Boosting Concurrency and Fault-Tolerance for Reconfigurable Shared Large Objects

ANDRIA TRIGEORGI, University of Cyprus, Aglantzia, Cyprus and Algolysis Ltd, Erimi, Cyprus
NICOLAS NICOLAOU, Algolysis Ltd, Erimi, Cyprus
CHRYSSIS GEORGIOU, University of Cyprus, Aglantzia, Cyprus
ANTONIO FERNÁNDEZ ANTA, IMDEA Networks Institute, Madrid, Spain
THEOPHANIS HADJISTASI, Algolysis Ltd, Erimi, Cyprus
EFSTATHIOS STAVRAKIS, Algolysis Ltd, Erimi, Cyprus

Nowadays the traditional file systems cannot handle the new requirements in terms of volume of data, high performance, fault-tolerance, and improved capabilities. So Distributed Storage Systems (DSS) took place to cover the need of a shared storage between separate systems, provide a scalable storage to serve thousands of servers, and improve the fault-tolerance. To this respect, a series of issues need to be properly addressed: scalability, the ability to handle large data, high performance even under heavy access concurrency, versioning, and fault-tolerance.

In this work, we propose CoBFS, a framework of a DSS designed to boost the concurrent access to large shared data objects (such as files), while maintaining strong consistency guarantees. CoBFS has two key design factors: data striping and versioning-based concurrency control (through coverability) to enable higher operation performance on large concurrent data objects. To this respect, we introduce the notions of a *block* as a "bounded" Read/Write register, of a *fragmented object* as a sequence of blocks, and of *fragmented coverable linearizability*, a strong consistency property suitable for fragmented objects.

CoBFS adopts a modular architecture, separating the object fragmentation process from the shared memory service allowing to use different shared memory implementations. At first, we use as storage a static atomic distributed shared memory (ADSM) emulation, the well known ABD, yielding CoABDF, which satisfies fragmented coverable linearizability. Then, we substitute the storage layer of CoBFS with a dynamic (reconfigurable) storage algorithm, called Ares, yielding CoAresF; CoAresF allows the addition and removal of servers without system interruptions and improves the storage efficiency due to the use of an erasure-coded mechanism. We conduct an extensive experimental evaluation on the Emulab and AWS EC2 testbeds, illustrating the benefits of our approaches, as well as other interesting tradeoffs. We believe that CoBFS's features (versioning, high concurrent accesses, handling large objects) has the potential of benefiting any static or dynamic storage algorithm to further extend its functionality for data-intensive applications at large scale.

All authors have contributed equally to this research.

This work has been funded by the Cyprus Research and Innovation Foundation, under the grant agreements POST-DOC/0916/0090, DUAL USE/0922/0048 and PHD IN INDUSTRY/1222/0121.

Authors' Contact Information: Andria Trigeorgi, University of Cyprus, Aglantzia, Nicosia, Cyprus and Algolysis Ltd, Erimi, Limassol, Cyprus; e-mail: andria.trigeorgi@algolysis.com; Nicolas Nicolaou, Algolysis Ltd, Erimi, Limassol, Cyprus; e-mail: nicolas@algolysis.com; Chryssis Georgiou (corresponding author), University of Cyprus, Aglantzia, Nicosia, Cyprus; e-mail: chryssis@ucy.ac.cy; Antonio Fernández Anta, IMDEA Networks Institute, Madrid, Madrid, Spain; e-mail: antonio.fernandez@imdea.org; Theophanis Hadjistasi, Algolysis Ltd, Erimi, Limassol, Cyprus; e-mail: hadjistasi@gmail.com; Efstathios Stavrakis, Algolysis Ltd, Erimi, Limassol, Cyprus; e-mail: stathis@algolysis.com.

ACM 1553-3077/2026/03-ART20
https://doi.org/10.1145/3789204

CCS Concepts: • **Computing methodologies → Distributed algorithms**; • **Computer systems organization → Dependable and fault-tolerant systems and networks;**

Additional Key Words and Phrases: Distributed storage, large objects, strong consistency, reconfiguration, fragmented objects

ACM Reference Format:
Andria Trigeorgi, Nicolas Nicolaou, Chryssis Georgiou, Antonio Fernández Anta, Theophanis Hadjistasi, and Efstathios Stavrakis. 2026. Boosting Concurrency and Fault-Tolerance for Reconfigurable Shared Large Objects. *ACM Trans. Storage* 22, 2, Article 20 (March 2026), 55 pages. https://doi.org/10.1145/3789204

1 Introduction

Motivation and Prior Work. The exponential growth of data leads to the continued improvements in the efficiency of storage technology. A **Distributed Storage System** (**DSS**) [57, 59] can split data across multiple servers for high availability, data redundancy, and recovery purposes. In such a replicated environment, consistency must be enforced across the data servers. When a replicated storage system behaves identical to that of a storage system running on a single machine, we say that it is strongly consistent. Leslie Lamport [45, 46] was the first to define the notion of atomic register, which is the strongest consistency semantic and provides the illusion that the storage is accessed sequentially. Herlihy and Wing [39] generalize Lamport's notion of atomic registers by defining the notion of *linearizability*. According to the CAP theorem [35], it is impossible to provide both strong consistency, in particular linearizability, and available operation in the presence of partitions (i.e., network failures). Thus, numerous platforms prefer high availability over consistency, due to the belief that strong consistency will burden the performance of their systems. Fault tolerance is another important property of distributed systems. It is the ability of a system to continue to be both available and reliable in the presence of failures. In synchronous systems, it is possible to detect a crash failure (using heartbeats or timeouts). Fischer, Lynch and Paterson [26] presented the FLP Impossibility result, which states that it is *impossible* in an asynchronous system for a set of nodes to reach agreement if at least one node fails, even by crashing; asynchrony prevents distinguishing between a process that has crashed and a process that is running slowly.

There are **Distributed Shared Memory** (**DSM**) emulations in theory and practical DSS in the market, and they are two distinct concepts for handling storage in distributed environments. On the one hand, a series of research works spanning over two decades, such as [2, 5, 19, 28, 30, 36, 49, 51], suggested solutions for building **Atomic Distributed Shared Memory** (**ADSM**) emulations, for both static, i.e., where replica participation does not change over time, and dynamic (reconfigurable) environments, i.e., where failed replicas may retire and new replicas may join the service in a non-blocking manner. Variations like coverability [50], were also introduced to guarantee version history and consistency across updates. Coverability extends linearizability to versioned objects by ensuring operations are ordered according to their versions, thus preventing outdated writes from overwriting newer ones. Currently, ADSM emulations are either limited to small-size shared objects or, if two writes occur concurrently on different parts of the object, only one of them prevails. These theoretical ADSMs come with rigorous proofs that guarantee the desired properties, such as atomic consistency and fault-tolerance, under specific assumptions. On the other hand, DSSs prioritize availability and scalability over strong consistency. As a result, they either often provide weaker consistency guarantees (like weak [57] and eventual consistency [42, 60]), devise strategies with limited concurrency (e.g., files in HDFS [38] restrict one writer at a time), or rely on centralized solutions to provide strong consistency. However, the guarantees they offer are often based on empirical evidence and real-world performance. Given the limitations of existing atomic algorithms and new challenges that arise in the context of exponentially growing data sizes, this work aims at

demonstrating, with provable guarantees, that it is possible to build robust and strongly consistent DSSs while providing highly concurrent access to its users at large scale.

Our Contributions. We propose CoBFS, a framework of a DSS designed to boost the concurrent access to large shared data objects (such as files), while maintaining strong consistency guarantees. CoBFS has two key design factors: data striping and versioning-based concurrency control (through coverability [50]) to enable higher operation performance on large concurrent data objects. CoBFS adopts a modular architecture, separating the object fragmentation process from the distributed shared memory implementation. rendering the CoBFS framework compatible with multiple application needs. Thus, CoBFS is the composition of two main modules: (*i*) **Fragmentation Module** (**FM**), and (*ii*) **Distributed Shared Memory Module** (**DSMM**). In the sequel, we present an implementation of the FM, and two implementations for the DSMM: one for the static environment based on the CoABD [50] algorithm, and one for the dynamic environment based on the Ares [51] algorithm. In particular, using CoABD we develop a static algorithm, we call CoABDF, that: (*i*) supports versioned objects, and (*ii*) is suitable for large objects. Then, we develop a dynamic algorithm, called CoAresF, which builds on Ares to bring versioning and large object support in dynamic systems, and in addition to introduce: (*i*) long-liveness, and (*ii*) storage-efficiency. We believe that our work makes a leap toward dynamic DSSs that are attractive for practical applications.

For clarity, a glossary mapping the main acronyms and algorithm names used throughout the article is provided at the end of this section (see Table 1).

Key contributions of this work include the definition of new data structures for fragmented objects, the introduction of a consistency model called *Fragmented Coverable Linearizability*, optimization of read/write operations in the distributed memory layer, integration of fragmentation with both CoABD and Ares algorithms to support large versioned objects, and the demonstration of superior scalability and performance through extensive experimental evaluation. In more detail:

- To form the data structure in our fragmentation strategy, we define two types of concurrent objects: (i) the *block* object, and (ii) the *fragmented* object. Blocks are treated as R/W objects, while fragmented objects are defined as sequences of block objects (Section 4.1).
- To enable concurrent modifications, we examine the consistency properties when allowing R/W operations on individual blocks of the fragmented object. Assuming that each block is linearizable, we define the precise consistency that the fragmented object provides, termed *Fragmented Coverable Linearizability* (Section 4.2).
- We provide a framework, called CoBFS, suitable for fragmented objects. CoBFS adopts a modular architecture, separating the object fragmentation process from the shared memory service, which allows to follow different fragmentation strategies and shared memory implementations (Section 4.3).
- Then we provide an implementation of a FM designed for seamless integration within the CoBFS framework (Section 5).
- We introduce a DSMM implementation based on an *optimized coverable variant* of the ABD algorithm, referred to as CoABD [50]. This optimization results in a reduced operational latency for read/write operations in the DSMM layer (Section 6.1).
- The integration of CoABD with the FM yields CoABDF, designed to handle *large* shared data objects and increased data access concurrency. We show that CoABDF preserves the validity of the fragmented object and satisfies *fragmented coverable linearizability* (Section 6.2).
- We then extend Ares (cf. Section 7.1) to support coverable objects. Thus, 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 7.2).

Table 1. Glossary of Algorithm Names and Acronyms

Term	Description
ABD	Attiya-Bar-Noy-Dolev algorithm for atomic registers in static systems
EC	Erasure Coding – data encoding strategy for efficient storage
CoABD	Coverable static ABD (no fragmentation) – baseline
CoABDF	Fragmented CoABD – supports concurrent block updates
Ares	dynamic/reconfigurable atomic storage
ABD-DAP	ABD-based Data Access Primitives used in Ares variants
EC-DAP	EC-based Data Access Primitives used in Ares variants
EC-DAPopt	Optimized EC-DAP
AresABD	Ares using ABD-based DAP
AresEC	Ares using EC-based DAP
CoAresABD	Coverable Ares using ABD-based DAP
CoAresABDF	Fragmented CoAresABD – supports fragmentation and concurrent updates
CoAresEC	Coverable Ares using EC-based DAP
CoAresECF	Fragmented CoAresEC – uses two-level striping
Cassandra	Commercial distributed storage system (baseline key-value store)
CassandraF	Chunked version of Cassandra to support large objects

— Then we adopt the idea of fragmentation as presented in CoBFS and integrate it with CoAres to obtain CoAresF (Section 7.4). 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 Ares (thus, by CoAres and CoAresF). This optimization has its own interest, as it could be applicable beyond Ares, i.e., by other erasure coded algorithms relying on tag-ordered DAPs (Section 7.3).
— We have performed an in-depth experimental evaluation to compare the performance of CoBFS using different storage emulations (CoABD, CoAresABD, CoAresEC) against the original ones without the fragmentation approach of CoBFS over both Emulab [20], a popular emulation testbed, and **Amazon Web Services** (**AWS**) EC2 [8], an overlay testbed (Section 8). 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 also compare against the commercial DSS Cassandra and its chunked variant CassandraF, showing that CoBFS consistently outperforms both, demonstrating superior scalability and performance for large file workloads.

2 State-of-the-art

We begin by discussing some of the early works on replication-based atomic storage. Next, we delve into reconfigurable atomic storage. Finally, we overview the main characteristics of prominent Distributed Storage Systems while presenting the strengths and weaknesses of each of them.

Replication-based Atomic Storage. There are many prior works [5, 19, 22, 28, 30, 37, 49] implementing algorithms for atomic (linearizable) shared memory emulation. Attiya, Bar-Noy and Dolev [5] present the first fault-tolerant emulation of atomic shared memory in an asynchronous message passing system, also known as ABD. It implements **Single-Writer Multi-Reader** (**SWMR**) registers in an asynchronous network, provided that at least a majority of the servers do not crash. The

writer completes write operations in a single round by incrementing its local timestamp and propagating the value with its new timestamp to the servers, waiting acknowledgments from a majority of them. The timestamp, represented as an integer, enables the system to keep track of old values in the face of asynchrony. The read operation completes in two rounds: (*i*) it discovers the maximum timestamp-value pair that is known to a majority of the servers, (*ii*) it propagates the pair to the servers, in order to ensure that a majority of them have the latest value, hence preserving atomicity.

The ABD algorithm was extended by Lynch and Shvartsman [49], who present an emulation of MWMR atomic registers in message-passing systems. Also, they generalized majorities into *quorums* [61]. A quorum is a subset of the set of servers, and each quorum has a common intersection with every other quorum in the system. In this work, both write and read operations perform 2 rounds, a query round to discover the maximum tag, and a propagation round to inform the servers of its local tag. The read operation is identical to the two-round protocol in SWMR ABD, the only difference being that tags are used instead of timestamps. A tag is a tuple (ts, id), where ts is a logical timestamp, and id is the writer's unique id that distinguishes the current write operation from all others. The only difference between a write and a read lies in the second round, where the write operation increments the maximum tag, while the read operation propagates this maximum tag. The first round ensures that the writer produces a tag that is higher than that of any preceding write.

In shared objects, a write operation should extend the latest written version of the object, and not overwrite any new value. In this respect, some works [14, 50] introduce consistency guarantees that extends linearizability in order to provide refined correctness conditions for concurrent objects. The notion of *coverability*, introduced in [50], extends linearizability and concerns versioned objects (cf. Definition 4 in Section 3). Coverability ensures that the operations on versioned objects are ordered with respect to their versions. This notion is useful in scenarios where versioning and historical consistency are crucial, such as in distributed storage systems. The original work [50] also presented an implementation of a coverable (versioned) object , which we call CoABD. This implementation is essentially the classical MWMR ABD algorithm enhanced with version tags, where each value is linked to a specific tag. Read operations are identical to those of MWMR ABD, with the difference that they return both the version and the value of the object. Write operations, on the other hand, attempt to write a "versioned" value on the object. If the reported version is older than the latest version of the object, then the write does not take effect and is converted into a read operation. This ensures that writes have updated the previously latest version of the object.

More flexible notions of linearizability are the set- and interval-linearizability [14]. *Set-linearizability* allows for the simultaneity of operations by grouping them into sets. Operations within the same set (as concurrency classes) are considered to occur simultaneously. This is useful when operations on concurrent objects need to be grouped together to reflect certain aspects of their behavior. *Interval-linearizability* extents set-linearizability by allowing operations to appear as executed concurrently with several consecutive, non-overlapping operations. Thus, an operation can span an interval of time, overlapping with other operations. This is useful when operations on concurrent objects can span longer durations or when there is a need to capture the timing of operations.

Several works have followed, presenting different ways to achieve one-round reads (e.g., SLIQ [33], CwFr [29], OhSam [37], OhMam [37], Fast [24], SFW [21], SF [30], ccHybrid [24], OhFast [24], Erato [28]). The above works on shared memory emulations were focused on small-size objects. Fan and Lynch [22] proposed an extension called *LDR*, which can cope with large-size objects such as files. The key idea was to maintain copies of the data objects separately from their metadata, involving two types of servers: replicas (storing the files) and directories (handling metadata and essentially running a version of ABD). However, in this approach, the entire object is still transmitted in every message exchanged between the clients and the replica servers. Furthermore, if two writes update different parts of the object concurrently, only one of them prevails.

Reconfigurable Atomic Storage. We now consider distributed storage systems that work in dynamic asynchronous environments, that is, the servers maintain the storage change over time. The set of servers maintaining the storage is called a *configuration*, and the process of changing the set is called a *reconfiguration*. Below, we describe algorithms for reconfigurable atomic storage, with and without consensus. A main challenge when supporting reconfiguration is to ensure consistency when multiple users submit concurrent reconfiguration requests.

Early implementations of reconfigurable storage systems, such as RAMBO [36] rely on consensus [43]. When several reconfiguration requests are issued concurrently, the clients can use consensus to agree on the next configuration and then transfer the state of the object to this new configuration. However, it was later discovered that replicated services that do not require consensus can be reconfigured in a purely asynchronous way. DynaStore [2] was the first asynchronous consensus-free implementation of reconfigurable storage. RAMBO uses consensus to choose only one successor for every configuration, whereas in DynaStore configurations can have multiple successors, while *lattice agreement* [43] or the **Speculating Snapshot** (***SpSn***) algorithm [27] can ensure that configurations are ordered. i.e., for two configurations, the changes realized in one of them are also part of the other. Lately, a lot of progress was made in defining abstract asynchronous reconfiguration in a simple and efficient way. Examples of works that improved in terms of efficiency, simplicity and modularity are the SM-Store [40] and SpSnStore [27]. Ares [51] is another reconfigurable algorithm, designed as a modular framework to implement dynamic, reconfigurable, fault-tolerant, read/write distributed linearizable (atomic) shared memory objects. Ares, like other reconfigurable algorithms, copes with changing participants via reconfigurations. Due to this modularity, Ares allows for reconfiguration between completely different protocols in principle, as long as they can be expressed using the primitives presented in Section 7.1, called DAPs. Ares shares similarities with consensus algorithms like RAMBO. Similar to DynaStore, Ares also requires reading of reconfiguration information (more than once in some cases) for each read and write operation. However, it is the only algorithm to combine a dynamic behavior with the use of *Erasure Codes*, while reducing the storage and communication costs associated with the read or write operations. Yet, the need to effectively handle large objects remains. Moreover, in Ares the number of rounds per write and read is at least as good as in any of the above algorithms. For a more detailed description of Ares, we refer the reader to Section 7.1.

Distributed Storage Systems. Scalability is the main concern that these systems manage to address. Those systems are designed to manage increasing amounts of data in an appropriate manner. The above systems are built to tolerate failures [10]. These failures may include crashes, hardware failures, or other types of faults.

The first-generation distributed file systems of Google and Facebook, that is, GFS [34] and HDFS [38] respectively, were not scalable enough because their centralized components did not allow them to support huge amounts of metadata. Thus, both Google and Facebook introduced next-generation storage systems that provide a more scalable and highly available metadata system. Google continues with Colossus [15] and Facebook with Tectonic [53]. Both systems store their metadata in a key-value store. Google used its BigTable key-value store to store Colossus' metadata. Facebook chose the ZippyDB key-value store for Tectonic.

Cassandra [44], Redis [55], Ceph [62] are also well-used high-performance data stores. Cassandra – a key-value store again introduced by Facebook – does not have a master-slave architecture; it follows a masterless architecture. Cassandra can work with multiple small records (structured data, record-level indexing), while large unstructured files are split into multiple parts. Cassandra is a disk-based, distributed NoSQL database that prioritizes availability and durability. Redis, on the other hand, is an in-memory, real-time data store that sacrifices durability for speed. Last but

not least, Ceph emphasizes scalability, fault tolerance, and flexibility. Unlike systems such as HDFS, Ceph does not rely on a centralized NameNode; instead, it distributes metadata across the cluster using a distributed placement algorithm.

