reader can recover a certain value $v$ from the responses and safely return it. The read returns a value with tag $t$ that is decodable and also satisfies some conditions, as specified by Lines 14 through 18 in Algorithm 1. The intuition behind Line 15 is that coded symbols corresponding to any old write operation can only be stored in at most $f$ servers at any point of the execution. Line 16 ensures that the returned value does not violate atomicity. The read protocol has an _abort_ internal action. In case the _abort_ action is invoked, the client does not return and the operation which invokes it does not terminate. The action indicates to the reader that the liveness bound is violated causing the read not to terminate. From the viewpoint of a practical storage system, we note that the _abort_ action can prompt the reader to invoke a read
request again; however, we do not formally incorporate such an invocation in the description of Algorithm 1. We comment on the possibility of invoking multiple rounds of read briefly in Section 3.3.

Algorithm 1: SWMR setting

**Write client protocol**

1: state variable: Tag \( t, t \in \mathbb{N} \). Initially, \( t \leftarrow 0 \).
2: function \( \text{write}(v) \)
3: \( t \leftarrow t + 1 \).
4: Let \((y_1, y_2, \ldots, y_N) = \Phi(v)\).
5: \[ \text{for } s \in \{1, 2, \ldots, N\} \]
6: \( \text{send put}(t, y_s) \)
7: \( \text{wait until receive acknowledgments from a quorum.} \)

**Write-back phase**

18: Let \( t = \text{max}(T) \), and \( v \) its value.
19: Let \((y_1, y_2, \ldots, y_N) = \Phi(v)\).
20: \[ \text{for } s \in \{1, 2, \ldots, N\} \]
21: \( \text{send put}(t, y_s) \)
22: \( \text{wait until receive acknowledgments from a quorum.} \)
23: \( \text{return } v. \)
24: else
25: \( \_\text{abort}_\).\)
26: end if
27: end function

**Read client protocol**

9: function \( \text{read}() \)
10: \[ \text{for } s \in \{1, 2, \ldots, N\} \]
11: \( \text{send request get() to server } s \)
12: \( \text{wait until receive responses from a quorum} \)
13: Let \( R \) be the set of response pairs.
14: Let \( T \) be the set of decodable tags \( t \) occurring in \( R \) such that
15: \( (i) \) \( t \) has at least \( f + 1 \) coded symbols,
16: \( (ii) \) or, the number of tags strictly higher than \( t \) is at most \( \nu \).
17: if \( T \neq \emptyset \) then
18: \[ \text{let } \]
19: \( \text{for } s \in \{1, 2, \ldots, N\} \]
20: \( \text{send put}(t, y_s) \)
21: \( \text{wait until receive acknowledgments from a quorum.} \)
22: \( \text{return } v. \)
23: \( \_\text{abort}_\).\)
24: else
25: \( \_\text{abort}_\).\)
26: end if
27: end function

**Server s protocol**

28: state variable at server \( s \): A pair \((t, y)\), where \( t \in \mathbb{N}, y \in W \).
29: Initially, server \( s \) stores \((t, y_s) = (0, y_s)\), where \( y_s \) is the \( s \)th component of \( \Phi(v_0) \).
30: Upon receipt of \( \text{get()} \) do
31: respond with \((t, y_s)\).
32: Upon receipt of \( \text{put}(t_{\text{new}}, y_{\text{new}}) \) do
33: If \( t_{\text{new}} > t \), then set \( t \leftarrow t_{\text{new}} \) and \( y \leftarrow y_{\text{new}} \). In any case respond with acknowledgement.

**Remark.** If \( \nu \geq N - 2f \), the code used in Algorithm 1 specializes to the replication-based ABD algorithm. We have \( k = 1 \), i.e., every server stores a full replica. Moreover, the reader recovers the value corresponding to the highest tag observed among the responses and the value satisfies the condition in Line 16.

