Fragmented Ares: Dynamic Storage for Large Objects

Chryssis Georgiou
University of Cyprus, Nicosia, Cyprus

Nicolas Nicolaou
Algolysis Ltd, Limassol, Cyprus

Andria Trigeorgi
University of Cyprus, Nicosia, Cyprus

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 developing efficient, scalable and reliable storage systems has become one of the major challenges for high performance computing. In this work, we develop and prove correct a dynamic, robust and strongly consistent distributed shared memory suitable for handling large objects (such as files) and utilizing erasure coding. We do so by integrating an Adaptive, Reconfigurable, Atomic memory framework, called Ares, with the CoBFS framework, 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 the data and to potentially improve storage efficiency at the replica servers and operation latency. Our development is complemented with an in-depth experimental evaluation on the Emulab and AWS EC2 testbeds, illustrating the benefits of our approach, as well as interesting tradeoffs.

2012 ACM Subject Classification Computing methodologies → Distributed algorithms

Keywords and phrases Distributed storage, Large objects, Strong consistency, High access concurrency, Erasure code, Reconfiguration

Digital Object Identifier 10.4230/LIPIcs.DISC.2022.25

Related Version *Full Version*: `https://arxiv.org/abs/2201.13292`

Supplementary Material *Dataset*: `https://github.com/atrigeorgi/fragmentedARES-data.git` [3]

1 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 [32, 34]. 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 on how to preserve consistency between the replica copies.

For more than two decades, a series of works [11, 26, 19, 15, 18, 23] 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. Linerazibility (atomicity) [24] 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 [22], DynaStore [7], SM-Store [25], SpSnStore [17] and Ares [28].

Currently, such reconfigurable 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). *Coverability* [27] extends linearizability 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. This is essential, for example, for files. When updating the content of a file, one expects that the update is on the previous version of the file; linearizable registers do not impose such restriction, i.e., a write operation might change the value of the object arbitrarily, independently of the previously written value. A recent work by Anta et al. [8], introduced a modular solution, called CoBFS, which combines a suitable data fragmentation strategy, implemented as a Fragmentation module (FM), with a distributed shared memory module (DSMM), to efficiently handle and boost concurrency of large objects, while maintaining strong consistency guarantees (coverability and linearizability), and minimizing operation latencies. The fragmentation strategy enables two (or more) concurrent write operations on different fragments of the object to both take effect, without violating the consistency of the object as a whole. These solutions, as well as the one proposed in [16], were designed for the static environment (fixed set of servers). *In this work we study whether it is plausible to bring coverability and fragmentation to dynamic environments, and how challenging such adaptation would be.*

Contributions. This work is the first to consider dynamic (reconfigurable) Distributed Shared Memory (DSM) tailored for versioned (coverable) and large (fragmentable) objects. At the same time, we aim to introduce solutions that maximize the concurrency of operations on the shared object while trading consistency on the whole object. In particular, we propose a dynamic DSM 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 DSM algorithm Ares [28] 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 extend 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 strongly consistent file sharing).
In summary, our contributions are the following:

- We propose and prove the correctness of the coverable version of Ares, CoAres, the first Fault-tolerant, Reconfigurable, Erasure coded, Atomic Memory, to support versioned objects (Section 4).
- We adopt the idea of fragmentation as presented in CoBFS [8], to obtain CoAresF, which enables CoAres to handle *large* shared data objects and increased data access concurrency (Section 5). The correctness of CoAresF is rigorously proven.
- To reduce the operational latency of the read/write operations in the DSMM 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 DAPs (Section 6).
- We have performed an in-depth experimental evaluation of our approach over both Emulab and Amazon Web Services (AWS) EC2 (Section 7). 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.

We note that although the extension of Ares and its integration with CoBFS might appear conceptually simple, handling reconfiguration was quite subtle, and proving the correctness of the integration was non-trivial. Appendix A provides a table comparing our work with other distributed storage algorithms and systems.

2 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 reconfigurable strongly consistent shared memory 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* ξ 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_ξ is the subsequence of the actions in ξ. A history H_ξ is *sequential* if it starts with an invocation action and each invocation is immediately followed by its matching response; otherwise, H_ξ is *concurrent*. Finally, H_ξ is *complete* if every invocation in H_ξ has a matching response in H_ξ, i.e., each operation in ξ is complete. An operation π_1 precedes an operation π_2 (or π_2 succeeds π_1), denoted by $\pi_1 \rightarrow \pi_2$, in H_ξ, if the response action of π_1 precedes the invocation action of π_2 in H_ξ. 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. (In our implementations, a client can perform any operation.)

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 3); (iv) $c.Con$: a consensus instance with the values from $\mathcal{C}$, implemented as a service 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) [22]. We refer to a server $s \in c.Servers$ as a *member* of configuration c.

Fragmented objects. As defined in [8], a *fragmented object* is a totally ordered sequence of *block objects*. Let $\mathcal{F}$ denote the set of fragmented objects, and $\mathcal{B}$ the set of block objects. A block $b \in \mathcal{B}$ is a concurrent R/W object with a unique id and is associated with two structures, *val* and *ver*: $val(b)$ is composed of a *pointer* that points to the next block in the sequence, and the *data* contained in the block; $ver(b) = \langle wid, bseq \rangle$, where $wid \in \mathcal{I}$ is the id of a writer and $bseq \in \mathbb{N}$ is a sequence number (initially 0). A *fragmented object* $f \in \mathcal{F}$ is a *sequence* of blocks from $\mathcal{B}$, with a value $val(f) = \langle b_0, b_1, b_2, \ldots \rangle$, where each $b_i \in \mathcal{B}$. Initially, a fragmented object contains an empty block, i.e., $val(f) = \langle b_0 \rangle$ with $val(b_0) = \varepsilon$; we refer to it as the *genesis* block.

Coverability and Fragmented Coverability. Our goal is to implement *fragmented linearizable coverable* objects. Linearizability [24] provides the illusion that a concurent object is accessed sequentially when in reality is accessed concurrently by multiple processes. *Coverability* is defined over a *totally ordered* set of *versions* and introduces the notion of *versioned objects*. According to [27], a *versioned object* is a type of R/W object where each value written is assigned with a version. *A coverable object* is a versioned object satisfying the properties *consolidation*, *continuity* and *evolution*.

Intuitively, *consolidation* specifies that write operations may revise the object 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 an object that can be introduced by any operation. Their formal definitions are given in Section 4.

In [27], the notion of a *successful* and *unsuccessful* write was introduced. A successful write is denoted as $cvr\text{-}\omega(ver)[ver', chg]_p$, which updates the object from version ver to ver' (along with the associated values), whereas an unsuccessful write is denoted as $cvr\text{-}\omega(ver)[ver', unchg]_p$ (i.e., it becomes a read). Note that in [27], *ver*s were implemented as *tags*.

Fragmented linearizable coverability [8] guarantees that concurrent write operations on different blocks would *all* prevail (as long as each write is tagged with the latest version of each block), whereas only one write operation on the same block eventually prevails (all other concurrent writes operations on the same block would become read operations).Thus, a fragmented object implementation satisfying this property may lead to higher access concurrency [8].

3 Ares: A Framework for Dynamic Storage

Ares [28] 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 $\langle tag, value \rangle$ pairs to order the operations on a shared object. In contrast to existing solutions, Ares does not define the exact methodology to access the object replicas. Rather, it relies on three, so called, *data access primitives* (DAPs): (*i*) the get-tag, which returns the tag of an object, (*ii*) the get-data, which returns a $\langle tag, value \rangle$ pair, and (*iii*) the put-data($\langle \tau, v \rangle$), which accepts a $\langle tag, value \rangle$ as an argument.

As seen in [28], 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., [10]). Using the DAPs, Ares achieves a modular design, agnostic of the data access strategies, and enables the use of different DAP implementation per configuration (something impossible for other solutions). For the DAPs to be useful, they need to satisfy a property, referred in [28] as **Property 1**, which involves two conditions: **(C1)** if a put-data($\langle \tau, v \rangle$ precedes a get-data (or get-tag) operation, then the latter operation returns a value associated with a tag $\tau' \geq \tau$, and **(C2)** if a get-data returns $\langle \tau', v' \rangle$ then there exists put-data($\langle \tau', v' \rangle$ that precedes or is concurrent to the get-data operation. A formal definition appears in [28].

DAP Implementations. To demonstrate the flexibility that DAPs provide, the authors in [28], expressed two different atomic shared R/W algorithms in terms of DAPs. These are the DAPs for the well celebrated ABD [10] algorithm, and the DAPs for an erasure coded based approach presented for the first time in [28]. In the rest of the manuscript we refer to the two DAP implementations as ABD-DAP and EC-DAP. An $[n, k]$-MDS erasure coding algorithm (e.g., Reed-Solomon [31]) encodes k object fragments into 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 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.

