A Coded Shared Atomic Memory Algorithm for Message Passing Architectures

Viveck R. Cadambe viveck@mit.edu
Nancy Lynch lynch@csail.mit.edu
Muriel Médard medard@mit.edu
Peter Musial peter.musial@emc.com

***Abstract*—This paper considers the communication and storage costs of emulating atomic (linearizable) multi-writer multi-reader shared memory in distributed message-passing systems. The paper contains two main contributions:**

(1) We present an atomic shared-memory emulation algorithm that we call *Coded Atomic Storage* (CAS). This algorithm uses *erasure coding* methods. In a storage system with N servers that is resilient to f server failures, we show that the communication cost of CAS is $\frac{N}{N-2f}$. The storage cost of CAS is unbounded.

(2) We present a variant of CAS known as CAS with Garbage Collection (CASGC). The CASGC algorithm is parametrized by an integer δ and has a bounded storage cost. We show that in every execution where the number of write operations that are concurrent with a read operation is no bigger than δ, the CASGC algorithm with parameter δ satisfies atomicity and liveness. We explicitly characterize the storage cost of CASGC, and show that it has the same communication cost as CAS.

I. Introduction

Since the late 1970s, emulation of shared-memory systems in distributed message-passing environments has been an active area of research [1]–[6], [8]–[10], [12]–[14], [19], [22], [23]. The traditional approach to building redundancy for distributed systems in the context of shared memory emulation is *replication*. In their seminal paper [5], Attiya, Bar-Noy, and Dolev presented a replication based algorithm for emulating shared memory that achieves atomic consistency [15], [16]. In this paper we consider a simple multi-writer generalization of their algorithm which we call the *ABD* algorithm[i]. This algorithm uses a quorum-based replication scheme, combined with read and write protocols to ensure that the emulated object is atomic [16] (linearizable [15]), and to ensure liveness, specifically, that each operation terminates provided that at most $\lceil \frac{N-1}{2} \rceil$ server nodes fail. Since the read and write protocols require multiple communication phases where entire replicas are sent, the ABD algorithm has a high communication cost.

The main goal of our paper is to develop shared memory emulation algorithms, based on the idea of *erasure coding*, that are efficient in terms of communication and storage costs. Erasure coding is a generalization of replication that is well known in the context of classical storage systems [17], [20], [21]. Specifically, in erasure coding, each server does not store the value in its entirety, but only a part of the value called a *coded element*. In the classical coding theory framework which studies storage of a single version of a data object, this approach is well known to lead to smaller storage costs as compared to replication (see Section III). Algorithms for shared memory emulation that use the idea of erasure coding to store multiple versions of a data object consistently have been developed in [1]–[3], [6], [8], [9], [14], [22]. In this paper, we build on the existing literature by developing new erasure coding based shared memory emulation algorithms and formally quantifying their costs. We next summarize our contributions and compare them with previous related work.

Contributions. We consider a static distributed message-passing setting where the universe of nodes is fixed and known, and nodes communicate using a reliable message-passing network. We assume that client and server nodes can fail. We define our system model, and communication and storage cost measures in Section II.

The CAS algorithm: We develop the *Coded Atomic Storage* (CAS) algorithm presented in Section IV, which is an erasure coding based shared memory emulation algorithm. We present a brief introduction of erasure coding in Section III. For a storage system with N nodes, we show[ii] in Theorem 6 that CAS ensures the following liveness property: all operations that are invoked by a non-failed client terminate provided that the number of *server* failures is bounded by a parameter f, where $f < \lceil \frac{N}{2} \rceil$ and regardless of the number of client failures. We also show in Lemma 6 that CAS ensures atomicity regardless of the number of (client or server) failures. In Theorem 2 in Section IV, we also analyze the communication cost of CAS. Specifically, in a storage system with N servers that is resilient to f server node failures, we show that the communication costs of CAS are equal to $\frac{N}{N-2f}$. We note that these communication costs of CAS are smaller than replication based schemes (see extended version of this paper [7]). The storage cost of CAS, however, is unbounded because each server stores the value associated with every version of the data object it receives. In comparison, in ABD which is based on replication, the storage cost is bounded because each node stores only the latest version of the data object (see [7]).

The CASGC algorithm: In Section V, we present a variant of CAS called the CAS with Garbage Collection (CASGC) algorithm, which achieves a bounded storage cost by *garbage collection*, i.e., discarding values associated with sufficiently old versions. CASGC is parametrized by an integer δ which, informally speaking, controls the number of tuples that each server stores. We show that CASGC satisfies atomicity in

Dr. Viveck R. Cadambe and Prof. Muriel Médard are with the Research Laboratory of Electronics (RLE), at the Massachusetts Institute of Technology, Cambridge MA, USA. Prof. Nancy Lynch is with the Computer Science and Artificial Intelligence Laboratory (CSAIL), Massachusetts Institute of Technology (MIT), Cambridge MA, USA. Dr. Peter Musial is with the Advanced Storage Division, EMC Corporation, Cambridge MA, USA.

This work was supported by in part by AFOSR contract no. FA9550-13-1-0042, NSF award no.s CCF-1217506 and 0939370-CCF, and by BAE Systems National Security Solutions Inc. award 739532-SLIN 0004.

[i]The algorithm of Attiya, Bar-Noy and Dolev [5] allows only a single node to act as a writer. Also, it does not distinguish between client and server nodes as we do in our paper.

[ii]We only provide brief sketches of the proofs of our results here. Full proofs of our theorems can be found in the extended version of this paper [7].

Theorem 3 by establishing a formal simulation relation [18] between CAS and CASGC. Because of the garbage collection at the servers, the liveness conditions for CASGC are more stringent than CAS. The liveness property satisfied by CASGC is described in Theorem 4 in Section V, where we argue that in an execution of CASGC where the number of write operations concurrent with a read operation is no bigger than a parameter δ, every operation terminates. The main technical challenge lies in careful design of the CASGC algorithm in order to ensure that an unbounded number of writes that fail do not prevent a future read from returning a value of the data object. In particular, failed writes that begin and end before a read is invoked are not treated as operations that are concurrent with the read, and therefore do not contribute to the concurrency limit of δ. While CASGC incurs the same communication costs as CAS, it incurs a bounded storage cost. A non-trivial bound on the storage cost of CASGC is described in Theorem 5. In particular, we note that a larger value of the parameter δ implies a larger storage cost.