Throughout the section, we assume that \( \nu \geq 2 \) and \( k > 1 \).

### 3.2 Safety Properties

In this section, we present safety properties satisfied by Algorithm 1. We first show in Lemma 5 that there always exists some value that can be recovered from the servers during the execution of Algorithm 1. Then, we show that Algorithm 1 emulates an atomic shared memory in Theorem 10, using Lemma 2 on the partial ordering \( \prec \) in Definition 3. To show this, we prove Lemmas 6, 7, 8 and 9. We finally prove a safety property in Lemma 11 that will be used in Section 3.3 where we describe liveness properties.

We now define the tag of an operation. In the definition, we use the fact that every read or write operation that completes propagates put messages to the servers with a particular
tag. Note that the tag of an operation is defined for every operation that completes in an execution. Furthermore, the tag is not defined for read operations that abort, since these operations do not propagate a put message and are not considered complete.

Definition 4 (Tag of an operation π). Let π be an operation in an execution α. The tag of operation π is defined to be the tag associated with the put messages that the operation propagates to the servers.

Lemma 5 (Persistence of data). The value written by either the latest complete write or a newer write is available from every set of at least \( N - f \) servers.

Proof. Let \( Q_w \) denote the quorum of servers that replied to the put message of the latest finished write \( π_w \), if no write has finished in the execution, \( Q_w \) can be any quorum from the set of live of servers, and \( T(π_w) \) is assumed to be 0. Thus, each server in \( Q_w \) has a tag that is at least as large as \( T(π_w) \). Because there is at most one single ongoing incomplete write operation for a single writer, the number of tags in \( Q_w \) is at most 2. Thus, one of the tags, say \( t \), appears in at least \( \left\lceil \frac{|Q_w|}{2} \right\rceil \geq \left\lceil \frac{N-2f}{f} \right\rceil = k \) servers and \( t \geq T(π_w) \). Therefore, the value corresponding to \( t \) is available in the system.

Lemma 6. Consider any execution \( α \) of the algorithm and consider a write or read operation \( π_1 \) that completes in \( α \). Let \( T(π_1) \) denote the tag of the operation \( π_1 \) and let \( Q_1 \) denote the quorum of servers from which responses are received by \( π_1 \) to its put message. Consider a read operation \( π_r \) in \( α \) that is invoked after the termination of the write operation \( π_1 \). Suppose that the read \( π_r \) receives responses to its get message from a quorum \( Q_r \). Then, (1) Every server \( s \) in \( Q_1 \cap Q_r \) responds to the get message from \( π_r \) with a tag that is at least as large as \( T(π_1) \). (2) If, among the responses to the get message of \( π_r \) from the servers in \( Q_1 \cap Q_r \), the number of tags is at most \( ν \), then there is some tag \( t \) such that (i) \( t \geq T(π_1) \), and (ii) from the servers in \( Q_1 \cap Q_r \), operation \( π_r \) receives at least \( k \) responses to its get message with tag \( t \).

Proof. Proof of (1). Consider any server \( s \) in \( Q_1 \cap Q_r \). From the server protocol we note that at every point after the reception of \( π_1 \)'s put message, it stores a tag that is no smaller than \( T(π_1) \). So it responds to the get message with a tag that is at least as large as \( T(π_1) \).

Proof of (2). Among the responses from \( Q_1 \cap Q_r \), the read \( π_r \) receives at most \( ν \) different tags. By the Pigeonhole principle, there is at least one tag \( t \) such that it receives at least \( \left\lceil \frac{|Q_1 \cap Q_r|}{ν} \right\rceil \) responses with \( t \). Since \( |Q_1 \cap Q_r| \geq N - 2f \), we infer that the operation \( π_r \) receives at least \( \left\lceil \frac{N-2f}{ν} \right\rceil = k \) responses with tag \( t \). From (1), we infer that \( t \geq T(π_1) \) to complete the proof.