We now provide a high-level description of the two main functionalities supported by ARES: (i) the reconfiguration of the servers, and (ii) the read/write operations on the shared object.

Reconfiguration. Reconfiguration is the process of changing the set of servers. A configuration sequence $cseq$ in ARES is defined as a sequence of pairs $\langle c, status\rangle$ where $c \in \mathcal{C}$, and $status \in \{P, F\}$ (P stands for pending 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 $nextC \in \mathcal{C} \cup \{\perp\} \times \{P, F\}$.

To perform a reconfiguration operation recon(c), a client r follows 4 steps. At first, 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 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 $nextC = \langle c', F\rangle$ in a quorum of servers of the last configuration in $cseq$ (or c_0 if no other configuration exists). As shown in [28], this reconfiguration procedure guarantees that configuration sequences obtained by any two clients $cseq_p$ and $cseq_q$, then either $cseq_p$ is a prefix of $cseq_q$, or vice versa.

Read/Write operations. A write (or read) operation π by a client p is executed by performing the following actions: (i) π invokes a read-config action to obtain the latest configuration sequence $cseq$, (ii) π 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 τ 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 π is a write and $\langle \tau', v'\rangle = \langle \tau, v\rangle$ if π is a read in the last configuration in $cseq$, and read-config to discover any new configuration, until no additional configuration is observed.

4 CoAres: Coverable Ares

In this section we present and analyze the coverable extension of ARES, which we refer to as COARES.

Description. Below we describe the modification that need to occur on ARES in order to support coverability. The reconfiguration protocol and the DAP implementations remain the same as they are not affected by the application of coverability. 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 $flag \in \{chg, unchg\}$, maintained by the write clients, is set to chg when the write operation is successful and to $unchg$ otherwise; initially it is set to $unchg$. 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. 1:14 (resp. line Alg. 1:43).

In the case of cvr-write, the writer w_i finds the last finalized entry in $cseq$, say μ, and performs a $cseq[j].conf$.get-data() action, for $\mu \leq j \leq |cseq|$ (lines Alg. 1:15–18). Thus, w_i retrieves all the $\langle \tau, v\rangle$ 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 $\langle \tau, v\rangle$ pair among all the returned replies. Lines Alg. 1:19 - 1:24 depict the main difference between the coverable cvr-write and the original one: if the maximum τ 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 τ and assigning $\langle \tau, v\rangle$ to $\langle\langle \tau.ts + 1, \omega_i\rangle, val\rangle$, where val is the value it wishes to write (lines Alg. 1:20–21). Otherwise, the state of the object does not change and the writer keeps the maximum $\langle \tau, v\rangle$ 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 τ (line Alg. 1:24).

Algorithm 1 Write and Read protocols for CoAres.

```
     CVR-Write Operation:
2:     at each writer $w_i$
       State Variables:
4:     $cseq[]s.t.cseq[j] \in \mathcal{C} \times \{F, P\}$
       $version \in \mathbb{N}^+ \times \mathcal{W} \cup \{\bot\}$ initially $\langle 0, \bot\rangle$
6:     Local Variables:
       $\mu \in \mathbb{N}^+$ initially 0, $\nu \in \mathbb{N}^+$ initially 0
8:     $\tau \in \mathbb{N}^+ \times \mathcal{W}$ initially $\langle 0, w_i\rangle$
       $v \in V$ initially $\bot$
10:    $flag \in \{chg, unchg\}$ initially $unchg$
       Initialization:
12:    $cseq[0] = \langle c_0, F\rangle$

       operation cvr-write($val$), $val \in V$
14:      $cseq \leftarrow$read-config($cseq$)
         $\mu \leftarrow \max(\{i : cseq[i].status = F\})$
16:      $\nu \leftarrow |cseq|$
         for $i = \mu : \nu$ do
18:        $\langle \tau, v\rangle \leftarrow \max(cseq[i].cfg.$get-data$(), \langle \tau, v\rangle)$
         if $version = \tau$ then
20:        $flag \leftarrow chg$
           $\langle \tau, v\rangle \leftarrow \langle\langle \tau.ts + 1, \omega_i\rangle, val\rangle$
22:      else
           $flag \leftarrow unchg$
24:      $version \leftarrow \tau$
         $done \leftarrow false$
26:      while not $done$ do
           $cseq[\nu].cfg.$put-data$(\langle \tau, v\rangle)$
28:        $cseq \leftarrow$read-config($cseq$)
           if $|cseq| = \nu$ then
30:          $done \leftarrow true$
           else
32:          $\nu \leftarrow |cseq|$
         end while
34:      return $\langle \tau, v\rangle, flag$
       end operation

36:  CVR-Read Operation:
       at each reader $r_i$
38:    State Variables:
       $cseq[]s.t.cseq[j] \in \mathcal{C} \times \{F, P\}$
40:    Initialization:
       $cseq[0] = \langle c_0, F\rangle$

42:    operation cvr-read( )
         $cseq \leftarrow$read-config($cseq$)
44:      $\mu \leftarrow \max(\{j : cseq[j].status = F\})$
         $\nu \leftarrow |cseq|$
46:      for $i = \mu : \nu$ do
           $\langle \tau, v\rangle \leftarrow \max(cseq[i].cfg.$get-data$(), \langle \tau, v\rangle)$
48:      $done \leftarrow$ false
         while not $done$ do
50:        $cseq[\nu].cfg.$put-data$(\langle \tau, v\rangle)$
           $cseq \leftarrow$read-config($cseq$)
52:        if $|cseq| = \nu$ then
             $done \leftarrow true$
54:        else
             $\nu \leftarrow |cseq|$
56:      end while
         return $\langle \tau, v\rangle$
58:    end operation
```

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. 1:46–47). The propagation of $\langle\tau, v\rangle$ in both cvr-write (lines Alg. 1:26–33) and cvr-read (lines Alg. 1:49–56)) remains the same. Finally, the cvr-write operation returns $\langle\tau, v\rangle$ and the $flag$, whereas the cvr-read operation only returns ($\langle\tau, v\rangle$).

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 completes. As shown in [28], Ares implements a linearizable object given that the DAP used satisfy Property 1. Given that CoAres uses the same reconfiguration and read operations, while the write operation might get converted to a read operation, then linearizability is not affected and can be shown that it holds in a similar way as in [28].

The validity and coverability properties, defined formally below as Definitions 1 and 2, 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 $c \in \mathcal{C}$ satisfies Property 1, we will show that any execution ξ of CoAres satisfies the properties of Definitions 1 and 2.

Proof challenges: The main challenge is to show that CoAres satisfies the coverability properties despite *any reconfiguration* in the system. In particular, we would like to ensure: (i) new values are not overwritten, i.e., if a write is successfully completed then no subsequent write successfully writes a value associated with an older version in any active configuration, (ii) versions are unique, and (iii) eventually a single version path prevails.

Definitions and proofs: In the lemmas that follow, we refer to a successful write operation as one that is not converted to a read operation. We say that a write operation *revises* a version ver of the versioned object to a version ver', or *produces* ver', in an execution ξ, if $cvr\text{-}\omega(ver)[ver']_{p_i}$ completes in H_ξ. Let the set of *successful write* operations on a history H_ξ be defined as $\mathcal{W}_{\xi,succ} = \{\pi : \pi = cvr\text{-}\omega(ver)[ver']_{p_i}$ completes in $H_\xi\}$. The set of the object's versions produced by writes operations in the history H_ξ is defined by $Versions_\xi = \{ver_i : cvr\text{-}\omega(ver)[ver_i]_{p_i} \in \mathcal{W}_{\xi,succ}\} \cup \{ver_0\}$, where ver_0 is the initial version of the object. Observe that the elements in $Versions_\xi$ are totally ordered. Now we present the *validity* property which defines explicitly the set of executions that are considered to be valid executions.

