PhD Thesis Proposal
Robust and Strongly Consistent Distributed Storage Systems

Andria Trigeorgi
Department of Computer Science
University of Cyprus, Nicosia, Cyprus

May 2023

Abstract

EMC Digital Universe (https://www.cycloneinteractive.com/our-work/emc-digital-universe/) estimates that the data in the digital universe doubles in size every two years. According to the International Data Corporation (https://www.idc.com/), the amount of data in the world was 26 Zettabytes (ZB) in 2017, and 64.2 ZB in 2020. The growth in 2020 was higher than expected because of the COVID-19 pandemic, as more people worked, learned, and entertained themselves online from home. By 2025, IDC says the amount of data will grow to 175 ZB. As more and more data are generated at a high speed, the challenges posed by handling Big Data gain an increasing importance. To cope with this explosion of data, large-scale infrastructures are needed. A Distributed Storage System (DSS) is an infrastructure that can split data across multiple servers.

The design and implementation of DSSs, that can handle such vast amount of data, involves many challenges due to the fact that the users and storage nodes are physically dispersed. Thus, the autonomous components (users and storage nodes) must decide how to modify the data in a way that ensures consistency while attempting to handle as many read/write requests concurrently as possible. The atomic consistency semantic (linearizability) simplifies application development and facilitates concurrency reasoning, but it is challenging to implement effectively in a distributed setting without sacrificing performance. Due to this difficulty, commercial solutions avoid the use of strong consistency. In research institutions, a plethora of algorithmic solutions along with proven correctness guarantees have been proposed to provide Atomic Distributed Shared Memory (ADSM) in a message passing system. ADSM provides the illusion of a sequential memory space despite asynchrony, network perturbations, and device failures. Therefore, it is important to provide the necessary abstractions required for high performance at large scale in atomic algorithms, otherwise the concurrency reasoning provided by atomicity will be constrained by poor data scalability. This proposal aims at demonstrating that it is possible to build a Robust and Strongly Consistent DSS with provable guarantees while providing highly concurrent access to its users at large scale.

Contents

Chapter 1

Introduction

The exponential growth of data leads to the continued improvements in the efficiency of storage technology. A Distributed Storage System (DSS) [1, 2] 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 [3, 4] 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 [5] generalize Lamport's notion of atomic registers by defining the notion of *linearizability*.

According to the CAP theorem [6], 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. As a result, they either often provide weaker consistency guarantees (like weak [1] and eventual consistency [7, 8]), devise strategies with limited concurrency (e.g., files in HDFS [9] restrict one writer at a time), or rely on centralized solutions to provide strong consistency.

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 [10] 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.

For more than two decades, a series of works, e.g., [11, 12, 13, 14, 15, 16, 17, 18], 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. Currently, such 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. Given

the limitations of existing atomic algorithms and new challenges that arise in the context of exponentially growing data sizes, this proposal aims with provable guarantees, demonstrating that it is possible to build a Robust and Strongly Consistent DSS while providing highly concurrent access to its users at large scale.

1.1 State-of-the-Art

We overview related work, focused to the topics around the problems considered in this proposal. A more comprehensive literature review is given in [19].

Linearizability/Atomic Consistency. Linearizability [5, 1] is a strong consistency model. A shared-memory system supports linearizability if all processes see all shared accesses in the same order. Accesses are ordered according to a global logical timestamp (e.g., an integer value). If the timestamp of an operation $op1$ is less than the timestamp of an operation $op2$, i.e., $t_{op1} < t_{op2}$, then the $op1$ must occur before $op2$ in the sequence seen by all processes. This model provides the illusion that operations happen in a sequential order. Linearizability is formally defined in Definition 2 of Section 2.1, and is the main consistency model we consider in this proposal.

Weak consistency models. Relaxed [20] and eventual [7, 8] consistency models are weaker consistency models than linearizability. These models trade off stronger consistency guarantees for higher availability, fault tolerance, and performance. Relaxed consistency introduces relaxations based on program order and write atomicity, which enable many of the optimizations that are constrained by sequential consistency. Eventual consistency guarantees that if no new updates are made to the object, eventually all accesses will return the last updated value.

Replication-based Atomic Storage. There are many prior works [11, 14, 12, 21, 15, 22, 13] implementing algorithms for atomic (linearizable) shared memory emulation. Attiya, Bar-Noy and Dolev [11] 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 is an integer number that lets the system keep track of old values. 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 [12], who present an emulation of MWMR atomic registers in message-passing systems. Also, they generalized majorities into *quorums*. 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 four-exchange 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 two exchanges ensure 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, the notion of *coverability* was introduced in [23], which is a consistency guarantee that extends linearizability and concerns versioned objects (cf. Definition 4 in Section 2.1). In that same work an implementation of a coverable (versioned) object was presented, which we call CoABD. 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 way, writes are prevented from overwriting a newer version of the object.

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 [17] rely on consensus [24]. 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 [18] was the first asynchronous implementation of reconfigurable storage. RAMBO uses consensus to choose only one successor for every configuration, in DynaStore configurations can have multiple successors, while *lattice agreement* [24] or Speculating Snapshot algorithm (*SpSn*) [25] can ensure that configurations are ordered. i.e., for two configurations, the changes realized in one of them is 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 [26] and SpSnStore [25]. Ares [16] 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 4.1, called DAPs (Data Access Primitives). ARES shares similarities with consenus 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 4.1.

Distributed Storage Systems. Table 1.1 presents a comparison of the main characteristics of storage systems that we will discuss below, and that can be summarized as follows:

- *Data Scalability*: the system can manage the growing data requirements, by expanding the current infrastructure with more resources or disks.
- *Data Access Concurrency*: how the system manages conflicting accesses to data.
- *Consistency guarantees*: the consistency model that the system follows.
- *Versioning*: whether the system records versions of changed data.
- *Data Striping*: whether the system divide the data across the set of disks.
- *Reconfiguration*: whether the system change the configuration of the system as it executes.

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 [27]. These failures may include crashes, hardware failures, or other types of faults. Thus the systems is designed to continue functioning even in the presence of one or more withstand failures.

The first-generation distributed file systems of Google and Facebook, that is, GFS [28] and HDFS [9] respectively, were not scalable enough because their centralized components did not allow them to support huge amount 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 [29] and Facebook with TECTONIC [30]. 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.

Other major concerns of DSSs are data availability and fault-tolerance. Some mechanisms to increase the availability of data and ensure fault-tolerance are replication mechanisms, versioning, snapshots, etc. In addition all the above DSSs (except CASSANDRA [31] and REDIS [32]) use data striping to manage and optimize concurrent access to the data. Furthermore, BLOBSEER [33] continues editing in a new version without blocking other clients who continue to use the current version transparently. Thus, saving data as BLOB (Binary Large Object), that is a long sequence of bytes, combined with a data striping mechanism can greatly improve the performance of such a system and allow it to handle larger files.

CASSANDRA and HDFS are both projects of Apache. Unlike HDFS, CASSANDRA does not have a master-slave architecture; it follows a masterless architecture. HDFS, like the other systems mentioned, can handle large files, using the data striping technique. On the other hand, CASSANDRA can work with multiple small records (structured data, record-level indexing). If one wants to store large unstructured files in CASSANDRA, they first need to split them into multiple parts. CASSANDRA and REDIS are two real-time databases. However, REDIS has a significantly faster response time than CASSANDRA, since it saves a lot of data in-memory rather that storing it to disk, as CASSANDRA. On the other hand, CASSANDRA can guarantee considerably better fault-tolerance than REDIS due to its architecture.

Regarding consistency guarantees, HDFS and BLOBSEER provide linearizability consistency. HDFS achieves linearizability by using centralized machine for metadata. Unlike HDFS, BLOBSEER does not centralize metadata on a single machine, but it uses a centralized version manager. Thus, linearizability is easy to achieve. 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, it can view theses 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.

DROPBOX is not even a far relative of the other file systems. It just synchronizes the files between local storage devices and the cloud for reasons like backup, share and access one's files from anywhere. On the other hand, to use the other systems as cloud storage, it is needed to build user-friendly and efficient systems as DROPBOX. But they will neither be so user-friendly nor efficient as DROPBOX which guarantees a weaker consistency. There are a limited number of papers [34, 35, 36, 37, 33, 30] 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.

Conclusion. In this section we have discussed about ADSMs and DSSs, in an attempt to showcase the current state-of-the-art. On the one hand, the advantage of ADSMs is that they come with *provable guarantees* on atomicity and reconfiguration correctness. However, ADSMs require to transmit the entire object over the network per read and write operation. Moreover, if two concurrent write operations

Table 1.1: Comparative table of storage systems.

System	Data scalability	Data access Concurr.	Consist. guarantees	Versioning	Data Striping	Non-blocking Reconfig.
GFS [28]	YES	concurrent appends	relaxed	YES	YES	YES (short down-time)
COLOSSUS [29]	YES	concurrent appends	relaxed	YES	YES	YES
HDFS [9]	YES	files restrict one writer at a time	atomic central-ized	NO	YES	YES
CASSANDRA [31]	YES	YES	tunable (default= eventual)	YES	NO	NO
DROPBOX [38]	YES	creates con-flicting copies	eventual	YES	YES	N/A
REDIS [32]	YES	YES	eventual	YES	NO	NO
BLOBSEER [33]	YES	YES	atomic central-ized	YES	YES	YES
TECTONIC [30]	YES	files restrict one writer at a time	Read-your-writes	YES	YES	YES

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. 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. To the best of our knowledge, currently there is no DSS that can provide provable atomic consistency guarantees in a decentralized environment in the absence of failure detection and coordination; and this is our motivation: to create a DSS that will provide and provably guarantee these characteristics.

1.2 Objectives, Methodology, and Contributions

In this section, we will outline the objectives, methodology, and the already-made contributions of this proposal. By using our method, we intend to advance the field of DSSs by bringing fresh perspectives and information to the table and by offering a structure for further study and advancement.

Objectives

Given the limitations of existing memory emulations and the new challenges arising from the constantly growing data sizes, the ultimate objective of this work is to development and implement a Robust and Strongly Consistent DSS. The DSS should provide provable atomic consistency guarantees, allows highly concurrent access to its users and facilitate data sharing at large scale under an asynchronous, crash-prone, distributed and even dynamic setting. In order to achieve the primary objective, this proposal seeks to accomplish a series of sub-objectives. The final sub-objective (#7) is left for future work, as it is beyond the scope of the thesis work.

1. To study and formally define principles for designing a DSS that is capable of efficiently handling large shared atomic objects.
2. To develop a DSS that includes a reconfiguration feature allowing the seamless addition and removal of servers in the system.
3. To convert the design principles into an algorithmic implementation of a DSS.
4. To conduct experimental evaluation tests each of which measures specific design principle of the DSS.
5. To identify and optimize any performance bottlenecks that are discovered during the system evaluation.
6. To develop a user-friendly interface that enables non-expert users to easily access and utilize our storage system.

