A coded shared atomic memory algorithm for message passing architectures

Citation: Cadambe, Viveck R., Nancy Lynch, Muriel Mèdard, and Peter Musial. "A Coded Shared Atomic Memory Algorithm for Message Passing Architectures." Distributed Computing 30, no. 1 (June 13, 2016): 49–73. doi:10.1007/s00446-016-0275-x.

As Published: http://dx.doi.org/10.1007/s00446-016-0275-x

Publisher: Springer Berlin Heidelberg

Persistent URL: http://hdl.handle.net/1721.1/107661

Version: Author's final manuscript: final author's manuscript post peer review, without publisher's formatting or copy editing

A Coded Shared Atomic Memory Algorithm for Message Passing Architectures

Viveck R. Cadambe · Nancy Lynch · Muriel Mèdard · Peter Musial

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 three 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 modification of the CAS algorithm 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 the CASGC algorithm satisfies atomicity. In every execution of CASGC where the number of server failures is no bigger than f, we show that every write operation invoked at a non-failing client terminates. We also show that in an execution of CASGC with parameter δ where the number of server failures is no bigger than f, a read operation terminates provided that the number of write operations that are concurrent with the read is no bigger than δ. We explicitly characterize the storage cost of CASGC, and show that it has the same communication cost as CAS.

(3) We describe an algorithm known as the Communication Cost Optimal Atomic Storage (CCOAS) algorithm that achieves a smaller communication cost than CAS and CASGC. In particular, CCOAS incurs read and write communication costs of $\frac{N}{N-f}$ measured in terms of number of object values. We also discuss drawbacks of CCOAS as compared with CAS and CASGC.

Keywords Shared Memory Emulation · Erasure Coding · Multi-Writer Multi-Reader Atomic Register · Concurrent Read and Write Operations · Storage Efficiency

This work was supported in part by AFOSR contract numbers FA9550-13-1-0023, FA9550-14-1-0043, NSF award numbers CCF-1217506, CCF-0939370, CCF-1553248, and by BAE Systems National Security Solutions, Inc., award 739532-SLIN 0004.

Viveck R. Cadambe
Department of Electrical Engineering,
Pennsylvania State University
viveck@engr.psu.edu
· Nancy Lynch
Computer Science and Artificial Intelligence Laboratory (CSAIL)
Department of Electrical Engineering and Computer Science
Massachusetts Institute of Technology (MIT)
Cambridge, MA, USA
lynch@theory.lcs.mit.edu
· Muriel Médard
Research Laboratory of Electronics (RLE)
Department of Electrical Engineering and Computer Science
Massachusetts Institute of Technology (MIT)
Cambridge, MA, USA
medard@mit.edu
· Peter Musial
Advanced Storage Division, EMC^2
Cambridge, MA, USA
peter.musial@emc.com

The results of this work have partially appeared in a conference paper [11]. The paper [11] does not contain proofs of atomicity, liveness, and the costs incurred by the published algorithms. [11] does not include the CCOAS algorithm of Section 6 as well.

1 Introduction

Since the late 1970s, emulation of shared-memory systems in distributed message-passing environments has been an active area of research [2–8,10,14–21,28,33,34]. The traditional approach to building redundancy for distributed systems in the context of shared memory emulation is *replication*. In their seminal paper [8], Attiya, Bar-Noy, and Dolev presented a replication based algorithm for emulating shared memory that achieves atomic consistency [22, 23]. In this paper we consider a simple multi-writer generalization of their algorithm which we call the *ABD* algorithm[ii]. This algorithm uses a quorum-based replication scheme [35], combined with read and write protocols to ensure that the emulated object is atomic [23] (linearizable [22]), and to ensure liveness, specifically, that each operation terminates provided that at most $\lceil \frac{N-1}{2} \rceil$ server nodes fail. A critical step in ensuring atomicity in ABD is the *propagate* phase of the read protocol, where the readers write back the value they read to a subset of the server nodes. Since the read and write protocols require multiple communication phases where entire replicas are sent, this algorithm has a high communication cost. In [16], Fan and Lynch introduced a directory-based replication algorithm known as the LDR algorithm that, like [8], emulates atomic shared memory in the message-passing model; however, unlike [8], its read protocol is required to write only some metadata information to the directory, rather than the value read. In applications where the data being replicated is much larger than the metadata, LDR is less costly than ABD in terms of communication costs.

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 [12, 13, 24, 32]. 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 3). 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 [2–4, 6, 7, 10, 14, 15, 20, 21, 33]. In this paper, we develop algorithms that improve on previous algorithms in terms of communication and storage costs. We summarize our main contributions and compare them with previous related work next.

[ii] The algorithm of Attiya, Bar-Noy and Dolev [8] allows only a single node to act as a writer. Also, it did not distinguish between client and server nodes as we do in our paper.

1.1 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 2.

The CAS algorithm: We develop the *Coded Atomic Storage* (CAS) algorithm presented in Section 4, which is an erasure coding based shared memory emulation algorithm. We present a brief introduction of the technique of erasure coding in Section 3. For a storage system with N nodes, we show in Theorem 3 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 Theorem 3 that CAS ensures atomicity regardless of the number of (client or server) failures. In Theorem 4 in Section 4, 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 using CAS to implement a shared memory object whose values come from a finite set $\mathcal{V}$ are equal to $\frac{N}{N-2f}$, measured in terms of the number of object values. We note that these communication costs of CAS are smaller than replication based schemes, which incur a communication cost of N (see Appendix B for an analysis of communication costs of ABD and LDR algorithms.). The storage cost of CAS, however, is unbounded because each server stores the value associated with the latest version of the data object it receives. Note that in comparison, in the ABD algorithm which is based on replication, the storage cost is bounded because each node stores only the latest version of the data object (see Appendix B for an explicit characterization of the storage cost incurred by ABD).

The CASGC algorithm: In Section 5, 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 Theorem 5 by establishing a simulation relation between CAS and CASGC. Because of the garbage collection at the servers, the liveness conditions for CASGC are weaker than CAS. The liveness property satisfied by CASGC is described in Theorem 6 in Section 5. In Theorem 6, we show that every write operation invoked at a non-failing client terminates provided that the number of server failures is no bigger than f. We also prove that in an execution of CASGC with parameter δ, if the number of server failures is no bigger than f, a read operation invoked at a non-failing client terminates provided that the number

of write operations concurrent with the read is no bigger than δ. The main technical challenge lies in careful design of the CASGC algorithm in order to ensure that an unbounded number of writes that fail before propagating enough coded elements do not prevent a future read from returning a value of the data object. In particular, failed writes that 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 larger value of δ results in an algorithm that requires servers to store a larger number of coded elements, and therefore results in a larger storage cost. A formal, nontrivial bound on the storage cost incurred by an execution of CASGC is described in Theorem 7.

Communication Cost Optimal Atomic Storage Algorithm: In Section 6 we describe a new algorithm called the Communication Cost Optimal Atomic Storage (CCOAS) algorithm that satisfies the same correctness conditions as CAS, but incurs smaller communication costs. However, CCOAS would not be easily generalizable to settings where channels could incur losses because, unlike CAS and CASGC, it requires that messages from clients to servers are delivered reliably even after operations associated with the message terminates. Therefore, it may not be possible to design a protocol based on CCOAS in a setting where the channel has losses. We describe CCOAS, analyse its communication costs, and discuss its drawbacks in Section 6.

1.2 Comparison with Related Work

Erasure coding has been used to develop shared memory emulation techniques for systems with crash failures in [3,4,15,33] and Byzantine failures in [2,7,10,14,20,21]. 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, [7,10,14,15,20,21] have similar correctness requirements as our paper; these references aim to emulate an atomic shared memory that supports concurrent operations in asynchronous networks. We compare our algorithms with the *ORCAS-A* and *ORCAS-B* algorithms of [15], the algorithm of [20], which we call the *GWGR* algorithm, the algorithm of [21], which we call the *HGR* algorithm, the *M-PoWerStore* algorithm of [14], the algorithm of [10], which we call the *CT* algorithm, and the *AWE* algorithm of [7]. We note that [15] assumes lossy channels and [10, 14, 21] assume Byzantine failures. Here, we interpret the algorithms of [10, 14, 15, 21] in our model that has lossless channels and crash failures.

We measure the storage cost at a point of an execution as the total number of bits stored by all the non-failed servers at the point. The storage cost of an execution is measured as the supremum of the storage costs over all points of the execution. The worst-case storage cost of a class of executions is the supremum of the storage costs over all possible executions in the class. The communication cost of an operation is the total number of bits sent on the channels on behalf of the operation. The worst-case communication cost of an algorithm over a class of executions is defined as the supremum of the communication costs, over every operation in every execution of the class. For our comparison here, we study three scenarios:

- worst-case communication and storage costs over all possible executions of the algorithm,
- worst-case communication and storage costs among a restricted class of executions, specifically, communication and storage costs for executions where the number of ongoing write operations[iii] at any point, the message delays and the rate of client failures are all bounded, and
- the storage costs at a point of an execution when there is no ongoing write operation.

The storage and communication costs and the liveness properties satisfied by the various algorithms are tabulated in Table 1.2. As noted in the table, a distinguishing feature of CASGC is that it simultaneously has small worst-case communication cost, a bounded storage cost and desirable liveness properties when we consider the class of executions where the number of ongoing write operations, message delays and the rate of client failures are bounded. Here, we make some remarks comparing the storage costs, liveness properties and communication costs of our algorithms with the algorithms of [7, 10, 14, 15, 20, 21].

Comparisons in terms of storage cost: The GWGR algorithm of [20] develops an erasure coding based algorithm which does not perform garbage collection, and

[iii] Informally, an operation π is *ongoing* at a point P in an execution β if the point P is after the invocation of the operation π, and there are steps taken on behalf of the operation π after point P in β.

[iv] In the storage costs shown in this column of the table, we assume that any failed operations have been garbage collected previously in the execution.

Algorithm	Worst-case among all executions.			Worst case among executions with bounded number of ongoing write operations, message delays, and rate of client failures.			Storage cost when there is no ongoing write operation[iv].
	Liveness	Comm. cost	Storage cost	Liveness	Comm. cost	Storage cost	
CASGC	Operations may not terminate	$\frac{N}{N-2f}$	Infinite	Operations terminate if parameter δ is sufficiently large	$\frac{N}{N-2f}$	Bounded	$(\delta+1)\frac{N}{N-2f}$
AWE	Always	$\frac{N}{N-2f}$	Infinite	Always	$\frac{N}{N-2f}$	Infinite	Proportional to the number of readers and writers
HGR	Operations may not terminate, (obstruction freedom)	Infinite	Infinite	Operations may not terminate (obstruction freedom)	Bounded	Bounded	$\frac{N}{N-2f}$
CT	Always	Infinite	N	Always	Bounded	N	$\frac{N}{N-f}$
ORCAS-B	Always	Infinite	Infinite	Always	Infinite	Bounded	$\frac{N}{N-2f}$
ORCAS-A	Always	N	N	Always	N	N	$\frac{N}{N-2f}$
ABD	Always	N	N	Always	N	N	N

Table 1 Comparison of various algorithms over (i) worst case executions, and (ii) over the worst case execution in the class of executions where the number of ongoing write operations, message delays and rate of client failures are all bounded, and (iii) at points of the execution where there are no ongoing write operations. The costs are expressed in terms of the number of object values. We only consider algorithms that perform garbage collection in the above table, and so we omit comparisons with CAS (Section 4), MPowerStore, and GWGR algorithms.

therefore incurs an infinite storage cost, like our CAS algorithm. CAS is essentially a restricted version of the *M-PoWerStore* algorithm of [14] for the crash failure model. The main difference between CAS and M-PoWerStore is that in CAS, servers perform gossip[v]. M-PoWerStore and CAS do not perform garbage collection and therefore incur infinite storage costs.