▶ **Definition 1** (Validity [27]). *An execution ξ (resp. its history H_ξ) is a* valid execution *(resp. history) on a versioned object, for any $p_i, p_j \in \mathcal{I}$:*

1. *$\forall cvr\text{-}\omega(ver)[ver']_{p_i} \in \mathcal{W}_{\xi,succ}, ver < ver'$,*
2. *for any operations $cvr\text{-}\omega(*)[ver']_{p_i}$ and $cvr\text{-}\omega(*)[ver'']_{p_j}$ in $\mathcal{W}_{\xi,succ}$, $ver' \neq ver''$, and*
3. *for each $ver_k \in Versions_\xi$ there is a sequence of versions $ver_0, ver_1, \ldots, ver_k$, such that $cvr\text{-}\omega(ver_i)[ver_{i+1}] \in \mathcal{W}_{\xi,succ}$, for $0 \le i < k$.*

▶ **Definition 2** (Coverability [27]). *A valid execution ξ is* ***coverable*** *with respect to a total order $<_\xi$ on operations in $\mathcal{W}_{\xi,succ}$ if:*

1. **(Consolidation)** *If $\pi_1 = cvr\text{-}\omega(*)[ver_i], \pi_2 = cvr\text{-}\omega(ver_j)[*] \in \mathcal{W}_{\xi,succ}$, and $\pi_1 \rightarrow_{H_\xi} \pi_2$ in H_ξ, then $ver_i \le ver_j$ and $\pi_1 <_\xi \pi_2$.*
2. **(Continuity)** *if $\pi_2 = cvr\text{-}\omega(ver)[ver_i] \in \mathcal{W}_{\xi,succ}$, then there exists $\pi_1 \in \mathcal{W}_{\xi,succ}$ s.t. $\pi_1 = cvr\text{-}\omega(*)[ver]$ and $\pi_1 <_\xi \pi_2$, or $ver = ver_0$.*
3. **(Evolution)** *let $ver, ver', ver'' \in Versions_\xi$. If there are sequences of versions $ver'_1, ver'_2, \ldots, ver'_k$ and $ver''_1, ver''_2, \ldots, ver''_\ell$, where $ver = ver'_1 = ver''_1$, $ver'_k = ver'$, and $ver''_\ell = ver''$ such that $cvr\text{-}\omega(ver'_i)[ver'_{i+1}] \in \mathcal{W}_{\xi,succ}$, for $1 \le i < k$, and $cvr\text{-}\omega(ver''_i)[ver''_{i+1}] \in \mathcal{W}_{\xi,succ}$, for $1 \le i < \ell$, and $k < \ell$, then $ver' < ver''$.*

We proceed with formal statements and proofs. Lemmas 3 to 5 help us show that CoAres satisfies *Validity*.

▶ **Lemma 3** (Version Increment). *In any execution ξ of* CoAres, *if ω is a successful write operation, and ver the maximum version it discovered during the* get-data *operation, then ω propagates a version $ver' > ver$.*

Proof. This lemma follows from the fact that CoAres uses a condition before the propagation phase in line Alg. 1:19. 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 4** (Version Uniqueness). *In any execution ξ of* CoAres, *if two write operations ω_1 and ω_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 ω_1 and w_2 the id of the writer that invoked ω_2. To show whether the versions generated by the two write operations are not equal we need to examine two cases: (a) both ω_1 and ω_2 are invoked by the same writer, i.e. $w_1 = w_2$, and (b) ω_1 and ω_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-formdness, writer w_1 can only invoke one operation at a time. Thus, the last put-data$(ver_1, *)$ of ω_1 completes before the first get-data of ω_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 ω_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 $\omega_1 \to \omega_2$, then by the sequence prefix guaranteed by the reconfiguration protocol of Ares (second property) the $cseq_1$ obtained during the read-config action in ω_1 is a prefix of the $cseq_2$ obtained during the same action in ω_2. Notice that c' is the last finalized configuration in $cseq_2$ as this is the configuration where the first get-data action of ω_2 is invoked. If c' precedes c in $cseq_2$ then by CoAres the write operation ω_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 precedes c' in $cseq_2$, then it must be the case that a reconfiguration operation r has been invoked concurrently or after ω_2 and added c'. By Ares [28], r, invoked a put-data(ver') in c' before finalizing c' with $ver' \geq ver_1$. So when ω_2 invokes get-data in c' by $C1$ 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 ω_1 generates a version $ver_1 = \{ts_1, w_1\}$ and ω_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 5.** *Each version we reach in an execution is derived (through a chain of operations) from the initial version of the register ver_0.*

Proof. Every tag is generated by extending the tag retrieved by a get-data operation starting from the initial tag (lines Alg. 1:20–21). 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. ◀

From this point onward we fix ξ to be a valid execution and H_ξ to be its valid history. We now show coverability (Definition 2).

▶ **Lemma 6.** *In any execution ξ of* CoAres*, all properties of Definition 2 are satisfied.*

Proof. For *consolidation* we need to show that for two write operations $\omega_1 = cvr\text{-}\omega(*)[\tau_1, chg]$ and $\omega_2 = cvr\text{-}\omega(\tau_2)[*, chg]$, if $\omega_1 \to_\xi \omega_2$ then $\tau_1 \leq \tau_2$. According to $C1$ of Property 1, since the get-data of ω_2 appears after the put-data of ω_1, the get-data of ω_2 returns a tag higher than the one written by ω_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 $\pi_1 = cvr\text{-}\omega(\tau)[\tau']$ generates a new tag τ' from τ such that $\tau'.ts = \tau.ts + 1$ (line Alg. 1:21). Consider sequences of tags $\tau_1, \tau_2, \ldots, \tau_k$ and $\tau'_1, \tau'_2, \ldots, \tau'_\ell$ such that $\tau_1 = \tau'_1$. Assume that $cvr\text{-}\omega(\tau_i)[\tau_{i+1}]$, for $1 \leq i < k$, and $cvr\text{-}\omega(\tau'_i)[\tau'_{i+1}]$, for $1 \leq i < \ell$, are successful writes. If $\tau_1.ts = \tau'_1.ts = z$, then $\tau_k.ts = z + k$ and $\tau'_\ell.ts = z + \ell$, and if $k < \ell$ then $\tau_k < \tau'_\ell$. ◀

Lemmas 3 to 6 show that CoAres satisfies validity (Def. 1) and coverability (Def. 2):

▶ **Theorem 7.** CoAres *implements a linearizable coverable object, given that the DAPs implemented in any configuration c satisfy Property 1.*

5 CoAresF: Integrate CoAres with a Fragmentation approach

The work in [8] developed a distributed storage framework, called CoBFS, which utilizes coverable fragmented objects. In this section we describe how CoAres can be integrated with CoBFS to obtain what we call CoAresF, thus yielding a dynamic distributed memory 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, making storage efficient at the servers. A particular challenge of this integration is how the fragmentation approach should invoke reconfiguration operations, since CoBFS in [8] considered only static (non-reconfigurable) systems. The **main challenge** of CoAresF, however, was to prove that the blocks' sequence of a fragmented object remains connected, despite the existence of concurrent read/write and reconfiguration operations.

Overview of CoBFS. 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 summary, the FM implements the fragmented object, which is a totally ordered sequence of blocks (where a block is a R/W object with limited value size; cf. Section 2), while the DSMM implements an interface to a shared memory service that allows operations on individual block objects. To this respect, CoBFS is flexible enough to utilize *any* underlying distributed shared memory implementation.

CoBFS mainly supports two operations, update and read, described next.

Update Operation. The update operation spans both modules, FM and DSMM. The FM uses a **Block Identification (BI) module**, which draws ideas from the RSYNC (Remote Sync) algorithm [33]. The BI includes three main modules, the *Block Division*, the *Block Matching* and *Block Updates*.

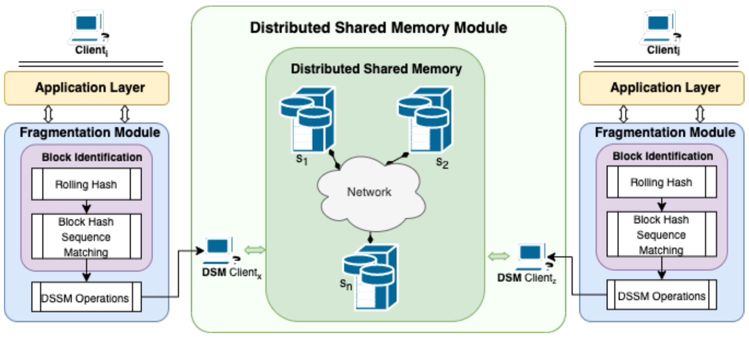


Figure 1 Basic architecture of CoBFS [8].

1. *Block Division:* This module splits a given fragmented object f into blocks. (This can be done, for example, in files, by using rolling hashing, such as *rabin fingerprints* [30], as we've done in our implementation.)
2. *Block Matching:* This module is used to find the differences between the new and the old blocks, yielding four *statuses*: (i) equality, (ii) modified, (iii) inserted, (iv) deleted. (As in our implementation, this can be done by using a string matching algorithm [13].)
3. *Block Updates:* Based on the retrieved statuses, the blocks of the fragmented object are then written on the DSM using the DSMM as an external service. In the case of equality, no operation is performed. In the case of modification, an *update* operation attempts to write the modified block. If new blocks are inserted after an existing block b, the *update* operation first writes the new blocks and then writes b so that the list of blocks remains connected. Delete is treated as a modification that sets an empty value to a block.

