Fragmented ARES: Dynamic Storage for Large Objects

Chryssis Georgiou  
*University of Cyprus*  
Nicosia, Cyprus  
chryssis@ucy.ac.cy

Nicolas Nicolaou  
*Algolysis Ltd*  
Limassol, Cyprus  
nicolas@algolysis.com

Andria Trigeorgi  
*University of Cyprus*  
Nicosia, Cyprus  
atrige01@cs.ucy.ac.cy

Abstract—Data availability is one of the most important features in distributed storage systems, made possible by data replication. Nowadays data are generated rapidly and the goal to develop efficient, scalable and reliable storage systems has become one of the major challenges for high performance computing. In this work, we develop a dynamic, robust and strongly consistent distributed storage implementation suitable for handling large objects (such as files). We do so by integrating an Adaptive, Reconfigurable, Atomic Storage framework, called ARES, with a distributed file system, called CoBFS, which relies on a block fragmentation technique to handle large objects. With the addition of ARES, we also enable the use of an erasure-coded algorithm to further split our data and to potentially improve storage efficiency at the replica servers and operation latency.

To put the practicality of our outcomes at test, we conduct an in-depth experimental evaluation on the Emulab and AWS EC2 testbeds, illustrating the benefits of our approaches, as well as other interesting tradeoffs.

Index Terms—Distributed storage, Large objects, Strong consistency, High access concurrency, Erasure code, Reconfiguration

I. INTRODUCTION

Motivation and prior work. Distributed Storage Systems (DSS) have gained momentum in recent years, following the demand for available, accessible, and survivable data storage [25], [27]. To preserve those properties in a harsh, asynchronous, fail prone environment (as a distributed system), data are replicated in multiple, often geographically separated devices, raising the challenge of how to preserve consistency between the replica copies.

For more than two decades, a series of works [7], [11], [13], [14], [20] suggested solutions for building distributed shared memory emulations, allowing data to be shared concurrently offering basic memory elements, i.e. registers, with strict consistency guarantees. Linearity (atomicity) [15] is the most challenging, yet intuitive consistency guarantee that such solutions provide.

The problem of keeping copies consistent becomes even more challenging when failed replica hosts (or servers) need to be replaced or new servers need to be added in the system. Since the data of a DSS should be accessible immediately, it is imperative that the service interruption during a failure or a repair should be as short as possible. The need to be able to modify the set of servers while ensuring service liveness yielded dynamic solutions and reconfiguration services. Examples of reconfigurable storage algorithms are RAMBO [18], DynaSTORE [2], SM-STORE [17], SPNSTORE [12] and ARES [22].

Currently, such emulations are limited to small-size, versionless, primitive objects (like registers), hindering the practicality of the solutions when dealing with larger, more common DSS objects (like files). A recent work by Anta et al. [4], introduced a modular solution, called CoBFS, which combines a suitable data fragmentation strategy with a distributed shared memory module to boost concurrency, while maintaining strong consistency guarantees, and minimizing operation latencies. The architecture of CoBFS is shown in Fig. 1 and it is composed of two main modules: (i) a Fragmentation Module (FM), and (ii) a Distributed Shared Memory Module (DSMM).

In short, the FM implements a fragmented object, which is a totally ordered sequence of blocks (where a block is a R/W object of restricted size), while the DSMM implements an interface to a shared memory service that allows operations on individual block objects. To this end a DSMM may encapsulate any DSM implementation. CoBFS implements coverable linearizable fragmented objects. Coverability [21] extends linearity with the additional guarantee that object writes succeed when associating the written value with the "current" version of the object. In a different case, a write operation becomes a read operation and returns the latest version and the associated value of the object. Coverable solutions have been proposed only for the static environment. Thus, CoBFS operated over a static architecture, with the replica hosts predefined in the beginning of its execution.

Contributions. In this work, we propose a dynamic DSS, that (i) supports versioned objects, (ii) is suitable for large objects (such as files), and (iii) is storage-efficient. To achieve this, we integrate the dynamic distributed shared memory algorithm ARES with the DSMM module in CoBFS. ARES is the first algorithm that enables erasure coded based dynamic DSM yielding benefits on the storage efficiency at the replica hosts. To support versioning we modify ARES to implement coverable objects, while high access concurrency is preserved by introducing support for fragmented objects. Ultimately, we
aim to make a leap towards dynamic DSS that will be attractive for practical applications (like highly concurrent and consistent file sharing).

In summary, our contributions are the following:

- We propose and prove the correctness of the coverable version of ARES, CoARES, a Fault-tolerant, Reconfigurable, Erasure coded, Atomic Storage, to support versioned objects (Section V).
- We adopt the idea of fragmentation as presented in CoBFS, to obtain CoARESF, which enables CoARES to handle large shared data objects and increased data access concurrency (Section V). The correctness of CoARESF is rigorously proven.
- To reduce the operational latency of the read/write operations in the DSM layer, we apply and prove correct an optimization in the implementation of the erasure coded data-access primitives (DAP) used by the ARES framework (which includes CoARES and CoARESF). This optimization has its own interest, as it could be applicable beyond the ARES framework, i.e., by other erasure coded algorithms relying on tag-ordered data-access primitives.
- We present an in-depth experimental evaluation of our approach over Emulab, a popular emulation testbed, and Amazon Web Services (AWS) EC2, an overlay (real-time) tested (Section VII). Our experiments compare various versions of our implementation, i.e., with and without the fragmentation technique or with and without Erasure Code or with and without reconfiguration, illustrating tradeoffs and synergies.

II. MODEL AND DEFINITIONS

In this section we present the system setting and define necessary terms we use in the rest of the manuscript. As mentioned, our main goal is to implement a highly consistent shared storage that supports large shared objects and favors high access concurrency. We assume read/write (R/W) shared objects that support two operations: (i) a read operation that returns the value of the object, and (ii) a write operation that modifies the value of the object.

**Executions and histories** An execution $\xi$ of a distributed algorithm $A$ is an alternating sequence of states and actions of $A$ reflecting the evolution in real time of the execution. A history $H_\xi$ is the subsequence of the actions in $\xi$. A history $H_\xi$ is sequential if it starts with an invocation action and each invocation is immediately followed by its matching response; otherwise, $H_\xi$ is concurrent. Finally, $H_\xi$ is complete if every invocation in $H_\xi$ has a matching response in $H_\xi$, i.e., each operation in $\xi$ is complete. An operation $\pi_1$ precedes an operation $\pi_2$ (or $\pi_2$ succeeds $\pi_1$), denoted by $\pi_1 \rightarrow_2 \pi_2$, in $H_\xi$, if the response action of $\pi_1$ appears before the invocation action of $\pi_2$ in $H_\xi$. Two operations are concurrent if none precedes the other.

**Clients and servers.** We consider a system composed of four distinct sets of crash-prone, asynchronous processes: a set $\mathcal{W}$ of writers, a set $\mathcal{R}$ of readers, a set $\mathcal{G}$ of reconfiguration clients, and a set $\mathcal{S}$ of servers. Let $\mathcal{I} = \mathcal{W} \cup \mathcal{R} \cup \mathcal{G}$ be the set of clients. Servers host data elements (replicas or encoded data fragments). Each writer is allowed to modify the value of a shared object, and each reader is allowed to obtain the value of that object. Reconfiguration clients attempt to introduce new configuration of servers to the system in order to mask transient errors and to ensure the longevity of the service.

**Configurations.** A configuration, $c \in \mathcal{C}$, consists of:

(i) $c$.Servers $\subseteq \mathcal{S}$: a set of server identifiers;
(ii) $c$.Quorums: the set of quorums on $c$.Servers, s.t. $\forall Q_1, Q_2 \in c$.Quorums, $Q_1, Q_2 \subseteq c$.Servers and $Q_1 \cap Q_2 \neq \emptyset$;
(iii) $DAP(c)$: the set of data access primitives (operations at level lower than reads or writes) that clients in $\mathcal{I}$ may invoke on $c$.Servers (cf. Section III);
(iv) $c$.Con: a consensus instance with the values from $c$, implemented and running on top of the servers in $c$.Servers; and
(v) the pair $(c$.tag, $c$.val): the maximum tag-value pair that clients in $\mathcal{I}$ have. A tag consists of a timestamp $ts$ (sequence number) and a writer id; the timestamp is used for ordering the operations, and the writer id is used to break symmetry (when two writers attempt to write concurrently using the same timestamp) [15]. We refer to a server $s \in c$.Servers as a member of configuration $c$.

The consensus instance $c$.Con in each configuration $c$ is used as a service that returns the identifier of the configuration that follows $c$.

**Fragmented objects and fragmented linearizability.** As defined in [4], a fragmented object is a totally ordered sequence of block objects. Let $\mathcal{F}$ denote the set of fragmented objects, and $B$ the set of block objects. A block object (or block) $b \in B$ is a concurrent R/W object with a unique id and is associated with two structures, $val$ and $ver$. The unique id of the block is a triplet $(fid, clid, clseq) \in F \times I \times N$, where $fid \in F$ is the id of the fragmented object in which the block belongs to, $clid \in I$ is the id of the client that created the block, and $clseq \in N$ is the client’s local sequence number of blocks that is incremented every time this client creates a block for this fragmented object. $val(b)$ is composed of: (i) a pointer that points to the next block in the sequence (⊥ denotes the null pointer), and (ii) the data contained in the block (⊥ means there are no data). $ver(b) = \langle wid, bseq \rangle$, where $wid \in \mathcal{I}$ is the id of the client that last updated $val(b)$ (initially is the id of the creator of the block) and $bseq \in \mathbb{N}$ is a sequence number of the block (initially 0) that it is incremented every time $val(b)$ is updated.

A fragmented object $f$ is a concurrent R/W object with a unique identifier from a set $\mathcal{F}$. Essentially, a fragmented
object is a sequence of blocks from \( \mathcal{B} \), with a value \( \text{val}(f) = (b_0, b_1, b_2, \ldots) \), where each \( b_i \in \mathcal{B} \). Initially, each fragmented object contains an empty block, i.e., \( \text{val}(f) = (b_0) \) with \( \text{val}(b_0) = \varepsilon \); we refer to this as the genesis block. We now proceed to present the formal definitions of linearizability and fragmented linearizability, as given in [4].

**Definition 1 (Linearizability).** An object \( f \) is linearizable [16] if, given any complete history \( H \), there exists a permutation \( \sigma \) of all actions in \( H \) such that:

1. \( \sigma \) is a sequential history and follows the sequential specification of \( f \), and
2. for operations \( \pi_1, \pi_2 \), if \( \pi_1 \rightarrow \pi_2 \) in \( H \), then \( \pi_1 \) appears before \( \pi_2 \) in \( \sigma \).

Given a history \( H \), we denote for an operation \( \pi \) the history \( H^\pi \) which contains the actions extracted from \( H \) and performed during \( \pi \) (including its invocation and response actions). Hence, if \( \text{val}(f) \) is the value returned by \( \text{read}(f) \), then \( H^\text{read}(f) \) contains an invocation and matching response for a \( \text{read}(b) \) operation, for each \( b \in \text{val}(f) \). Then, from \( H \), we can construct a history \( H_f \) that only contains operations on the whole fragmented object.