Lemma 7. Consider an execution \( α \) of Algorithm 1. Let \( π_1 \) be a write or read operation that completes in \( α \), and let \( π_2 \) be a read operation that completes in \( α \). Let \( T(π_1) \) denote the tag of operation \( π_1 \) and \( T(π_2) \) denote the tag of operation \( π_2 \). If \( π_2 \) begins after the termination of \( π_1 \), then \( T(π_2) \geq T(π_1) \).

Proof. Let \( Q_1 \) denote the quorum that responds to \( π_1 \)'s put message. Let \( Q_2 \) denote the quorum that responds to \( π_2 \)'s get message. Let \( Q = Q_1 \cap Q_2 \). We prove the claim by contradiction. Suppose that \( T(π_2) < T(π_1) \). Either Line 15 or Line 16 should be satisfied so that \( π_2 \) completes.

From (1) in Lemma 6, we infer that every server in \( Q \) responds to the get message of \( π_2 \) with a tag that is at least as large as \( T(π_1) \). Because \( T(π_1) > T(π_2) \), the value returned by \( π_2 \) must have been obtained using the responses from servers in \( Q_2 \setminus Q_1 \). Thus \( |\{ u = T(π_2) | u = tag(r), \text{ for some } r \in R \}| \leq |Q_2 \setminus Q_1| \leq N - (N - f) = f \). Thus Line 15 cannot be satisfied.
Assume Line 16 is satisfied. Because every server in $Q$ responds with a tag that is at least as large as $T(\pi_1)$, which is greater than $T(\pi_2)$, and because $Q \subseteq Q_2$, we infer that the number of distinct response tags from $Q$ that are larger than $T(\pi_2)$ is at most $\nu$. Property (2) in Lemma 6 implies that there exists a tag $t > T(\pi_1)$ that appears in at least $k$ responses from $Q$. From the read protocol, we infer that the tag of the read operation should be at least $t$. That is, $T(\pi_2) \geq t$. But we know that $t \geq T(\pi_1) > T(\pi_2)$, which is a contradiction. □

> **Remark.** There can be multiple values that can be safely returned by a read operation. Indeed, as can be inferred from the proof of Lemma 7, any value satisfying Line 15 in Algorithm 1 can be returned safely, even if a higher tag can be recovered by the read operation.

> **Remark.** If $k > f$, the reader protocol is simplified so that Lines 15 and 16 are omitted. Indeed, any decodable tag has at least $k > f + 1$ coded symbols, which automatically satisfies Line 15 in Algorithm 1, and can be safely returned because of the previous remark.

> **Lemma 8.** Consider an execution $\alpha$ of Algorithm 1. Let $\pi_1$ be a write or read operation that completes in $\alpha$, and $\pi_2$ be a write operation that completes in $\alpha$. Let $T(\pi_1)$ denote the tag of operation $\pi_1$ and $T(\pi_2)$ denote the tag of operation $\pi_2$. If $\pi_2$ begins after the termination of $\pi_1$, then $T(\pi_2) > T(\pi_1)$.

**Proof.** The result follows from the single writer assumption and the fact that the writer increments its tag at the beginning of a new write in Line 3. □

> **Lemma 9.** Let $\pi_1, \pi_2$ be write operations that terminate in an execution $\alpha$ of Algorithm 1. Then, $T(\pi_1) \neq T(\pi_2)$.

The proof follows because of the SWMR setting and Lemma 8. The following theorem states the main result on atomicity, and the proof follows from Definitions 3 and 4, combined with Lemmas 2, 7, 8, and 9. The detailed proof is in [26].

> **Theorem 10.** Algorithm 1 emulates an atomic read-write object.

Next, we use Lemma 6 to show a safety property that will be used later.