Read Operation. When the system receives a read request from a client, the FM issues, to the DSMM, a series of read operations on the fragmented object'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 as a fragmented object.

Integration of COARES in COBFS. Integration with the CoBFS is achieved by using CoAres as the external DSMM service. To accommodate the dynamic nature of CoAres, we need to introduce the reconfiguration operation in CoAresF as shown next.

Reconfig Operation. The specification of reconfig on the DSMM is given in Alg. 2, while the specification of reconfig on a fragmented object is given in Alg. 3.

When the system receives a reconfig request from a client, the FM issues a series of reconfig operations on the fragmented object's blocks, starting from the genesis block and proceeding to the last block by following the next block ids (Alg. 3). The $reconfig$ operation executes the *block* $reconfig$ operations on the shared memory (Alg. 2) using dsmm-reconfig operations.

Algorithm 2 DSMM: Reconfig operation on block b at client p.

```
1: function dsmm-reconfig$(c)_{b,p}$
2:     $b$.reconfig$(c)$
3: end function
```

Algorithm 3 FM: Reconfig operation on fragmented object f at client p.

```
1: State Variables:
2: $\mathcal{L}_f$ a sequence of blocks, initially $\langle b_0 \rangle$;

3: function fm-reconfig(c)$_{f,p}$
4:    $b \leftarrow val(b_0).ptr$
5:    $\mathcal{L}_f \leftarrow \langle b_0 \rangle$
6:    while $b$ not NULL do
7:       dsmm-reconfig(c)$_{b,p}$
8:       $b \leftarrow val(b).ptr$
9:    end while
10: end function
```

Correctness of CoAresF. When a reconfig(c) operation is invoked in CoAres, a reconfiguration client requests to change the configuration of the servers hosting the single R/W object. By design, each instance of CoAres handles a single R/W object. In the case of a fragmented object f, each block composing f is handled as a separate atomic object, and thus assigned to a different Ares instance. Therefore, the **main challenge** of CoAresF is to ensure that the sequence composing f remains connected and composed of the most recent blocks, despite concurrent read/write and reconfig operations. Note that each individual block may exist in different configurations and be accessed by different DAPs.

In the remainder we show that *fragmented coverability* (see Section 2) cannot be violated. Before we prove any lemmas, we first state a claim that follows directly from the algorithm.

▷ Claim 8. For any block $b \neq b_0$, where b_0 the genesis block, created by an 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 sequence obtained by a read operation, there is a successful update operation that wrote this block. Its proof follows the proof of Lemma 4 presented in [9].

▶ **Lemma 9.** *In any execution ξ of CoAresF, if ρ is a read operation on f that returns a sequence $\mathcal{L}$, then for any block $b \in \mathcal{L}$, there exists a successful update operation on f that either precedes or is concurrent to ρ.*

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 10.** *Suppose that ρ is a* dsmm-reconfig$(c_2)_{b,*}$ *operation and ω a successful* cvr-write$(v)_{b,*}$ *operation that changes the version of b to ver, s.t. $\omega \rightarrow \rho$ in an execution ξ of CoAresF. Then ρ invokes c_2.*put-data$(\langle ver', * \rangle)$ *in c_2, s.t. $ver' \geq ver$.*

Proof. Let $cseq_\omega$ be the last configuration sequence returned by the read-config action at ω (Alg. 1:28), and $cseq_\rho$ the configuration sequence returned by the first read-config action at ρ (see Alg. 2:8 in [28]). By the prefix property of the reconfiguration protocol, $cseq_\omega$ will be a prefix of $cseq_\rho$.

Let c_ℓ the last configuration in $cseq_\omega$, and c_1 the last finalized configuration in $cseq_\rho$. There are two cases to examine: (i) c_1 precedes c_ℓ in $cseq_\rho$, and (ii) c_1 appears after c_ℓ in $cseq_\rho$. If (i) is the case then during the update-config action, ρ will perform a c_ℓ.get-data() action. By $C1$ in Property 1, the c_ℓ.get-data() will return a version $ver'' \geq ver$. Since the ρ function will execute c_2.put-data$(\langle ver', * \rangle)$, s.t. ver' is the max discovered version, then $ver' \geq ver'' \geq ver$.

In case (ii) it follows that the reconfiguration operation that proposed c_1 has finalized the configuration. So either that reconfiguration operation moved a version ver'' of b s.t. $ver'' \geq ver$ in the same way as described in case (i) in c_1, or the write operation would observe c_1 during a read-config action. In the latter case c_1 will appear in $cseq_\omega$ and ω will invoke a c_ℓ.put-data($\langle ver, * \rangle$) s.t. either $c_\ell = c_1$ or c_ℓ a configuration that appears after c_1 in $cseq_\omega$. Since c_1 is the last finalized configuration in $cseq_\rho$, then in any of the cases described ρ will invoke a c_ℓ.get-data(). Thus, it will discover and put in c_2 a version $ver' \geq ver$ 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. *This corresponds to the most challenging part of the integration.*

▶ **Lemma 11.** *In any execution ξ of* CoAresF*, if ρ is a read operation on f that returns a sequence of blocks $\mathcal{L} = \{b_0, b_1, \dots, b_n\}$, then it must be the case that (i) $b_0.ptr = b_1$, (ii) $b_i.ptr = b_{i+1}$, for $i \in [1, n-1]$, and (iii) $b_n.ptr = \perp$.*

Proof. Assume by contradiction that there exist some $b_i \in \mathcal{L}$, s.t. $val(b_i).ptr \neq b_{i+1}$ (or $val(b_0).prt \neq b_1$). By Lemma 9, a block b_i may appear in the sequence returned by a read operation only if it was created by a successful update operation π, on block b. Let $\mathcal{B} = \langle b_1, \dots, b_k \rangle$ be the set of $k-1$ blocks created in π, with $b_i \in \mathcal{B}$. Let us assume w.l.o.g. that all those blocks appear in $\mathcal{L}$ 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 b_k, by pointing b to b_1 and each b_j to b_{j+1}, for $1 \leq j < k$. Block b_k points to the block pointed by b at the invocation of π, say b'. So there exists a path $b \to b_1 \to \dots \to b_i$ that also leads to b_i. According again to the algorithm, $b_{j+1} \in \mathcal{B}$ is created and written before b_j, for $q \leq j < k$. So when the b_j.cvr-write is invoked, the operation b_{j+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 b_j written by π, ρ will observe a configuration sequence $b_j.cseq_\rho$, s.t. $b_j.cseq_\pi$ is a prefix of $b_j.cseq_\rho$, and hence c_π appears in $b_j.cseq_\rho$. If c_π appears after the last finalized configuration c_ℓ in $b_j.cseq_\rho$, then the read operation will invoke c_π.get-data() and by the coverability property and property C1, will obtain a version $ver' \geq ver$. In case c_π precedes c_ℓ then a new configuration was invoked after or concurrently to π and then by Lemma 10 it follows that the version of b in c_ℓ is again $ver' \geq ver$. So we need to examine the following three cases for b_i: (i) b_i is b, (ii) b_i is b_k, and (iii) b_i is one of the blocks b_j, for $1 \leq j < k$.

Case (i): If b_i is the block b then we should examine whether $b_i.ptr \neq b_1$. Let ver the version of b written by π and ver' the version of b as retrieved by ρ. If $ver = ver'$ then ρ retrieved the block written by ω as the versions by Lemma 4 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 ω' that written block b with ver'. There are two cases to consider based on whether ω' introduced new blocks or not. If not then the $b.ptr = b_1$ contradicting our assumption. If it introduced a new sequence of blocks $\{b'_1, \dots, b'_k\}$, then it should have written those blocks before writing b. In that case ρ would observe $b.ptr = b'_1$ and b'_1 would have been part of $\mathcal{L}$ which is not the case as the next block from b in $\mathcal{L}$ is b_1, leading to contradiction.

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

Case (iii): If now $b_i = b_k$, then we should examine whether $b_i.ptr \neq b'$. Since b was pointing to b' at the invocation of π 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 9, 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 $\mathcal{L}$ when b_i is created and thus b_i points to an existing block. Furthermore, since π was successful, then it successfully written b and hence only the blocks in $\mathcal{B}$ were inserted between b and b' at the response of π. In case the version of b_i was ver' and larger than the version written on b_k by π then either b_k was not extended and contains new data, or the new block is impossible as $\mathcal{L}$ should have included the blocks extending b_k. So b' must be the next block after b_i in $\mathcal{L}$ at the response of π and there is a path between b and b'. This completes the proof. ◀

We conclude with the main result of this section.