7. To develop a fully-functioning DSS with the developed user interface and security guarantees. and deploy this DSS on a set of networked devices and evaluate it in real-life situations.

Methodology

There are many characteristics that can be adopted to design the best of a DSS. In particular, the system should manage the growing data requirements by expanding the current infrastructure with more resources or disks (Data Scalability). Another important characteristic is how the system manages conflicting accesses to data (Data Access Concurrency). The Consistency guarantees, i.e., the consistency model that the system follows, is of big interest in this work, since we need to provide strong consistency guarantees. Also, the support of versioning unlocks the potential to access data in a highly concurrent manner, resulting to better performance. This approach is combined with data striping which divides the data across the set of disks, so that concurrent accesses are distributed at large scale among disks. Finally, it is important that the system can support reconfiguration mechanisms to be able to change the configuration of the system without blocking ongoing access operations.

For the purpose of accomplishing a Robust Distributed Storage System, with strong consistency guarantees, the proposed research includes the following stages (covering Objectives 1 to 6):

1. From existing emulations of distributed shared memory ([11, 23, 21, 39]), we want to implement an optimized version of the most suitable for our purpose one, in order to establish *linearizable consistency* to a unit of storage, we call a *block*. (Objective 1)

2. Survey Data Fragmentation Strategies, focusing on block strategies. This would lead to our own fragmentation strategy, aiming to reduce the communication cost of write/read operations by splitting data into smaller atomic data objects (blocks), while enabling concurrent access to these blocks. (Objective 1)

3. Integrate the different algorithmic modules and strategies, envisioned via a system architecture (cf. Section 3.3). This would entail our design and implementation framework. Based on that, we aim to introduce new consistency guarantees, that characterize the consistency of the whole object, which is composed by smaller atomic objects (blocks). (Objective 3)

4. Next, dynamic solutions need to be explored, such as the implementation of ARES [16] presented in Section 4.1. ARES is a reconfigurable, erasure-coded, and atomic consistent storage algorithm. ARES promises that can mask host failures or switching between storage algorithms without service interruptions. (Objective 2)

5. ARES divides the object into encoded fragments and deliver each fragment to one server. The object can be recovered from a subset of the fragments.

However, operations are still applied on the entire object. Thus, we can evaluate the performance of the system by combining our fragmentation approach with *ARES*. (Objectives 1, 2, 3)

6. It would be interesting to evaluate the performance of our implementation against open-source and commercial solutions. (Objective 4)

7. Deploy and evaluate the system in network testbeds. An emulation testbed such as Emulab [40] will be used for developing and debugging the components of our system. However, an overlay planetary-scale testbet such as AWS [41] will help us examine the performance of our system in highly-adverse, uncontrolled, real-time environments. Also, Fed4FIRE+ is the largest federation of testbeds in Europe [42]. (Objective 4)

8. Identify performance bottlenecks using distributed tracing and devise algorithmic solutions on how to overcome the performance barriers on our implementation . (Objective 5)

9. Design easy-to-use user interfaces to facilitate the use of our storage system by users that are not necessarily highly technology-trained. This platform will integrate to DSS for monitoring, managing, and accessing the shared memory. (Objective 6)

Stage of Research. Stages 1 to 5 and part of Stage 6 have been completed. We conducted an extensive evaluation on all referred testbeds in Stage 7; however, the undeveloped stages also require evaluation. We are currently working on Stages 8 and 9.

Contributions and Document Organization

The structure of the work is divided into four main technical parts, which are outlined in detail below. The first part is dedicated to covering Stages 1 through 3, while the second part focuses on Stages 4 and 6. Stage 5 is the focus of the third part, and the fourth part is dedicated to covering the remaining work related to Stages 8 and 9 . All the parts perform Stage 7. As already mentioned, Objective 7) is regarded as future work. **Chapter 2** provides the model of computation, necessary definitions and the experimental evaluation setup.

The first part (Chapter 3) introduces the core contribution of this work: CoBFS (stands for Coverable Boosting Fragmentation Strategy), a distributed data storage service that can handle increasing data sizes and addresses several of the challenges that were discussed above. Section 3.1 defines 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 lists of block objects. Section 3.2 examines the consistency properties when allowing R/W operations on individual blocks of the fragmented object, in order to enable concurrent modifications. Assuming that each block is coverable [23], we define the precise consistency that the fragmented object provides, termed *fragmented coverable linearizability* or *fragmented coverability*, for short. Section 3.3 provides an algorithm that

implements coverable fragmented objects. Then, we use it to build a prototype implementation of a distributed file system, called CoBFS, by representing each file as a linked-list of coverable block objects. CoBFS preserves the validity of the fragmented object and satisfies fragmented coverability. CoBFS adopts a modular architecture, separating the object fragmentation process from the distributed shared memory module (DSMM), which allows to follow different fragmentation strategies and shared memory implementations. From the survey of existing emulations, we choose the most suitable for our purpose one, CoABD [23], the coverable version of the well-known ABD [11, 43] (cf. Section 1.1), in order to establish linearizable consistency to the DSMM. Thus in the initial implementation, we integrate the static emulation CoABD with the DSMM module in CoBFS. In particular, we propose a static framework that: (i) supports versioned objects, and (ii) is suitable for large objects (such as files). Section 3.4 presents an experimental development and deployment of CoBFS on the Emulab testbed [40]. Results are presented, comparing our proposed algorithm to its non-fragmented counterpart. Results suggest that a fragmented object implementation boosts concurrency while reducing the latency of operations.

The second part (Chapter 4) focuses on the fault tolerance and efficiency of the shared memory module (DSMM) of CoBFS. Until this point, CoBFS distributes data over a fixed set of replica servers. We need to introduce dynamic reconfiguration, which enables the system to dynamically replace failed nodes with healthy ones without interrupting the system operation. In addition, we aim to reduce the operation latency and storage space needs when under heavy access concurrency. Thus, ARES (cf. Section 4.1) seems to be the perfect solution to enable the dynamic reconfiguration, fault tolerance, and reduce the storage space and execution time. A first evidence to support this claim is to implement and evaluate ARES algorithm on an emulation testbed, Emulab. Thus, Section 4.2 describes the implementation of ARES, while Section 4.3 presents an experimental evaluation of it. It is worth mentioning that ARES may use any DSM algorithm at its core, providing the flexibility to adjust its performance based on the application demands. A next question that we try to answer is how such an algorithm may compare to commercially used solutions. That is, no evidence exists to date to examine what are the gains from commercial solutions to adopt less than intuitive guarantees. Thus, we set to put ARES and chosen open-source, commercial solutions (CASSANDRA [31] and Redis [32]) in a head-to-head comparison in order to answer the question: *Is it worth to trade consistency for performance?* We deployed experiments in real testbeds (supported by Fed4FIRE+ project), distributed in the European Union (EU) and the USA. Such deployment helped us obtain real-condition results and evaluate the algorithms over cross-Atlantic setups. To the best of our knowledge, this is the first work to conduct such comparison. Our experimental study, in Section 4.4, focuses on measuring the average operation latency (communication and computation), over three performance test categories. Our experimentation results suggest, perhaps surprisingly, that ARES has a similar or sometimes better performance than the competition.

The third part (Chapter 5) integrates CoBFS with ARES. The individual block-processing design of CoBFS enables heavy concurrent accesses to data. Also due to its decentralized components, it provides high data throughput. However, the efficiency of the storage is crucial for the performance of CoBFS. To show the benefits of the fragmentation approach, we integrate different distributed shared object algorithms into CoBFS. To integrate a shared memory emulation in CoBFS, it must support versioning, i.e., we have to covert its READ/WRITE Registers to concurrent versioned registers that use coverability. We implement various versions of our storages, i.e., with and without the fragmentation technique or with and without Erasure Code or with and without reconfiguration. In one of these, we integrate ARES in the core of CoBFS, which allows dynamic participation and flexibility on the redundancy mechanism it deploys (replication and erasure coding). In Section 5.1, 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. Then we are ready to adopt the idea of fragmentation as presented in CoBFS, to obtain CoARESF, which enables CoARES to handle *large* shared data objects and increased data access concurrency (Section 5.3). The correctness of CoARESF is rigorously proven. Section 5.2 proposes a reduction of the operational latency of the read/write operations in the DSMM layer. We apply and prove correct this optimization in the implementation of the erasure coded *data-access primitives* (DAP) used by the ARES framework (which includes CoARES and CoARESF). This optimization has its own interest, as it could be applicable beyond the ARES framework, i.e., by other erasure coded algorithms relying on tag-ordered DAPs. Finally, we have performed an in-depth experimental evaluation of our approach over both Emulab, a popular emulation testbest, and Amazon Web Services (AWS) EC2, an overlay (real-time) testbed (Section 5.4). 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 trade-offs and synergies. Ultimately, we aim to make a leap towards dynamic DSS that will be attractive for practical applications (like highly concurrent and strongly consistent file sharing).

The fourth part (Chapters 6, 7). Chapter 6 presents the remainder of the work for this thesis. Building on state-of-the-art DSS, the remainder of the work is to enhance the performance of our DSS (Section 6.1) and design a web-based User Interface (Section 6.2). For the former we should devise methodologies to reduce the latency of read and write operations, making services more practical and attractive for commercial use. To reach our goals we plan to identify performance bottlenecks using *Distributed Tracing* and devise algorithmic solutions on how to overcome those performance barriers. With distributed tracing one may inject checkpoints in a distributed system for monitoring the performance of individual procedures. Once bottlenecks are detected, developed solutions should ensure performance boost while preserving the correctness conditions of the original algorithms. Subsequently, it is essential to create a user-friendly interface that enables secure, seamless, and intuitive access to our DSS. To achieve this, we will first analyze the requirements and needs of the interface, specify the design re-

quirements, develop the interface, and integrate it with our DSS. Finally, Chapter 7 lists the contributions, talks about the limitations of our work, and offers a number of intriguing future perspectives, the development of a fully-functional DSS with a Reconfiguration Orchestration module, security guarantees, accessible through the user interface.

Chapter 2

System Settings, Definitions, and Experimental Evaluation Setup

In this chapter, we will present the system settings and definitions that apply to the entire work, as well as the evaluation setup that will be used to assess all implementations.

2.1 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 $\mathsf{read}()$, that takes no arguments and returns the value of the object, and (ii) a *write operation*, denoted by $\mathsf{write}(v)$, 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* [44] 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. To this respect, there are 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 setting). 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 4.1).

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.

Linearizability. We now formally define linearizability [5], following [45]. 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* $\mathsf{read}()$ *operation, then the response event of π returns v, and*
- *if π is a* $\mathsf{write}(v')$ *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.*

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

Coverability. Coverability [23] 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 [23], 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 *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)**.** *An execution ξ (resp. its history H_ξ) is a* valid execution *(resp. history) on a versioned object, for any $p_i, p_j \in \mathcal{I}$:*

- $\forall cvr\text{-}\omega(ver)[ver']_{p_i} \in \mathcal{W}_{\xi,succ}, ver < ver'$,
- *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 [23])**.** *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 $\tau_2.wid > \tau_1.wid$.