**Definition 2 (Fragmented Linearizability [4]).** Let \( f \in \mathcal{F} \) be a fragmented object, \( H \) a complete history on \( f \), and \( \text{val}(f)_H \subseteq \mathcal{B} \) the value of \( f \) at the end of \( H \). Then, \( f \) is fragmented linearizable if there exists a permutation \( \sigma_0 \) over all the actions on \( b \) in \( H \), \( \forall b \in \text{val}(f)_H \), such that:

1. \( \sigma_0 \) is a sequential history that follows the sequential specification of \( f \) and
2. for operations \( \pi_1, \pi_2 \) that appear in \( H_f \) extracted from \( H \), if \( \pi_1 \rightarrow \pi_2 \) in \( H_f \), then all operations on \( b \) in \( H^{\pi_1} \) appear before any operation on \( b \) in \( H^{\pi_2} \) in \( \sigma_0 \).

Fragmented linearizability guarantees that all concurrent operations on different blocks prevail, and only concurrent operations on the same blocks are conflicting. The second point guarantees the total ordering of the operations on the fragmented object with respect to their real-time ordering. For example, considering two read operations on \( f \), say \( \rho_1 \rightarrow \rho_2 \), it must be a case that \( \rho_2 \) returns a supersequence of the sequence returned by \( \rho_1 \).

**Coverability and fragmented coverability.** Coverability is defined over a totally ordered set of versions, say \( \text{Versions} \), and introduces the notion of versioned (coverable) objects. According to [21], a coverable object is a type of R/W object where each value written is assigned with a version from the set \( \text{Versions} \). Denoting a successful write as \( \text{tr-write}(\text{ver}) \langle \text{ver}', \text{chg} \rangle_p \) (updating the object from version \( \text{ver} \) to \( \text{ver}' \)), and an unsuccessful write as \( \text{tr-write}(\text{ver}) \langle \text{ver}', \text{unchg} \rangle_p \), a coverable implementation satisfies the properties consolidation, continuity and evolution as formally defined below in Definition 4.

Intuitively, consolidation specifies that write operations may revise the register with a version larger than any version modified by a preceding write operation, and may lead to a version newer than any version introduced by a preceding write operation. Continuity requires that a write operation may revise a version that was introduced by a preceding write operation, according to the given total order. Finally, evolution limits the relative increment on the version of a register that can be introduced by any operation.

We say that a write operation revises a version \( \text{ver} \) of the versioned object to a version \( \text{ver}' \) (or produces \( \text{ver}' \)) in an execution \( \xi \), if \( \text{tr-write}(\text{ver}) \langle \text{ver}' \rangle_p \) completes in \( H_{\xi} \). The set of successful write operations on a history \( H_k \) can be defined as \( \mathcal{W}_{\xi,\text{suc}} = \{ \pi : \pi = \text{tr-write}(\text{ver}) \langle \text{ver}' \rangle_p \text{ completes in } H_k \} \). The set now of produced versions in the history \( H_k \) is defined by \( \text{Versions}_{\xi} = \{ \text{ver}_i : \text{tr-write}(\text{ver}) \langle \text{ver}_i \rangle_p \in \mathcal{W}_{\xi,\text{suc}} \cup \{ \text{ver}_0 \} \} \), where \( \text{ver}_0 \) is the initial version of the object. Observe that the elements of \( \text{Versions}_{\xi} \) are totally ordered.

**Definition 3 (Validity).** An execution \( \xi \) (resp. its history \( H_{\xi} \)) is a valid execution (resp. history) on a versioned object, for any \( \pi_i, \pi_j \in \mathcal{I} \):

1. For any operations \( \text{tr-write}(\text{ver}) \langle \text{ver}' \rangle_p \), \( \text{tr-write}(\text{ver}) \langle \text{ver}'' \rangle_p \), and \( \text{tr-write}(\text{ver}) \langle \text{ver}'', \text{chg} \rangle_p \), \( \text{tr-write}(\text{ver}) \langle \text{ver}'' \rangle_p \) such that \( \text{ver}' \neq \text{ver}'' \), and for each \( \text{ver}_k \in \text{Versions}_{\xi} \), there is a sequence of versions \( \text{ver}_0, \text{ver}_1, \ldots, \text{ver}_k \), such that \( \text{tr-write}(\text{ver}_i) \langle \text{ver}_{i+1} \rangle \in \mathcal{W}_{\xi,\text{suc}} \), for \( 0 \leq i < k \).

**Definition 4 (Coverability [21]).** A valid execution \( \xi \) is coverable with respect to a total order \( \preceq_\xi \) on operations in \( \mathcal{W}_{\xi,\text{suc}} \) if:

1. (Consolidation) If \( \pi_1 = \text{tr-write}(\text{ver}) \langle \text{ver}_1 \rangle_p, \pi_2 = \text{tr-write}(\text{ver}) \langle \text{ver}_2 \rangle |* \in \mathcal{W}_{\xi,\text{suc}}, \) and \( \pi_1 \rightarrow \pi_2 \) in \( H_{\xi} \), then \( \text{ver}_1 \preceq \text{ver}_2 \) and \( \pi_1 \preceq_\xi \pi_2 \).
2. (Continuity) If \( \pi_2 = \text{tr-write}(\text{ver}) \langle \text{ver}_2 \rangle_p \in \mathcal{W}_{\xi,\text{suc}}, \) then there exists \( \pi_1 \in \mathcal{W}_{\xi,\text{suc}} \) s.t. \( \pi_1 = \text{tr-write}(\text{ver}) \langle \text{ver} \rangle_p \) and \( \pi_1 \preceq_\xi \pi_2, \) or \( \text{ver} = \text{ver}_1 \).
3. (Evolution) \( \text{ver}, \text{ver}', \text{ver}'' \in \text{Versions}_{\xi} \). If \( \text{there are sequences of versions } \text{ver}_1, \text{ver}_2, \ldots, \text{ver}_k \) and \( \text{ver}', \text{ver}''_1, \ldots, \text{ver}''_k \), where \( \text{ver} = \text{ver}_1 = \text{ver}_1' = \text{ver}_2, \ldots, \text{ver}''_k = \text{ver}'' \), and \( \text{ver}'' = \text{ver}'' \) such that \( \text{tr-write}(\text{ver}) \langle \text{ver}_{i+1} \rangle \in \mathcal{W}_{\xi,\text{suc}} \) for \( 1 \leq i < k \), and \( \text{tr-write}(\text{ver}) \langle \text{ver}''_i \rangle \in \mathcal{W}_{\xi,\text{suc}} \) for \( 1 \leq i < \ell \), and \( k < \ell \), then \( \text{ver}' < \text{ver}'' \).

If a fragmented object utilizes coverable blocks, instead of linearizable blocks, then Definition 2 together with Definition 4 provide what we would call fragmented coverability: Concurrent update operations on different blocks would always prevail (as long as each update is tagged with the latest version of each block), whereas only one update operation on the same block would prevail (all the other updates on the same block that are concurrent with this would become a read operation).

III. ARES: A FRAMEWORK FOR DYNAMIC STORAGE

ARES [22] is a modular framework, designed to implement dynamic, reconfigurable, fault-tolerant, read/write distributed
linearizable (atomic) shared memory objects.

Similar to traditional implementations, ARES uses \((tag, value)\) pairs to order the operations on a shared object. In contrast to existing solutions, ARES does not explicitly define the exact methodology to access the object replicas. Rather, it relies on three, so called, data access primitives (DAPs): (i) the get-tag primitive which returns the tag of an object, (ii) the get-data primitive which returns a \((tag, value)\) pair, and (iii) the put-data\((\tau, v)\) primitive which accepts a \((tag, value)\) as argument.

As seen in [23], these DAPs may be used to express the data access strategy (i.e., how they retrieve and update the object data) of different shared memory algorithms (e.g., [6]). Therefore, the DAPs help ARES to achieve a modular design, agnostic of the data access strategies, and in turn enables ARES to use different DAP implementation per configuration (something impossible for other solutions). Linearizability is then preserved by ARES given that the DAPs satisfy the following property in any given configuration \(c\):

**Property 1** (DAP Consistency Properties). In an execution \(\xi\) we say that a DAP operation in \(\xi\) is complete if both the invocation and the matching response step appear in \(\xi\). If \(\Pi\) is the set of complete DAP operations in execution \(\xi\) then for any \(\phi, \pi \in \Pi\):

- **C1** If \(\phi = c.\text{put-data}(\langle \tau_\phi, v_\phi \rangle), \langle \tau_\phi, v_\phi \rangle \in T \times V,\) and \(\pi\) is \(c.\text{get-tag}()\) (or \(c.\text{get-data}())\) that returns \(\pi_\tau \in T\) (or \(\langle \pi_\tau, v_\tau \rangle \in T \times V\)) and \(\phi \rightarrow \pi\) in \(\xi\), then \(\pi_\tau \geq \tau_\phi\).

- **C2** If \(\phi = c.\text{get-data}()\) that returns \(\langle \pi_\tau, v_\pi \rangle \in T \times V,\) then there exists \(\pi\) such that \(\pi\) is a \(c.\text{put-data}(\langle \tau_\pi, v_\pi \rangle)\) and \(\phi\) did not complete before the invocation of \(\pi\). If no such \(\pi\) exists in \(\xi\), then \(\langle \tau_\pi, v_\pi \rangle\) is equal to \(\langle t_0, v_0 \rangle\).

**DAP Implementations:** To demonstrate the flexibility that DAPs provide, the authors in [10], expressed two different atomic shared R/W algorithms in terms of DAPs. These are the DAPs for the well celebrated ABD [6] algorithm, and the DAPs for an erasure coded based approach presented for the first time in [10]. In the rest of the manuscript we refer to the two DAP implementations as ABD-DAP and EC-DAP. Erasure-coded approaches became popular in implementing atomic R/W objects as they offer fault tolerance and storage efficiency at the replica hosts. In particular, an \([n, k]\)-MDS erasure coding algorithm (e.g., Reed-Solomon) splits the object into \(k\) equally sized fragments. Then erasure coding is applied to these fragments to obtain \(n\) coded elements, which consist of the \(k\) encoded data fragments and \(m\) encoded parity fragments. The \(n\) coded fragments are distributed among a set of \(n\) different replica servers. Any \(k\) of the \(n\) coded fragments can then be used to reconstruct the initial object value. As servers maintain a fragment instead of the whole object value, EC based approaches claim significant storage benefits. By utilizing the EC-DAP, ARES became the first erasure coded dynamic algorithm to implement an atomic R/W object.

Given the DAPs we can now provide a high-level description of the two main functionalities supported by ARES: (i) the reconfiguration of the data replicas, and (ii) the read/write operations on the shared object.

**Reconfiguration.** Reconfiguration is the process of changing the set of servers that hold the object replicas. A configuration sequence \(cseq\) in ARES is defined as a sequence of pairs \(\langle c, status \rangle\) where \(c \in C\), and \(status \in \{P, F\}\) \((P\) stands for proposed and \(F\) for finalized). Configuration sequences are constructed and stored in clients, while each server in a configuration \(c\) only maintains the configuration that follows \(c\) in a local variable \(nextc \in C \cup \{\bot\} \times \{P, F\}\).

To perform a reconfiguration operation \(\text{recon}(c)\), a client follows 4 steps. At first, the reconfiguration client \(r\) executes a sequence traversal to discover the latest configuration sequence \(cseq\). Then it attempts to add \(\langle c, P \rangle\) at the end of \(cseq\) by proposing \(c\) to a consensus mechanism. The outcome of the consensus may be a configuration \(c'\) (possibly different than \(c\)) proposed by some reconfiguration client. Then the client determines the maximum tag-value pair of the object, say \(\langle \tau, v \rangle\) by executing get-data operation starting from the last finalized configuration in \(cseq\) to the last configuration in \(cseq\), and transfers the pair to \(c'\) by performing put-data\((\langle \tau, v \rangle)\) on \(c'\). Once the update of the value is complete, the client finalizes the proposed configuration by setting \(nextc = \langle c', F \rangle\) on a quorum of servers of the last configuration in \(cseq\) (or \(c_0\) if no other configuration exists).