Comparison with Related Work. Erasure coding has been used to develop shared memory emulation techniques for systems with crash failures in [2], [3], [9], [22] and Byzantine failures in [1], [6], [8], [14]. In erasure coding, note that each server stores a coded element, so a reader has to obtain enough coded elements to decode and return the value. The main challenge in extending replication based algorithms such as ABD to erasure coding lies in handling partially completed or failed writes. In replication, when a read occurs during a partially completed write, servers simply send the stored value and the reader returns the latest value obtained from the servers. However, in erasure coding, the challenge is to ensure that a read that observes the trace of a partially completed or failed write obtains a enough coded elements corresponding to the same version to return a value. Different algorithms have different approaches in handling this challenge of ensuring that the reader decodes a value of the data object. As a consequence, the algorithms differ in the liveness properties satisfied, and the communication and storage costs incurred. We discuss the differences here briefly.

Among the previous works, [6], [8], [9], [14] have similar correctness requirements as our paper; these references aim to emulate an atomic shared memory that supports concurrent operations in asynchronous networks. We note that the algorithm of [6] cannot be generalized to lossy channel models (see discussion in [9]). We compare our algorithms with the *ORCAS-B* algorithm of [9][iii], the algorithm of [14], which we call the *HGR* algorithm, and the *M-PoWerStore* algorithm of [8]. We note that [9] assumes lossy channels and [8], [14] assume Byzantine failures. Here, we interpret the algorithms of [8], [9], [14] in our model that has lossless channels and crash failures, and use worst-case costs for comparison.

The CAS and CASGC algorithms resemble the M-PoWerStore and HGR algorithms in their structure. These algorithms handle partially completed or failed writes by *hiding* ongoing writes from a read until enough number of coded elements have been propagated to the servers. The write communication costs of CAS, CASGC, M-PoWerStore, HGR and ORCAS-B are all the same. However, there are differences between these algorithms in the liveness properties, garbage collection strategies and read communication costs.

CAS is essentially a restricted version of the *M-PoWerStore* algorithm of [8] for the crash failure model. The main difference between CAS and M-PoWerStore is that in CAS, servers perform gossip[iv]. However, M-PoWerStore does not involve garbage collection and therefore incurs an infinite storage cost. The garbage collection strategies of HGR and ORCAS-B are similar to that of CASGC with the parameter δ set to 1. In fact, the garbage collection strategy of CASGC may be viewed as a non-trivial generalization of the garbage collection strategies of ORCAS-B and HGR. We next discuss differences between these algorithms in terms of their liveness properties and communication costs.

The ORCAS-B algorithm satisfies the same liveness properties as ABD and CAS, which are stronger than the liveness conditions of CASGC. However, in ORCAS-B, to handle partially completed writes, a server sends coded elements corresponding to multiple versions to the reader. This is because, in ORCAS-B, a server, on receiving a request from a reader, registers the client and sends all the incoming coded elements to the reader until the read receives a second message from a client. Therefore, the read communication cost of ORCAS-B grows with the number of writes that are concurrent with a read. In fact, in ORCAS-B, if a read client fails in the middle of a read operation, servers may send all the coded elements it receives from future writes to the reader. In contrast, CAS and CASGC have smaller communication costs because each server sends only one coded element to a client per read operation, irrespective of the number of writes that are concurrent with the read.

In HGR, read operations satisfy *obstruction freedom*, that is, a read returns if there is a period during the read where no other operation takes steps for sufficiently long. Therefore, in HGR, operations may terminate even if the number of writes concurrent with a read is arbitrarily large, but it requires a sufficiently long period where concurrent operations do not take steps. On the contrary, in CASGC, by setting δ to be bigger than 1, we ensure that read operations terminate even if concurrent operations take steps, albeit at a larger storage cost, so long as the number of writes concurrent with a read is bounded by δ. Interestingly, the read communication cost of HGR is larger than CASGC, and increases with the number of writes concurrent to the read to allow for read termination in presence of a large number of concurrent writes.

We note that the server protocol of the CASGC algorithm is more complicated as compared with previous algorithms. In particular, unlike ORCAS-B, HGR and M-PoWerStore, the CASGC algorithm requires gossip among the servers to ensure operation termination in presence of multiple writes at a bounded storage cost and low communication cost. Another feature that distinguishes our paper is that we provide rigorous definitions and analysis of communication and storage costs. We also include correctness proofs of our algorithms through the development of invariants and simulation relations, which may be of independent interest.

[iii] The *ORCAS-A* algorithm of [9], although uses erasure coding, has the same *worst case* communication and storage costs as ABD.

[iv] As we shall see later, the server gossip is not essential to correctness of CAS. It is however useful as a theoretical tool to prove correctness of CASGC.

II. Problem Statement

Deployment setting. We assume a *static asynchronous deployment setting* where all the nodes and the network connections are known a priori and the only sources of dynamic behavior are node stop-failures (or simply, failures) and processing and communication delays. We consider a message-passing setting where nodes communicate via point-to-point reliable channels. We assume a universe of nodes that is the union of *server* and *client* nodes, where the client nodes are *reader* or *writer* nodes. $\mathcal{N}$ represents the set of server nodes; N denotes the cardinality of $\mathcal{N}$. We assume that server and client nodes can fail (stop execution) at any point[v]. We assume that the number of server node failures is at most f. There is no bound on the number of client failures.

Shared memory emulation. We consider algorithms that emulate multi-writer, multi-reader (MWMR) read/write atomic shared memory using our deployment platform. We assume that read clients receive read requests (invocations) from some local external source, and respond with object values. Write clients receive write requests and respond with acknowledgments. The requests follow a "handshake" discipline, where a new invocation at a client waits for a response to the preceding invocation at the same client. We require that the overall external behavior of the algorithm corresponds to atomic (linearizable) memory. For simplicity, in this paper we consider a shared-memory system that consists of just a single object.

We represent each version of the data object as a $(tag, value)$ pair. When a write client processes a write request, it assigns a *tag* to the request. We assume that the tag is an element of a totally ordered set $\mathcal{T}$ that has a minimum element t_0. The tag of a write request serves as a unique identifier for that request, and tags associated with successive write requests at a particular client increase monotonically. We assume that $value$ is a member of a finite set $\mathcal{V}$ that represents the set of values that the data object can take on; note that $value$ can be represented by $\log_2 |\mathcal{V}|$ bits[vi]. We assume that all servers are initialized with a default initial state.

Requirements. The key correctness requirement on the targeted shared memory service is *atomicity.* A shared atomic object is one that supports concurrent access by multiple clients and where the observed global external behaviors "look like" the object is being accessed sequentially. Another requirement is *liveness*, by which we mean here that an operation of a non-failed client is guaranteed to terminate provided that the number of server failures is at most f, and irrespective of the failures of other clients[vii].