With regards to consistency guarantees, Blobseer provides linearizability by using a central version manager to order the accesses and the values written on the data. HDFS stores the metadata centrally on a single device – hence trivially achieving linearizable metadata access – and adopts a write-once architecture for data access. On the other hand, Tectonic provides a type of causal consistency, more specifically read-your-writes consistency. This means that when a user performs changes, they can view these changes (read data) right after making those changes. However, like its ancestor, HDFS, it allows a single writer per file. This simplifies the complexity of serializing writes to a file from multiple writers. Nevertheless, if a tenant needs multi-writer semantics, it will have to build its own write serialization semantics on top of Tectonic. Ceph provides strong consistency for metadata and read-your-writes semantics for data, which allows clients to observe their own changes. However, Ceph does not prevent concurrent writes to the same object, leaving applications responsible for coordinating access and preventing conflicts. This design allows high throughput and scalability but sacrifices stronger guarantees like linearizability unless additional mechanisms are employed at the application level. Additionally, Cassandra offers tunable consistency for read and write operations, allowing the system to guarantee weaker or stronger consistency, as required by the client application. Finally, ⍰ differs from other file systems as it simply syncs files between local storage and the cloud for backup, sharing, and remote access. There is a limited number of articles, e.g., [9, 12, 17, 41, 47, 53], that perform proof-of-concept experimental analysis of the above DSSs, in order to verify that the DSS will function as envisioned and not violate its guarantees.

Summary. Table 2 presents a comparison of the main characteristics of the distributed algorithms and storage systems, including the algorithms that are developed and implemented in the current work (CoBFS, CoAresABD, CoAresEC, CoAresABDF, CoAresECF). The discussed algorithms and systems are assessed in terms of data scalability, data access concurrency, consistency guarantees, versioning, data striping, and reconfiguration capabilities. These aspects encompass the systems' abilities to handle growing data requirements, manage conflicting data accesses, follow specific consistency models, record versions of changed data, employ data striping across disks, and support dynamic reconfiguration.

On the one hand, the advantage of ADSMs is that they come with *provable guarantees* on strong consistency and reconfiguration correctness. By strong consistency, we either mean linearizability (cf. Section3), coverable (cf. Section 3), or fragmented coverable linearizability (cf. Section 4.2). However, ADSMs require to transmit the entire object over the network per read and write operation. Moreover, if two concurrent write operations affect different "parts" of the object, only one of them would prevail, despite the updates not being directly conflicting. So, a challenge would be to make ADSMs suitable for handling large-sized objects while providing linearizable consistency guarantees.

On the other hand, there are dozens of DSSs that exist on the market today that can handle large objects. However, these systems provide weaker consistency guarantees, such as relaxed or eventual consistency, thus they have serious issues when conflicting writes occur. Relaxed [1] and eventual [42, 60] consistency types are weaker than linearizability. These consistency types tradeoff stronger consistency guarantees for higher availability, fault tolerance, and performance. Relaxed consistency introduces relaxations based on program order and write atomicity, which enable many optimizations, such as allowing writes to proceed without waiting and reads to return values before updates are visible to all processors. In addition, most of these systems, like GFS for example, use centralized components, and this may lead to high congestion or to a bottleneck effect.

Table 2. Comparative Table of Distributed Algorithms and Storage Systems

Algorithm/ System	Data scalability	Data access Concurrency	Consistency guarantees	Versioning	Data Striping	Non-blocking Reconfiguration
GFS [34]	YES	concurrent appends	relaxed	YES	YES	YES (short downtime)
Colossus [15]	YES	concurrent appends	relaxed	YES	YES	YES
HDFS [38]	YES	files restrict one writer at a time	strong (metadata), write-once-read-many (data)	NO	YES	YES
Ceph [62]	YES	concurrent writes managed by app	strong (metadata), read-your-writes (data)	YES (optional)	YES	YES
Cassandra [13]	YES	YES	tunable (default= eventual)	YES	NO	NO
Dropbox [18]	YES	creates conflicting copies	eventual	YES	YES	N/A
Redis [55]	YES	YES	eventual	YES	NO	NO
Blobseer [12]	YES	YES	strong (centralized)	YES	YES	YES
Tectonic [53]	YES	files restrict one writer at a time	strong (metadata), read-your-writes (data)	YES	YES	YES
ABD [5, 49]	NO	NO	strong	NO	NO	NO
CoABD [50]	NO	NO	strong	YES	NO	NO
CoABDF*	YES	YES	strong	YES	YES	NO
RAMBO [36]	NO	NO	strong	NO	NO	YES
DynaStore [2]	NO	NO	strong	NO	NO	YES
SM-Store [40]	NO	NO	strong	NO	NO	YES
SpSnStore [27]	NO	NO	strong	NO	NO	YES
AresABD [51]	NO	NO	strong	NO	NO	YES
AresEC [51]	NO	NO	strong	NO	YES	YES
CoAresABD*	NO	NO	strong	YES	NO	YES
CoAresEC*	NO	NO	strong	YES	YES	YES
CoAresABDF*	YES	YES	strong	YES	YES	YES
CoAresECF*	YES	YES	strong	YES	YES (two levels of striping)	YES

*The algorithms marked with an asterisk (*) are introduced in this work.

To the best of our knowledge, there was no commercial DSS that could provide provable atomic consistency guarantees in a decentralized environment in the absence of failure detection and coordination for large objects; and that was our motivation: to develop a reconfigurable DSS that provides and provably guarantees these characteristics.

3 System Settings and Definitions

We consider an asynchronous message-passing system in which processes communicate by exchanging messages via asynchronous point-to-point reliable channels; messages may be reordered. As mentioned, our main goal is to implement a strongly consistent system 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*, denoted by read(), that takes no arguments and returns the value of the object, and (*ii*) a *write operation*, denoted by write(v), that modifies the value of the object to v.

Clients and Servers. The system is a collection of crash-prone, asynchronous processors with unique identifiers (ids) from a totally-ordered set $\mathcal{I}$, composed of two main disjoint sets of processes: (a) a set $\mathcal{C}$ of client processes ids that may perform operations on a replicated object, and (b) a set $\mathcal{S}$ of server processes ids that each holds a replica of the object.

A *quorum* is defined as a subset of $\mathcal{S}$. A *quorum system* [61] is a collection of pair-wise intersecting quorums. In this work, we address problems in both static and dynamic settings, based on the type of *configuration*. Configuration includes the set of servers and some additional information that is needed, while *reconfiguration* refers to the process of modifying or changing this setup, such as adding or removing servers. In a static environment, the configuration of the system, including the set of S, remains fixed, even if servers fail. In dynamic environments the configuration of the system may dynamically change over time due to servers removal or addition.

We consider three distinct sets of client processes: a set $\mathcal{W}$ of writers, a set $\mathcal{R}$ of readers, and a set $\mathcal{G}$ of reconfiguration clients (only for dynamic settings). 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 (cf. Section 7.1).

In the algorithms presented in this work, any subset of client processes and up to a certain number of servers (specified by the algorithm[1]) may crash at any time during an execution. A *crash* failure occurs when a node halts once, and then stops responding completely, and becomes unresponsive (aka, it crashes).

Executions, Histories and Operations. 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. We say that an operation π is *invoked* (starts) in an execution ξ when the *invocation action* of π appears in H_ξ, and π responds to the environment (ends or completes) when the *response action* appears in H_ξ. An operation is *complete* in ξ when both its invocation and *matching* response actions appear in H_ξ in that order. 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). We say that an operation π *precedes in real time* an operation π' (or π' *succeeds in real time* π) in an execution ξ, denoted by $\pi \rightarrow \pi'$, if the response of π appears before the invocation of π' in H_ξ. Two operations are *concurrent* if neither precedes the other.

[1]The correctness of the algorithm is associated with a bond on the number of server crashes tolerated.

Linearizability. We now formally define linearizability [39], following [7]. First, we define the notion of the sequential specification of a R/W object.

Definition 1 (Sequential Specification). The sequential specification of a R/W object O over history H is defined as follows: Initially, the value of the object O is $\perp$. If at the invocation event of an operation π in H the value of the object O is v, then:

— if π is a read() operation, then the response event of π returns v, and
— if π is a write(v') operation, then at the response event of π, the value of the object O is v'.

Definition 2 (Linearizability). A R/W object O is *linearizable* if, given any complete history H, there exists a permutation σ of all actions in H such that:

— σ is a sequential history and follows the sequential specification of the object O, and
— for operations π_1, π_2, if $\pi_1 \rightarrow \pi_2$ in H, then π_1 appears before π_2 in σ.

Coverability. Coverability [50] extends linearizability by ensuring that a write operation is performed on the latest version of the object. Thus, this consistency guarantee is defined over a *totally ordered* set of *versions*, say *Versions*, and introduces the notion of *versioned (coverable) objects*. According to [50], a *coverable object* is a type of R/W object where each value written is assigned with a version from the set *Versions*. Denoting a successful write as $cvr\text{-}\omega(ver)[ver', chg]_p$ (updating the object from version ver to ver'), and an unsuccessful write as $cvr\text{-}\omega(ver)[ver', unchg]_p$, a coverable implementation satisfies the properties of *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 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 now of produced versions 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 of $Versions_\xi$ are totally ordered.

Next, we present the *validity* property, which defines explicitly the set of executions (histories) that are considered to be valid executions (histories).

Definition 3 (Validity [50]). An execution ξ (respectively its history H_ξ) is a *valid execution* (respectively history) on a versioned object, for any $p_i, p_j \in \mathcal{J}$:

— $\forall cvr\text{-}\omega(ver)[ver']_{p_i} \in \mathcal{W}_{\xi,succ}, ver < ver'$,
— 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
— 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 $0 \leq i < k$.

Definition 4 (Coverability [50]). A valid execution ξ is **coverable** with respect to a total order $<_\xi$ on operations in $\mathcal{W}_{\xi,succ}$ if:

— (**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 $ver_i \leq ver_j$ and $\pi_1 <_\xi \pi_2$.

– (**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 $ver = ver_0$.
– (**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 $1 \leq i < k$, and $cvr\text{-}\omega(ver''_i)[ver''_{i+1}] \in \mathcal{W}_{\xi,succ}$, for $1 \leq i < \ell$, and $k < \ell$, then $ver' < ver''$.

Tags. We use logical tags to order operations. A tag τ is defined as a pair (ts, wid), where $ts \in \mathbb{N}$, a timestamp, and $wid \in \mathcal{W}$, an ID of a writer. Let $\mathcal{T}$ be the set of all tags. Notice that tags could be defined in any totally ordered domain and given that this domain is countably infinite, then there can be a direct mapping to the domain we assume. For any $\tau_1, \tau_2 \in \mathcal{T}$ we define $\tau_2 > \tau_1$ if (i) $\tau_2.ts > \tau_1.ts$ or (ii) $\tau_2.ts = \tau_1.ts$ and $\tau_2.wid > \tau_1.wid$. In the rest of the document, we use tags to declare the version of a shared object; thus, $\mathcal{T} = Versions$.

4 Framework for Fragmented Objects

In this section, we set the goal to study and formally define the consistency guarantees we can provide when fragmenting a large R/W object into smaller objects (blocks), so that operations are still issued on the former but are applied on the latter. Next, we introduce a framework, denoted as CoBFS, designed for the implementation of coverable fragmented objects.

To motivate this, consider a simple example where a string object stores "Hello world!" and is fragmented into two blocks: "Hello " and "world!". Suppose that two clients concurrently try to update the entire file, one to "Hi folks!" and another to "Hello all!". According to linearizability [39], the system must choose one order, either Update 1 → Update 2 ("Hello all!" persists) or Update 2 → Update 1 ("Hi folks!" persists). Coverability [50] extends linearizability by adding version awareness. If the first client starts at version v_0 ("Hello world!"), its update succeeds by creating v_1 ("Hi folks!"), while a concurrent write with stale v_0 fails and transparently becomes a read returning the current v_1 ("Hi folks!"). This prevents lost updates but still requires whole-object coordination. The new consistency guarantee we introduce in this section, called Fragmented linearizability relaxes the model by requiring linearizability only per block, thus concurrent updates to different blocks (e.g., changning "Hello" to "Hi" and "world!" to "all!") proceed without ordering the blocks, allowing states like "Hi world!" or "Hi all!" during updates. Fragmented coverability then adds version control: a stale update on the first block (e.g., using version v_1 when the current version is v_3) fails and returns the current value, while valid updates to other blocks succeed (e.g., second block from v_2 to "all!"). This combination provides block-level consistency with versioning, enabling us to support concurrent modifications while preventing overwrites.

4.1 Concurrent R/W Objects: Blocks and Fragmented Objects

A *fragmented object* is a concurrent object (e.g., can be accessed concurrently by multiple processes) that is composed of a finite sequence of *blocks*. Section 4.1.1 formally defines the notion of a *block*, and Section 4.1.2 gives the formal definition of a *fragmented object*.

4.1.1 Block Object. A *block* b is a concurrent R/W object with a unique identifier from a set $\mathcal{B}$. A block has a value $val(b) \in \Sigma^*$, extracted from an alphabet Σ. For performance reasons it is convenient to bound the block length. Hence, we denote by $\mathcal{B}^\ell \subset \mathcal{B}$, the set that contains bounded length blocks, s.t. $\forall b \in \mathcal{B}^\ell$ the length of $|val(b)| \leq \ell$. We use $|b|$ to denote the length of the value of b when convenient. An *empty block* is a block b whose value is the empty string ε, i.e., $|b| = 0$. Operation create(b, D) is used to introduce a new block $b \in \mathcal{B}^\ell$, initialized with value D, such that $|D| \leq \ell$. Once created, block b supports the following two operations: (i) read$()_b$ that returns the value of the object b, and (ii) write$(D)_b$ that sets the value of the object b to D, where $|D| \leq \ell$.

A block object is linearizable if is satisfies the linearizability properties (see Section 4.2) with respect to its create (which acts as a write), read, and write operations. Once created, a block object is an atomic register [48] whose value cannot exceed a predefined length ℓ.

4.1.2 Fragmented Object. A *fragmented object* f is a concurrent R/W object with a unique identifier from a set F. Essentially, a fragmented object is a *sequence* of blocks from $\mathcal{B}$, with a value $val(f) = \langle b_0, b_1, \dots, b_n \rangle$, where $b_i \in \mathcal{B}$, for $i \in [0, n]$. Initially, each fragmented object contains an empty block, i.e., $val(f) = \langle b_0 \rangle$ with $val(b_0) = \varepsilon$. We say that f is *valid* and $f \in F^\ell$ if $\forall b_i \in val(f)$, $b_i \in \mathcal{B}^\ell$. Otherwise, f is *invalid*. Being a R/W object, one would expect that a fragmented object $f \in F^\ell$, for any ℓ, supports the following operations:

- $\text{read}()_f$ returns the list $\langle val(b_0), \dots, val(b_n) \rangle$, where $val(f) = \langle b_0, b_1, \dots, b_n \rangle$
- $\text{write}(\langle D_0, \dots, D_n \rangle)_f$, $|D_i| \leq \ell$, $\forall i \in [0, n]$, sets the value of f to $\langle b_0, \dots, b_n \rangle$ s.t. $val(b_i) = D_i$, $\forall i \in [0, n]$.

Having the write operation to modify the values of all blocks in the list may hinder in many cases the concurrency of the object. For instance, consider the following execution ξ. Let $val(f) = \langle b_0, b_1 \rangle$, $val(b_0) = D_0$, $val(b_1) = D_1$, and assume that ξ contains two concurrent writes by two different clients, one attempting to modify block b_0, and the other attempting to modify block b_1: $\pi_1 = \text{write}(\langle D'_0, D_1 \rangle)_f$ and $\pi_2 = \text{write}(\langle D_0, D'_1 \rangle)_f$, followed by a $\text{read}()_f$. By linearizability, the read will return either the list written in π_1 or in π_2 on f (depending on how the operations are ordered by the linearizability property). However, as blocks are independent objects, it would be expected that both writes could take effect, with π_1 updating the value of b_0 and π_2 updating the value of b_1. To this respect, we redefine the write to only update *one* of the blocks of a fragmented object. Since the update does not manipulate the value of the whole object, which would include also new blocks to be written, it should allow the update of a block b with a value $|D| > \ell$. This essentially leads to the generation of new blocks in the sequence. More formally, the update operation is defined as follows:

- $\text{update}(b_i, D)_f$ updates the value of block $b_i \in f$ such that:
  - if $|D| \leq \ell$: sets $val(b_i) = D$;
  - if $|D| > \ell$: partition $D = \{D_0, \dots, D_k\}$ such that $|D_j| \leq \ell$, $\forall j \in [0, k]$, set $val(b_i) = D_0$ and create blocks b_i^j, for $j \in [1, k]$ with $val(b_i^j) = D_j$, so that f remains valid.

With the update operation in place, fragmented objects resemble store-collect objects presented in [6]. However, fragmented objects aim at minimizing the communication overhead by exchanging individual blocks (in a consistent manner) instead of exchanging the list (view) of block values in each operation. Since the update operation only affects a block in the list of blocks of a fragmented object, it potentially allows for a higher degree of concurrency. It is still unclear what are the consistency guarantees we can provide when allowing concurrent updates on different blocks to take effect. Thus, we will consider that only operations read and update are issued in fragmented objects. Note that the list of blocks of a fragmented object cannot be reduced. The contents of a block can be deleted by invoking an update with an empty value.

Observe that, as a fragmented object is composed of block objects, its operations are implemented by using read, write, and create block operations. The $\text{read}()_f$ performs a sequence of read block operations (starting from block b_0 and traversing the list of blocks) to obtain and return the value of the fragmented object. Regarding update operations, if $|D| \leq \ell$, then the $\text{update}(b_i, D)_f$ operation performs a write operation on the block b_i as $\text{write}(D)_{b_i}$. However, if $|D| > \ell$, then D is partitioned into substrings $D_0, \dots, D_k$ each of length at most ℓ. The update operation modifies the value of b_i as $\text{write}(D_0)_{b_i}$. Then, k new blocks $b_i^1, \dots, b_i^k$ are created as $\text{create}(b_i^j, D_j)$, $\forall j \in [1, k]$, and are

inserted in f between b_i and b_{i+1} (or appended at the end if $i = |f|$). The sequential specification of a fragmented object extends the definition of sequential specification for read/write (R/W) objects, as defined in Definition 1 of Section 3. It is defined as follows:

Definition 5 (Sequential Specification). The *sequential specification* of a fragmented object $f \in F^{\ell}$ over the complete sequential history H is defined as follows: Initially $val(f) = \langle b_0 \rangle$ with $val(b_0) = \varepsilon$. If at the invocation action of an operation π in H has $val(f) = \langle b_0, \ldots, b_n \rangle$ and $\forall b_i \in f, val(b_i) = D_i$, and $|D_i| \leq \ell$. Then:

- if π is a read()$_f$, then π returns $\langle val(b_0), \ldots, val(b_n) \rangle$. At the response action of π, it still holds that $val(f) = \langle b_0, \ldots, b_n \rangle$ and $\forall b_i \in f, val(b_i) = D_i$.
- if π is an update$(b_i, D)_f$ operation, $b_i \in f$, then at the response action of π, $\forall j \neq i, val(b_j) = D_j$, and
  - if $|D| \leq \ell$: $val(f) = \langle b_0, \ldots, b_n \rangle$, $val(b_i) = D$;
  - if $|D| > \ell$: $val(f) = \langle b_0, \ldots, b_i, b_i^1, \ldots, b_i^k, b_{i+1}, \ldots, b_n \rangle$, such that $val(b_i) = D^0$ and $val(b_i^j) = D^j, \forall j \in [1, k]$, where $D = D^0|D^1| \cdots |D^k$ and $|D^j| \leq \ell, \forall j \in [0, k]$.[2]

4.2 Fragmented Coverable Linearizability

A fragmented object is linearizable if it satisfies both the *Liveness* (termination) and *Linearizability* (atomicity) properties (cf. Definition 2 of Section 3). A fragmented object implemented by a single linearizable block is trivially linearizable as well. Here, we focus on fragmented objects that may contain a list of multiple linearizable blocks, and consider only read and update operations. As defined, update operations are applied on single blocks, which allows multiple update operations to modify different blocks of the fragmented object concurrently. Termination holds since read and update operations on the fragmented object always complete. It remains to examine the consistency properties. When multiple blocks within a fragmented object can be updated concurrently, a critical question arises: What consistency model should the system provide? This section introduces the concept of *Fragmented Coverable Linearizability* as the consistency guarantee for such systems.

Fragmented Linearizability. Let H_{ξ} be a sequential history of update and read invocations and responses on a fragmented object f. This definition is an extension of the linearizability definition for R/W objects, as defined in Definition 2 of Section 3, and it incorporates the sequential specification of a fragmented object, as defined in Definition 5 of Section 4.1.