The traversal and reconfiguration procedure in ARES preserves three crucial properties: (i) configuration uniqueness, i.e., the configuration sequences in any two processes have identical configuration at any index \(i\), (ii) sequence prefix, i.e., the configuration sequence witnessed by an operation is a prefix of the sequence witnessed by any succeeding operation, and (iii) sequence progress, i.e., if the configuration with index \(i\) is finalized during an operation, then a configuration \(j\), for \(j \geq i\), will be finalized for a succeeding operation.

**Read/Write operations.** A write (or read) operation \(\pi\) by a client \(p\) is executed by performing the following actions: (i) \(\pi\) invokes a read-config action to obtain the latest configuration sequence \(cseq\), (ii) \(\pi\) invokes a get-tag (if a write) or get-data (if a read) in each configuration, starting from the last finalized to the last configuration in \(cseq\), and discovers the maximum \(\tau\) or \(\langle \tau, v \rangle\) pair respectively, and (iii) repeatedly invokes put-data\((\langle \tau', v' \rangle)\), where \(\langle \tau', v' \rangle = \langle \tau + 1, v' \rangle\) if \(\pi\) is a write and \(\langle \tau', v' \rangle = \langle \tau, v \rangle\) if \(\pi\) is a read in the last configuration in \(cseq\), and read-config to discover any new configuration, until no additional configuration is observed.

**IV. COARES: COVERABLE ARES**

In this section we replace the R/W objects of ARES with versioned objects that use coverability (cf. Section I), yielding the coverable variant of ARES, which we refer as COARES. We first present the algorithms and then its correctness proof.

**A. Description**

In this section we describe the modification that need to occur on ARES in order to support coverability. The reconfiguration protocol and the DAP implementations remain the
the changes occur in the specification of read/write operations, which we detail below.

**Read/Write operations.** Algorithm 1 specifies the read and write protocols of COARES. The blue text annotates the changes when compared to the original ARES read/write protocols. The local variable \( flag \in \{ \text{chg, unchg} \} \), maintained by the write clients, is set to \( \text{chg} \) when the write operation is successful and to \( \text{unchg} \) otherwise; initially it is set to \( \text{unchg} \).

The state variable \( version \) is used by the client to maintain the tag of the coverable object. At first, in both cvr-read and cvr-write operations, the read/write client issues a read-config action to obtain the latest introduced configuration; cf. line Alg. 111 (resp. line Alg. 140).

In the case of cvr-write, the writer \( w_i \) finds the last finalized entry in \( cseq \), say \( \mu \), and performs a \( cseq[j].conf.get-data() \) action, for \( \mu \leq j \leq |cseq| \) (lines Alg. 112-115). Thus, \( w_i \) retrieves all the \( (\tau, v) \) pairs from the last finalized configuration and all the pending ones. Note that in cvr-write, get-data is used in the first phase instead of a get-tag, as the coverable version needs both the highest tag and value and not only the tag, as in the original write protocol. Then, the writer computes the maximum \( (\tau, v) \) pair among all the returned replies. Lines Alg. 116 - 120 depict the main difference between the coverable cvr-write and the original one: if the maximum \( \tau \) is equal to the state variable \( version \), meaning that the writer \( w_i \) has the latest version of the object, it proceeds to update the state of the object \( (\langle \tau, v \rangle) \) by increasing \( \tau \) and assigning \( \langle \tau, v \rangle \) to \( \langle (\tau.ts + 1, \omega_i), val \rangle \), where \( val \) is the value it wishes to write (lines Alg. 117-118). Otherwise, the state of the object does not change and the writer keeps the maximum \( (\tau, v) \) pair found in the first phase (i.e., the write has become a read). No matter whether the state changed or not, the writer updates its \( version \) with the value \( \tau \) (line Alg. 121).

In the case of cvr-read, the first phase is the same as the original, that is, it discovers the maximum tag-value pair among the received replies (lines Alg. 113-114). The propagation of \( (\tau, v) \) in both cvr-read (lines Alg. 116-119) and cvr-write (lines Alg. 123-129) remains the same as the original. Finally, the cvr-write operation returns \( (\tau, v) \) and the \( flag \), whereas the cvr-read operation only returns \( (\langle \tau, v \rangle) \).

**B. Correctness of COARES**

COARES is correct if it satisfies liveness (termination) and safety (i.e., linearizable coverability). Termination holds since read, update and reconfig operations on the COARES always complete given that the DAP complete. As shown in 22, ARES, implements an atomic object given that the DAP used satisfy Property 1. Given that COARES uses the same reconfiguration and read operations, and only the write operation is sometime converted to read operation then linearizability is not affected and can be shown that it holds in a similar way as in 22.

The validity and coverability properties, as defined in Definitions 3 and 4, remain to be examined. In COARES, we use tags to denote the version of the register. Given that the DAP(\( c \)) used in any configuration \( c \in C \) satisfies Property 1 we will show that any execution \( \xi \in C \) of COARES satisfies the properties of Definitions 3 and 4. In the lemmas that follow we refer to a successful write operation to the one that is not converted to a read operation.

We begin with some lemmas that help us show that COARES satisfies Validity.

**Lemma 5** (Version Increment). In any execution \( \xi \) of COARES, if \( \omega \) is a successful write operation, and \( ver \) the maximum version it discovered during the get-data operation, then \( \omega \) propagates a version \( ver' > ver \).

**Proof.** This lemma follows from the fact that COARES uses a condition before the propagation phase in line Alg. 116. The writer checks if the maximum tag retrieved from the get-data action is equal to the local \( version \). If that holds, then the writer generates a new version larger than its local version by incrementing the tag found.

**Lemma 6** (Version Uniqueness). In any execution \( \xi \) of COARES, if two write operations \( \omega_1 \) and \( \omega_2 \), write values associated with versions \( ver_1 \) and \( ver_2 \) respectively, then \( ver_1 \neq ver_2 \).

**Proof.** A tag is composed of an integer timestamp \( ts \) and the id of a process \( wid \). Let \( w_1 \) be the id of the writer that invoked \( \omega_1 \) and \( w_2 \) the id of the writer that invoked \( \omega_2 \). To show whether the versions generated by the two write operations are not equal we need to examine two cases: (a) both \( \omega_1 \) and \( \omega_2 \) are invoked by the same writer, i.e. \( w_1 = w_2 \), and (b) \( \omega_1 \) and \( \omega_2 \) are invoked by two different writers, i.e. \( w_1 \neq w_2 \).

**Case a:** In this case, the uniqueness of the versions is achieved due to the well-formedness assumption and the C1 term in Property 1. By well-formedness, writer \( w_1 \) can only invoke one operation at a time. Thus, the last put-data(\( ver_1, * \)) of \( \omega_1 \) completes before the first get-data of \( \omega_2 \).

If both operations are invoked and completed in the same configuration \( c \) then by C1, the version \( ver' \) returned by \( c.get-data, \) is \( ver' \geq ver_1 \). Since the version is incremented in \( \omega_2 \) then \( ver_2 = ver' + 1 > ver_1 \), and hence \( ver_1 \neq ver_2 \) as desired.

It remains to examine the case where the put-data was invoked in a configuration \( c \) and the get-data in a configuration \( c' \). Since by well-formedness \( \omega_1 \rightarrow \omega_2 \), then by the sequence prefix guaranteed by the reconfiguration protocol of ARES (second property) the \( cseq_1 \) obtained during the read-config action in \( \omega_1 \) is a prefix of the \( cseq_2 \) obtained during the same action in \( \omega_2 \). Notice that \( c' \) is the last finalized configuration in \( cseq_2 \) as this is the configuration where the first get-data action of \( \omega_2 \) is invoked. If \( c' \) appears before \( c \) in \( cseq_2 \) then by COARES the write operation \( \omega_2 \) will invoke a get-data operation in \( c \) as well and with the same reasoning as before will generate a \( ver_2 \neq ver_1 \). If now \( c \) appears before \( c' \) in \( cseq_2 \), then it must be the case that a reconfiguration operation \( r \) has been invoked concurrently or after \( \omega_2 \) and added \( c' \). By ARES 22, \( r \) invoked a put-data(\( ver' \)) in \( c' \) before finalizing \( c' \) with \( ver' \geq ver_1 \). So when \( \omega_2 \) invokes get-data in \( c' \) by C1
Algorithm 1 Write and Read protocols for CoARES.

CVR-Write Operation:
2: at each writer \( w_i \)
3: State Variables:
4: \( cseq[s.t.cseq][j] \in C \times \{F, P\} \)
5: \( version \in \mathbb{N}^* \times W \) initially \((0, \bot)\)
6: Local Variables:
7: \( flag \in \{chg, unchg\} \) initially unchg
8: Initialization:
9: \( cseq[0] = (\omega, F) \)
10: \( \omega \) put-data
11: \( \mu \leftarrow \max(\{i : cseq[i], status = F\}) \)
12: \( \nu \leftarrow |cseq| \)
13: \( \langle \tau, v \rangle \leftarrow \max(\{\langle j : cseq[j], status = F\rangle\}) \)
14: for \( i = \mu : \nu \) do
15: \( flag \leftarrow chg \)
16: \( \langle \tau, v \rangle \leftarrow ((\tau, ts_i + 1, \omega_1), val) \)
17: else
18: \( flag \leftarrow unchg \)
19: \( version \leftarrow \tau \)
20: done \leftarrow false
21: while not done do
22: \( cseq[i].cfg \) get-data\((\tau, v)) \)
23: \( cseq \) read-config\((cseq) \)
24: if \( |cseq| = \nu \) then
25: \( done \leftarrow true \)
26: end while
27: return \( (\tau, v), flag \)
28: \( return \( (\tau, v), done \)
29: end while
30: return \( (\tau, v), flag \)
31: end operation
32: end operation
33: CVR-Read Operation:
34: at each reader \( r_i \)
35: State Variables:
36: \( cseq[s.t.cseq][j] \in C \times \{F, P\} \)
37: \( cseq[0] = (\omega, F) \)
38: \( \omega \) get-data
39: \( \mu \leftarrow \max(\{i : cseq[i], status = F\}) \)
40: \( \nu \leftarrow |cseq| \)
41: for \( i = \mu : \nu \) do
42: \( \langle \tau, v \rangle \leftarrow \max(\{\langle j : cseq[j], status = F\rangle\}) \)
43: \( done \leftarrow false \)
44: while not done do
45: \( cseq[i].cfg \) put-data\((\tau, v)) \)
46: \( cseq \) read-config\((cseq) \)
47: if \( cseq[i] = \nu \) then
48: \( done \leftarrow true \)
49: else
50: \( \nu \leftarrow |cseq| \)
51: end while
52: return \( (\tau, v), done \)
53: end while
54: return \( (\tau, v), done \)
55: end operation

will obtain a version \( ver'' \geq ver' \geq ver_1 \). Hence \( ver_2 > ver'' \) and thus \( ver_2 \neq ver_1 \) as needed.

Case b: When \( w_1 \neq w_2 \) then \( \omega_1 \) generates a version \( ver_1 = \{ts_1, w_1\} \) and \( \omega_2 \) generates some version \( ver_2 = \{ts_2, w_2\} \). Even if \( ts_1 = ts_2 \) the two version differ on the unique id of the writers and hence \( ver_1 \neq ver_2 \). This completes the case and the proof.