Communication cost. Informally speaking, the communication cost is the number of bits transferred over the point-to-point links in the message-passing system. For a message that can take any value in some finite set $\mathcal{M}$, we measure its communication cost as $\log_2 |\mathcal{M}|$ bits. We separate the cost of communicating a value of the data object from the cost of communicating the tags and other metadata. Specifically, we assume that each message is a triple (t, w, d) where $t \in \mathcal{T}$ is a tag, $w \in \mathcal{W}$ is the (only) component of the triple that depends on the value associated with tag t, and $d \in \mathcal{D}$ is any additional metadata that is independent of the value. Here, $\mathcal{W}$ is a finite set of values that the second component of the message can take on, depending on the value of the data object. $\mathcal{D}$ is a finite set that contains all the possible metadata elements for the message. These sets are assumed to be known a priori to the sender and recipient of the message. In this paper, we make the approximation: $\log_2 |\mathcal{M}| \approx \log_2 |\mathcal{W}|$, that is, the costs of communicating the tags and the metadata are negligible as compared to the cost of communicating the data object values. We assume that every message is sent on behalf of some read or write operation. We next define the read and write communication costs of an algorithm.

For a given shared memory algorithm, consider an execution α. The communication cost of a write operation in α is the sum of the communication costs of all the messages sent over the point-to-point links on behalf of the operation. The write communication cost of the execution α is the supremum of the costs of all the write operations in α. The write communication cost of the algorithm is the supremum of the write communication costs taken over all executions. The read communication cost of an algorithm is defined similarly.

Storage cost. Informally speaking, at any point of an execution of an algorithm, the *storage cost* is the total number of bits stored by the servers. Specifically, we assume that a server node stores a set of triples with each triple of the form (t, w, d), where $t \in \mathcal{T}$, w depends on the value of the data object associated with tag t, and d represents additional metadata that is independent of the values stored. We neglect the cost of storing the tags and the metadata; so the cost of storing the triple (t, w, d) is measured as $\log_2 |\mathcal{W}|$ bits. The storage cost of a server is the sum of the storage costs of all the triples stored at the server. For a given shared memory algorithm, consider an execution α. The storage cost at a particular point of α is the sum of the storage costs of all the non-failed servers at that point. The storage cost of the execution α is the supremum of the storage costs over all points of α. The storage cost of an algorithm is the supremum of the storage costs over all executions of the algorithm.

III. Erasure Coding - Background

Erasure coding is a generalization of replication that has been widely studied for purposes of failure-tolerance in storage systems (see [17], [20], [21]). The key idea of erasure coding involves splitting the data into several *coded elements*, each of which is stored at a different server node. As long as a sufficient number of coded elements can be accessed, the original data can be recovered. Informally speaking, given two positive integers m, k, $k < m$, *an* (m, k) *Maximum Distance Separable (MDS) code maps a* k*-length vector to an* m*-length vector, where the input* k*-length vector can be recovered from any* k *coordinates of the output* m*-length vector.* This implies that an (m, k) code, when used to store a k-length vector on m server nodes - each server node storing one of the m coordinates of the output - can tolerate $(m-k)$ node failures in the absence of any consistency requirements. We next define the notion of an MDS code formally.

Given an arbitrary finite set $\mathcal{A}$ and any set $S \subseteq \{1, 2, \ldots, m\}$, let π_S denote the *natural projection mapping* from $\mathcal{A}^m$ onto the coordinates corresponding to S, i.e., denoting $S = \{s_1, s_2, \ldots, s_{|S|}\}$, where $s_1 < s_2 \ldots < s_{|S|}$, the

[v] Our model can be formally described as an I/O automaton by composing the automata of all the nodes and channels in the system. Our notions of an *execution*, a *fair execution*, etc. follow from the standard definitions [18].

[vi] Strictly speaking, we need $\lceil \log_2 |\mathcal{V}| \rceil$ bits since the number of bits has to be an integer. We ignore this rounding error.

[vii] We assume $N > 2f$, since correctness cannot be ensured if $N \leq 2f$ [18].

function $\pi_S : \mathcal{A}^m \rightarrow \mathcal{A}^{|S|}$ is defined as $\pi_S(x_1, x_2, \ldots, x_m) = (x_{s_1}, x_{s_2}, \ldots, x_{s_{|S|}})$.

Definition 1 (Maximum Distance Separable (MDS) code)**.** *Let $\mathcal{A}$ denote any finite set. For positive integers k, m such that $k < m$, an (m, k) code over $\mathcal{A}$ is a map $\Phi : \mathcal{A}^k \rightarrow \mathcal{A}^m$. An (m, k) code Φ over $\mathcal{A}$ is said to be* Maximum Distance Separable *(MDS) if, for every $S \subseteq \{1, 2, \ldots, m\}$ where $|S| = k$, there exists a function $\Phi_S^{-1} : \mathcal{A}^k \rightarrow \mathcal{A}^k$ such that: $\Phi_S^{-1}(\pi_S(\Phi(\mathbf{x})) = \mathbf{x}$ for every $\mathbf{x} \in \mathcal{A}^k$, where π_S is the natural projection mapping.*

We refer to each of the m coordinates of the output of an (m, k) code Φ as a *coded element*. Classical m-way replication, where the input value is repeated m times, is in fact an $(m, 1)$ MDS code. Another example is the *single parity code*: an $(m, m-1)$ MDS code over $\mathcal{A} = \{0, 1\}$ which maps the $(m-1)$-bit vector $x_1, x_2, \ldots, x_{m-1}$ to the m-bit vector $x_1, x_2, \ldots, x_{m-1}, x_1 \oplus x_2 \oplus \ldots \oplus x_{m-1}$.

We review the classical coding-theoretic model, where a single version of a data object with value $v \in \mathcal{V}$ is stored over N servers using an (N, k) MDS code. We assume that $\mathcal{V} = \mathcal{W}^k$ for some finite set $\mathcal{W}$ and that an (N, k) MDS code $\Phi : \mathcal{W}^k \rightarrow \mathcal{W}^N$ exists over $\mathcal{W}$ (see [7] for a discussion). The value v of the data object can be used as an input to Φ to get N coded elements over $\mathcal{W}$; each of the N servers, respectively, stores one of these coded elements. Since each coded element belongs to the set $\mathcal{W}$, whose cardinality satisfies $|\mathcal{W}| = |\mathcal{V}|^{1/k} = 2^{\frac{\log_2 |\mathcal{V}|}{k}}$, *each coded element can be represented as a $\frac{\log_2 |\mathcal{V}|}{k}$ bit-vector, i.e., the number of bits in each coded element is a fraction $\frac{1}{k}$ of the number of bits in the original data object*. When we employ an (N, k) code in the context of storing multiple versions, the size of a coded element is closely related to communication and storage costs incurred by our algorithms (see Theorems 2 and 5).