Definition 6 (Linearizability). A fragmented object f is *linearizable* if, given any complete history H, there exists a permutation σ of all actions in H such that:

- σ is a sequential history and follows the sequential specification of f, and
- for operations π_1, π_2, if $\pi_1 \rightarrow \pi_2$ in H, then π_1 appears before π_2 in σ.

Observe, that in order to satisfy Definition 6, the operations must be totally ordered. Let us consider again the sample execution ξ from Section 4.1. Since we decided not to use write operations, the execution changes as follows: Initially, $val(f) = \langle b_0, b_1 \rangle$, $val(b_0) = D_0$, $val(b_1) = D_1$, and then ξ contains two concurrent update operations by two different clients, one attempting to modify the first block, and the other attempting to modify the second block: $\pi_1 = \text{update}(b_0, D_0')_f$ and $\pi_2 = \text{update}(b_1, D_1')_f$ ($|D_0'| \leq \ell$ and $|D_1'| \leq \ell$), followed by a read()$_f$ operation. In this case, since both update operations operate on different blocks, independently of how π_1 and π_2 are ordered in

[2]The operator "|" denotes concatenation. The exact way D is partitioned is left to the implementation.

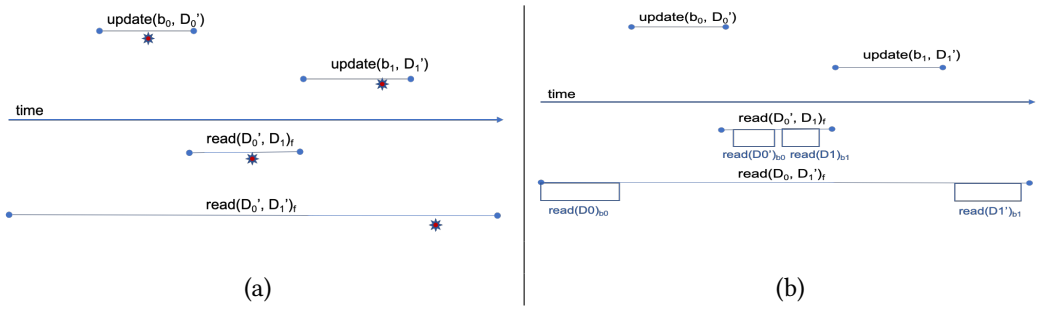


Fig. 1. Executions showing the operations on a fragmented object. Figure (a) shows linearizable reads on the fragmented object (and serialization points), and (b) reads on the fragmented object that are implemented with individual linearizable reads on blocks.

the permutation σ, the read()$_f$ operation will return $\langle D'_0, D'_1 \rangle$. Therefore, the use of these update operations has increased the concurrency in the fragmented object.

Using linearizable read operations on the entire fragmented object can ensure the linearizability of the fragmented object as can be seen in the example presented in Figure 1(a). However, providing a linearizable read when the object involves multiple R/W objects (i.e., an atomic snapshot) can be expensive or impact concurrency [16]. Thus, it is cheaper to take advantage of the atomic nature of the individual blocks and invoke one read operation per block in the fragmented object. ***But, what is the consistency guarantee we can provide on the entire fragmented object in this case?***

As seen in the example of Figure 1(b), two reads concurrent with two update operations may violate linearizability on the entire object. According to the real time ordering of the operations on the individual blocks, block linearizability is preserved if the first read on the fragmented object should return (D'_0, D_1), while the second read returns (D_0, D'_1). Note that we cannot find a permutation on these concurrent operations that follows the sequential specification of the fragmented object. Thus, the execution in Figure 1(b) violates linearizability. This leads to the definition of *fragmented linearizability* on the fragmented object, which relying on the fact that *each individual block is linearizable*, it allows executions like the one seen in Figure 1(b). Essentially, fragmented linearizability captures the consistency one can obtain on a collection of linearizable objects, when these are accessed concurrently and individually, but under the "umbrella" of the collection.

In this respect, we specify each read()$_f$ operation of a certain process, as a sequence of read()$_b$ operations on each block $b \in f$ by that process. In particular, a read operation read()$_f$ that returns $\langle val(b_0), \ldots, val(b_n) \rangle$ is specified by $n+1$ individual read operations read()$_{b_0}$,..., read()$_{b_n}$, that return $val(b_0)$, ..., $val(b_n)$, respectively, where read()$_{b_0} \rightarrow, \ldots, \rightarrow$ read()$_{b_n}$.

Then, given a history H, we denote for an operation π the history H^{π} which contains the actions extracted from H and performed during π (including its invocation and response actions). Hence, if $val(f)$ is the value returned by read()$_f$, then $H^{\text{read()}_f}$ contains an invocation and matching response for a read()$_b$ operation, for each $b \in val(f)$. Then, from H, we can construct a history $H|_f$ that only contains operations on the whole fragmented object. In particular, $H|_f$ is the same as H with the following changes: for each read()$_f$, if $\langle val(b_0), \ldots, val(b_n) \rangle$ is the value returned by the read operation, then we replace the invocation of read()$_{b_0}$ operation with the invocation of the read()$_f$ operation and the response of the read()$_{b_n}$ block with the response action for the read()$_f$ operation. Then, we remove from $H|_f$ all the actions in $H^{\text{read()}_f}$.

Definition 7 (Fragmented Linearizability). Let $f \in F^{\ell}$ be a fragmented object, H a complete history on f, and $val(f)_H \subseteq \mathcal{B}$ the value of f at the end of H. Then, f is *fragmented linearizable* if there exists a permutation σ_b over all the actions on b in H, $\forall b \in val(f)_H$, such that:

— σ_b is a sequential history that follows the sequential specification of b,[3] and
— for operations π_1, π_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 operations on b in $H^{\pi 2}$ in σ_b.

Fragmented linearizability guarantees that all concurrent operations on different blocks prevail, and only concurrent operations on the same blocks are conflicting. Consider two reads r_1 and r_2, s.t. $r_1 \rightarrow r_2$; then r_2 must return a supersequence of blocks with respect to the sequence returned by r_1, and that for each block belonging in both sequences, its value returned by r_2 is the same or newer than the one returned by r_1.

Fragmented Coverable Linearizability. When writing a value to a linearizable R/W object, the value written does not need to be dependent on the previous written value. However, in some objects (e.g., files), it is expected that a value update will build upon (and thus avoid to overwrite) the current value of the object. In such cases, a writer should be aware of the latest value of the object (i.e., by reading the object) before updating it. Although a **read-modify-write** (**RMW**) semantic would be more appropriate for this type of objects, it can only be achieved through consensus, which is known to be merely impossible to solve in an asynchronous environment with crashes [25].

As already discussed, in [50] the notion of coverability (cf. Definition 4 of Section 3) was introduced to leverage the solvability of R/W object implementations, while providing a *weak* RMW object. Recall that coverability 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.

If a fragmented object utilizes coverable blocks, instead of linearizable blocks, then Definition 7 provides what we would call ***fragmented coverable linearizability*** or for short, ***fragmented coverability***: Concurrent update operations on different blocks would *all* 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). As we see in the reminder of this work, fragmented coverability is a good alternative to RMW semantics to implement large objects, like files, of which any new value may depend on the current value of the object.

4.3 CoBFS: Framework for Fragmented Coverable Objects

After laying out the theoretical foundation of Fragmented Objects, we now introduce a framework for a Distributed Storage System, which we refer to as CoBFS.

Coverability, as described in the Section 4.2, is crucial for updating the content of a fragmented object. When updating a fragmented object, one expects the update to be on the previous version of the object. Therefore, a write operation cannot change the value of the object arbitrarily, but must always update it independently of the previously written value. By utilizing coverable blocks, our storage system provides fragmented coverability as a consistency guarantee. The underlying theoretical formulation allows to support any kind of large objects (e.g., a plain text or binary file).

Overview of the Basic Architecture: The basic architecture of CoBFS appears in Figure 2. CoBFS is composed of two main modules: (*i*) a FM, and (*ii*) a DSMM. In summary, the FM implements the

[3]The sequential specification of a block is similar to that of a R/W register [48], whose value has bounded length.

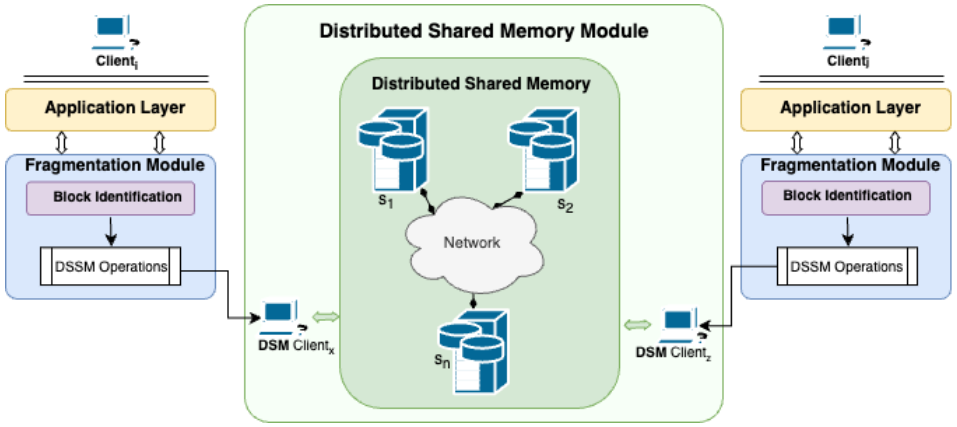


Fig. 2. Basic architecture of CoBFS.

fragmented object while the DSMM implements an interface to a shared memory service that allows read/write operations on individual block objects. Following this architecture, clients may access the storage system through the FM, while the blocks of each fragmented object are maintained by servers through the DSMM.

Fragmentation Module: Within FM, an essential function is Block Identification. This process entails partitioning the object into smaller blocks and ensuring that each of these blocks adheres to a predefined length (as specified in Section 4.1). The FM uses the DSMM as an external service to write and read blocks to the shared memory. To this respect, CoBFS is flexible enough to utilize various fragmentation strategies and adapt to any underlying distributed shared object algorithm. In Section 5, we present a specific implementation of the Fragmentation Module designed for seamless integration within the COBFS framework.

Distributed Shared Memory Module: The DSMM exposes three operations for a block b: dsmm-read$_b$, dsmm-write$(v)_b$, and dsmm-create$(v)_b$. The specification of each operation is shown in Algorithm 1. For each block b, the DSMM maintains its latest known version ver_b and its associated value val_b. Upon receipt of a read request for a block b, the DSMM invokes a cvr-read operation on b and returns the value received from that operation. The dsmm-create and dsmm-write operations invoke cvr-write operations to update the value of the shared block b. Their main difference is that version $\langle 0, \perp \rangle$ is used during a dsmm-create operation to indicate that this is the first time that the block is written. Notice that the cvr-write in dsmm-create will always be considered *successful* as it will introduce a new, never before written block. However, the cvr-write in dsmm-write may be converted to a read operation, thus retrieving and returning the latest value of b. In such cases, the update is considered *unsuccessful*.

5 Fragmentation Module: Design and Implementation Details

This section presents a specific and detailed implementation of the FM designed for seamless integration within the CoBFS framework.

Implement a Coverable Fragmented Object: A fragmented object is modeled with its blocks being coverable objects. The fragmented object is implemented as a ***linked-list of blocks*** with the first block being a special block $b_g \in \mathcal{B}$, which we call the ***genesis block***, and then each block having a pointer *ptr* to its next block, whereas the last block has a null pointer. Initially each fragmented

ALGORITHM 1: DSMM: Operations on a coverable block object b at client p

```
1: State Variables:
2: ver_b ∈ ℕ⁺ × 𝒲 initially ⟨0, ⊥⟩; val_b ∈ 𝒱 initially ⊥;

3: function dsmm-read( )_{b,p}
4:     ⟨ver_b, val_b⟩ ← b.cvr-read()
5:     return val_b
6: end function

7: function dsmm-create(val)_{b,p}
8:     ⟨ver_b, val_b⟩ ← b.cvr-write(val, ⟨0, ⊥⟩)
9: end function

10: function dsmm-write(val)_{b,p}
11:     ⟨ver_b, val_b⟩ ← b.cvr-write(val, ver_b)
12:     return val_b
13: end function
```

object contains only the genesis block; the genesis block contains special purpose (meta) data. The $val(b)$ of b is set as a tuple, $val(b) = \langle ptr, data \rangle$.

Object and Block Id Assignment: A key aspect of our implementation is the unique assignment of ids to both fragmented objects and individual blocks. A fragmented object $f \in F$ is assigned a pair $\langle cfid, cfseq \rangle \in \mathcal{C} \times \mathbb{N}$, where $cfid \in \mathcal{C}$ is the universally unique identifier of the client that created the object (i.e., the owner) and $cfseq \in \mathbb{N}$ is the client's local sequence number, incremented every time the client creates a new object and ensuring the uniqueness of the objects created by the same client.

In turn, a block $b \in \mathcal{B}$ of a fragmented object is identified by a triplet $\langle fid, cid, cseq \rangle \in F \times \mathcal{C} \times \mathbb{N}$, where $fid \in F$ is the identifier of the object to which the block belongs, $cid \in \mathcal{C}$ is the identifier of the client that created the block (this is not necessarily the owner/creator of the fragmented object), and $cseq \in \mathbb{N}$ is the client's local sequence number of blocks that is incremented every time this client creates a block for this object (this ensures the uniqueness of the blocks created by the same client for the same fragmented object).

Fragmentation Module: The FM is the core concept of CoBFS implementation. Each client has an FM responsible for (*i*) fragmenting the fragmented object into blocks and identifying modified blocks, and (*ii*) following a specific strategy to store and retrieve the object's blocks from the R/W shared memory. As we show later, the block update strategy followed by FM is necessary in order to preserve the structure of the fragmented object and sufficient to preserve the properties of fragmented coverability. The FM's external signature includes the two main operations of a fragmented object: read$_f$, and update$_f$. Their specifications appear in Algorithm 2.

Update Operation - fm-update$(b, D)_{f,p}$**.** The update strategy of the FM is the most challenging part of our work. For the fragmented object division of the blocks and the identification of the newly created blocks, the FM contains a *Block Identification (BI) module* that utilizes known approaches for data fragmentation and diff extraction. Given the data D of a fragmented object f, the goal of BI is to break D into data blocks $\langle D_0, \ldots, D_n \rangle$, s.t. the size of each D_i is less than a predefined upper bound ℓ. By drawing ideas from the RSYNC (Remote Sync) algorithm [58], given two versions of the same fragmented object, say f and f', the BI tries to identify blocks that (*a*) may exist in f but not in f' (and vice-versa), or (*b*) they have been changed from f to f'. To achieve these goals BI proceeds in two steps: (1) it fragments D into blocks, using the *Rabin fingerprints* rolling hash algorithm [54], and (2) it compares the hashes of the blocks of the current and the previous version of the fragmented object using a string matching algorithm [11] to determine the modified/new data blocks. The role of BI within the architecture of CoBFS and its process flow appears in Figure 3, while its specification is provided in Algorithm 2. A high-level description of *BI* (fm-block-identify()$_{f,p}$) has as follows:

— ***Block Division:*** Initially, the BI partitions a given fragmented object f into data blocks based on its contents, using *Rabin fingerprints*. This scheme allows to divide f into blocks of at most

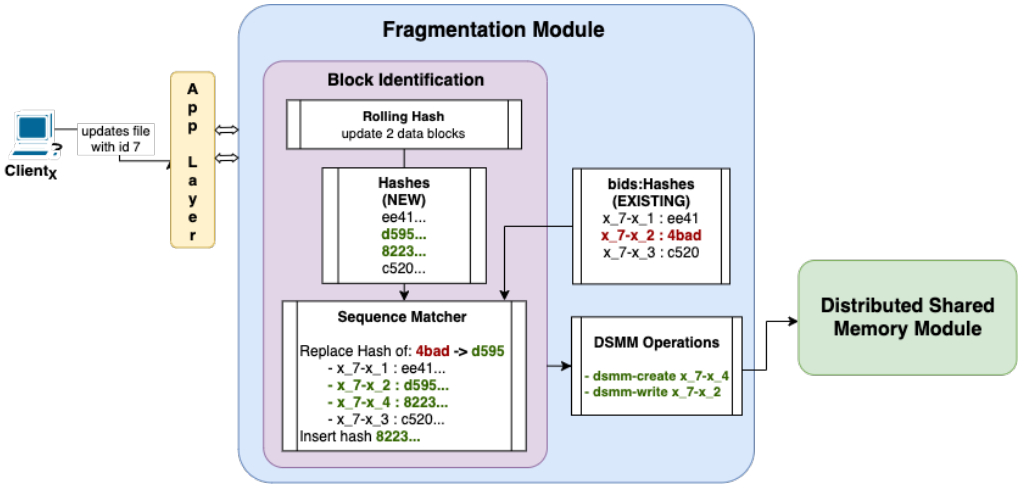


Fig. 3. Example of a writer x writing text at the beginning of the second block of a fragmented object with id $f_{id} = 7$. The hash value of the existing second block "4*bad*.." is replaced with "*d*595.." and a new block with hash value "8223.." is inserted immediately after. The block b_{id} = x_7-x_2 and the new block b_{id} = x_7-x_4 are sent to the DSM.

ALGORITHM 2: Fragmentation Module: BI and Operations on a fragmented object f at client p

State Variables:
H initially $\emptyset$; $\ell \in \mathbb{N}$;
$\mathcal{L}_f$ a linked-list of blocks, initially $\langle b_g \rangle$;
$bc_f \in \mathbb{N}$ initially 0;

function fm-block-identify( )$_{f,p}$
  $\langle newD, newH \rangle \leftarrow$ RabinFingerprints(f, ℓ)
  $curH = hash(\mathcal{L}_f)$
    ▷ hashes of the data of the blocks in $\mathcal{L}_f$
  $C \leftarrow$ SMatching$(curH, newH)$
    ▷ blocks modified
  for $\langle h(b_j), h_k \rangle \in C.mods$ s.t.
$h(b_j) \in curH, h_k \in newH$ **do**
    $D \leftarrow \{D_k : D_k \in newD \ \wedge \ h_k = hash(D_k)\}$
    fm-update$(b_j, D)_{f,p}$
    ▷ blocks inserted
  for $\langle h(b_j), S \rangle \in C.inserts$ s.t.
$h(b_j) \in curH, S \subseteq newH$ **do**
    $D \leftarrow \{D_i : h_i \in S \wedge D_i \in newD \ \wedge \ h_i = hash(D_i)\}$
    fm-update$(b_j, D)_{f,p}$
end function

function Assemble$(\mathcal{L}_f)$
  $f \leftarrow \perp$
  for $i = 0 : |\mathcal{L}_f|$ **do**
    $f \leftarrow f \oplus L_f[i]$
  return f
end function

function fm-read( )$_{f,p}$
  $b \leftarrow val(b_g).ptr$
  $\mathcal{L}_f \leftarrow \langle b_g \rangle$ ▷ reset $\mathcal{L}_f$
  while b *not* NULL **do**
    $val(b) \leftarrow$ dsmm-read()$_{b,p}$
    $\mathcal{L}_f.insert(val(b))$
    $b \leftarrow val(b).ptr$
  end while
  return Assemble$(\mathcal{L}_f)$
end function

function fm-update$(b, D = \langle D_0, D_1, \ldots, D_k \rangle)_{f,p}$
  for $j = k : 1$ **do**
    $b_j \leftarrow \langle f, p, bc_f{+}{+} \rangle$ ▷ set block id
    $val(b_j).data = D_j$ ▷ set block data
    if $j < k$ **then**
      $val(b_j).ptr = b_{j+1}$ ▷ set block ptr
    else
      $val(b_j).ptr = val(b).ptr$
      ▷ point last to b ptr
    $\mathcal{L}_f.insert(val(b_j))$
    dsmm-create$(val(b_j))_{b_j}$
  $val(b).data = D_0$
  if $k > 0$ **then**
    $val(b).ptr = b_1$ ▷ change b ptr if $|D| > 1$
  dsmm-write$(val(b))_b$
end function

size ℓ, which are identified by their hashes (fingerprints). When used in two versions of the same fragmented object, the scheme guarantees that only the hash of changed blocks (and at most their respective next blocks) will be affected. To this end, any data that may cause a changed block to overflow will yield new blocks.