Lemma 7. Each version we reach in an execution is derived (through a chain of operations) from the initial version of the register \( ver_0 \). From this point onward we fix \( \xi \) to be a valid execution and \( H_\xi \) to be its valid history.

Proof. Every tag is generated by extending the tag retrieved by a get-data operation starting from the initial tag (lines Alg. 111718). In turn, each get-data operation returns a tag written by a put-data operation or the initial tag (as per C2 in Property 1). Then, applying a simple induction, we may show that there is a sequence of tags leading from the initial tag to the tag used by the write operation.

Lemma 8. In any execution \( \xi \) of CoARES, all the coverability properties of Definition 4 are satisfied.

Proof. For consolidation we need to show that for two write operations \( \omega_1 = cvr-\omega(\tau_1, chg) \) and \( \omega_2 = cvr-\omega(\tau_2)[*, chg] \), if \( \omega_1 \rightarrow \omega_2 \) then \( \tau_1 \leq \tau_2 \). According to C1 of Property 1 since the get-data of \( \omega_2 \) appears after the put-data of \( \omega_1 \), the get-data of \( \omega_2 \) returns a tag higher than the one written by \( \omega_1 \).

Continuity is preserved as a write operation first invokes a get-data action for the latest tag before proceeding to put-data to write a new value. According to C2 of Property 1 the get-data action returns a tag already written by a put-data or the initial tag of the register.

To show that evolution is preserved, we take into account that the version of a register is given by its tag, where tags are compared lexicographically. A successful write \( \tau' = cvr-\omega(\tau) \) generates a new tag \( \tau' \) from \( \tau \) such that \( \tau'.ts = \tau.ts + 1 \) (line Alg. 1118). Consider sequences of tags \( \tau_1, \tau_2, \ldots, \tau_k \) and \( \tau_1', \tau_2', \ldots, \tau_k' \) such that \( \tau_1 = \tau_1' \). Assume that \( cvr-\omega(\tau_i)[\tau_i + 1] \), for \( 1 \leq i < k \), and \( cvr-\omega(\tau_i)[\tau_i' + 1] \), for \( 1 \leq i \leq k \), are successful writes. If \( \tau_1.ts = \tau_1' ts = z \), then \( \tau_1.ts = z + k \) and \( \tau_1'.ts = z + k + \ell \) and if \( k < \ell \) then \( \tau_1 < \tau_1' \).

The main result of this section follows:

Theorem 9. CoARES implements an atomic coverable object given that the DAPs implemented in any configuration \( c \) satisfy Property 7.

Proof. Atomicity follows from the fact that ARES implement an atomic object if the DAPs satisfy Property 1, Lemmas 5, 6 and 7 show that CoARES satisfies validity (see Definition 3), and Lemma 8 that CoARES satisfies the coverability properties (see Definition 4). Thus the theorem follows.

V. CoARES: INTEGRATE CoARES WITH A FRAGMENTATION APPROACH

The work in [4] developed a distributed storage framework, called CoBFS, which utilizes coverable fragmented objects. CoBFS adopts a modular architecture, separating the object fragmentation process from the shared memory service allowing it to use different shared memory implementations.

In this section we describe how CoARES can be integrated with CoBFS to obtain what we call CoARES, thus yielding a
dynamic consistent storage suitable for large objects. Furthermore, this enables to combine the fragmentation approach of CoBFS with a second level of striping when EC-DAP is used with CoARES, making this version of CoARESF more storage efficient at the replica hosts. A particular challenge of this integration is how the fragmentation approach should invoke reconfiguration operations, since CoBFS in [4] considered only static (non-reconfigurable) systems. We first describe CoARESF and then we present its (non-trivial) proof of correctness.

A. Description

We proceed with a description of the update, read and reconfig operations. From this point onward, we consider files, as an example of fragmented objects. To this respect, we view a file as a linked-list of data blocks. Here, the first block, i.e., the genesis block $b_0$, is a special type of a block that contains specific file information (such as the number of blocks, etc); see [4] for more details.

Update Operation (Fig. 2). The update operation spans two main modules: (i) the Fragmentation Module (FM), and (ii) the Distributed Shared Memory Module (DSMM). The FM uses a Block Identification (BI) module, which draws ideas from the RSYNC (Remote Sync) algorithm [26]. The BI includes three main modules, the Block Division, the Block Matching and Block Updates.

1) Block Division: The BI splits a given file $f$ into data blocks based on its contents, using rabin fingerprints [24]. BI has to match each hash, generated by the rabin fingerprint from the previous step, to a block identifier.

2) Block Matching: At first, BI uses a string matching algorithm [9] to find the differences between the new hashes and the old hashes in the form of four statuses: (i) equality, (ii) modified, (iii) inserted, (iv) deleted.

3) Block Updates: Based on the hash statuses, the blocks of the fragmented object are updated. In the case of equality, no operation is performed. In case of modification, an update operation is then performed to modify the data of the block. If new hashes are inserted after the hash of a block, then an update operation is performed to create the new blocks after that. The deleted one is treated as a modification that sets an empty value.

Subsequently, the FM uses the DSMM as an external service to execute the block update operations on the shared memory. As we already mentioned, we use CoARES as storage which is based on the $(n,k)$-Reed-Solomon code. It splits the value $v$ of a block into $k$ elements and then creates $n$ coded elements, and stores one coded element per server.

Read Operation (Fig. 3). When the system receives a read request from a client, the FM issues a series of read operations on the file’s blocks, starting from the genesis block and proceeding to the last block by following the next block ids. As blocks are retrieved, they are assembled in a file.

As in the case of the update operation, the read executes the block read operations on the shared memory. CoARES regenerates the value of a block using data from parity disks and surviving data disks.

Reconfig Operation. The specification of reconfig on the DSS is given in Algorithm 2, while the specification of reconfig on a file (fragmented object) is given in Algorithm 3. When the system receives a reconfig request from a client, the FM issues a series of reconfig operations on the file’s blocks, starting from the genesis block and proceeding to the last block by following the next block ids (Algorithm 3). The reconfig operation executes the block reconfig operations on the shared memory (Algorithm 2) using dsmm-reconfig operations.

As shown in Theorem 14, the blocks’ sequence of a fragmented object remains connected despite the existence of concurrent read/write and reconfiguration operations.

Algorithm 2 DSM Module: Operations on a coverable block object $b$ at client $p$

1: function dsmm-reconfig(c)$_b,p$
2: $b$.reconfig($c$)
3: end function

Algorithm 3 Fragmentation Module: BI and Operations on a file $f$ at client $p$

1: State Variables:
2: $L_f$ a linked-list of blocks, initially $(b_0)$;
3: function fm-reconfig(c)$_f,p$
4: $b ← val(b_0).ptr$
5: $L_f ← (b_0)$
6: while $b$ not NULL do
7: dsmm-reconfig(c)$_b,p$
8: $b ← val(b).ptr$
9: end while
10: end function

B. Correctness of CoARESF

When a reconfig($c$) operation is invoked in ARES, a reconfiguration client is requesting to change the configuration of the servers hosting the single R/W object. In the case of a file (fragmented object) $f$, which is composed of multiple blocks, the fragmentation manager attempts to introduce the new configuration for every block in $f$. To this end, CoARESF, as presented in Algorithm 3 issues a dsmm-reconfig($c$)$_b,p$ operation for each block $b_i$ in $f$. Concurrent write operations may introduce new blocks in the same file. So, how can we ensure that any new value of the blocks are propagated in any recently introduced configuration? In the rest of this section we show that fragmented coverability (see Section II) cannot be violated.

Before we prove any lemmas, we first state a claim that follows directly from the algorithm.

Claim 10. For any block $b \neq b_0$, where $b_0$ the genesis block, created by an fm-update operation, it is initialized with a configuration sequence $cseq_b = cseq_0$, where $cseq_0$ is the initial configuration.

Notice that we assume that a single quorum remains correct in $cseq_0$ at any point in the execution. This may change in practical settings by having an external service to maintain
and distribute the latest cseq that will be used in a created block.

We begin with a lemma that states that for any block in the list obtained by a read operation, there is a successful update operation that wrote this block.

**Lemma 11.** In any execution ξ of COARES, if ρ is a fm-read operation returns a list L, then for any block b ∈ L, there exists a successful fm-update(∗) operation that either precedes or is concurrent to ρ

**Proof.** This lemma follows the proof of Lemma 4 presented in [3].

In the following lemma we show that a reconfiguration moves a version of the object larger than any version written by a preceding write operation to the installed configuration.

**Lemma 12.** Suppose that ρ is a dsmm-reconfig(c2) operation and ω a successful cvr-write(v) operation that changes the version of b to ver, s.t. ω → ρ in execution ξ of COARES. Then ρ invokes c2.put-data(⟨ver′, ∗⟩) in c2, s.t. ver′ ≥ ver.

**Proof.** Let cseqω be the last configuration sequence returned by the read-config action at ω (Alg. 1:25), and cseqρ the configuration sequence returned by the first read-config action at ρ (see Alg. 2:8 in [10]). By the prefix property of the reconfiguration protocol, cseqω will be a prefix of cseqρ.

Let cℓ the last configuration in cseqω, and c1 the last finalized configuration in cseqρ. There are two cases to examine: (i) c1 appears before cℓ in cseqρ, and (ii) c1 appears after cℓ in cseqρ.

If (i) is the case then during the update-config action, ρ will perform a cℓ.get-data() action. By term C1 in Property [1] the cℓ.get-data() will return a version ver′ ≥ ver. Since the ρ function will execute c2.put-data(⟨ver′, ∗⟩), s.t. ver′ is the maximum discovered version, then ver′ ≥ ver′ ≥ ver.

In case (ii) it follows that the reconfiguration operation that proposed c1 has finalized the configuration. So either that reconfiguration operation moved a version ver′′ of b s.t. ver′′ ≥ ver in the same way as described in case (i) in c1, or the write operation would observe c1 during a read-config action. In the latter case c1 will appear in cseqω and ω will invoke a cℓ.put-data(⟨ver, ∗⟩) s.t. either cℓ = c1 or cℓ a configuration that appears after c1 in cseqω. Since c1 is the last finalized configuration in cseqρ, then in any of the cases described ρ will invoke a cℓ.get-data(). Thus, it will discover and put in c2 a version ver′ ≥ ver completing our proof.

Next we need to show that any sequence returned by any read operation is connected, despite any reconfiguration operations that may be executed concurrently.

**Lemma 13.** In any execution ξ of COARES, if fm-read operation f is a read operation on f that returns a list of blocks L = {b0, b1, . . . , bn}, then it must be the case that (i) b0.ptr = b1, (ii) bi.ptr = bi+1, for i ∈ [1, n − 1], and (iii) bn.ptr = ⊥.

**Proof.** Assume by contradiction that there exist some bi ∈ L, s.t. val(bi).ptr = bi+1 or val(bi).ptr = ¯0. By Lemma 11 a block bi may appear in the list returned by a read operation only if it was created by a successful update operation, say π = update(b, D) where D = ⟨D0, . . . , Dk⟩ and B = {b1, . . . , bk} be the set of k − 1 blocks created in π, with bi ∈ B. Let us assume w.l.o.g. that all those blocks appear in L as written by π (i.e., without any other blocks between any pair of them).