2.2 Experimental Evaluation Setup

After the presentation of the implementation of an algorithm, we present an experimental evaluation of it. In this section, we describe general information that applies to all these sets of experiments.

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 [40] as an emulation testbed and Amazon Web Services (AWS) EC2 [41] as an overlay testbed. We also used the Fed4FIRE+ project (`https://www.fed4fire.eu/`), which is the largest federation of testbeds in Europe. In particular, we use jFed [46], a GUI tool that was developed within the Fed4FIRE+ project, to get access and reserve virtual and physical machines in various experimental testbeds.

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

- CoABD. This is the coverable version of the traditional, static ABD algorithm [11, 12], as presented in [23] with some optimization. It will be used as an overall baseline.
- CoABDF. This is the version of CoABD that provides fragmented coverability, as presented in Section 3.3. It can be considered as a baseline algorithm of the CoBFS framework.
- AresABD. This is a version of Ares that uses the ABD-DAP implementation (cf. Section 4.2). It can be considered as the dynamic (reconfigurable) version of ABD.
- CoAresABD. This is a coverable version of Ares (CoAres), presented in Section 5.1, that uses the ABD-DAP implementation (cf. Section 4.2). The ABD-DAP implementation is optimized similarly to the EC-DAPopt presented in Section 5.2. CoAresABD can be considered as the dynamic (reconfigurable) version of CoABD.
- AresABDF. This is CoAresF (cf. Section 5.3) together with the ABD-DAP implementation, i.e., it is the fragmented version of CoAresABD.

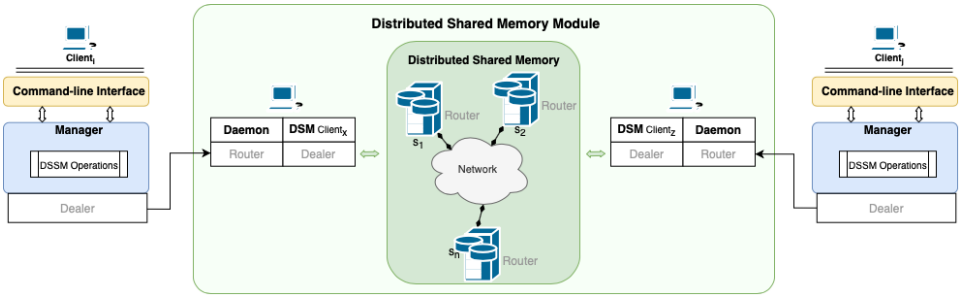


Figure 2.1: The architecture of our implementation.

- ARESEC. This is a version of ARES that uses the EC-DAP implementation (cf. Section 4.2).
- COARESEC. This is a coverable version of ARES (COARES) (see Section 5.1) that uses the EC-DAPopt implementation (see Section 5.2).
- ARESECF. This is the two-level striping algorithm presented in Section 5.3 when used with the EC-DAPopt implementation of Section 5.2, i.e., it is the fragmented version of ARESEC.

Note that we have implemented all the above algorithms using the same baseline code and communication libraries. They are based on the architecture depicted in Fig. 2.1. This includes the modules composing the infrastructure as well as the communication layer between these modules. The system is composed of two main modules: (i) a Manager, and (ii) a Distributed Shared Memory Module (DSMM). The manager provides an interface to each client for accessing the DSM (in our case a command line interface - CLI). Each client has its own manager that handles its commands (read and write operations through the Command Line Interface - CLI). Following this architecture, clients may access the file system through the Manager, while the shared objects are maintained by the servers through the DSMM. Notice that the Manager uses the DSMM as an external service to write and read objects to the shared memory. To this respect, our architecture is flexible enough to utilize any underlying DSM algorithm to implement the DSMM. 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 [47].

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.

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* [48], 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 Metris. We assess performance using: (i) *operational latency*, and (ii) *the update success ratio*. The operational latency includes both communication and computational delays. The operational latency is computed as the average of all clients' average operational latencies. When dealing with fragmented algorithms, computational latency includes the time taken to break down a file object into blocks and generate hashes for each of 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. The results in each scenario are compiled as averages over multiple samples per each scenario (the specific number of repetitions is indicated in each relevant chapter).

Chapter 3

Fragmented Objects: Boosting Concurrency of Shared Large Objects

In this chapter we deal with the storage and use of shared readable and writable data in unreliable distributed systems. Distributed systems are subject to perturbations, which may include failures (e.g., crashes) of individual computers, or delays in processing or communication. In such settings, large (in size) objects are difficult to handle. Even more challenging is to provide linearizable consistency guarantees to such objects.

Researchers usually break large objects into smaller linearizable building blocks, with their composition yielding the complete consistent large object. For example, a linearizable shared R/W memory is composed of a set of linearizable shared R/W objects [49]. By design, those building blocks are usually independent, in the sense that changing the value of one does not affect the operations performed on the others, and that operations on the composed objects are defined in terms of operations invoked on the (smallest possible) building blocks. Operations on individual linearizable registers do not violate the consistency of the larger composed linearizable memory space.

Some large objects, however, cannot be decomposed into independent building blocks. For example, a file object can be divided into *fragments* or *blocks*, so that write operations (which are still issued on the whole file) modify individual fragments. However, the composition of these fragments does not yield a linearizable file object: it is unclear how to order writes on the file when those are applied on different blocks concurrently. At the same time, it is practically inefficient to handle large objects as single objects and use traditional algorithms (like the one in [49]) to distribute it consistently.

As discussed, although the distributed storage emulations managed to meet the requirements of strong consistency, fault-tolerance, availability, and reliability of the data, they are not able to respond to the new requirements in terms of volume of data and high performance. In this chapter 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. In Section 3.1, we define two types of concurrent objects: (i) the *block* object, and (ii) the *fragmented* object. In Section 3.2, we define the precise consistency that the fragmented object provides, termed *Fragmented Coverable Linearizability*. Section 3.3 presents the algorithmic implementation of a distributed file system, called CoBFS, based on the theoretical framework of fragmented objects. Finally, Section 3.4 overviews an experimental development and deployment of CoBFS on the Emulab testbed [40].

3.1 Fragmented Objects

A *fragmented object* is a concurrent object (e.g., can be accessed concurrently by multiple processes) that is composed of a finite list of *blocks*. In this section we introduce the notion of a *block*, and gives the informal definition of a *fragmented object*. This section formally defines the notion of a *block*, and gives the formal definition of a *fragmented object*.

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 $\mathsf{create}(b, D)$ 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) $\mathsf{read}()_b$ that returns the value of the object b, and (ii) $\mathsf{write}(D)_b$ that sets the value of the object b to D, where $|D| \leq \ell$.

A block object is linearizable if is satisfies linearizability (cf. Definition 2 of Section 2.1) with respect to its create (which acts as a write), read, and write operations. Once created, a block object is an atomic register [50] whose value cannot exceed a predefined length ℓ.

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 $val(f) = \langle b_0, b_1, \ldots, b_n \rangle$, 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:

- $\mathsf{read}()_f$ returns the list $\langle val(b_0), \ldots, val(b_n) \rangle$, where $val(f) = \langle b_0, b_1, \ldots, b_n \rangle$
- $\mathsf{write}(\langle D_0, \ldots, D_n \rangle)_f$, $|D_i| \leq \ell, \forall i \in [0, n]$, sets the value of f to $\langle b_0, \ldots, b_n \rangle$ 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: $\pi_1 = \mathsf{write}(\langle D'_0, D_1 \rangle)_f$ and $\pi_2 = \mathsf{write}(\langle D_0, D'_1 \rangle)_f$, followed by a $\mathsf{read}()_f$. 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:

- $\mathsf{update}(b_i, D)_f$ updates the value of block $b_i \in f$ such that:
  - if $|D| \leq \ell$: sets $val(b_i) = D$;
  - if $|D| > \ell$: partition $D = \{D_0, \ldots, D_k\}$ 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 [51]. However, fragmented objects aim to minimize 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 $\mathsf{read}()_f$ 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 $\mathsf{update}(b_i, D)_f$ operation performs a write operation on the block b_i as $\mathsf{write}(D)_{b_i}$. However, if $|D| > \ell$, then D is partitioned into substrings $D_0, \ldots, D_k$ each of length at most ℓ. The update operation modifies the value of b_i as $\mathsf{write}(D_0)_{b_i}$. Then, k new blocks $b_i^1, \ldots, b_i^k$ are created as $\mathsf{create}(b_i^j, D_j), \forall j \in [1, k]$, 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 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* $\mathsf{read}()_f$*, 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* $\mathsf{update}(b_i, D)_f$ *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* $|D^j| \leq \ell, \forall j \in [0, k]$*.*[1]

3.2 Fragmented Coverable Linearizability

A fragmented object is linearizable if it satisfies both the *Liveness* (termination) and *Linearizability* (atomicity) properties (cf. Definition 2 of Section 2.1). 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.

Linearizability. We first proceed to present the formal definition of linearizability for a fragmented object. This definition is an extension of the linearizability definition for R/W objects, as defined in Definition 2 of Section 2.1, and it incorporates the sequential specification of a fragmented object, as defined in Definition 5 of Section 3.1.

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 3.1. 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

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

block, and the other attempting to modify the second block: $\pi_1 = \mathsf{update}(b_0, D'_0)_f$ and $\pi_2 = \mathsf{update}(b_1, D'_1)_f$ ($|D'_0| \leq \ell$ and $|D'_1| \leq \ell$), followed by a $\mathsf{read}()_f$ operation. In this case, since both update operations operate on different blocks, independently of how π_1 and π_2 are ordered in the permutation σ, the $\mathsf{read}()_f$ 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 Fig. 3.1a. However, providing a linearizable read when the object involves multiple R/W objects (i.e., an atomic snapshot) can be expensive or impact concurrency [52]. 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 Fig. 3.1b, 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 Fig. 3.1b 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 Fig. 3.1b. 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.

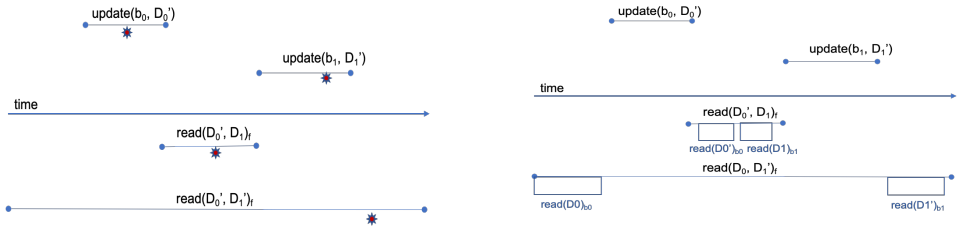


Figure 3.1: Executions showing the operations on a fragmented object. Fig. (a) shows linearizable reads on the fragmented object (and serialization points), and (b) reads on the fragmented object that are implemented with individual linearizable reads on blocks.