— ***Block Matching:*** Given the set of blocks $\langle D_0, \dots, D_m \rangle$ and associated block hashes $\langle h_0, \dots, h_m \rangle$ generated by the Rabin fingerprint algorithm, the BI tries to match each hash to a block identifier, based on block ids produced during the previous division of fragmented object f, say $\langle b_0, \dots, b_n \rangle$. We produce the vector $\langle h(b_0), \dots, h(b_n) \rangle$ where $h(b_i) = hash(val(b_i).data)$ from the current blocks of f, and using a string matching algorithm [11], we compare the two hash vectors to obtain one of the following statuses for each entry: (*i*) equal, (*ii*) modified, (*iii*) inserted, (*iv*) deleted.

— ***Block Updates:*** Based on the hash statuses computed through block matching previously, the blocks of the fragmented object are updated. In particular, in the case of equality, if a $h_i = h(b_j)$ then D_i is identified as the data of block b_j. In case of modification, e.g., $(h(b_j), h_i)$, an fm-update$(b_j, \{D_i\})_{f,p}$ action is issued to modify the data of b_j to D_i (Lines 10:14). In case new hashes (e.g., $\langle h_i, h_k \rangle$) are inserted after the hash of block b_j (i.e., $h(b_j)$), the action fm-update$(b_j, \{val(b_j).data, D_i, D_k\})_{f,p}$ is performed to create the new blocks after b_j (Lines 15:19). In our formulation block deletion is treated as a modification that sets an empty data value thus, in our implementation *no blocks are deleted*. Note that an fm-update operation is considered *successful* only when the underlying dsmm-write that modifies the data of block b (line Algorithm 2:51) succeeds. Otherwise, the entire update is considered *unsuccessful*.

Read Operation - fm-read()$_{f,p}$**.** To retrieve the value of a fragmented object f, a client p may invoke a fm-read$_{f,p}$ to the fragmented object. The FM issues a series of reads on fragmented object's blocks; starting from the genesis block of f and proceeding to the last block by following the pointers in the linked-list of blocks $\mathcal{L}_f$ comprising the fragmented object. The ordered list of blocks $\mathcal{L}_f$ is passed to the Assemble() function, which concatenates the blocks together into one fragmented object, maintaining the order specified in the list $\mathcal{L}_f$.

6 Fragmented ABD: Static Storage for Large Objects

In this section, we integrate the static ADSM algorithm, ABD [4, 5], with the DSMM module in CoBFS. To accomplish this integration, we first implement the coverable variant of ABD, referred to as CoABD [50]. Subsequently, we combine the CoABD algorithm with the DSMM module in CoBFS, thereby creating the combined system named CoABDF.

6.1 CoABD Optimization

Before proceeding to the integration of the CoABD to CoBFS, we introduce an optimization of the CoABD (see Algorithm 3), aiming at reducing the amount of data transmitted during read and write operations, particularly benefiting scenarios involving fragmented objects.

Description of Optimized CoABD***:*** The blue text in Algorithm 3 annotates the changes when compared to the original CoABD read/write protocols as presented in [50]. To reduce the data transmitted per read, we apply a simple yet very effective optimization: servers avoid to reply with the object value when the client already knows a more recent version of the object, and readers avoid value propagation when enough servers are aware of the latest version of the object. In particular, when a server receives a READ request it replies with both its local tag and object content only if the tag enclosed in the READ request is smaller than its local tag; otherwise it replies with its local tag without the object content. Once the reader receives replies from a majority of servers,

ALGORITHM 3: Optimized coverable ABD

```
1: at each reader r for object b
2: State Variables:
3: tg_b ∈ ℕ⁺ × 𝒲 initially ⟨0, ⊥⟩; val_b ∈ V, init. ⊥;

4: function cvr-read( )
5:    send ⟨READ, tg_b⟩ to all servers                    ▷ Query Phase
6:    wait until (|S|+1)/2 reply
7:    maxP ← max({⟨tg′, v′⟩})
8:    if maxP.tg > tg_b then
9:       send (WRITE, maxP) to all servers                ▷ Propagation Phase
10:      wait until (|S|+1)/2 servers reply
11:      ⟨tg_b, val_b⟩ ← maxP
12:   return(⟨tg_b, val_b⟩)
13: end function

14: at each server s for object b
15: State Variables:
16: tg_b ∈ ℕ⁺ × 𝒲 initially ⟨0, ⊥⟩; val_b ∈ V, init. ⊥;

17: function rcv(M)_q                                    ▷ Reception of a message from q
18:   if M.type ≠ READ and M.tg > tg_b then
19:      ⟨tg_b, val_b⟩ ← ⟨M.tg, M.v⟩
20:   if M.type = READ and M.tg ≥ tg_b then
21:      send(⟨tg_b, ⊥⟩) to q                             ▷ Reply without content
22:   else
23:      send(⟨tg_b, val_b⟩) to q                         ▷ Reply with content
24: end function
```

it detects the maximum tag among the replies, and checks if it is higher than the local known tag. If it is, then it forwards the tag and its associated object content to a majority of servers; if not, then the read operation returns the locally known tag and object content without performing the propagation phase. While this optimisation makes little difference on the non-fragmented version of the ABD (under read/write contention), it makes a significant difference in the case of the fragmented objects. For example, if each read is concurrent with a write causing the execution of a propagation phase, then the read sends the complete object to the servers; in the case of fragmented objects only the fragments that changed by the write will be sent over to the servers, resulting in significant reductions. During the query phase of the write operation, we apply the same optimization as in the read operation. The only difference between the write and read operations is that the former performs the propagation phase in any execution, whereas the read operation performs a propagation phase only if the reader does not have the latest version, as explained above. The cvr-write operation is omitted due to slight changes in its code; specifically, in the query phase, it now includes its local tag in the READ request message.

Correctness of Optimized CoABD. We now argue that this simple optimization does not affect the correctness of the algorithm.

LEMMA 8. *In any execution ξ of* CoABD*, if τ_{max} is the maximum tag discovered at the query phase of a read/write operation π from process p, and τ_π the local tag in p after the completion of π, then $\tau_\pi \geq \tau_{max}$. The proof of this lemma is provided in Appendix A.1.*

The next lemma states that if two operations are separated in time then the latter operation will observe/return a "fresher" tag than the first operation.

LEMMA 9. *Let π_1 and π_2 denote completed read/write operations in an execution ξ, from processes $p_1, p_2 \in \mathcal{J}$ respectively, such that $\pi_1 \rightarrow \pi_2$. If τ_{π_1} and τ_{π_2} are the local tags at p_1 and p_2 after the completion of π_1 and π_2 respectively, then $\tau_{\pi_1} \leq \tau_{\pi_2}$; if π_2 is a successful write operation then $\tau_{\pi_1} < \tau_{\pi_2}$. The proof of this lemma is provided in Appendix A.2.*

THEOREM 10. *The optimized* CoABD *implements R/W coverable objects.*

PROOF. Lemma 9 shows that CoABD satisfies linearizability. So, let's focus on coverability. When both the read and write operations perform two phases, the correctness of the algorithm is derived from Theorem 10 in [50]. The propagation phase of a read is omitted when all the servers reply with a tag smaller or equal to the local tag of the reader r (cf. line Algorithm 3:8). Since however, a read propagates its local tag to a majority of servers at every tag update, then every subsequent

operation will observe (and return) the latest value of the object to be associated with a tag at least as high as the local tag of r. □

6.2 CoABDF: **Integrate** CoABD **with** CoBFS

In this section, we integrate CoABD with CoBFS to obtain what we refer to as CoABDF. This results in a static distributed storage designed for handling large objects. Also, we have to prove that the CoABDF system satisfies certain properties, including coverability for block operations, adhesion to a limiting block size by **Block Identification** (**BI**) (cf. Section 5), and the creation of a valid fragmented object according to the criteria in Definition 7.

Integration of CoABD *in* CoBFS. In the CoBFS framework, we utilize the FM detailed in Section 5 and, in the DSMM component of the framework, we leverage the optimized CoABD version outlined in Section 6.1.

Read Operation: When the system receives a read request from a client, the *FM* issues a series of block read operations, detailed in Algorithm 2. The read operation executes the block read operations on the shared memory using the dsmm-read function in Algorithm 1. Finally, the shared memory executes the cvr-read as defined in Algorithm 3.

Update Operation: Upon receiving an update operation, the *FM* uses the BI module for the division of the objects into data blocks utilizing update in Algorithm 2. Thus, $D = \langle D_0, \ldots, D_k \rangle$, for $k \geq 0$, with the size $|D| = \sum_{i=0}^{k} |D_i|$ and the size of each $|D_i| \leq \ell$ for some maximum block size ℓ. Then, the *FM* attempts to update the value of each block with identifier b in fragmented object f with the data in D. Depending on the size of D the update operation using Algorithm 1 will either perform a write on the block (i.e., dsmm-write) if $k = 0$, or it will create new blocks (i.e., dsmm-create) and update the block pointers in case $k > 0$. Assuming that $val(b).ptr = b'$ then:

- $k = 0$: In this case update, for block b, calls write$(\langle val(b).ptr, D_0 \rangle, \langle p, bseq \rangle)_b$.
- $k > 0$: Given the sequence of chunks $D = \langle D_0, \ldots, D_k \rangle$ the following block operations are performed in this particular order:
  - → create$(b_k = \langle f, p, bc_p{+}{+} \rangle, \langle b', D_k \rangle, \langle p, 0 \rangle)$ ** Block b_k ptr points to b' **
  - → ...
  - → create$(b_1 = \langle f, p, bc_p{+}{+} \rangle, \langle b_2, D_1 \rangle, \langle p, 0 \rangle)$ ** Block b_1 ptr points to b_2 **
  - → write$(\langle b_1, D_0 \rangle, \langle p, bseq \rangle)_b$ ** Block b ptr points to b_1 **

Finally, the block operations are executed on the DSMM using cvr-write. (Given the slight deviation from the original operation, as depicted in Figure 2 in [50], the description of this operation is omitted. Please refer to Section 6.1 for further details.)

Correctness of CoABDF. The provided proof demonstrates the correctness of the CoABDF, addressing challenges related to read and update operations. The main challenge involves coordinating the update operation to maintain the integrity of the fragmented object while avoiding interference with concurrent operations.

We commence with a proof addressing the correctness of the CoABD implementation in the DSMM.

Lemma 11. *DSMM implements R/W coverable block objects. The proof of this lemma is provided in Appendix A.3.*

Also, the challenge in update operation was to insert the list of blocks without causing any concurrent operation to return a divided fragmented object, while also avoiding blocking any ongoing operations. To achieve that, create operations are executed in a reverse order: we first create block b_k pointing to b', and we move backward until creating b_1 pointing to block b_2. The last

operation, write, tries to update the value of block b_0 with value $\langle b_1, D_0 \rangle$. If the last coverable write completes successfully, then all the blocks are inserted in f and the update is *successful*; otherwise, none of the blocks appears in f and thus the update is *unsuccessful*. This is captured by the following lemma:

Lemma 12. *In any execution ξ of* CoABDF*, if ξ contains an $\pi = \text{update}(b, D)_{f,p}$, then π is successful iff the operation b.*cvr-write *called within* dsmm-write$(val(b))_{b,p}$*, is successful. The proof of this lemma is provided in Appendix A.4.*

Now a read operation may return a list that contains a block b_i only if b_i was written by a successful update operation. More formally:

Lemma 13. *In any execution ξ of* CoABDF*, if a $\rho = \text{read}_{f,p}$ operation returns a list $\mathcal{L}$ then for any block $b \in \mathcal{L}$ there exists successful $\text{update}(*)_{f,*}$ operation that either precedes or is concurrent to ρ and invokes* sm-create$(val(b))_b$ *operation. The proof of this lemma is provided in Appendix A.5.*

The above lemma will help us to show that the linked-list used for implementing our fragmented object stays connected in any execution.

Lemma 14. *In any execution ξ of* CoABDF*, if a $\text{read}_{f,p}$ operation returns a list $\mathcal{L} = \langle b_g, b_1, \ldots, b_n \rangle$ for a fragmented object f, then $val(b_g).ptr = b_1$, $val(b_i).ptr = b_{i+1}$, for $1 \leq i < n-1$, and $val(b_n).ptr = \perp$. The proof of this lemma is provided in Appendix A.6.*

We now show that a subsequent read operation always contains the blocks from a preceding read, with versions that are equally or more recent.

Lemma 15. *Let ξ be an execution of* CoABDF *with two reads $\rho_1 = \text{read}_{f,p}$ and $\rho_2 = \text{read}_{f,q}$ from clients p and q on the fragmented object f, s.t. $\rho_1 \rightarrow \rho_2$. If ρ_1 returns a list of blocks $\mathcal{L}_1$ and ρ_2 a list $\mathcal{L}_2$, then $\forall b_i \in \mathcal{L}_1$, then $b_i \in \mathcal{L}_2$ and the version of each $b_i \in \mathcal{L}_1$ is smaller than or equal to the version of $b_i \in \mathcal{L}_2$. The proof of this lemma is provided in Appendix A.7.*

This leads us to the following:

Theorem 16. CoABDF *implements a R/W Fragmented Coverable object.*

Proof. By Lemma 11, every block operation in CoABDF satisfies coverability and together with Lemma 15 it follows that CoABDF implements a coverable fragmented object satisfying the properties presented in Definition 7. Also, the BI ensures that the size of each block is limited under a bound ℓ and Lemma 14 ensures that each operation obtains a connected list of blocks. Thus, CoABDF implements a *valid* fragmented object. □

7 Fragmented Ares: Dynamic Storage for Large Objects

In this section, we bring the coverability and fragmentation strategy of CoBFS (cf. Sections 4.3 and 5) to a dynamic environment. To achieve this, we integrate the dynamic ADSM algorithm Ares (cf. Section 7.1) with the DSMM module in CoBFS. We first provide a coverable version of Ares, i.e., CoAres (cf. Section 7.2), and then integrate CoAres with CoBFS to obtain CoAresF (cf. Section 7.4). We also present an optimization of the DAP-EC (the Data Access Primitive utilizing Erasure Coding), which has its own indepentent interest (cf. Section 7.3).

7.1 Overview of Ares

We begin by providing an overview of Ares, including its design principles and key features. For more information, please refer to [51]; there one can also find information about the implementation of Ares, as well as relevant experiments carried out by some of the authors of the current work.

Ares is a modular algorithm, designed to implement dynamic, reconfigurable, fault-tolerant, read/write distributed linearizable (atomic) shared memory objects. We begin Ares's overview by explaining how a configuration is defined in Ares.

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 $c.Servers$ (see more info below); (*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) [36]. We refer to a server $s \in c.Servers$ as a *member* of configuration c.

Data Access Primitives (DAPs). 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($\langle tag, value \rangle$), which accepts a $\langle tag, value \rangle$ as an argument.

As seen in [51], 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., [4]). 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). Different DAP_s can be used in different configurations, as long as the following consistency properties hold:

Property 1 (DAP Consistency Properties). A *DAP* operation in an execution ξ is complete if both the invocation and the matching response steps appear in ξ. If Π is the set of complete DAP operations in execution ξ then for any $\phi, \pi \in \Pi$:

- C1 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 ϕ completes before π is invoked in ξ, then $\tau_\pi \geq \tau_\phi$.
- C2 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 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).

DAP Implementations. To demonstrate the flexibility that DAPs provide, the authors in [51], expressed two different atomic shared R/W algorithms in terms of DAPs. These are the DAPs for the well celebrated ABD algorithm (see Section 8.1 in article [51]), and the DAPs for an erasure coded based approach presented for the first time in Section 5 of [51]. 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 [56]) 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.

Ares *Operations.* 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 [51], 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 $\langle \tau', v'\rangle = \langle\langle \tau.ts + 1, p\rangle, v'\rangle$ 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.

7.2 CoAres: **Coverable** Ares

We now present and analyze the coverable extension of Ares, which we refer to as CoAres.

Description. Below, we describe the modification that needs 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 4 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 Algorithm 4:14 (respectively line Algorithm 4:47).

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 Algorithm 4:15–18). 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 Algorithm 4:19 - 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 Algorithm 4:20–21). Otherwise, the state of the object does not change and the writer keeps the maximum $\langle \tau, v\rangle$ pair found in the

ALGORITHM 4: Write and Read protocols for CoAres.

```
1:  CVR-Write Operation:
2:    at each writer w_i
3:    State Variables:
4:    cseq[]s.t.cseq[j] ∈ C × {F, P}
5:    version ∈ ℕ⁺ × W ∪ {⊥} initially ⟨0, ⊥⟩
6:    Local Variables:
7:    μ ∈ ℕ⁺ initially 0, ν ∈ ℕ⁺ initially 0
8:    τ ∈ ℕ⁺ × W initially ⟨0, w_i⟩
9:    v ∈ V initially ⊥
10:   flag ∈ {chg, unchg} initially unchg
11:   Initialization:
12:   cseq[0] = ⟨c_0, F⟩

13:   operation cvr-write(val), val ∈ V
14:     cseq ←read-config(cseq)
15:     μ ← max({i : cseq[i].status = F})
16:     ν ← |cseq|
17:     for i = μ : ν do
18:       ⟨τ, v⟩ ← max(cseq[i].cfg.get-data(), ⟨τ, v⟩)
19:     if version = τ then
20:       flag ← chg
21:       ⟨τ, v⟩ ← ⟨⟨τ.ts + 1, ω_i⟩, val⟩
22:     else
23:       flag ← unchg
24:     version ← τ
25:     done ← false
26:     while not done do
27:       cseq[v].cfg.put-data(⟨τ, v⟩)
28:       cseq ←read-config(cseq)
29:       if |cseq| = v then
30:         done ← true
31:       else
32:         v ← |cseq|
33:     end while
34:     return ⟨τ, v⟩, flag
35:   end operation

36:   CVR-Read Operation:
37:     at each reader r_i
38:     State Variables:
39:     cseq[]s.t.cseq[j] ∈ C × {F, P}
40:     Local Variables:
41:     μ ∈ ℕ⁺ initially 0, ν ∈ ℕ⁺ initially 0
42:     τ ∈ ℕ⁺ × W initially ⟨0, w_i⟩
43:     v ∈ V initially ⊥
44:     Initialization:
45:     cseq[0] = ⟨c_0, F⟩

46:   operation cvr-read( )
47:     cseq ←read-config(cseq)
48:     μ ← max({j : cseq[j].status = F})
49:     ν ← |cseq|
50:     for i = μ : ν do
51:       ⟨τ, v⟩ ← max(cseq[i].cfg.get-data(), ⟨τ, v⟩)
52:     done ← false
53:     while not done do
54:       cseq[v].cfg.put-data(⟨τ, v⟩)
55:       cseq ←read-config(cseq)
56:       if |cseq| = v then
57:         done ← true
58:       else
59:         v ← |cseq|
60:     end while
61:     return ⟨τ, v⟩
62:   end operation
```

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 Algorithm 4:24).

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 Algorithm 4:50–51). The propagation of $\langle \tau, v \rangle$ in both cvr-write (lines Algorithm 4:26–33) and cvr-read (lines Algorithm 4:53–60)) 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 [51], 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 [51].

The validity and coverability properties, defined formally as 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 $c \in \mathcal{C}$ satisfies Property 1, we will show that any execution ξ of CoAres satisfies the properties of Definitions 3 and 4.

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: We proceed with formal statements and proofs. Lemmas 17 to 19 help us show that CoAres satisfies *Validity*.

Lemma 17 (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 $ver' > ver$. The proof of this lemma is provided in Appendix A.8.*

Lemma 18 (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 $ver_1 \neq ver_2$. The proof of this lemma is provided in Appendix A.9.*

Lemma 19. *Each version we reach in an execution is derived (through a chain of operations) from the initial version of the register ver_0. The proof of this lemma is provided in Appendix A.10.*

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 4).