By the design of the algorithm π generates a single linked path from b to bk, by pointing b to b1 and each bj to bj+1, for 1 ≤ j < k. Block bk points to the block pointed by b at the invocation of π, say b′. So there exists a path b → b1 → . . . → bk that also leads to b1. According again to the algorithm, bj+1 ∈ B is created and written before bj, for q ≤ j < k. So when the bj.cvr-write is invoked, the operation bj+1.cvr-write has already been completed, and thus when b is written successfully all the blocks in the path are written successfully as well.

By the prefix property of the reconfiguration protocol it follows that for each bj written by π, ρ will observe a configuration sequence bj.cseqπ, s.t. bj.cseqπ is a prefix of bj.cseqρ, and hence cπ appears in bj.cseqρ. If cπ appears after the last finalized configuration cℓ in bj.cseqρ, then the read operation will invoke cℓ.get-data() and by the coverability property and property C1, will obtain a version ver′ ≥ ver. In case cπ appears before cℓ then a new configuration was invoked after or concurrently to π and then by Lemma 12 it follows that the version of b in cℓ is again ver′ ≥ ver. So we need to examine the following three cases for bj: (i) bj is b, (ii) bj is bk, and (iii) bj is one of the blocks bj, for 1 ≤ j < k.
Case iii: If $b_i$ is the block $b$ then we should examine if $b_i$.ptr $\neq b_1$. Let ver the version of $b$ written by $\pi$ and ver' the version of $b$ as retrieved by $\rho$. If ver = ver' then $\rho$ retrieved the block written by $\omega$ as the versions by Lemma 9 are unique. Thus, $b_i$.ptr = $b_1$ in this case contradicting our assumption.

In case ver' $>$ ver then there should be a successful update operation $\omega'$ that written block $b$ with ver'. There are two cases to consider based on whether $\omega'$ introduced new blocks or not. If not then the $b$.ptr = $b_1$ contradicting our assumption.

If it introduced a new list of blocks $\{b'_1, \ldots, b'_k\}$, then it should have written those blocks before writing $b$. In that case $\rho$ would observe $b$.ptr = $b'_i$ and $b'_i$ would have been part of $L$ which is not the case as the next block from $b$ in $L$ is $b_1$, leading to contradiction.

Case ii: The case (ii) can be proven in the same way as case (i) for each block $b_j$, for $1 \leq j < k$.

Case iii: If now $b_1 = b_1$, then we should examine if $b_1$.ptr $\neq b'_1$. Since $b$ was pointing to $b'$ at the invocation of $\pi$ then $b'$ was either (i) created during the update operation that also created $b$, or (ii) was created before $b$. In both cases $b'$ was written before $b$.

In case (i) by Lemma 11, the update operation that created $b'$ was successful and thus $b'$ must be created as well. In case (ii) it follows that $b'$ is the last inserted block of an update and is assigned to point to $b'$. Since no block is deleted, then $b'$ remains in $L$ when $b_1$ is created and thus $b_1$ points to an existing block. Furthermore, since $\pi$ was successful, then it successfully written $b$ and hence only the blocks in $B$ were inserted between $b$ and $b'$ at the response of $\pi$. In case the version of $b_1$ was ver' and larger than the version written on $b_k$ by $\pi$ then either $b_k$ was not extended and contains new data, or the new block is impossible as $L$ should have included the blocks extending $b_k$. So $b'$ must be the next block after $b_1$ in $L$ at the response of $\pi$ and there is a path between $b$ and $b'$. This completes the proof.

We conclude with the main result of this section.

**Theorem 14.** COARESF implements an atomic coverable fragmented object.

**Proof.** By the correctness proof in Section IV-B follows that every block operation in COARESF satisfies atomic coverability and together with Lemma 13 which shows the connectivity of blocks, it follows that COARESF implements a coverable fragmented object satisfying the properties of fragmented coverability as defined in Section II.

VI. EC-DAP OPTIMIZATION

In this section, we present an optimization in the implementation of the erasure coded DAP. EC-DAP, to reduce the operational latency of the read/write operations in DSMM layer. As we show in this section, this optimized EC-DAP, which we refer to as EC-DAPopt, satisfies all the items in Property 1 and thus can be used by any algorithm that utilizes the DAPs, like any variant of AREES. We first present the optimization and then prove its correctness.

A. Description

The main idea of the optimization stems from the work [4] which avoids unnecessary object transmissions between the clients and the servers that host the replicas.

In summary, we apply the following optimization: in the get-data primitive, each server sends only the tag-value pairs with a larger or equal tag than the client’s tag. In the case where the client is a reader, it performs the put-data action (propagation phase), only if the maximum tag is higher than its local one. EC-DAPopt is presented in Algorithms 4 and 5. The main idea of the optimization stems from the work [4] which avoids unnecessary object transmissions between the clients and the servers that host the replicas.

We now proceed with the details of the optimization. Note that the get-data primitive remains the same as the original, that is, the client discovers the highest tag among the servers’ replies in c.Servers and returns it.

**Primitive** c.get-data(): A client, during the execution of a c.get-data() primitive, queries all the servers in c.Servers for their List, and awaits responses from $\lceil \frac{n+k}{2} \rceil$ servers. Each server generates a new list (List') where it adds every (tag, coded-element) from the List, if the tag is higher than the c.tag of the client and the (tag, ⊥) if the tag is equal to c.tag; otherwise it does not add the pair, as the client already has a newer version. Once the client receives Lists from $\lceil \frac{n+k}{2} \rceil$ servers, it selects the highest tag $t$, such that:

(i) its corresponding value $v$ is decodable from the coded elements in the lists; and

(ii) $t$ is the highest tag seen from the responses of at least $k$ Lists (see lines Alg. 4:6–8) and returns the pair $(t, v)$. Note that in the case where any of the above conditions is not satisfied, the corresponding read operation does not complete. The main difference with the original code is that in the case where variable c.tag is the same as the highest decodable tag ($\max_{i=1}^{\text{max}} t_i$), the client already has the latest decodable version and does not need to decode it again (see line Alg. 4:10).

**Primitive** c.put-data((tw, v)): This primitive is executed only when the incoming $t_w$ is greater than c.tag (line Alg. 4:20). In this case, the client computes the coded elements and sends the pair $(t_w, \Phi_i(v))$ to each server $s_i \in c.Servers$. Also, the client has to update its state (c.tag and c.val). If the condition does not hold, the client does not perform any of the above, as it already has the latest version, and so the servers are up-to-date. When a server $s_i$ receives a message (PUT-DATA, $t_w$, $c_i$), it adds the pair in its local List and trims the pairs with the smallest tags exceeding the length $(\delta + 1)$ (see line Alg. 5:17).

**Remark.** Experimental results conducted on Emulab show that by using EC-DAPopt over EC-DAP we gain significant reductions especially on read latencies, which concern the majority of operations in practical systems (see Fig. 4 in
Algorithm 4 EC-DAPopt implementation

at each process \( p_i \in I \)
1. procedure \( c \cdot \text{get-data}() \)
2. \( \text{send} \langle \text{QUERY-LIST}, c \cdot \text{tag} \rangle \) to each \( s \in c \cdot \text{Servers} \)
3. if \( p_i \) receives \( List_s \) from each \( s \in S_g \)
4. \( \tau \leftarrow \text{s.t.} |S_g| = \frac{n \cdot k}{2} \) and \( S_g \subseteq c \cdot \text{Servers} \)
5. \( Tags_{\text{dec}}^c \) = set of tags that appears in \( S_g \) lists
6. \( \text{dec} = \max Tags_{\text{dec}}^c \)
7. if \( c \cdot \text{tag} \leq \text{dec} \)
8. \( t \leftarrow \text{dec} \)
9. \( v \leftarrow \text{decode value for } t \)
10. return \( t,v \)
11. else if \( Tags_{\text{dec}}^c \neq \) then
12. \( t \leftarrow \text{decode} \) max
13. \( v \leftarrow \text{decode value for } t \)
14. return \( t,v \)

Algorithm 5 The response protocols at any server \( s_i \in S \) in EC-DAPopt for client requests.

at each server \( s_i \in S \) in configuration \( c_k \)
1. procedure \( c \cdot \text{put-data}(\langle \tau, v \rangle) \)
2. \( \text{send} \langle \text{PUT-DATA}, \tau, v \rangle \) to each \( s \in c \cdot \text{Servers} \)
3. if \( c \cdot \text{tag} \geq \text{dec} \)
4. \( \text{do} \langle \tau, v \rangle \) to each \( s \in c \cdot \text{Servers} \)
5. \( \text{until } p_i \) receives \( ACK \) from \( \frac{n \cdot k}{2} \) servers in \( c \cdot \text{Servers} \)
6. \( c \cdot \text{val} \leftarrow v \)
7. return \( c \cdot \text{val} \)

Section VII. The great benefits are observed especially in the fragmented variants of the algorithm and when the objects are large, as read operations avoid the transmission of many unchanged blocks.

B. Correctness of EC-DAPopt

To prove the correctness of EC-DAPopt, we need to show that it is safe, i.e., it ensures the necessary Property 1 and live, i.e., it allows each operation to terminate. In the following proof, we will not refer to the get-tag access primitive that the EC-DAP algorithm uses \[22\], as the optimization has no effect on this operation, so it should preserve safety as shown in \[10\].

For the following proofs we fix the configuration to \( c \) as it suffices that the DAPs preserve Property 1 in any single configuration. Also we assume an \([n,k] \) MDS code, \([c \cdot \text{Servers}] = n \) of which no more than \( \frac{n \cdot k}{2} \) may crash, and that \( \delta \) is the maximum number of put-data operations concurrent with any get-data operation.

We first prove Property 1-C2 as it is later being used to prove Property 1-C1.

Lemma 15 (C2). Let \( \xi \) be an execution of an algorithm \( A \) that uses the EC-DAPopt. If \( \phi \) is a \( c \cdot \text{get-data}() \) that returns \( \langle \tau_v, v_\phi \rangle \in T \times V \), then there exists \( \pi \) such that \( \pi \) is a \( c \cdot \text{put-data}(\langle \tau, v_\pi \rangle) \) and \( \phi \) did not complete before the invocation of \( \pi \). If no such \( \pi \) exists in \( \xi \), then \( \langle \tau_v, v_\pi \rangle \) is equal to \( \langle t_0, v_0 \rangle \).

Proof. It is clear that the proof of property C2 of EC-DAPopt is identical with that of EC-DAP. This happens as the initial value of the \( List \) variable in each servers \( s \in S \) is still \( \{ \langle t_0, \Phi_k(v_0) \rangle \} \), and the new tags are still added to the \( List \) only via put-data operations. Thus, each server during a get-data operation includes only written tag-value pairs from the \( List \) to the \( List' \).

Lemma 16 (C1). Let \( \xi \) be an execution of an algorithm \( A \) that uses the EC-DAPopt. If \( \phi \) is a \( c \cdot \text{put-data}(\langle \tau_\phi, v_\phi \rangle) \), for \( c \in C \), \( \langle \tau_\phi, v_\phi \rangle \in T \times V \), and \( \pi \) is a \( c \cdot \text{get-data}() \) that returns \( \langle \tau_\pi, v_\pi \rangle \in T \times V \) and \( \phi \rightarrow \pi \in \xi \), then \( \tau_\pi \geq \tau_\phi \).

Proof. Let \( p_\phi \) and \( p_\pi \) denote the processes that invokes \( \phi \) and \( \pi \) in \( \xi \). Let \( S_\phi \subseteq S \) denote the set of \( \lfloor \frac{n \cdot k}{2} \rfloor \) servers that responds to \( p_\phi \), during \( \phi \), and by \( S_\pi \) the set of \( \lfloor \frac{n \cdot k}{2} \rfloor \) servers that responds to \( p_\pi \), during \( \pi \).