The ORCAS-A algorithm of [15] stores, during a write operation, the entire value being written in each server. Therefore ORCAS-A incurs a worst-case storage cost that is as large as the cost of a replication based algorithm such as ABD. The CT algorithm of [10] uses the message dispersal primitive of [9] and a reliable broadcast primitive using server gossip to ensure that servers store only one coded element when there is no ongoing write operation. During the write operation, the storage cost of implementing the message dispersal primitive during an operation can be as large as the storage cost of replication. The storage and garbage collection strategies of HGR [21] and ORCAS-B of [15] are similar to that of CASGC with the parameter δ set to 0. In fact, the garbage collection strategy of CASGC may be viewed as a generalization of the garbage collection strategies of HGR and ORCAS-B. It is instructive to note that the storage costs of CASGC, HGR and ORCAS-B are all bounded if the number of ongoing write operations, the message delays and the rate of client failures are bounded. The storage costs of these algorithms can be much smaller than the cost of replication based algorithms depending on the parameters that bound the number of ongoing write operations, the message delays and the rate of client failures.

For the ORCAS-A, ORCAS-B, HGR, CT algorithms and the CASGC algorithm when $\delta = 0$, every server stores one coded element at a point of the execution when there is no ongoing write operation, assuming that all the coded elements corresponding to failed writes have been garbage collected. In fact, as noted in Table 1.2, the storage cost of the CT algorithm can be slightly smaller than the storage cost of other algorithms when there is no ongoing write operation.

[v] 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.

The AWE algorithm of [7] presents a novel approach to garbage collection. In the AWE algorithm, the servers keep track of read operations in progress, and preserve the coded elements corresponding to these read operations until completion of the read operation. The worst case storage cost is analyzed in [7] to be proportional to the product of the number of read clients and the number of write clients. In the case where there are an unbounded number of read or write clients, however, the storage cost of [7] is infinite. In fact, in AWE, the coded element of a failed write or read operation may never be removed (garbage collected) from the system; therefore, a large number of failed read or write operations could result in a correspondingly large storage cost even if the rate of client failures is small. Unlike AWE, the coded elements of failed operations are garbage collected in the CASGC algorithm so long as a future read or write operations terminate. Therefore, CASGC can store a finite number of coded elements, even if the number of failed clients is infinite, so long as failed write operations are interspersed with a sufficient number of terminating operations. We anticipate that the approach of [7] is desirable when the number of read and write clients is small, since it provides strong guarantees on operation termination even in the presence of unbounded number of concurrent read/write operations. The CASGC algorithm is desirable in the presence of a large number of read/write clients, since the storage cost is bounded and operations terminate so long as the number of write operations that are concurrent with a read operation is limited.

Comparisons in terms of communication cost: In the HGR, CT, GWGR and ORCAS-B algorithms, the coded elements from ongoing write operations are not hidden from read operations. As a consequence, servers may send several coded elements per read operation to a reader. In fact, in these algorithms, the number of coded elements sent to the readers grows with the number of write operations that are concurrent with the read operation. The message dispersal algorithm of [9] involves the transmission of coded elements via server gossip, and therefore, the CT algorithm incurs a significantly higher communication cost as compared to even the HGR and CT algorithms. In contrast to HGR, CT, GWGR and ORCAS-B algorithms, in CAS and CASGC, the communication cost an operation is exactly one coded element per server. The MPowerStore and AWE algorithms incur the same communication cost as CAS and CASGC.

In the ORCAS-A algorithm, the writers send the entire value to the servers, and, in certain scenarios, the servers may send entire values to the readers. Therefore, the communication cost of ORCAS-A is much larger than the cost of CASGC, even if the number of writes that are concurrent with a read operation are bounded. In the ORCAS-B algorithm, a server, on receiving a request from a reader, registers the client[vi] 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. Therefore, the communication cost of a read operation in ORCAS-B can be infinite even in executions where the number of ongoing write operations, the message delays, and the rates of client failure are bounded.

Comparisons in terms of liveness properties: It is worth noting that HGR, CT, GWGR, ORCAS-A, ORCAS-B and AWE all satisfy the same liveness properties as ABD and CAS, which are stronger than the liveness properties of CASGC. CASGC with parameter δ can satisfy desirable liveness properties for executions where the number of write operations that are concurrent with every read operation is bounded by δ. 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.

From a technical standpoint, our liveness guarantee uses a new notion of concurrency that is carefully crafted to ensure that failed operations are not treated as concurrent with every future operation. Our contributions also include the CCOAS algorithm, complete correctness proofs of all our algorithms through the development of invariants and simulation relations, and careful characterizations of communication and storage costs, which may be of independent interest. Generalizations of CAS and CASGC algorithms to the models of [10, 14, 15, 21], which consider Byzantine failures and lossy channel models, is an interesting direction for future research.

2 System Model

2.1 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

[vi] The idea of registering a client's identity was introduced originally in [29] and plays an important role in our CCOAS algorithm as well.

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. We assume that the number of server node failures is at most f. There is no bound on the number of client failures.

2.2 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 the tags associated with successive write requests at a particular write 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[vii]. We assume that all servers are initialized with a default initial state.

2.3 Requirements

The key correctness requirement on the targeted shared memory service is *atomicity*. Briefly, an atomic shared memory object is one where the invocations and response look like the object is and where the observed global external behaviors "look like" the object is being accessed sequentially.

Informally, an atomic shared memory object is one that supports concurrent write and read operations where, in every execution it is possible to do all of the following:

1. for each completed operation π, to insert a serialization point $*_\pi$ somewhere between the invocation and response of π,
2. to select a subset Φ of incomplete operations,
3. for each operation in Φ, to select a response,
4. and for each operation π in Φ, to insert a serialization point $*_\pi$ somewhere after the invocation of π.

The operations and responses must be selected, and the serialization points must be inserted so that, if we move the invocation and response of each completed operation and each operation in Φ to its serialization point, and remove all the incomplete operations that are not in Φ, then the trace corresponds to the trace of a read-write variable type. We refer the reader to Chapter 13 in [26] for a formal definition of atomicity.

We require our algorithms to satisfy liveness properties related to termination of operations. To describe the liveness properties of our algorithms, we define the *tasks* for each component of the system [26, 27]. A fair execution is defined in the standard manner (See reference [26], p. 212). In that definition, a fair execution is one where every automaton in the composition gets infinitely many turns to perform each of its tasks.

In this case, a fair execution is one where every message on every channel is eventually delivered, and every message that non-failing server or client prepares to send is eventually sent, and every response that a non-failing client prepares to send is eventually sent to the environment. Formally, every client, server and channel is an I/O automaton, and the system is a composition of all the client, server and channels. The tasks are as follows:

(i) *Client automaton:* Each individual channel input action corresponding to a message send to a channel, and each individual invocation is a singleton task.

(ii) *Server automaton:* Each channel input action is a singleton task

(iii) *Channel automaton:* For every message in the channel, the corresponding channel output action is a singleton task.

A client or server failure is modeled as a *fail* input action that disables every non-input action at the node.

The liveness properties of our algorithms are related to termination of operations invoked at a non-failing client in a fair execution where the number of server failures is no larger than f[viii]. The precise statements of the liveness properties of our algorithms are provided in Theorems 3, 6, and 10.

Remark 1 In a fair execution, a channel gets infinitely many turns to deliver a message, even if the node that sent the message fails. As a consequence, in a fair execution, the channels eventually deliver all their messages, even if a node that sent some of the messages fails before the points of their delivery. Although the reliable channel assumption is an implicit consequence of the usual shared memory emulation model, we expose some of its drawbacks later in Section 6.

[vii] 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.

[viii] We assume that $N > 2f$, since correctness cannot be guaranteed if $N \leq 2f$ [26].

2.4 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 a 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.

2.5 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.

3 Erasure Coding - Background

Erasure coding is a generalization of replication that has been widely studied for purposes of failure-tolerance in storage systems (see [12, 13, 24, 30, 32]). 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 (for example, see [1]). We proceed to 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 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 31 (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 now review the use of an MDS code in 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 Appendix A 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 4 and 7).

4 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 Section 5, we present a variant of CAS that has efficient storage costs as well (in addition to having the same communication costs as CAS).

Handling of incomplete writes is not as simple when erasure coding is used because, unlike in replication based techniques, no single server has a complete replica of the value being written. In CAS, we solve this problem by *hiding* ongoing write operations from reads until enough information has been stored at servers. Our approach essentially mimics [14], projected to the setting of crash failures. We describe CAS in detail next.

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. We show in Apppendix C that $\mathcal{Q}$ satisfies the following property:

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.

4.1 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. Figures 1, 2 and 3 respectively contain descriptions of the read, write and server protocols of CAS. Here, we provide an overview of the algorithm.

Each server node maintains a set of (tag, $coded\text{-}element$, $label$)[ix] 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 of servers, 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[x]. 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, where server gossip plays a critical role. The CASGC algorithm is presented in Section 5.

4.2 Statements and proofs of correctness

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, Theorem 2, and liveness, Theorem 3.

4.2.1 Atomicity

Theorem 2 *CAS(k) is atomic.*

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

[x] 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 Section 3) 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.

Fig. 1 Write protocol of the CAS algorithm.

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 31) and terminate by returning $value$.

Fig. 2 Read protocol of the CAS algorithm.

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. 3 Server protocol of the CAS algorithm.

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 [26]. We state these sufficient conditions in Lemma 2 next.

Lemma 2 (Paraphrased Lemma 13.16 [26].) *Suppose that the environment is well-behaved, meaning that an operation is invoked at a client only if no other operation was performed by the client, or the client received a response to the last operation it initiated. Let β be a (finite or infinite) execution of a read/write object, where β consists of invocations and responses of read and write operations and where all operations terminate. Let Π be the set of all operations in β.*

Suppose that $\prec$ is an irreflexive partial ordering of all the operations in Π, satisfying the following properties: **(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 in Π and π_2 is any operation in Π, 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 v_0, if there is no such write).*

The following definition will be useful in defining a partial order on operations in an execution of CAS that satisfies the conditions of Lemma 2.