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

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 in-

vocation and response actions). Hence, if $val(f)$ is the value returned by $\mathsf{read}()_f$, then $H^{\mathsf{read}()_f}$ contains an invocation and matching response for a $\mathsf{read}()_b$ 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 $\mathsf{read}()_f$, if $\langle val(b_0), \ldots, val(b_n) \rangle$ is the value returned by the read operation, then we replace the invocation of $\mathsf{read}()_{b_0}$ operation with the invocation of the $\mathsf{read}()_f$ operation and the response of the $\mathsf{read}()_{b_n}$ block with the response action for the $\mathsf{read}()_f$ operation. Then, we remove from $H|_f$ all the actions in $H^{\mathsf{read}()_f}$.

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* [2]*, 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 [53].

As already discussed, in [23] the notion of coverability (cf. Definition 4 of Section 2.1) 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.

[2] The sequential specification of a block is similar to that of a R/W register, cf. Definition 1, whose value has bounded length.

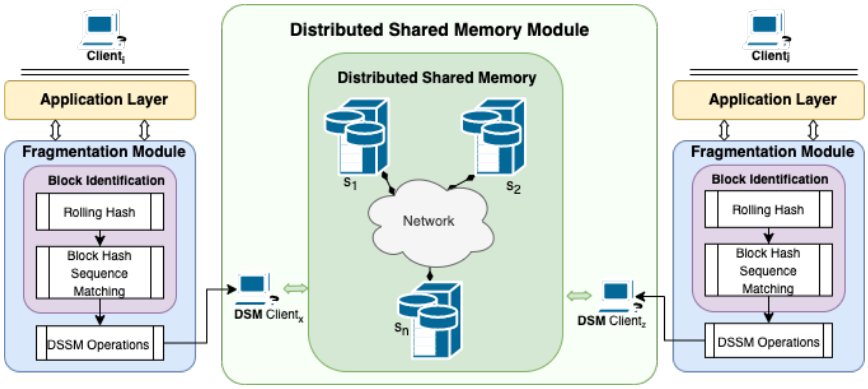


Figure 3.2: Architecture of CoBFS

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 next section 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.

3.3 Implementing Files as Fragmented Coverable Objects

Having laid out the theoretical framework of Fragmented Objects, we now briefly present a prototype implementation of a Distributed File System, we call CoBFS.

Coverability, as described in the Section 3.2, is crucial for updating the content of a file. When updating a file, one expects the update to be on the previous version of the file. 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 file system provides *fragmented coverability* as a consistency guarantee. In our prototype implementation we consider each object to be a file (either text or binary), however the underlying theoretical formulation allows for extending this implementation to support any kind of large object. More details can be found in [54].

File as a coverable fragmented object. Each file is modeled as a fragmented object with its blocks being coverable objects. The file 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 file contains only the genesis block; the genesis block contains special purpose (meta) data. The $val(b)$ of b is set as a tuple, $val(b) = \langle ptr, data \rangle$.

Algorithm 1 DSM Module: Operations on a coverable block object b at client p

1: **State Variables:**
2: $ver_b \in \mathbb{N}$ initially 0; $val_b \in \mathcal{V}$ initially $\perp$;
3: **function** dsmm-read($)_{b,p}$
4: $\langle ver_b, val_b \rangle \leftarrow b$.cvr-read()
5: **return** val_b
6: **end function**
7: **function** dsmm-create$(val)_{b,p}$
8: $\langle ver_b, val_b \rangle \leftarrow b$.cvr-write$(val, 0)$
9: **end function**
10: **function** dsmm-write$(val)_{b,p}$
11: $\langle ver_b, val_b \rangle \leftarrow b$.cvr-write$(val, ver_b)$
12: **return** val_b
13: **end function**

Overview of the Architecure. The architecture of COBFS appears in Fig. 3.2. COBFS is composed of two main modules: (i) a Fragmentation Module (FM), and (ii) a Distributed Shared Memory Module (DSMM). In summary, the FM implements the fragmented object 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 file system through the FM, while the blocks of each file are maintained by servers through the DSMM. 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 any underlying distributed shared object algorithm.

File and block id assignment. A key aspect of our implementation is the unique assignment of ids to both fragmented objects (i.e., files) and individual blocks. A file $f \in F$ 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 file (i.e., the owner) and $cfseq \in \mathbb{N}$ is the client's local sequence number, incremented every time the client creates a new file and ensuring uniqueness of the objects created by the same client.

In turn, a block $b \in \mathcal{B}$ of a file 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 file 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 file), 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 file (this ensures the uniqueness of the blocks created by the same client for the same file).

Distributed Shared Memory Module. The DSMM implements a distributed R/W shared memory based on an *optimized coverable variant* of the ABD algorithm, called COABD [23]. The module 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.

To reduce the number of blocks transmitted per read, we apply a simple yet very effective optimization (Algorithm 2): a read sends a READ request to all the servers including its local version in the request message. When a server receives a READ request it replies with both its local tag and block content only if the tag enclosed in the READ request is smaller than its local tag; otherwise it replies with its local

Algorithm 2 Optimized coverable ABD

1: at each reader r for object b
2: **State Variables:**
3: $tg_b \in \mathbb{N}^+ \times \mathcal{W}$ initially $\langle 0, \perp \rangle$; $val_b \in V$, init. $\perp$;

4: **function** cvr-read( )
5: **send** $\langle \text{READ}, ver_b \rangle$ to all servers ▷ Query Phase
6: **wait until** $\frac{|\mathcal{S}|+1}{2}$ reply
7: $maxP \leftarrow \max(\{\langle tg', v' \rangle\})$
8: **if** $maxP.tg > tg_b$ **then**
9: **send** (WRITE, $maxP$) to all servers
10: ▷ Propagation Phase
11: **wait until** $\frac{|\mathcal{S}|+1}{2}$ servers reply
12: $\langle tg_b, val_b \rangle \leftarrow maxP$
13: return($\langle tg_b, val_b \rangle$)
14: **end function**

15: at each server s for object b
16: **State Variables:**
17: $tg_b \in \mathbb{N}^+ \times \mathcal{W}$ initially $\langle 0, \perp \rangle$; $val_b \in V$, init. $\perp$;

18: **function** rcv$(M)_q$ ▷ Reception of a message from q
19: **if** $M.type \neq$ READ and $M.tg > tg_b$ **then**
20: $\langle tg_b, val_b \rangle \leftarrow \langle M.tg, M.v \rangle$
21: **if** $M.type =$ READ and $M.tg \geq tg_b$ **then**
22: send($\langle tg_b, \perp \rangle$) to q ▷ Reply without content
23: **else**
24: send($\langle tg_b, val_b \rangle$) to q ▷ Reply with content
25: **end function**

tag without the block content. Once the reader receives replies from a majority of servers, 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 block content to a majority of servers; if not then the read operation returns the locally known tag and block content without performing the second phase. While this optimisation makes a 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 second phase, then the read sends the complete file 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 first round of the write operation, in the READ phase, 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 second round in any situation, whereas the read operation performs it if the reader does not have the latest value, as explained above.

The create and write operations invoke cvr-write operations to update the value of the shared block b. Their main difference is that version 0 is used during a create operation to indicate that this is the first time that the block is written. Notice that the write in create will always succeed as it will introduce a new, never before written block, whereas operation write may be converted to a read operation, thus retrieving and returning the latest value of b. We refer the reader to [23] for the implementation of cvr-read and cvr-write, which are simple variants of the corresponding implementations of ABD [49].

Fragmentation Module. The FM is the core concept of our implementation. Each client has a FM responsible for (i) fragmenting the file into blocks and identify modified blocks, and (ii) follow a specific strategy to store and retrieve the file 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. For the file division of the blocks and the identification of the newly created blocks, the

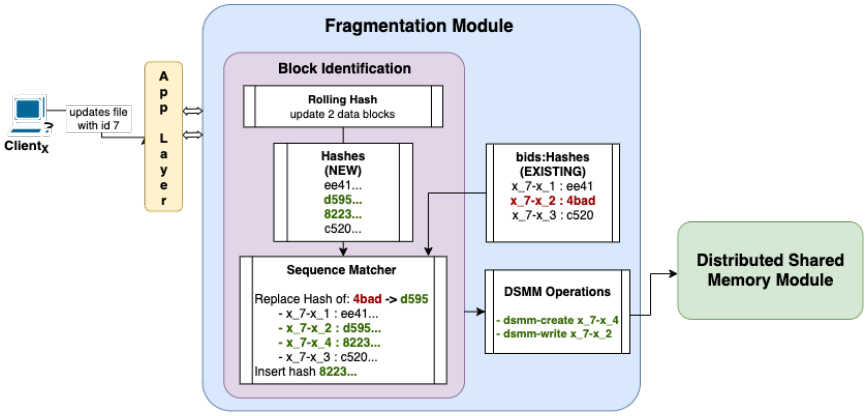


Figure 3.3: Example of a writer x writing text at the beginning of the second block of a text file with id $f_{id} = 7$. The hash value of the existing second block "$4bad$.." is replaced with "$d595$.." and a new block with hash value "8223.." is inserted immediately after. The block b_{id} = x_7-x_2 and the new block b_{id} = x_7-x_4 are sent to the DSM.

FM contains a *Block Identification (BI) module* that utilizes known approaches for data fragmentation and diff extraction.