> **Lemma 11.** Consider any execution $\alpha$ of Algorithm 1. Let $\pi_r$ denote a read operation in $\alpha$ that receives a quorum $Q_r$ of responses to its get message. Let $S$ denote the set of all writes that terminate before the invocation of $\pi_r$ in $\alpha$. If $S$ is non-empty, let $t_w$ denote the largest among the tags of the operations in $S$. If $S$ is empty, let $t_w = 0$.

If the number of writes concurrent with the read $\pi_r$ is smaller than $\nu$, then there is some tag $t$ such that

1. $t \geq t_w$,
2. $\pi_r$ receives at least $k$ responses to its get message with tag $t$, and
3. the number of tags that are higher than $t$ is smaller than $\nu$.

**Proof.** We first argue that, among the responses to the read’s get message from $Q_r$, the reader gets fewer than $\nu$ distinct response tags that are larger than $t_w$. By definition of the tag $t_w$, if the read receives a tag $t$ that is larger than $t_w$, then the tag $t$ corresponds to the tag of a write operation $\pi$ that is concurrent with $\pi_r$. Since the number of writes that are concurrent with the read is smaller than $\nu$, the read receives fewer than $\nu$ distinct tags that are larger than $t_w$.

We assume that $S \neq \emptyset$. The case $S = \emptyset$ can be treated in a similar way. Consider the write operation $\pi_w$ in $S$ whose tag is $t_w$. Let $Q_w$ denote the quorum of servers from which
responses were received by the writer to the put message of \( \pi_w \). Property (1) in Lemma 6 implies that every server \( s \in Q_w \cap Q_r \) responds with a tag that is at least as large as \( t_w \). Note that we have already shown that \( \pi_r \) receives fewer than \( \nu \) distinct tags to its get message that are larger than \( t_w \). That is, among the responses received by \( \pi_r \) from servers in \( Q_w \cap Q_r \) to its get message, there are at most \( \nu \) distinct tags. Property (2) of Lemma 6 implies the statement of the lemma, since it implies that there is at least one tag \( t \) which is no smaller than \( t_w \) such that at least \( k \) responses with tag \( t \) are received by \( \pi_r \). It can be seen that (3) holds.

▶ Remark. A write operation \( \pi_1 \) needs to be counted as “concurrent” with the read \( \pi_r \) only if its tag \( T(\pi_1) \) is larger than \( t_w \) defined in Lemma 11. In particular, suppose \( \pi_1 \) is a failed write operation, then it is not counted as concurrent with the read unless \( T(\pi_1) > t_w \).

### 3.3 Liveness Properties

We state the liveness of Algorithm 1. The detailed proofs are shown in [26]. Recall that we focus here on the SWMR algorithm.

▶ Theorem 12 (Termination of writes). Consider any fair execution \( \alpha \) of Algorithm 1 where the number of server failures is at most \( f \), and the write client does not fail. Then, every write operation terminates in \( \alpha \).

▶ Theorem 13 (Termination of reads). Consider any fair execution \( \alpha \) of Algorithm 1 where the number of server failures is at most \( f \). Consider any read operation that is invoked at a non-failing client in \( \alpha \). If the number of writes concurrent with the read is strictly smaller than \( \nu \), and the read client does not fail, then the read operation completes in \( \alpha \).

▶ Remark. The condition of concurrency being smaller than \( \nu \) in Theorem 13 is a sufficient condition for the termination of a read operation, but it is not necessary. Indeed, as highlighted in the remarks of Section 3.2, the reader may safely return a value \( v \) with tag \( t \) when the number of tags strictly higher than \( t \), denoted by \( u \), satisfies \( u \geq \nu \).

▶ Remark. In practice, if a read operation aborts, the reader in Algorithm 1 can repeatedly invoke new read operations until it can decode a value that it can safely return. In this case, it can be shown that Algorithm 1 satisfies finite-write termination liveness. We refer the reader to [26] for more details.

### 3.4 Storage and Communication Costs of Algorithm 1

By the design of Algorithm 1, the following result follows.