Lemma 20. *In any execution ξ of* CoAres*, all properties of Definition 4 are satisfied. The proof of this lemma is provided in Appendix A.11.*

Lemmas 17 to 20 show that CoAres satisfies validity (Definition 3) and coverability (Definition 4):

Theorem 21. CoAres *implements a linearizable coverable object, given that the DAPs implemented in any configuration c satisfy Property 1.*

7.3 EC-DAP Optimization

Here 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 stems from the optimization of Algorithm 3 in Section 6.1, which 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 Algorithms 5 and 6. 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 [51], each server s_i maintains a local state variable, $List_i$, which stores a set of up to $(\delta + 1)$ (tag, coded-element) pairs. The variable δ is the maximum number of concurrent put-data

ALGORITHM 5: EC-DAPopt implementation

```
   at each process p_i ∈ 𝒥
2: procedure c.get-data()
     send (QUERY-LIST, c.tag) to each s ∈ c.Servers
4:   until p_i receives List_s from each server s ∈ 𝒮_g
         ↪ s.t. |𝒮_g| = ⌈(n+k)/2⌉
               and 𝒮_g ⊂ c.Servers
6:   ~~Tags_s^{≥k} = set of tags that appears in k lists~~
     Tags_dec^{≥k} = set of tags that appears in k lists
8:             with values
     ~~t*_max ← max(Tags_s^{≥k})~~
10:  t^dec_max ← max(Tags_dec^{≥k})
     ~~if t^dec_max = t*_max then~~
12:  if c.tag = t^dec_max then
       t ← c.tag
14:    v ← c.val
       return ⟨t, v⟩
16:  else if Tags_dec^{≥k} ≠ ⊥ then
       t ← t^dec_max
18:    v ← decode value for t^dec_max
       return ⟨t, v⟩
20: end procedure

   procedure c.put-data(⟨τ, v⟩))
22:  if τ > c.tag then
       code-elems = [(τ, e_1), ..., (τ, e_n)], e_i
24:              = Φ_i(v)
       send (PUT-DATA, ⟨τ, e_i⟩) to each s_i
           ↪ ∈ c.Servers
26:    until p_i receives ACK from ⌈(n+k)/2⌉ servers in
           ↪ c.Servers
       c.tag ← τ
28:    c.val ← v
   end procedure
```

operations that overlap with the get-data, until the time the client has all data needed to attempt decoding a value. 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*, if the tag is higher than the *c.tag* of the client and the (tag, $\perp$) 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 Algorithm 5:8–10) 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^{dec}_{max}), the client already has the latest decodable version and does not need to decode it again (see line Algorithm 5:12).

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 Algorithm 5:22). 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 Algorithm 6:16).

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 [51], as the optimization has no effect on this operation, so it should preserve safety as shown in [51].

ALGORITHM 6: The response protocols at any server $s_i \in \mathcal{S}$ in EC-DAPopt for client requests.

1: at each server $s_i \in \mathcal{S}$ in configuration c_k

2: **State Variables:**
$List \subseteq \mathcal{T} \times \mathcal{C}_s$, initially $\{(t_0, \Phi_i(v_0))\}$
3: **Local Variables:**
$List' \subseteq \mathcal{T} \times \mathcal{C}_s$, initially $\perp$

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

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

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 22 (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). The proof of this lemma is provided in Appendix A.12.*

Lemma 23 (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 $\phi \rightarrow \pi$ in ξ, then $\tau_\pi \geq \tau_\phi$. The proof of this lemma is provided in Appendix A.13.*

Theorem 24 (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 5. Then every pair of operations $\phi, \pi \in \Pi$ satisfy Property 1.*

Proof. Follows directly from Lemmas 22 and 23. □

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 25 (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 5 (lines Algorithm 5:21–29), 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

ALGORITHM 7: DSMM: Reconfig operation on block b at client p

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

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 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 $|S_\omega| = |S_\pi| = \lceil \frac{n+k}{2} \rceil$ 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 $|S_\omega \cap S_\pi^k| \geq \lceil \frac{n+k}{2} \rceil + \lceil \frac{n+1}{3} \rceil \geq 1$, 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 Algorithm 6:6–9 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. □

7.4 CoAresF: Integrate CoAres with CoBFS

We are now ready to 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 (utilizing the FM implementation described in Section 5) 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 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.

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 Algorithm 7, while the specification of reconfig on a fragmented object is given in Algorithm 8.

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

ALGORITHM 8: FM: Reconfig operation on fragmented object f at client p

```
1: function fm-reconfig(c)_{f,p}
2:     b ← val(b_0).ptr
3:     while b not NULL do
4:         dsmm-reconfig(c)_{b,p}
5:         b ← val(b).ptr
6:     end while
7: end function
```

the last block by following the next block ids (Algorithm 1). The reconfig operation executes the *block reconfig* operations on the shared memory (Algorithm 7) using dsmm-reconfig operations.

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 4.2) cannot be violated. Before we prove any lemmas, we first state a claim that follows directly from the algorithm.

Claim 26. *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 [23].

Lemma 27. *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 wrote by a preceding write operation to the installed configuration.

Lemma 28. *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. $ver' \geq ver$. The proof of this lemma is provided in Appendix A.14.*

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 29. *In any execution ξ of* CoAresF*, if ρ is a read operation on f that returns a sequence of blocks $\mathcal{L} = \{b_0, b_1, \ldots, b_n\}$, 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) $b_n.ptr = \perp$. The proof of this lemma is provided in Appendix A.15.*

We conclude with the main result of this section.

Theorem 30. CoAresF *implements a linearizable coverable fragmented object.*

Proof. By the correctness proof in Section 7.2 follows that every block operation in CoAresF satisfies linearizable coverability and together with Lemma 29, which shows the connectivity of blocks, it follows that CoAresF implements a linearizable coverable fragmented object satisfying the properties of *fragmented linearizable coverability* (see Section 4.2). □

8 Experimental Evaluation

After presenting the implementations, we now proceed to an experimental evaluation. Section 8.1 describes general information for the setup of experiments and Section 8.2 presents the experiments. Our implementation is a prototype of a Distributed File System, thus each file is modeled as a fragmented object with blocks as presented in Section 5. While in these experiments we consider text files with random byte strings, our implementations support any file type.

8.1 Experimental Evaluation Setup

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 [20] as an emulation testbed and AWS EC2 [8] as an overlay testbed. **Evaluated Algorithms.** We have implemented and evaluated the performance of the following algorithms:

- CoABD. This is the coverable version of the traditional static ABD [5, 49], as introduced in [50], incorporating the optimization detailed in Section 6.1. It serves as our overall baseline.
- CoABDF. This is the version of optimized CoABD that provides fragmented coverability, as presented in Section 4. It can be considered as a baseline algorithm of the CoBFS framework.
- CoAresABD. This is a coverable version of Ares (CoAres), presented in Section 7.2, that uses the ABD-DAP implementation (cf. Section 7.1). The ABD-DAP implementation is optimized similarly to the EC-DAPopt presented in Section 7.3. It can be considered as the dynamic (reconfigurable) version of CoABD.
- CoAresABDF. This is CoAresF (cf. Section 7.4) together with the ABD-DAP implementation, i.e., it is the fragmented version of CoAresABD.
- CoAresEC. This is a version of CoAres (see Section 7.2) that uses the EC-DAPopt implementation (see Section 7.3).
- CoAresECF. This is the two-level striping algorithm presented in Section 7.4 when used with the EC-DAPopt implementation of Section 7.3, i.e., it is the fragmented version of CoAresEC.
- Cassandra. Cassandra is a highly available, tunable consistent key-value store. We use it to provide a performance reference for comparison (see Section 2 and the implementation details below).
- CassandraF. This is the fragmented variant of Cassandra, adapted to support object chunking in order to handle large objects (see the implementation details below).

Note that we have implemented all the above algorithms using the same baseline code and communication libraries. All the modules of the algorithms are written in Python, and the asynchronous communication between layers is achieved by using DEALER and ROUTER sockets, from the ZeroMQ library [63].

In the remainder, for ease of presentation, and when appropriate, we will be referring to algorithms CoABD(F) and CoAresABD(F) as the ABD-based algorithms and to algorithms CoAresEC(F) as the EC-based algorithms.

Overview of the Experiments. In a first phase, to further appreciate the proposed approach from an applied point of view, we performed a preliminary evaluation of CoBFS with CoABD storage (CoABDF) against the original CoABD. Due to the design of the two algorithms, CoABD will transmit the entire file per read/update operation, while CoABDF will transmit as many blocks as necessary for an update operation, but perform as many reads as the number of blocks during a read operation. The two algorithms use the read optimization of Algorithm 3. This preliminary experimentation evaluated the performance of read and write operations and of the read optimization.

In a second phase, we compared the performance of CoBFS using different storage emulations (CoABD, CoAresABD, CoAresEC) against the original ones without the fragmentation approach of CoBFS. We have evaluated the impact of using CoBFS instead of the original storage. To this end, we show the clear benefits of using CoBFS over different storage, especially in the case of concurrent accesses to the same huge file, and find the optimized storage for this framework. We just display a small portion of the preliminary experiments because these extended ones contain all the findings.

Deployment and Execution. For the deployment and remote execution of the experimental tasks, we have an extra physical machine, the controller, which orchestrates the experiments. The controller used *Ansible* [3], a tool to automate different IT tasks, such as cloud provisioning, configuration management, application deployment, and intra-service orchestration. There are two main steps to run an experiment: (*i*) booting up the client (either writer or reader) and the server nodes, and (*ii*) executing each scenario using Ansible Playbooks, scripts written in the YAML language. The scripts get pushed to target machines, they are executed, and then get removed. In our experiments, one instance node was dedicated as a controller to orchestrate the experiments. For the execution of the experiment, Ansible automated the provision of the executables in each machine, the execution of the operations in the experiment, and the collection of the logs for our analysis.

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

- **writer** $w \in \mathcal{W} \subseteq \mathcal{C}$**:** a client that sends write requests to all servers and waits for a quorum of the servers to reply.
- **reader** $r \in \mathcal{R} \subseteq \mathcal{C}$**:** a client that sends read requests to servers and waits for a quorum of the servers to reply.
- **reconfigurer** $g \in \mathcal{G} \subseteq \mathcal{C}$**:** 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** $s \in \mathcal{S}$**:** 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 Metrics. We assess performance using: (i) *operational latency*, and (ii) *the update success ratio*. The operational latency is computed as the sum of communication and computation delays. In the case of algorithms that use CoBFS, computational latency encompasses the time necessary for the FM to fragment a file object and generate the respective hashes for its blocks. The update success ratio is the percentage of update operations that have not been converted to reads (and thus successfully changed the value of the indented object). In the case of non-fragmented algorithms, we compute the percentage of successful updates on the file as a whole over the number of all updates. For fragmented algorithms, we compute the percentage of file updates, where all individual block updates succeed. The performance of ABD-based algorithms shown in the results can be used as a reference point in the presented experiments since the rest algorithms combine

ideas from it. To perform a fair comparison and to yield valuable observations, the results shown are compiled as averages over several samples per each scenario. The Emulab results are compiled as averages over five samples per each scenario. However, the AWS results are complied as averages over three samples for the Scalability scenario, while the rest scenarios run only once.

Distributed Experimental Setup on Emulab. For the preliminary experiments, we used XEN virtual machines with a routable IP address, while for the extended experiments, we used physical machines. All 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 node 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 node 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.

Reconfiguration Service of Any Variant of Ares. We used an external implementation of Raft [52] 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.

Cassandra Implementation. We deployed the Apache Cassandra 4.1 on each node. In order to guarantee atomicity, as in Ares and ABD, we set the consistency level parameter of Cassandra to "quorum". This means that a majority of nodes of the replicas must respond. Thus, if n is the total number of available replicas, and replication factor is n, then $\lfloor \frac{n}{2} \rfloor + 1$ must respond. To send read and write request we created a script using the Cassandra-driver Python library. First, the script creates connections to the cluster nodes, giving their IPs and ports. Then we specify a keyspace (a namespace that defines data replication on nodes) and create a table (a list of key-value pairs). Once that is done, the client can send write and read requests, using the *insert* and *select* statements, respectively. We implemented two distinct approaches for interacting with Cassandra: (i) Non-fragmented Implementation: In this approach, we treat each file as a single large object. A writer inserts a tuple ($fileid$, $value$), where the value is a byte string of type blobs (binary large objects) in Cassandra. A reader selects the value providing the file's id; (ii) Fragmented Implementation: This approach (CassandraF) simulates fragmentation by breaking down each file into smaller, fixed-size chunks. For writing, each chunk is inserted as a separate entry, with a tuple ($fileid$, $chunk\_number$, $value$). For reading, the client iterates through *chunk_number* values to retrieve all chunks associated with a file id and then reassembles them to reconstruct the original file.

Parameters of Algorithms. The quorum size of the EC-based algorithms is $\lceil \frac{n+k}{2} \rceil$, while the quorum size of the ABD-based algorithms is $\lfloor \frac{n}{2} \rfloor + 1$. 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 δ in EC-based algorithms is the maximum number of concurrent put-data operations, i.e., the number

of writers. For the CASSANDRA algorithm, we set the consistency level to the majority, as in the case of ABD-based algorithms.

8.2 Experimental Scenarios and Results

Here, we describe the scenarios we constructed and the settings for each of them. In all scenarios a writer bootstraps the system by writing a text file with a specific size, i.e., the initial size. As the writers keep updating the file, its size increases.

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 $[1..wInt]$. If there is a reconfigurer, it invokes its next operation every $15sec$ and performs a total of 5 reconfigurations.

Scenarios. We present three 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).

Overall, our results suggest that the efficiency of CoBFS is inversely proportional to the number of block operations, rather than the size of the file. This is primarily due to the individual block-processing nature of CoBFS.

Performance vs. Initial File Sizes. The first scenario measures the performance of algorithms when the writers update a file whose size gradually increases. We varied the f_{size} by doubling the file size in each simulation 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. This demonstrates that non-fragmented algorithms cannot handle large data objects in a relatively large deployment (11 servers) without exceeding memory limits, whereas fragmented algorithms complete the operations successfully. The maximum, minimum and average block sizes (*rabin fingerprints* parameters) were set 1 MB, 512 kB, and 512 kB, respectively. In the preliminary Emulab experiments, we focused on comparing only CoABDF and CoABD to evaluate the impact of fragmentation and the optimization in Algorithm 3. These early experiments validated the benefit of fragmentation in isolation. The extended experiments expand the evaluation to all algorithm variants under larger deployments in Emulab and AWS.

Preliminary Experiments: $|\mathcal{W}| = 5, |\mathcal{R}| = 5, |\mathcal{S}| = 5$, f_{size} from 1 MB to 1 GB. In total, each writer performed 5 updates and each reader 5 reads.

Extended Experiments: f_{size} from 1 MB to 512 MB. The ABD-based algorithms have quorums of size 3.

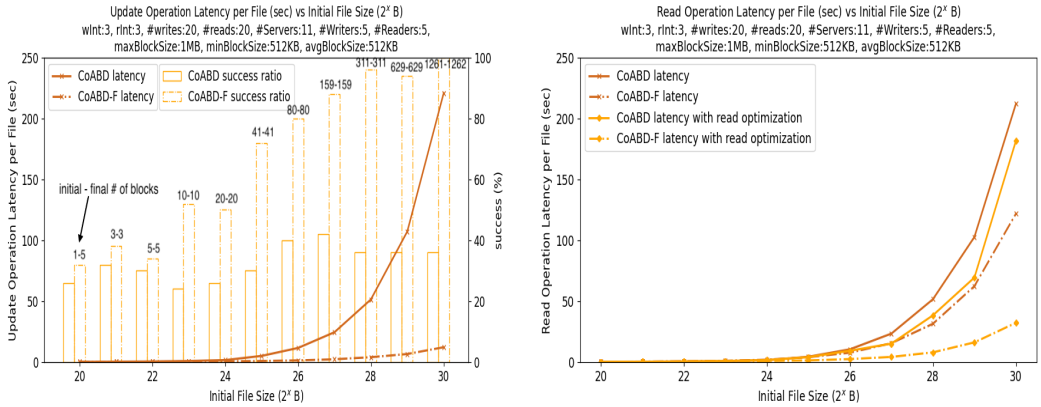
Emulab Parameters. $|\mathcal{W}| = 5, |\mathcal{R}| = 5, |\mathcal{S}| = 11$. For EC-based algorithms we used parity $m \in \{1, 5\}$ yielding quorum sizes of 11 and 9, respectively. For ABD-based algorithms and CASSANDRA we used quorums of size 6. In total, each writer performs 20 writes and each reader 20 reads.

AWS Parameters. $|\mathcal{W}| = 1, |\mathcal{R}| = 1, |\mathcal{S}| = 6$. For EC-based algorithms we used parity $m \in \{1, 4\}$ yielding quorum sizes of 6 and 4, respectively. For ABD-based algorithms and CASSANDRA we used quorums of size 4. In total, each writer performs 50 writes and each reader 50 reads. The above experimental parameters are summarized in Table 3.

Results: We measure the read and write operation latencies for both original and the fragmented variant of algorithms; the results can be seen on Figures 4, 5, and 6. As shown in Figures 5(a) and 6(a), the fragmented algorithms that use the *FM* achieve significantly smaller write latency,

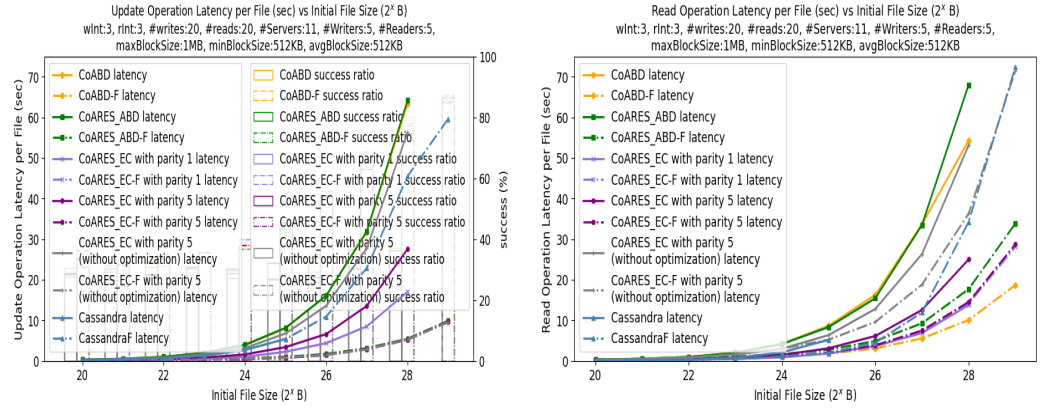
Table 3. Experimental Parameters Summary for File Size Experiments

Parameter	Preliminary Experiments Emulab	Extended Experiments Emulab	Extended Experiments AWS
File Size Range	1 MB – 1 GB	1 MB – 512 MB	1 MB – 512 MB
$\|\mathcal{S}\|$	5	11	6
$\|\mathcal{W}\|$	5	5	1
$\|\mathcal{R}\|$	5	5	1
Ops per Client	5	20	50
Parity m (EC-based)	—	1 or 5	1 or 4
Quorum Size (ABD-based/CASSANDRA)	3	6	4
Quorum Size (EC-based)	—	11 for $m = 1$ or 9 for $m = 5$	6 for $m = 1$ or 4 for $m = 4$



(a) (b)

Fig. 4. Emulab preliminary results for File Size experiments. (a) Write operation latency as file size increases. (b) Read operation latency as file size increases.



(a) (b)

Fig. 5. Emulab results for file size experiments. (a) Write operation latency as file size increases. (b) Read operation latency as file size increases.

