MIT Libraries | DSpace@MIT

MIT Open Access Articles

A Layered Architecture for Erasure-Coded Consistent Distributed Storage

The MIT Faculty has made this article openly available. ***Please share*** how this access benefits you. Your story matters.

Citation: Konwar, Kishori M., Prakash, N., Lynch, Nancy and Médard, Muriel. 2017. "A Layered Architecture for Erasure-Coded Consistent Distributed Storage."

As Published: 10.1145/3087801.3087832

Publisher: ACM

Persistent URL: https://hdl.handle.net/1721.1/137772

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

Terms of use: Creative Commons Attribution-Noncommercial-Share Alike

Massachusetts Institute of Technology

A Layered Architecture for Erasure-Coded Consistent Distributed Storage

Kishori M. Konwar
EECS, MIT
kishori@csail.mit.edu

N. Prakash
EECS, MIT
prakashn@mit.edu

Nancy Lynch
EECS, MIT
lynch@csail.mit.edu

Muriel Médard
EECS, MIT
medard@mit.edu

ABSTRACT

Motivated by emerging applications to the *edge computing* paradigm, we introduce a two-layer erasure-coded fault-tolerant distributed storage system offering atomic access for read and write operations. In edge computing, clients interact with an edge-layer of servers that is geographically near; the edge-layer in turn interacts with a back-end layer of servers. The edge-layer provides low latency access and temporary storage for client operations, and uses the back-end layer for persistent storage. Our algorithm, termed Layered Data Storage (LDS) algorithm, offers several features suitable for edge-computing systems, works under asynchronous message-passing environments, supports multiple readers and writers, and can tolerate $f_1 < n_1/2$ and $f_2 < n_2/3$ crash failures in the two layers having n_1 and n_2 servers, respectively. We use a class of erasure codes known as regenerating codes for storage of data in the back-end layer. The choice of regenerating codes, instead of popular choices like Reed-Solomon codes, not only optimizes the cost of back-end storage, but also helps in optimizing communication cost of read operations, when the value needs to be recreated all the way from the back-end. The two-layer architecture permits a modular implementation of atomicity and erasure-code protocols; the implementation of erasure-codes is mostly limited to interaction between the two layers. We prove liveness and atomicity of LDS, and also compute performance costs associated with read and write operations. In a system with $n_1 = \Theta(n_2), f_1 = \Theta(n_1), f_2 = \Theta(n_2)$, the write and read costs are respectively given by $\Theta(n_1)$ and $\Theta(1) + n_1\mathcal{I}(\delta > 0)$. Here δ is a parameter closely related to the number of write operations that are concurrent with the read operation, and $\mathcal{I}(\delta > 0)$ is 1 if $\delta > 0$, and 0 if $\delta = 0$. The cost of persistent storage in the back-end layer is $\Theta(1)$. The impact of temporary storage is minimally felt in a multi-object system running N independent instances of LDS, where only a small fraction of the objects undergo concurrent accesses at any point during the execution. For the multi-object system, we identify a condition on the rate of concurrent writes in the system such that the overall storage cost is dominated by that of persistent storage in the back-end layer, and is given by $\Theta(N)$.

CCS CONCEPTS

• **Mathematics of computing → Coding theory**; • **Information systems → Distributed storage**; • **Theory of computation → Distributed algorithms**;

KEYWORDS

object storage; asynchronous distributed network; edge computing; consistency; liveness; storage cost; communication cost; erasure codes; regenerating codes; fault tolerance

1 INTRODUCTION

We introduce a two-layer erasure-coded fault-tolerant distributed storage system offering atomic access [23] for read and write operations. Providing consistent access to stored data is a fundamental problem in distributed computing. The most desirable form of consistency is atomicity, which in simple terms, gives the users of the data service the impression that the various concurrent read and write operations take place sequentially. Our work is motivated by applications to decentralized *edge computing*, which is an emerging distributed computing paradigm where processing of data moves closer to the users instead of processing the entire data in distant data centers or cloud centers [4, 12, 24, 29]. Edge computing is considered to be a key enabler for Internet of Things. In this form of computing, the users or clients interact with servers in the edge of the network, which forms the first layer of servers. The edge servers in turn interact with a second layer servers in the back-end, which is either a distant data-center or a cloud center. Geographic proximity of edge servers to clients permits high speed operations between clients and the edge layer, whereas communication between the edge and the back-end layer is typically much slower [24]. Thus, it is desirable whenever possible to complete client operations via interaction only with the edge layer. The edge servers however are severely restricted in their total storage capacity. We envisage a system that handles millions of files, which we call objects; the edge servers clearly do not have the capacity to store all the objects for the entire duration of execution. In practice, at any given time, only a tiny fraction of all objects undergo concurrent accesses; in our system, the limited storage space in the edge layer acts as a temporary storage for those objects that are getting accessed. The

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

PODC '17, July 25-27, 2017, Washington, DC, USA
© 2017 Association for Computing Machinery.
ACM ISBN 978-1-4503-4992-5/17/07…$15.00
https://doi.org/10.1145/3087801.3087832

second layer of servers provide permanent storage for all the objects for the entire duration of execution. The servers in the first layer act as virtual clients of the second layer servers.

An important requirement in edge-computing systems is to reduce the cost of operation of the back-end layer, by making efficient use of the edge layer [29]. Communication between the two layers, and persistent storage in the second layer contribute to the cost of operation of the second layer. We address both these factors in our system design. The layered approach to implementing an atomic storage service carries the advantage that, during intervals of high concurrency from write operations on any one object, the edge layer can be used to *retain* the more recent versions of the object that are being (concurrently) written, while *filtering* out the outdated versions. The ability to avoid writing every version to the second layer decreases the overall write communication cost between the two layers. Our architecture also permits the edge layer to be configured as a *proxy cache* layer for objects that are frequently read, and thus avoids the need to read from the back-end layer for such objects.

In this work, we use a recent class of erasure codes known as *regenerating codes* [8] for storage of data in the back-end layer. From a storage cost view-point, these are as efficient as popular erasure codes like Reed-Solomon codes [27]. In our system, usage of regenerating codes, instead of Reed-Solomon codes, provides the extra advantage of reducing read communication cost when the object needs to be recreated from the coded data in the cloud layer. Specifically, we rely on class of regenerating codes known as *minimum bandwidth regenerating codes* for simultaneously optimizing read and storage costs.

While this may be the first work that explicitly uses regenerating codes for consistent data storage, the study of erasure codes—like Reed-Solomon codes—in implementations of consistent distributed storage, is an active area of research by itself [6, 7, 11, 17, 30]. In the commonly used single-layer storage systems, for several regimes of operation, cost metrics of Reed-Solomon-code based implementations [1, 6, 11, 17] outperform those of replication based implementations [3]. In comparison with single layer systems, the layered architecture naturally permits a layering of the protocols needed to implement atomicity, and erasure code in the cloud layer. The protocols needed to implement atomicity are largely limited to interactions between the clients and the edge servers, while those needed to implement the erasure code are largely limited to interactions between the edge and cloud servers. From an engineering viewpoint, the modularity of our implementation makes it suitable even for situations that does not necessarily demand a two-layer system.

1.1 Our Algorithm for the Two-Layer System

We propose the *Layered Distributed Storage* (LDS) algorithm for implementing a multi-writer, multi-reader atomic storage service over a two-layer asynchronous network. The algorithm is designed to address the various requirements described above for edge computing systems. A write operation completes after writing the object value to the first layer; it does not wait for the first layer to store the corresponding coded data in the second layer. For a read operation, concurrency with write operations increases the chance of it being served directly from the first layer; otherwise, servers in the first layer *regenerate* coded data from the second layer, which are then relayed to the reader. Servers in the first layer interact with those of second layer via the well defined actions (which we call as *internal operations*) *write-to-L2* and *regenerate-from-L2* for implementing the regenerating code in the second layer. The algorithm is designed to tolerate $f_1 < n_1/2$ and $f_2 < n_2/3$ crash failures in the first and second layers, having n_1 and n_2 servers, respectively. We prove liveness and atomicity properties of the algorithm, and also calculate various performance costs. In a system with $n_1 = \Theta(n_2)$, $f_1 = \Theta(n_1)$, $f_2 = \Theta(n_2)$, the write and read costs are respectively given by $\Theta(n_1)$ and $\Theta(1) + n_1\mathcal{I}(\delta > 0)$. Here δ is a parameter closely related to the number of write or internal *write-to-L2* operations that are concurrent with the read operation, and $\mathcal{I}(\delta > 0)$ is 1 if $\delta > 0$, and 0 if $\delta = 0$.. Our ability to reduce the read cost to $\Theta(1)$, when $\delta = 0$ comes from the usage of minimum bandwidth regenerating (MBR) codes (see Section 2). In order to ascertain the contribution of temporary storage cost to the overall storage cost, we carry out a multi-object (say N) analysis, where each of the N objects is implemented by an independent instance of the LDS algorithm. The multi-object analysis assumes bounded latency for point-to-point channels. We identify conditions on the total number of concurrent write operations per unit time, such that the permanent storage cost in the second layer dominates the temporary storage cost in the first layer, and is given by $\Theta(N)$. Further, we compute bounds on completion times of successful client operations, under bounded latency.