▶ Theorem 14. The worst-case storage cost of Algorithm 1 is \( \frac{N}{k} \). The write communication cost is \( \frac{N}{k} \). The read communication cost is \( 2\frac{N}{k} \).

## 4 Multi-Writer Multi-Reader Algorithm

### 4.1 Algorithm Description

Assume in this section that \( \nu < N - 2f, \nu \geq 1 \) and thus \( k > 1 \). We extend Algorithm 1 to the MRMW setting, in which we assume the presence of an arbitrary number of writers,
Algorithm 2: MWMR setting

Write client protocol
1: function write(v)
2: query phase
3: for s ∈ {1, 2, . . . , N} do
4: wait until receive responses from a quorum.
5: Select the largest tag from the query phase; let its integer component be z.
6: Form a new tag t as (z + 1, id), where id is the identifier of the client performing the operation.
7: pre-write phase
8: for s ∈ {1, 2, . . . , k + 2f} do
9: wait until receive responses from k + f servers.
10: finalize phase
11: let (y1, y2, . . . , yN) = Φ(v0).
12: for s ∈ {1, 2, . . . , N} do
13: wait until receive responses from a quorum.
14: end function

Read client protocol
15: function read()
16: for s ∈ {1, 2, . . . , N} do
17: Send get() message to server s
18: wait until receive responses from a quorum.
19: Let R be the set of response pairs.
20: Let T be the set of decodable tags t occurring in R such that
21: (i) t appears in at least f + 1 responses,
22: (ii) or, the number of tags strictly higher than t is at most ν.
23: if T ̸= ∅ then
24: Let t = max(T), and v its value.
25: write-back phase
26: for s ∈ {1, 2, . . . , k + 2f} do
27: wait until receive responses from k + f servers.
28: Let (y1, y2, . . . , yN) = Φ(v).
29: for s ∈ {1, 2, . . . , N} do
30: send put(t, v) to server s
31: wait until receive responses from a quorum.
32: return v.
33: else
34: _abort_.
35: end if
36: end function

Server s protocol
37: state variable at server s: A pair (t, x), where t ∈ T, x ∈ V ∪ W.
38: Initially, server s stores (t0, y), where y is the s-th component of Φ(v0).
39: Upon receipt of get_tag() do
40: respond with the stored tag t.
41: Upon receipt of get() do
42: respond with (t, *), where * can be a coded symbol or a full value.
43: Upon receipt of put(tnew, vnew) such that vnew ∈ V, do
44: If tnew > t, then set t ← tnew and x ← vnew. In any case respond with acknowledgement.
45: Upon receipt of put(tnew, ynew) such that ynew ∈ W, do
46: If tnew ≥ t, then set t ← tnew and x ← ynew. In any case respond with acknowledgement.
and an arbitrary number of readers. The proposed algorithm, referred to as Algorithm 2, combines replication and multi-version codes while achieving consistency and low storage costs.

Algorithm 2 contains a description of the protocol. Each server maintains a \((tag, element)\) tuple, where \(element\) can be a full replica or a coded symbol. We assume that tags are tuples of the form \((z, id)\), where \(z\) is an integer and \(id\) is an identifier of a write client. The ordering on the set of tags \(T\) is defined lexicographically, using the usual ordering on the integers and a predefined ordering on the client identifiers. In the write protocol, the \(query\) phase first obtains the tags of the servers, in order to generate a higher tag. In the \(pre-write\) phase, a writer propagates a full replica to at least \(k + f\) servers, to ensure that the consistency of the data is not compromised in the presence of concurrent writers. In the \(finalize\) phase, coded symbols are sent to a quorum of servers. The servers only maintain the highest tag, and the corresponding element. As \(k = \lceil \frac{N - 2f}{\nu} \rceil\), it follows that \(N - f \geq f + k + (\nu - 1)k \geq f + k\). Hence, the \(pre-write\) quorum is smaller than the \(finalize\) quorum. The read protocol of Algorithm 2 is essentially similar to Algorithm 1, except that a reader can receive coded symbols and/or replicas. In particular, a decodable tag may come from a replica and/or \(k\) matching coded symbols. The \(write-back\) in the read is the same as the write protocol, but with no \(query\) phase.