Definition 41 *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. We next argue that the partial ordering $\prec$ satisfies the conditions of 2. We first show in Lemma 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, Lemma 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 Lemma 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 Lemma 4 we show Lemma 5, which states that the tag acquired by each write operation is unique. Then we show that Lemma 4 and Lemma 5 imply conditions **(1)** and **(2)** of Lemma 2. By examination of the algorithm, we show that CAS also satisfies condition **(3)** of Lemma 2.

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 $t = T(\pi)$.*

Proof The proof is the same whether π is a read or a write operation. The operation π terminates after completing its *finalize* phase, during which it receives responses from a quorum, say $Q_{fw}(\pi)$, to its *finalize* message. This means that every server s in $Q_{fw}(\pi)$ responded to the *finalize* message from π at some point before the point of termination of π. From the server protocol, we can observe that every server s in $Q_{fw}(\pi)$ stores the triple $(t, *, \text{'fin'})$ at the point of responding to the *finalize* message of π, where $*$ is either a coded element or 'null'. Furthermore, the server s stores the triple at every point after the point of responding to the *finalize* message of π and hence at every point after the point of termination of π.

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 $T(\pi_2) > T(\pi_1)$.*

Proof To establish the lemma, it suffices to show that the tag acquired in the *query* phase of π_2, denoted as $\hat{T}(\pi_2)$, is at least as big as $T(\pi_1)$, that is, it suffices to show that $\hat{T}(\pi_2) \geq T(\pi_1)$. This is because, by examination of the client protocols, we can observe that if π_2 is a read, $T(\pi_2) = \hat{T}(\pi_2)$, and if π_2 is a write, $T(\pi_2) > \hat{T}(\pi_2)$.

To show that $\hat{T}(\pi_2) \geq T(\pi_1)$ we use Lemma 3. We denote the quorum of servers that respond to the *query* phase of π_2 as $\hat{Q}(\pi_2)$. We now argue that every server s in $\hat{Q}(\pi_2) \cap Q_{fw}(\pi_1)$ responds to the *query* phase of π_2 with a tag that is at least as large as $T(\pi_1)$. To see this, since s is in $Q_{fw}(\pi_1)$, Lemma 3 implies that s has a tag $T(\pi_1)$ with label 'fin' at the point of termination of π_1. Since s is in $\hat{Q}(\pi)$, it also responds to the *query* message of π_2, and this happens at some point after the termination of π_1 because π_2 is invoked after π_1 responds. From the server protocol, we infer that server s responds to the *query* message of π_2 with a tag that is no smaller than $T(\pi_1)$. Because of Lemma 1, there is at least one server s in $\hat{Q}(\pi_2) \cap Q_{fw}(\pi_1)$ implying that operation π_2 receives at least one response in its *query* phase with a tag that is no smaller than $T(\pi_1)$. Therefore $\hat{T}(\pi_2) \geq T(\pi_1)$.

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

Proof Let π_1, π_2 be two write operations that terminate in execution β. Let C_1, C_2 respectively indicate the identifiers of the client nodes at which operations π_1, π_2 are invoked. We consider two cases.
Case 1, $C_1 \neq C_2$: From the write protocol, we note that $T(\pi_i) = (z_i, C_i)$. Since $C_1 \neq C_2$, we have $T(\pi_1) \neq T(\pi_2)$.
Case 2, $C_1 = C_2$: Recall that operations at the same client follow a "handshake" discipline, where a new invocation awaits the response of a preceding invocation. This means that one of the two operations π_1, π_2 should complete before the other starts. Suppose that, without loss of generality, the write operation π_1 completes before the write operation π_2 starts. Then, Lemma 4 implies that $T(\pi_2) > T(\pi_1)$. This implies that $T(\pi_2) \neq T(\pi_1)$.

Proof of Theorem 2. Recall that we define our ordering $\prec$ 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.

We first verify that the above ordering is a partial order, that is, if $\pi_1 \prec \pi_2$, then it cannot be that $\pi_2 \prec \pi_1$. We prove this by contradiction. Suppose that $\pi_1 \prec \pi_1$ and $\pi_2 \prec \pi_1$. Then, by definition of the ordering, we have that $T(\pi_1) \leq T(\pi_2)$ and vice-versa, implying that $T(\pi_1) = T(\pi_2)$. Since $\pi_1 \prec \pi_2$ and $T(\pi_1) = T(\pi_2)$, we have that π_1 is a write and π_2 is a read. But a symmetric argument implies that π_2 is a write and π_1 is a read, which is a contradiction. Therefore $\prec$ is a partial order.

With the ordering $\prec$ defined as above, we now show that the three properties of Lemma 2 are satisfied. For property $(\mathbf{1})$, consider an execution β and two distinct operations π_1, π_2 in β such that π_1 returns before π_2 is invoked. If π_2 is a read, then Lemma 4 implies that $T(\pi_2) \geq T(\pi_1)$. By definition of the ordering, it cannot be the case that $\pi_2 \prec \pi_1$. If π_1 is a write, then Lemma 4 implies that $T(\pi_2) > T(\pi_1)$ and so, $\pi_1 \prec \pi_2$. Since $\prec$ is a partial order, it cannot be the case that $\pi_2 \prec \pi_1$.

Property $(\mathbf{2})$ follows from the definition of the $\prec$ in conjunction with Lemma 5.

Now we show property $(\mathbf{3})$: The value returned by each read operation is the value written by the last preceding write operation according to $\prec$, or v_0 if there is no such write. Note that every version of the data object written in execution β is *uniquely* associated with a write operation in β. Lemma 5 implies that every version of the data object being written can be uniquely associated with *tag.* Therefore, to show that a read π returns the last preceding write, we only need to argue that the read returns the value associated with $T(\pi)$. From the write, read, and server protocols, it is clear that a value and/or

its coded elements are always paired together with the corresponding tags at every state of every component of the system. In particular, the read returns the value from k coded elements by inverting the MDS code Φ; these k coded elements were obtained at some previous point by applying Φ to the value associated with $T(\pi)$. Therefore Definition 31 implies that the read returns the value associated with $T(\pi)$. □

4.2.2 Liveness

We now state the liveness condition satisfied by CAS.

Theorem 3 (Liveness) *CAS(k) satisfies the following* liveness *condition: If $1 \leq k \leq N - 2f$, then every non-failing*[xi] *operation terminates in every fair execution of CAS(k) where the number of server failures is no bigger than f .*

Proof 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. Let us first examine the *query* phase of a read/write operation; note that termination of the *query* phase of a client is contingent on receiving responses from a quorum. Every non-failed server responds to a *query* message with the highest locally available tag marked 'fin'. Since every server is initialized with $(t_0, v_0, \text{'fin'})$, every non-failed server has at least one tag associated with the label 'fin' and hence responds to the client's *query* message. Since the client receives responses from every non-failed server, property (**ii**) of Lemma 1 ensures that the *query* phase receives responses from at least one quorum, and hence terminates. We can similarly show that the *pre-write* phase and *finalize* phase of a writer terminate. In particular, termination of each of these phases is contingent on receiving responses from a quorum. Their termination is guaranteed from property (**ii**) of Lemma 1 in conjunction with the fact that every non-failed server responds, at some point, to a *pre-write* message and a *finalize* message from a write with an acknowledgment.

It remains to show the termination of a reader's *finalize* phase. By using property (**ii**) of Lemma 1, we can show that a quorum, say Q_{fw} of servers responds to a reader's *finalize* message. For the *finalize* phase of a read to terminate, there is an additional requirement that at least k servers include coded elements in their responses. To show that this requirement is satisfied, 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. Let us 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 its coded element w_s. From Lemma 1, it is clear that $|Q_{pw}(t) \cap Q_{fw}| \geq k$ implying that the reader receives at least k coded elements in its *finalize* phase and hence terminates.

[xi] An operation is said to have failed if the client performing the operation fails after its invocation but before its termination.

4.3 Cost Analysis

We analyze the communication costs of CAS in Theorem 4. 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 4 *The write and read communication costs of the CAS(k) are at most $N/k \log_2 |\mathcal{V}|$ bits.*

Proof For either protocol, observe that messages carry coded elements which have size $\frac{\log_2 |\mathcal{V}|}{k}$ bits. More formally, each message is an element from $\mathcal{T} \times \mathcal{W} \times \{\text{'pre'}, \text{'fin'}\}$, where, $\mathcal{W}$ is a coded element corresponding to one of the N outputs of the MDS code Φ. As described in Section 3, $\log_2 |\mathcal{W}| = \frac{\log_2 |\mathcal{V}|}{k}$. The only messages that incur communication costs are the messages sent from the client to the servers in the *pre-write* phase of a write and the messages sent from the servers to a client in the *finalize* phase of a read. It can be seen that the total communication cost of read and write operations of the CAS algorithm are at most $\frac{N}{k} \log_2 |\mathcal{V}|$ bits.

Remark 2 It can be noted that the bound of Theorem 4 is tight because a cost of N/k is incurred in certain worst-case executions of CAS(k).

5 Storage-Optimized Variant of CAS

Although CAS is efficient in terms of communication costs, it incurs an infinite storage cost because servers 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 certain 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).

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. 3.

On receipt of *query* message: Similar to Fig. 3, 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. 3 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 and send an acknowledgement. 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. 4 Server protocol for CASGC(k, δ).

The parameter δ is related to the number of coded elements stored at each server under "normal conditions", that is, at a point where there are no ongoing write operations, and every message corresponding to every write operation has been delivered. A smaller value of δ leads to a smaller storage cost, although it results in weaker guarantee on the termination of a read operation. We first provide an algorithm description. We describe the safety and liveness properties of CASGC in Section 5.2 and analyze the storage cost in Section 5.3.

5.1 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. 4. 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.

5.2 Statements and proofs of correctness

We next describe the correctness conditions satisfied by CASGC. We begin with a formal statement and proof of atomicity of CASGC in Section 5.2.1. In Section 5.2.2, we show that CASGC(k, δ) satisfies the following liveness condition: in an execution where the number of servers is at most f, every write operation invoked at a non-failing client terminates, and a read operation invoked at a non-failing client terminates provided that the number of write operations that are *concurrent* with the read is at most δ. Our notion of concurrency in Section 5.2.2 is based on a new definition of end-points, which applies for even failed operations. While server gossip is not necessary in CAS, it plays an important role in proving termination of read operations in CASGC.

5.2.1 Atomicity

Theorem 5 (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 next.

Proof (Proof) Note that, formally, CAS is an I/O automaton formed by composing the automata of all the nodes and communication channels in the system. We show atomicity in two steps. In the first step, we construct a I/O automaton CAS$'$ which differs from CAS in that some of the actions of the servers in CAS$'$ are non-deterministic. We show that CAS$'$ simulates CAS, that is, we show that from the perspective of its external behavior (i.e., its invocations, responses and failure events), the trace of an arbitrary execution α' of CAS$'$ is the trace of an execution α of CAS. Since CAS satisfies atomicity, α' has atomic behavior implying that CAS$'$ satisfies atomicity. In the second step, we will show that CASGC simulates CAS$'$. These two steps suffice to show that CASGC satisfies atomicity.