IV. Coded Atomic Storage

We now present the *Coded Atomic Storage* (CAS) algorithm which takes advantage of erasure coding techniques to reduce the communication cost for emulating atomic shared memory. CAS is parameterized by an integer k, $1 \leq k \leq N - 2f$; we denote the algorithm with parameter value k by CAS(k). CAS, like ABD and LDR, is a quorum-based algorithm. Later, in Sec. V, we present a variant of CAS that has efficient storage costs as well (in addition to having the same communication costs as CAS). We next describe CAS.

Quorum specification. We define our quorum system, $\mathcal{Q}$, to be the set of all subsets of $\mathcal{N}$ that have at least $\lceil \frac{N+k}{2} \rceil$ elements (server nodes). We refer to the members of $\mathcal{Q}$, as quorum sets. The following property is shown in [7], [11]:

Lemma 1. *Suppose that $1 \leq k \leq N - 2f$. (**i**) If $Q_1, Q_2 \in \mathcal{Q}$, then $|Q_1 \cap Q_2| \geq k$. (**ii**) If the number of failed servers is at most f, then $\mathcal{Q}$ contains at least one quorum set Q of non-failed servers.*

The CAS algorithm can, in fact, use any quorum system that satisfies properties (**i**) and (**ii**) of Lemma 1.

Algorithm description. In CAS, we assume that tags are tuples of the form $(z, \text{'id'})$, where z is an integer and 'id' is an identifier of a client node. The ordering on the set of tags $\mathcal{T}$ is defined lexicographically, using the usual ordering on the integers and a predefined ordering on the client identifiers.

We add a 'gossip' protocol to CAS, whereby each server sends each *item* from $\mathcal{T} \times \{\text{'fin'}\}$ that it ever receives once (immediately) to every other server. As a consequence, in any fair execution, if a non-failed server initiates 'gossip' or receives 'gossip' message with item $(t, \text{'fin'})$, then, every non-failed server receives a 'gossip' message with this item at some point of the execution. Fig. 1 contains a description of the read and write protocols, and the server actions of CAS. Here, we provide an overview of the algorithm.

Each server node maintains a set of $(tag, coded\text{-}element, label)$[viii] triples, where we specialize the metadata to $label \in \{\text{'pre'}, \text{'fin'}\}$. 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 *query* phase is identical in both protocols and it allows clients to discover a recent *finalized object version*, i.e., a recent version with a 'fin' tag. The goal of the *pre-write* phase of a write is to ensure that each server gets a tag and a coded element with label 'pre'. Tags associated with label 'pre' are not visible to the readers, since the servers respond to *query* messages only with finalized tags. Once a quorum, say Q_{pw}, has acknowledged receipt of the coded elements to the pre-write phase, the writer proceeds to its *finalize* phase. In this phase, it propagates a finalize ('fin') label with the tag and waits for a response from a quorum, say Q_{fw}. The purpose of propagating the 'fin' label is to record that the coded elements associated with the tag have been propagated to a quorum[ix]. In fact, when a tag appears anywhere in the system associated with a 'fin' label, it means that the corresponding coded elements reached a quorum Q_{pw} with a 'pre' label at some previous point. The operation of a writer in the two phases following its *query phase* helps overcome the challenge of handling writer failures. In particular, notice that only tags with the 'fin' label are visible to the reader. This ensures that the reader gets at least k unique coded elements from any quorum of non-failed nodes in response to its finalize messages, because such a quorum has an intersection of at least k nodes with Q_{pw}. Finally, the reader helps propagate the tag to a quorum, and this helps complete possibly failed writes as well.

We note that the server gossip is not necessary for correctness of CAS. We use 'gossip' in CAS mainly because it simplifies the proof of atomicity of the *CASGC* algorithm, which is presented in Section V. We next state the main result of this section.

Theorem 1. *CAS emulates shared atomic read/write memory.*

To prove Theorem 1, we show atomicity, Lemma 2, and liveness, Lemma 6.

Lemma 2. *CAS(k) is atomic.*

The main idea of our proof of atomicity involves defining, on the operations of any execution β of CAS, a partial order $\prec$ that satisfies the sufficient conditions for atomicity described by Lemma 13.16 of [18]. Specifically, if β is an execution where every operation terminates, then, for any two operations

[viii] The 'null' entry indicates that no coded element is stored; the storage cost associated storing a null coded element is negligible.

[ix] It is worth noting that Q_{fw} and Q_{pw} need not be the same quorum.

write$(value)$

query: Send query messages to all servers asking for the highest tag with label 'fin'; await responses from a quorum.

pre-write: Select the largest tag from the *query* phase; let its integer component be z. Form a new tag t as $(z+1, \text{'id'})$, where 'id' is the identifier of the client performing the operation. Apply the (N, k) MDS code Φ (see Sec. III) to the value to obtain coded elements $w_1, w_2, \ldots, w_N$. Send $(t, w_s, \text{'pre'})$ to server s for every $s \in \mathcal{N}$. Await responses from a quorum.

finalize: Send a *finalize* message $(t, \text{'null'}, \text{'fin'})$ to all servers. Terminate after receiving responses from a quorum.

read

query: As in the writer protocol.

finalize: Send a *finalize* message with tag t to all the servers requesting the associated coded elements. Await responses from a quorum. If at least k servers include their locally stored coded elements in their responses, then obtain the $value$ from these coded elements by inverting Φ (see Definition 1) and terminate by returning $value$.

server

state variable: A variable that is a subset of $\mathcal{T} \times (\mathcal{W} \cup \{\text{'null'}\}) \times \{\text{'pre'}, \text{'fin'}\}$

initial state: Store $(t_0, w_{0,s}, \text{'fin'})$ where s denotes the server and $w_{0,s}$ is the coded element corresponding to server s obtained by apply Φ to the initial value v_0.

On receipt of *query* message: Respond with the highest locally known tag that has a label 'fin', i.e., the highest tag such that the triple $(tag, *, \text{'fin'})$ is at the server, where $*$ can be a coded element or 'null'.

On receipt of *pre-write* message: If there is no record of the tag of the message in the list of triples stored at the server, then add the triple in the message to the list of stored triples; otherwise ignore. Send acknowledgment.

On receipt of *finalize* from a writer: Let t be the tag of the message. If a triple of the form $(t, w_s, \text{'pre'})$ exists in the list of stored triples, then update it to $(t, w_s, \text{'fin'})$. Otherwise add $(t, \text{'null'}, \text{'fin'})$ to list of stored triples[xvi]. Send acknowledgment. Send 'gossip' message with item $(t, \text{'fin'})$ to all other servers.

