Fast Lean Erasure-Coded Atomic Memory Object

Kishori M. Konwar
Department of EECS, MIT, Cambridge, USA
kishori@mit.edu

N. Prakash [1]
Intel Inc, OR, USA
prakashnarayanamoorthy@gmail.com

Muriel Médard
Department of EECS, MIT, Cambridge, USA
medard@mit.edu

Nancy Lynch
Department of EECS, MIT, Cambridge, USA
lynch@csail.mit.edu

Abstract

In this work, we propose FLECKS, an algorithm which implements atomic memory objects in a multi-writer multi-reader (MWMR) setting in asynchronous networks and server failures. FLECKS substantially reduces storage and communication costs over its replication-based counterparts by employing erasure-codes. FLECKS outperforms the previously proposed algorithms in terms of the metrics that to deliver good performance such as storage cost per object, communication cost a high fault-tolerance of clients and servers, guaranteed liveness of operation, and a given number of communication rounds per operation, etc. We provide proofs for liveness and atomicity properties of FLECKS and derive worst-case latency bounds for the operations. We implemented and deployed FLECKS in cloud-based clusters and demonstrate that FLECKS has substantially lower storage and bandwidth costs, and significantly lower latency of operations than the replication-based mechanisms.

2012 ACM Subject Classification Theory of computation → Distributed computing models

Keywords and phrases Atomicity, Distributed Storage System, Erasure-codes

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2019.12

1 Introduction

In the recent years, the demand for efficient and reliable large-scale distributed storage systems (DSSs) has grown at an unprecedented scale. DSSs that store massive data sets across several hundreds of servers are commonly used for both industrial and scientific applications, and numerous Internet-based applications. Many applications demand concurrent and consistent access to the stored data by multiple writers and readers. Therefore, some form of consistency must be guaranteed of the stored objects is essential for the application developer to reason about the correctness of the application. The consistency model we adopt is *atomicity*, also often referred to as *strong consistency*. Atomic consistency gives the users of the data service the impression that the various concurrent read and write operations happen sequentially. Therefore, strong consistency or linearizability is the most preferred form of consistency guarantee. However, providing strong consistency is a non-trivial task in most practical distributed storage systems due the asynchronous behavior of the communication and component failures endemic in any large network. Also, the ability to withstand failures and network delays are essential features of any robust DSS. The traditional solution for

[1] This work was done while the author was still at MIT.

emulating an atomic fault-tolerant shared storage system involves replication of data across the servers. Perhaps, the earliest of replication-based algorithms atomic memory emulation in asynchronous networks appear in the work by Attiya, Bar-Noy and Dolev [4] (we refer to this as the ABD algorithm).Replication based strategies incur high storage costs; for example, to store a *value* (an abstraction of a data file) of size 1 MB across a 5-server system, the ABD algorithm replicates the value in all the 5 servers, which blows up the worst-case *storage cost* to 5 MB. Additionally, every write or read operation has a worst-case *communication cost* of 5 MB. The communication cost, or simply the cost, associated with a read or write operation is the amount of total data in bytes that gets transmitted in the various messages sent as part of the operation. Since the focus in this paper is on large data objects, the storage and communication costs include only the total sizes of stable storage and messages dedicated to the data itself. Ephemeral storage and the cost of control communication is assumed to be negligible. Under this assumption, we further *normalize* both the storage and communication costs with respect to the size of the value, say v, that is written, i.e., we simply assume that the size of v is 1 unit (instead of 1 MB), and say that the worst-case storage or read or write cost of the ABD algorithm is n units, for a system consisting of n servers.

Erasure codes provide an alternative way to emulate fault-tolerant shared atomic storage, with the added benefit of reducing storage cost. In comparison with replication, algorithms based on erasure codes significantly reduce both the storage and communication costs of the implementation. An $[n, k]$ erasure code splits the value v of size 1 unit into k elements, each of size $\frac{1}{k}$ units, creates n *coded elements*, and stores one coded element per server. The size of each coded element is also $\frac{1}{k}$ units, and thus the total storage cost across the n servers is $\frac{n}{k}$ units. For example, if we use an $[n = 5, k = 3]$ MDS code, the storage cost is simply 1.67 per unit of data, instead of 5 as in the case of replication-based algorithms, such as ABD. A class of erasure codes known as Maximum Distance Separable (MDS) codes have the property that value v can be reconstructed from any k out of these n coded elements. In systems that are centralized and synchronous, the parameter k is simply chosen as $n - f$, where f denotes the number of server crash failures that need to be tolerated. In this case, the read cost, write cost and total storage cost can all be simultaneously optimized. The usage of MDS codes to emulate atomic shared storage in decentralized, asynchronous settings is way more challenging, and often results in additional communication or storage costs for a given level of fault tolerance, when compared to the synchronous setting. Even then, as has been shown in the past [6, 10], significant gains over replication-based strategies can still be achieved while using erasure codes. The works in [6, 10] contain algorithms based on MDS codes for emulating fault-tolerant shared atomic storage, and offer different trade-offs between storage and communication costs.

The performance of a DSS that stores millions of objects, and accessed concurrently by hundreds of thousands of clients must excel in terms of several performance measures. While designing FLECKS algorithm we focused on the following key performance metrics that are often used by the systems researchers to evaluate the performance of such system. (*i*) *Storage cost* is the total number of bytes stored across all servers, must be low, which essentially increases the capacity of the storage system, and also reduces the cost of storing data for the user. (*ii*) *Maximum number of server failures* the system can experience without service interruption directly contributed to increases in data durability. (*iii*) *Number of rounds per operation* reduces the latency of operations, thereby increasing the throughput of clients' operations and also reduces overall messaging in the network. (*iv*) *Read cost* is the amount of data transmitted in order to complete a read operation. In most practical systems reads are several orders of magnitude more frequent than writes. Therefore, read cost, must be as

low as possible. (v) *Write cost* is the number of bytes transmitted during a write operation should be as low as possible, which would reduce latency of write and network bandwidth consumption.

Our Contributions. In this work, we present FLECKS, an erasure-code based, fault-tolerant algorithm for implementing MWMR atomic memory in asynchronous networks, with optimized storage and communication costs. When compared to other erasure-code based or replication-based atomic memory emulation algorithms, FLECKS achieves superior or comparable values for the performance metrics mentioned above. Moreover, FLECKS is the only such algorithm that scores reasonable values across all of the performance metrics (see Table 1), making it suitable for implementations in practical systems. Firstly, the storage cost of FLECKS is $(1+\delta)\frac{n}{k}$, where δ is the maximum number of writes concurrent with any read. In a typical DSS, the frequency of reads is 10,000+ fold more than that of writes [8]. Therefore, δ is rarely larger than 1 as reported in [7]. FLECKS exploits this to provide one-round reads, but occasionally, in the presence of concurrent writes, carries out a second round. This results in lower latency for most reads and increases throughput of the system. Writes always take two rounds. We would like to emphasize that δ is not explicitly hard-coded in FLECKS; therefore, is a run-time property. The underpinning idea behind FLECKS achieving lower storage cost is to use writes help garbage collect stale values, i.e., values introduced by previous writes. As a result, during the course of an execution, the additional storage cost due to the temporary increase of δ for individual object is small and transient. In a system with several hundred or more stored objects, the fraction of reads that experiences concurrent writes would be tiny (see third plot in Fig. 1). Therefore, when considered system wide, FLECKS achieves storage cost very close to the optimal value $\frac{n}{k}$ (discussed later in the context of Fig. 2 (a)). FLECKS can tolerate a maximum of $n-k$ server crashes, which is the maximum number of erasures tolerated by and MDS $[n,k]$ code. The read and write-communication costs are very comparable to the synchronous EC-based scenarios (see Table 1). We provide analytical proofs of *atomicity* and *liveness* properties of FLECKS. We also derived bounds for the read and write latency based on maximum message delay of Δ for any point-to-point message in the network. Finally, we implemented FLECKS, deployed our implementation, and ran experiments where our implementation can emulate a large number of atomic objects. We compare our results with an optimized replication-based algorithm adapted from ABD where we emulate a shared storage of up to 10,000 objects of various sizes. Our results corroborate our design goals and theoretical results on storage and communication cost bounds, and lower latency of reads and writes in FLECKS. For example, Fig. 1 shows that FLECKS (EC) has much lower latency, compared to the replication-based method (REP) for the read and write operations. Furthermore, it shows that most of the reads (GET) comprise of a single-round.