We now describe CAS$'$. The CAS$'$ automaton is identical to CAS with respect to the read and write protocols, and to the server actions on receipt of *query* and *pre-write* messages and *finalize* messages from writers. A server's response to a *finalize* message from a read operation can be different in CAS$'$ as compared to CAS. In CAS$'$, at the point of the receipt of the *finalize* message at the server, the server could respond either with the coded element, or not respond at all (even if it has the coded element). More precisely, the server action on receipt of a finalize message is as follows.

On receipt of *finalize* from a reader: Let t be the tag of the message. If a triple of the form $(t, w_s, *)$ appears in the list of stored triples where $*$ can be 'pre' or 'fin', then update it to $(t, w_s,$ 'fin'$)$; nondeterminstically either send (t, w_s) to the reader or do not send any message. If no such triple appears, add $(t,$ 'null', 'fin'$)$ to the list of triples at the server and send an acknowledgment. Send 'gossip' message with item $(t,$ 'fin'$)$ to all other servers.

We show that CAS$'$ "simulates" CAS[xii], that is, we show that for every execution α' of CAS$'$, there is an execution α of CAS with the same external trace. We describe execution α, step by step, as follows. In particular, for every step of α', we describe the corresponding step at α. The execution α that we construct has the following properties:

(i) At a particular point of α, every client and server is at the same state as the corresponding client/server at the corresponding point of α'.

(ii) At any point of α, the set of messages in a channel contains the messages in the corresponding channel at the corresponding point of α'. A channel in α may contain extra messages that are not contained in the corresponding channel at the corresponding point in α'.

We construct execution α next. Every component in execution α has the same initial state in α and α'. For every step of α', if a client or channel takes an action, or if a server takes an action in response to a query, pre-write, gossip or write's finalize message, or if a server sends a gossip message, then, at the corresponding step in α, the corresponding client, channel or server takes the same action. If, in a step of α', a server responds to a read's finalize message with a coded element or an acknowledgement in α', the server takes the same action in α. If, in a step of α', a server does not respond to a read's finalize message with a coded element even though it stores it, we assume that in α, the server responds to the read with the stored coded element as per its protocol spefication in CAS; the message containing the coded element is delayed indefinitely in α.

Thus, in α, at every step, the client actions and states, and the server states are the same as in α'. The only difference is that in α, at a particular step, a server may send some message that will be indefinitely delayed in the channels. Since at every step, every client performs the same action in α as in α', the external trace of α is the same as α'. Since α is an execution of CAS, for any execution α' of CAS$'$, we have shown that there is an execution α of CAS with the same set of external actions. Since CAS satisfies atomicity, α has atomic behavior. Therefore α' is atomic, and implying that CAS$'$ satisfies atomicity.

Now, we show that CASGC simulates CAS$'$. That is, for every execution α_{gc} of CASGC, we construct a corresponding execution α' of CAS$'$ such that α' has the same external behavior (i.e., the same invocations, responses and failure events) as that of α_{gc}. We first describe the execution α' step-by-step, that is, we consider a step of α_{gc} and describe the corresponding step of α'. We then show that the execution α' that we have constructed is consistent with the CAS$'$ automaton.

We construct α' as follows. We first set the initial states of all the components of α' to be the same as they are in α_{gc}. At every step, the states of the client nodes and the message passing system in α' are the same as the states of the corresponding components in the corresponding step of α_{gc}. A server's responses on receipt of a message is the same in α' as that of the corresponding server's response in α_{gc}. In particular, we note that a server's external responses are the same in α_{gc} and α' even on receipt of a reader's *finalize* message, that is, if a server ignores a reader's finalize message in α_{gc}, it ignores the reader's finalize message in α' as well. Similarly, if a server sends a message as a part of 'gossip' in α_{gc}, it sends a message in α' as well. The only difference

[xii] It is instructive to note that CAS$'$ does not satisfy the same liveness properties as CAS since servers may never respond to finalize messages from a reader in CAS$'$, even in a fair execution.

between α_{gc} and α' is in the change to the server's internal state at a point of receipt of a *finalize* message from a reader or a writer. At such a point, the server may perform garbage collection in α_{gc}, whereas it does not perform garbage collection in α'. Note that the initial state, the server's response, and the client states at every step of α' are the same as the corresponding step of α_{gc}. Also note that a server that fails at a step of α_{gc} fails at the corresponding step of α' (even though the server states could be different in general because of the garbage collection). Hence, at every step, the external behavior of α' and α_{gc} are the same. This implies that the external behavior of the entire execution α' is the same as the external behavior of α_{gc}.

We complete the proof by noting that execution α' consistent with the CAS$'$ automaton. In particular, since the initial states of all the components are the same in the CAS$'$ and CASGC algorithms, the initial state of α' is consistent with the CAS$'$ automaton. Also, every step of α' is consistent with CAS$'$. Therefore, CASGC simulates CAS$'$. Since CAS$'$ is atomic, α_{gc} has atomic behavior. So CASGC is atomic.

5.2.2 Liveness

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 conditions for termination of a write operation in CASGC is similar to CAS, and are stated formally in Theorem 6. We carefully describe conditions for termination of read operation here. Informally speaking, we show that in an execution of CASGC(k, δ) where $1 \leq k \leq N - 2f$, a read operation invoked at a non-failing client terminates in an execution where the number of failing servers is no bigger than f, provided that the number of writes *concurrent* with the read is no bigger by δ[xiii]. Before we proceed to formally state our liveness conditions in Theorem 6, we give a formal definition of the notion of concurrent operations in an execution of CASGC. For any operation π that completes its query phase, the tag of the operation $T(\pi)$ is defined as in Definition 41. We begin with defining the *end-point* of an operation.

Definition 51 (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.

[xiii] If the number of writes that are concurrent with a read operation is larger than δ, then the read simply may not terminate.

For a write operation that terminates, there is a point in the execution where (a) is satisfied. If a write operation fails, then either (a) or (b) is satisfied. Therefore, a write operation that either terminates or fails has an end-point. If neither condition (a) nor (b) is satisfied, then the write operation has no end-point.

Definition 52 (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.*

Note that a read operation that either terminates or fails has an end-point. A read operation invoked at non-failing client has no end-point if it does not terminate.

Definition 53 (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.*

We next describe the liveness property satisfied by CASGC.

Theorem 6 (Liveness) *Let $1 \leq k \leq N - 2f$. Consider a fair execution β of CASGC(k, δ) where the number of server failures is at most f. Then, every write operation invoked at a non-failing client terminates in β. If the number of write operations that are concurrent to a read operation is at most δ and the read operation is invoked at a non-failing client, then the read operation terminates in β.*

The main challenge in proving Theorem 6 lies in showing termination of read operations. In Lemma 6, 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 then use the lemma to show Theorem 6 later in this section. We begin by stating and proving Lemma 6.

Lemma 6 *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 read operation invoked at a non-failing client in β that does not terminate. Then, the number of writes that are concurrent with π is at least $\delta + 1$.*

To prove Lemma 6, we prove Lemmas 7 and 8. Lemma 7 implies that if a non-failing server receives a finalize message corresponding to a tag at some point, then, eventually every non-failing server receives a finalize message with that tag. We note that the server gossip plays a crucial role in showing Lemma 7. Using Lemma 7, we then show Lemma 8 which states that if the finalize message of an operation π reaches any non-failing server in a fair execution, then any operation invoked at a non-failing client that begins after the endpoint of π acquires a tag at least as large as the tag of π. Then, using Lemma 8, we show Lemma 6.

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 no bigger than f. Consider a write operation π that acquires tag t. Suppose that 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 and at the end-point, there is a quorum of non-failing servers each with an element of the form $(t, x, \text{'fin'})$ or $(t, \text{'null'}, (\text{'fin'}, \text{'gc'}))$ where $x \in \mathcal{W} \cup \{\text{'null'}\}$.*

Proof Notice that every server that receives a *finalize* message with tag t invokes the 'gossip' protocol. If a non-failing server s stores tag t with the 'fin' label at some point of β, then from the server protocol we infer that it received a *finalize* message with tag t from a client or another server at some previous point. Since server s receives the *finalize* message with tag t, every non-failing server also receives a *finalize* message with tag t at some point of the execution because of 'gossip'. Since a server that receives a *finalize* message with tag t stores the 'fin' label after receiving the message, and the server does not delete the label associated with the tag at any point, eventually, every non-failing server stores the 'fin' label with the tag t. Since the number of server failures is no bigger than f, there is a quorum of non-failing servers that stores tag t with the 'fin' label at some point of β. Therefore, operation π has an end-point in β, with the end-point being the first point of β where a quorum of non-failing servers have the tag t with the 'fin' label.

Lemma 8 *Consider any execution β of CASGC(k, δ), and consider a write operation π with tag t in β. If there is a point in β such that at least one non-failing server s stores an element of the form $(t, x, \text{'fin'})$ or $(t, \text{'null'}, (\text{'fin'}, \text{'gc'}))$ where $x \in \mathcal{W} \cup \{\text{'null'}\}$, then the operation π has an end-point in β and the tag of any operation that begins after the end point of π is at least as large as t.*

Proof By Lemma 7, we know that π has an end-point in β and at the end-point of π, there exists at least one quorum $Q(\pi)$ of non-failing servers such that each server has the tag t with the 'fin' label. Furthermore, from the server protocol, we infer that each server in quorum $Q(\pi)$ has the tag t with the 'fin' label at every point after the end point of the operation π.

Now, suppose operation π' is invoked after the end point of π. We show that the tag acquired by operation π' is at least as large as t. Denote the quorum of servers that respond to the *query* phase of π' as $Q(\pi')$. We now argue that every server s in $Q(\pi) \cap Q(\pi')$ responds to the *query* phase of π' with a tag that is at least as large as t. To see this, since s is in $Q(\pi)$, it has a tag t with label 'fin' at the end-point of π. Since s is in $Q(\pi')$, it also responds to the *query* message of π', and this happens at some point after the end-point of π because π' is invoked after the end-point of π. Therefore server s responds with a tag that is at least as large as t. This completes the proof.

Proof (Proof of Lemma 6) Note that the termination of the query phase of the read is contingent on receiving a quorum of responses. By noting that every non-failing server responds to the read's query message, we infer from Lemma 1 that the query phase terminates. It remains to consider termination of the read's finalize phase. Consider an operation π whose finalize phase does not terminate. We argue that there are at least $\delta + 1$ write operations 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.

$(\mathbf{i})$ 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.