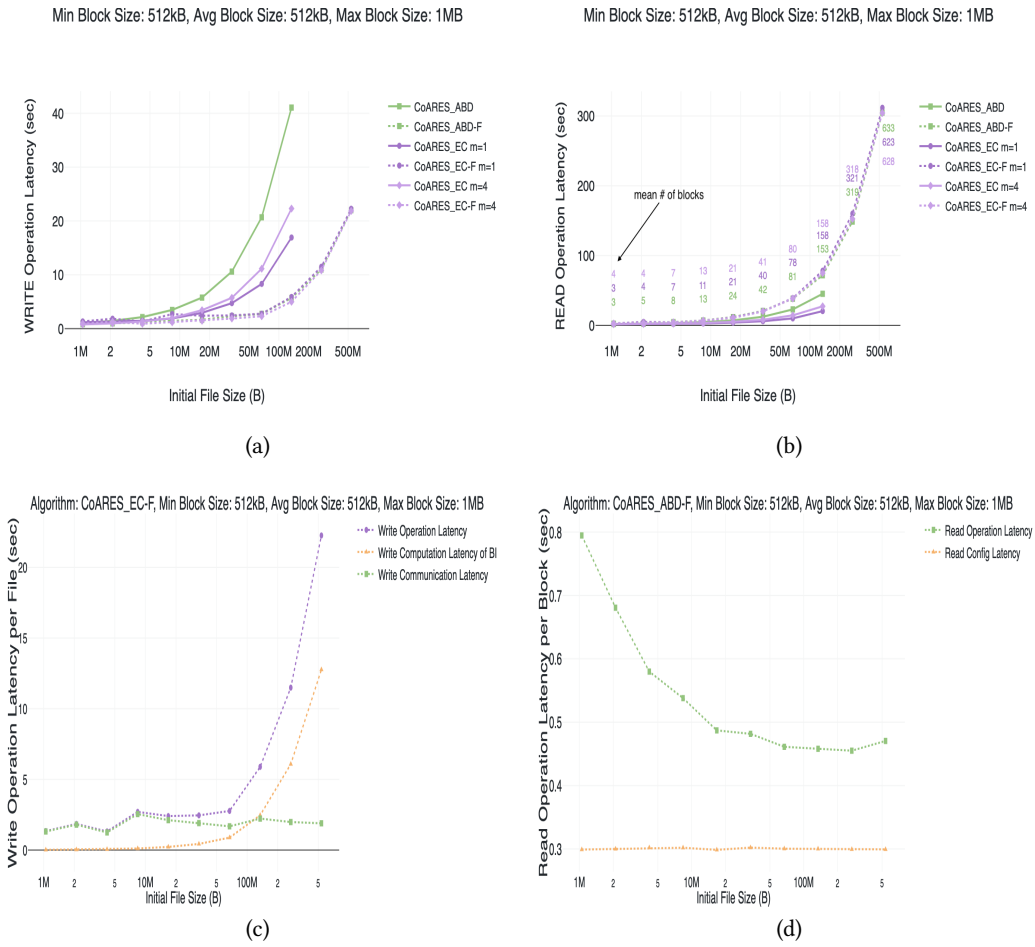


Fig. 6. AWS results for File Size experiments. (a) Write operation latency as file size increases. (b) Read operation latency and average number of blocks read as file size increases. (c) Breakdown of CoAresECF write latency: total write, BI computation, and communication as file size increases. (d) Read operation latency and read-config latency as file size increases.

when the file size increases, which is a result of the block distribution strategy. Both Cassandra and its fragmented counterpart CassandraF (Figure 5(a)) are significantly outperformed by all of our non-fragmented algorithms, as well as by all optimized EC-based algorithms, whether fragmented or not. This highlights the scalability and efficiency of our designs in handling large data workloads. In Figure 5(a), the lines of fragmented algorithms are very closed to each other. The fact that the CoAresECF with m=1 (Figure 6(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. Regarding CassandraF, for write operations it outperforms Cassandra for file sizes up to 32 MB (2^{25} bytes), which is the maximum file size Cassandra can handle without errors. 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 (Figure 5(a)). On the contrary, the non-fragmented algorithms do not experience any improvement as it always manipulates the file as a whole. We do not measure CASSANDRA's success ratio as it does not provide a versioning mechanism; hence, all updates are considered successful.

The BI computation latency contributes significantly to the increase of fragmented algorithms' update latency in larger file sizes, as shown in Figure 6(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. An optimization of the rabin algorithm or a use of a different algorithm for managing blocks could possibly lead to improved BI computation latency; this is a subject for future work. The CoBFS update communication latency remains almost stable, since it depends primarily on the number and size of update block operations. That is in contrast to the update latency exhibited in non-fragmented algorithms which appears to increase linearly with the file size. This was expected, since as the file size increases, it takes longer to update the whole file.

As shown in Figure 5(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. CASSANDRAF outperforms CASSANDRA read latency for file sizes up to 32 MB (2^{25} bytes), which is the maximum size CASSANDRA can handle without errors. In this range, CASSANDRAF achieves latencies close to those of our fragmented algorithms. However, for file sizes beyond 64 MB (2^{26} bytes), the latency of CASSANDRAF increases significantly, exceeding that of both our fragmented algorithms and even non-fragmented EC-based algorithms.

On the contrary, the read latency of CoARES in the corresponding AWS experiment (Figure 6(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 Figure 6(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 Figure 6(b). This reveals an important observation: in static algorithms (e.g., CoABDF), fragmentation improves read latency because only modified blocks are transmitted, and no extra metadata exchange is needed. In contrast, reconfigurable algorithms (e.g., CoARESF) may incur higher latencies despite transferring less data, due to the need to retrieve or synchronize configuration state. To address this in a follow-up work [31], several optimizations have been proposed and evaluated: (*i*) piggybacking configuration metadata on client messages to reduce discovery overhead, (*ii*) a garbage collection mechanism that prunes obsolete configurations and propagates updated ones, and (*iii*) batching reconfiguration actions across multiple objects, which is particularly effective in fragmented settings. 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.

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.

Table 4. Experimental Parameters Summary for Scalability Experiments

Parameter	Emulab	AWS
File Size	4 MB	4 MB
Block Sizes	Min: 512 kB, Avg: 512 kB, Max: 1 MB	Min: 512 kB, Avg: 512 kB, Max: 1 MB
$\|\mathcal{S}\|$	3, 5, 7, 9, 11	3, 5, 7, 9, 11
$\|\mathcal{R}\|$	5 to 25	5 to 25
$\|\mathcal{W}\|$	5 to 25	5 to 25
Ops per Client	20	20
Client Placement	All clients on different nodes	Round-robin across nodes
Parity m (EC-based)	1 ($\|\mathcal{S}\| = 3$), 2 ($\|\mathcal{S}\| = 5$), 3 ($\|\mathcal{S}\| = 7$), 4 ($\|\mathcal{S}\| = 9$), 5 ($\|\mathcal{S}\| = 11$)	1 ($\|\mathcal{S}\| = 3$), 2 ($\|\mathcal{S}\| = 5$), 3 ($\|\mathcal{S}\| = 7$), 4 ($\|\mathcal{S}\| = 9$), 5 ($\|\mathcal{S}\| = 11$)
Quorum Size (ABD-based)	2 ($\|\mathcal{S}\| = 3$), 3 ($\|\mathcal{S}\| = 5$), 4 ($\|\mathcal{S}\| = 7$), 5 ($\|\mathcal{S}\| = 9$), 6 ($\|\mathcal{S}\| = 11$)	2 ($\|\mathcal{S}\| = 3$), 3 ($\|\mathcal{S}\| = 5$) 4 ($\|\mathcal{S}\| = 7$), 5 ($\|\mathcal{S}\| = 9$), 6 ($\|\mathcal{S}\| = 11$)
Quorum Size (EC-based)	3 ($\|\mathcal{S}\| = 3$), 4 ($\|\mathcal{S}\| = 5$), 6 ($\|\mathcal{S}\| = 7$), 7 ($\|\mathcal{S}\| = 9$), 9 ($\|\mathcal{S}\| = 11$)	3 ($\|\mathcal{S}\| = 3$), 4 ($\|\mathcal{S}\| = 5$) 6 ($\|\mathcal{S}\| = 7$), 7 ($\|\mathcal{S}\| = 9$), 9 ($\|\mathcal{S}\| = 11$)

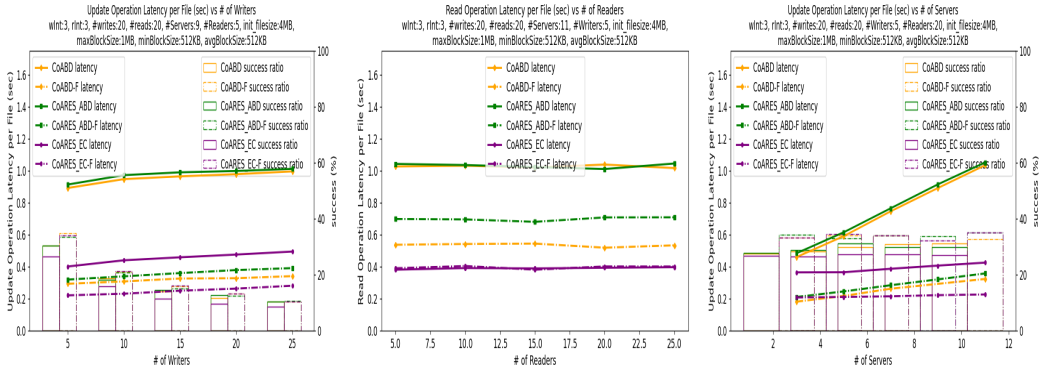
In the preliminary experiments, we compared the read latencies of CoABDF and CoABD with and without the optimization of Algorithm 3.As seen in Figure 4(b), the CoABD read latency increases sharply, even when using the optimized reads. This is in line with our initial hypothesis, as CoABD requires reads to request and propagate the whole file each time a newer version of the file is discovered. Similarly, when the optimization is not used in CoABDF, the latency is close of CoABD. Notice that each read that discovers a new version of the file needs to request and propagate the content of each individual block. On the contrary, the optimization decreases significantly the CoABDF read latency, as reads transmit only the contents of the blocks that have changed.

In Figure 5(a) and (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).

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 $|R|$ 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. 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 $|\mathcal{S}| = 11$. The above experimental parameters are summarized in Table 4.

Results: The results obtained in this scenario are presented in Figure 7. 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



(a) (b) (c)

Fig. 7. Emulab results for Scalability experiments. (a) Write operation latency as the number of writers increases. (b) Read operation latency as the number of readers increases. (c) Write operation latency as the number of servers increases.

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 Figure 7(a)).This is because the non-fragmented writer updates the whole file, while each fragmented writer updates only a subset of blocks. To complement the latency analysis, we also measured the number of operations per second (throughput) each algorithm can sustain. The results confirm the improved scalability of fragmented algorithms. For instance, while the non-fragmented CoABD achieves only 1.61 ± 0.51 update ops/sec, its fragmented counterpart CoABDF sustains 4.45 ± 0.99 update ops/sec. Similarly, CoAresEC handles 2.57 ± 0.30 update ops/sec, compared to 4.89 ± 0.35 by CoAresECF. 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 (Figures 7(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 wait for responses from more servers, CoAresEC gains the advantage of 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. Figure 7(a) and (b)), explaining the slower CoAresF read operation (cf. Figures 7(b)). The corresponding AWS findings show similar trends.

With Reconfiguration: We built three extra experiments in Emulab to verify the correctness of the variants of Ares when reconfigurations coexist with read/write operations. We assume a crash-stop failure model, where processes (including servers and reconfigurers) may fail by halting. No failures were explicitly injected during these experiments. The reconfigurations simulate planned, dynamic membership changes, rather than recovery from node crashes. Our goal was to evaluate the correctness and performance of the system under frequent reconfiguration scenarios. The three experiments differ in the way the reconfigurer works; two 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.

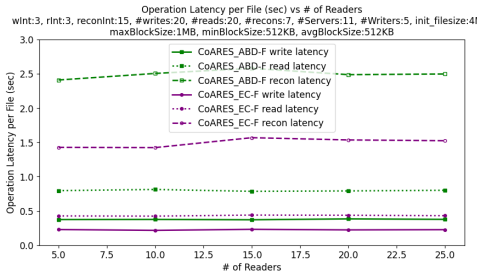


Fig. 8. Emulab results for read, write, and reconfiguration operations as the number of readers increases when reconfiguring to the same DAP_s.

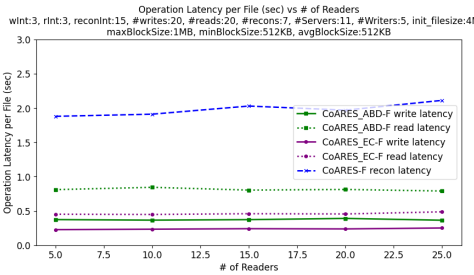


Fig. 9. Emulab results for read, write, and reconfiguration operations as the number of readers increases when reconfiguring DAP_s Randomly.

— **Reconfiguring to the same DAP**: 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.
— **Reconfiguring to a random DAP**: The reconfigurer chooses randomly between the two DAP_s.
— **Reconfiguring to a different number of servers**: The reconfigurer switches between the two DAP_s and at the same time chooses randomly the number of servers between $[3, 5, 7, 9, 11]$.

Results: As expected, in all scenarios, the reconfiguration delays are higher than those of read and write operations (Figures 8, 9, and 10), since a reconfiguration involves more communication rounds, including interactions with the external consensus service and the execution of DAP operations. A detailed tracing analysis of operation latencies is provided in a follow-up work [32].

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. Figure 8). 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.

Figure 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 reconfiguration operation. We observe that the cost of accessing multiple DAPs within a single read/write operation is more noticeable in non-fragmented variants. In these cases, the protocols must transfer the entire object across configurations, which significantly increases latency. In contrast, fragmented versions of the protocols exhibit a much smaller increase in latency, since they move smaller fragments. This highlights an important advantage of our fragmentation approach, it not only lowers latencies but also reduces the overhead introduced by dynamic transitions across heterogeneous DAPs.

The last scenario in Figure 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 (Figure 10(b)) has shorter update and read latencies than CoAres (Figure 10(a)).

Performance vs. Min/Avg Block Sizes. We varied the minimum and average b_{sizes} of fragmented algorithms from 8 kB to 1 MB. The size of the initial file used was set to 4 MB, while the maximum

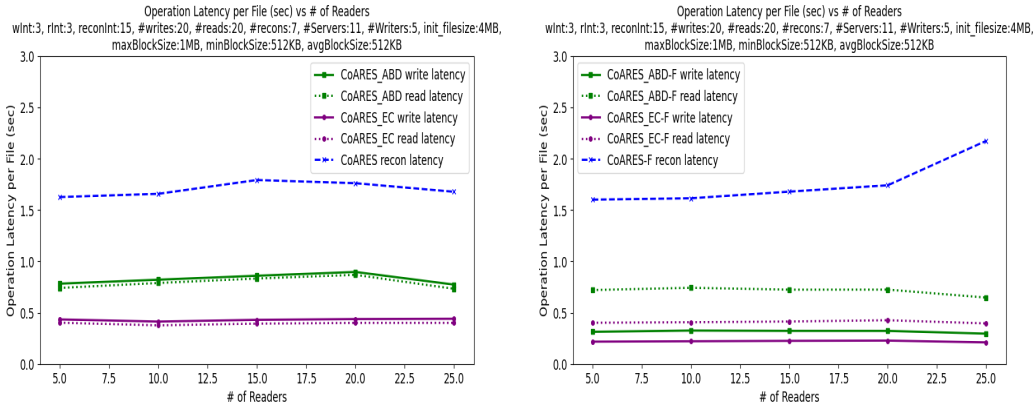


Fig. 10. Emulab results for read, write, and reconfiguration operations as the number of readers increases when Reconfiguring DAP_s Alternately and Servers Randomly. (a) Non-fragmented algorithms. (b) Fragmented algorithms.

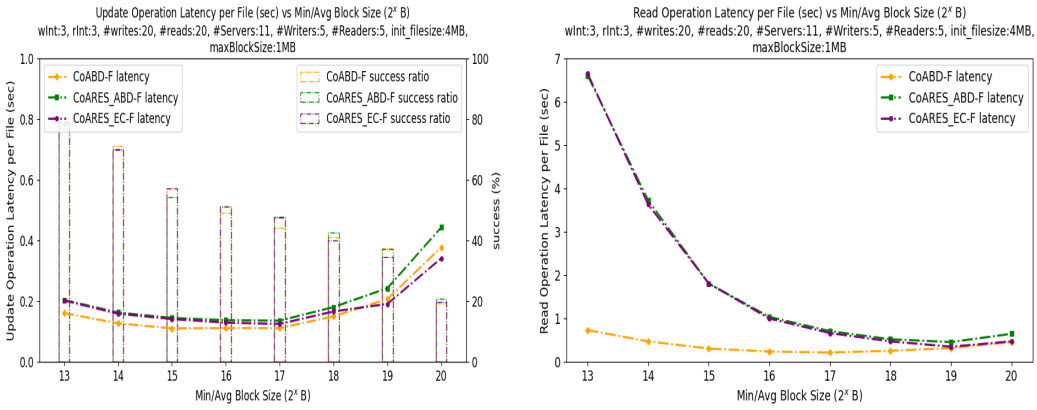


Fig. 11. Emulab results for Min/Avg block sizes' experiments. (a) Write operation latency as Min/Avg block sizes increases. (b) Read operation latency as Min/Avg block sizes increases.

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.

Emulab Parameters. $|\mathcal{W}| = 5, |\mathcal{R}| = 5, |\mathcal{S}| = 5$. For EC-based algorithms, $m = 2$ and the quorum size is 4. 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. The above experimental parameters are summarized in Table 5.

Results: From Figure 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,

Table 5. Parameters for Block Size Experiments

Parameter	Emulab	AWS
File Size	4 MB	4 MB
Min/Avg Block Size Range	8 kB – 1 MB	8 kB – 1 MB
Max Block Size	1 MB	1 MB
$\|\mathcal{S}\|$	5	6
$\|\mathcal{R}\|$	5	1
$\|\mathcal{W}\|$	5	1
Ops per Client	20	50
Parity m (EC-based)	2	1
Quorum Size (ABD-based)	4	4
Quorum Size (EC-based)	4	6

resulting in more update block operations, and hence slightly higher update latency. In Figure 11(b), smaller block sizes require more read block operations to obtain the file's value. As the minimum and average b_{sizes} increase, lower number of rather small blocks need to be read. Thus, further increase of the minimum and average b_{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_{sizes} and increasing them does not give better (or worse) read latency. The corresponding AWS findings show similar trends.

Performance vs. Min/Avg/Max Block Sizes. 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 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.

Results: 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 (Figures 13(a), and 12(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, as shown in Figure 13(b). However, in real time conditions of AWS, the read latency of a higher number of relatively large blocks (Figure 12(c)) has a significant impact on overall latency, resulting in a larger read latency (Figure 12(b)).

9 Conclusions

Existing methods for managing data in distributed storage systems have several shortcomings, such as centralized metadata management, weaker guarantees, and a lack of support for efficient versioning of data in concurrency. The development of CoBFS, a highly efficient distributed data storage framework, has addressed these limitations and facilitated data sharing at a large scale. The benefits of CoBFS were demonstrated through various experiments on both emulation and real conditions environments, meeting the main objective proposed in this work. We highlight the usefulness of the contributions and provide future perspectives for further development.

CoBFS**: Efficient, Large-scale Data Storage.** We proposed CoBFS, a distributed storage service that illustrates design principles for building a distributed storage system that can scale to large size data. To develop it, we had to introduce the notion of linearizable and coverable fragmented objects that are based in two design principles: data striping and versioning-based concurrency control in a storage to enable efficient data management of large objects.

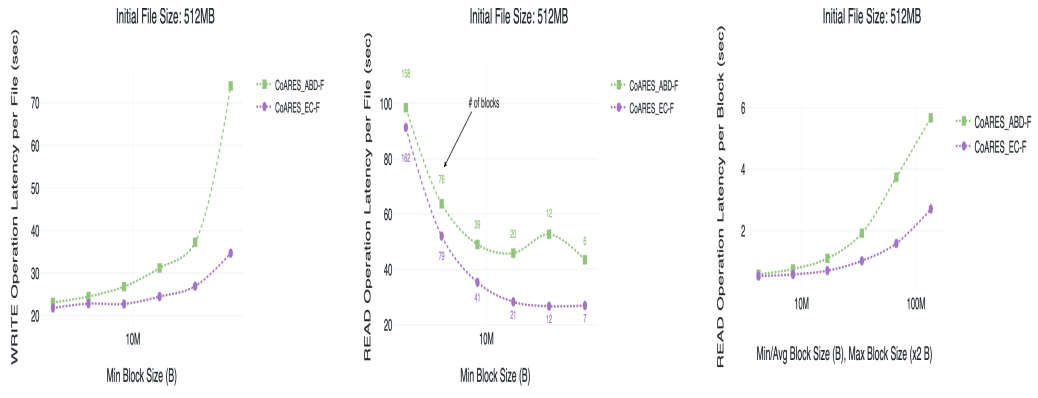


Fig. 12. AWS results for Min/Avg/Max block sizes' experiments. (a) Write operation latency per file as Min/Avg block sizes increases. (b) Read operation latency per file as Min/Avg block sizes increases. (c) Read operation latency per block as Min/Avg block sizes increases.