1.1 Comparison with Other Algorithms, and Related Work

There is a rich history of erasure coding based shared memory emulation algorithms [5, 6, 10–12, 14, 18]. In Table 1, we provide a comparison between FLECKS and other atomic memory algorithms. We add ABD as a benchmark to compare the performance metrics of the erasure-coded algorithms with replication based schemes. In [6], the authors provide two algorithms - CAS and CASGC - based on $[n,k]$ MDS codes, and these are primarily motivated with a goal of reducing the communication costs. Both algorithms tolerate up to $f=\frac{n-k}{2}$ server crashes, and incur a communication cost (per read or write) of $\frac{n}{n-2f}$. The CAS algorithm is a precursor to CASGC, and its storage cost is not optimized. In CASGC,

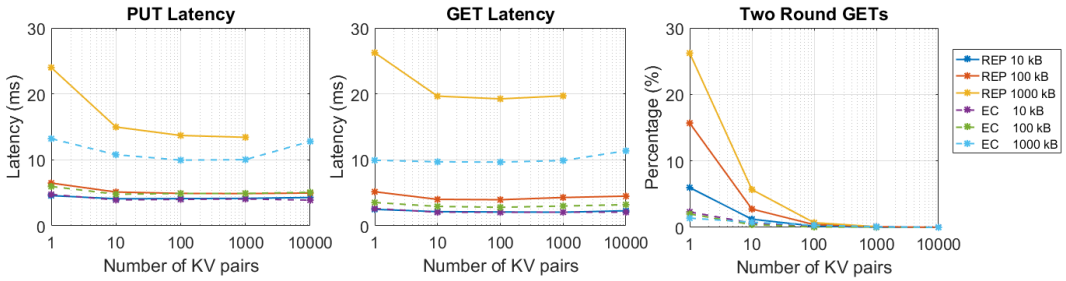


Figure 1 READ (GET) and WRITE (PUT) latencies, and percentage of READs with 2 phases for the multi object experiment. For each operation, a client accesses a object chosen uniformly at random. We compare $[n = 5, k = 3]$ FLECKS (EC) against 5-way replication (REP), for objects of sizes 10KB, 100KB and 1MB.

each server stores coded elements (of size $\frac{1}{k}$) for up to $\delta + 1$ different versions of the value v, where δ is a hard-coded upper bound on the number of writes that are concurrent with a read A garbage collection mechanism, which removes all the older versions, is used to reduce the storage cost. The worst-case total storage cost of CASGC is shown to be $\frac{n}{n-2f}(\delta + 1)$. Liveness and atomicity of CASGC are proved under the assumption that the number of writes concurrent with a read never exceeds δ. On the other hand, SODA [14] is designed to optimize the storage cost rather than communication cost, where a write cost is very high (n^2). In SODA, the parameter δ_w, which indicates the number of writes concurrent with a read, to bound the read cost. However, neither liveness or atomicity of SODA depends on the knowledge of δ_w; the parameter appears only in the analysis and not in the algorithm. But the effect of the parameter δ in CASGC is rather *rigid*. In CASGC, any time after $\delta + 1$ successful writes occurs during an execution, the total storage cost remains fixed at $\frac{n}{n-2f}(\delta+1)$, irrespective of the actual number of concurrent writes during a read. For a given $[n, k]$ MDS code, CASGC tolerates only up to $f = \frac{n-k}{2}$ failures, whereas SODA tolerates up to $f = n - k$ failures.

In [10], the authors present the ORCAS-A and ORCAS-B algorithms for asynchronous crash-recovery models. In this model, a server is allowed to undergo a temporary failure such that when it returns to normal operation, contents of temporary storage (like memory) are lost while those of permanent storage are not. Only the contents of permanent storage count towards the total storage cost. Furthermore they do not assume reliable point-to-point channels. The ORCAS-A algorithm offers better storage cost than ORCAS-B when the number of concurrent writers is small. Like SODA, in ORCAS-B coded elements corresponding to multiple versions are sent by a writer to reader, until the read completes. However, unlike in SODA, a failed reader might cause servers to keep sending coded elements indefinitely. RADON [15], an erasure-code based atomic memory algorithm which allows servers restarts, provides liveness guarantees under most practical network settings and allows efficient repair of crashed nodes. ARES [18] improves on the number of rounds compared to the previously known erasure-code based algorithms. From Table 1 is it evident that FLECKS strikes a balance among all the erasure-code based algorithms performs in all of the measures of performance.

1.2 Other related works

In [19], the authors consider algorithms that use erasure codes for emulating *regular* registers. Regularity [16] is a weaker consistency notion than atomicity. Applications of erasure codes to Byzantine fault tolerant DSS are discussed in [5, 12].

During the last few years several erasure-code-based DSS with strongly consistent distributed storage have become available. Cocytus [20] is an in-memory key-value store that guarantees strong consistency and reduces storage cost using erasure codes. The values are erasure coded and the coded elements are stored among a subset or group of servers, referred to as coding group, from the set of available servers.