$(\mathbf{ii})$ 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 $(\mathbf{i})$, we argue in a manner that is similar to Theorem 3 that the read receives responses to its finalize message from quorum Q_{fw} of which at least k responses include coded elements. We repeat the argument here for completeness. From examination of CASGC, 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. Let us 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 does not contain an element of the form $(t, \text{'null'}, (*, \text{'gc'}))$ implying that it has not garbage collected the coded element at the point of receipt of the reader's finalize message. Therefore, it 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 its coded element $w_{s'}$. From Lemma 1, it is clear that $|Q_{pw}(t) \cap Q_{fw}| \geq k$ implying that the reader receives at least k coded elements in its *finalize* phase and hence terminates. Therefore the finalize phase of π terminates, contradicting our assumption that it does not. Therefore $(\mathbf{i})$ is impossible.

We next argue that in case $(\mathbf{ii})$, there are at least $\delta + 1$ write operations that are concurrent with the read operation π. In case $(\mathbf{ii})$, 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, and because $t < t_i$, we infer from Lemma 8 that write operation π_i has an end point which is after the point of invocation of operation π. Therefore operations $\pi_1, \pi_2, \ldots, \pi_{\delta+1}$ are concurrent with read operation π.

A proof of Theorem 6 follows from Lemma 6 in a manner that is similar to Theorem 3. We give a formal argument here.

Proof (Proof of Theorem 6) 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. We first consider a write operation. Note that termination of the *query* phase of a write operation is contingent on receiving responses from a quorum. Every non-failed server responds to a *query* message with the highest locally available tag marked 'fin'. Since every server is initialized with $(t_0, v_0, \text{'fin'})$, every non-failed server has at least one tag associated with the label 'fin' and hence responds to the writer's *query* message. Since the writer receives responses from every non-failed server, property **(ii)** of Lemma 1 ensures that the *query* phase receives responses from at least one quorum, and hence terminates. We similarly show that the *pre-write* phase and *finalize* phase of a writer terminate. In particular, termination of each of these phases is contingent on receiving responses from a quorum. Their termination is guaranteed from property **(ii)** of Lemma 1 in conjunction with the fact that every non-failed server responds, at some point, to a *pre-write* message and a *finalize* message from a write with an acknowledgment.

It remains to consider the termination of a read operation. Suppose that a read operation π_r invoked at a non-failing client does not terminate. Then, from Lemma 6, we infer that there are at least $\delta + 1$ writes that are concurrent with the read. Therefore a read operation invoked at a non-failing client terminates if the number write operations that are concurrent with the read operation is no larger than δ.

5.3 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 9, we describe conditions under which coded elements corresponding to the value of a write operation are garbage collected at *all* the servers. Lemma 9 naturally leads to a storage cost bound in Theorem 7. We begin with a definition of an ω-*superseded write operation* for a point in an execution, for a positive integer ω.

Definition 54 (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 show in Lemma 9 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 state and prove Lemma 9 next. We then use Lemma 9 to describe a bound on the storage cost of any execution of CASGC(k, δ) in Theorem 7.

Lemma 9 *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 the write operation π at point P.*

Proof (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 operation π_i is larger than $T(\pi)$, and
- every message on behalf of operation 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 point 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 does not exist at server s at point P. This completes the proof.

Theorem 7 *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 $\frac{wN}{k} \log_2 |\mathcal{V}|$.*

Proof Consider an execution β where 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. Consider an arbitrary point P of the execution β, and consider a server s that is non-failed at point P. We infer from the write

and server protocols that, at point P, server s does not store a coded element corresponding to any write operation that has not completed its query phase by point P. We also infer from Lemma 9 that server s does not store a coded element corresponding to an operation that is $(\delta+1)$-superseded at point P. Therefore, if server s stores a coded element corresponding to a write operation at point P, we infer that the write operation has completed its query phase but is not $(\delta+1)$-superseded by point P. By assumption on the execution β, the number of coded elements at point P of β at server s is upper bounded by w. Since each coded element has a size of $\frac{1}{k}\log_2|\mathcal{V}|$ bits and we considered an arbitrary server s, the storage cost at point P, summed over all the non-failed servers, is upper bounded by $\frac{wN}{k}\log_2|\mathcal{V}|$ bits. Since we considered an arbitrary point P, the storage cost of the execution is upper bounded by $\frac{wN}{k}\log_2|\mathcal{V}|$ bits.

We note that Theorem 7 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 number of writes that are not $(\delta+1)$-superseded, which can then be used to bound the storage cost. In an execution β of CASGC(k,δ) where there are no write client failures, if there exists a point P where every write operation invoked before point P has terminated, and every message corresponding to every write operation has been delivered before P, then the number of $(\delta+1)$-superseded write operations at P is $\delta+1$. Therefore, the storage cost at point P in execution β is $\frac{(\delta+1)N}{k}\log_2|\mathcal{V}|$.

6 Communication Cost Optimal Algorithm

A natural question is whether one might be able to prove a lower bound to show that communication costs of CAS and CASGC are optimal. Here, we describe a new "counterexample algorithm" called *Communication Cost Optimal Atomic Storage* (CCOAS) algorithm, which shows that such a lower bound cannot be proved. We show in Theorem 11 that CCOAS has write and read communication costs of $\frac{N}{N-f}\log_2|\mathcal{V}|$ bits, which is smaller than the communication costs of CAS and CASGC. Because elementary coding theoretic bounds imply that these costs can be no smaller than $\frac{N}{N-f}\log_2|\mathcal{V}|$ bits, CCOAS is optimal from the perspective of communication costs. CCOAS, however, is infeasible in practice because of certain drawbacks described later in this section.

6.1 Algorithm description

CCOAS resembles CAS in its structure. Like CAS$(N-2f)$, its quorum $\mathcal{Q}$ consists of the set of all subsets of $\mathcal{N}$ that have at least $N-f$ elements. We also use terms "query", "pre-write", and "finalize" for the various phases of operations. We provide a formal description of CCOAS in Fig. 7. Here, we informally describe the differences between CAS and CCOAS.

- In CCOAS, the writer uses an $(N, N-f)$ MDS code to generate coded elements. Note the contrast with CAS(k) which uses an (N,k) code, where the parameter k is at most $N-2f$. Because we use an $(N,N-f)$ code in CCOAS, the size of each coded element is equal to $\frac{\log_2|\mathcal{V}|}{N-f}$ bits, and as a consequence, the read and write communication costs are equal to $\frac{N}{N-f}\log_2|\mathcal{V}|$ bits.
- In CCOAS, a reader requires $N-f$ responses with coded elements for termination of its finalize phase. In CAS, in general, at most $N-2f$ responses with coded elements are required.
- In CCOAS, the servers respond to finalize messages from a read with coded elements only. This is unlike CAS, where a server that does not have a coded element corresponding to the tag of a reader's finalize message at the point of reception responds simply with an acknowledgement. In CCOAS, if a server does not have a coded element corresponding to the tag t of a reader's finalize message at the point of reception, then, in addition to adding a triple of the form $(t, \text{'null'}, \text{'fin'})$ to its local storage, the server registers this read along with tag t in its logs. When the corresponding coded element with tag t arrives at a later point, the server, in addition to storing the coded element, sends it to every reader that is registered with tag t. We show in our proofs of correctness that, in CCOAS, every non-failing server responds to a finalize message from a read with a coded element at some point.

6.2 Proof of correctness and communication cost

We next describe a formal proof of the correctness of CCOAS.

6.2.1 Atomicity

Theorem 8 *CCOAS emulates shared atomic read/write memory.*

The main challenge in showing Theorem 8 lies in showing termination of read operations, specifically to show that every non-failing server sends a coded element in response to a reader's finalize message. The theorem follows from Theorems 10 and 9, which are stated next.

Theorem 9 *The CCOAS algorithm satisfies atomicity.*

Proof Atomicity can be shown via a simulation relation with CAS. We provide a brief informal sketch of the relation here. We argue that for every execution β of

write(*value*)

query: Same as in CAS($N - 2f$).

pre-write: Select the largest tag from the *query* phase; form a new tag t by incrementing integer by 1 and adding its 'id'. Apply an $(N, N - f)$ MDS code Φ to *value* and obtain coded elements $w_1, \ldots, w_N$. Send $(t, w_s, \text{'pre'})$ to every server s. Await responses from a quorum.

finalize: Same as in CAS($N - 2f$).

Fig. 5 The write protocol of the CCOAS algorithm.

read

query: Same as in CAS($N - 2f$).

finalize: Select largest tag t from the query phase. Send *finalize* message $(t, \text{'null'}, \text{'fin'})$ to all servers requesting the associated coded elements. Await responses with coded elements from a quorum. Obtain the *value* by inverting Φ, and terminate by returning $value$.

Fig. 6 The read protocol of the CCOAS algorithm.

server

state variables: State is a subset of $\mathcal{T} \times (\mathcal{W} \cup \{\text{'null'}\}) \times \{\text{'pre'}, \text{'fin'}\} \times 2^{\mathcal{C}}$.

initial state: $(t_0, w_{0,s}, \text{'fin'}, \{\})$.

Response to *query*: Send highest locally known tag that has label 'fin'.

Response to *pre-write*: If the tag t of the message is not available in the locally stored set of tuples, add the tuple $(t, w_s, \text{'pre'}, \{\})$ to the locally stored set. If $(t, \text{'null'}, \text{'fin'}, \mathcal{C}_0)$ exists in the locally stored set of tuple for some set of clients $\mathcal{C}_0$, then send (t, w_s) to every client in $\mathcal{C}_0$ and modify the locally stored tuple to $(t, w_s, \text{'fin'}, \{\})$. Send acknowledgement to the writer.

Response to *finalize* of write: Let t denote the tag of the message. If $(t, w_s, \text{'pre'}, \{\})$ exists in the locally stored set of tuple where $*$ can be 'pre' or 'fin', update to $(t, w_s, \text{'fin'}, \{\})$. If no tuple exists in the locally stored set with tag t, add $(t, \text{'null'}, \text{'fin'}, \{\})$ to the locally stored set. Send acknowledgement.

Response to *finalize* of read: Let t denote the tag of the message and $C \in \mathcal{C}$ denote the identifier of the client sending the message. If $(t, w_s, *, \mathcal{C}_0)$ exists in the locally stored set, update the tuple as $(t, w_s, \text{'fin'}, \mathcal{C}_0)$ and send (t, w_s) to reader. If $(t, \text{'null'}, \text{'fin'}, \mathcal{C}_0)$ exists at the server, update it as $(t, \text{'null'}, \text{'fin'}, \mathcal{C}_0 \cup \{C\})$. Otherwise, add $(t, \text{'null'}, \text{'fin'}, \{C\})$ to the list of locally stored tags.

Fig. 7 The server protocol of the CCOAS algorithm. We denote the (possibly infinite) set of clients by $\mathcal{C}$. The notation $2^{\mathcal{C}}$ denotes the power set of the set of clients $\mathcal{C}$.