▶ **Theorem 12.** CoAresF *implements a* linearizable coverable fragmented object.

Proof. By the correctness proof in Section 4 follows that every block operation in CoAresF satisfies linearizable coverability and together with Lemma 11, which shows the connectivity of blocks, it follows that CoAresF implements a linearizable coverable fragmented object satisfying the properties of *fragmented linearizable coverability* (cf. Section 2). ◀

6 EC-DAP Optimization

In this section, we present an optimization in the implementation of EC-DAP, to reduce the operational latency of the read/write operations in DSMM layer. We show that this optimized EC-DAP, which we refer to as EC-DAPopt, satisfies Property 1, and thus can be used by any algorithm that utilizes the DAPs, like any variant of Ares (e.g., CoAres and CoAresF).

Description of EC-DAPopt. The main idea of the optimization is to avoid unnecessary object transmissions between the clients and the servers. Specifically, 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 Alg. 4 and 5. Text in blue annotates the changed or newly added code, whereas ~~struck out blue text~~ annotates code that has been removed from the original implementation.

Following [28], each server s_i stores a state variable, $List$, which is a set of up to $(\delta + 1)$ (tag, coded-element) pairs; δ is the maximum number of concurrent put-data operations. In EC-DAPopt, we need another two state variables, the tag of the configuration ($c.tag$) and its associated value ($c.val$). We now proceed with the details of the optimization. Note that the c.get-tag() primitive remains the same as the original.

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 $List$,

Algorithm 4 EC-DAPopt implementation.

```
    at each process $p_i \in \mathcal{I}$

2:  procedure c.get-data()
      send (QUERY-LIST,$c.tag$) to each $s \in c.Servers$
4:    until $p_i$ receives $List_s$ from each server $s \in \mathcal{S}_g$
           ↪ s.t. $|\mathcal{S}_g| = \lceil \frac{n+k}{2} \rceil$
           and $\mathcal{S}_g \subset c.Servers$
6:    ~~$Tags_*^{\geq k}$ = set of tags that appears in $k$ lists~~
      $Tags_{dec}^{\geq k}$ = set of tags that appears in $k$ lists
8:         with values
      ~~$t_{max}^* \leftarrow \max Tags_*^{\geq k}$~~
10:   $t_{max}^{dec} \leftarrow \max Tags_{dec}^{\geq k}$
      ~~if $t_{max}^{dec} = t_{max}^*$ then~~
12:   if $c.tag = t_{max}^{dec}$ then
        $t \leftarrow c.tag$
14:     $v \leftarrow c.val$
        return $\langle t, v \rangle$
16:   else if $Tags_{dec}^{\geq k} \neq \bot$ then
        $t \leftarrow t_{max}^{dec}$
18:     $v \leftarrow$ decode value for $t_{max}^{dec}$
        return $\langle t, v \rangle$
20: end procedure

    procedure c.put-data($\langle \tau, v \rangle$))
22:   if $\tau > c.tag$ then
        $code\text{-}elems = [(\tau, e_1), \ldots, (\tau, e_n)]$, $e_i$
24:          $= \Phi_i(v)$
        send (PUT-DATA, $\langle \tau, e_i \rangle$) to each $s_i$
             ↪ $\in c.Servers$
26:     until $p_i$ receives ACK from $\lceil \frac{n+k}{2} \rceil$ servers in
             ↪ $c.Servers$
        $c.tag \leftarrow \tau$
28:     $c.val \leftarrow v$
    end procedure
```

Algorithm 5 The response protocols at any server $s_i \in \mathcal{S}$ in EC-DAPopt for client requests.

```
    at each server $s_i \in \mathcal{S}$ in configuration $c_k$
2:  State Variables:
    $List \subseteq \mathcal{T} \times \mathcal{C}_s$, initially $\{(t_0, \Phi_i(v_0))\}$
    Local Variables:
    $List' \subseteq \mathcal{T} \times \mathcal{C}_s$, initially $\bot$

4:  Upon receive (QUERY-LIST, $tg_b$) $_{s_i, c_k}$ from $q$
      for $\tau, v$ in $List$ do
6:      if $\tau > tg_b$ then
          $List' \leftarrow List' \cup \{\langle \tau, e_i \rangle\}$
8:      else if $\tau = tg_b$ then
          $List' \leftarrow List' \cup \{\langle \tau, \bot \rangle\}$
10:   Send $List'$ to $q$
    end receive

12: Upon receive (PUT-DATA, $\langle \tau, e_i \rangle$) $_{s_i, c_k}$ from $q$
      $List \leftarrow List \cup \{\langle \tau, e_i \rangle\}$
14:   if $|List| > \delta + 1$ then
        $\tau_{min} \leftarrow \min\{t : \langle t, * \rangle \in List\}$
          /* remove the coded value */
16:     $List \leftarrow List \backslash \ \{\langle \tau, e \rangle : \tau = \tau_{min} \ \wedge \langle \tau, e \rangle$
                       $\in List\}$
18:     ~~$List \leftarrow List \cup \{(\tau_{min}, \bot)\}$~~
      Send ACK to $q$
20: end receive
```

if the tag is higher than the $c.tag$ of the client and the (tag, $\bot$) 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. 4:8–10) 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 (t_{max}^{dec}), the client already has the latest decodable version and does not need to decode it again (see line Alg. 4:12).

Primitive c.put-data($\langle t_w, v \rangle$). This primitive is executed only when the incoming t_w is greater than $c.tag$ (line Alg. 4:22). 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).

Correctness of EC-DAPopt. We prove the following theorem.

▶ **Theorem 13** (Safety + Liveness)**.** *EC-DAPopt satisfies both conditions of Property 1, and given that no more than δ write operations are concurrent with a read they guarantee that any operation terminates.*

The complete proof is given in Appendix B. The **main challenge** of the proof is to show that reducing the values returned by the servers does not violate linearizability, and at the same time, it does not prevent operations from reconstructing the written values, preserving liveness. We prove safety by showing that EC-DAPopt satisfies both conditions of Property 1. Particularly, we prove that the tag returned by a get-data() operation is larger than or equal to the tag written by any preceding put-data() operation, and the value returned by a get-data() operation is either written by a put-data() operation or it is the initial value of the object. Liveness is proven by showing that any put-data and get-data operation defined by EC-DAPopt terminates. In the proof, we assume an $[n, k]$ MDS code, $|c.Servers| = n$ of which no more than $\frac{n-k}{2}$ may crash, and that δ is the maximum number of put-data operations concurrent with any get-data operation. Without this assumption on δ, a get-data operation may not be able to discover a decodable value, and hence fail.

7 Experimental Evaluation

We now overview the experimental evaluation we conducted for evaluating our approach. Additional results are given in Appendix C. For a more extensive exposition of our experimental evaluation and obtained results, see [20]. The collected data are available in [3], so one could validate our analysis.

We have implemented and evaluated the following algorithms: *(i)* CoABD: the coverable version of the static ABD algorithm [27]; *(ii)* CoABDF: the fragmented version of CoABD [8]; *(iii)* CoAresABD: CoAres that uses ABD-DAP; *(iv)* CoAresABDF: fragmented CoAresABD; *(v)* CoAresEC: CoAres that uses EC-DAPopt; and *(vi)* CoAresECF: fragmented CoAresEC.

In our implementations, we consider *files*, as an example of fragmented objects. In 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 block that contains specific file information (such as the file path). For the evaluation we generate a text file with random byte strings whose size increases as the writers keep updating it. However, our implementations support any file type. The algorithms were evaluated in terms of *operational latency* and the *percentage of successful file writes.*

The experiments were executed on the emulation testbest Emulab [5], and the overlay testbed Amazon Web Services (AWS) EC2 [12]. On Emulab we used a LAN using a DropTail queue without delay or packet loss, consisting of physical nodes with one 2.4 GHz 64-bit Quad Core Xeon E5530 “Nehalem” processor and 12 GB RAM. While on AWS we used a cluster with 8 nodes of type t2.medium with 4 GB RAM, 2 vCPUs and 20 GB storage. For each experiment on Emulab we reported the average over five runs, while AWS experiments run only once.

Performance VS. Initial File Sizes. We varied the f_{size} from 1 MB to 512 MB 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. For Emulab we used $|\mathcal{W}| = 5, |\mathcal{R}| = 5, |\mathcal{S}| = 11$, while for AWS we used $|\mathcal{W}| = 1, |\mathcal{R}| = 1, |\mathcal{S}| = 6$. Each client in Emulab performs 20 operations and in AWS 50 operations. We used a *stochastic* invocation scheme in which clients pick a random time between the interval [1...3*sec*] to invoke their next operations.