1.2 Related Work

Replication based algorithms for implementing atomic shared memory appears in [3], [13]. The model in [13] uses a two-layer system, with one layer dedicated exclusively for meta-data, and other layer for storage. The model is suitable when actual data is much larger than meta-data, and permits easy scalability of the storage layer. However, clients interact with servers in both layers, and thus is not directly comparable to our model, where clients only interact with the first layer. Both [3], [13] use quorums for implementing atomicity; variations of these algorithms appear in practical systems like Cassandra [21]. Replication based algorithms in single-layer systems, for dynamic settings appear in RAMBO [22], DynaStore [2]. Dynamic setting allow servers to leave and enter the system; these algorithms rely on reconfiguration of quorums. Erasure-code based implementations of consistent data storage in single layer systems appear in [1, 6, 11, 17, 30]. Bounds on the performance costs for erasure-code based implementations appear in [7, 30]. In [5], [10], [15], erasure codes are used in algorithms for implementing atomic memory in settings that tolerate Byzantine failures. In [11, 14, 18], authors provide algorithms that permit repair of crashed servers (in a static setting), while implementing consistent storage. In the content of our work, it is of future interest to develop protocols for recovery of crashed servers in the second-layer, which implements permanent coded storage.

We rely on regenerating codes which were introduced in [8] with the motivation of enabling efficient repair of failed servers in distributed storage systems. For the same storage-overhead and resiliency, the communication cost for repair, termed repair-bandwidth, is substantially less than what is needed by popular codes like Reed-Solomon codes. There has been significant theoretical progress since the work of [8]; a survey appears in [9]. Several systems works show usefulness of these codes or their variations in practical systems for immutable data [20, 25, 28]. In this work, we cast internal read operations by virtual clients in the first layer as repair operations, and this enables us to reduce the overall read cost. We rely on code constructions from [26] for the existence of MBR codes needed in our work, and these codes offer *exact repair*. A different class of codes known as Random Linear Network Codes [16] permit implementation of regenerating codes via *functional repair*. These codes offer probabilistic guarantees, and permit near optimal operation of regenerating codes for any choice of operating point suggested by [8]. In the context of our work, it will be interesting to find out the probabilistic guarantees that can be obtained if we use RLNCs instead of the codes in [26].

System Model and definitions appear in Section 2. The pseudo code of the LDS algorithm, along with its description is presented in Section 3. In Section 4, we state several properties of the algorithm, which are tied together to prove its liveness and atomicity properties. Performance cost analysis appears in Section 5. Our conclusions appear in Section 6. Proofs of various claims can be found in an extended version of the paper [19].

2 SYSTEM MODEL AND DEFINITIONS

Model of Computation. We assume a distributed storage system consisting of asynchronous processes of three types: writers ($\mathcal{W}$), readers ($\mathcal{R}$) and servers ($\mathcal{S}$). The servers are organized into two logical layers $\mathcal{L}_1$ and $\mathcal{L}_2$, with $\mathcal{L}_i$ consisting of n_i, $i = 1, 2$ servers. Each process has a unique id, and the ids are totally ordered. Client (reader/writer) interactions are limited to servers in $\mathcal{L}_1$, the servers in $\mathcal{L}_1$ in turn interact with servers in $\mathcal{L}_2$. Further, the servers in $\mathcal{L}_1$ and $\mathcal{L}_2$ are denoted by $\{s_1, s_2, \ldots, s_{n_1}\}$ and $\{s_{n_1+1}, s_{n_1+2}, \ldots, s_{n_1+n_2}\}$, respectively. We assume the clients to be well-formed, i.e., a client issues a new operation only after completion of its previous operation, if any. The $\mathcal{L}_1$-$\mathcal{L}_2$ interaction happens via the well defined actions *write-to-L2* and *regenerate-from-L2*. We will refer to these actions as internal operations initiated by the servers in $\mathcal{L}_1$. We assume a crash failure model for processes. Once a process crashes, it does not execute any further steps for the rest of the execution. The LDS algorithm is designed to tolerate f_i crash failures in layer $\mathcal{L}_i$, $i = 1, 2$, where $f_1 < n_1/2$ and $f_2 < n_2/3$. Any number of readers and writers can crash during the execution. Communication is modeled via reliable point-to-point links between any two processes. This means that as long as the destination process is non-faulty, any message sent on the link is guaranteed to eventually reach the destination process. The model allows the sender process to fail after placing the message in the channel; message-delivery depends only on whether the destination is non-faulty.

Liveness and Atomicity. We implement one object, say x, via the LDS algorithm supporting read/write operations. For multiple objects, we simply run multiple instances of the LDS algorithm. The object value v come from the set $\mathcal{V}$; initially v is set to a distinguished value v_0 ($\in \mathcal{V}$). Reader r requests a read operation on object x. Similarly, a write operation is requested by a writer w. Each operation at a non-faulty client begins with an *invocation step* and terminates with a *response step*. An operation π is *incomplete* in an execution when the invocation step of π does not have the associated response step; otherwise we say that π is *complete*. In an execution, we say that an operation (read or write) π_1 *precedes* another operation π_2, if the response step for π_1 precedes the invocation step of π_2. Two operations are *concurrent* if neither precedes the other.

By liveness, we mean that during any well-formed execution of the algorithm, any read or write operation initiated by a non-faulty reader or writer completes, despite the crash failure of any other client and up to f_1 server crashes in $\mathcal{L}_1$, and up to f_2 server crashes in $\mathcal{L}_2$. By atomicity of an execution, we mean that the read and write operations in the execution can be arranged in a sequential order that is consistent with the order of invocations and responses. We refer to [23] for formal definition of atomicity. We use the sufficient condition presented in Lemma 13.16 of [23] to prove atomicity of LDS.

Regenerating Codes. We introduce the framework as in [8], and then see its usage in our work. In the regenerating-code framework, a file $\mathcal{F}$ of size B symbols is encoded and stored across n servers such that each server stores α symbols. The symbols are assumed to be drawn from a finite field $\mathbb{F}_q$, for some q. The content from any k servers ($k\alpha$ symbols) can be used to decode the original file $\mathcal{F}$. For repair of a failed server, the replacement server contacts any subset of $d \geq k$ surviving servers in the system, and downloads β symbols from each of the d servers. The β symbols from a *helper* server is possibly a function of the α symbols in the server. The parameters of the code, say C, shall be denoted as $\{(n, k, d)(\alpha, \beta)\}$. It was shown in [8] that the file-size B is upper bounded by $B \leq \sum_{i=0}^{k-1} \min(\alpha, (d-i)\beta)$. Two extreme points of operation correspond to the minimum storage overhead (MSR) operating point, with $B = k\alpha$ and minimum repair bandwidth (MBR) operating point, with $\alpha = d\beta$. In this work, we use codes at the MBR operating point. The file-size at the MBR point is give by $B_{MBR} = \sum_{i=0}^{k-1}(d-i)\beta$. We also focus on exact-repair codes, meaning that the content of a replacement server after repair is identical to what was stored in the server before crash failure (the framework permits *functional repair* [8] which we do not consider). Code constructions for any set of parameters at the MBR point appear in [26], and we rely on this work for existence of codes. In this work, the file $\mathcal{F}$ corresponds to the object value v that is written.