CCOAS, there is an execution β' of CAS with the same trace. To see this, we note that the write protocol of CCOAS is essentially identical to the write protocol in CAS, with the only difference between the two algorithms being the erasure code used in the pre-write phase. Similarly, the query phase of the read protocols of both algorithms are the same. Also note that the server responses to messages from a writer and query messages from a reader are identical in both CAS and CCOAS. The main differences between CCOAS and CAS in the server actions. The first difference is that, in CCOAS, the servers do not perform 'gossip'. The second difference is that in CCOAS, if the server does not have a coded element corresponding to the tag of the reader's finalize message, then the server does not respond at this point. Instead, the server sends a coded element to the reader at the point of receipt of the pre-write message with this tag. We essentially create β' from β by delaying all messages 'gossip' messages indefinitely, and delaying reader's finalize messages so that they arrive at each server at the point of, or after the receipt of the corresponding pre-write message by the server. This delaying ensures that the server actions are identical in both β and β'.

Specifically, we create β' as follows. In β' the points of

- invocations of operations,
- sending and receipt of messages between writers and servers,
- sending and receipt of query messages between readers and servers,
- and sending of finalize messages from the readers

are identical to β. The server 'gossip' messages in β' are delayed indefinitely. A crucial difference between β and β' lies in the points of receipt of reader's finalize messages at the servers. Consider a read operation that acquired tag t in β and let P denote the point of receipt of a reader's finalize message to server s. Let P' denote the point of receipt of a pre-write message with tag t at server s in β. Now, consider the corresponding read operation that acquired tag t in β'. Now, if P precedes P' in β, then the reader's finalize message with tag t arrives at server s at P' in β', else, it arrives at point P in β'. This implies that server s responds to reader's finalize messages at the same points in β and β'. Finally, we complete our specification of β' by letting a server's response to the reader's finalize message arrive at the client at the same point in β' as in β.

Note that if an operation acquires tag t in β, the corresponding operation in β' also acquires tag t. Also note that the points of invocation, responses of operations and

the values returned by read operations are the same in both β and β'. Therefore, there exists an execution β' of CAS with the same trace as an arbitrary execution β of CCOAS. Since CAS is atomic, β' has atomic behavior, and so does β. Therefore, CCOAS satisfies atomicity.

6.2.2 Liveness

We next state the liveness condition of CCOAS.

Theorem 10 *CCOAS satisfies the liveness condition: in every fair execution where the number of failing servers is no bigger than f, every non-failing operation terminates.*

To show Theorem 10, we first state and prove Lemma 10. Informally speaking, Lemma 10 implies that every non-failing server responds to a reader's finalize message with a coded element. As a consequence, every read operation gets $N - f$ coded elements in response to its finalize messages. Therefore its finalize phase implying that the operation returns implying Theorem 10. We first state and prove Lemma 10. Then we prove Theorem 10.

Lemma 10 *Consider any fair execution α of CCOAS and a server s that does not fail in α. Then, for any read operation in α with tag t, the server s responds to the read's finalize message with the coded element corresponding to tag t at some point of α.*

Proof (Proof) Consider a server s that does not fail in α and consider the point P of α where server s receives a finalize message with tag t from a reader. Since the read operation at the reader acquired tag t, a server s must have responded to the read's query message with tag t. Since server s responded to the read's query message with tag t, the server received a 'fin' label from either a read or a write operation at some point. This implies that a write operation π_w with tag t completed its pre-write phase before the server responded to the read's query message. From the write protocol, note that this implies that the write operation π_w sent a coded element with tag t to every server in its pre-write phase. In particular, the writer sent coded element w_s to server s. Since the channels are reliable and since s does not fail in α, this means that at some point P' of α, the server s receives the coded element w_s. There are only two possible scenarios. First, P' precedes P in α, and second, P precedes P'. To complete the proof, we show that, in the first scenario the server responds to the reader's finalize message with w_s at point P, and in the second scenario[xiv], the server responds to the reader's finalize message with w_s at point P'.

[xiv] Note that in this second scenario, the server does not respond with a coded element in CAS, where the server only sends an acknowledgement. In contrast to the proof here, the liveness proof of CAS involved showing that at least k servers satisfy the condition imposed by the first scenario.

In the first scenario, note that the server has a coded element w_s at the point P. By examining the server protocol, we observe that server s responds to the reader's finalize message with a coded element w_s.

In the second second scenario, point P' comes after P in α. Because of the server protocol on receipt of the reader's finalize message, server s adds a tuple of the form $(t, \text{'null'}, \text{'fin'}, \mathcal{C}_0)$, where $C \in \mathcal{C}_0$, to the local state at point P. Also, note that, at point P', the server stores a tuple of the form $(t, \text{'null'}, \text{'fin'}, \mathcal{C}_1)$, where $C \in \mathcal{C}_1$. Finally, based on the server protocol on receipt of a pre-write message, we note that at point P', the server sends w_s to all the clients in $\mathcal{C}_1$ including client C. This completes the proof.

We next prove Theorem 10.

Proof (Proof of Theorem 10) To prove liveness, it suffices to show that in any fair execution α where at most f servers fail, every phase of every operation terminates. The proof of termination of a write operation, and the query phase of a read operation is similar to CAS and omitted here for brevity. Here, we present a proof of termination of the finalize phase of a read in any fair execution α where at most f servers fail.

To show the termination of a read, note from Lemma 10 that in execution α, every non-failed server s responds to a reader's finalize message with a coded element. Because the number of servers that fail in α is at most f, this implies that reader obtains at least $N - f$ messages with coded elements in response to its finalize message. From the read protocol, we observe that this suffices for termination of the finalize phase of a read. This completes the proof.

6.2.3 Communication cost

We next state the communication cost of CCOAS.

Theorem 11 *The write and read communication costs of CCOAS are both equal to $\frac{N}{N-f} \log |\mathcal{V}|$.*

The proof of Theorem 11 is similar to the proof of Theorem 4 and is omitted here for brevity.

6.3 Drawbacks of CCOAS

CCOAS incurs a smaller communication cost than CAS and CASGC mainly because the reader acquires $N - f$ coded elements for a read operation, whereas in CAS and CASGC, a reader acquires at most $N - 2f$ coded elements for an operation. In particular, because the reader acquires $N-f$ coded elements, a writer uses an $(N, N-f)$ MDS code in CCOAS. Since a write operation returns after getting responses from some quorum, there are executions of our algorithm where, at the point of termination of a write operation, only a quorum Q_{pw} containing $N-f$ servers have received its pre-write messages. Now,

if one of the servers in Q_{pw} fails after the termination of the write, then, since a reader that intends to acquire the value written requires $N-f$ coded elements, it is important that at least one of the pre-write messages sent by the writer to a server outside of Q_{pw} reaches the server. In other words, it is crucial for liveness of read operations that the pre-write messages sent by the write operation are delivered to every non-failing server, even if some of these messages have not been delivered at the point of termination of the write. We use this assumption implicitly in the proof of correctness of CCOAS.

In the standard message passing model, in a fair execution, every channel eventually delivers the messages that are input in the channel. In particular, under the standard definition of fairness, the channel eventually delivers all its messages even if the any of the nodes that input the messages fails before the message is delivered. The fact that operation termination in CCOAS depends critically on a reliable message delivery assumption is a significant drawback of CCOAS. The modeling assumption of reliable channels is often an implicit abstraction of a lossy channel and an underlying primitive that retransmits lost messages until they are delivered. From a practical point of view, however, it is not well-motivated to assume that this underlying primitive retransmits lost messages corresponding to operations that have terminated, especially if the client performing the operation fails. The limited practicality of CCOAS exposes a subtle drawback of the standard message passing model for the study shared memory emulation algorithms, especially when we aspire to have a smaller communication costs than CASGC. CAS and CASGC do not share the drawback of CCOAS, because in these algorithms, a write operation ensures that its coded elements reach a quorum before the point of termination. An interesting future exercise is to generalize CAS and CASGC to lossy channel models (see, for example, the model used in [15]).

7 Conclusions

We have proposed low-cost algorithms for atomic shared memory emulation in asynchronous message-passing systems. We have also contributed to this body of work through rigorous definitions and analysis of (worst-case) communication and storage costs. We have shown that our algorithms have desirable properties in terms of the amount of communication and storage costs.

There are several relevant follow up research directions in this topic. We list some of them below.

- In our CASGC algorithm, although we garbage collect the coded elements, we do not garbage collect the metadata. In particular, in an execution with an infinite number of write operations, each server may store the tag and a label for every write operation and therefore, may store infinitely large amount of metadata. The question of whether the metadata can be removed in the garbage collection step without violating atomicity and liveness of CASGC remains open.
- Our CAS and CASGC algorithms are developed in a model with reliable channels. Our discussion in Section 6 reveals the importance of understanding the properties of shared memory emulation algorithms in a model with lossy channels. Extending CAS and CASGC to a model with lossy channels is an important direction of future work.
- Recently, a coding theoretic formulation inspired by the need to ensure atomicity in storage systems has been presented in [36]. An interesting question is whether the storage cost can be reduced through using the ideas of [36], or through other sophisticated coding techniques.
- When erasure coding is used for shared memory emulation, the communication and storage costs of various algorithms in literature depend on the number of concurrent operations or the number of clients. In particular, in algorithms in literature, an infinite number of incomplete/failed operations can lead to either violations of operation termination or an infinite communication or storage cost; for instance, in CASGC, an unbounded number of failed write operations can lead to an unbounded storage cost if they are not interspersed with a sufficient number of operations that terminate. A natural question is whether there exist fundamental lower bounds that capture this behavior, or whether there exist algorithms that can achieve low communication and storage costs which do not grow with the degree of concurrency in the system.
- The AWE algorithm of [7] presents an algorithm with desirable liveness properties and storage cost even if the number of write operations that are concurrent with a read operation is large, provided that the number of clients is limited. The CASGC algorithm, in contrast, provides reasonable conditions on operation termination and storage cost even if there are an unbounded number of clients, provided that the number of write operations that are concurrent with a read operation is limited. Our work motivates that search for an algorithm that combines the desirable properties of the AWE and CASGC algorithms.
- Generalizing CAS and CASGC to dynamic settings possibly through modifications of RAMBO [19] is an unexplored research direction.