\[\begin{align*}
\text{Remark.} & \quad \text{During the } finalize \text{ phase of the write protocol, the writer does not need to send coded symbols to all the servers. Indeed, the writer sends full replicas to the first } k + 2f \text{ servers in the } pre-write \text{ phase. Then, during the } finalize \text{ phase, the writer sends coded symbols to the remaining } N - k - 2f \text{ servers, and only a finalize message with the corresponding tag to the first } k + 2f \text{ servers so as to minimize the communication cost.}
\end{align*}\]

### 4.2 Safety Properties

\[\begin{align*}
\text{Definition 15 (Tag of an operation } \pi \text{).} & \quad \text{The tag of operation } \pi \text{ in an execution } \alpha \text{ is defined to be the tag associated with the put messages that the operation propagates to the servers. The put message used in the definition can be either from a pre-write or a finalize phase since they have the same tag.}
\end{align*}\]

\[\begin{align*}
\text{Lemma 16 (Persistence of data).} & \quad \text{The value with the highest tag can be fully recovered at any point of an execution, as long as the number of server failures is bounded by } f. \\
\text{Proof.} & \quad \text{The Lemma follows using the fact that writers propagate full replicas to at least } k + f \text{ servers in the } pre-write \text{ phase. See [26] for details.}
\end{align*}\]

The proof of atomicity follows along the lines of the SWMR algorithm, with minor modifications (see [26]). In particular, the statements of Lemmas 6, 7, and 8 hold by considering the write quorum to be the \(finalize\) quorum. The statement of Lemma 9 follows from the \(query\) phase.

\[\begin{align*}
\text{Theorem 17.} & \quad \text{Algorithm 2 emulates an atomic read-write object.}
\end{align*}\]

The statement of the safety property in Lemma 11 holds also in the setting of Algorithm 2, and can be used to prove the liveness properties of Algorithm 2.

### 4.3 Liveness Properties

The proofs of the liveness properties of Algorithm 2 are similar to those of Algorithm 1, and are omitted.
Theorem 18 (Termination of writes). Consider any fair execution $\alpha$ of Algorithm 2 where the number of server failures is at most $f$, and the write client does not fail. Then, every write operation terminates in $\alpha$.

Theorem 19 (Termination of reads). Consider any fair execution $\alpha$ of Algorithm 2 where the number of server failures is at most $f$. Consider any read operation that is invoked at a non-failing client in $\alpha$. If the number of writes concurrent with the read is strictly smaller than $\nu$, and the read client does not fail, then the read operation completes in $\alpha$.

Remark. Similar to Algorithm 1, if a read operation aborts, the reader in Algorithm 2 can repeatedly invoke new read operations until it can decode a value that it can safely return. It can be also shown that Algorithm 2 guarantees in this case finite-write termination [26].

4.4 Storage and Communication Costs of Algorithm 2

Theorem 20. The worst-case storage cost of Algorithm 2 is $k + 2f + \frac{N - k - 2f}{k}$. The steady-state storage cost is given by $\frac{N}{k}$. The write communication cost is $k + 2f + \frac{N - k - 2f}{k}$. The read communication cost is $2(k + 2f + \frac{N - k - f}{k})$. Here $k = \left\lceil \frac{N - 2f}{\nu} \right\rceil$.

4.5 Algorithm 2-A: Algorithm with Asymptotically Optimal Storage Cost

In this section, we assume that $\nu \geq 2$ and $k > 1$. We present a MRMW algorithm that is similar to Algorithm 2, except that the write procedure is limited to two phases. We further assume throughout this section that the following condition is satisfied:

Condition 1: at any point during the execution, the number of concurrent writes is smaller than $\nu$.