Block Identification (BI). Given the data D of a file 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 [55], given two versions of the same file, 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 [56], and (2) it compares the hashes of the blocks of the current and the previous version of the file using a string matching algorithm [57] to determine the modified/new data blocks. The role of BI within the architecture of CoBFS and its process flow appears in Fig. 3.3, while its specification is provided in Algorithm 3. A high-level description of *BI* has as follows:

- **Block Division:** Initially, the BI partitions a given file f into data blocks based on its contents, using *Rabin fingerprints*. This scheme allows to divide f into blocks of at most size ℓ, which are identified by their hashes (fingerprints). When used in two versions of the same file, 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 $\langle D_0, \ldots, D_m \rangle$ and associated block hashes $\langle h_0, \ldots, h_m \rangle$ generated by the Rabin fingerprint algorithm, the BI tries to match each hash to a block identifier, based on the block ids produced during the previous division of file f, say $\langle b_0, \ldots, b_n \rangle$. We produce the vector

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

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

5: **function** fm-block-identify($)_{f,p}$
6: $\langle newD, newH \rangle \leftarrow$ RabinFingerprints(f, ℓ)
7: $curH = hash(\mathcal{L}_f)$
8: ▷ hashes of the data of the blocks in $\mathcal{L}_f$
9: $C \leftarrow$ SMatching$(curH, newH)$
10: ▷ blocks modified
11: **for** $\langle h(b_j), h_k \rangle \in C.mods$ s.t.
12: $h(b_j) \in curH, h_k \in newH$ **do**
13: $D \leftarrow \{D_k : D_k \in newD$
14: $\wedge\, h_k = hash(D_k)\}$
15: fm-update$(b_j, D)_{f,p}$
16: ▷ blocks inserted
17: **for** $\langle h(b_j), S \rangle \in C.inserts$ s.t.
18: $h(b_j) \in curH, S \subseteq newH$ **do**
19: $D \leftarrow \{D_i : h_i \in S \wedge D_i \in newD$
20: $\wedge\, h_i = hash(D_i)\}$
21: fm-update$(b_j, D)_{f,p}$
22: **end function**

23: **function** fm-read($)_{f,p}$
24: $b \leftarrow val(b_g).ptr$
25: $\mathcal{L}_f \leftarrow \langle b_g \rangle$ ▷ reset $\mathcal{L}_f$
26: **while** b *not* NULL **do**
27: $val(b) \leftarrow$ dsmm-read$()_{b,p}$
28: $\mathcal{L}_f.insert(val(b))$
29: $b \leftarrow val(b).ptr$
30: **end while**
31: **return** Assemble$(\mathcal{L}_f)$
32: **end function**

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

$\langle h(b_0), \ldots, h(b_n) \rangle$ where $h(b_i) = hash(val(b_i).data)$ from the current blocks of f, and using a string matching algorithm [57] 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 update$(b_j, \{D_i\})_{f,p}$ action is then issued to modify the data of b_j to D_i (Lines 10:15). In case new hashes (e.g., $\langle h_i, h_k \rangle$) are inserted after the hash of block b_j (i.e., $h(b_j)$), then the action update$(b_j, \{val(b_j).data, D_i, D_k\})_{f,p}$ is performed to create the new blocks after b_j (Lines 16:21). In our formulation block deletion is treated as a modification that sets an empty data value thus, in our implementation *no blocks are deleted*.

FM Operations. The FM's external signature includes the two main operations of a fragmented object: read$_f$, and update$_f$. Their specifications appear in Algorithm 3.

Read operation - read$()_{f,p}$**:** To retrieve the value of a file f, a client p may invoke a read$_{f,p}$ to the fragmented object. Upon receiving, the FM issues a series of reads on file'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 comprising the file. All the blocks are assembled into one file via the Assemble() function. The reader p issues a read for all the blocks in the file.

Update operation - $\mathsf{update}(b, D)_{f,p}$**:** Here we expect that the update operation accepts a block id and a set of data blocks (instead of a single data object), since the division is performed by the BI module. 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. Client p attempts to update the value of a block with identifier b in file f with the data in D. Depending on the size of D the update operation will either perform a write on the block if $k = 0$, or it will create new blocks 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 $\mathsf{write}(\langle val(b).ptr, D_0 \rangle, \langle p, bseq \rangle)_b$.
- $k > 0$: Given the sequence of chunks $D = \langle D_0, \ldots, D_k \rangle$ the following block operations are performed in this particular order:

  $\rightarrow$ $\mathsf{create}(b_k = \langle f, p, bc_p\text{++} \rangle, \langle b', D_k \rangle, \langle p, 0 \rangle)$ ** Block b_k ptr points to b' **

  $\rightarrow$ ...

  $\rightarrow$ $\mathsf{create}(b_1 = \langle f, p, bc_p\text{++} \rangle, \langle b_2, D_1 \rangle, \langle p, 0 \rangle)$ ** Block b_1 ptr points to b_2 **

  $\rightarrow$ $\mathsf{write}(\langle b_1, D_0 \rangle, \langle p, bseq \rangle)_b$ ** Block b ptr points to b_1 **

The challenge here 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 backwards 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*.

In [54], we rigorously prove the following:

Theorem 8. CoBFS *implements a R/W Fragmented Coverable object.*

Proof sketch. We assume that a minority of servers (less than $|\mathcal{S}|/2$) may crash. We prove that every block operation in CoBFS satisfies coverability and it follows that CoBFS 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 we ensure that each operation obtains a connected list of blocks. Thus, CoBFS implements a *valid* fragmented object. □

3.4 Experimental Evaluation of CoBFS

We give an overview of our evaluation, and more results can be found in [54].

Evaluated Algorithms. To further appreciate the proposed approach from an applied point of view, we performed an evaluation of CoBFS against CoABD on Emulab [40]. Due to the design of the two algorithms, CoABD will transmit the entire file per read/update operation, while CoBFS 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 2.

Distributed Experimental Setup. On Emulab we used physical nodes with one 2.4 GHz 64-bit Quad Core Xeon E5530 "Nehalem" processor and 12 GB RAM. For the evaluation we generate a text file with random byte strings whose size increases as the writers keep updating it.

Parameters. ABD-based algorithms assume that only a minority of servers (less than $|\mathcal{S}|/2$) may crash. Thus, the quorum size of the ABD-based algorithms is $\lfloor \frac{n}{2} \rfloor + 1$. The parameter n is the total number of servers.

Experimental Scenarios and Results. We used a *stochastic* invocation scheme in which clients pick a random time between the interval $[1...4sec]$ to invoke their next operations. The results shown are compiled as averages over five samples per each scenario.

Performance VS. Initial File Sizes. We varied the f_{size} from $1\,\mathrm{MB}$ to $1\,\mathrm{GB}$ by doubling the file size in each simulation run. The number of writers, readers and servers was fixed to 5. In total, each writer performed 5 updates and each reader 5 reads. The maximum, minimum and average block sizes were set to $1\,\mathrm{MB}$, $512\,\mathrm{kB}$ and $512\,\mathrm{kB}$ respectively.

Results. Figure 3.4a demonstrates that the update operation latency of CoBFS remains at extremely low levels. The main factor that significantly contributes to the slight increase of CoBFS update latency is the FM computation latency, Fig. 3.4b. 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 FM 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 CoABD which 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.
Despite the higher success rate of CoBFS, the read latency of the two algorithms is comparable due to the low number of update operations. The read latencies of the two algorithms with and without the read optimization can be seen in Fig. 3.4c. 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 read optimization is not used in CoBFS, the latency is close of

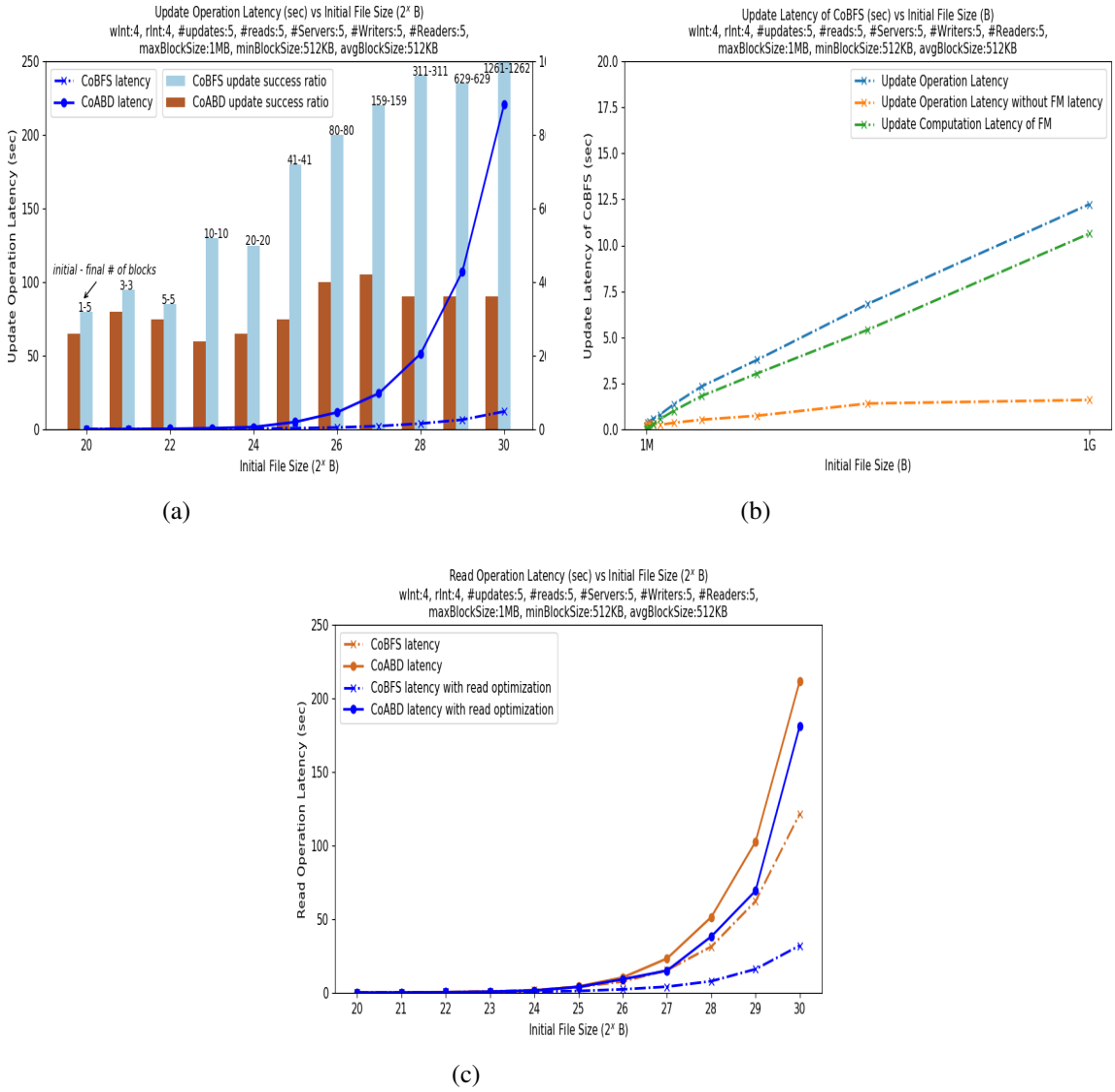
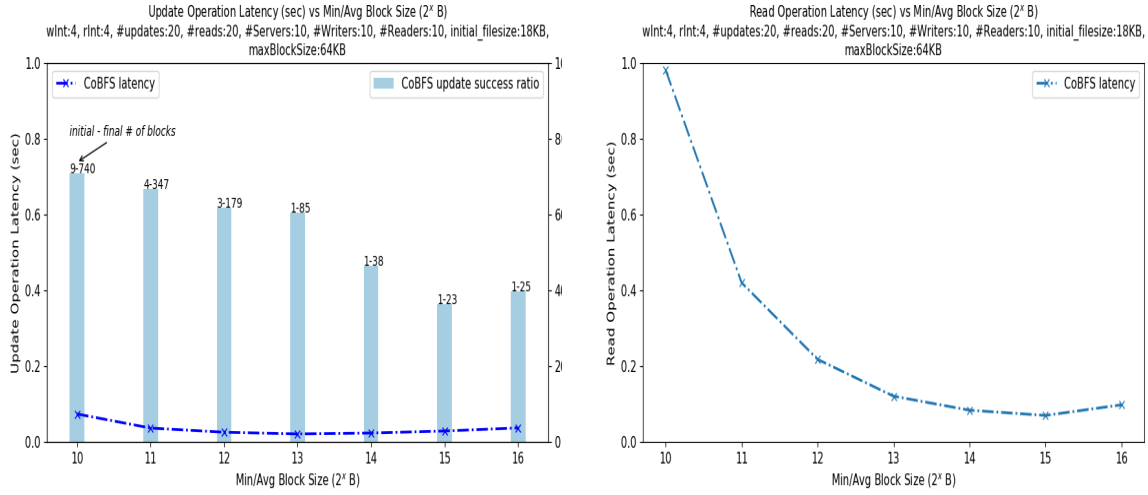


Figure 3.4: Filesize results for algorithms CoABD and CoBFS.