On receipt of *finalize* from a reader: Let t be the tag of the message. If a triple of the form $(t, w_s, *)$ exists in the list of stored triples where $*$ can be 'pre' or 'fin', then update it to $(t, w_s, \text{'fin'})$ and send (t, w_s) to the reader. Otherwise add $(t, \text{'null'}, \text{'fin'})$ to the list of triples at the server and send an acknowledgment. Send 'gossip' message with item $(t, \text{'fin'})$ to all other servers.

On receipt of 'gossip' message: Let t be the tag of the message. If a triple of the form $(t, x, *)$ exists in the list of stored triples where $*$ is 'pre' or 'fin' and x is a coded element of 'null', then update it to $(t, x, \text{'fin'})$. Otherwise add $(t, \text{'null'}, \text{'fin'})$ to the list of triples at the server.

Fig. 1. Write, read, and server protocols of the CAS algorithm.

π_1, π_2 in β, we define a partial ordering $\prec$ satisfies the following three conditions: **(1)** If the response for π_1 precedes the invocation for π_2 in β, then it cannot be the case that $\pi_2 \prec \pi_1$. **(2)** If π_1 is a write operation π_2 is any operation, 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 an initial value v_0, if there is no such write). To define a partial order that satisfies these sufficient conditions, we use the following definition.

Definition 2. *Consider an execution β of CAS and consider an operation π that terminates in β. The* tag *of operation π, denoted as $T(\pi)$, is defined as follows: If π is a read, then, $T(\pi)$ is the highest tag received in its* query *phase. If π is a write, then, $T(\pi)$ is the new tag formed in its* pre-write *phase.*

We define our partial order $\prec$ as follows: In any execution β of CAS, we order operations π_1, π_2 as $\pi_1 \prec \pi_2$ if **(i)** $T(\pi_1) < T(\pi_2)$, or **(ii)** $T(\pi_1) = T(\pi_2)$, π_1 is a write and π_2 is a read. In [7], we formally state the three sufficient conditions for atomicity and prove that $\prec$ satisfies them. Here we provide a brief sketch of the arguments involved. We first show in Lem. 3 that, in any execution β of CAS, at any point after an operation π terminates, the tag $T(\pi)$ has been propagated with the 'fin' label to at least one quorum of servers. Intuitively speaking, Lem. 3 means that if an operation π terminates, the tag $T(\pi)$ is visible to any operation that is invoked after π terminates. We crystallize this intuition in Lem. 4, where we show that any operation that is invoked after an operation π terminates acquires a tag that is at least as large as $T(\pi)$. Using Lem. 4 we show Lem. 5, which states that the tag acquired by each write operation is unique. Then we show that Lem. 4 and Lem. 5 imply conditions **(1)** and **(2)**. By examination of the algorithm, we show that CAS satisfies **(3)** also. We state Lemmas 3, 4, 5, and 2 here, and prove them in [7].

Lemma 3. *In any execution β of CAS, for an operation π that terminates in β, there exists a quorum $Q_{fw}(\pi)$ such that the following is true at every point of the execution β after π terminates: Every server of $Q_{fw}(\pi)$ has $(t, *, \text{'fin'})$ in its set of stored triples, where $*$ is either a coded element or 'null', and $t = T(\pi)$.*

Lemma 4. *Consider any execution β of CAS, and let π_1, π_2 be two operations that terminate in β. Suppose that π_1 returns before π_2 is invoked. Then $T(\pi_2) \geq T(\pi_1)$. Furthermore, if π_2 is a write, then $T(\pi_2) > T(\pi_1)$.*

Lemma 5. *Let π_1, π_2 be write operations that terminate in an execution β of CAS. Then $T(\pi_1) \neq T(\pi_2)$.*

We now state the liveness condition satisfied by CAS.

Lemma 6 (Liveness)**.** *CAS(k) satisfies the following* liveness *condition: If $1 \leq k \leq N - 2f$, then every non-failed operation terminates in every fair execution of CAS(k) where the number of failed server nodes is no bigger than f .*

Proof (Sketch.): By examination of the algorithm we observe that termination of any operation depends on termination of its phases. So, to show liveness, we need to show that each phase of each operation terminates. Termination of a write operation and the *query* phase of a read are contingent on receiving responses from a quorum of non-failed servers in the execution; property **(ii)** of Lemma 1 guarantees the existence of such a quorum, and thus ensures their termination (see [7] for formal arguments).

We show the termination of a reader's *finalize* phase here since it is more challenging. By using property **(ii)** of Lem. 1, we can show that a quorum, say Q_{fw} of servers responds to a reader's *finalize* message. We show that at least k servers include coded elements in their responses implying that the finalize phase terminates. Suppose that the read acquired a tag t in its *query* phase. From examination of CAS, we infer that, at some point before the point of termination of the read's *query* phase, a writer propagated a *finalize* message with tag

t. Denote by $Q_{pw}(t)$, the set of servers that responded to this write's *pre-write* phase. We argue that all servers in $Q_{pw}(t) \cap Q_{fw}$ respond to the reader's *finalize* message with a coded element. To see this, let s be any server in $Q_{pw}(t) \cap Q_{fw}$. Since s is in $Q_{pw}(t)$, the server protocol for responding to a *pre-write* message implies that s has a coded element, w_s, at the point where it responds to that message. Since s is in Q_{fw}, it also responds to the reader's *finalize* message, and this happens at some point after it responds to the *pre-write* message. So it responds with coded element w_s. From Lem. 1, we know $|Q_{pw}(t) \cap Q_{fw}| \geq k$ implying that the read receives at least k coded elements in its *finalize* phase and hence terminates. □

Cost analysis. We analyze the communication costs of CAS in Theorem 2. The theorem implies that the read and write communication costs can be made as small as $\frac{N}{N-2f} \log_2 |\mathcal{V}|$ bits by choosing $k = N - 2f$.

Theorem 2. *The write and read communication costs of the CAS(k) are equal to $N/k \log_2 |\mathcal{V}|$ bits.*

V. Storage-Optimized Variant of CAS

Although CAS is efficient in terms of communication costs, it incurs an infinite storage cost because a server can store coded elements corresponding to an arbitrarily large number of versions. We here present a variant of the CAS algorithm called *CAS with Garbage Collection* (CASGC), which has the same communication costs as CAS and incurs a bounded storage cost under reasonable conditions. CASGC achieves a bounded storage cost by using *garbage collection*, i.e., by discarding coded elements with sufficiently small tags at the servers. CASGC is parametrized by two positive integers denoted as k and δ, where $1 \leq k \leq N - 2f$; we denote the algorithm with parameter values k, δ by CASGC(k, δ). Like CAS(k), we use an (N, k) MDS code in CASGC(k, δ). The parameter δ is related to the number of coded elements stored at each server.