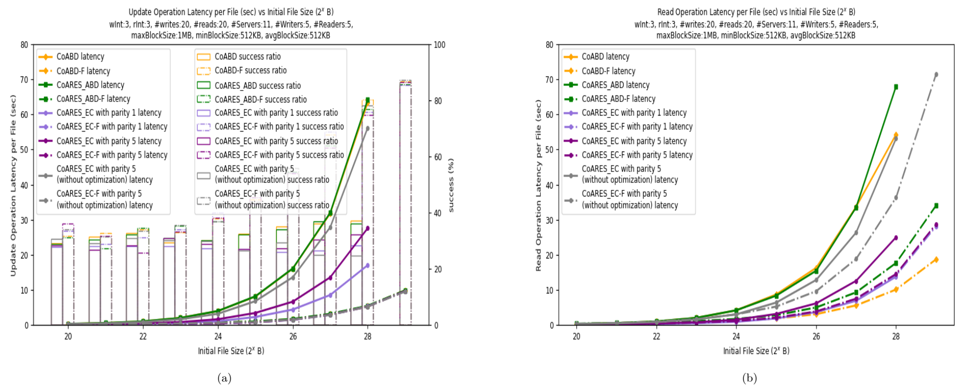


Figure 2 Emulab results for File Size experiments.

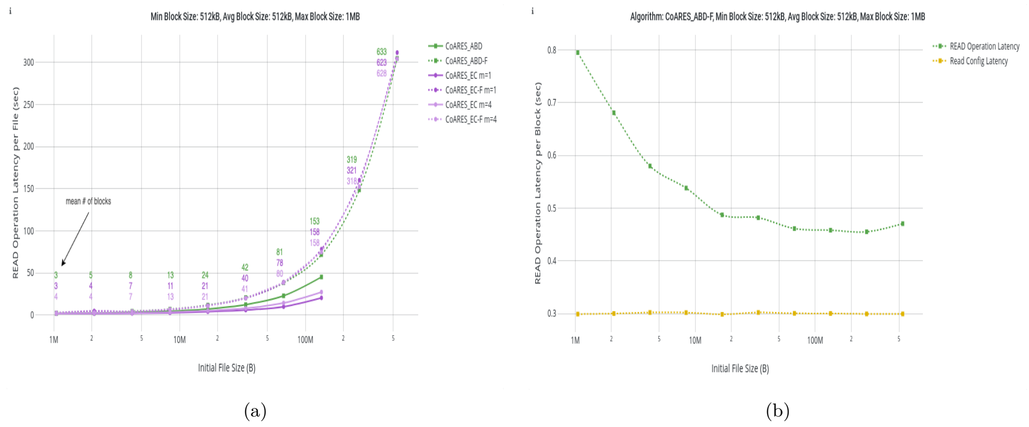


Figure 3 AWS results for File Size experiments.

Results. As shown in Fig. 2(a), the fragmented algorithms on Emulab achieve significantly smaller write latency, since the FM writes only the new and modified blocks. Also, their success ratio is higher as the file size increases, since the probability of two writes to collide on a single block decreases. The corresponding AWS findings show similar trends.

As shown in Fig. 2(b), all the fragmented algorithms on Emulab 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. On the contrary, the read latency of CoAres on AWS (Fig. 3(a)) has not improved with the fragmentation strategy. The CoAresF operations perform at least two additional rounds (compared to CoABDF), in order to read the configuration before each of the two phases. Thus, when the FM module sends multiple read block requests, has a significant stable overhead for each block request in the real network conditions of AWS (Fig. 3(b)).

We can also observe from the Figs. 2(a)-(b), 3(a) that the further increase of the parity (m) of CoAresEC and CoAresECF algorithms (and thus higher fault-tolerance) the larger the latency. In addition, the read and write latency of these algorithms when used with EC-DAP are double than of the ones when our optimized DAP (EC-DAPopt) is used.

Trade-offs. During the deployment, the main trade-offs we have identified are the following:

Block size of FM. The performance of data striping highly depends on the block size. There is a trade-off between splitting the object into smaller blocks, for improving the concurrency in the system, and paying for the cost of sending these blocks in a distributed fashion. Therefore, it is crucial to discover the "golden" spot with the minimum communication delays (while having a large block size) that will ensure a small expected probability of collision (as a parameter of the block size and the delays in the network).

Parity of EC. There is a trade-off between operation latency and fault-tolerance in the system: the further increase of the parity (and thus higher fault-tolerance) the larger the latency.

Parameter δ of EC. The value of δ is equal to the number of writers. As a result, as the number of writers increases, the latency of the first phase of EC also increases, since each server sends the list with all the concurrent values. In this point, we can understand the importance of the optimization (EC-DAPopt) in the DSMM layer.

8 Conclusions

In this paper we have presented and rigorously proved correct CoAresF, the first dynamic distributed shared memory that utilizes coverable fragmented objects and enables the use of erasure coding. To achieve this, we developed a coverable version of Ares and integrated it with CoBFS. When CoAresF is used with an (optimized) Erasure Coded DAP we obtain a two-level striping dynamic and robust distributed shared memory system providing strong consistency and high access concurrency to large objects (e.g., files). We have complemented our development with an extensive experimental evaluation over the Emulab and AWS testbeds. Compared to the approach that does not use the fragmentation layer of CoBFS (CoAres), CoAresF is optimized with an efficient access to shared data under heavy concurrency. For future work, we plan to explore how to reduce the overhead of read operations. In addition, as our service achieves highly scalable performance, it seems suitable for a P2P environment; any physical node could serve both as a client and a data host.

References

1 Cassandra. `https://cassandra.apache.org/_/index.html`.

2 Colossus. `https://cloud.google.com/blog/products/storage-data-transfer/a-peek-behind-colossus-googles-file-system`.

3 Data repository. `https://github.com/atrigeorgi/fragmentedARES-data.git`.

4 Dropbox. `https://www.dropbox.com/`.

5 Emulab network testbed. `https://www.emulab.net/`.

6 Hdfs. `https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html`.

7 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 17–25, New York, NY, USA, 2009. ACM.

8 A.F. Anta, C. Georgiou, T. Hadjistasi, E. Stavrakis, and A. Trigeorgi. Fragmented Object : Boosting Concurrency of Shared Large Objects. *In Proc.of SIROCCO*, pages 1–18, 2021.

9 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. `arXiv:2102.12786`.

10 H. Attiya. Robust Simulation of Shared Memory: 20 Years After. *Bulletin of the EATCS*, 100:99–114, 2010.

11 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.

12 AWS EC2. `https://aws.amazon.com/ec2/`.

13 Paul Black. Ratcliff pattern recognition. *Dictionary of Algorithms and Data Structures*, 2021.

14 A. Carpen-amarie. BlobSeer as a Data-Storage Facility for Clouds: Self-Adaptation, Integration, Evaluation, PhD Thesis, France, 2012.

15 P. Dutta, R. Guerraoui, R.R. Levy, and A. Chakraborty. How fast can a distributed atomic read be? *In Prof. of PODC*, pages 236–245, 2004.

16 Rui Fan and Nancy A. Lynch. Efficient replication of large data objects. In Faith Ellen Fich, editor, *Distributed Computing, 17th International Conference, DISC 2003, Sorrento, Italy, October 1-3, 2003, Proceedings*, volume 2848 of *Lecture Notes in Computer Science*, pages 75–91. Springer, 2003. `doi:10.1007/978-3-540-39989-6_6`.

17 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. `doi:10.1007/978-3-662-48653-5_10`.

18 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.

19 Chryssis Georgiou, Theophanis Hadjistasi, Nicolas Nicolaou, and Alexander A. Schwarzmann. Implementing three exchange read operations for distributed atomic storage. *J. Parallel Distributed Comput.*, 163:97–113, 2022. `doi:10.1016/j.jpdc.2022.01.024`.

20 Chryssis Georgiou, Nicolas Nicolaou, and Andria Trigeorgi. Fragmented ARES: dynamic storage for large objects. *CoRR*, abs/2201.13292, 2022. `arXiv:2201.13292`.

21 Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google File System. *The Google File System*, 53(1):79–81, 2003.

22 Seth Gilbert, Nancy A. Lynch, and Alexander A. Shvartsman. RAMBO: A robust, reconfigurable atomic memory service for dynamic networks. *Distributed Comput.*, 23(4):225–272, 2010. `doi:10.1007/s00446-010-0117-1`.

23 Vincent Gramoli, Nicolas Nicolaou, and Alexander A. Schwarzmann. *Consistent Distributed Storage*. Synthesis Lectures on Distributed Computing Theory. Morgan & Claypool Publishers, 2021. `doi:10.2200/S01069ED1V01Y202012DCT017`.

24 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.

25 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.

26 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.