References

1. Common RAID disk data format specification. SNIA, Advanced Storage and Information Technology Standard, version 2 (2009)
2. Abd-El-Malek, M., Ganger, G.R., Goodson, G.R., Reiter, M.K., Wylie, J.J.: Fault-scalable byzantine fault-tolerant services. In: ACM SIGOPS Operating Systems Review, vol. 39, pp. 59–74 (2005)
3. Agrawal, A., Jalote, P.: Coding-based replication schemes for distributed systems. IEEE Transactions on Parallel and Distributed Systems **6**(3), 240 –251 (1995). DOI 10.1109/71.372774
4. Aguilera, M.K., Janakiraman, R., Xu, L.: Using erasure codes efficiently for storage in a distributed system. In: Proceedings of International Conference on Dependable Systems and Networks (DSN), pp. 336–345. IEEE (2005)
5. Aguilera, M.K., Keidar, I., Malkhi, D., Shraer, A.: Dynamic atomic storage without consensus. J. ACM **58**, 7:1–7:32 (2011). URL `http://doi.acm.org/10.1145/1944345.1944348`
6. Anderson, E., Li, X., Merchant, A., Shah, M.A., Smathers, K., Tucek, J., Uysal, M., Wylie, J.J.: Efficient eventual consistency in pahoehoe, an erasure-coded key-blob archive. In: IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), pp. 181–190. IEEE (2010)
7. Androulaki, E., Cachin, C., Dobre, D., Vukolić, M.: Erasure-coded byzantine storage with separate metadata. Principles of Distributed Systems, Springer pp. 76–90 (2014)
8. Attiya, H., Bar-Noy, A., Dolev, D.: Sharing memory robustly in message-passing systems. Journal of the ACM (JACM) **42**(1), 124–142 (1995)
9. Cachin, C., Tessaro, S.: Asynchronous verifiable information dispersal. Distributed Computing pp. 503–504 (2005)
10. Cachin, C., Tessaro, S.: Optimal resilience for erasure-coded byzantine distributed storage. In: 2006 International Conference on Dependable Systems and Networks (DSN), pp. 115–124. IEEE (2006)
11. Cadambe, V.R., Lynch, N., Medard, M., Musial, P.: A coded shared atomic memory algorithm for message passing architectures. In: 13th International Symposium on Network Computing and Applications (NCA), pp. 253–260. IEEE (2014)
12. Cassuto, Y.: What can coding theory do for storage systems? ACM SIGACT News **44**(1), 80–88 (2013)
13. Datta, A., Oggier, F.: An overview of codes tailor-made for better repairability in networked distributed storage systems. ACM SIGACT News **44**(1), 89–105 (2013)
14. Dobre, D., Karame, G., Li, W., Majuntke, M., Suri, N., Vukolić, M.: PoWerStore: proofs of writing for efficient and robust storage. In: Proceedings of the 2013 ACM SIGSAC conference on Computer & Communications security, pp. 285–298. ACM (2013)
15. Dutta, P., Guerraoui, R., Levy, R.R.: Optimistic erasure-coded distributed storage. In: Distributed Computing, pp. 182–196. Springer (2008)
16. Fan, R., Lynch, N.: Efficient replication of large data objects. In: Proceedings of the 17th International Symposium on Distributed Computing (DISC), pp. 75–91 (2003)
17. Fekete, A., Lynch, N., Shvartsman, A.: Specifying and using a partitionable group communication service. ACM Trans. Comput. Syst. **19**(2), 171–216 (2001). DOI http://doi.acm.org/10.1145/377769.377776
18. Gifford, D.K.: Weighted voting for replicated data. In: Proceedings of the seventh ACM symposium on Operating systems principles, SOSP '79, pp. 150–162. ACM, New York, NY, USA (1979). URL `http://doi.acm.org/10.1145/800215.806583`
19. Gilbert, S., Lynch, N., Shvartsman, A.: RAMBO: A robust, reconfigurable atomic memory service for dynamic networks. Distributed Computing **23**(4), 225–272 (2010)
20. Goodson, G.R., Wylie, J.J., Ganger, G.R., Reiter, M.K.: Efficient byzantine-tolerant erasure-coded storage. In: 2004 International Conference on Dependable Systems and Networks, pp. 135–144. IEEE (2004)
21. Hendricks, J., Ganger, G.R., Reiter, M.K.: Low-overhead Byzantine fault-tolerant storage. Proceedings of the seventh ACM symposium on Operating systems principles (SOSP) **41**(6), 73–86 (2007)
22. Herlihy, M.P., Wing, J.M.: Linearizability: a correctness condition for concurrent objects. ACM Trans. Program. Lang. Syst. **12**, 463–492 (1990). URL `http://doi.acm.org/10.1145/78969.78972`
23. Lamport, L.: On interprocess communication. Part I: Basic formalism. Distributed Computing **2**(1), 77–85 (1986)
24. Lin, S., Costello, D.J.: Error Control Coding, Second Edition. Prentice-Hall, Inc., Upper Saddle River, NJ, USA (2004)
25. Lynch, N., Shvartsman, A.: Robust emulation of shared memory using dynamic quorum-acknowledged broadcasts. In: Twenty-Seventh Annual International Symposium on Fault-Tolerant Computing, FTCS-27. Digest of Papers, pp. 272–281. IEEE (1997)
26. Lynch, N.A.: Distributed Algorithms. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (1996)
27. Lynch, N.A., Tuttle, M.R.: An introduction to input/output automata. CWI Quarterly **2**, 219–246 (1989)
28. Malkhi, D., Reiter, M.: Byzantine quorum systems. Distributed Computing **11**(4), 203–213 (1998). URL `http://dx.doi.org/10.1007/s004460050050`
29. Martin, J.P., Alvisi, L., Dahlin, M.: Minimal byzantine storage. In: Distributed Computing, pp. 311–325. Springer (2002)
30. Plank, J.S.: T1: erasure codes for storage applications. In: Proc. of the 4th USENIX Conference on File and Storage Technologies., pp. 1–74 (2005)
31. Reed, I.S., Solomon, G.: Polynomial codes over certain finite fields. Journal of the Society for Industrial & Applied Mathematics **8**(2), 300–304 (1960)
32. Roth, R.: Introduction to coding theory. Cambridge University Press (2006)
33. Saito, Y., Frølund, S., Veitch, A., Merchant, A., Spence, S.: Fab: building distributed enterprise disk arrays from commodity components. In: ACM SIGARCH Computer Architecture News, vol. 32, pp. 48–58. ACM (2004)
34. Thomas, R.: A majority consensus approach to concurrency control for multiple copy databases. ACM Transactions on Database Systems **4(2)**, 180–209 (1979)
35. Vukolić, M.: Quorum systems: With applications to storage and consensus. Synthesis Lectures on Distributed Computing Theory **3**(1), 1–146 (2012). URL `http://dx.doi.org/10.2200/S00402ED1V01Y201202DCT009`
36. Wang, Z., Cadambe, V.R.: Multi-version coding in distributed storage. In: 2014 IEEE International Symposium on Information Theory (ISIT). (2014)

A Discussion on Erasure Codes

For an (N, k) code, the ratio $\frac{N}{k}$ - also known as the *redundancy factor* of the code - represents the storage cost overhead in the classical erasure coding model. Much literature in coding theory involves the design of (N, k) codes for which the redundancy factor[xv] can be made as small as possible. In the classical erasure coding model, the extent to which the redundancy factor can be reduced depends on f - the maximum number of server failures that are to be tolerated. In particular, an (N, k) MDS code, when employed to store the value of the data object, tolerates $N - k$ server node failures; this is because the definition of an MDS code implies that the data can be recovered from any k surviving nodes. Thus, for an N-server system that uses an MDS code, we must have $k \leq N - f$, meaning that the redundancy factor is at least $\frac{N}{N-f}$. It is well known [32] that, given N and f, the parameter k cannot be made larger than $N - f$ so that the redundancy factor is lower bounded by $\frac{N}{N-f}$ for *any* code even if it is not an MDS code; In fact, an MDS code can equivalently be defined as one which attains this lower bound on the redundancy factor. In coding theory, this lower bound is known as the Singleton bound [32]. Given parameters N, k, the question of whether an (N, k) MDS code exists depends on the alphabet of code $\mathcal{W}$. We next discuss some of the relevant assumptions that we (implicitly) make in this paper to enable the use of an (N, k) MDS code in our algorithms.

A.1 Assumption on $|\mathcal{V}|$ due to Erasure Coding

Recall that, in our model, each value v of a data object belongs to a finite set $\mathcal{V}$. In our system, for the use of coding, we assume that $\mathcal{V} = \mathcal{W}^k$ for some finite set $\mathcal{W}$ and that $\Phi : \mathcal{W}^k \rightarrow \mathcal{W}^N$ is an MDS code. Here we refine these assumptions using classical results from erasure coding theory. In particular, the following result is useful.

Theorem 12 *Consider a finite set $\mathcal{W}$ such that $|\mathcal{W}| \geq N$. Then, for any integer $k < N$, there exists an (N, k) MDS code $\Phi : \mathcal{W}^k \rightarrow \mathcal{W}^N$.*

One proof for the above in coding theory literature is constructive. Specifically, it is well known that when $|\mathcal{W}| \geq N$, then Φ can be constructed using the Reed-Solomon code construction [24,31,32]. The above theorem implies that, to employ a Reed-Solomon code over our system, we shall need the following two assumptions:

- k divides $\log_2 |\mathcal{V}|$, and
- $\log_2 |\mathcal{V}|/k \geq \log_2 N$.

Thus all our results are applicable under the above assumptions.

In fact, the first assumption above can be replaced by a different assumption with only a negligible effect on the communication and storage costs. Specifically, if $\log_2 |\mathcal{V}|$ were not a multiple of k then, one could pad the value with $\left(\lceil \frac{\log_2 |\mathcal{V}|}{k} \rceil k - \log_2 |\mathcal{V}|\right)$ "dummy" bits, all set to 0, to ensure that the (padded) object has a size that is multiple of k; note that this padding is an overhead. The size of the padded object would be $\lceil \frac{\log_2 |\mathcal{V}|}{k} \rceil k$ bits and the size of each coded element would be $\lceil \frac{\log_2 |\mathcal{V}|}{k} \rceil$ bits. If we assume that $\log_2 |\mathcal{V}| \gg k$ then, $\lceil \frac{\log_2 |\mathcal{V}|}{k} \rceil \approx \frac{\log_2 |\mathcal{V}|}{k}$ meaning that the padding overhead can be neglected. Consequently, the first assumption can be replaced by the assumption that $\log_2 |\mathcal{V}| \gg k$ with only a negligible effect on the communication and storage costs.

[xv] Literature in coding theory literature often studies the *rate* $\frac{N}{k}$ of a code, which is the reciprocal of the redundancy factor, i.e., the rate of an (N, k) code is $\frac{k}{N}$. In this paper, we use the redundancy factor in our discussions since it enables a somewhat more intuitive connection with the costs of our algorithms in Theorems 13, 14, 4, 7.

B Descriptions of the ABD and LDR Algorithms

As baselines for our work we use the MWMR versions of the ABD and LDR algorithms [8, 16]. Here, we describe the ABD and LDR algorithms, and evaluate their communication and storage costs. We present the ABD algorithm in Figures 8, 9 and 10. We present the LDR algorithm in Figures 11, 12 and 13. The costs of these algorithms are stated in Theorems 13 and 14.