Algorithm description. The CASGC(k, δ) algorithm is essentially the same as CAS(k) with an additional garbage collection step at the servers. In particular, the only differences between the two algorithms lie in the server actions on receiving a *finalize* message from a writer or a reader or 'gossip'. The server actions in the CASGC algorithm are described in Fig. 2. In CASGC(k, δ), each server stores the latest $\delta + 1$ triples with the 'fin' label plus the triples corresponding to later and intervening operations with the 'pre' label. For the tags that are older (smaller) than the latest $\delta + 1$ finalized tags received by the server, it stores only the metadata, not the data itself. On receiving a *finalize* message either from a writer or a reader, the server performs a garbage collection step before responding to the client. The garbage collection step checks whether the server has more than $\delta + 1$ triples with the 'fin' label. If so, it replaces the triple $(t', x, *)$ by $(t', \text{'null'}, (*, \text{'gc'}))$ for every tag t' that is smaller than all the $\delta+1$ highest tags labeled 'fin', where $*$ is 'pre' or 'fin', and x can be a coded element or 'null'. If a reader requests, through a *finalize* message, a coded element that is already garbage collected, the server simply ignores this request.

Statements and proofs of correctness. We begin with a formal statement of atomicity of CASGC. Later, we describe the liveness properties of CASGC.

Theorem 3 (Atomicity)**.** *CASGC is atomic.*

To show the above theorem, we observe that, from the perspective of the clients, the only difference between CAS and CASGC is in the server response to a read's *finalize* message. In CASGC, when a coded element has been garbage collected, a server ignores a read's *finalize* message. Atomicity follows similarly to CAS, since, in any execution of CASGC, operations acquire essentially the same tags as they would in an execution of CAS. We show this formally in [7] using a simulation relation between CASGC and CAS.

Showing operation termination in CASGC is more complicated than CAS. This is because, in CASGC, when a reader requests a coded element, the server may have garbage collected it. The liveness property we show essentially articulates conditions under which read operations terminate in spite of the garbage collection. Informally speaking, we show that CASGC satisfies the following liveness property: every operation terminates in an execution where the number of failed servers is no bigger than f and the number of writes *concurrent* with a read is bounded by $\delta+1$. Before we proceed to formally state our liveness conditions, we begin with some definitions. For any operation π that completes its query phase, the tag of the operation $T(\pi)$ is defined as in Definition 2. We next define the *end-point* of an operation.

Definition 3 (End-point of a write operation)**.** *In an execution β of CASGC, the end point of a write operation π in β is defined to be*

(a) the first point of β at which a quorum of servers that do not fail in β has tag $T(\pi)$ with the 'fin' label, where $T(\pi)$ is the tag of the operation π, if such a point exists,
(b) the point of failure of operation π, if operation π fails and (a) is not satisfied.

Note that if neither condition (a) nor (b) is satisfied, then the write operation has no end-point.

Definition 4 (End-point of a read operation)**.** *The end point of a read operation in β is defined to be the point of termination if the read returns in β. The end-point of a failed read operation is defined to be the point of failure.*

A read that does not fail or terminate has no end-point.

Definition 5 (Concurrent Operations)**.** *One operation is defined to be concurrent with another operation if it is not the case that the end point of either of the two operations is before the point of invocation of the other operation.*

Note that if both operations do not have end points, then they are concurrent with each other. We next describe the liveness property satisfied by CASGC.

Theorem 4 (Liveness)**.** *Let $1 \leq k \leq N - 2f$. Consider a fair execution β of CASGC(k, δ) where the number of write operations concurrent to any read operation is at most δ, and the number of server node failures is at most f. Then, every non-failing operation terminates in β.*

The main challenge in proving Theorem 4 lies in showing termination of read operations. In Lemma 7, we show that if a read operation does not terminate in an execution of CASGC(k, δ), then the number of write operations that are concurrent with the read is larger than δ. We provide a statement and proof sketch for Lemma 7 here. We use the lemma to show Theorem 4 in [7].

servers

state variable: A variable that is a subset of $\mathcal{T} \times (\mathcal{W} \cup \{\text{'null'}\}) \times \{\text{'pre'}, \text{'fin'}, (\text{'pre'}, \text{'gc'}), (\text{'fin'}, \text{'gc'})\}$
initial state: Same as in Fig. 1.
On receipt of *query* message: Similar to Fig. 1, respond with the highest locally available tag labeled 'fin', i.e., respond with the highest tag such that the triple $(tag, x, \text{'fin'})$ or $(tag, \text{'null'}, (\text{'fin'}, \text{'gc'}))$ is at the server, where x can be a coded element or 'null'.

On receipt of a *pre-write* message: Perform the actions as described in Fig. 1 except the sending of an acknowledgement. Perform garbage collection. Then send an acknowledgement.

On receipt of a *finalize* from a writer: Let t be the tag of the message. If a triple of the form $(t, x, \text{'fin'})$ or $(t, \text{'null'}, (\text{'fin'}, \text{'gc'}))$ is stored in the set of locally stored triples where x can be a coded element or 'null', then ignore the incoming message. Otherwise, if a triple of the form $(t, w_s, \text{'pre'})$ or $(t, \text{'null'}, (\text{'pre'}, \text{'gc'}))$ is stored, then upgrade it to $(t, w_s, \text{'fin'})$ or $(t, \text{'null'}, (\text{'fin'}, \text{'gc'}))$. Otherwise, add a triple of the form $(t, \text{'null'}, \text{'fin'})$ to the set of locally stored triples. Perform garbage collection. Send 'gossip' message with item $(t, \text{'fin'})$ to all other servers.

On receipt of a *finalize* message from a reader: Let t be the tag of the message. If a triple of the form $(t, w_s, *)$ exists in the list of stored triples where $*$ can be 'pre' or 'fin', then update it to $(t, w_s, \text{'fin'})$, perform garbage collection, and send (t, w_s) to the reader. If $(t, \text{'null'}, (*, \text{'gc'}))$ exists in the list of locally available triples where $*$ can be either 'fin' or 'pre', then update it to $(t, \text{'null'}, (\text{'fin'}, \text{'gc'}))$ and perform garbage collection, but do *not* send a response. Otherwise add $(t, \text{'null'}, \text{'fin'})$ to the list of triples at the server, perform garbage collection, and send an acknowledgment. Send 'gossip' message with item $(t, \text{'fin'})$ to all other servers.