27 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. `doi:10.1109/NCA.2016.7778622`.

28 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 Storage (TOS)*, 2022. Accepted. Also in `arXiv:1805.03727`.

29 Satadru Pan, Yunqiao Zhang, Atul Sikaria, Pavel Zakharov, Abhinav Sharma, Shiva Shankar, Mike Shuey, Richard Wareing, Monika Gangapuram, Guanglei Cao, Christian Preseau, Pratap Singh, Kestutis Patiejunas, J R Tipton, Theano Stavrinos, Ethan Katz-Bassett, and Wyatt Lloyd. Facebook's Tectonic Filesystem: Efficiency from Exascale, 2021. URL: `https://www.usenix.org/conference/fast21/presentation/pan`.

30 M O Rabin. Fingerprinting by random polynomials, 1981. URL: `http://www.xmailserver.org/rabin.pdf`.

31 Irving S. Reed and Gustave Solomon. Polynomial codes over certain finite fields. *Journal of The Society for Industrial and Applied Mathematics*, 8:300–304, 1960.

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

33 A. Tridgell and P. Mackerras. The rsync algorithm. *Imagine*, 1996.
34 P. Viotti and M. Vukolic. Consistency in non-transactional distributed storage systems. *ACM Computing Surveys (CSUR)*, 49:1–34, 2016.

A Our and Prior Work: A Comparative Table

Table 1 presents a comparison of the main characteristics of the distributed algorithms and storage systems. As we can see, systems that use relaxed or eventual consistency have serious issues when conflicting writes appear. Others guarantee strong consistency but have centralized components. To our opinion the most appropriate model able to provide high consistency, concurrency and availability seems to be the Atomic / Linearizability Consistency model. Some previous attempts, such as LDR [16], were promising, but they seem to suffer from communication delays and communication overheads since the whole object is still transmitted in every message exchanged between the clients and the replica servers. Also, the table shows well-known algorithms for reconfigurable atomic storage.

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 [28], as the optimization has no effect on this operation, so it should preserve safety as shown in [28].

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, $|c.Servers| = n$ of which no more than $\frac{n-k}{2}$ may crash, and that δ 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 14** (C2). *Let ξ be an execution of an algorithm A that uses the* EC-*DAPopt. If ϕ is a c.*get-data() *that returns $\langle \tau_\pi, v_\pi \rangle \in \mathcal{T} \times \mathcal{V}$, then there exists π such that π is a c.*put-data($\langle \tau_\pi, v_\pi \rangle$) *and ϕ did not complete before the invocation of π. If no such π exists in ξ, then (τ_π, v_π) is equal to (t_0, v_0).*

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 $\mathcal{S}$ is still $\{(t_0, \Phi_s(v_\pi))\}$, 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 15** (C1). *Let ξ be an execution of an algorithm A that uses the* EC-*DAPopt. If ϕ is c.*put-data($\langle \tau_\phi, v_\phi \rangle$)*, for $c \in \mathcal{C}$, $\langle \tau_\phi, v_\phi \rangle \in \mathcal{T} \times \mathcal{V}$, and π is c.*get-data() *that returns $\langle \tau_\pi, v_\pi \rangle \in \mathcal{T} \times \mathcal{V}$ and $\phi \to \pi$ in ξ, then $\tau_\pi \geq \tau_\phi$.*

Proof. Let p_ϕ and p_π denote the processes that invoke ϕ and π in ξ. Let $S_\phi \subset \mathcal{S}$ denote the set of $\lceil \frac{n+k}{2} \rceil$ servers that responds to p_ϕ, during ϕ, and by S_π the set of $\lceil \frac{n+k}{2} \rceil$ servers that responds to p_π, during π.

Per Alg. 5:13, 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. 5:14–17).

Table 1 Comparative table of distributed algorithms and storage systems.

Algorithm/ System	Data scalability	Data access Concurrency	Consistency guarantees	Versioning	Data Striping	Non-blocking Reconfiguration
GFS [21]	YES	concurrent appends	relaxed	YES	YES	YES (short downtime)
HDFS [6]	YES	files restrict one writer at a time	strong (centralized)	NO	YES	YES
Cassandra [1]	YES	YES	tunable (default= eventual)	YES	NO	NO
Dropbox [4]	YES	creates conflicting copies	eventual	YES	YES	N/A
Colossus [2]	YES	concurrent appends	relaxed	YES	YES	YES
Blobseer [14]	YES	YES	strong (centralized)	YES	YES	YES
Tectonic [29]	YES	files restrict one writer at a time	strong	YES	YES	YES
CoABD [27]	NO	YES	strong	YES	NO	NO
CoBFS [8]	YES	YES	strong	YES	YES	NO
RAMBO [22]	NO	NO	strong	NO	NO	YES
DynaStore [7]	NO	NO	strong	NO	NO	YES
SM-Store [25]	NO	NO	strong	NO	NO	YES
SpSnStore [17]	NO	NO	strong	NO	NO	YES
AresABD [28]	NO	NO	strong	NO	NO	YES
AresEC [28]	NO	NO	strong	NO	YES	YES
CoAresABD [our work]	NO	NO	strong	YES	NO	YES
CoAresEC [our work]	NO	NO	strong	YES	YES	YES
CoAresABDF [our work]	YES	YES	strong	YES	YES	YES
CoAresECF [our work]	YES	YES	strong	YES	YES (2 striping methods)	YES

When replying to π, each server in S_π includes a tag in $List'$, only if the tag is larger or equal to the tag associated to the last value decoded by p_π (lines Alg. 5:6–9). Notice that as $|S_\phi| = |S_\pi| = \lceil\frac{n+k}{2}\rceil$, the servers in $|S_\phi \cap S_\pi| \geq k$ reply to both π and ϕ. 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 π, and (b) the $\langle\tau_\phi, v_\phi\rangle$ appeared in fewer than k servers in S_π.

Case a: In the first case, since π discovered τ_ϕ in at least k servers it follows by the algorithm that the value associated with τ_ϕ will be decodable. Hence $t^{dec}_{max} \geq \tau_\phi$ and $\tau_\pi \geq \tau_\phi$.

Case b: In this case τ_ϕ was discovered in less than k servers in S_π. Let τ_ℓ denote the last tag returned by p_π. We can break this case in two subcases: (i) $\tau_\ell > \tau_\phi$, and (ii) $\tau_\ell \leq \tau_\phi$.

In case (i), no $s \in S_\pi$ included τ_ϕ in $List'_s$ before replying to π. By Lemma 14, the c.put-data($\langle\tau_\ell, *\rangle$) was invoked before the completion of the $*$.get-data() operation from p_π that returned τ_ℓ. It is also true that p_π discovered $\langle\tau_\ell, *\rangle$ in more than k servers since it managed to decode the value. Therefore, in this case $t^{dec}_{max} \geq \tau_\ell$ and thus $\tau_\pi > \tau_\phi$.

In case (ii), a server $s \in S_\phi \cap S_\pi$ will not include τ_ϕ iff $|Lists'_s| = \delta + 1$, and therefore the local $List$ of s removed τ_ϕ as the smallest tag in the list. According to our assumption though, no more than δ 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 π. Since τ' is also written in $|S'| = \frac{n+k}{2}$ servers then $|S_\pi \cap S'| \geq k$ and hence π will be able to decode the value associated to τ', and hence $t^{dec}_{max} \geq \tau_\ell$ and $\tau_\pi > \tau_\phi$, completing the proof of this lemma. ◀

▶ **Theorem 16** (Safety). *Let ξ be an execution of an algorithm A that contains a set Π of complete* get-data *and* put-data *operations of Algorithm 4. Then every pair of operations $\phi, \pi \in \Pi$ satisfy Property 1.*

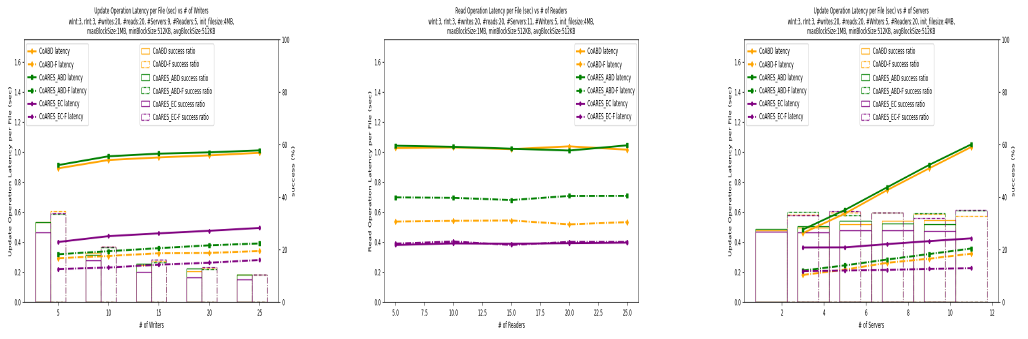
Proof. Follows directly from Lemmas 14 and 15. ◀

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.