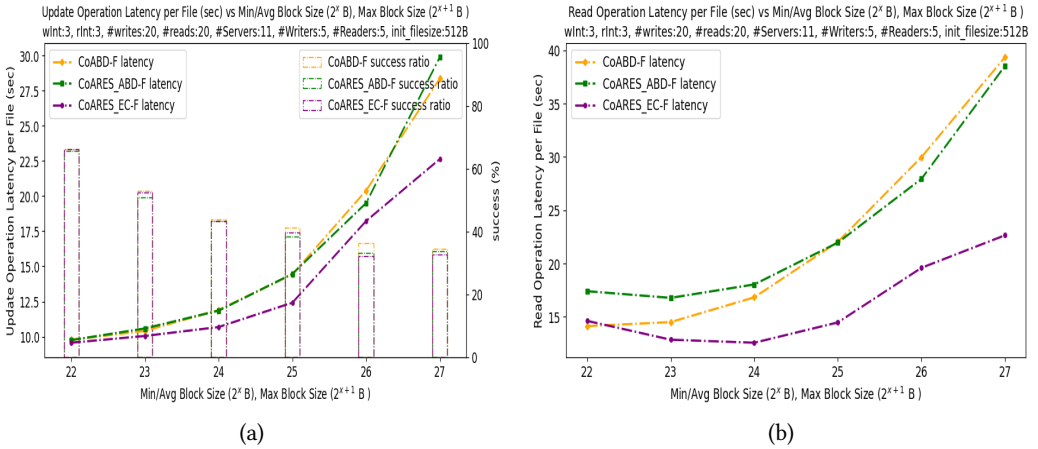


Fig. 13. Emulab results for Min/Avg/Max block sizes' experiments. (a) Write operation latency as Min/Avg/Max block sizes increases. (b) Read operation latency as Min/Avg/Max block sizes increases.

CoBFS **Improves Performance of Distributed Shared Memory.** The efficiency of CoBFS is mainly affected by the storage layer. Thanks to its modular architecture, we are able to use different storage emulations with different design principles (such as versioning, fault-tolerance, erasure-codes, block replication, fast operations). Among different storages we test on the core of CoBFS, we developed CoAresF, the first dynamic distributed shared memory that utilizes coverable fragmented objects and enables the use of erasure coding. Compared to the approach that does not use the fragmentation layer of CoBFS (CoAres), CoAresF is optimized for efficient access to shared data under heavy concurrency.

Theoretical Principles Illustrated through Extensive Experiments. To validate the theoretical principles of each developed algorithm, we performed extensive experimental scenarios. In all evaluations, we examine the design principles that each algorithm promises, e.g., atomic consistency, data striping, erasure coding, access to the same files under heavy concurrency, fault-tolerance, reconfiguration.

Below we discuss the main *tradeoffs* that we have observed during the implementation and deployment.

Block size of *FM*. The performance of data striping highly depends on the block size. There is a tradeoff 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 ensures 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 tradeoff 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 equals 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 understand the importance of the optimization in the DSMM layer.

Our storage service demonstrates strong potential for future development and improvement, as described below. In fact, new features can be leveraged to enhance the system's capabilities.

Enhance the Performance of our DSS. Although the work has achieved promising results, there is still potential for further improvement or optimization. During the extensive experiments in this work, we have already identified performance bottlenecks in distributed setting. The main goal is to devise methodologies to reduce the latency of read, write, and reconfig operations, making our DSS prototype more practical and attractive for commercial use. In this work, we observed the stable overhead of read-config operations in the experiment sets of Section 8.2. Another issue of the Ares algorithms is the old or even unviable configurations that exist in *cseq* in any variant of Ares (cf. Section 7.1). Subsequent work [32], introduced distributed tracing to find the performance bottlenecks of DSM algorithms, specifically the Ares algorithm and its variants. By identifying bottlenecks, key optimizations we implemented in [31] that enhanced the algorithm's efficiency, such as batched operations, garbage collection, and piggybacking configuration data. These improvements offer significant performance gains while maintaining correctness. The optimizations include piggy-backing data on read/write messages to speed up configuration discovery, implementing a garbage collection mechanism to remove outdated configurations and update older ones for improved service longevity and faster discovery, and introducing a batching mechanism to apply a single configuration to multiple objects at once, accelerating reconfiguration. Certainly, there is more room for improvement as several aspects remain open for future exploration. In particular, integrating lightweight data integrity mechanisms (e.g., checksums) and adopting client/server-side caching techniques–common in existing DDS like HDFS or Tectonic– will further enhance the reliability and performance of CoBFS, and thus, broaden the system's applicability in real-world settings. Future work will also focus on advanced management of DSM via a distributed memory management module, enabling orchestration of multiple DSM instances, optimized object placement and migration, and improved scalability, fault tolerance, and resource utilization.

Design Reconfiguration Orchestration Strategies for Dynamic DSS. Utilizing the reconfiguration mechanism offered by CoAresF, a future goal is to design and analyze (smart) **Reconfiguration Orchestration Strategies** (**ROS**) on when reconfigurations should be invoked on CoAresF and how the membership of the service should change. Proposed approaches will specify which environmental parameters may affect the decisions on when and how to reconfigure, and reconfiguration decisions will utilize (existing) tools that monitor these parameters, eliminating or minimizing human intervention. The developed service will be designed to interact with any dynamic reconfigurable service (beyond CoAresF) via the reconfiguration mechanism.

Real-world Implementations of the DSMs. The DSM systems presented in this work, are being actively integrated into practical, real-world applications that span both edge and cloud domains. For instance, we are deploying ADSM in maritime surveillance scenarios involving autonomous **unmanned vehicles** (**UVs**), such as aerial drones and underwater robots, which operate in dynamic edge environments. Here, ADSM functions as a lightweight coordination and data-sharing backbone to support tasks like swarm intelligence and real-time monitoring. In parallel, we are developing a web-based platform to facilitate the deployment, configuration, and use of ADSM via a user-friendly UI, built on Laravel. This platform enables users to manage configurations, monitor system state, and perform read/write operations efficiently. Together, these efforts demonstrate the flexibility and applicability of our DSM solutions in both mission-critical and general-purpose computing environments.

References

[1] S.V. Adve and K. Gharachorloo. 1996. Shared memory consistency models: A tutorial. *Computer* 29, 12 (1996), 66–76. DOI: 10.1109/2.546611

[2] M. K. Aguilera, I. Keidar, D. Malkhi, and A. Shraer. 2009. Dynamic atomic storage without consensus. In *Proceedings of the 28th ACM Symposium on Principles of Distributed Computing*. ACM, New York, NY, USA, 17–25.

[3] Ansible [n. d.]. Ansible. Retrieved May 04, 2023 from https://www.ansible.com/overview/how-ansible-works

[4] H. Attiya. 2010. Robust simulation of shared memory: 20 years after. *Bulletin of the EATCS* 100 (2010), 99–114.

[5] H. Attiya, A. Bar-Noy, and D. Dolev. 1995. Sharing memory robustly in message-passing systems. *Journal of the ACM* 42, 1 (1995), 124–142.

[6] H. Attiya, S. Kumari, A. Somani, and J. L. Welch. 2020. Store-Collect in the Presence of Continuous Churn with Application to Snapshots and Lattice Agreement. arXiv:2003.07787. Retrieved from https://arxiv.org/abs/2003.07787

[7] Hagit Attiya and Jennifer L. Welch. 1994. Sequential consistency versus linearizability. *ACM Trans. Comput. Syst.* 12, 2 (May 1994), 91–122. DOI: https://doi.org/10.1145/176575.176576

[8] AWS EC2 [n. d.]. AWS EC2. Retrieved May 04, 2023 from https://aws.amazon.com/ec2/

[9] Leander Beernaert, Pedro Gomes, Miguel Matos, Ricardo Vilaça, and Rui Oliveira. 2013. Evaluating cassandra as a manager of large file sets. In *Proceedings of the 3rd International Workshop on Cloud Data and Platforms* (Prague, Czech Republic). Association for Computing Machinery, New York, NY, USA, 25–30. DOI: 10.1145/2460756.2460761

[10] Ken Birman. 2005. *Reliable Distributed Systems: Technologies, Web Services, and Applications*. Springer. xxxi, 668 p. pages. DOI: 10.1007/0-387-27601-7

[11] Paul E. Black. 2004. Ratcliff/obershelp pattern recognition. Retrieved from https://xlinux.nist.gov/dads/HTML/ratcliffObershelp.html

[12] A. Carpen-amarie. 2012. *BlobSeer as a Data-Storage Facility for Clouds: Self-Adaptation, Integration, Evaluation*. Ph. D. Dissertation. Université de Rennes, France.

[13] Cassandra [n. d.]. Cassandra. Retrieved October 08, 2022 from https://cassandra.apache.org/_/index.html

[14] Armando Castañeda, Sergio Rajsbaum, and Michel Raynal. 2022. A linearizability-based hierarchy for concurrent specifications. *Communications of the ACM* 66, 1 (2022), 86–97. DOI: 10.1145/3546826

[15] Colossus [n. d.]. Colossus. Retrieved October 08, 2022 from https://cloud.google.com/blog/products/storage-data-transfer/a-peek-behind-colossus-googles-file-system

[16] C. Delporte-Gallet, H. Fauconnier, S. Rajsbaum, and M. Raynal. 2018. Implementing snapshot objects on top of crash-prone asynchronous message-passing systems. *IEEE Trans. Parallel Distrib. Syst.* 29, 9 (2018), 2033–2045. DOI: 10.1109/TPDS.2018.2809551

[17] Giacinto Donvito, Giovanni Marzulli, and Domenico Diacono. 2014. Testing of several distributed file-systems (HDFS, Ceph and GlusterFS) for supporting the HEP experiments analysis. *Journal of Physics: Conference Series* 513, 4 (2014), 042014. DOI: 10.1088/1742-6596/513/4/042014

[18] Dropbox [n. d.]. Dropbox. Retrieved October 08, 2022 from https://www.dropbox.com/

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

[20] Emulab [n. d.]. Emulab Network Testbed. Retrieved October 08, 2022 from https://www.emulab.net/

[21] Burkhard Englert, Chryssis Georgiou, Peter M. Musial, Nicolas Nicolaou, and Alexander A. Shvartsman. 2009. On the efficiency of atomic multi-reader, multi-writer distributed memory. In *Proceedings of the 13th International Conference on Principles of Distributed Systems* (Nîmes, France). Springer-Verlag, Berlin, 240–254. DOI: 10.1007/978-3-642-10877-8_20

[22] Rui Fan and Nancy Lynch. 2003. Efficient replication of large data objects. In *Distributed Computing*. Springer Berlin Heidelberg, Berlin, Heidelberg. 75–91.

[23] Antonio Fernández Anta, Chryssis Georgiou, Theophanis Hadjistasi, Nicolas Nicolaou, Efstathios Stavrakis, and Andria Trigeorgi. 2021. Fragmented objects: Boosting concurrency of shared large objects. In *Structural Information and Communication Complexity: 28th International Colloquium, SIROCCO 2021, Wrocław, Poland, June 28 – July 1, 2021, Proceedings*. Springer-Verlag, Berlin, 106–126. DOI: 10.1007/978-3-030-79527-6_7 Also at arXiv:2102.12786.

[24] A. Fernández Anta, Theophanis Hadjistasi, Nicolas Nicolaou, Alexandru Popa, and Alexander A. Schwarzmann. 2021. Tractable low-delay atomic memory. *Distributed Computing* 34, 1 (2021), 33–58.

[25] M. J. Fischer, N. Lynch, and M. Paterson. 1985. Impossibility of distributed consensus with one faulty process. *Journal of ACM* 32, 2 (1985), 374–382.

[26] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. 1985. Impossibility of distributed consensus with one faulty process. *J. ACM* 32, 2 (1985), 374–382. DOI: 10.1145/3149.214121

[27] E. Gafni and D. Malkhi. 2015. Elastic configuration maintenance via a parsimonious speculating snapshot solution. *International Symposium on Distributed Computing* 9363 (2015), 140–153. DOI: 10.1007/978-3-662-48653-5_10

[28] Chryssis Georgiou, Theophanis Hadjistasi, Nicolas Nicolaou, and Alexander A. Schwarzmann. 2022. Implementing three exchange read operations for distributed atomic storage. *Journal of Parallel and Distributed Computing* 163, 5 (2022), 97–113. DOI: 10.1016/j.jpdc.2022.01.024

[29] Chryssis Georgiou, Nicolas Nicolaou, Alexander C. Russell, and Alexander A. Shvartsman. 2011. Towards feasible implementations of low-latency multi-writer atomic registers. In *Proceedings of the 2011 IEEE International Symposium on Network Computing and Applications*. 75–82. DOI: 10.1109/NCA.2011.18

[30] C. Georgiou, N. Nicolaou, and A. A. Shvartsman. 2009. Fault-tolerant semifast implementations of atomic read/write registers. *Journal of Parallel and Distributed Computing* 69, 1 (2009), 62–79.

[31] Chryssis Georgiou, Nicolas Nicolaou, and Andria Trigeorgi. 2024. ARES II: Tracing the flaws of a (Storage) god. In *Proceedings of the 2024 43rd International Symposium on Reliable Distributed Systems*. 187–197. DOI: 10.1109/SRDS64841.2024.00027

[32] Chryssis Georgiou, Nicolas Nicolaou, and Andria Trigeorgi. 2024. Tracing the latencies of ARES: A DSM case study. In *Proceedings of the 2024 Workshop on Advanced Tools, Programming Languages, and PLatforms for Implementing and Evaluating Algorithms for Distributed Systems* (Nantes, France). Association for Computing Machinery, New York, NY, USA, 1–6. DOI: 10.1145/3663338.3665826

[33] Chryssis Georgiou, Nicolas C. Nicolaou, and Alexander A. Shvartsman. 2008. On the robustness of (Semi) fast quorum-based implementations of atomic shared memory. In *Proceedings of the Distributed Computing*. Gadi Taubenfeld (Ed.), Springer, Berlin, 289–304.

[34] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. 2003. The Google file system. *The Google File System* 53, 1 (2003), 79–81.

[35] Seth Gilbert and Nancy Lynch. 2002. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News* 33, 2 (2002), 51–59. DOI: 10.1145/564585.564601

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

[37] Theophanis Hadjistasi, Nicolas Nicolaou, and Alexander A. Schwarzmann. 2017. Oh-RAM! one and a half round atomic memory. In *Networked Systems*. Springer International Publishing, Cham, 117–132.

[38] HDFS [n. d.]. HDFS. Retrieved October 08, 2022 from https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html

[39] Maurice P. Herlihy and Jeannette M. Wing. 1990. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems* 12, 3 (1990), 463–492.

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

[41] Yordan Kalmukov, Milko Marinov, Tsvetelina Mladenova, and Irena Valova. 2021. Analysis and experimental study of HDFS performance. *TEM Journal* 10 (2021), 806–814. DOI: 10.18421/TEM102-38

[42] Lindsey Kuper and Peter Alvaro. 2019. Toward domain-specific solvers for distributed consistency. *Leibniz International Proceedings in Informatics, LIPIcs* 136, 10 (2019), 1–10. DOI: 10.4230/LIPIcs.SNAPL.2019.10

[43] Petr Kuznetsov and Andrei Tonkikh. 2020. Asynchronous reconfiguration with byzantine failures. *Leibniz International Proceedings in Informatics, LIPIcs* 179, 27 (2020), 1–17. arXiv:2005.13499 DOI: 10.4230/LIPIcs.DISC.2020.27

[44] Avinash Lakshman and Prashant Malik. 2010. Cassandra: A decentralized structured storage system. *ACM SIGOPS Operating Systems Review* 44, 2 (2010), 35–40. DOI: 10.1145/1773912.1773922

[45] Leslie Lamport. 1986. On interprocess communication - Part II: Algorithms. *Distributed Computing* 1, 2 (1986), 86–101. DOI: 10.1007/BF01786228

[46] Leslie Lamport. 1986. On interprocess communication, part I: Basic formalism. *Distributed Computing* 1, 2 (1986), 77–85.

[47] Bo Li, Mengdi Wang, Yongxin Zhao, Geguang Pu, Huibiao Zhu, and Fu Song. 2015. Modeling and verifying Google file system. In *Proceedings of the 2015 IEEE 16th International Symposium on High Assurance Systems Engineering*. 207–214. DOI: 10.1109/HASE.2015.38

[48] N. A. Lynch. 1996. *Distributed Algorithms*. Morgan Kaufmann Publishers.
[49] N. A. Lynch and A. A. Shvartsman. 1997. Robust emulation of shared memory using dynamic quorum-acknowledged broadcasts. *In Proceedings of IEEE 27th International Symposium on Fault Tolerant Computing*. 272–281.
[50] N. Nicolaou, A. F. Anta, and C. Georgiou. 2016. Cover-ability: Consistent versioning in asynchronous, fail-prone, message-passing environments, In *Proceedings of the 2016 IEEE 15th International Symposium on Network Computing and Applications*. 224–231. DOI: 10.1109/NCA.2016.7778622
[51] Nicolas Nicolaou, Viveck Cadambe, N. Prakash, Andria Trigeorgi, Kishori Konwar, Muriel Medard, and Nancy Lynch. 2022. ARES: Adaptive, reconfigurable, erasure coded, atomic storage. *ACM Transactions on Storage* 18, 4 (2022), 39 pages. DOI: 10.1145/3510613
[52] Diego Ongaro and John Ousterhout. 2014. In search of an understandable consensus algorithm. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*. USENIX Association, Berkeley, CA, USA, 305–320.
[53] Satadru Pan, Theano Stavrinos, Yunqiao Zhang, Atul Sikaria, Pavel Zakharov, Abhinav Sharma, Shiva Shankar P, Mike Shuey, Richard Wareing, Monika Gangapuram, et al. 2021. Facebook's tectonic filesystem: Efficiency from exascale. In *Proceedings of the 19th USENIX Conference on File and Storage Technologies*. USENIX Association, 217–231. Retrieved from https://www.usenix.org/conference/fast21/presentation/pan
[54] Michael O. Rabin. 1981. *Fingerprinting by Random Polynomials*. Center for Research in Computing Technology, Harvard University.
[55] Redis [n. d.]. Redis. Retrieved October 08, 2022 from https://redis.io
[56] Irving S. Reed and Gustave Solomon. 1960. Polynomial codes over certain finite fields. *Journal of The Society for Industrial and Applied Mathematics* 8, 2 (1960), 300–304.
[57] M. V. Steen and A. S. Tanenbaum. 2017. *Distributed Systems, 3rd ed.* distributed-systems.net.
[58] Andrew Tridgell and Paul Mackerras. 1996. The rsync algorithm. The Australian National University.
[59] P. Viotti and M. Vukolic. 2016. Consistency in non-transactional distributed storage systems. *ACM Computing Surveys* 49, 1 (2016), 1 – 34.
[60] Werner Vogels. 2008. Eventually consistent. *Queue* 6, 6 (2008), 14–19. DOI: 10.1145/1466443.1466448
[61] Marko Vukolic. 2012. *Quorum Systems: With Applications to Storage and Consensus*. Morgan and Claypool Publishers. DOI: 10.2200/S00402ED1V01Y201202DCT009
[62] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Carlos Maltzahn. 2006. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation* (Seattle, Washington). USENIX Association, Berkeley, CA, USA, 307–320. Retrieved from http://dl.acm.org/citation.cfm?id=1298455.1298485
[63] ZeroMQ [n. d.]. ZeroMQ. Retrieved May 04, 2023 from https://zeromq.org

Appendix

A Proofs

This appendix contains the proofs of lemmas omitted from the main text.