Table 1 Performance metrics of replication-based, FLECKS and other algorithms with erasure-codes (for MDS code of dimension $[n, k]$) for atomic read/write memory emulation. δ is the maximum number of concurrent writes with any read during the course of an execution of the algorithm. In practice, $\delta < 4$ [7]. The optimal case is the use of EC in a synchronous system.

algorithm	max failures	rounds/ write	rounds/ read	repl or EC	storage cost	read cost	write cost	explicit δ?
ABD [4]	$\lfloor \frac{n-1}{2} \rfloor$	2	2	Repl.	n	$2n$	n	-
CASGC [6]	$\lfloor \frac{n-k}{2} \rfloor$	3	2	EC	$(\delta+1)\frac{n}{k}$	$\frac{n}{k}$	$\frac{n}{k}$	Yes
SODA [14]	$n-k$	2	2	EC	$\frac{n}{k}$	$(\delta+1)\frac{n}{k}$	$\frac{n^2}{k}$	No
ORCAS-A [10]	$\lfloor \frac{n-k}{2} \rfloor$	3	≥ 2	EC	n	n	n	Yes
ORCAS-B [10]	$\lfloor \frac{n-k}{2} \rfloor$	3	3	EC	∞	∞	∞	-
RADON$_c$ [15]	$\lfloor \frac{n-k}{2} \rfloor$	2	2	EC	$(\delta+1)\frac{n}{k}$	$(\delta+2)\frac{n}{k}$	$\frac{n}{k}$	Yes
ARES [18]	$\lfloor \frac{n-k}{2} \rfloor$	2	2	EC	$(\delta+1)\frac{n}{k}$	$(\delta+1)\frac{n}{k}$	$\frac{n}{k}$	Yes
FLECKS	$n-k$	2	≤ 2	EC	$(\delta+1)\frac{n}{k}$	$(\delta+1)\frac{n}{k}$	$\frac{n}{k}$	No
SYNCH EC	$n-k$	1	1	EC	$\frac{n}{k}$	1	$\frac{n}{k}$	-

Giza [7] is a recently proposed strongly-consistent multi-version object store and heavily used in Microsoft's OneDrive storage system. Giza is designed for cross-data center (cross-DC) object storage, which is deployed over 11 data-centers around the world. Giza uses FastPaxos [17] which, in the absence of concurrent writes, completes in one round trip, but in the case of concurrent updates, uses the more expensive consensus algorithm Paxos.

Recently, a large class of new erasure codes have been proposed and employed (see [9] for a survey) in DSS where the focus is on the efficient storage of immutable (like archival) data. Recovery of contents in failed servers is an important operation in such systems. These new codes offer the dual benefits of reduced storage cost as well as reduced repair cost during recovery from server failures. It remains to be seen whether the advantages of these codes carry over to systems that have consistency/concurrency requirements.

Document Structure. In Section 2, we provide the models and definitions. In Section 3 we describe FLECKS. Section 4 provides the proof for correctness and liveness guarantees for FLECKS along with bounds for storage and communication costs, and latency analysis of the operations. In Section 5, we discuss the implementation and experimental validation of FLECKS. Finally, in Section 6 we conclude our paper. Due to lack of space some of the proofs are omitted.

2 Model and Definitions

A shared atomic storage can be emulated by composing individual atomic objects. Therefore, we aim to implement a single atomic read/write memory object. Each data object takes a value from a set $\mathcal{V}$. We assume a system consisting of three distinct sets of processes: a set $\mathcal{W}$ of writers, a set $\mathcal{R}$ of readers and $\mathcal{S}$, a set of servers. Servers host data elements (replicas or encoded data fragments). Each writer is allowed to WRITE the value of a shared object, and each reader is allowed to READ the value of that object. Processes communicate via *messages* through *asynchronous*, *reliable* channels.

Executions. An *execution* of an algorithm A is an alternating sequence of states and actions of A starting with the initial state and ending in a state. An execution ξ is *well-formed* if each client does not invoke a one operation until it completed the previously invoked operation and it is *fair* if enabled actions perform a step infinitely often. In the rest of the paper we consider executions that are fair and well-formed. When process p *crashes* it stops executing any further step.

Write and Read Operations. An implementation of a read or a write operation contains an *invocation* action and a *response* action (such as a return from the procedure). An operation π is *complete* in an execution, if it contains both the invocation and the *matching* response actions for π; otherwise π is *incomplete*. We say that an operation π *precedes* an operation π' in an execution ξ, denoted by $\pi \to \pi'$, if the response step of π appears before the invocation step of π' in ξ. Two operations are *concurrent* if neither precedes the other.

Erasure Codes. ***Background on Erasure coding:*** In $FLECKS$, we use an $[n, k]$ linear MDS code [13] over a finite field $\mathbb{F}_q$ to encode and store the value v among the n servers. An $[n, k]$ MDS code has the property that any k out of the n coded elements can be used to recover (decode) the value v. For encoding, v is divided[2] into k elements $v_1, v_2, \ldots v_k$ with each element having size $\frac{1}{k}$ (assuming size of v is 1). The encoder Φ takes the k elements as input and produces n coded elements $c_1, c_2, \ldots, c_n$ as output, i.e., $[c_1, \ldots, c_n] = \Phi([v_1, \ldots, v_k])$. For ease of notation, we simply write $\Phi(v)$ to mean $[c_1, \ldots, c_n]$. The vector $[c_1, \ldots, c_n]$ is referred to as the codeword corresponding to the value v. Each coded element c_i also has size $\frac{1}{k}$. In our scheme we store one coded element per server. Without loss of generality, we associate the coded element c_i with server i, $1 \leq i \leq n$.

Liveness of operations. We require algorithms to satisfy certain liveness properties, specifically, in every fair execution that satisfies certain restrictions in terms of the number of failed nodes, we require every operation by a non-faulty client completes, irrespective of the behavior of other clients.

Storage and Communication Costs. We define the total storage cost as the size of the data stored across all servers, at any point during the execution of the algorithm. The communication cost associated with a read or write operation is the size of the total data that gets transmitted in the messages sent as part of the operation. We assume that metadata, such as version number, process ID, etc. used by various operations is of negligible size, and therefore, ignore this in the calculation of storage and communication cost. Further, we normalize both the costs with respect to the size of the value v; in other words, we compute the costs under the assumption that v has size 1 unit.

3 The FLECKS algorithm

The FLECKS algorithm is presented in three parts in Pseudocodes. 1, 2 and 3, corresponding to a writer, reader and server, respectively. The erasure-code parameter k is chosen as $k = n - f$, where f is the desired server-fault tolerance. By assumption, $f < n/2$, and thus

[2] In practice v is a file, which is divided into many stripes based on the choice of the code, various stripes are individually encoded and stacked against each other. We omit details of represent-ability of v by a sequence of symbols of $\mathbb{F}_q$, and the mechanism of data striping, since these are fairly standard in the coding theory literature.

Algorithm 1 Writer protocol in FLECKS: WRITE(v) at writer w.

Variables:
opnum, indicates the operation number for the writer. Initially 1.
put-data:
  Compute coded elements $c_1, \ldots, c_n$ from v
  Send $(opnum, c_i)$ to server s_i, $1 \leq i \leq n$.
  Wait for responses from k servers. Let the responses be $\{z_i, 1 \leq i \leq k\}$.
  Compute $z = \max_i z_i$

put-tag:
  Let $t = (w, z)$
  Send $(t, opnum)$ to server s_i, $1 \leq i \leq n$.
  $opnum++$. Terminate after receiving k acknowledgments.

Algorithm 2 Reader protocol in FLECKS: READ at reader r.

get-tag-data:
  Request final tuple from all servers
  Wait for responses from k servers.

  if all k responses have common tag **then**
    decode the corresponding value
    return the value.
  else
    compute the maximum received tag and call it t_{req}
    Let $opnum_{req}$ be the corresponding $opnum$.
    collect all coded elements corresponding to t_{req} in list D_L

get-data:
  Send $(t_{req}, opnum_{req})$ to all servers.
  Collect every response $(t, opnum, c)$ into D_L.
  for received tuple $(t, opnum, c)$, **do**
    if $\exists$ k coded elements t in D_L **then**.
      decode the value v for tag t
      send *read-complete* to all servers
      return v
    else
      if $t > t_{req}$ **then**
        send *commit-tag*$(t, opnum)$ to all servers,
        Continue to wait for more tuples.

we get that $k > n/2$. The algorithm relies on the notion of quorums during both phases of the WRITE operation, and the first phase of the READ operation. The parameter k denotes the size of quorum in these phases, and is at least a majority since $k > n/2$.