▶ **Theorem 17** (Liveness). *Let ξ be an execution of an algorithm A that utilises the* EC-DAPopt*. Then any* put-data *or* get-data *operation π invoked in ξ will eventually terminate.*

Proof. Given that no more than $\frac{n-k}{2}$ servers may fail, then from Algorithm 4 (lines Alg. 4:21–29), it is easy to see that there are at least $\frac{n+k}{2}$ 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 π. Let p_ω and p_π be the processes that invoke the put-data operation ω and get-data operation π. Let S_ω be the set of $\lceil\frac{n+k}{2}\rceil$ servers that responds to p_ω, in the put-data operations, in ω. Let S_π be the set of $\lceil\frac{n+k}{2}\rceil$ servers that responds to p_π during the get-data step of π. Note that in ξ at the point execution T_1, just before the execution of π, none of the write operations in Λ is complete. Let T_2 denote the earliest point of time when p_π receives all the $\lceil\frac{n+k}{2}\rceil$ responses. Also, the set Λ includes all the put-data operations that starts before T_2 such that $tag(\lambda) > tag(\omega)\}$. Observe that, by algorithm design, the coded-elements corresponding to t_ω are garbage-collected from the $List$ variable of a server only if more than δ 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 t_ω is not garbage collected in ξ, at least until execution point T_2 in any of the servers in S_ω. Therefore, during the execution fragment between the execution points T_1 and T_2 of the execution ξ, the tag and coded-element pair is present in the $List$ variable of every server in S_ω that is active. As a result, the tag



(a) (b) (c)

Figure 4 Emulab results for Scalability experiments.

and coded-element pairs, $(t_\omega, \Phi_s(v_\omega))$ exists in the *List* received from any $s \in S_\omega \cap S_\pi$ during operation π. 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 $t_\omega \in Tags_{dec}^{\geq k}$, the set of decode-able tag, i.e., the value v_ω can be decoded by p_π in π, which demonstrates that $Tags_{dec}^{\geq k} \neq \emptyset$.

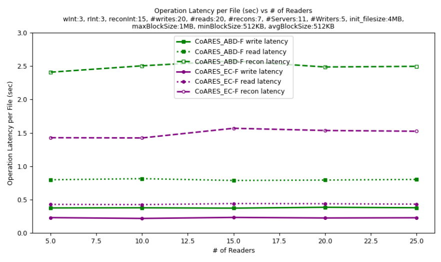
Next we want to argue that t_{max}^{dec} is the maximum tag that π discovers via a contradiction: we assume a tag t_{max}, which is the maximum tag π discovers, but it is not decode-able, i.e., $t_{max} \notin Tags_{dec}^{\geq k}$ and $t_{max} > t_{max}^{dec}$. Let $S_\pi^k \subset S$ be any subset of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

C Additional Experimental Results

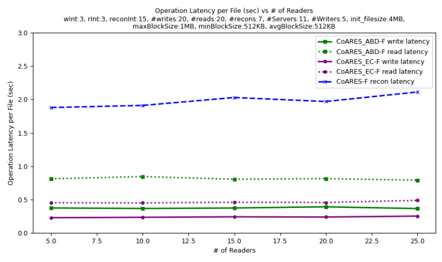
C.1 Performance VS. Scalability of Nodes Under Concurrency

This scenario is constructed to compare the read, write and recon latency of the algorithms, as the number of service participants increases.

Without Reconfiguration. In both Emulab and AWS, we varied the number of readers $|\mathcal{R}|$ and the number of writers $|\mathcal{W}|$ from 5 to 25, while the number of servers $|\mathcal{S}|$ 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. To match the fault-tolerance of ABD-BASED algorithms, we used a different parity for EC-BASED algorithms (except in the case of 3 servers to avoid replication). With this, the EC client has to wait for responses from a larger quorums. The parity value of the EC-BASED algorithms is set to $m = 1$ for $|\mathcal{S}| = 3$, $m = 2$ for $|\mathcal{S}| = 5$, $m = 3$ for $|\mathcal{S}| = 7$, $m = 4$ for $|\mathcal{S}| = 9$ and $m = 5$ for $|\mathcal{S}| = 11$.



■ **Figure 5** Emulab results when Reconfiguring to the Same DAP_s.

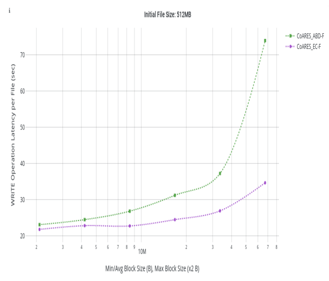


■ **Figure 6** Emulab results when Reconfiguring DAP_s Randomly.

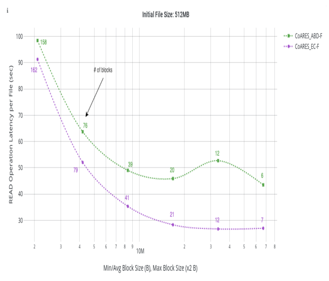
Results. The results obtained in this scenario are presented in Fig. 4. As expected, CoAresEC has the lowest update latency among non-fragmented algorithms because of the striping level. Each object is divided into k encoded fragments that reduce the communication latency (since it transfers less data 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 Fig. 4(a)).This is because the non-fragmented writer updates the whole file, while each fragmented writer updates only a subset of blocks. We observe that the update operation latency in algorithms CoABD and CoAresABD increases as the number of servers increases, while the operation latency of CoAresEC decreases or stays the same (Figs. 4(c)) That 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 waits for responses the decreased message size. When going from 7 to 9 servers, we observe a decrease in latency. This is due the choice of parity value (parameter of EC-based algorithms) that we select for 7 servers. Due to the block allocation strategy in fragment algorithms, more data are successfully written (cf. Fig. 4(a), 4(b)), explaining the slower CoAresF read operation (cf. Figs. 4(b)). The corresponding AWS findings show similar trends.

With Reconfiguration. We built four extra experiments in Emulab to verify the correctness of the variants of Ares when reconfigurations coexist with read/write operations. The experiments differ in the way the reconfigurer works; three experiments use $|\mathcal{S}| = 11$ and are based on the way the reconfigurer chooses the next storage algorithm (i.e., two reconfiguring to the same DAP and one reconfiguring to a random DAP); one in which the reconfigurer changes the storage algorithm and the quorum of servers. In the latter scenario the reconfigurer chooses randomly between $[3, 5, 7, 9, 11]$ servers. All experiments run on CoAres and CoAresF use one reconfigurer.

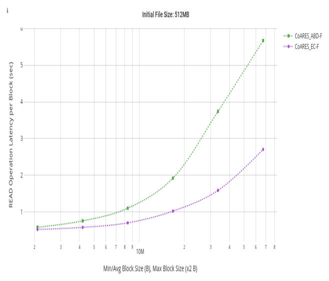
Results. Due to space limit, we report only one of the experiments (all results can be found in [20]). As we mentioned earlier, our choice of k minimizes the coded fragment size but introduces bigger quorums and thus larger communication overhead. As a result, in smaller file sizes, Ares (either fragmented or not) may not benefit so much from the coding, bringing the delays of the CoAresEC and CoAresABD closer to each other (cf. Fig. 5). However, the read latency of CoAresECF is significant lower than of CoAresABDF. This is because the CoAresECF takes less time to transfer the blocks to the new configuration.



(a)



(b)



(c)

Figure 7 AWS results for Min/Avg/Max Block Sizes' experiments.

C.2 Performance VS. Block Sizes

This scenario evaluates how the block size impacts the latencies when having a rather large file size. We varied the minimum and average b_{sizes} from 2 MB to 64 MB and the maximum b_{size} from 4 MB to 1 GB. In total, each writer performs 20 writes and each reader 20 reads. The size of the initial file used was set to 512 MB.

Emulab parameters: $|\mathcal{W}| = 5, |\mathcal{R}| = 5, |\mathcal{S}| = 11$. For EC-BASED algorithms, $m = 1$ and the quorum size is 11. For ABD-BASED algorithms we used quorums of size 4.

AWS parameters: $|\mathcal{W}| = 1, |\mathcal{R}| = 1, |\mathcal{S}| = 6$. For EC-BASED algorithms, $m = 1$ and the quorum size is 6. For ABD-BASED algorithms we used quorums of size 4.

Results. 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 7(a)). CoAresECF 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. 7(c)) has a significant impact on overall latency, resulting in a larger read latency (Fig. 7(b)).