On receipt of a 'gossip' message: Let t denote the tag of the message. If a triple of the form $(t, x, \text{'fin'})$ or $(t, \text{'null'}, (\text{'fin'}, \text{'gc'}))$ is stored in the set of locally stored triples where x can be a coded element or 'null', then ignore the incoming message. Otherwise, if a triple of the form $(t, w_s, \text{'pre'})$ or $(t, \text{'null'}, (\text{'pre'}, \text{'gc'}))$ is stored, then upgrade it to $(t, w_s, \text{'fin'})$ or $(t, \text{'null'}, (\text{'fin'}, \text{'gc'}))$. Otherwise, add a triple of the form $(t, \text{'null'}, \text{'fin'})$ to the set of locally stored triples. Perform garbage collection.

garbage collection: If the total number of tags of the set $\{t : (t, x, *)$ is stored at the server, where $x \in \mathcal{W} \cup \{\text{'null'}\}$ and $* \in \{\text{'fin'}, (\text{'fin'}, \text{'gc'})\}\}$ is no bigger than $\delta + 1$, then return. Otherwise, let $t_1, t_2, \ldots t_{\delta+1}$ denote the highest $\delta + 1$ tags from the set, sorted in descending order. Replace every element of the form $(t', x, *)$ where t' is smaller than $t_{\delta+1}$ by $(t', \text{'null'}, (*, \text{'gc'}))$ where $*$ can be either 'pre' or 'fin' and $x \in \mathcal{W} \cup \{\text{'null'}\}$.

Fig. 2. Server Actions for CASGC(k, δ).

Lemma 7. *Let $1 \leq k \leq N-2f$. Consider any fair execution β of CASGC(k, δ) where the number of server failures is upper bounded by f. Let π be a non-failing read operation in β that does not terminate. Then, the number of writes that are concurrent with π is at least $\delta + 1$.*

To prove Lemma 7, we prove Lemmas 8 and 9. Lemma 8 implies that in a fair execution where the number of server failures is bounded by f, if a non-failing server receives a finalize message corresponding to a tag at some point, then the write operation corresponding to that tag has an end-point in the execution. We note that the server gossip plays a crucial role in showing Lemma 8. We then show Lemma 9 which states that in an execution, if a write operation π has an end-point, then every operation that begins after the end-point of π acquires a tag that is at least as large as the tag of π. Using Lemmas 8 and 9, we then show Lemma 7. Here, we state Lemmas 8 and 9 and provide a sketch of the proof of Lemma 7. We provide proofs of Lemmas 7, 8 and 9 in [7].

Lemma 8. *Let $1 \leq k \leq N - 2f$. Consider any fair execution β of CASGC(k, δ) where the number of server failures is no bigger than f. Consider a write operation π that acquires tag t. If at some point of β, at least one non-failing server has a triple of the form $(t, x, \text{'fin'})$ or $(t, \text{'null'}, (\text{'fin'}, \text{'gc'}))$ where $x \in \mathcal{W} \cup \{\text{'null'}\}$, then operation π has an end-point in β.*

Lemma 9. *Consider any execution β of CASGC(k, δ). If write operation π with tag t has an end-point in β, then the tag of any operation that begins after the end point of π is at least as large as t.*

Proof of Lemma 7: The query phase of the read finishes if the reader receives a quorum of responses. Since every non-failed server responds to the reads query message, we infer from Lemma 1 that the query phase terminates. It remains to consider termination of the read's finalize phase. We argue that, if the finalize phase of a read operation π does not terminate, then there are at least $\delta + 1$ writes that are concurrent with π.

Let t be the tag acquired by operation π. By property (**ii**) of Lemma 1, we infer that a quorum, say Q_{fw} of non-failing servers receives the read's *finalize* message. There are only two possibilities. (**i**) There is no server s in Q_{fw} such that, at the point of receipt of the read's finalize message at server s, a triple of the form $(t, \text{'null'}, (*, \text{'gc'}))$ exists at the server. (**ii**) There is at least one server s in Q_{fw} such that, at the point of receipt of the read's finalize message at server s, a triple of the form $(t, \text{'null'}, (*, \text{'gc'}))$ exists at the server.

In case (**i**), we argue in [7] in a manner that is similar to Lemma 6 that the read receives responses to its finalize message from quorum Q_{fw} of which at least k responses include coded elements. Therefore the finalize phase of π terminates, contradicting our assumption that it does not. Therefore (**i**) is impossible. We next argue that in case (**ii**), there are at least $\delta+1$ write operations that are concurrent with the read operation π. In case (**ii**), from the server protocol of CASGC, we infer that at the point of receipt of the reader's finalize message at server s, there exist tags $t_1, t_2, \ldots, t_{\delta+1}$, each bigger than t, such that a triple of the form $(t_i, x, \text{'fin'})$ or $(t_i, \text{'null'}, (\text{'fin'}, \text{'gc'}))$ exists at the server. We infer from the write and server protocols that, for every i in $\{1, 2, \ldots, \delta+1\}$, a write operation, say π_i, must have committed to tag t_i in its *pre-write* phase before this point in β. Because s is non-failing in β, we infer from Lemma 8 that operation π_i has an end-point in β for every $i \in \{1, 2, \ldots, \delta + 1\}$. Since $t < t_i$ for every $i \in \{1, 2, \ldots, \delta + 1\}$, we infer from Lemma 9 that the end point of write operation π_i is after the point of invocation of operation π. Therefore operations $\pi_1, \pi_2, \ldots, \pi_{\delta+1}$ are concurrent with π. ■

Bound on storage cost. We bound the storage cost of an execution of CASGC by providing a bound on the number of coded elements stored at a server at any particular point of the execution. In particular, in Lemma 10, we describe conditions under which coded elements corresponding to the value of a write operation are garbage collected at *all* the servers. Lemma 10 naturally leads to a storage cost bound in Theorem 5. We begin with a definition of an *ω-superseded write operation* for a point in an execution, for a positive integer ω.

Definition 6 (ω-superseded write operation)**.** *In an execution β of CASGC, consider a write operation π that completes*

its query phase. Let $T(\pi)$ denote the tag of the write. Then, the write operation is said to be ω-superseded at a point P of the execution if there are at least ω terminating write operations, each with a tag that is bigger than $T(\pi)$, such that every message on behalf of each of these operations (including 'gossip' messages) has been delivered by point P.

We next show in Lemma 10 that in an execution of CASGC(k, δ), if a write operation is $(\delta + 1)$-superseded at a point, then, no server stores a coded element corresponding to the operation at that point because of garbage collection. We then use Lemma 10 to describe a bound on the storage cost of any execution of CASGC(k, δ) in Theorem 5.