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, read optimization decreases significantly the CoBFS read latency, as reads transmit only the contents of the blocks that have changed.

Performance VS. Block Size. We varied the minimum and average b_{sizes} of CoBFS from 1 kB to 64 kB. The number of writers, readers and servers was fixed to 10. In total, each writer performed 20 updates and each reader 20 reads. The size of the initial file used was set to 18 kB, while the maximum block size was set to 64 kB.



(a) (b)

Figure 3.5: Block Size results for algorithms CoABD and CoBFS.

Results. From Figs. 3.5a, 3.5b we can infer that when smaller blocks are used, the update and read latencies reach their highest values. In both cases, small b_{size} results in the generation of larger number of blocks from the division of the initial file. Additionally, as seen in Fig. 3.5a, the small b_{size} leads to the generation of more new blocks during update operations, resulting in more update block operations, and hence higher latencies. As the minimum and average b_{sizes} increase, lower number of blocks need to be added when an update is taking place. Unfortunately, smaller number of blocks leads to a lower success rate. Similarly, in Fig. 3.5b, 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 blocks need to be read. Thus, further increase of the minimum and average b_{sizes} forces the decrease of the latencies, reaching a plateau in both graphs. This means that the emulation finds optimal minimum and average b_{sizes} and increasing them does not give better (or worse) latencies.

Observations Regarding Concurrency. The percentage of successful file updates achieved by CoBFS are significantly higher than those of CoABD. As we observed in [54], this holds for both cases where the number of writers increased and the number of servers increased. This demonstrates the boost of concurrency achieved by CoBFS. We notice that as the number of writers increases (hence, concurrency increases), CoABD suffers greater number of unsuccessful updates, i.e., updates that have become reads per the coverability property. Concurrency is also affected when the number of blocks increases, Fig. 3.4a. The probability of two writes to collide on a single block decreases, and thus CoBFS eventually allows all the updates (100%) to succeed. CoABD does not experience any improvement as it always manipulates the file as a whole.

3.5 Conclusions

We have introduced the notion of coverable fragmented objects and proposed an algorithm that implements coverable fragmented files. This algorithm is then used to build CoBFS, a prototype distributed file system in which each file is specified as a linked-list of coverable blocks. CoBFS adopts a modular architecture, separating the object fragmentation process from the shared memory service. This allows CoBFS to follow different fragmentation strategies and shared memory implementations. We showed that it preserves the validity of the fragmented object (file) and satisfies fragmented coverability. The deployment of CoBFS on Emulab serves as a proof of concept implementation. The evaluation demonstrates the potential of our approach in boosting the concurrency and improving the efficiency of R/W operations on strongly consistent large objects.

Chapter 4

Implementation and Experimental Evaluation of ARES

For more than two decades, a series of works, e.g., [11, 12, 13, 14, 15, 16, 17, 18], suggested solutions for building 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.

It is apparent that those solutions cannot be found readily and were not adopted by commercial distributed storage applications. Commercial Disrtibuted Storage Systems (DSS), such as Dropbox, HDFS, CASSANDRA and REDIS, avoid providing strong consistency guarantees (such as atomicity) as they are considered costly and difficult to implement in an asynchronous, fail prone, message passing environment. Hence, such solutions either choose to offer weaker or tunable guarantees to achieve better performance when atomicity is not preserved.

Indeed, initial implementations of ADSM had high demands in communication, storage, and sometimes computation. Recent works, however, e.g., [16, 58], invest in algorithms that may reduce the overheads on the aforementioned parameters. ARES [16] is a recent ADSM algorithm, which proposes a modular approach for providing a *dynamic* shared memory space. ARES may use any ADSM algorithm at its core, providing the flexibility to adjust its performance based on the application demands.

In this chapter, we provide an overview of ARES (Section 4.1) and describe its implementation (Section 4.2). Next, we present experiments comparing its efficiency and adaptiveness of the algorithm (Section 4.3). We also conduct experiments to compare ARES with commercial solutions, such as REDIS and CASSANDRA (Section 4.4). These experiments involve measuring various performance metrics, including throughput, latency, and scalability, under different workloads and network conditions.

4.1 Overview of ARES

In this section, we provide an overview of ARES, including its design principles and key features. For more information, please refer to [16].
ARES is a modular framework, designed to implement dynamic, reconfigurable, fault-tolerant, read/write distributed linearizable (atomic) shared memory objects. We should begin by explaining how a configuration is explained 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) [17]. 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 [16], 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., [43]). 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 [16], 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 paper [16]), and the DAPs for an erasure coded based approach presented for the first time in Section 5 of [16]. 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 [59]) 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 $\mathsf{recon}(c)$, 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 [16], this reconfiguration procedure guarantees that configuration sequences obtained by any two clients $cseq_p$ and $cseq_q$, then either $cseq_p$ is a prefix of $cseq_q$, or vice versa.

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

4.2 Implementation of ARES

The theoretical findings in [16] suggest that ARES is an algorithm to provide robustness and flexibility on shared memory implementations, without sacrificing strong consistency. In this section we present a *proof-of-concept* implementation of ARES.

For the purposes of this study we have developed our own implementation of ARES. Our implementation is based on the architecture depicted in Fig. 2.1. In our case we implemented two algorithms. At first, we integrated algorithm ABD to our DSM Module. Next, we implemented algorithm ARES with two different DAPs (ABD and EC) and then we integrated that implementation to the DSM Module. Python was chosen as the programming language and ZeroMQ [47] messaging library written in Python (the Dealer-Router paradigm) for the underlying communication.

We used an external implementation of Raft [60] consensus algorithms, which was used for the service reconfiguration (line 16 of Alg. 2 in [16]) 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. The Python implementation of Raft used for consensus is PySyncObj [61]. Some modifications were done to allow the execution of Raft in the ARES environment. We built an HTTP API for the management of the Raft subsystem. Our API is a Key−Value storage where the key is the config ID and the value is the configuration. A reconfigurer can propose a configuration at a particular index in the configuration sequence by sending a POST request to the url of each Raft node, and receives a response from the RAFT on which configuration is decided for that index. The value to be written must be in a json format. The response will be also in a json format showing the latest value regarding the key. This latest value may be the one just written by the reconfigurer, perhaps an older one if its write did not succeed.

Implementation of DAP_s. Clients initialize the appropriate configuration objects to handle any request. Notice that the client creates a configuration object when it is the first time that the client requests an operation or when doing a reconfiguration operation. Once the configuration object is initialized, it is stored on the client $cseq$ and it is retrieved directly on any subsequent request from the client. Therefore, the DAP_s procedures are called from a configuration object.

Erasure Coding. The type of erasure coding we use is (n,k)-Reed-Solomon code, which guarantees that any k of n coded fragments is enough to reassemble the original object. The parameter k is the number of encoded data fragments, n is the total number of servers and m is the number of parity fragments, i.e. $n-k$. 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. In practice, the get-data and put-data functions from algorithm 5 in [16] integrate the standard Reed-Solomon implementation provided by liberasurecode from the PyEClib Python library [62].

4.3 Experimental Evaluation of ARES

We provide an overview of our evaluation, and further findings can be found in [16].

Evaluated Algorithms. In our scenarios we constructed the DAP_s and used two different atomic storage algorithms in ARES: (i) the erasure coding based algorithm presented in Section 5 of [16], and (ii) the ABD algorithm (see Section 8.1 in paper [16]). In particular, our experiments measure the latency of each read, write, and reconfig operations, and examine the persistence of consistency even when the service is reconfigured between configurations that add/remove servers and utilize different shared memory algorithms.

Experimental Setup. We ran experiments on two different setups: (i) simulated locally on a single machine, and (ii) on a LAN. Both type of experiments run on *Emulab* [40]. We used nodes with two 2.4 GHz 64-bit 8-Core E5-2630 "Haswell" processors, 64 GB RAM, with 1 GB and 10 GB NICs. For the evaluation we generate a text file whose size is fixed through the experiments.

Local Experimental Setup. The local setup was used to have access to a global synchronized clock (the clock of the local machine) in order to examine whether our algorithm preserves global ordering and hence atomicity even when using different algorithms between configurations. Therefore, all the instances are hosted on the same physical machine avoiding the skew between computer clocks in a distributed system. Furthermore, the use of one clock guarantees that when an event occurs after another, it will assign a later time.

Parameters. ABD assumes that only a minority of servers (less than $|\mathcal{S}|/2$) may crash, whereas EC can tolerate the failure of at most $|\frac{|\mathcal{S}|-k}{2}|$ servers, where k is the number of encoded data fragments (cf. Section 4.1). Thus, 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. In all scenarios, the number of servers is fixed to 10. The number of writers and the value of delta are set to 5. The parity value of the EC is set to 2 in order to minimize the redundancy, leading to 8 data servers and 2 parity servers. It is worth mentioning that the quorum size of the EC algorithm is $\lceil\frac{10+8}{2}\rceil = 9$, while the quorum size of ABD algorihtm is $\lfloor\frac{10}{2}\rfloor+1 = 6$. In relation to the EC algorithm, 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.

Experimental Scenarios and Results. As a general observation, ARES with EC storage provides data redundancy with a lower communicational and storage cost compared to the ABD storage that uses a strict replication technique.
For better estimations, each experiment in every scenario was repeated 6 times. In the graphs, we use error bars to illustrate the standard error of the mean (SEM) from the 6 repeated experiments. In most graphs, almost all the SEM values are low which shows that the calculated mean at each point is precise. In total, each client performs 60 operations. We used a *stochastic* invocation scheme in which clients pick a random time between the interval $[1...2sec]$ to invoke their next operations. The reconfigurer invokes its next operation every $15sec$ and performs a total of 6 reconfigurations.

Performance VS. File Size Scalability (Emulab). The first scenario is made to evaluate how the read and write latencies are affected by the size of the shared object. There are two separated runs, one for each examined storage algorithm. The file size is doubled from $1\,\mathrm{MB}$ to $128\,\mathrm{MB}$. The number of readers is fixed to 5, without any reconfigurers.