In this work, we use an $\{(n = n_1 + n_2, k, d)(\alpha, \beta)\}$ MBR code C. The parameters k and d are such that $n_1 = 2f_1 + k$ and $n_2 = 2f_2 + d$. We define two additional codes C_1 and C_2 that are derived from the code C. The code C_1 is obtained by restricting attention to the first n_1 coded symbols of C, while the code C_2 is obtained by restricting attention to the last n_2 coded symbols of C. Thus if $[c_1\ c_2\ \ldots c_{n_1}\ c_{n_1+1}\ \ldots c_{n_1+n_2}], c_i \in \mathbb{F}_q^{\alpha}$ denotes a codeword of C, the vectors $[c_1\ c_2\ \ldots c_{n_1}]$ and $[c_{n_1+1}\ \ldots c_{n_1+n_2}]$ will be codewords of C_1 and C_2, respectively. We associate the code symbol c_i with server s_i, $1 \leq i \leq n_1 + n2$.

The usage of these three codes is as follows. Each server in $\mathcal{L}_1$, having access to the object value v (at an appropriate point in the execution) encodes v using code C_2 and sends coded data c_{n_1+i} to server s_{n_1+i} in $\mathcal{L}_2, 1 \leq i \leq n_2$. During a read operation, a server say s_j in $\mathcal{L}_1$ can potentially reconstruct the coded data c_j using content from $\mathcal{L}_2$. Here we think of c_j as part of the code C, and c_j gets reconstructed via a repair procedure (invoked by server s_j in $\mathcal{L}_1$) where the d helper servers belong to $\mathcal{L}_2$. By operating at the MBR point, we minimize the cost that need by the server s_j to reconstruct c_j. Finally, in our algorithm, we permit the possibility that the reader receives k coded data elements from k servers in $\mathcal{L}_1$, during a read operation. In this case, the reader uses the code C_1 to attempt decoding the object value v.

An important property of the MBR code construction in [26], which is needed in our algorithm, is the fact the a helper server only needs to know the index of the failed server, while computing the helper data, and does not need to know the indices of the other $d - 1$ helpers whose helper data will be used in repair. Not all regenerating code constructions, including those of MBR codes, have this property that we need. In our work, a server $s_j \in \mathcal{L}_1$ requests for help from all servers in $\mathcal{L}_2$, and does not know a priori, the subset of d servers that will form the helper servers. As we shall see in the algorithm, the server s_j simply relies on the first d responses that it receives, and considers these as the helper data for repair. In this case, it is crucial that any server in $\mathcal{L}_2$ that computes its β symbols does so without any assumption on the specific set of d servers in $\mathcal{L}_2$ that will eventually form the helper servers for the repair operation.

Storage and Communication Costs. The communication cost associated with a read or write operation is the (worst-case) size of the total data that gets transmitted in the messages sent as part of the operation. While calculating write-cost, we also include costs due to internal *write-to-L2* operations initiated as a result of the write, even though these internal *write-to-L2* operations do not influence the termination point of the write operation. The storage cost at any point in the execution is the worst-case total amount of data that is stored in the servers in $\mathcal{L}_1$ and $\mathcal{L}_2$. The total data in $\mathcal{L}_1$ contributes to temporary storage cost, while that in $\mathcal{L}_2$ contributes to permanent storage cost. Costs contributed by meta-data (data for book keeping such as tags, counters, etc.) are ignored while ascertaining either storage or communication costs. Further the costs are normalized by the size of v; in other words, costs are expressed as though size of v is 1 unit.

3 *LDS* ALGORITHM

In this section, we present the *LDS* algorithm. The protocols for clients, servers in $\mathcal{L}_1$ and servers in $\mathcal{L}_2$ appear in Figs. 1, 2 and 3 respectively. Tags are used for version control of object values. A tag t is defined as a pair (z, w), where $z \in \mathbb{N}$ and $w \in \mathcal{W}$ denotes the ID of a writer. 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 the set $\mathcal{T}$.

Each server s in $\mathcal{L}_1$ maintains the following state variables: $a)$ a list $L \subseteq \mathcal{T} \times \mathcal{V}$, which forms a temporary storage for tag-value pairs received as part of write operations, $b)$ $\Gamma \subseteq \mathcal{R} \times \mathcal{T}$, which indicates the set of readers being currently served. The pair $(r, t_{req}) \in \Gamma$ indicates that the reader r requested for tag t_{req} during the read operation. $c)$ t_c: committed tag at the server, $d)$ K : a key-value set used by the server as part of internal *regenerate-from-L2* operations. The keys belong to $\mathcal{R}$, and values belong to $\mathcal{T} \times \mathcal{H}$. Here $\mathcal{H}$ denotes the set of all possible helper data corresponding to coded data elements $\{c_s(v), v \in \mathcal{V}\}$. Entries of $\mathcal{H}$ belong to $\mathbb{F}_q^{\beta}$. In addition to these, the server also maintains three counter variables for various operations. The state variable for a server in $\mathcal{L}_2$ simply consists of one (tag, coded-element) pair. For any server s, we use the notation $s.y$ to refer its state variable y. Further, we write $s.y|_T$ to denote the value of $s.y$ at point T of the execution. Following the spirit of I/O automata [23], an execution fragment of the algorithm is simply an alternating sequence of (the collection of all) states and actions. By an action, we mean a block of code executed by any one process without waiting for further external inputs.

In our algorithm, we use a *broadcast* primitive for certain meta-data message delivery. The primitive has the property that if the message is consumed by any one server in $\mathcal{L}_1$, the same message is eventually consumed by every non-faulty server in $\mathcal{L}_1$. An implementation of the primitive, on top of reliable communication channels (as in our model), can be found in [17]. In this implementation, the idea is that the process that invokes the broadcast protocol first sends, via point-to-point channels, the message to a fixed set S_{f_1+1} of $f_1 + 1$ servers in $\mathcal{L}_1$. Each of these servers, upon reception of the message for first time, sends it to all the servers in $\mathcal{L}_1$, before consuming the message itself. The primitive helps in the scenario when the process that invokes the broadcast protocol crashes before sending the message to all servers.

3.1 Write Operation

The write operation has two phases, and aims to temporarily store the object value v in $\mathcal{L}_1$ such that up to f_1 failures in $\mathcal{L}_1$ does not result in loss of the value. During the first phase *get-tag*, the writer w determines the new tag for the value to be written. To this end, the writer queries all servers in $\mathcal{L}_1$ for maximum tags, and awaits responses from $f_1 + k$ servers in $\mathcal{L}_1$. Each server that gets the *get-tag* request responds with the maximum tag present in the list L, i.e. $\max_{(t,*)\in L} t$. The writer picks the maximum tag, say t, from among the responses, and creates a new and higher tag $t_w = tag(\pi)$.

In the second phase *put-data*, the writer sends the new (tag, value) pair to all severs in $\mathcal{L}_1$, and awaits acknowledgments from $f_1 + k$ servers in $\mathcal{L}_1$. A server that receives the new (tag, value) pair (t_{in}, v_{in}) as a first step uses the *broadcast* primitive to send a data-reception message to all servers in $\mathcal{L}_1$. Note that this message contains only meta-data information and not the actual value. Each server $s \in \mathcal{L}_1$ maintains a committed tag variable t_c to indicate the highest tag that the server either finished writing or is currently writing to $\mathcal{L}_2$. After sending the broadcast message, the server adds the pair (t_{in}, v_{in}) to its temporary storage list L if $t_{in} > t_c$. If $t_{in} < t_c$, the server simply sends an acknowledgment back to the writer, and completes its participation in the ongoing write operation. If the server adds the pair (t_{in}, v_{in}) to L, it waits to hear the *broadcast* message regarding the ongoing write operation from at least $f_1 + k$ servers before sending acknowledgment to the writer w. We implement this via the *broadcast-resp* action. It may be noted

Fig. 1 The *LDS* algorithm for a writer $w \in \mathcal{W}$ and reader $r \in \mathcal{R}$.

```
LDS steps at a writer w:
get-tag:
    Send QUERY-TAG to servers in $\mathcal{L}_1$
    Wait for responses from $f_1 + k$ servers, and select max tag $t$.
put-data:
    Create new tag $t_w = (t.z + 1, w)$.
    Send (PUT-DATA, $(t_w, v)$) to servers in $\mathcal{L}_1$
    Wait for responses from $f_1 + k$ servers in $\mathcal{L}_1$, and terminate
```

```
LDS steps at a reader r:
get-commited-tag:
    Send QUERY-COMM-TAG to servers in $\mathcal{L}_1$
    Await $f_1 + k$ responses, and select max tag $t_{req}$
get-data:
    Send (QUERY-DATA, $t_{req}$) to servers in $\mathcal{L}_1$
    Await responses from $f_1 + k$ servers such that at least one of
    them is a (tag, value) pair, or at least $k$ of them are (tag, coded-
    element) pairs corresponding to some common tag. In the latter
    case, decode corresponding value using code $C_1$. Select the $(t_r, v_r)$
    pair corresponding to the highest tag, from the available (tag,
    value) pairs.
put-tag:
    Send (PUT-TAG, $t_r$) to servers in $\mathcal{L}_1$
    Await responses from $f_1 + k$ servers in $\mathcal{L}_1$. Return $v_r$
```

that a server sends acknowledgment to writer via the *broadcast-resp* action only if it had added the (tag, value) pair to L during the *put-data-resp* action.