Lemma 10. *Consider an execution β of CASGC(k, δ) and consider any point P of β. If a write operation π is $(\delta + 1)$-superseded at point P, then no non-failed server has a coded element corresponding to the value of operation π at point P.*

Proof: Consider an execution β of CASGC(k, δ) and a point P in β. Consider a write operation π that is $(\delta + 1)$-superseded at point P. Consider an arbitrary server s that has not failed at point P. We show that server s does not have a coded element corresponding to operation π at point P. Since operation π is $(\delta + 1)$-superseded at point P, there exist at least $\delta + 1$ write operations $\pi_1, \pi_2, \ldots, \pi_{\delta+1}$ such that, for every $i \in \{1, 2, \ldots, \delta + 1\}$,

- operation π_i terminates in β,
- the tag $T(\pi_i)$ acquired by π_i is larger than $T(\pi)$, and
- every message on behalf of π_i is delivered by point P.

Since operation π_i terminates, it completes its *finalize* phase where it sends a finalize message with tag $T(\pi_i)$ to server s. Furthermore, the *finalize* message with tag $T(\pi_i)$ arrives at server s by point P. Therefore, by P, server s has received at least $\delta + 1$ finalize messages, one from each operation in $\{\pi_i : i = 1, 2, \ldots, \delta + 1\}$. The garbage collection executed by the server on the receipt of the last of these finalize messages ensures that the coded element corresponding to tag $T(\pi)$ does not exist at server s at point P. This completes the proof. ■

Theorem 5. *Consider an execution β of CASGC(k, δ) such that, at any point of the execution, the number of writes that have completed their query phase by that point and are not $(\delta + 1)$-superseded at that point is upper bounded by w. The storage cost of the execution is at most $\frac{wN}{k} \log_2 |\mathcal{V}|$.*

We refer the reader to [7] for a proof of Theorem 5. We note that the theorem can be used to obtain a bound on the storage cost of executions in terms of various parameters of the system components. For instance, the theorem can be used to obtain a bound on the storage cost in terms of an upper bound on the delay of every message, the number of steps for the nodes to take actions, the rate of write operations, and the rate of failure. In particular, the above parameters can be used to bound the writes that are not $(\delta+1)$-superseded, which can then be used to bound the storage cost.

VI. Conclusions

We have proposed low-cost algorithms for atomic shared memory emulation in asynchronous message-passing systems. We also contribute to this body of work through rigorous definitions and analysis of (worst-case) communication and storage costs. Among the open questions in this topic, we emphasize the need for lower bounds on storage costs, generalizations of CASGC to lossy channels, and to dynamic settings possibly through modifications of RAMBO [13].

References

[1] M. Abd-El-Malek, G. R. Ganger, G. R. Goodson, M. K. Reiter, and J. J. Wylie. Fault-scalable byzantine fault-tolerant services. In *ACM SIGOPS Operating Systems Review*, volume 39, pages 59–74, 2005.

[2] A. Agrawal and P. Jalote. Coding-based replication schemes for distributed systems. *Parallel and Distributed Systems, IEEE Transactions on*, 6(3):240 –251, March 1995.

[3] M. K. Aguilera, R. Janakiraman, and L. 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] M. K. Aguilera, I. Keidar, D. Malkhi, and A. Shraer. Dynamic atomic storage without consensus. *J. ACM*, 58:7:1–7:32, April 2011.

[5] H. Attiya, A. Bar-Noy, and D. Dolev. Sharing memory robustly in message-passing systems. In *Proceedings of the ninth annual ACM symposium on Principles of distributed computing*, PODC '90, pages 363–375, New York, NY, USA, 1990. ACM.

[6] C. Cachin and S. Tessaro. Optimal resilience for erasure-coded byzantine distributed storage. In *2006 International Conference on Dependable Systems and Networks (DSN),*, pages 115–124. IEEE, 2006.

[7] V. R. Cadambe, N. Lynch, M. Medard, and P. Musial. A coded shared atomic memory algorithm for message passing architectures. *arxiv preprint*, July 2014. available at http://arxiv.org.

[8] D. Dobre, G. Karame, W. Li, M. Majuntke, N. Suri, and M. Vukolić. PoWerStore: proofs of writing for efficient and robust storage. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 285–298. ACM, 2013.

[9] P. Dutta, R. Guerraoui, and R. R. Levy. Optimistic erasure-coded distributed storage. In *Distributed Computing*, pages 182–196. Springer, 2008.

[10] R. Fan and N. Lynch. Efficient replication of large data objects. In *In Proceedings of the 17th International Symposium on Distributed Computing (DISC)*, pages 75–91, 2003.

[11] S. Frolund, A. Merchant, Y. Saito, S. Spence, and A. Veitch. A decentralized algorithm for erasure-coded virtual disks. In *2004 Intl. Conf. on Dependable Systems and Networks*, pages 125–134, June 2004.

[12] D. K. Gifford. Weighted voting for replicated data. In *Proceedings of the seventh ACM symposium on Operating systems principles*, SOSP '79, pages 150–162, New York, NY, USA, 1979. ACM.

[13] S. Gilbert, N. Lynch, and A. Shvartsman. RAMBO: A robust, reconfigurable atomic memory service for dynamic networks. *Distributed Computing*, 23(4):225–272, December 2010.

[14] J. Hendricks, G. R. Ganger, and M. K. Reiter. Low-overhead byzantine fault-tolerant storage. *SOSP*, pages 73–86, 2007.

[15] M. P. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12:463–492, July 1990.

[16] L. Lamport. On interprocess communication. Part I: Basic formalism. *Distributed Computing*, 2(1):77–85, 1986.

[17] S. Lin and D. J. Costello. *Error Control Coding, Second Edition*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2004.

[18] N. A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1996.

[19] D. Malkhi and M. Reiter. Byzantine quorum systems. *Distributed Computing*, 11(4):203–213, October 1998.

[20] J. S. Plank. T1: erasure codes for storage applications. In *Proc. of the 4th USENIX Conference on File and Storage Technologies.*, pages 1–74, 2005.

[21] R. Roth. *Introduction to coding theory*. Cambridge University Press, 2006.

[22] Y. Saito, S. Frølund, A. Veitch, A. Merchant, and S. Spence. Fab: building distributed enterprise disk arrays from commodity components. *ACM SIGOPS Operating Systems Review*, 38(5):48–58, 2004.

[23] R. Thomas. A majority consensus approach to concurrency control for multiple copy databases. *ACM Transactions on Database Systems*, 4(2):180–209, 1979.