Tags are used for version control of key values. A tag t is defined as a pair (z, w), where z is an positive integer and $w \in \mathcal{W}$ denotes the writer ID. We use $\mathcal{T}$ to denote the set of all possible tags. For any two tags $t_1, t_2 \in \mathcal{T}$ we say $t_2 > t_1$ if (i) $t_2.z > t_1.z$ or (ii) $t_2.z = t_1.z$ and $t_2.w > t_1.w$. The relation $>$ imposes a total order on $\mathcal{T}$.

Server-side Local Variables: Each server maintains the following local variables: a) a List $L \subset Tags \times \mathbb{N} \times$ coded elements $\times \{Pre, Fin\}$, which forms a temporary storage for tag and coded-elements pairs during WRITE operations. The second entry indicates the operation number (opnum) of the writer whose entry is stored. The last entry's meaning will be described further in the text. b) A finalized tuple $(t_f, opnum_f, c_{i,f})$. We refer to t_f as the finalized tag, $opnum_f$ as the finalized opnum, and $c_{i,f}$ as the finalized coded-element, c) $Op(w), w \in \mathcal{W}$, indicating the last opnum received from writer w, and d) the set $\mathcal{R}$ of outstanding READ requests. An element of $\mathcal{R}$ is the form $(r, t_{req}, opnum_{req})$.

We now describe the WRITE and READ operations with the help of Pseudocode 1, 2 and 3, and a high-level schematic diagram for the read and write operations are given in Fig. 2.

Algorithm 3 Server response protocol in FLECKS: at server s_i, $1 \leq i \leq n$.

Variables:
List $L \in Tags \times \mathbb{N} \times \mathbb{F}_{256} \times \{Pre, Fin\}$,
initially empty.
Last Opnum received from each writer: $Op(w), w \in \mathcal{W}$
Final tuple $(t_f, opnum_f, c_{i,f})$, initially $(t_0, opnum_0, c_{i,0})$.
The set $\mathcal{R}$ of outstanding READ requests.
An element of $\mathcal{R}$ is the form $(r, t_{req}, opnum_{req})$. Initially, empty.

put-data-resp **received** $(opnum, c_i)$ **from** w :
  $Op(w) = \max(Op(w), opnum)$
/*change from Fin to Pre for writing of algorithm*/
  if $((w, \tilde{z}), opnum, \perp, Fin) \in L$ **then**
    $L \leftarrow L \cup \{((w, \tilde{z}), opnum, c_i, Pre)\}$
    Do *commit-tag*$((w, \tilde{z}), opnum)$
  else
    Let $t_{in} = (w, t_f.z + 1)$.
    $L \leftarrow L \cup \{(t_{in}, opnum, c_i, Pre)\}$.
    Send t_{in} to writer w.

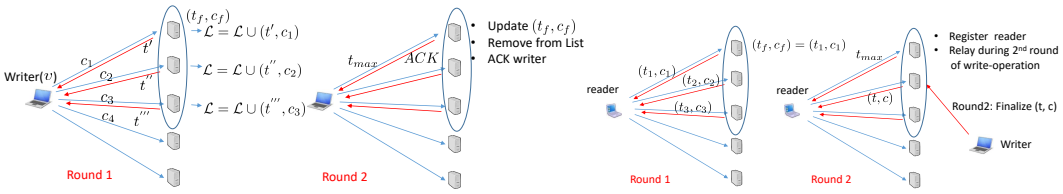
put-tag-resp **received** $(t, opnum)$ **from** w :
  Do *commit-tag*$(t, opnum)$
  Send ACK to writer w.

get-tag-data-resp **request received from** r :
  Send final tuple $(t_f, opnum_f, c_{i,f})$ to reader r

get-data-resp **received** $(t_{req}, opnum_{req})$ **from** r :
  $\mathcal{R} = \mathcal{R} \cup (r, t_{req}, opnum_{req})$.
  if $t_f \geq t_{req}$ **then**
    send $(t_f, opnum_f, c_{i,f})$ to reader r.
  Do *commit-tag*$(t_{req}, opnum_{req})$

read-complete-resp **request received from** r :
  $\mathcal{R} = \mathcal{R} \backslash (r, *, *)$.

commit-tag-resp$(t, opnum)$:
  Let $t = (w, z)$.
  if $((t.w, *), opnum, c_i, Pre) \in L$ **then**
    Update final tuple:
    if $t > t_f$ **then**
      $(t_f, opnum_f, c_{i,f}) \leftarrow (t, opnum, c_i)$.
    Relay: Send $(t, opnum, c_i)$ to every $r, (r, t_{req}, *) \in \mathcal{R}$ s.t. $t \geq t_{req}$.
    Remove from list: $L = L \backslash \{((w, *), opnum, c_i, *)\}$.
  else if $opnum > Op(t.w)$
    For Future:$L \leftarrow L \cup (t, opnum, \perp, Fin)$



(a) WRITE operation. **(b)** READ Operation.

Figure 2 High Level schematic overview of the WRITE and READ protocols of FELCKS.

The write Operation. Assume that a writer w wishes to WRITE (update to) value v. The writer computes the n coded elements $[c_1, \ldots, c_n]$. The WRITE operation consists of two rounds. At a high level, the first round is the temporary storage phase, where the server adds the coded element into the list. Once the writer gathers that k servers have done so, it starts the second round where a commit command is issued whereby the server updates the finalized tuple using the entry in the list (if the entry is newer). A pictorial overview of the WRITE protocol appears in Pseudocode 1. We now explain the two rounds in detail.

In the first round *put-data*, the writer sends the pair $(opnum, c_i)$ to server $s_i, 1 \leq i \leq n$, where *opnum* denote the writer's operation number for the ongoing WRITE operation. The server responds via *put-data-resp*. Upon receiving the message, under normal circumstances (the **else** part of the **if** statement), the server computes a new tag for this WRITE operation. This is obtained as $t_{in} = (w, t_f.z + 1)$, where t_f denotes the finalized tag stored by s_i, and $t_f.z$ denotes the integer part of the t_f. The server adds the tuple $(t_{in}, opnum, c_i, Pre)$ to the temporary storage list L, and responds to the writer by sending t_{in}. The **if** part of the pseudo-code is to take care of the rare case, when the message from the writer arrives too slow at the server, where the server has already learned by other means that the WRITE operation has already been *committed* by a quorum of servers. In this case, server s_i directly commits the message $(opnum, c_i)$ in round 1. The commit step, under normal circumstances, is part of the second round response of the WRITE operation, and is explained below. The writer waits to hear tags from k servers, and computes maximum z of the integer parts of the received tags. This completes round 1.