Results. Fig. 4.1a shows the results of the file size scalability experiments. The read and write latencies of both storage algorithms remain in low levels until $16\,\mathrm{MB}$. In bigger sizes we observe the latancies of all operations to grow significantly. It is worth noting that the fragmentation applied by the EC algorithm, benefits its write operations which follow a slower increasing curve than the rest of the operations. From the rest the reads seem to suffer the worst delay hit, as they are engaged in more communication phases. Nevertheless, the larger messages sent by ABD result in slower read operations. We had noticed that EC has lower SEM values than ABD, which indicates that the calculated mean latencies of EC align very closely throughout the experiments. As EC breaks each file into smaller fragments, in combination with the fact that the variation is smaller when using smaller files in ABD, may lead to the conclusion that the file size has a significant impact on the error variation. To this end it appears that larger file sizes introduce higher variation on the delivery times of the file and hence higher statistical errors.

Performance VS. Changing Reconfigurations (Emulab). In this scenario, we evaluate how the read and write latencies are affected when increasing the number of readers, while also changing the storage algorithm. The reconfigurer switches between the two storage algorithms. The size of the file used is $4\,\mathrm{MB}$.

Results. In Fig. 4.1b, we can find the results of the experiments. Note that there are cases where a single read/write operation may access configurations that implement both ABD and EC algorithms, when concurrent with a recon operation. Thus, the latencies of such operations are accounted in both ABD and EC latencies. 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, the ARES may not benefit so much from the coding, bringing the delays of the two storage algorithms closer to each other. It is

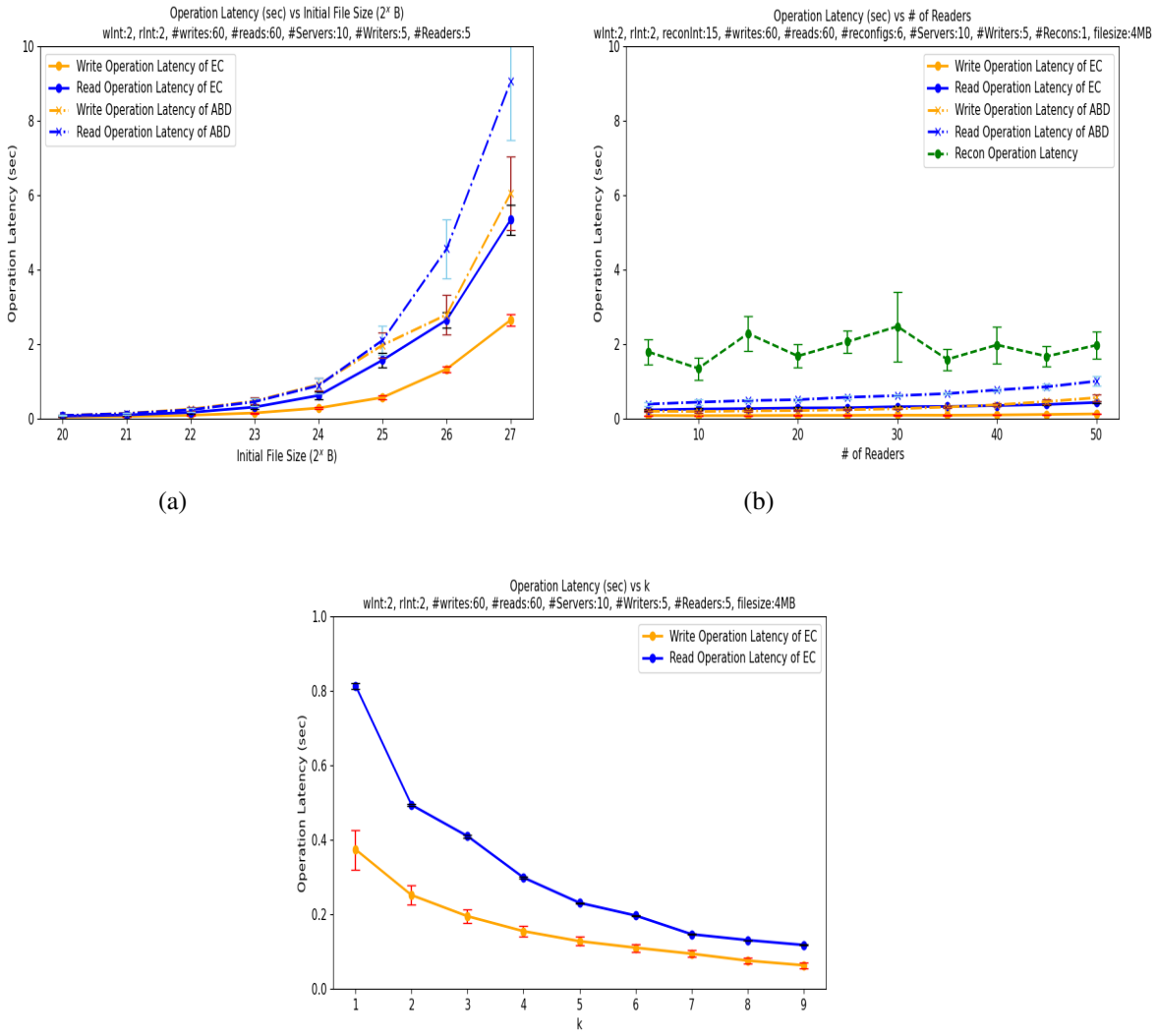


Figure 4.1: Simulation results.

again obvious that the reconfiguration delays are higher than the delays of all other operations.

Performance VS. k ***Scalability (Emulab, EC only).*** In this scenario, we examine the read and write latencies with different numbers of k (a parameter of Reed-Solomon). We increase the k of the EC algorithm from 1 to 9. The number of readers is fixed to 5, without any reconfigurers. The size of the file used is $4\,\mathrm{MB}$.

Results. From Figs. 4.1c we can infer that when smaller k are used, the write and read latencies reach their highest values. In both cases, small k results in the generation of smaller number of data fragments and thus bigger sizes of the fragments

and higher redundancy. For example we can see that for RS(10,8) and RS(10,7) we have the same size of quorum, equal to 9, whereas the latter has more redundant information. As a result, with a higher number of m (i.e. smaller k) we achieve higher levels of fault-tolerance, but that it would waste storage efficiency. The write latency seems to be less affected by the number of k since the encoding is considerably faster as it requires less computation. In conclusion, there appears to be a trade-off between operation latency and fault-tolerance in the system: the further increase of the k (and thus lower fault-tolerance) the smaller the latency of read/write operations. This experiment proves that the object size plays a significant role on the error variation. Notice that while k is small, and thus the object we send out is bigger, the error is higher. As k goes bigger and the fragments get smaller the SEM minimizes. This is an indication that communication of larger data over the wire may fluctuate the delivery times (as also seen in the file size scenario).

4.4 Experimental Comparison: ARES VS. CASSANDRA VS. REDIS

We present an overview of our comparison, and for more results, readers may refer to [63] or visit our project website[1]. The website presents interactive plots where the user may choose the parameters to apply. The collected data for these experiments are available in [64], in case one would like to validate our analysis.

Evaluated Algorithms. In our experiments, we conducted the well-known ABD [11] algorithm, the ARES with two different DAP_s (cf. Section 4.1): (i) the erasure coding based algorithm, and (ii) the ABD algorithm, CASSANDRA [31] and REDIS [32].

CASSANDRA **Implementation.** We deployed the Apache Cassandra 4 on multiple nodes with Ubuntu 18.04.1 LTS or 20.04 LTS. In order to guarantee atomicity, as in ARES and ABD, we set the CL 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 RF is n, then $n/2$ +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. 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.

REDIS **Implementation.** We deployed REDIS 5 on multiple nodes with Ubuntu 18.04.1 LTS or 20.04 LTS. We implement two variants of REDIS, with and without

[1] `https://projects.algolysis.com/ares-ngi/results/`

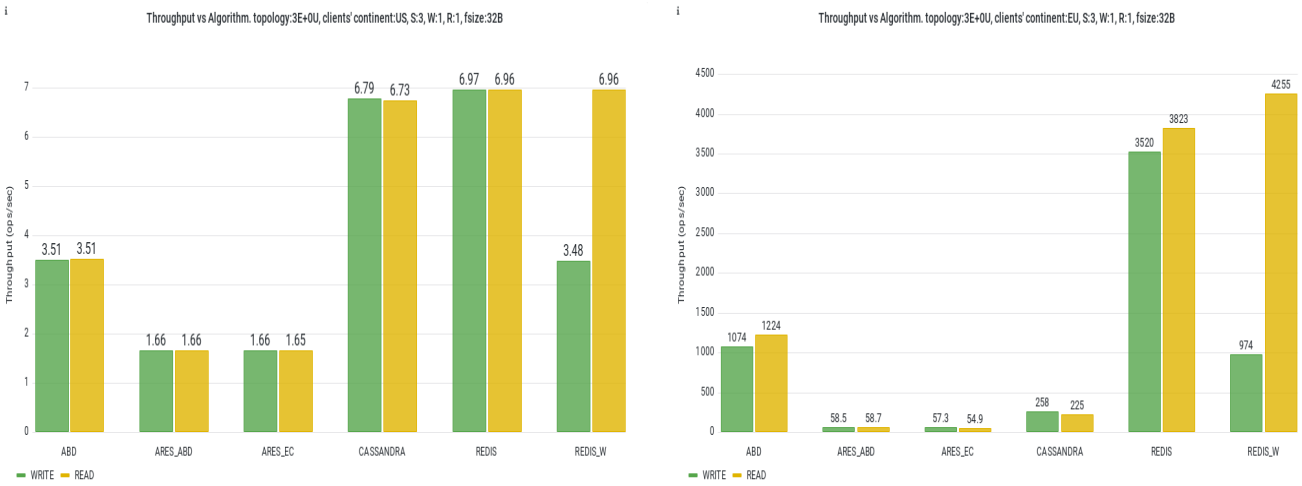
the WAIT command during a write operation, i.e., REDIS_W and REDIS, respectively. For the REDIS_W, we specified the number of waiting write acknowledgments with a majority, i.e., $n/2 + 1$, to match the ABD algorithm. To send read and write requests we created a script using the Redis-driver Python library. First, the script creates a connection to REDIS, giving the IP and port of the master node. Once connected to REDIS, the client can write and read with REDIS command functions, *set* and *get* respectively. A writer assigns a file's byte string value to the REDIS key; it uses the file's id as the key, while a reader gets the value giving the file's id. We note that the number of reader clients can dynamically increase or decrease. However, if the Master crashes, the writes will be blocked, as the replica nodes are read only, until a new replica becomes the new master; with this respect, reconfiguration in REDIS is blocking.

Distributed Experimental Setup. Our main goal was to conduct real-life experiments, exposed to the perturbations, delays, and uncertainty of network communication. We picked devices both in the EU and the USA using the jFed GUI tool [46] , thus, examining the impact of long (cross-Atlantic) communication on the performance of each algorithm. For the evaluation we generate a text file with random byte strings whose size increases as the writers keep updating it.