Algorithm 2-A description: By virtue of Condition 1, the write protocol in Algorithm 2 can be simplified as the need for a pre-write phase is obviated. After acquiring its tag, a writer propagates coded symbols to a quorum of servers before terminating. The read and server protocols are exactly the same as in Algorithm 1.

We note that if the number of writers is less than $\nu$ (e.g., in applications with predetermined write clients), then Condition 1 is ensured by default. In particular, the SWMR setting is a special case of Condition 1. That is the reason that Algorithm 2-A is almost the same as Algorithm 1.

The proof techniques used for Algorithm 1 hold for Algorithm 2-A. For instance, the persistence of data property follows from Lemma 11. The only difference lies in showing that any two write operations have distinct tags, which follows from the query phase of Algorithm 2-A. Finally, atomicity and liveness, i.e., Theorems 17, 18, and 19 can be proved for Algorithm 2-A.

Theorem 21. The storage cost of Algorithm 2-A is $\left\lceil \frac{N}{N-\nu} \right\rceil$ at any point in any execution, which is asymptotically optimal for $N \gg f, 1 \ll \nu \leq f + 2$.

Proof. The storage cost at any point is $\left\lfloor \frac{N}{N-\nu} \right\rfloor = \frac{N}{1 + \frac{\nu - 1}{N - f - \nu}} = \frac{\nu N}{N - 2f + \nu - 1}$ by Equation (1). On the other hand, a careful analysis of [6, Theorem 6.5] shows that a worst-case storage lower bound of $\frac{(\nu - 1)N}{N - f + \nu - 2}$ holds under the following weaker liveness requirement:

\footnote{In [6, Theorem 6.5], the algorithm is restricted to one that write protocols consist of phases and at most one phase in a write operation sends messages dependent on the object value, which is satisfied by our Algorithm 2-A.}
**Liveness requirement for [6, Theorem 6.5]:** Consider a fair execution where the number of server failures is no more than \(f\), the number of writers is less than \(\nu\), \(\nu \leq f + 2\), and there is one reader. Each client invokes at most 1 operation. A write operation should terminate if the writer does not fail. If, at a point in the execution, all the writers fail, and a read operation is invoked, then the read operation should terminate.

In particular, this liveness condition is satisfied by Algorithm 2-A under Condition 1.

When \(N \gg f, 1 \ll \nu \leq f + 2\), the worst-case storage of Algorithm 2-A asymptotically matches the lower bound, i.e., \(\frac{\nu N}{N-2f+\nu-1} \approx \frac{\nu N}{N-2f+\nu-2}\).

## 5 Discussion and Conclusion

Different from most previous erasure-code-based algorithms, our algorithms are parameterized by the liveness parameter \(\nu\). We discuss now the merits and the disadvantages of our approach. We mainly focus on MWMR algorithms listed in Table 1.

We compare Algorithm 2 with ABD. While Algorithm 2 has higher worst-case storage than ABD, for \(\nu < 2f + 1\) its steady-state storage outperforms ABD by

\[
2f + 1 - \frac{N}{k} = \frac{(N - 2f - 1)(2f + 1 - \nu)}{N - 2f + \nu - 1} = \frac{k - 1}{k}(2f + 1 - \nu).
\]

(2)

Compared to ABD, our algorithms make use of a higher number of servers to offer storage reduction, at the expense of liveness if a read operation is concurrent with at least \(\nu\) write operations.

We compare Algorithm 2 to CASGC [5] with parameters \((k_{\text{CASGC}}, \delta) = (N - 2f, \nu - 1)\). Both algorithms satisfy \(\nu\)-concurrence wait-freedom. CASGC has a steady-state storage given by \(\frac{\nu N}{N-2f}\). However, the worst-case storage of CASGC depends on the write concurrency at a point and can be unbounded. Algorithm 2 is advantageous compared to CASGC. It offers: (i) smaller steady-state storage size given by \(N/\lceil\frac{\nu N}{N-2f}\rceil\), which can be close to twice as efficient (see Example 22), and (ii) bounded worst-case storage.