3.2 Additional Steps during *broadcast-resp*

The server performs a few additional steps during the *broadcast-resp* action, and these aid ongoing read operations, garbage collection of the temporary storage, and also help to offload the coded elements to $\mathcal{L}_2$. The execution of these additional steps do not affect the termination point of the write operation. We explain these steps next.

Update committed tag t_c: The server checks if t_{in} is greater than the committed tag t_c, and if so updates the t_c to t_{in}. We note that even though the server added (t_{in}, v_{in}) to L only after checking $t_{in} > t_c$, the committed tag might have advanced due to concurrent write operations corresponding to higher tags and thus it is possible that $t_{in} < t_c$ when the server does the check. Also, if $t_{in} > t_c$, it cab be shown that $(t_{in}, v_{in}) \in L$. In other words the value v_{in} has not been garbage collected yet from the temporary storage L. We explain the mechanism of garbage collection shortly.

Serve outstanding read requests: The server sends the pair (t_{in}, v_{in}) to any outstanding read request whose requested tag $t_{req} \leq t_{in}$. In this case, the server also considers the read operation as being served and will not send any further message to the corresponding reader. We note this is only one of the various possibilities to serve a reader. Read operation is discussed in detail later.

Garbage collection of older tags: Garbage collection happens in two ways in our algorithm. We explain one of those here, the other will be explained as part of the description of the internal *write-to-L2* operation. The server replaces any (tag, value) pair (t, v) in the list L corresponding to $t < t_c = t_{in}$ with $(t, \perp)$, and thus removing the value associated with tags which are less than the committed tag. The combination of our method of updating the committed tag and garbage collection described here ensures that during intervals of concurrency from multiple write operations, we only offload the more recent (tag, value) pairs (after the tags get committed) to the back-end layer. Also the garbage collection described here eliminates values corresponding to older write operations which might have failed during the execution (and thus, which will not get a chance to get garbage collected via the second option which we describe below).

Internal write-to-L2 operation: The server computes the coded elements $\{c_{n_1+1}, \ldots, c_{n_1+n_2}\}$ corresponding to value v_{in} and sends (t_{in}, c_{n_1+i}) to server s_{n_1+i} in $\mathcal{L}_2$, $1 \leq i \leq n_2$. In our algorithm, each server in $\mathcal{L}_2$ stores coded data corresponding to exactly one tag at any point during the execution. A server in $\mathcal{L}_2$ that receives (tag, coded-element) pair (t, c) as part of an internal *write-to-L2* operation replaces the local pair (tag, coded-element) pair (t_ℓ, c_ℓ) with the incoming one if $t > t_\ell$. The *write-to-L2* operation initiated by server $s \in \mathcal{L}_1$ terminates after it receives acknowledgments from $f_1 + d$ servers in $\mathcal{L}_2$. Before terminating, the server also garbage collects the pair (t_{in}, v_{in}) from its list.

A pictorial illustration of the events that occur as a result of an initiation of a *put-data* phase by a writer is shown in Fig. 4.

3.3 Read operation

The idea behind the read operation is that the reader gets served (tag, value) pairs from temporary storage in $\mathcal{L}_1$, if it overlaps with concurrent write or internal *write-to-L2* operations. If not, servers in $\mathcal{L}_1$ regenerate (tag, coded-element) pairs via *regenerate-from-L2* operations, which are then sent to the reader. In the latter case, the reader needs to decode the value v using the code C_1. A read operation consists of three phases. During the first phase *get-commited-tag*, the reader identifies the minimum tag, t_{req}, whose corresponding value it can return at the end of the operation. Towards this, the reader collects committed tags from $f_1 + k$ servers in

Fig. 2 The *LDS* algorithm for any server in $\mathcal{L}_1$.

```
  LDS state variables & steps at an $\mathcal{L}_1$ server, $s_j$:
State Variables:
$L \subseteq \mathcal{T} \times \mathcal{V}$, initially $\{(t_0, \perp)\}$
$\Gamma \subseteq \mathcal{R} \times \mathcal{T}$, initially empty
$t_c \in \mathcal{T}$ initially $t_c = t_0$
$commitCounter[t] : t \in \mathcal{T}$, initially 0 $\forall t \in \mathcal{T}$
$readCounter[r]$: $r \in \mathcal{R}$, initially 0 $\forall r \in \mathcal{R}$
$writeCounter[t]$: $t \in \mathcal{T}$, initially 0 $\forall t \in \mathcal{T}$
$K$ : key-value set; keys from $\mathcal{R}$, values from $\mathcal{T} \times \mathcal{H}$

get-tag-resp (QUERY-TAG) from $w \in \mathcal{W}$:
  send max$\{t : (t, *) \in L\}$ to $w$

put-data-resp (PUT-DATA, $(t_{in}, v_{in})$) received:
  broadcast(COMMIT-TAG, $t_{in}$) to $\mathcal{L}_1$
  if $t_{in} > t_c$ then
    $L \leftarrow L \cup \{(t_{in}, v_{in})\}$
  else
    send ACK to writer $w$ of tag $t_{in}$

broadcast-resp (COMMIT-TAG, $t_{in}$) received:
  $commitCounter[t_{in}] \leftarrow commitCounter[t_{in}] + 1$
  if $(t_{in}, *) \in L \wedge$ $commitCounter[t_{in}] \geq f_1 + k$ then
    send ACK to writer $w$ of tag $t_{in}$
    if $t_{in} > t_c$ then
      $t_c \leftarrow t_{in}$
      for each $\gamma \in \Gamma$ such that $t_c \geq \gamma.t_{req}$,
        send $(t_{in}, v_{in})$ to reader $\gamma.r$
        $\Gamma \leftarrow \Gamma - \{\gamma\}$
      for each $(t, *) \in L$ s.t. $t < t_c$ //delete older value
        $L \leftarrow L - \{(t, *)\} \cup \{(t, \perp)\}$
      initiate write-to-L2($t_{in}, v_{in}$) // write $v_{in}$ to $\mathcal{L}_2$

write-to-L2($t_{in}, v_{in}$):
  for each $s_{n_1+i} \in \mathcal{L}_2$
    Compute coded element $c_{n_1+i}$ for value $v$
    send (WRITE-CODE-ELEM, $(t_{in}, c_{n_1+i})$) to $s$

write-to-L2-complete (ACK-CODE-ELEM, $t$) received:
  $writeCounter[t] \leftarrow writeCounter[t] + 1$
  if $writeCounter[t] = n_2 - f_2$ then
    $L \leftarrow L - \{(t, *)\} \cup \{(t, \perp)\}$

get-commited-tag-resp (QUERY-COMM-TAG) from $r \in \mathcal{R}$:
  send $t_c$ to $r$

get-data-resp (QUERY-DATA, $t_{req}$) from $r \in \mathcal{R}$:
  if $(t_{req}, v_{req}) \in L$ then
    send $(t_{req}, v_{req})$ to reader $r$
  else
    if $t_c > t_{req} \wedge (t_c, v_c) \in L$ then
      send $(t_c, v_c)$ to reader $r$
    else
      $\Gamma \leftarrow \Gamma \cup \{(r, t_{req})\}$
      initiate regenerate-from-L2($r$)

regenerate-from-L2($r$):
  for each $s \in \mathcal{L}_2$
      send (QUERY-CODE-ELEM, $r$) to $s$

regenerate-from-L2-complete(SEND-HELPER-ELEM, $(r, t, h_{n_1+i,j})$) recv:
  $readCounter[r] \leftarrow readCounter[r] + 1$
  $K[r] \leftarrow K[r] \cup \{(t, h_{n_1+i,j})\}$
  if $readerCounter[r] = n_2 - f_2 = f_2 + d$ then
    $(\widehat{t}, \widehat{c_j}) \leftarrow$ regenerate highest possible tag using $K[r]$
    //$(\perp, \perp)$ if failed to regenerate any tag
    clear $K[r]$
    if $\widehat{c_j} \neq \perp \wedge \widehat{t} \geq \gamma.t_{req}$ then // where $\gamma = (r, t_{req})$
      send $(\widehat{t}, \widehat{c_j})$ to $r$
    else
      send $(\perp, \perp)$ to $r$

put-tag-resp (PUT-TAG, $(t_{in})$) received from $r \in \mathcal{R}$:
  $\Gamma \leftarrow \Gamma - \{\gamma'\}$ // $\gamma' = (r, t_{req})$
  if $t_{in} > t_c$ then
    $t_c \leftarrow t_{in}$
    if $(t_c, v_c) \in L$ then
      for each $\gamma \in \Gamma$ s. t. $t_c \geq \gamma.t_{req}$
          send $(t_c, v_c)$ to reader $\gamma.r$,
          $\Gamma \leftarrow \Gamma - \{\gamma\}$.
      initiate write-to-L2($t_{in}, v_{in}$)
    else
      $L \leftarrow L \cup \{(t_c, \perp)\}$
      $\bar{t} \leftarrow$ max$\{t : t < t_c \wedge (t, v) \in L\}$
      // $\bar{t} = \perp$, if none exists
      for each $\gamma \in \Gamma$ such that $\bar{t} \geq \gamma.t_{req}$,
        send $(\bar{t}, \bar{v})$ to reader $\gamma.r$
        $\Gamma \leftarrow \Gamma - \{\gamma\}$
    for each $(t, *) \in L$ s.t. $t < t_c$
        $L \leftarrow L - \{(t, *)\} \cup \{(t, \perp)\}$
  send ACK to $r$
```