A.1 Proof of Lemma 8

Proof. Let τ be the local tag at the invocation of the operation π. According to Algorithm 3, both read and write operations perform a query phase, during which they collect the $\langle tag, value\rangle$ pairs from a majority of servers. During the query phase, π identifies the maximum pair $\langle \tau_{max}, v_{max}\rangle$ from the majority of servers. We consider two cases based on the condition before the propagation phase (line Algorithm 3:8 for a read and line Figure 2:10 in [50] for a write): (i) $\tau_{max} > \tau$, or (ii) $\tau_{max} \leq \tau$. In the case (i), irrespective whether π is a write or a read operation, it updates its local tag to $\tau_\pi = \tau_{max}$ (line Algorithm 3:11 and Figure 2:15 in [50]). In case (ii), if π is a read, it skips the propagation phase since the condition in line Algorithm 3:8 is not met, resulting in its local tag remaining the same, i.e., $\tau_\pi = \tau$, hence $\tau_\pi \geq \tau_{max}$. If π is a write, according to the algorithm (lines Figure 2:11–13 in [50]), $\tau_\pi = \langle \tau_{max}.ts + 1, p_2\rangle$, hence $\tau_\pi > \tau_{max}$. Hence, in any case $\tau_\pi \geq \tau_{max}$ and the claim of the lemma follows. □

A.2 Proof of Lemma 9

Proof. Let us assume that $v_{\pi_1}, v_{\pi_2} \in \mathcal{V}$, the values associated with τ_{π_1} and τ_{π_2}, respectively. According to Algorithm 3, both read and write operations perform a query phase, during which

they collect the $\langle tag, value\rangle$ pairs from a majority of servers. Let $\mathcal{S}_{q_1}$ and $\mathcal{S}_{q_2}$ the sets of servers that reply during the query phase of π_1 and π_2, respectively. Process p_1 discovers the max pair $\langle \tau_{max}, v_{max}\rangle$ from a server $s \in \mathcal{S}_{q_1}$. Given τ_{max} and τ to be the local tag value in p_1 at the invocation of π_1 we have two cases to consider: (*i*) either $\tau_{max} > \tau$, or (*ii*) $\tau_{max} \leq \tau$. We will investigate those cases separately. In addition to the previous notation, let τ' be the local tag at the invocation of π_2 and 2 be the state in ξ before the invocation of π_2. Since $\pi_1 \rightarrow \pi_2$, then π_1 completed before 2. As read and write operations have identical query phase and differ only in the propagation phase, for the rest of the proof the operations are type independent, unless when the type is specifically stated.

Case (i): In this case π_1 sets its local pair $\langle \tau_{\pi_1}, v_{\pi_1}\rangle$ to $\langle \tau_{max}, v_{max}\rangle$ (if π_1 is a write is converted to a read by [50]) and propagates this pair to a majority of servers, say $\mathcal{S}\pi_1$, before the completion of π_1 and thus before state 2 (lines Algorithm 3:9–11 and Figure 2:15–17 in [50]). When operation π_2 is invoked in discovers the maximum of all the pairs received from the servers $\in \mathcal{S}_{q_2}$ during the query phase; denoted as $\langle \tau'_{max}, v'_{max}\rangle$. Since both $\mathcal{S}\pi_1$ and $\mathcal{S}q_2$ contain at least the majority of servers, then there exists at least one server $s \in \mathcal{S}_{\pi_1} \cap \mathcal{S}_{q_2}$ that received the pair $\langle \tau_{\pi_1}, v_{\pi_1}\rangle$ from π_1 before replying to the query phase of π_2. According to the algorithm, a server updates its local copy only if a higher tag is received (cf. line Algorithm 3:19). Thus, the tag is increasing monotonically at each server, and hence $s \in \mathcal{S}_{q_2}$ replies to π_2 with a tag $\tau_s \geq \tau_{\pi_1}$. So p_2 will discover a max tag $\tau'_{max} \geq \tau_{\pi_1}$ from the replies of the servers in $\mathcal{S}q_2$. By Lemma 8, the local tag τ_{π_2} is at least as large as τ'_{max}, i.e., $\tau_{\pi_2} \geq \tau'_{max}$. Thus, combining the previous two results, we conclude that $\tau_{\pi_2} \geq \tau'_{max} \geq \tau_{\pi_1}$.

Case (ii): In this case π_1 takes different steps depending whether it is a write or a read. If π_1 is a write, it increments the maximum timestamp τ_{max} discovered during the query phase, generating $\tau_{\pi_1} = \langle \tau_{max}.ts+1, p_1\rangle$ and propagating it along with the value it wishes to write (lines Figure 2:11–13 in [50]). Since, π_1 sends the pair to a majority of servers before completing, then with similar arguments as in Case (i), we can show that there is a server s that replies both during the propagation phase of π_1 and during the query phase of π_2. Hence, π_2 will discover a $\tau_{max} \geq \tau_{\pi_1}$, and by Lemma 8, will set $\tau_{\pi_2} \geq \tau_{max} \geq \tau_{\pi_1}$.

If π_1 is a read, then in Case (ii) the propagation phase is omitted since the max tag received from the servers in $\mathcal{S}_{q_1}$ is $\tau_{max} \leq \tau$ (cf. line Algorithm 3:8). Since, however process p_1 is initialized with the tag $\langle 0, p_1\rangle$, then either $\tau = \langle 0, p_1\rangle$, in which case no write operation has been completed before π_1, or p_1 invoked an operation $\pi' \rightarrow \pi_1$, in which it discovered $\tau' \geq \tau_{max}$. Let us assume w.l.o.g. that π' is the first operation in ξ, in which p_1 discovers τ' during the query phase of π'. If that is the case, then π' either propagates $\tau = \tau'$ if π' is a read, or $\tau = \langle \tau.ts + 1, p_1\rangle$ if π' is a write to a majority of servers, say $\mathcal{S}_{\pi'}$. So, during π_1, there is a server in $\mathcal{S}\pi' \cap \mathcal{S}q_1$, such that it replies to both the propagation phase to π' and the query phase of π_1. Since the tags in the servers are monotonically increasing then s will reply to π_1 with a tag $\tau_s \geq \tau$. According to Case (ii), however, p_1 observed a maximum tag $\tau_{max} \leq \tau$. Hence, s replied with the maximum tag and it must be the case that $\tau_{max} = \tau = \tau_{\pi_1}$. Finally, since $\pi' \rightarrow \pi_1$, then by the transitivity of $\rightarrow$, it must be the case that $\pi' \rightarrow \pi_2$ as well. Therefore, there exists a server $s' \in \mathcal{S}_{q_2} \cap \mathcal{S}_{\pi'}$, that replies to the query phase of π_2 with a tag $\tau_{s'} \geq \tau$ and hence $\tau_{s'} \geq \tau_{\pi_1}$. Thus, p_2 will discover a maximum tag $\tau'_{max} \geq \tau_{s'} \geq \tau_{\pi_1}$ during the query phase of π_2, and therefore by Lemma 8 it follows that $\tau_{\pi_2} \geq \tau'_{max} \geq \tau_{\pi_1}$.

Therefore, regardless of whether π_2 is a read or write operation, we have $\tau_{\pi_1} \leq \tau_{\pi_2}$, with strict inequality when π_2 is a successful write operation (and thus it increments the maximum timestamp discovered). This concludes the proof. □

A.3 Proof of Lemma 11

Proof. DSMM relies on the correctness of the optimized CoABD algorithm. According to Theorem 10, the optimized version of the CoABD algorithm is indeed capable of implementing R/W coverable objects, and hence block objects. □

A.4 Proof of Lemma 12

Proof. It is easy to see that if $\pi = \text{update}(b, D)_{f,p}$ is successful, then all the dsmm-write operations invoked within π, including dsmm-write$(val(b))_{b,p}$, are successful. It remains to show that π can only be unsuccessful whenever dsmm-write$(val(b))_{b,p}$ is unsuccessful. In the case where D contains a single chunk, i.e., $D = \langle D_0 \rangle$ then π invokes a single dsmm-write$(val(b))_{b,p}$ with $val(b).data = D_0$. If the cvr-write invoked in that operation is unsuccessful then π is also unsuccessful. In the case where $k > 0$, π invokes $k-1$ create operations with new block identifiers (due to the incremented block counter bc). The cvr-write operation on every such block will be successful as (i) the block id $\langle f, p, bc \rangle$ (and thus the block) can only be generated by process p, and (ii) the block is not yet inserted in the link-list. So no other write operation will attempt to cvr-write the same block concurrently. So the only operation that may fail in this case as well, is the dsmm-write$(val(b))_{b,p}$ as b was a part of the list and may be accessed concurrently by a writer $q \neq p$. □

A.5 Proof of Lemma 13

Proof. According to our protocol it is clear that a block with id b appears in the list of f only if that is created and written during an update$_{f,*}$ operation. Also, if the block is created by an update that precedes ρ, then no other block in the list will point to b, ρ will not invoke a sm-read$_b$ operation for b, and thus $b \notin \mathcal{L}$.

So it remains to examine the case where ρ may obtain b from an unsuccessful update$_{f,*}$. Let us assume by contradiction that a read operation may return a block b for a file f created by an unsuccessful update. Let $b \in \langle b_1, \ldots, b_n \rangle$, the list of blocks that the update needs to write on the DSM. In particular, the operation will create all the blocks $\langle b_2, \ldots, b_n \rangle$ and attempt to write block b_1. There are two cases to consider: (*i*) either b is equal to b_1, or (*ii*) b is in $\langle b_2, \ldots, b_n \rangle$.

If case (*i*) is true, then p will invoke a sm-write$(val(b))_b$ as b is the block that is updated. However, since we assume that the update was not successful, then by Lemma 12, the write operation is not successful. Thus, according to the coverable DSM, b was never written and this contradicts the assumption that p obtain $b \in \mathcal{L}$.

If case (*ii*) holds, then b was created by p (an operation that cannot fail). However, since the update is not successful, then b_1 was not written in the list. It is also true that there is no link path leading to b since the only path was $b_1 \rightarrow b_2 \rightarrow \ldots \rightarrow b$. So, during the traversal of the blocks, the read operation will not see b_1 and thus will never reach and obtain b, contradicting again our initial assumption. □

A.6 Proof of Lemma 14

Proof. Assume by contradiction that there exist some $b_i \in \mathcal{L}$, s.t. $val(b_i).ptr \neq b_{i+1}$ (or $val(b_g).prt \neq b_1$). By Lemma 13, a block b_i may appear in the list returned by a read operation only if it was created by a successful update operation, say w.l.o.g. $\pi = \text{update}(b, D)_{f,*}$. Let $D = \langle D_0, \ldots, D_k \rangle$ and $\mathcal{B} = \langle b_1, \ldots, b_k \rangle$ be the set of $k-1$ blocks created in π, with $b_i \in \mathcal{B}$. By the design of the algorithm we create 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 $b \rightarrow b_1 \rightarrow \ldots \rightarrow b_i$ 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 completed, and thus when b is written successfully all the blocks in the path are inserted successfully in f. So, if now b_i is different than b_k by the construction of the update then both b_i and b_{i+1} are in the list with $val(b_i).ptr = b_{i+1}$ contradicting our assumption.

If now $b_i = b_k$, then $val(b_i).ptr = b'$. 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 case (*i*), by Lemma 12, the update operation that created b was successful and thus b' must be created and inserted in f as well. In case (*ii*) it follows that b is the last inserted block of an update and is assigned to point to b'. With a simple induction one may show that the update operation that created b' must precede the update that created 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 wrote b and hence only the blocks in $\mathcal{B}$ were inserted between b and b' at the response of π. 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 our proof. □

A.7 Proof of Lemma 15

Proof. We begin by showing that if the two lists share an arbitary block b_i then the version of $b_i \in \mathcal{L}_1$ is smaller than or equal to the version of the same $b_i \in \mathcal{L}_2$. Let τ_{ρ_1} be the maximum tag (version) of $b_i \in \mathcal{L}_1$ as observed during ρ_1 and τ_{ρ_2} be the maximum tag (version) of $b_i \in \mathcal{L}_2$ as observed by ρ_2. Since each block is a coverable object, and since $\rho_1 \to \rho_2$ then by Lemma 9 it follows that if both ρ_1 and ρ_2 retrieve the same block b_i, then $\tau_{\rho_1} \leq \tau_{\rho_2}$. So for the rest of the proof it remains to show the first part of the lemma, i.e., that $\forall b_i \in \mathcal{L}_1$, then $b_i \in \mathcal{L}_2$.

Note that, by Algorithm 2, the only operation that may alter the pointers of the blocks, thus affecting the block lists is the fm-update$_f$ action. Let T_1 denote a set of points in the execution ξ occurring after the completion of ρ_1 and before the invocation of ρ_2. Consider the set $Upd ates$, which consists of all fm-update$_f$ operations initiated at any execution point in T_1. Let us assume by contradiction that $\exists b_i \in \mathcal{L}_1$, s.t. $b_i \notin \mathcal{L}_2$. There are three cases to consider based on the length of the two lists: (*i*) $|\mathcal{L}_1| = |\mathcal{L}_2|$, (*ii*) $|\mathcal{L}_1| > |\mathcal{L}_2|$, (*iii*) $|\mathcal{L}_1| < |\mathcal{L}_2|$.

Case (*i*) holds when no new blocks are added at any execution point in T_1. This can occurred if any fm-update$_f \in Upd ates$ either: (*a*) is successful (i.e., the dsmm-write is not converted to a read) and does not add any new block, or (*b*) is unsuccessful. Case (*a*) holds if fm-update(b, D) is called for a block b and $|D| = 1$, i.e., the data fit into the block b. In this case, the algorithm does not generate new blocks (Algorithm 2:10–14), and the pointer of b remains the same following the completion of the fm-update operation. Since according to case (a) no new blocks are added, and since all updates happen after the completion of ρ_1 then the pointers of any block $b \in \mathcal{L}_1$ remain unchanged until the invocation of ρ_2. Thus, since both ρ_1 and ρ_2 access the same file f, they will obtain the the same genesis block b_g which in both cases points to the same block b_1. Both operations will request b_1, and since the pointer remains unchanged then both reads will request the same block after b_1. With a simple induction we can show that ρ_2 will request the exact same blocks as ρ_1, resulting to $\mathcal{L}_1 = \mathcal{L}_2$. If the second subcase (*b*) is true, according to Lemma 13 the read operation ρ_2 will never obtain blocks from unsuccessful fm-update$_f$ operations in $Upd ates$. Thus, the newly created blocks from any such update operation will not be added to $\mathcal{L}_2$ and ρ_2 will obtain the same blocks as ρ_1, and $\mathcal{L}_1 = \mathcal{L}_2$ in this case as well. Therefore for any block $b_i \in \mathcal{L}_1$, then $b_i \in \mathcal{L}_2$ as well contradicting our initial assumption.

Case (*ii*) arises when any successful fm-update$(b, D)_f$ with $|D| > 1$ in $Upd ates$ adds new blocks (lines Algorithm 2:15–19). Let $D = \{D_0, D_1, \ldots, D_k\}$, for $k \geq 1$, where D_0 the data to be written in block b, and each D_j the data to be written in a separate block. The fm-update$(b, D)_f$ operation

first creates a block b_k for the data D_k using the dsmm-create function. The pointer of the newly created block points to the pointer of b, i.e., $b_k.ptr = b.ptr$ (line Algorithm 2:44). This way b_k points to the previously next pointer of b, denoted as b'. Then the algorithm works backward through the remaining data blocks $(D_{k-1}, ...D_2, D_1)$, creating a new block b_j for each D_j, and pointing $b_j.ptr = b_{j+1}$, for $1 \leq j \leq k-1$ (line Algorithm 2:42). Finally, the modified block b points to the last newly created block b_1.

Consequently, every block b_j created during an update in the set $Upd ates$ cannot be in $\mathcal{L}_1$ since it is added after the completion of ρ_1. Let's proceed with examining the behavior of the first update operation, say π, on block $b_i \in \mathcal{L}_1$ after the completion of ρ_1. There are three cases to consider: (a) $b_i = b$, (b) $b_i = b'$, and (c) b_i is any block before b or after b'. If $b_i = b$, this implies that b_i is the block being modified during the update operation, and also that its next pointer changed to point to the last created block. Since $b_i \in \mathcal{L}_1$, by Lemma 13, this block was created by a successful fm-update operation, that preceded or is concurrent to ρ_1. If now $b_i = b'$, b_i remains unchanged, since it is not affected by π. Since b was pointing to b_i at the completion of ρ_1 then b_i was either (i) created during the update operation that also created b, or (ii) was created before b, or (iii) was created after b. In case (i), by Lemma 12, the update operation that created b was successful and thus b_i must be created during that update as well. In case (ii) it follows that b is the first created block of an update and is assigned to point to b_i, the next existing block. In case (iii) it follows that b_i is the last created block of an update and b is assigned to point to b_i. In case (c), if b_i is any other block before b or after b', both the value and the pointer of b_i remain unchanged, since it is not affected by π, as well as the block that points to it. Thus, for any $b_i \in \mathcal{L}_1$, then there exists a path from b_g to b_i following the completion of π. This is true for any $\pi \in Upd ates$, and therefore, since ρ_2, reads b_g, then it must be the case that $\mathcal{L}_2 \geq \mathcal{L}_1$. This makes the case impossible.

Case (iii) may hold if ρ_1 returns more blocks than ρ_2, leading to $|\mathcal{L}_1| > |\mathcal{L}_2|$. According to the algorithm's construction (see Algorithm 2), block deletion is handled as an update operation. In particular, when the data of a block b need to be deleted then fm-update$(b, \perp)$ replaces the content of b with empty data. As deletions are not applied, it remains to examine how block pointers are updated when a successful fm-update$_f$ operation creates new blocks. Let's assume by contradiction that updating a block pointer to a subsequent block is indeed allowed. More formally if $L = b \rightarrow b' \rightarrow b''$ is a sequence of pointers from b to b'' we need to show whether is possible to construct a sequence where $L' = b' \rightarrow ... \rightarrow b''$, s.t. b' does not appear in L'. New blocks can be created during a successful fm-update$(b, D)_f \in Upd ates$ when $|D| > 1$, i.e., the data $(D = \langle D_0, \ldots, D_k \rangle)$ do not fit into the block b. In this case, the Algorithm 2 generates new blocks (lines Algorithm 2:15–19). As seen in case (ii), during a successful fm-update$(b, D)_f$ in $Upd ates$ when $|D > 1|$, new blocks are created, and the modified block b points to the first newly created block, and b_k points to the previously next pointer of b, b'. Thus, if $b_1 \rightarrow b2 \rightarrow ... \rightarrow b_k$ the newly created blocks, the list after the completion of the update operation will be $L_{new} = b \rightarrow b_1 \ldots b_k \rightarrow b' \rightarrow b''$ if the write of b is successful or $b \rightarrow b' \rightarrow b''$ otherwise. In any case there is a path from b to b' following the update operation, contradicting our assumption that the new list will not contain b'. Therefore, the case where $|\mathcal{L}_1| > |\mathcal{L}_2|$ is not possible. □

A.8 Proof of Lemma 17

Proof. This lemma follows from the fact that CoAres uses a condition before the propagation phase in line Algorithm 4:19. 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. □

A.9 Proof of Lemma 18

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 $ver_2 = \langle ver'.ts + 1, \omega_2 \rangle > ver_1$, 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 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 [51], 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. □

A.10 Proof of Lemma 19

Proof. Every tag is generated by extending the tag retrieved by a get-data operation starting from the initial tag (lines Algorithm 4:20–21). 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. □

A.11 Proof of Lemma 20

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 $\omega_1 \rightarrow_\xi \omega_2$ 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 Algorithm 4:21). 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 $\tau_k < \tau_\ell'$. □

A.12 Proof of Lemma 22

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'*. □

A.13 Proof of Lemma 23

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 Algorithm 6:13, 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* (Algorithm 6:14–16).

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 Algorithm 6:6–9). 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 22, 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. □

A.14 Proof of Lemma 28

Proof. Let $cseq_\omega$ be the last configuration sequence returned by the read-config action at ω (Algorithm 4:28), and $cseq_\rho$ the configuration sequence returned by the first read-config action at ρ (see Algorithm 2:8 in [51]). 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. □

A.15 Proof of Lemma 29

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 27, 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 $\mathcal{B} = \langle b_1, \ldots, b_k \rangle$ 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 $b \rightarrow b_1 \rightarrow \ldots \rightarrow b_i$ 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 28 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 18 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 $\{b'_1, \ldots, b'_k\}$, 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 27, 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. □

Received 17 January 2025; revised 29 September 2025; accepted 21 December 2025