Theorem 13 *The write and read communication costs of ABD are respectively equal to $N \log |\mathcal{V}|$ and $2N \log |\mathcal{V}|$ bits. The storage cost is equal to $N \log_2 |\mathcal{V}|$ bits.*

The LDR algorithm divides its servers into *directory servers* that store metadata, and *replica servers* that store object values. The write protocol of LDR involves the sending of object values to $2f + 1$ replica servers. The read protocol is less taxing since in the worst-case, it involves retrieving the data object values from $f + 1$ replica servers. We state the communication costs of LDR next (for formal proof, see Appendix B.)

Theorem 14 *In LDR, the write communication cost is $(2f + 1) \log_2 |\mathcal{V}|$ bits, and the read communication cost is $(f+1) \log_2 |\mathcal{V}|$ bits.*

In the LDR algorithm, each replica server stores every version of the data object it receives[xvi]. Therefore, the (worst-case) storage cost of the LDR algorithm is unbounded.

Proof of Theorem 13. We first present arguments that upper bound the communication and storage cost for every execution of the ABD algorithm. The ABD algorithm presented here is fitted to our model. Specifically in [8, 25] there is no clear cut separation between clients and servers. However, this separation does not change the costs of the algorithm. Then we present worst-case executions that incur the costs as stated in the theorem.

Upper bounds: First consider the write protocol. It has two phases, *get* and *put*. The *get* phase of a write involves transfer of a tag, but not of actual data, and therefore has negligible communication cost. In the *put* phase of a write, the client sends a value from the set $\mathcal{T} \times \mathcal{V}$ to every server node; the total communication cost of this phase is at most $N \log_2 |\mathcal{V}|$ bits. Therefore the total write communication cost is at most $N \log_2 |\mathcal{V}|$ bits. In the *get* phase of the read protocol, the message from the client to the servers contains only metadata, and therefore has negligible communication cost. However, in this phase, each of the N servers could respond to the client with a message from $\mathcal{T} \times \mathcal{V}$; therefore the total communication cost of the messages involved in the *get* phase is upper bounded by $N \log_2 |\mathcal{V}|$ bits. In the *put* phase of the read protocol, the read sends an element of $\mathcal{T} \times \mathcal{V}$ to N servers. Therefore, this phase incurs a communication cost of at most $N \log_2 |\mathcal{V}|$ bits. The total communication cost of a read is therefore upper bounded by $2N \log_2 |\mathcal{V}|$ bits.

The storage cost of ABD is no bigger than $N \log_2 |\mathcal{V}|$ bits because each server stores at most one value - the latest value it receives.

Worst-case executions: Informally speaking, due to asynchrony and the possibility of failures, clients always send requests to all servers and in the worst case, all servers respond. Therefore the upper bounds described above are tight.

For the write protocol, the client sends the value to all N nodes in its *put* phase. So the write communication cost in an execution where at least one write terminates is $N \log_2 |\mathcal{V}|$ bits. For the read protocol, consider the following execution, where there is one read operation, and one write operation that is concurrent with this read. We will assume that none of the N servers fail in this execution. Suppose that the writer completes its get phase, and commits to a tag t. Note that t is the highest tag in the system at this point. Suppose that among the N messages that the writer sends in its put phase with the value and tag t, Now the writer begins its put

[xvi] This is unlike ABD where the servers store only the latest version of the data object received.

write$(value)$

get: Send query request to all servers, await (tag) responses from a majority of server nodes. Select the largest tag; 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.

put: Send the pair $(t, value)$ to all servers, await acknowledgment from a majority of server nodes, and then terminate.

Fig. 8 Write protocol of the ABD algorithm.

read

get: Send query request to all servers, await $(tag, value)$ responses from a majority. Select a tuple with the largest tag, say (t, v).

put: Send (t, v) to all servers, await acknowledgment from a majority, and then terminate by returning the value v.

Fig. 9 Read protocol of the ABD algorithm.

server

state variable: A variable which contains an element of $\mathcal{T} \times \mathcal{V}$

initial state: Store the default $(tag, value)$ pair (t_0, v_0).

On receipt of *get* message from a read: Respond with the locally available $(tag, value)$ pair.

On receipt of *get* message from a write: Respond with the locally available tag.

On receipt of *put* message: If the tag of the message is higher than the locally available tag, store the $(tag, value)$ pair of the message at the server. In any case, send an acknowledgment.

Fig. 10 Server protocol of the ABD algorithm.

write$(value)$

get-metadata: Send query request to directory servers, and await $(tag, location)$ responses from a majority of directory servers. Select the largest tag; let its integer component be z. Form a new tag t as $(z+1, \text{'id'})$, where 'id' represents the identifier of the client performing the operation.

put: Send $(t, value)$ to $2f+1$ replica servers, await acknowledgment from $f+1$. Record identifiers of the first $f+1$ replica servers that respond, call this set of identifiers $\mathcal{S}$.

put-metadata: Send $(t, \mathcal{S})$ to all directory servers, await acknowledgment from a majority, and then terminate.

Fig. 11 Write protocol of the LDR algorithm

read

get-metadata: Send query request to directory servers, and await $(tag, location)$ responses from a majority of directory servers. Choose a $(tag, location)$ pair with the largest tag, let this pair be $(t, \mathcal{S})$.

put-metadata: Send $(t, \mathcal{S})$ to all directory servers, await acknowledgment from a majority.

get: Send *get object* request to any $f+1$ replica servers recorded in $\mathcal{S}$ for tag t. Await a single response and terminate by returning a value.

Fig. 12 Read protocol of the LDR algorithm

replica server

state variable: A variable that is subset of $\mathcal{T} \times \mathcal{V}$

initial state: Store the default $(tag, value)$ pair (t_0, v_0).

On receipt of *put* message: Add the $(tag, value)$ pair in the message to the set of locally available pairs. Send an acknowledgment.

On receipt of *get* message: If the value associated with the requested tag is in the set of pairs stored locally, respond with the value. Otherwise ignore.

directory server

state variable: A variable that is an element of $\mathcal{T} \times 2^{\mathcal{R}}$ where $2^{\mathcal{R}}$ is the set of all subsets of $\mathcal{R}$.

initial state: Store $(t_0, \mathcal{R})$, where $\mathcal{R}$ is the set of all replica servers.

On receipt of *get-metadata* message: Send the $(tag, \mathcal{S})$ be the pair stored locally.

On receipt of *put-metadata* message: Let $(t, \mathcal{S})$ be the incoming message. At the point of reception of the message, let $(tag, \mathcal{S}_1)$ be the pair stored locally at the server. If t is equal to the tag stored locally, then store $(t, \mathcal{S} \cup \mathcal{S}_1)$ locally. If t is bigger than tag and if $|\mathcal{S}| \geq f+1$, then store $(t, \mathcal{S})$ locally. Send an acknowledgment.

Fig. 13 Replica and directory server protocols of the LDR algorithm

phase where it sends N messages with the value and tag t. At least one of these messages, say the message to server 1, arrives.the remaining messages are delayed, i.e., they are assumed to reach after the portion of the execution segment described here. At this point, the read operation begins and receives $(tag, value)$ pairs from all the N server nodes in its get phase. Of these N messages, at least one message contains the tag t and the corresponding value. Note that t is the highest tag it receives. Therefore, the put phase of the read has to sends N messages with the tag t and the corresponding value - one message to each of the N servers that which responded to the read in the get phase with an older tag.

The read protocol has two phases. The cost of a read operation in an execution is the sum of the communication costs of the messages sent in its *get* phase and those sent in its *put* phase. The *get* phase involves communication of N messages from $\mathcal{T} \times \mathcal{V}$, one message from each server to the client, and therefore incurs a communication cost of $N \log_2 |\mathcal{V}|$ bits provided that every server is active. The *put* phase involves the communication of a message in $\mathcal{T} \times \mathcal{V}$ from the client to every server thereby incurring a communication cost of $N \log_2 |\mathcal{V}|$ bits as well. Therefore, in any execution where all N servers are active, the communication cost of a read operation is $2N \log_2 |\mathcal{V}|$ bits and therefore the upper bound is tight.

The storage cost is equal to $N \log_2 |\mathcal{V}|$ bits since each of the N servers store exactly one value from $\mathcal{V}$. □

Proof of Theorem 14.

Upper bounds: In LDR servers are divided into two groups: *directory* servers used to manage object metadata, and *replication* servers used for object replication. Read and write protocols have three sequentially executed phases. The *get-metadata* and *put-metadata* phases incur negligible communication cost since only metadata is sent over the message-passing system. In the *put* phase, the writer sends its messages, each of which is an element from $\mathcal{T} \times \mathcal{V}$, to $2f + 1$ replica servers and awaits $f + 1$ responses; since the responses have negligible communication cost, this phase incurs a total communication cost of at most $(2f + 1) \log_2 |\mathcal{V}|$ bits. The read protocol is less taxing, where the reader during the *get* phase queries $f+1$ replica servers and in the worst case, all respond with a message containing an element from $\mathcal{T} \times \mathcal{V}$ thereby incurring a total communication cost of at most $(f + 1) \log_2 |\mathcal{V}|$ bits.

Worst-case executions: It is clear that in every execution where at least one writer terminates, the writer sends out $(2f+1)$ messages to replica servers that contain the value, thus incurring a write communication cost of $(2f + 1) \log_2 |\mathcal{V}|$ bits. Similarly, for a read, in certain executions, all $(f + 1)$ replica servers that are selected in the *put phase* of the read respond to the *get* request from the client. So the upper bounds derived above are tight. □

C Proof of Lemma 1

Proof of property **(i)**: By the definition, each $Q \in \mathcal{Q}$ has cardinality at least $\lceil \frac{N+k}{2} \rceil$. Therefore, for $Q_1, Q_2 \in \mathcal{Q}$, we have

$$\begin{aligned} |Q_1 \cap Q_2| &= |Q_1| + |Q_2| - |Q_1 \cup Q_2| \\ &\geq 2 \left\lceil \frac{N+k}{2} \right\rceil - |Q_1 \cup Q_2| \\ &\overset{(a)}{\geq} 2 \left\lceil \frac{N+k}{2} \right\rceil - N \geq k, \end{aligned}$$

where we have used the fact that $|Q_1 \cup Q_2| \leq N$ in (a).

Proof of property **(ii)**: Let $\mathcal{B}$ be the set of all the server nodes that fail in an execution, where $|\mathcal{B}| \leq f$. We need to show that there exists at least one quorum set $Q \in \mathcal{Q}$ such that $Q \subseteq \mathcal{N} - \mathcal{B}$, that is, at least one quorum survives. To show this, because of the definition of our quorum system, it suffices to show that $|\mathcal{N}-\mathcal{B}| \geq \lceil \frac{N+k}{2} \rceil$. We show this as follows:

$$|\mathcal{N} - \mathcal{B}| \geq N - f \overset{(b)}{\geq} N - \left\lfloor \frac{N-k}{2} \right\rfloor = \left\lceil \frac{N+k}{2} \right\rceil,$$

where, (b) follows because $k \leq N-2f$ implies that $f \leq \lfloor \frac{N-k}{2} \rfloor$.