$\mathcal{L}_1$, and computes the requested tag t_{req} as the maximum of these $f_1 + k$ committed tags.

During the second *get-data* phase, the reader sends t_{req} to all the servers in $\mathcal{L}_1$, awaits responses from $f_1 + k$ distinct servers such that 1) at least one of the responses contains a tag-value pair, say (t_r, v_r), $t_r \geq t_{req}$ or 2) at least k of the responses contain coded elements corresponding to some fixed tag, say t_r such that $t_r \geq t_{req}$. In the latter case, the reader uses the code C_2 to decode the value v_r corresponding to tag t_r. If more than one candidate is found for the (tag, value) pair that can be returned, the reader picks the pair

Fig. 3 The *LDS* algorithm for any server in $\mathcal{L}_2$.

```
    LDS state variables & steps at an $\mathcal{L}_2$ server $s_{n_1+i}$:
State Variables:
$(t, c) \in \mathcal{T} \times \mathbb{F}_q^{\alpha}$, initially $(t_0, c_0)$

    write-to-L2-resp (WRITE-CODE-ELEM, $(t_{in}, c_{in})$) from $s_j$:
  if $t_{in} > t$ then
      $(t, c) \leftarrow (t_{in}, c_{in})$
    send (ACK-CODE-ELEM, $t_{in}$) to $s_j$

  regenerate-from-L2-resp(QUERY-CODE-ELEM, $r$) from $s_j$:
    Compute helper data $h_{n_1+i,j} \in \mathbb{F}_q^{\beta}$ for repairing $c_j$
    send (SEND-HELPER-ELEM, $(r, t, h_{n_1+i,j})$) to $s_j$
```

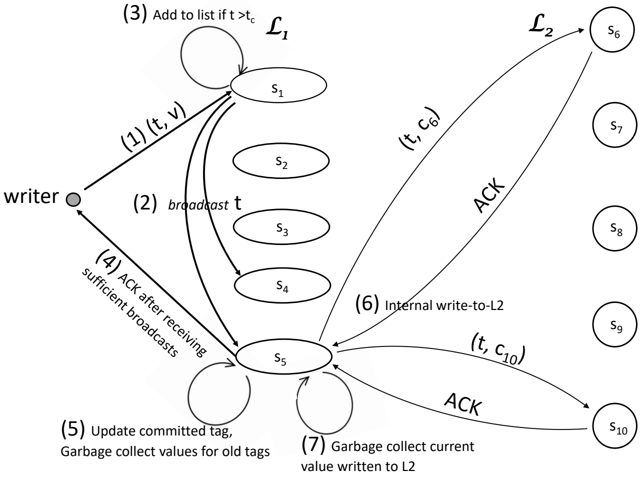
(3) Add to list if t >t_c
$\mathcal{L}_1$
$\mathcal{L}_2$
s_1
s_2
s_3
s_4
s_5
s_6
s_7
s_8
s_9
s_{10}
(1) (t, v)
writer
(2) broadcast t
(t, c_6)
ACK
(4) ACK after receiving sufficient broadcasts
(6) Internal write-to-L2
(t, c_{10})
ACK
(5) Update committed tag, Garbage collect values for old tags
(7) Garbage collect current value written to L2

Figure 4: An illustration of the events that occur as a result of an initiation of a *put-data* phase by a writer. The illustration is only representative, and does not cover all possible executions. In this illustration, the steps occur in the order (1), (2) **and so on. Steps** (5), (6), (7) **occur after sending ACK to the writer, and hence does not affect the termination point of the write operation.**

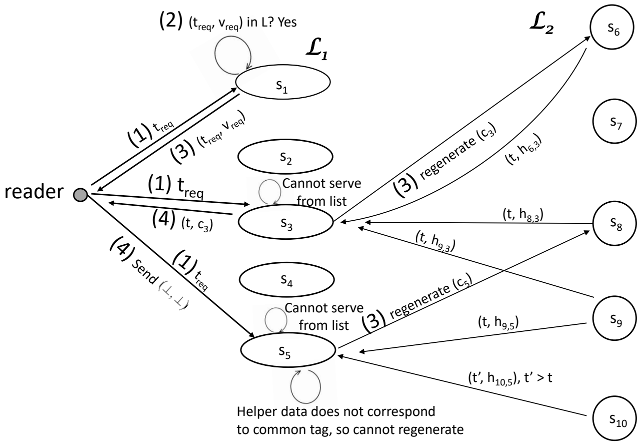

Figure 5: An illustration of the events that occur as a result of an initiation of a *get-data* phase by a reader. Once again, the illustration is only representative, and does not cover all possible executions. Server s_1 responds to the reader using content from local list, server s_3 regenerates successfully, server s_5 fails to regenerate successfully.

corresponding to the maximum tag. From the servers' point of view, a server $s \in \mathcal{L}_1$ upon reception of the *get-data* request checks if either (t_{req}, v_{req}) or $(t_c, v_c), t_c > t_{req}$ is in its list; in this case, s responds immediately to the reader with the corresponding pair. Otherwise, s adds the reader to its list Γ of outstanding readers, initiates an internal *regenerate-from-L2* operation in which s attempts to regenerate a tag-coded data element pair $(t', c'_s), t' \geq t_{req}$ via a repair process taking help from servers in $\mathcal{L}_2$. Towards this, the server s contacts all servers in $\mathcal{L}_2$, and each server $\bar{s} \in \mathcal{L}_2$, using its state variable $(\bar{t}, \bar{c}_s)$, computes and sends the helper data $(\bar{t}, \bar{h}_s)$ back to the server s. We note that the MBR code that we use (from [26]) has the property that $\bar{h}_s$ can be uniquely computed given $\bar{c}_s$ and the id of the server s which invokes the *regenerate-from-L2* operation. The server s waits for $d + f_2$ responses, and if at least d responses correspond to a common tag, say $t', t' \geq t_{req}$, regenerates the pair (t', c'), and sends (t', c') back to the reader. It is possible that regeneration fails, and this can happen in two ways: 1) the server s regenerates a pair (t'', c''), however $t'' < t_{req}$, 2) among the $f_2 + d$ responses received by the server s, there is no subset of d responses corresponding to a common tag. In our algorithm, if regeneration from $\mathcal{L}_2$ fails, the server s simply sends $(\perp, \perp)$ back to the reader. The reader interprets the response $(\perp, \perp)$ as a sign of unsuccessful regeneration. We note that irrespective of if regeneration succeeds or not, the server does not remove the reader from its list of outstanding readers. In the algorithm, we allow the server s to respond to a registered reader with a tag-value pair, during the *broadcast-resp* action as explained earlier. It is possible that while the server awaits responses from $\mathcal{L}_2$ towards regeneration, a new tag t gets committed by s via the *broadcast-resp* action; in this case, if $t \geq t_c$, server s sends (t, v) to r, and also unregisters r from its outstanding reader list. A pictorial illustration of the events that occur as a result of an initiation of a *get-data* phase by a writer is shown in Fig. 5.