In the second round *put-tag*, the writer w creates the new tag $t = (w, z)$, and sends the pair $(t, opnum)$ to all servers. Upon receiving the message, a server performs, via *put-tag-resp*, the *commit-tag* step. Under normal circumstances (the **if** clause of *commit-tag-resp*), as part of the commit-tag-response, the server updates the finalized tuple with the entry in the list corresponding to $(t.w, opnum)$, if[3] $t > t_f$. The server also removes the entry from the list. This ensures that for any successful WRITE operation, every non-faulty server eventually automatically garbage-collects the temporary storage entry in the list. The **if** clause of the *commit-tag-resp* contains a *Relay* step that is used to server outstanding READ requests. This is explained as part of the READ operation below. The **else** part of *commit-tag-resp* step is executed during rare circumstances, when the server initiates the *commit-tag* step not as part of the round 2 of the corresponding WRITE operation, but learns from a reader that the WRITE operation has already begun the second round but this server has not even received the first round message form the writer yet. In the case, the server adds an indicator entry to the list L (using the forth Pre/Fin part of the entry), so that when the writer message arrives in future, the server can directly proceed to commit the coded-element. Finally, the writer terminates after receiving acknowledgments from k servers.

The read Operation. The reader during the first round contacts all the servers for the finalized tuples, and waits for responses from k servers. If all the responses have the same tag, clearly the reader can decode using the k responses, and the READ ends in the first round itself. Otherwise, the reader computes the maximum tag from among the tags received as part of the finalized tuples, and we call this the request tag t_{req}. The corresponding

[3] It is possible that the local temporary tag for corresponding the entry in list is higher than the received tag t. The reason is that the writer computes the tag by computing maximum among a quorum, and not all the servers. This local temporary tag is simply ignored, and the finalized tuple is saved using the tag received from the writer. The local temporary tag is used during the second round only to identify the correct entry in the list that must be committed.

$opnum_{req}$ is called request opnum. The goal in the second round is to use the relay-technique to let the reader decode a value corresponding to a tag that is at least as high as t_{req}. A pictorial overview of READ protocol appears in Pseudocode 2.

In the second round, the reader sends the pair $(t_{req}, opnum_{req})$ to all servers. Any server that receives the message *registers* the read-request, as part of the *get-data-resp* by adding the tuple $(r, t_{req}, opnum_{req})$ to the set $\mathcal{R}$ of outstanding READ requests. Further, if the finalized tag is at least as high as the request tag, the server sends finalized tuple to the reader. The goal of the reader registration is to enable *relaying* to the reader until the reader gathers k coded elements corresponding to some common tag. The relaying (to outstanding READ requests) happens whenever the server executes the *commit-tag-resp* step for a pair $(t, opnum)$ such that $t \geq t_{req}$. Recall that *commit-tag-resp* step is executed as part of the second round response of WRITE operations. It may be noted that a server only sends those (tag, coded-element) pairs that are committed, and thus form potential candidates for the finalized tuple. In this regard, from the point of view of the reader, the temporary storage list L can be thought as elongating the channel from the writer to the server such that a (tag, coded-element) pair is ready for consumption by the server only after the writer executes the second round.

As part of the *get-data-resp* step, the server also performs the *commit-tag* step for the pair $(t_{req}, opnum_{req})$. This is to handle the case where the writer crash fails half-way into the second round for the WRITE operation corresponding to $(t_{req}, opnum_{req})$. In this case, only a partial set of the servers would have performed *commit-tag* step for the pair $(t_{req}, opnum_{req})$, while the rest of the servers still hold the coded elements in the temporary storage list L. The execution of the *commit-tag* step as part of the READ operation is in spirit analogous to the reader-write-back (read-repair) operation performed replication algorithms [4], and helps complete a partially completed WRITE operation.

The reader collects (tag, coded-element) pairs until it receives k corresponding to a common tag, say t_r, whose corresponding value is decoded. During this process, if the reader receives a coded-element for a tag $t > t_{req}$, then (while waiting for further pairs), the reader sends out *commit-tag*$(t, opnum)$ message to the servers. The purpose of this commit tag is exactly the same as that of the *commit-tag*$(t_{req}, opnum_{req})$ described above. It may be noted that the utility of these messages only arise when the WRITE corresponding to $(t, opnum)$ failed half-way. Under normal circumstances, these messages are simply ignored by the server that has already seen the writer *commit-tag* message. In fact, as we shall see in the experiments, even with read-write ratio of 1, the number of reads needing the second round is a tiny fraction.

Finally, once the reader decodes, it sends a READ complete message so that the servers can stop relaying. Note that no responses are expected for these read-complete messages.

Handling Client Failures. While we show that FLECKS ensures linearizable executions and wait-freedom availability corresponding to non-faulty client processes despite failure of a reader or and writer process, we note that a failed reader/writer process introduces the need for additional intervention for performance optimization. A failed reader can result in servers relaying to the reader indefinitely. While it is definitely possible to stop relaying algorithmically as in [14] via a gossip protocol among the servers, the protocol is redundant for successful reads, and thus contributes high burden on the system from a practical point of view. Alternate practical solutions include letting the server stop the relaying after a certain timeout duration or threshold number relay messages. In fact, if point-to-point channel latency is bounded by Δ, any READ operation completes within 6Δ (see Section 4),

independent of the number of concurrent writes. In the rare event when the relaying stops even before the READ completes (when the point-to-point latency bound is not respected), one can always timeout the reader, and restart the read.

Similarly, a WRITE that fails during the first round leaves entries in the temporary storage list L that is not garbage collected by the algorithm. In our implementation, each server additionally garbage collects any entry in the list that is older than a certain threshold time that is set sufficiently high from a practical viewpoint.

4 Liveness and Atomicity of FLECKS

Liveness. Now we state and prove the liveness property of FLECKS. We recall that the algorithm uses an $[n, k]$ MDS code. We assume if a client has already started an operation (say π), the (same) client will wait until π is completed before starting a new operation.

▶ **Theorem 1.** *(**Liveness**) Consider any well-formed execution of FLECKS in which at most $f = n - k$ servers crash fail during the execution. Then, an operation corresponding to a non-faulty client completes irrespective of any past, ongoing or future successful or failed client operations.*

Proof. Liveness of a WRITE operation is easily verified from an inspection of the algorithm. For a READ operation, there is nothing to prove if the READ completes in the first round itself. The non-trivial part is proving liveness of a READ operation that executes the second phase. Let π be such a READ operation corresponding to reader r. As in the algorithm, let $(t_{req}, opnum_{req})$ denote the message sent by the reader during the *get-data* phase. Without loss of generality, let $s_1, \ldots, s_k$ denote the set of k servers that never fail during the execution. Let T_i denote the point of execution when s_i receives the *get-data* request from reader r. Let $T_{max} = max_{1 \leq i \leq k} T_i$. Next, let $t_i = s_i.t_f|_{T_{max}}$, i.e., t_i denotes the finalized tag stored by server s_i at T_{max}. Further, let $t_{max} = max_{1 \leq i \leq k} t_i$. The tags t_{max} and t_{req} are not necessarily ordered in any specific way. We now divide the discussion into the following cases:

Case a) $t_{max} \leq t_{req}$: In this case, we show that corresponding to every server $s_i, 1 \leq i \leq k$, there exists a point of execution $\hat{T}_i$ when s_i will send the message $(t_{req}, opnum_{req}, c_i)$ to reader r, unless s_i received read-complete message before $\hat{T}_i$. In this case, it is clear that the reader gets k coded elements corresponding to the tag t_{req} and thus, can definitely decode the value corresponding to t_{req}, after receiving the k^{th} coded-element, unless the READ is complete even before. We consider two sub cases here:

Subcase i) Sever s_i did not receive put-data request with message $(t_{req}.w, opnum_{req}, c_i)$ until T_i: We know that the server s_i registers the READ request at T_i (by adding the corresponding entry to $\mathcal{R}$). Further, by assumption the channel from every writer to every server is ordered, and thus if the server has not received the *put-data* request with message $(t_{req}.w, opnum_{req}, c_i)$ until T_i, this means that $s_i.Op(w)|_{T_i} < opnum_{req}$. In this case, the server adds the tuple $(t_{req}, opnum_{req}, \perp, Fin)$ to its list as part of the execution of *commit-tag* step of *get-data-resp*. Let $\tilde{T}_i > T_i$ denote the point of execution when s_i receives *put-data* request with message $(t_{req}.w, opnum_{req}, c_i)$. Such a point in the execution necessarily exists because the tag t_{req} is committed tag, and thus at least one server received the *put-tag* request with message $(t_{req}, opnum_{req})$ directly from writer $t_{req}.w$. This means that the writer $t_{req}.w$ necessarily completed the *put-data* phase in which messages were sent to all n servers (since it already executed at least a part of the second phase). We recall here our channel model assumption that once message is placed in the channel, it is eventually delivered to the destination process, as long as the destination is non-faulty. In the current

proof, the server s_i is non-faulty, and thus will eventually receive $(t_{req}.w, opnum_{req}, c_i)$. This completes our justification of the existence of the point of execution $\tilde{T}_i$.

To continue with the proof, we note that during the *put-data-resp* action corresponding to $(t_{req}.w, opnum_{req}, c_i)$, server s_i finds that the WRITE operation has an entry in the list with Fin in the last field, and consequently executes *commit-tag* for the same WRITE operation. In this case, if s_i did not receive read-complete message until $\tilde{T}_i$, it is clear that server will relay the tuple $(t_{req}, opnum_{req}, c_i)$ to reader r, as part of the execution of *commit-tag-resp*$(t_{req}, opnum_{req})$. Note that in this case, we have $\hat{T}_i = \tilde{T}_i$.

Subcase ii) Sever s_i received put-data request with message $(t_{req}.w, opnum_{req}, c_i)$ before T_i: In this case, we first note that $s_i.tf|_{T_i} \leq s_i.tf|_{T_{max}} \leq t_{max} \leq t_{req}$. If $s_i.tf|_{T_i} = t_{req}$, then the server sends the tuple $(t_{req}, opnum_{req}, c_i)$ to reader r as part of execution Step 2. of *get-data-resp* corresponding to message $(t_{req}, opnum_{req})$. If $s_i.tf|_{T_i} < t_{req}$, then it is clear that s_i never received *commit-tag*$(t_{req}, opnum_{req})$ request until T_i, and hence it must be true that the tuple $(t_{req}.w, opnum_{req}, c_i) \in s_i.L|_{T_i}$. In this case, the tuple $(t_{req}.w, opnum_{req}, c_i)$ is relayed to the reader r as part of the execution of Step 3, *commit-tag-resp*$(t_{req}.w, opnum_{req})$, of the *get-data-resp* action.

Case b) $t_{max} > t_{req}$: In this case, we show that corresponding to every server $s_i, 1 \leq i \leq k$, there exists a point of execution $\hat{T}_i$ when s_i will send the message $(t_{max}, opnum_{max}, c_i)$ to reader r, unless s_i received read-complete message before $\hat{T}_i$. In this case, it is clear that the reader gets k coded elements corresponding to the tag t_{max} and thus, can definitely decode the value corresponding to t_{max}, after receiving the k^{th} coded-element, unless the READ is complete even before.

To prove this, observe that there exists a server $s_j \in \{s_1, \ldots, s_k\}$ such that $s_j.tf|_{T_{max}} = t_{max}$. We know that $T_j \leq T_{max}$, and hence $s_j.tf|_{T_j} \leq s_j.tf|_{T_{max}} = t_{max}$. If $s_j.tf|_{T_j} = t_{max}$ (trivially true if $T_{max} = T_j$), the server s_j sends the tuple $(t_{max}, opnum_{max}, c_j)$ to reader r as part of the execution Step 2 of *get-data-resp*. If $s_j.tf|_{T_j} < t_{max}$, it is clear that there exists a point of execution $\hat{T}_j, T_j < \hat{T}_j < T_{max}$, where server s_j executes *commit-tag-resp*$(t_{max}, opnum_{max})$ and changes the finalized tag to t_{max}. Thus, the server s_j relays the tuple $(t_{max}, opnum_{max}, c_i)$ to reader r at $\hat{T}_j$, if the server s_j has not yet received read-complete response. In summary, we have shown that there exists one server s_j among the set of non-faulty servers that will definitely send the tuple corresponding to $(t_{max}, opnum_{max})$ to the reader. Once the reader gets the first coded element corresponding to the pair $(t_{max}, opnum_{max})$, since $t_{max} > t_{req}$, the reader sends the *commit-tag*$(t_{max}, opnum_{max})$ message to all the servers.

It remains to be shown that every other server $s_i \in \{s_1, \ldots, s_k\} \backslash \{s_j\}$ also sends coded element corresponding to $(t_{max}, opnum_{max})$ to the reader. To show this, we once again observe that $s_i.tf|_{T_i} \leq s_i.tf|_{T_{max}} \leq t_{max}$. If $s_i.tf|_{T_i} = t_{max}$, it is clear that the server s_i sends the tuple $(t_{max}, opnum_{max}, c_i)$ to reader r as part of the execution Step 2 of *get-data-resp*. Now consider the case $s_i.tf|_{T_i} < t_{max}$. The READ request is clearly registered. From the discussion so far, we note that the server s_i will eventually receive both the *put-data* request corresponding to message $(t_{max}.w, opnum_{max}, c_i)$, and also the *commit-tag* request corresponding to message $(t_{max}, opnum_{max})$. The *put-data* request is eventually received since the writer has definitely completed the Phase 1 of the WRITE operation, and we know from the channel assumption that once a message is placed in the channel, it eventually arrives at the destination. The *commit-tag* request is eventually received since as observed above the reader sends the *commit-tag*$(t_{max}, opnum_{max})$ message to all the servers (useful if the writer failed during the execution of Phase 2 of the corresponding WRITE operation). Further, the algorithm is designed in such a way that the ordering of the arrivals of these two messages does not matter; arguments (using the Pre/Fin indicator) similar to those

used in Case a) can be used to show that the tuple $(t_{max}, opnum_{max}, c_i)$ is committed at the earliest point in the execution when both these messages are received. In this case, the server s_i relays the tuple corresponding to $(t_{max}, opnum_{max})$ to the reader, if s_i did not get *read-complete* message yet. This completes the proof of Case b), and hence the proof of liveness of a READ operation corresponding to a non-faulty reader. ◀

Atomicity. Below we state and prove the atomicity property of the FLECKS algorithm.