Per Alg. \[5\], every server \( s \in S_\phi \), inserts the tag-value pair received in its local \( List \). Note that once a tag is added to \( List \), its associated tag-value pair will be removed only when the \( List \) exceeds the length \( (\delta + 1) \) and the tag is the smallest in the \( List \) (Alg. \[5\]12–14).

When replying to \( \pi \), each server in \( S_\pi \) includes a tag in \( List' \), only if the tag is larger or equal to the tag associated to the last value decoded by \( p_\pi \) (lines Alg. \[5\]19). Notice that as \( |S_\phi| = |S_\pi| = \lfloor \frac{n \cdot k}{2} \rfloor \), the servers in \( |S_\phi \cap S_\pi| \geq k \) reply to both \( \pi \) and \( \phi \). So there are two cases to examine: (a) the pair \( \langle \tau_\phi, v_\phi \rangle \in Lists' \) of at least \( k \) servers \( S_\phi \cap S_\pi \) replied to \( \pi \), and (b) the \( \langle \tau_\phi, v_\phi \rangle \) appeared in fewer than \( k \) servers in \( S_\pi \).

Case a: In the first case, since \( \pi \) discovered \( \tau_\phi \) in at least \( k \) servers it follows by the algorithm that the value associated with \( \tau_\phi \) will be decodable. Hence \( t_{\text{max}}^\text{dec} \leq \tau_\phi \) and \( \tau_\pi \geq \tau_\phi \).
Case b: In this case \( \tau_0 \) was discovered in less than \( k \) servers in \( S_\pi \). Let \( \tau_y \) denote the last tag returned by \( p_\pi \). We can break this case into two subcases: (i) \( \tau_y > \tau_0 \), and (ii) \( \tau_y \leq \tau_0 \).

In case (i), no \( s \in S_\pi \) included \( \tau_0 \) in \( List_s' \) before replying to \( \pi \). By Lemma 15 the c.put-data(\( \langle \tau_y, s \rangle \)) was invoked before the completion of the *get-data() operation from \( p_\pi \) that returned \( \tau_y \). It is also true that \( p_\pi \) discovered \( \langle \tau_y, s \rangle \) in more than \( k \) servers since it managed to decode the value. Therefore, in this case \( \tau_{\max} \geq \tau_y \) and thus \( \tau_0 > \tau_y \).

In case (ii), a server \( s \in S_\pi \) will not include \( \tau_0 \) iff \( |List_s'| = \delta + 1 \), and therefore the local \( List \) of \( s \) removed \( \tau_0 \) as the smallest tag in the list. According to our assumption though, no more than \( \delta \) put-data operations may be concurrent with a get-data operation. Thus, at least one of the put-data operations that wrote a tag \( \tau' \in Lists_s' \) must have completed before \( \pi \). Since \( \tau' \) is also written in \( |S'| = \delta + k \) servers then \( |S_\pi \cap S'| \geq k \) and hence \( \pi \) will be able to decode the value associated to \( \tau' \), and hence \( \tau_{\max} \geq \tau_y \) and \( \tau_0 > \tau_y \), completing the proof of this lemma.

\[ \text{Theorem 17 (Safety). Let } \xi \text{ be an execution of an algorithm } \]
\[ \text{A that contains a set } \Pi \text{ of complete get-data and put-data operations of Algorithm 4. Then every pair of operations } \]
\[ \phi, \pi \in \Pi \text{ satisfy Property } \]

\[ \text{Proof. Follows directly from Lemmas 15 and 16.} \]

Liveness requires that any put-data and get-data operation defined by EC-DAPOpt terminates. The following theorem captures the main result of this section.

\[ \text{Theorem 18 (Liveness). Let } \xi \text{ be an execution of an algorithm } \]
\[ \text{A that utilises the EC-DAPOpt. Then any put-data or get-data operation } \]
\[ \pi \text{ invoked in } \xi \text{ will eventually terminate.} \]

\[ \text{Proof. Given that no more than } \frac{n-k}{2} \text{ servers may fail, then from Algorithm 4 (lines Alg. 17, 19, 26), it is easy to see that there are at least } \frac{n+k}{2} \text{ servers that remain correct and reply to the put-data operation. Thus, any put-data operation completes.} \]

Now we prove the liveness property of any get-data operation \( \pi \). Let \( p_\omega \) and \( p_\omega \) be the processes that invokes the put-data operation \( \omega \) and get-data operation \( \omega \). Let \( S_\omega \) be the set of \( \left\lfloor \frac{n+k}{2} \right\rfloor \) servers that responds to \( p_\omega \), in the put-data operations, in \( \omega \). Let \( S_\omega \) be the set of \( \left\lfloor \frac{n+k}{2} \right\rfloor \) servers that responds to \( p_\omega \) during the get-data step of \( \pi \). Note that in \( \xi \) at the point execution \( T_1 \), just before the execution of \( \pi \), none of the write operations in \( \Lambda \) is complete. Let \( T_2 \) denote the earliest point of time when \( p_\omega \) receives all the \( \left\lfloor \frac{n+k}{2} \right\rfloor \) responses. Also, the set \( \Lambda \) includes all the put-data operations that starts before \( T_3 \) such that \( tag(\lambda) > tag(\omega) \). Observe that, by algorithm design, the coded-elements corresponding to \( \omega \) are garbage-collected from the List variable of a server only if more than \( \delta \) higher tags are introduced by subsequent writes into the server. Since the number of concurrent writes \( |\Lambda| \), s.t. \( \delta > |\Lambda| \) the corresponding value of tag \( \omega \) is not garbage collected in \( \xi \), at least until execution point \( T_2 \) in any of the servers in \( S_\omega \). Therefore, during the execution fragment between the execution points \( T_1 \) and \( T_2 \) of the execution \( \xi \), the tag and coded-element pair is present in the List variable of every server in \( S_\omega \) that is active. As a result, the tag and coded-element pairs, \( (\omega, \Phi_\omega(v_\omega)) \) exists in the List received from any \( s \in S_\omega \cap S_\pi \) during operation \( \pi \). Note that since \( |S_\omega| = |S_\pi| = \left\lceil \frac{n+k}{2} \right\rceil \) hence \( |S_\omega \cap S_\pi| \geq k \) and hence \( t_\omega \in Tag_{\omega_{\max}}^{\geq k} \), the set of decode-able tag, i.e., the value \( v_\omega \) can be decoded by \( p_\pi \) in \( \pi \), which demonstrates that \( Tag_{\omega_{\max}}^{\geq k} \neq \emptyset \).

Next we want to argue that \( t_{\max} \) is the maximum tag that \( \pi \) discovers via a contradiction: we assume a tag \( t_{\max} \) which is the maximum tag \( \pi \) discovers, but it is not decode-able, i.e., \( t_{\max} \notin Tag_{\omega_{\max}}^{\geq k} \) and \( t_{\max} > t_{\omega_{\max}} \). Let \( S_k^\pi \subseteq S \) be any subset of \( k \) servers that responds with \( t_{\max} \) in their \( List' \) variables to \( p_\pi \). Note that since \( k > n/3 \) hence \( |S_\omega \cap S_k^\pi| \geq \left\lfloor \frac{n+k}{2} \right\rfloor + \left\lfloor \frac{n+1}{3} \right\rfloor \geq 1 \), i.e., \( S_\omega \cap S_k^\pi \neq \emptyset \). Then \( t_{\max} \) must be in some servers in \( S_\omega \) at \( T_2 \) and since \( t_{\max} > t_{\omega_{\max}} \). Now since \( |\Lambda| < \delta \) hence \( (t_{\max}, \Phi_\omega(v_{\max})) \) cannot be removed from any server at \( T_2 \) because there are not enough concurrent write operations (i.e., writes in \( \Lambda \)) to garbage-collect the coded-elements corresponding to tag \( t_{\max} \). Also since \( \pi \) cannot have a local tag larger than \( t_{\max} \) according to the lines Alg. 5/6/9 each server in \( S_\pi \) includes the \( t_{\max} \) in its replies. In that case, \( t_{\max} \) must be in \( Tag_{\omega_{\max}}^{\geq k} \), a contradiction.

VII. EXPERIMENTAL EVALUATION

Distributed systems are often evaluated on an emulation or an overlay testbed. Emulation testbeds give users full control over the host and network environments, their experiments are repeatable, but their network conditions are artificial. The environmental conditions of overlay testbeds are not repeatable and provide less control over the experiment, however they provide real network conditions and thus provide better insight on the performance of the algorithms in a real deployment. We used Emulab [11] as an emulation testbed and Amazon Web Services (AWS) EC2 [8] as an overlay testbed.

A. Evaluated Algorithms and Experimental Setup

Evaluated Algorithms. We have implemented and evaluated the performance of the following algorithms:

- **COABD.** This is the coverable version of the traditional, static ABD algorithm [1], [20], as presented in [21]. It will be used as an overall baseline.
- **COABDF.** This is the version of COABD that provides fragmented coverability, as presented in [4]. It can be considered as a baseline algorithm of the CBFS framework.
- **CORESABD.** This is a version of CORES that uses the ABD-DAP implementation [22] (cf. Section III). It can be considered as the dynamic (reconfigurable) version of COABD.
- **CORESABDF.** This is CORESDF together with the ABD-DAP implementation, i.e., it is the fragmented version of CORESABD.
- **CORESEC.** This is a version of CORES (see Section IV) that uses the EC-DAPOpt implementation (see Section VI).
Node Types: During the experiments, we use four distinct types of nodes, writers, readers, reconfigurers and servers. Their main role is listed below:

- **writer** $w \in W \subseteq I$: a client that sends write requests to all servers and waits for a quorum of the servers to reply.
- **reader** $r \in R \subseteq I$: a client that sends read requests to servers and waits for a quorum of the servers to reply.
- **reconfigurer** $g \in G \subseteq I$: a client that sends reconfiguration requests to servers and waits for a quorum of the servers to reply. This type of node is used only in any variant of ARES algorithm.
- **server** $s \in S$: a server listens for read and write and reconfiguration requests, it updates its object replica according to the DSMM implementation and replies to the process that originated the request.

### Performance Metric

The metric for evaluating the algorithms is operational latency. This includes both communication and computational delays. The operational latency is computed as the average of all clients’ average operational latencies. The performance of CoABD shown in the Emulab results can be used as a reference point in the following experiments since the rest algorithms combine ideas from it.

### Distributed Experimental Setup on Emulab

All physical nodes were placed on a single LAN using a DropTail queue without delay or packet loss. We used nodes with one 2.4 GHz 64-bit Quad Core Xeon E5530 “Nehalem” processor and 12 GB RAM. Each physical machine runs one server or client process. This guarantees a fair communication delay between a client and a server node. We have an extra physical node, the controller, which orchestrates the experiments. A client’s physical machine has one Daemon that listens for its requests.

### Distributed Experimental Setup on AWS

For the File Sizes and Block Sizes experiments, we create a cluster with 8 node instances. All of them have the same specifications, their type is t2.medium with 4 GB RAM, 2 vCPUs and 20 GB storage. For the Scalability experiments, we create a cluster with 11 node instances. Ten of them have the same specifications, their type is t2.small with 2 GB RAM, 1 vCPU and 20 GB storage, and one is of type t2.medium. In all experiments one medium node has also the role of controller to orchestrate the experiments. In order to guarantee a fair communication delay between a client and a server node, we placed at most one server process on each physical machine. Each instance with clients has one Daemon to listen for clients’ requests.