In the third phase *put-tag*, the reader *writes-back* tag t_r corresponding to v_r, and ensures that at least $f_1 + k$ servers in $\mathcal{L}_1$ have their committed tags at least as high as t_r, before the read operation completes. However, the value v_r is not written back in this third phase, and this is important to decrease the read cost. When a server $s \in \mathcal{L}_1$ receives the *put-tag* request for tag t_r, it checks if (t_r, v_t) is in its list. In this case, the server thinks of the *put-tag* request simply as a proxy for having encountered the event $commitCounter[t_r] = f_1 + k$ during the *broadcast-resp* action, and carries out all the steps that it would have done during the the *broadcast-resp* action (except sending an ACK to the writer). However, if the server sees the tag t_r for the first time during the execution, it still updates its committed tag to t_r, and simply adds $(t_r, \perp)$ to its list. Further, the server carries out a sequence of steps similar to the case when $(t_r, v_r) \in L$ (except initiating *write-to-L2*) before sending ACK to reader. The third phase also helps in unregistering the reader from the servers in $\mathcal{L}_1$.

4 PROPERTIES OF THE ALGORITHM

We state several interesting properties of the LDS algorithm. These will be found useful while proving the liveness and atomicity properties of the algorithm. We let $S_a \subset \mathcal{L}_1, |S_a| = f_1 + k$ to denote the set of $f_1 + k$ servers in $\mathcal{L}_1$ that never crash fail during the execution. The following lemmas are only applicable to servers that are alive at the concerned point(s) of execution appearing in the lemmas.

For every operation π in Π corresponding to a non-faulty reader or writer, we associate a $(tag, value)$ pair that we denote as $(tag(\pi)$, $value(\pi))$. For a write operation π, we define the $(tag(\pi), value(\pi))$ pair as the message (t_w, v) which the writer sends in the *put-data* phase. If π is a read, we define the $(tag(\pi), value(\pi))$ pair as (t_r, v) where v is the value that gets returned, and t_r is the associated tag. We also define tags, in a similar manner for those failed write operations that at least managed to complete the first round of the write operation. This is simply the tag t_w that the writer would use in *put-data* phase, if it were alive. In our discussion, we ignore writes that failed before completion of the first round.

For any two points T_1, T_2 in an execution of LDS, we say $T_1 < T_2$ if T_1 occurs earlier than T_2 in the execution. The following three lemmas describe properties of committed tag t_c, and tags in the list.

Lemma 4.1 (**Monotonicity of committed tag**). *Consider any two points T_1 and T_2 in an execution of LDS, such that $T_1 < T_2$. Then, for any server $s \in \mathcal{L}_1$, $s.tc|_{T_1} \leq s.tc|_{T_2}$.*

Lemma 4.2 (**Garbage collection of older tags**). *For any server $s \in \mathcal{L}_1$, at any point T in an execution of LDS, if $(t, v) \in s.L|_T$, we have $t \geq s.t_c|_T$.*

Lemma 4.3 (**Persistence of tags corresponding to completed operations**). *Consider any successful write or read operation ϕ in an execution of LDS, and let T be any point in the execution after ϕ completes. For any set S' of $f_1 + k$ servers in $\mathcal{L}_1$ that are non-faulty at T, there exists $s \in S'$ such that $s.t_c|_T \geq tag(\phi)$ and $\max\{t : (t, *) \in s.L|_T\} \geq tag(\phi)$.*

The following lemma shows that an internal *regenerate-from-L2* operation respects previously completed internal *write-to-L2* operations. Our assumption that $f_2 < n_2/3$ is used in the proof of this lemma.

Lemma 4.4 (**Consistency of Internal Reads with respect to Internal Writes**). *Let σ_2 denote a successful internal write-to-L2(t, v) operation executed by some server in $\mathcal{L}_1$. Next, consider an internal regenerate-from-L2 operation π_2, initiated after the completion of σ_2, by a server $s \in \mathcal{L}_1$ such that a tag-coded-element pair, say (t', c') was successfully regenerated by the server s. Then, $t' \geq t$; i.e., the regenerated tag is at least as high as what was written before the read started.*

The following three lemmas are central to prove the liveness of read operations.

Lemma 4.5 (**If internal *regenerate-from-L2* operation fails**). *Consider an internal regenerate-from-L2 operation initiated at point T of the execution by a server $s_1 \in \mathcal{L}_1$ such that s_1 failed to regenerate any tag-coded-element pair based on the responses. Then, there exists a point $\widetilde{T} > T$ in the execution such that the following statement is true: There exists a subset S_b of S_a such that $|S_b| = k$, and $\forall s' \in S_b$ $(\widetilde{t}, \widetilde{v}) \in s'.L|_{\widetilde{T}}$, where $\widetilde{t} = \max_{s \in \mathcal{L}_1} s.t_c|_{\widetilde{T}}$.*

Lemma 4.6 (**If internal *regenerate-from-L2* operation regenerates a tag older than the request tag**). *Consider an internal regenerate-from-L2 operation initiated at point T of the execution by a server $s_1 \in \mathcal{L}_1$ such that s_1 only manages to regenerate (t, c) based on the responses, where $t < t_{req}$. Here t_{req} is the tag sent by the associated reader during the get-data phase. Then, there exists a point $\widetilde{T} > T$ in the execution such that the following statement is true: There exists a subset S_b of S_a such that $|S_b| = k$, and $\forall s' \in S_b$ $(\widetilde{t}, \widetilde{v}) \in s'.L|_{\widetilde{T}}$, where $\widetilde{t} = \max_{s \in \mathcal{L}_1} s.t_c|_{\widetilde{T}}$.*

Lemma 4.7 (**If two Internal *regenerate-from-L2* operations regenerate differing tags**). *Consider internal regenerate-from-L2 operations initiated at points T and T' of the execution, respectively by servers s and s' in $\mathcal{L}_1$. Suppose that s and s' regenerate tags t and t' such that $t < t'$. Then, there exists a point $\widetilde{T} > T$ in the execution such that the following statement is true: There exists a subset S_b of S_a such that $|S_b| = k$, and $\forall s' \in S_b$ $(\widetilde{t}, \widetilde{v}) \in s'.L|_{\widetilde{T}}$, where $\widetilde{t} = \max_{s \in \mathcal{L}_1} s.t_c|_{\widetilde{T}}$.*

Theorem 4.8 (**Liveness**). *Consider any well-formed execution of the LDS algorithm, where at most $f_1 < n_1/2$ and $f_2 < n_2/3$ servers crash fail in layers $\mathcal{L}_1$ and $\mathcal{L}_2$, respectively. Then every operation associated with a non-faulty client completes.*

Theorem 4.9 (**Atomicity**). *Every well-formed execution of the LDS algorithm is atomic.*

5 COST COMPUTATION: STORAGE, COMMUNICATION AND LATENCY

In this section we discuss storage and communication costs associated with read/write operations, and also carry out a latency analysis of the algorithm, in which estimates for durations of successful client operations are provided. We also analyze a multi-object system, under bounded latency, to ascertain the contribution of temporary storage toward the overall storage cost. We calculate costs for a system in which the number of servers in the two layers are of the same order, i.e., $n_1 = \Theta(n_2)$. We further assume that the parameters k, d of the regenerating code are such that $k = \Theta(n_2), d = \Theta(n_2)$. This assumption is consistent with usages of codes in practical systems.