**Example 22.** Consider the case of \(\nu = N - 2f - 1\). Then, Algorithm 2 uses coding parameter \(k = 1 + \frac{N - (2f + 1)}{N - 2f + 1} = 2\). The steady-state storage of Algorithm 2 is then \(\frac{N}{2}\).

The steady-state storage of CASGC with parameters \((k_{\text{CASGC}}, \delta) = (N - 2f, \nu - 1)\) is \(\frac{\nu N}{N - 2f} = N(1 - \frac{1}{N - 2f})\), which goes to \(N\) as \(N - 2f\) increases. Therefore, Algorithm 2 can be close to twice as efficient as CASGC. Moreover, for the same choice of \(\nu = N - 2f - 1\), whenever \(\nu < 2f + 1 \iff N < 4f + 2\), Algorithm 2 improves upon ABD in terms of steady-state storage.

Next we compare Algorithm 2 with SCCK [24], which is a MWMR algorithm that combines replication and erasure codes. Assume its coding parameter is \(k_{\text{SCCK}} = N - 2f\), such that the steady-state storage is minimized. The worst-case storage of SCCK is \(2N\), which is at least twice the storage of ABD and Algorithm 2. Meanwhile, SCCK has a steady-state storage \(\frac{N}{2N - 2f}\), which is lower than Algorithm 2. Note that SCCK only provides finite-write termination guarantees.

For \(\nu < 2f + 1, N \gg f\), in descending order of the worst-case storage cost, we have CASGC, SCCK, Algorithm 2 and ABD. In descending order of the steady-state storage cost, we have ABD, CASGC, Algorithm 2, and SCCK.

For executions such that there are less than \(\nu\) concurrent writes at any point, a worst-case storage lower bound of \(\Omega(min(f, \nu))\) is given in [24] under lock-freedom, which is a weaker liveness property than \(\nu\)-concurrence wait-freedom. Algorithm 2-A also meets this bound.
when $N \gg f$, $1 \ll \nu \leq f+2$. Other algorithms have also been proposed that match this bound asymptotically. For example, the server can store all the concurrent coded symbols with coding parameter $N - 2f$, achieving the worst-case storage of $\frac{\nu N}{N-2f}$ (e.g. [5], and [24] for $N \gg f, \nu \leq f+1$). Algorithm 2-A has a smaller multiplicative constant by a factor of up to 2, which can significantly reduce the cost for systems such as memory-based data stores.

In conclusion, we proposed MASIN, fault-tolerant algorithms for emulating a shared memory over an asynchronous, distributed message-passing network. Our algorithms guarantee liveness of the read as long as the number of writes concurrent with the read is smaller than a design parameter $\nu$. The parameter $\nu$ illustrates the tradeoff between liveness of read operations and the storage size per node. The overall steady-state storage and communication costs of our algorithms outperform ABD when $\nu < 2f+1$. While the steady-state storage of the MRMW algorithms presented in this paper is higher compared to some other coding-based schemes, the in-place update for each write operation and the simple structure of the algorithms make them appealing from practical perspective and easy to implement. We note that the worst-case storage of Algorithm 2 is incurred during each write operation. Hence, an open problem is how to dynamically adapt to the concurrency level when it changes over time, so that the liveness condition is strengthened and the storage cost is lowered.

References

10 Dan Dobre, Ghassan Karame, Wenting Li, Matthias Majunke, Neeraj Suri, and Marko Vukolić. PoWerStore: proofs of writing for efficient and robust storage. In *Proceedings of
MASIN: Multi-version codes for Atomic Storage with INplace-update