We used an external implementation of Raft [23] consensus algorithms, which was used for the service reconfiguration and was deployed on top of small RPi devices. Small devices introduced further delays in the system, reducing the speed of reconfigurations and creating harsh conditions for longer periods in the service.

For the deployment and remote execution of the experimental tasks on both Emulab and AWS, the controller used Ansible [3], a tool to automate different IT tasks. More specifically, we used Ansible Playbooks, scripts written in YAML format. These scripts get pushed to target nodes, do their work (over SSH) and get removed when finished.

### B. Overview of the experiments

#### Node Types

During the experiments, we use four distinct types of nodes, writers, readers, reconfigurers and servers. Their main role is listed below:

- **writer** $w \in W \subseteq I$: a client that sends write requests to all servers and waits for a quorum of the servers to reply.
- **reader** $r \in R \subseteq I$: a client that sends read requests to servers and waits for a quorum of the servers to reply.
- **reconfigurer** $g \in G \subseteq I$: a client that sends reconfiguration requests to servers and waits for a quorum of the servers to reply. This type of node is used only in any variant of ARES algorithm.
- **server** $s \in S$: a server listens for read and write and reconfiguration requests, it updates its object replica according to the DSMM implementation and replies to the process that originated the request.

### Performance Metric

The metric for evaluating the algorithms is operational latency. This includes both communication and computational delays. The operational latency is computed as the average of all clients’ average operational latencies. The performance of CoABD shown in the Emulab results can be used as a reference point in the following experiments since the rest algorithms combine ideas from it.

The Emulab results are compiled as averages over five samples per each scenario. However, the AWS results are compiled as averages over three samples for the Scalability scenario, while the rest scenarios run only once.

#### C. Experimental Scenarios

Here we describe the scenarios we constructed and the settings for each of them. We executed every experiment of each scenario in two steps. First, we performed a boot-up phase where a single client writes a file of a specific initial size and the other readers and writers are informed about it. Second, operations write and read data to this file concurrently and we have measured the performance under various scenarios. During all the experiments, as the writers kept updating the file, its size increased (we generate text files with random bytes strings).

### Parameters of algorithms

The quorum size of the EC-based algorithms is $\left\lceil \frac{n+k}{2} \right\rceil$, while the quorum size of the ABD-based algorithms is $\left\lceil \frac{n}{2} \right\rceil + 1$. The parameter $n$ is the total number of servers, $k$ is the number of encoded data fragments, and $m$ is the number of parity fragments, i.e., $n - k$. In relation to EC-based algorithms, we can conclude that the parameter $k$ is directly proportional to the quorum size. But as the value of $k$ and quorum size increase, the size of coded elements decreases. Also, a high number of $k$ and consequently a small number of $m$ means less redundancy with the system tolerating fewer failures. When $k = 1$ we essentially converge to replication. Parameter $\delta$ in EC-based algorithms is the maximum number of concurrent put-data operations, i.e., the number of writers.

### Distributed Experiments

For the distributed experiments (in both testbeds) we use a stochastic invocation scheme in which readers and writers pick a random time uniformly distributed (discrete) between intervals to invoke their next operations. Respectively the intervals are $[1...rInt]$ and $[1...wInt]$, where $rInt, wInt = 3$ sec. If there is a reconfigurer, it invokes its next operation every 15 sec and performs a total of 5 reconfigurations.

We present three main types of scenarios:

- **Node Types**: During the experiments, we use four distinct types of nodes, writers, readers, reconfigurers and servers. Their main role is listed below:
- **writer** $w \in W \subseteq I$: a client that sends write requests to all servers and waits for a quorum of the servers to reply.
- **reader** $r \in R \subseteq I$: a client that sends read requests to servers and waits for a quorum of the servers to reply.
- **reconfigurer** $g \in G \subseteq I$: a client that sends reconfiguration requests to servers and waits for a quorum of the servers to reply. This type of node is used only in any variant of ARES algorithm.
- **server** $s \in S$: a server listens for read and write and reconfiguration requests, it updates its object replica according to the DSMM implementation and replies to the process that originated the request.

### Performance Metric

The metric for evaluating the algorithms is operational latency. This includes both communication and computational delays. The operational latency is computed as the average of all clients’ average operational latencies. The performance of CoABD shown in the Emulab results can be used as a reference point in the following experiments since the rest algorithms combine ideas from it. The Emulab results are compiled as averages over five samples per each scenario. However, the AWS results are compiled as averages over three samples for the Scalability scenario, while the rest scenarios run only once.

#### C. Experimental Scenarios

Here we describe the scenarios we constructed and the settings for each of them. We executed every experiment of each scenario in two steps. First, we performed a boot-up phase where a single client writes a file of a specific initial size and the other readers and writers are informed about it. Second, operations write and read data to this file concurrently and we have measured the performance under various scenarios. During all the experiments, as the writers kept updating the file, its size increased (we generate text files with random bytes strings).

### Parameters of algorithms

The quorum size of the EC-based algorithms is $\left\lceil \frac{n+k}{2} \right\rceil$, while the quorum size of the ABD-based algorithms is $\left\lceil \frac{n}{2} \right\rceil + 1$. The parameter $n$ is the total number of servers, $k$ is the number of encoded data fragments, and $m$ is the number of parity fragments, i.e., $n - k$. In relation to EC-based algorithms, we can conclude that the parameter $k$ is directly proportional to the quorum size. But as the value of $k$ and quorum size increase, the size of coded elements decreases. Also, a high number of $k$ and consequently a small number of $m$ means less redundancy with the system tolerating fewer failures. When $k = 1$ we essentially converge to replication. Parameter $\delta$ in EC-based algorithms is the maximum number of concurrent put-data operations, i.e., the number of writers.

### Distributed Experiments

For the distributed experiments (in both testbeds) we use a stochastic invocation scheme in which readers and writers pick a random time uniformly distributed (discrete) between intervals to invoke their next operations. Respectively the intervals are $[1...rInt]$ and $[1...wInt]$, where $rInt, wInt = 3$ sec. If there is a reconfigurer, it invokes its next operation every 15 sec and performs a total of 5 reconfigurations.

We present three main types of scenarios:
• Performance VS. Initial File Sizes: examine performance when using different initial file sizes.
• Performance VS. Scalability of nodes under concurrency: examine performance as the number of service participants increases.
• Performance VS. Block Sizes: examine performance under different block sizes (only for fragmented algorithms).

D. Performance VS. Initial File Sizes

The first scenario is made to measure the performance of algorithms when the writers update a file whose size gradually increases. We varied the \( f_{\text{size}} \) from 1MB to 512MB by doubling the file size in each experimental run. The performance of some experiments is missing as the non-fragmented algorithms crashed when testing larger file sizes due to an out-of-memory error. The maximum, minimum and average block sizes (rabin fingerprints parameters) were set 1MB, 512kB and 512kB respectively.

a) Emulab parameters: We have 5 writers, 5 readers and 11 servers. We run twice the EC-based algorithms with different value of parity, one with \( m=1 \) and one with \( m=5 \). Thus, the quorum size of the EC-based algorithms with \( m=1 \) is \( \left\lceil \frac{11+10}{2} \right\rceil = 11 \), while the quorum size of EC-based algorithms with \( m=5 \) is \( \left\lceil \frac{11+6}{2} \right\rceil = 9 \). The quorum size of ABD-based algorithms is \( \left\lceil \frac{11}{2} \right\rceil + 1 = 6 \). In total, each writer performs 20 writes and each reader 20 reads.

b) AWS parameters: We have 1 writer, 1 reader and 6 servers. We run twice the EC-based algorithms with different value of parity, one with \( m=1 \) and one with \( m=4 \). Thus the quorum size of the EC-based algorithms with \( m=1 \) is 6, while the quorum size of EC-based algorithms with \( m=4 \) is 4. The quorum size of ABD-based algorithms is 4. In total, each writer performs 50 writes and each reader 50 reads.

We measure the read and write operation latencies for both original and the fragmented variant of algorithms; the results can be seen on Figs. 4 and 5. As shown in Figs. 4(a) and 5(a), the fragmented algorithms that use the FM achieve significantly smaller write latency, when the file size increases, which is a result of the block distribution strategy. In Fig. 4(a), the lines of fragmented algorithms are very close to each other. The fact that the CoARESECF with \( m=1 \) (Fig. 5(a)) at smaller file sizes does not benefit so much from the fragmentation, is because the client waits more responses for each block request compared to ABD-based algorithms with fragmentation. However, the update latency exhibited in non-fragmented algorithms appears to increase linearly with the file size. This was expected, since as the file size increases, it takes longer latency to update the whole file. Also, the successful file updates achieved by fragmented algorithms are significantly higher as the file size increases since the probability of two writes to collide on a single block decreases as the file size increases (Fig. 4(a)). On the contrary, the non-fragmented algorithms do not experience any improvement as it always manipulates the file as a whole.

The Block Identification (BI) computation latency contributes significantly to the increase of fragmented algorithms’ update latency in larger file sizes, as shown in Fig. 5(c). We have set the same parameters for the rabin fingerprints algorithm for all the initial file sizes, which may have favored some file sizes but burdened others.

As shown in Fig. 4(b), all the fragmented algorithms have smaller read latency than the non-fragmented ones. This happens since the readers in the shared memory level transmit only the contents of the blocks that have a newer version. While in the non-fragmented algorithms, the readers transmit the whole file each time a newer version of the file is discovered. This explains the increasing curve of non-fragmented compared to their counterpart with fragmentation.

On the contrary, the read latency of CoARES in the corresponding AWS experiment (Fig. 5(b)) has not improved with the fragmentation strategy. This is due to the fact that the AWS testbed provides real network conditions. The CoARESF read/write operation has at least two more rounds of communication to perform than CoABDF in order to read the configuration before each of the two phases. As we can see in Fig. 5(d), the read-config operations of CoARESABDF during a block read operation have a stable overhead in latency. Thus, when the FM module sends multiple read block requests, waiting each time for a reply, the client has this stable overhead for each block request. The average number of blocks read in each experiment is shown in the Fig. 5(b). It is also worth mentioning that the decoding of the read operation in EC-based algorithms is slower than the encoding of the write as it requires more computation. It would be interesting to examine whether the multiple read block requests in CoBFS could be sent in parallel, reducing the overall communication delays.

EC-based algorithms with \( m=5, k=6 \) in Emulab and with \( m=4, k=2 \) in AWS results in the generation of smaller number of data fragments and thus bigger sizes of fragments and higher redundancy, compared to EC-based algorithms with \( m=1 \). As a result, with a higher number of \( m \) (i.e. smaller \( k \)) we achieve higher levels of fault-tolerance, but with wasted storage efficiency. The write latency seems to be less affected by the number of \( m \) since the encoding is faster as it requires less computation.

In Figs. 4(a)(b), we can additionally observe the write and read latency of CoARESEC and CoARESECF (with \( m=5 \)) when EC-DAP is used instead of EC-DAPopt in the DSMM layer. Both algorithms, when using the optimization (i.e., EC-DAPopt) incur significant reductions on the read latency (in half), especially for large files. Furthermore, the write latency of CoARESEC is significantly reduced (in half); there is no much gain for the write latency of CoARESECF, which was expected since it is already very low due to fragmentation (the optimization was aiming the read latency anyway).