In this analysis, we assume that corresponding to any failed write operation π, there exists a successful write operation π' such that $tag(\pi') > tag(\pi)$. This essentially avoids pathological cases where the execution is a trail of only unsuccessful writes. Note that the restriction on the nature of execution was not imposed while proving liveness or atomicity.

Like in Section 4, our lemmas in this section apply only to servers that are non-faulty at the concerned point(s) of execution appearing in the lemmas. Also, we continue to ignore writes that failed before the completion of the first round.

Lemma 5.1 (**Temporary Nature of $\mathcal{L}_1$ Storage**). *Consider a successful write operation $\pi \in \beta$. Then, there exists a point of execution $T_e(\pi)$ in β such that for all $T' \geq T_e(\pi)$ in β, we have $s.t_c|_{T'} \geq tag(\pi)$ and $(t, v) \notin s.L|_{T'}, \forall s \in \mathcal{L}_1, t \leq tag(\pi)$.*

For a failed write operation $\pi \in \beta$, let π' be the first successful write in β such that $tag(\pi') > tag(\pi)$ (i,e., if we linearize all write operations, π' is the first successful write operation that appears after π - such a write operation exists because of our assumption in this section). Then, it is clear that for all $T' \geq T_e(\pi')$ in β, we have $(t, v) \notin s.L|_{T'}, \forall s \in \mathcal{L}_1, t \leq tag(\pi)$, and thus Lemma 5.1 indirectly applies to failed writes as well. Based on this observation, for any failed write $\pi \in \beta$, we define the termination point $T_{end}(\pi)$ of π as the point $T_e(\pi')$, where π' is the first successful write in β such that $tag(\pi') > tag(\pi)$.

*Definition 5.2 (***Extended write operation***).* Corresponding to any write operation $\pi \in \beta$, we define a hypothetical extended write operation π_e such that $tag(\pi_e) = tag(\pi)$, $T_{start}(\pi_e) = T_{start}(\pi)$ and $T_{end}(\pi_e) = \max(T_{end}(\pi), T_e(\pi))$, where $T_e(\pi)$ is as obtained from Lemma 5.1.

The set of all extended write operations in β shall be denoted by Π_e.

*Definition 5.3 (***Concurrency Parameter δ_ρ***).* Consider any successful read operation $\rho \in \beta$, and let π_e denote the last extended write operation in β that completed before the start of ρ. Let $\Sigma = \{\sigma_e \in \Pi_e | tag(\sigma) > tag(\pi_e)$ and σ_e overlaps with $\rho\}$. We define concurrency parameter δ_ρ as the cardinality of the set Σ.

Lemma 5.4 (**Write, Read Cost**). *The communication cost associated with any write operation in β is given by $n_1 + n_1 n_2 \frac{2d}{k(2d-k+1)} = \Theta(n_1)$. The communication cost associated with any successful read operation ρ in β is given by $n_1(1 + \frac{n_2}{d})\frac{2d}{k(2d-k+1)} + n_1 I(\delta_\rho > 0) = \Theta(1) + n_1 I(\delta_\rho > 0)$. Here, $I(\delta_\rho > 0)$ is 1 if $\delta_\rho > 0$, and 0 if $\delta_\rho = 0$.*

Remark 1. *Our ability to reduce the read cost to $\Theta(1)$ in the absence of concurrency from extended writes comes from the usage of regenerating codes at MBR point. Regenerating codes at other operating points are not guaranteed to give the same read cost. For instance, in a system with equal number of servers in either layer, also with identical fault-tolerance (i.e., $n_1 = n_2, f_1 = f_2$), it can be shown that usage of codes at the MSR point will imply that read cost is $\Omega(n_1)$ even if $\delta_\rho = 0$.*

Lemma 5.5 (**Single Object Permanent Storage Cost**). *The (worst case) storage cost in $\mathcal{L}_2$ at any point in the execution of the LDS algorithm is given by $\frac{2dn_2}{k(2d-k+1)} = \Theta(1)$.*

Remark 2. *Usage of MSR codes, instead of MBR codes, would give a storage cost of $\frac{n_2}{k} = \Theta(1)$. For fixed n_2, k, d, the storage-cost due to MBR codes is at most twice that of MSR codes. As long as we focus on order-results, MBR codes do well in terms of both storage and read costs; see Remark 1 as well.*

5.1 Bounded Latency Analysis

For bounded latency analysis, we assume the delay on the various point-to-point links are upper bounded as follows: 1) τ_1, for any link between a client and a server in $\mathcal{L}_1$, 2) τ_2, for any link between a server in $\mathcal{L}_1$ and a server in $\mathcal{L}_2$, and 3) τ_0, for any link between two servers in $\mathcal{L}_1$. We also assume that the local computations on any process take negligible time when compared to delay on any of the links. In edge computing systems, τ_2 is typically much higher than both τ_1 and τ_0.

Lemma 5.6 (**Write, Read Latency**). *A successful write operation in β completes within a duration of $4\tau_1 + 2\tau_0$. The associated extended write operation completes within a duration of $\max(3\tau_1 + 2\tau_0 + 2\tau_2, 4\tau_1 + 2\tau_0)$. A successful read operation in β completes within a duration of $\max(6\tau_1 + 2\tau_2, 5\tau_1 + 2\tau_0 + \tau_2)$.*

5.1.1 *Impact of Number of Concurrent Write Operations on Temporary Storage, via Multi-Object Analysis.*

Consider implementing N atomic objects in our two-layer storage system, via N independent instances of the LDS algorithm. The value of each of the objects is assumed to have size 1. Let θ denote an upper bounded on the total number of concurrent extended write operations experienced by the system within any duration of τ_1 time units. We show that under appropriate conditions on θ, the total storage cost is dominated by that of permanent storage in $\mathcal{L}_2$. We make the following simplifying assumptions: 1) System is symmetrical so that $n_1 = n_2, f_1 = f_2 (\implies k = d)$ 2) $\tau_0 = \tau_1$, and 3) All the invoked write operations are successful. We note that it is possible to relax any of these assumptions and give a more involved analysis. Also, let $\mu = \tau_2/\tau_1$.

Lemma 5.7 (**Relative Cost of Temporary Storage**). *At any point in the execution, the worst case storage cost in $\mathcal{L}_1$ and $\mathcal{L}_2$ are upper bounded by $\lceil 5 + 2\mu \rceil \theta n_1$ and $\frac{2Nn_2}{k+1}$. Specifically, if $\theta << \frac{Nn_2}{kn_1\mu}$, the overall storage cost is dominated by that of permanent storage in $\mathcal{L}_2$, and is given by $\Theta(N)$.*

An illustration of Lemma 5.7 is provided in Fig. 6. In this example, we assume $n_1 = n_2 = 100, k = d = 80, \tau_2 = 10\tau_1$ and $\theta = 100$, and plot $\mathcal{L}_1$ and $\mathcal{L}_2$ storage costs as a function of the number N of objects stored. As claimed in Lemma 5.7, for large N, overall storage cost is dominated by that of permanent storage in $\mathcal{L}_2$, and increases linearly with N. Also, for this example, we see that the $\mathcal{L}_2$ storage cost per object is less than 3. If we had used replication in $\mathcal{L}_2$ (along with a suitable algorithm), instead of MBR codes, the $\mathcal{L}_2$ storage cost per object would have been $n_2 = 100$.