▶ **Theorem 2.** *(**Atomicity**) Any well-formed execution of FLECKS is atomic.*

Latency Analysis and Storage Cost. Although FLECKS is designed for asynchronous message passing settings, in the case of a reasonably well-behaved network we can bound the latency of an operation. Assume that any message sent on a point-to-point channel is delivered at the corresponding destination (if non-faulty) within a duration $\Delta > 0$, and local computations take negligible amount of time compared to Δ. Thus, latency in any operation is dominated by the time take taken for the delivery of all point-to-point messages involved. Under these assumptions, the latency bounds for successful WRITE and READ operations in FLECKS are as follows.

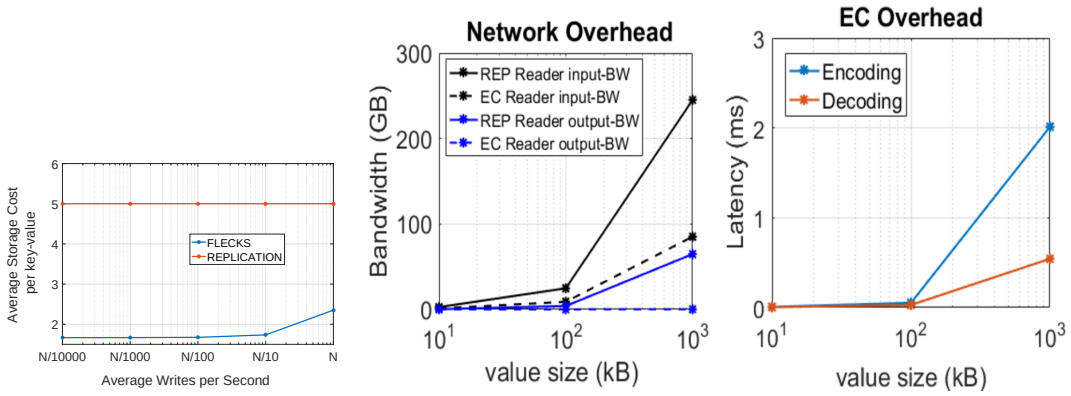
▶ **Theorem 3.** *The duration of a* WRITE *or a* READ *in FLECKS is at most* 4Δ *and* 6Δ*, respectively.*

Recall that READ operations use the technique of relaying for completion, and a new relay to the reader potentially occurs due to every concurrent WRITE operation. While this may happen, the above result guarantees a bound on the READ completion time that is independent of the number of concurrent writes experienced by the read.

Storage Costs. We now provide bounds on the total storage cost incurred by FLECKS under the bounded latency model. The storage cost at any point in the execution is the total amount of data that is stored in the servers. The cost at any server arises due to the storage of finalized coded-element as well as the storage of temporary coded-elements in the list - we account for both of these in our calculation. Costs contributed by meta-data are ignored while ascertaining either storage costs.

Consider a system storing N key-value pairs, where each pair is implemented via an instance of FLECKS. We assume using an $[n, k]$ MDS code for each of these instances. Further, every value is assumed to have the same size, and let us normalize it to 1 unit of space. Let ρ denote the average number of writes per second experienced by the system, where each WRITE can happen on any of the N objects allowing for concurrency. Further let θ denote the fraction of writes that fail (due to writer crashes). We know from the algorithm that the coded elements from such writes can potentially linger around in the temporary list until an external mechanism garbage collects them. Let τ denote the maximum duration for which any entry is retained in the list by a server - we assume that after τ seconds of adding an entry into the list, the server simply garbage collects the entry if it was not removed until then (automatically by the algorithm). The following theorem gives the average storage cost in the system in terms of the above parameters under the bounded latency model.

▶ **Theorem 4.** *The average storage cost per key-value pair incurred by a system running FLECKS under the bounded latency model is given by* $\frac{n}{k}\left[1 + \frac{(4\Delta+\theta\tau)\rho}{N}\right]$.



(a) Storage cost per object. **(b)** Bandwidth cost of reads **(c)** Delay due to encode or decode

Figure 3 (a) Average storage cost per object for ABD (5-way replication) and FLECKS using an $[n = 5, k = 3]$ erasure code is plotted as a function of number of WRITEs per second with $N = 10^4$ objects. Even for one WRITE per object per second, FLECKS significantly saves storage over ABD, for similar fault tolerance. (b) The total bandwidth consumed by each reader after executing $50,000$ reads. (c) The average latency to encode or decode a value. The plots are for runs with frequency of READ and WRITE is 1.

Proof. Cost at server s is given by $C_s = C_{s,1} + C_{s,2}$, where $C_{s,1}$ is the cost due to finalized entries, and $C_{s,2}$ is due to the entries in the list. The total storage cost C is then given by

$$C = \sum_s C_{s,1} + \sum_s C_{s,2} = N\frac{n}{k} + \sum_s C_{s,2}, \tag{1}$$

where Nn/k is the total cost in the system due to the finalized entries. Note that the total number of servers in the system does not appear anywhere in our analysis. To estimate the second term, we note that any point T in the execution, the average number of active writes retained by the system is given by $4\Delta\rho$. This follows because we know the from Theorem 3 that a WRITE completes within 4Δ seconds, and on average there are $4\Delta\rho$ writes that started within the time interval $[4\Delta - T, T]$ that remain active at time T. We also need to count the number of failed writes retained by the system at time T. The average number of failed writes retained by system at time T is given by $\tau\theta\rho$, and the argument is similar to the one for active writes. Thus, if $\sum_s C_{s,2}$ denotes the average cost due to the entries in the list across all servers, then this is given by $\sum_s C_{s,2} = \frac{(4\Delta\rho+\theta\tau\rho)n}{k}$. Now, the average cost per key-value pair in the system is given by $C/N = \frac{n}{k} + \frac{(4\Delta\rho+\theta\tau\rho)n}{Nk} = \frac{n}{k}\left[1 + \frac{(4\Delta+\theta\tau)\rho}{N}\right]$. ◀

An illustration of the storage cost bound is provided in Fig. 3 (a). In this example, we assume an $[n = 5, k = 3]$ code for a system storing $N = 10^4$ key-value pairs, where 0.01% of writes fail, i.e., $\theta = 10^{-4}$. We fix $\Delta = 100$ ms and $\tau = 100$ s, and these two numbers are based on observations from our own experiments. The storage cost is plotted as a function of writes per second in the system. For comparison, we also plot the storage cost that would be incurred by a 5-way replicated system.

5 Implementation and Experimental Validation

Here we briefly describe our experimental evaluation of FLECKS against an optimized version of the ABD algorithm. The algorithms (FLECKS and ABD) are implemented in Golang

version *go 1.6.3* with additional libraries for messaging (ZMQ [3]), erasure-coding (ISA-L [1]) and stats collection (libstatgrab [2]). The software is deployed via docker containers. For point to point communication among the processes, we use *ZMQ* 3.2.0 [3], which is a distributed (without a centralized broker) messaging library built on top of TCP/IP sockets. For the erasure-coding part of the implementation we use the open-source version of Intel's ISA-L [1]. We use the Cauchy matrix based MDS codes. We chose Galois field of size 256, since $GF(256)$ is fairly standard in the storage industry.