E. Performance VS. Scalability of nodes under concurrency

This scenario is constructed to compare the read, write and recon latency of the algorithms, as the number of service participants increases. In both Emulab and AWS, we varied the number of readers \( |R| \) and the number of writers \( |W| \) from 5 to 25, while the number of servers \( |S| \) varies from
3 to 11. In AWS, the clients and servers are distributed in a round-robin fashion. We calculate all possible combinations of readers, writers and servers where the number of readers or writers is kept to 5. In total, each writer performs 20 writes and each reader 20 reads. The size of the file used is 4 MB. The maximum, minimum and average block sizes were set to 1 MB, 512 kB and 512 kB respectively. For each number of servers, we set different parity for EC-based algorithms in order to achieve the same fault-tolerance with ABD-based algorithms in each case, except in the case of 3 servers (to avoid replication). With this however, the EC client has to wait for responses from a larger quorum size. The parity value of the EC-based algorithms is set to $m=1$ for 3 servers, $m=2$ for 5 servers, $m=3$ for 7 servers, $m=4$ for 9 servers and $m=5$ for 11 servers.

The results obtained in this scenario are presented in Fig. 6 and Fig. 7 for Emulab and AWS respectively. As expected, COARESE can has the lowest update latency among non-fragmented algorithms because of the striping level. Each object is divided into $k$ encoded fragments that reduce the communication latency (since it transfers less data over the network) and the storage utilization. The fragmented algorithms perform significantly better update latency than the non-fragmented ones, even when the number of writers increases (see Figs. 6(a), 7(a)). This is because the non-fragmented writer updates the whole file, while each fragmented writer updates a subset of blocks which are modified or created. We observe that the update operation latency in algorithms COABD and COARESABD increases even more as the number of servers increases, while the operation latency of COARESE decreases or stays the same (Figs. 6(c), 7(c)). This is because when increasing the number of servers, the quorum size grows.
but the message size decreases. Therefore, while both non-fragmented ABD-based algorithms and COARESEC wait for responses from more servers, COARESEC gains the advantage of decreased message size. However, when going from 7 to 9 servers, we find that there is a decrease in latency. This is due the choice of the parity value (parameter of EC-based algorithms) selected for 7 servers.

Due to the block allocation strategy in fragment algorithms, more data are successfully written (cf. Fig. 6(a), 6(b)), explaining the slower COARESF read operation (cf. Figs. 6(b), 7(b)).

We built four extra experiments in Emulab to verify the correctness of the variants of ARES when reconfigurations coexist with read/write operations. The four experiments differ in the way the reconfigurer works; three experiments are based on the way the reconfigurer chooses the next storage algorithm and one in which the reconfigurer changes concurrently the next storage algorithm and the quorum of servers. In these experiments the number of servers $|S|$ is fixed to 11 and there is one reconfigurer. All of the scenarios below are run for both COARES and COARESF.

- **Changing to the Same Reconfigurations**: We execute two separate runs, one for each $DAP$. We use only one reconfigurer which requests recon operations that lead to the same shared memory emulation and server nodes.
- **Changing Reconfigurations Randomly**: The reconfigurer chooses randomly between the two $DAP$s.
- **Changing Reconfigurations with different number of servers**: The reconfigurer switches between the two $DAP$s and at the same time chooses randomly the number of servers between $[3, 5, 7, 9, 11]$.

As we mentioned earlier, our choice of $k$ minimizes the coded fragment size but introduces bigger quorums and thus larger communication overhead. As a result, in smaller file sizes, ARES (either fragmented or not) may not benefit so much from the coding, bringing the delays of the COARESEC and COARESABD closer to each other (cf. Fig. 8). However, the read latency of COARESECDF is significantly lower than of COARESABDF. This is because the COARESECDF takes less time to transfer the blocks to the new configuration.

Fig. 9 illustrates the results of COARESF experiments with the random storage change. During the experiments, there are cases where a single read/write operation may access configurations that implement both ABD-DAP and EC-DAPopt, when concurrent with a recon operation.

The last scenario in Fig. 10 is constructed to show that the service is working without interruptions despite the existence of concurrent read/write and reconfiguration operations that may add/remove servers and switch the storage algorithm in the system. Also, we can observe that COARESF (Fig. 10(b)) has shorter update and read latencies than COARES (Fig. 10(a)).

**F. Performance VS. Block Sizes**

1) **Performance VS. Min/Avg Block Sizes**: We varied the minimum and average $b_{sizes}$ of fragmented algorithms from 8kB to 1MB. The size of the initial file used was set to
Fig. 8. Emulab results when Changing to the Same DAPs.

Fig. 9. Emulab results when Changing DAPs Randomly.

Fig. 10. Emulab results when Changing DAPs Alternately and Servers Randomly.

Fig. 11. Emulab results for Min/Avg Block Sizes’ experiments.

Fig. 12. AWS results for Min/Avg/Max Block Sizes’ experiments.
4 MB, while the maximum block size was set to 1 MB. In Emulab, each writer performs 20 writes and each reader 20 reads, whereas in AWS each writer performs 50 writes and each reader 50 reads.

a) **Emulab parameters:** We have 5 writers, 5 readers and 11 servers. The parity value of the EC-based algorithms is set to 1. Thus the quorum size of the EC-based algorithms is 11, while the quorum size of ABD-based algorithms is 4.

b) **AWS parameters:** We have 1 writer, 1 reader and 6 servers. The parity value of the EC-based algorithms is set to 1. Thus the quorum size of the EC-based algorithms is 6, while the quorum size of ABD-based algorithms is 4.

From Figs. 11(a), we can infer in general that when larger min/avg block sizes are used, the update latency reaches its highest values since larger blocks need to be transferred. However, too small min/avg block sizes lead to the generation of more new blocks during update operations, resulting in more update block operations, and hence slightly higher update latency. In Figs. 11(b), smaller block sizes require more read block operations to obtain the file’s value. As the minimum and average \( b_{\text{sizes}} \) increase, lower number of rather small blocks need to be read. Thus, further increase of the minimum and average \( b_{\text{sizes}} \) forces the decrease of the read latency, reaching a plateau in the graph. This means that the scenario finds optimal minimum and average \( b_{\text{sizes}} \) and increasing them does not give better (or worse) read latency. The corresponding AWS findings show similar trends.

2) **Performance VS. Min/Avg/Max Block Sizes:** We varied the minimum and average \( b_{\text{sizes}} \) from 2 MB to 64 MB and the maximum \( b_{\text{size}} \) from 4 MB to 1 GB. In Emulab and AWS, this scenario has the same settings as the prior block size scenario. In total, each writer performs 20 writes and each reader 20 reads. The size of the initial file used was set to 512 MB.

This scenario evaluates how the block size impacts the latencies when having a rather large file size. As all examined block sizes are enough to fit the text additions no new blocks are created. All the algorithms achieve the maximal update latency as the block size gets larger (Fig 12(a)). **CoARES** has the lower impact as block size increases mainly due to the extra level of striping. Similar behaviour has the read latency in Emulab. However, in real time conditions of AWS, the read latency of a higher number of relatively large blocks (Fig. 12(c)) has a significant impact on overall latency, resulting in a larger read latency (Fig. 12(b)).

**REFERENCES**

[1] Emulab network testbed. https://www.emulab.net/  
[2] M.K. Aguilera, I. Keidar, D. Malkhi, and A. Shraer. Dynamic atomic storage without consensus. In Proceedings of the 28th ACM symposium on Principles of distributed computing (PODC ’09), pages 17–27, New York, NY, USA, 2009. ACM.  
[3] Ansible. https://www.ansible.com/overview/how-ansible-works.  
[4] A.F. Anta, C. Georgiou, T. Hadjistasi, E. Stavrakis, and A. Trigeorgi. Fragmented Object : Boosting Concurreny of Shared Large Objects. In Proc.of SIROCCO, pages 1–18, 2021.  
[5] Antonio Fernández Anta, Chryssis Georgiou, Theophanis Hadjistasi, Nicolas Nicolaou, Efstathios Stavrakis, and Andria Trigeorgi. Fragmented objects: Boosting concurrency of sharedlarge objects. CoRR, abs/2102.12786, 2021.  
[6] H. Attiya. Robust Simulation of Shared Memory: 20 Years After. Bulletin of the EATCS, 100:99–114, 2010.  
[7] H. Attiya, A. Bar-Noy, and D. Dolev. Sharing Memory Robustly in Message-Passing Systems. Journal of the ACM (JACM), 42(1):124–142, 1995.  
[8] AWS EC2. https://aws.amazon.com/ec2/.  
[9] Paul Black. Ratcliff pattern recognition. Dictionary of Algorithms and Data Structures, 2021.  
[10] Viveck R. Cadambe, Nicolas C. Nicolaou, Kishori M. Konwar, N. Prakash, Nancy A. Lynch, and Muriel Mératd. ARES: adaptive, reconfigurable, erasure coded, atomic storage. CoRR, abs/1805.03727, 2018.  
[11] P. Dutta, R. Guerraoui, R.R. Levy, and A. Chakraborty. How fast can a distributed atomic read be? In Proc. of PODC, pages 236–245, 2004.  
[12] E. Gafni and D. Malkhi. Elastic configuration maintenance via a parsimonious speculating snapshot solution. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), 9363:140–153, 2015.  
[13] C. Georgiou, T. Hadjistasi, N. Nicolaou, and A. Schwarzmann. Unleashing and speeding up readers in atomic object implementations. In Proc. of NETYS, 2018.  
[14] C. Georgiou, N. Nicolaou, and A.A. Shvartsman. Fault-tolerant semifast implementations of atomic read/write registers. Journal of Parallel and Distributed Computing, 69(1):62–79, 2009.
[15] M.P. Herlihy and J.M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, 1990.

[16] M.P. Herlihy and J.M. Wing. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463–492, 1990.

[17] L. Jehl, R. Vitenberg, and H. Meling. Smartmerge: A new approach to reconfiguration for atomic storage. In *International Symposium on Distributed Computing*, pages 154–169. Springer, 2015.

[18] N. Lynch and A.A. Shvartsman. RAMBO: A reconfigurable atomic memory service for dynamic networks. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2508(June):173–190, 2002.

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

[20] N.A. Lynch and A.A. Shvartsman. Robust emulation of shared memory using dynamic quorum-acknowledged broadcasts. In *Proc. of FTCS*, pages 272–281, 1997.

[21] N. Nicolaou, A.F. Anta, and C. Georgiou. Cover-ability: Consistent versioning in asynchronous, fail-prone, message-passing environments. In *Proc. of IEEE NCA 2016*, pages 224–231. Institute of Electrical and Electronics Engineers Inc., 2016.

[22] Nicolas Nicolaou, Viveck Cadambe, N. Prakash, Andria Trigeorgi, Kishori M. Konwar, Muriel Medard, and Nancy Lynch. ARES: Adaptive, Reconfigurable, Erasure coded, Atomic Storage. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 2022.

[23] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC’14, pages 305–320, Berkeley, CA, USA, 2014. USENIX Association.

[24] M O Rabin. Fingerprinting by random polynomials, 1981.

[25] M.V. Steen and A.S. Tanenbaum. *Distributed Systems, 3rd ed.*. distributed-systems.net, 2017.

[26] A. Tridgell and P. Mackerras. The rsync algorithm. *Imagine*, 1996.

[27] P. Viotti and M. Vukolic. Consistency in non-transactional distributed storage systems. *ACM Computing Surveys (CSUR)*, 49:1 – 34, 2016.

[28] ZeroMQ. https://zeromq.org.