6 CONCLUSION

In this paper we proposed a two-layer model for strongly consistent data-storage, while supporting read/write operations. Our model and LDS algorithm were both motivated by the proliferation of edge computing applications. In the model, the first layer is closer (in terms of network latency) to the clients and the second layer stores bulk data. In the presence of frequent read and write operations,

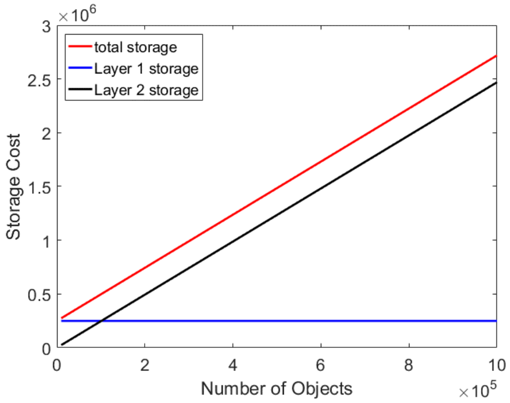

Figure 6: Illustration of the variation of $\mathcal{L}_1$ and $\mathcal{L}_2$ storage costs as a function of the number of objects stored. In this example, we assume $n_1 = n_2 = 100, k = d = 80, \tau_2 = 10\tau_1$ and $\theta = 100$.

most of the operations are served without the need to communicate with the back-end layer, thereby decreasing the latency of operations. In this regard, the first layer behaves as a proxy cache. In our algorithm, we use regenerating codes to simultaneously optimize storage and read costs. Several interesting avenues for future work exist. It is of interest to extend the framework to carry out repair of erasure-coded servers in $\mathcal{L}_2$. A model for repair in single-layer systems using erasure codes was proposed in [18]. The modularity of implementation possibly makes the repair problem in $\mathcal{L}_2$ simpler that the one in [18]. Furthermore, we would like to explore if the modularity of implementation could be advantageously used to implement a different consistency policy like regularity without affecting the implementation of the erasure codes in the back-end. Similarly, it is also of interest to study feasibility of other codes from the class of regenerating codes (like RLNCs [16]) in the back-end layer, without affecting client protocols.

7 ACKNOWLEDGMENTS

The work is supported in part by AFOSR under grants FA9550-13-1-0042, FA9550-14-1-043, FA9550-14-1-0403, and in part by NSF under awards CCF-1217506, CCF-0939370.

REFERENCES

[1] M. K. Aguilera, R. Janakiraman, and L. Xu. 2005. Using erasure codes efficiently for storage in a distributed system. In *Proceedings of International Conference on Dependable Systems and Networks (DSN)*. 336–345.

[2] M. K. Aguilera, I. Keidar, D. Malkhi, and A. Shraer. 2011. Dynamic atomic storage without consensus. *J. ACM* (2011), 7:1–7:32.

[3] H. Attiya, A. Bar-Noy, and D. Dolev. 1996. Sharing Memory Robustly in Message Passing Systems. *J. ACM* 42(1) (1996), 124–142.

[4] Flavio Bonomi, Rodolfo Milito, Jiang Zhu, and Sateesh Addepalli. 2012. Fog Computing and Its Role in the Internet of Things. In *Proceedings of the First Ed. MCC Workshop Mobile Cloud Computing (MCC 12)*. 13âĂŞ–16.

[5] C. Cachin and S. Tessaro. 2006. Optimal Resilience for Erasure-Coded Byzantine Distributed Storage. In *Proceedings of International Conference on Dependable Systems and Networks (DSN)*. 115–124.

[6] V. R. Cadambe, N. A. Lynch, M. Médard, and P. M. Musial. 2014. A Coded Shared Atomic Memory Algorithm for Message Passing Architectures. In *Proceedings of 13th IEEE International Symposium on Network Computing and Applications (NCA)*. 253–260.

[7] Viveck R Cadambe, Zhiying Wang, and Nancy Lynch. 2016. Information-theoretic lower bounds on the storage cost of shared memory emulation. In *Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing*. ACM, 305–313.

[8] Alexandros G Dimakis, P Brighten Godfrey, Yunnan Wu, Martin J Wainwright, and Kannan Ramchandran. 2010. Network coding for distributed storage systems. *Information Theory, IEEE Transactions on* 56, 9 (2010), 4539–4551.

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

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

[11] P. Dutta, R. Guerraoui, and R. R. Levy. 2008. Optimistic Erasure-Coded Distributed Storage. In *Proceedings of the 22nd international symposium on Distributed Computing (DISC)*. Berlin, Heidelberg, 182–196.

[12] Dave Evans. 2011. The Internet of Things: How the Next Evolution of the Internet Is Changing Everything. *Cisco Internet Business Solutions Group (IBSG)* (2011).

[13] R. Fan and N. Lynch. 2003. Efficient Replication of Large Data Objects. In *Distributed algorithms (Lecture Notes in Computer Science)*. 75–91.

[14] R. Guerraoui, R. R. Levy, B. Pochon, and J. Pugh. 2008. The collective memory of amnesic processes. *ACM Trans. Algorithms* 4, 1 (2008), 1–31.

[15] J. Hendricks, G. R. Ganger, and M. K. Reiter. 2007. Low-overhead byzantine fault-tolerant storage. In *ACM SIGOPS Operating Systems Review*, Vol. 41. 73–86.

[16] T. Ho, M. Medard, R. Koetter, D. R. Karger, M. Effros, J. Shi, and B. Leong. 2006. A Random Linear Network Coding Approach to Multicast. *IEEE Transactions on Information Theory* 52, 10 (Oct 2006), 4413–4430.

[17] K. M. Konwar, N. Prakash, E. Kantor, N. Lynch, M. Medard, and A. A. Schwarzmann. 2016. Storage-Optimized Data-Atomic Algorithms for Handling Erasures and Errors in Distributed Storage Systems. In *30th IEEE International Parallel & Distributed Processing Symposium (IPDPS)*.

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

[19] Kishori M. Konwar, N. Prakash, Nancy A. Lynch, and Muriel Médard. 2017. A Layered Architecture for Erasure-Coded Consistent Distributed Storage. *CoRR* abs/1703.01286 (2017). http://arxiv.org/abs/1703.01286

[20] M Nikhil Krishnan, N Prakash, V Lalitha, Birenjith Sasidharan, P Vijay Kumar, Srinivasan Narayanamurthy, Ranjit Kumar, and Siddhartha Nandi. 2014. Evaluation of Codes with Inherent Double Replication for Hadoop.. In *HotStorage*.

[21] Avinash Lakshman and Prashant Malik. 2010. Cassandra: A Decentralized Structured Storage System. *SIGOPS Oper. Syst. Rev.* 44, 2 (April 2010), 35–40.

[22] N. Lynch and A. A. Shvartsman. 2002. RAMBO: A reconfigurable atomic memory service for dynamic networks. In *Proceedings of 16th International Symposium on Distributed Computing (DISC)*. 173–190.

[23] N. A. Lynch. 1996. *Distributed Algorithms*. Morgan Kaufmann Publishers.

[24] Erik Nygren, Ramesh K. Sitaraman, and Jennifer Sun. 2010. The Akamai network: a platform for high-performance internet applications. *Operating Systems Review* 44 (2010), 2–19.

[25] K. V. Rashmi, P. Nakkiran, J. Wang, N. B. Shah, and K. Ramchandran. 2015. Having Your Cake and Eating It Too: Jointly Optimal Erasure Codes for I/O, Storage, and Network-bandwidth. In *13th USENIX Conference on File and Storage Technologies (FAST)*. 81–94.

[26] Korlakai Vinayak Rashmi, Nihar B Shah, and P Vijay Kumar. 2011. Optimal exact-regenerating codes for distributed storage at the MSR and MBR points via a product-matrix construction. *IEEE Transactions on Information Theory* 57, 8 (2011), 5227–5239.

[27] I. S. Reed and G. Solomon. 1960. Polynomial codes over certain finite fields. *Journal of the society for industrial and applied mathematics* 8, 2 (1960), 300–304.

[28] M. Sathiamoorthy, M. Asteris, D. Papailiopoulos, A. G. Dimakis, R. Vadali, S. Chen, and D. Borthakur. 2013. XORing elephants: novel erasure codes for big data. In *Proceedings of the 39th international conference on Very Large Data Bases*. 325–336.

[29] Weisong Shi, Jie Cao, Quan Zhang, Youhuizi Li Li, and Lanyu Xu. 2016. Edge Computing: Vision and Challenges. *IEEE INTERNET OF THINGS* 3, 5 (October 2016).

[30] Alexander Spiegelman, Yuval Cassuto, Gregory Chockler, and Idit Keidar. 2016. Space Bounds for Reliable Storage: Fundamental Limits of Coding. In *Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing (PODC '16)*. ACM, New York, NY, USA.