System Setting. We deployed each server and client process on a separate virtual machine (VM) running Ubuntu Linux 16.04 LTS configured with 8 GB of RAM and a 4-core CPU. The VMs were part on an OpenStack cloud platform. The bisectional bandwidth of the platform is about 10 Gbps.

In our experiments we stored up to 10000 atomic objects, where each object is implemented via an independent instance of FLECKS. Each server runs as a single threaded process handling all the objects associated with that server. A client process can access any of the objects. All data is stored in memory. For simulating crash failure of server process, we simply kill the process.

Latency of read and write operations. In Fig. 1, we plot average latency for reads and writes while accessing multiple objects (1, 10, 100, 1000 and 10000 objects) in executions of FLECKS and ABD. For this scenario, we use 5 readers, 5 writers, and 5 servers. We compare 5-way replication ABD with FLECKS based on $[5, 3]$ erasure-code. We notice that FLECKS has substantially reduction in latency and this improvement is more prominent as the size of payload increases.

Bandwidth cost for operations. Fig. 3(b) shows the total incoming and outgoing network bandwidth (BW) consumed by a single reader client in FLECKS and ABD. With 50000 operations and 5-way replication ABD, we expect incoming BW to be about 250 GB when object size is 1000 kB. From Fig. 1, we see that about 27% of that READs have two phases in ABD, and thus outgoing BW, dominated by two phase READs, is around $0.27 * \times 250 = 67$ GB. In FLECKS, the incoming BW is dominated by 1 phase READs, and is about $1/3 \times 250 = 83$ GB. Unlike replication, the 2 phase READs (roughly 3%) in FLECKS does not write-back actual data, and hence outgoing BW of FLECKS is negligible.

Latency due to encoding and decoding. Fig. 3(c) also shows the contribution of erasure code encoding and decoding time during a WRITE or a READ in FLECKS. Clearly, latency is minimally affected by the erasure-coding operations, consistent with other recent works in literature [20].

Server failures. To test the effect of server failures, we setup 1000 objects on 10 servers as in the experiment. After deployment, we kill two of the server processes (chosen at random). In agreement to our liveness guarantees the read and writes operations continue to complete. For a replicated system, increasing the number of replicas per object increases latency of operation.

Effect of Increasing Number of Readers. For a practical system, one expects to see a near-linear scaling of overall READ throughput against the number of readers. While we see this behavior for both replication and FLECKS, we noted that FLECKS permits a significantly better throughput scaling. The advantage can be directly attributed to the lower READ latency of FLECKS.

6 Conclusion

We investigated the feasibility of erasure-codes in atomic memory algorithms to reduce storage cost, bandwidth costs and latency. With that in mind We designed FLECKS for

asynchronous networks, that reduces, storage cost for the stored object and bandwidth cost for the operation. FLECKS completes the read operations in just one round in the absence of concurrent writes. FLECKS design is based on practical settings. FLECKS guarantees liveness of operations in the present of any client crash failures and up to $n-k$ server crashes. We proved the atomicity and liveness properties of FLECKS. We implemented FLECKS according to our algorithmic specifications. We performed extensive experiments on an actual network environment. Future work will invoke extending FLECKS to allow repair of crashed servers.

References

1 Intel® Intelligent Storage Acceleration Library (Intel® ISA-L). `https://software.intel.com/en-us/storage/ISA-L`. [Online; accessed 23-August-2018].

2 libstatgrab. `https://www.i-scream.org/libstatgrab/`. [Online; accessed 23-August-2018].

3 ZeroMQ: Distributed Messaging. `http://zeromq.org/`. [Online; accessed 23-August-2018].

4 H. Attiya, A. Bar-Noy, and D. Dolev. Sharing Memory Robustly in Message Passing Systems. *Journal of the ACM*, 42(1):124–142, 1996.

5 Christian Cachin and Stefano Tessaro. Optimal Resilience for Erasure-Coded Byzantine Distributed Storage. In *2006 International Conference on Dependable Systems and Networks (DSN 2006), 25-28 June 2006, Philadelphia, Pennsylvania, USA, Proceedings*, pages 115–124, Los Alamitos, CA, USA, 2006. IEEE Computer Society. `doi:10.1109/DSN.2006.56`.

6 Viveck R. Cadambe, Nancy A. Lynch, Muriel Médard, and Peter M. Musial. A coded shared atomic memory algorithm for message passing architectures. *Distributed Computing*, 30(1):49–73, 2017.

7 Yu Lin Chen Chen, Shuai Mu, and Jinyang Li. Giza: Erasure coding objects across global data centers. In *Proceedings of the 2017 USENIX Annual Technical Conference (USENIX ATC '17)*, pages 539–551, 2017.

8 Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's Highly Available Key-value Store. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 205–220, New York, NY, USA, 2007. ACM. `doi:10.1145/1294261.1294281`.

9 A. G. Dimakis, K. Ramchandran, Y. Wu, and C. Suh. A survey on network codes for distributed storage. *Proceedings of the IEEE*, 99(3):476–489, 2011.

10 Partha Dutta, Rachid Guerraoui, and Ron R. Levy. Optimistic Erasure-Coded Distributed Storage. In *DISC '08: Proceedings of the 22nd international symposium on Distributed Computing*, pages 182–196, Berlin, Heidelberg, 2008. Springer-Verlag. `doi:10.1007/978-3-540-87779-0_13`.

11 G.R. Goodson, J.J. Wylie, G.R. Ganger, and M.K. Reiter. In *Dependable Systems and Networks, 2004 International Conference on*. `doi:10.1109/DSN.2004.1311884`.

12 James Hendricks, Gregory R Ganger, and Michael K Reiter. Low-overhead byzantine fault-tolerant storage. *ACM SIGOPS Operating Systems Review*, 41(6):73–86, 2007.

13 W. C. Huffman and V. Pless. *Fundamentals of error-correcting codes*. Cambridge university press, 2003.

14 K. M. Konwar, N. Prakash, E. Kantor, N. Lynch, M. Médard, and A. A. Schwarzmann. Storage-Optimized Data-Atomic Algorithms for Handling Erasures and Errors in Distributed Storage Systems. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 720–729, May 2016.

15 Kishori M Konwar, N Prakash, Nancy Lynch, and Muriel Médard. RADON: Repairable atomic data object in networks. In *The International Conference on Distributed Systems (OPODIS)*, 2016.

16 Leslie Lamport. On Interprocess Communication, Part I: Basic Formalism. *Distributed Computing*, 1(2):77–85, 1986.

17 Leslie Lamport. Fast Paxos. *Distributed Computing*, 19:79–103, October 2006.

18 N Nicolaou, V Cadambe, N. Prakash, K.M. Konwar, M. Medard, and N Lynch. ARES: Adaptive, reconfigurable, erasure coded, atomic storage implementing a register in a dynamic distributed system. In *International Conf. on Distributed Computing Systems (ICDCS)*, 2019.

19 A. Spiegelman, Y. Cassuto, G. Chockler, and I. Keidar. Space Bounds for Reliable Storage: Fundamental Limits of Coding. In *Proceedings of the International Conference on Principles of Distributed Systems (OPODIS2015)*, 2015.

20 Heng Zhang, Mingkai Dong, and Haibo Chen. Efficient and Available In-memory KV-Store with Hybrid Erasure Coding and Replication. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, pages 167–180, 2016.