Experimental Scenarios and Results. In throughput experiments, operations are invoked without any delay (i.e., an operation is invoked once the previous operation by the same client is completed), and the clients perform 1000 operations each. For all other experiments we use a stochastic invocation scheme: each client waits a **random interval** each time it terminates an operation and before invoking the next one. Reads and writes are scheduled at a random interval between $[1 \dots 3]$ seconds. In total, each writer performs 50 writes and each reader 50 reads. Each reconfigurer invokes one operation every $15sec$ and performs a total of 15 reconfigurations. The results shown are compiled as averages over 3 samples per each scenario and 5 samples for the topology scenario.

Stress Test – Topology. This scenario aims to measure how the performance of the algorithms is affected under different topologies and server participation. In this case we measure the throughput (average number of operations per second) of each algorithm. To avoid any delays due to operation contention, we chose to use 2 clients (1 reader and 1 writer), the minimum number of servers to form a majority, i.e. 3, and a simple object of $32\,\mathrm{B}$. As we deployed machines on both EU and USA, our servers are split in such a way to either force all of them or their majority to be in a single continent. In particular, the 3 servers selected based the following topologies: $0E + 3U$, $1E + 2U$, $2E + 1U$, $3E + 0U$, where xY means that x servers are deployed in Y continent for $E = EU$ and $U = USA$. Similarly we deployed the clients either close (i.e., to the same continent) or away from the server majority. Last, we tested the throughput of the algorithms when the number of servers is growing from 3 to 15. In this case, for every server deployed in EU, we deployed 2 servers in the USA.

Results. Some results from these experiments appear in Figs. 4.2a, 4.2b and 4.3. Overall the topology played a major role on the performance, and in particular



(a) (b)

Figure 4.2: Throughput vs Algorithm. Topology: $3E + 0U$.

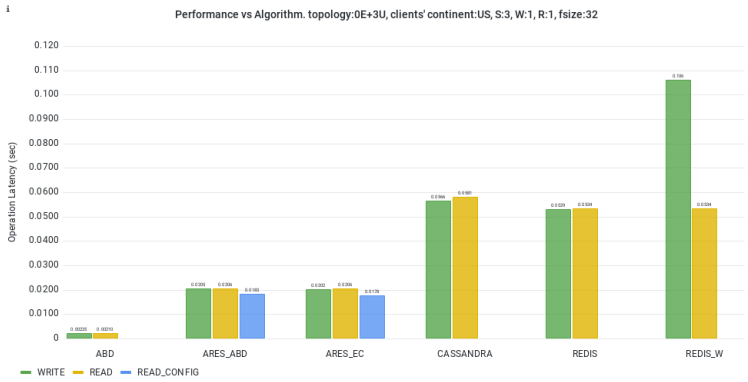
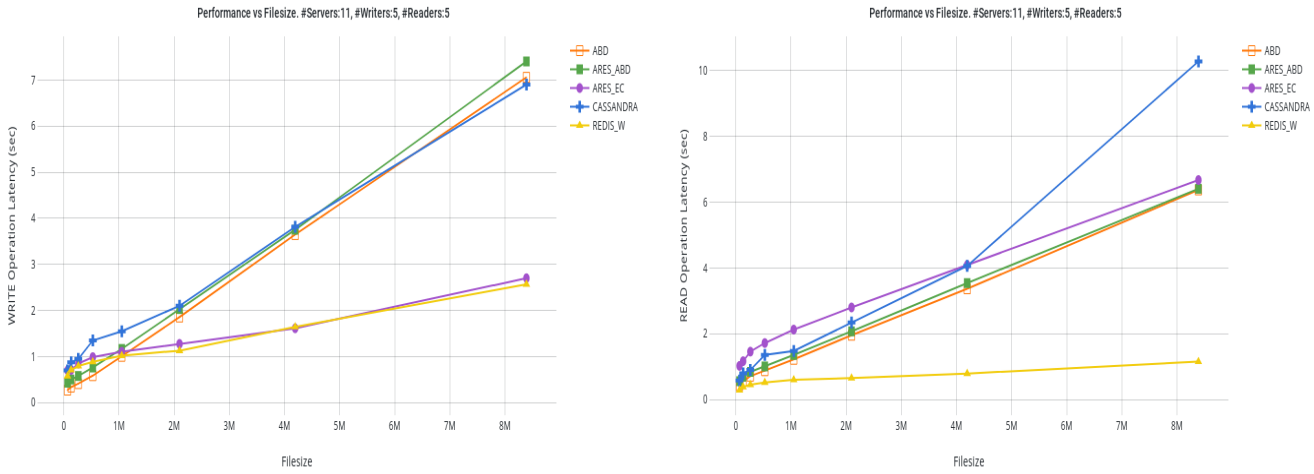


Figure 4.3: Performance vs Algorithm. Topology: $0E + 3U$

throughput, of all the algorithms we studied. All of the algorithms (including the ADSM algorithms we implemented, i.e., ABD, AresABD, and AresEC), achieve their maximum read and write throughput when the servers and the clients are deployed in the same continent.

For the ADSM algorithms, there appears to be no difference when the experiment contains non-concurrent or concurrent operations. The small $fsize$ (32 B), amplified the impact of the stable overhead of read-config operations, and they constitute a significant percentage of the total operation latency (see blue bar in Fig. 4.3). From the same figure we interestingly observe that the setup where all servers and clients are deployed in the USA, favored the ADSM algorithms over both Cassandra and Redis.

On the other hand, Cassandra shows different behavior. It achieves the maximum read throughput when both servers and clients are deployed in the EU. It



(a) (b)

Figure 4.4: Filesize Results.

demonstrates a small lead over the ADSM algorithms in most cases on both operations. However, it shows some performance degradation when write and read operations are invoked concurrently.
Finally, REDIS and REDIS_W outperform the rest of the algorithms in all scenarios. REDIS shows consistent performance for both reads and writes due to the weaker consistency requirements and thus smaller communication footprint. The impact of the communication overhead is obvious in REDIS_W, where the writer waits before completing.

Stress Tests – Object Size. This scenario is made to evaluate how the read and write latencies are affected by the size of the shared object. The file size doubled from $64\,\text{kB}$ to $8\,\text{MB}$. The number of servers is fixed to 11. The number of writers and the number of readers is fixed to 5. For ARES, there are two separated runs, one for ARESABD and one for ARESEC. The parity value of ARESEC is set to $m = 5$, and thus the fragmentation parameter is $k = 6$. The quorum size of the ARESEC is $\left\lceil \frac{|S|+k}{2} \right\rceil = \left\lceil \frac{11+6}{2} \right\rceil = 9$, while the quorum size of ARESABD is $\left\lfloor \frac{|S|}{2} \right\rfloor + 1 = \left\lfloor \frac{11}{2} \right\rfloor + 1 = 6$. For CASSANDRA, we set the consistency level (CL) to the majority, i.e., 6. The writers of REDIS_W also wait for a majority (6) servers to reply.

Results. The results for the write performance in these experiments are captured in Fig. 4.4a. We observe that the write latencies of all operations, except ARESEC and REDIS_W, grow significantly, as the $fsize$ increases. The fragmentation applied by the ARESEC benefits its write operations, which follow a slower increasing curve like the REDIS_W. The write latencies of all other algorithms are close to each other. Results in Fig. 4.4b show that the read operations of ARESEC suffer the most delays until $4\,\text{MB}$. The first phase of the read operation does decoding, which is slower than the first phase of the write, which simply finds the maximum tag,

contributed to this overhead. However, at larger file sizes (8 MB) CASSANDRA has the slowest read operations. As expected, the REDIS_W read operations provide the best results, and its write operations with the WAIT command have higher latency compared to the read operations. However, both of them remain at low levels as the $fsize$ increases.

4.5 Conclusions

Our main goal was to establish that non-blocking reconfiguration is feasible and compatible with EC based atomic data storage. Our experimental study in Section 4.3 is designed as a *proof-of-concept* prototype to validate the correctness properties we have developed, and show some benefits.
As a general finding of Section 4.4, achieving strong consistency is more costly than providing weaker semantics as we experienced with REDIS and REDIS_W. However, the performance gap is not prohibitively large and future optimizations of ARES may close it enough so as to substantiate trading performance for consistency. Compared to the atomic version of CASSANDRA, ADSM algorithms seem to scale better, but lack behind in the throughput when dealing with small objects. Both approaches seem to be affected by the object size, but ARESEC suggests that fragmentation may be the solution to this problem. Finally, we demonstrated that ARES may handle efficiently failures in the system, and reconfiguring from one DAP to another without service interruptions. Also, by examining the erasure coding parameter, parity, we have exposed trade-offs between operation latency and fault-tolerance in the system: the further increase of the parity (and thus higher fault-tolerance), the larger the latency.
ARES, an algorithm that always offers provable guarantees, competes closely and in many cases outperforms existing DSS solutions (even when offering weaker consistency guarantees). It would be interesting to study how optimizations may improve the performance of ARES. For example, fragmentation techniques as presented in Chapter 5 have a positive impact on the performance of the algorithm.

Chapter 5

Fragmented ARES: Dynamic Storage for Large Objects

In this chapter we study whether it is plausible to bring coverability and fragmentation strategy of COBFS (cf. Section 3.3) to dynamic environments, and how challenging such adaptation would be. This work is the first to consider dynamic (reconfigurable) DSS tailored for versioned (coverable) and large (fragmentable) objects. At the same time, we aim to introduce solutions that maximize the concurrency of operations on the shared object while trading consistency on the whole object. In particular, we propose a dynamic DSS that: (i) supports versioned objects, (ii) is suitable for large objects (such as files), and (iii) is storage-efficient. To achieve this, we integrate the dynamic ADSM algorithm ARES (cf. Section 4.1) with the DSMM module in COBFS. ARES is the first algorithm that enables erasure coded based dynamic ADSM yielding benefits on the storage efficiency at the replica hosts. To support versioning we extend ARES to implement coverable objects, while high access concurrency is preserved by introducing support for fragmented objects. Ultimately, we aim to make a leap towards dynamic DSS that will be attractive for practical applications (like highly concurrent and strongly consistent file sharing).
In Section 5.1, we propose and prove the correctness of the coverable version of ARES, COARES. In Section 5.2, we present optimization in the implementation of the erasure coded *data-access primitives* (DAP) used by the ARES framework. In Section 5.3, we integrate COARES with COBFS [54] to obtain COARESF. Finally, Section 5.4 shows an overview of the in-depth experimental evaluation we have performed over both Emulab and Amazon Web Services (AWS) EC2.

5.1 COARES: Coverable ARES

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

Description. Below we describe the modification that need to occur on ARES in order to support coverability. The **reconfiguration protocol** and the **DAP imple-**

mentations 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.

Algorithm 4 Write and Read protocols for CoAres.

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

     operation cvr-write(val), val ∈ V
14:    cseq ←read-config(cseq)
       μ ← max({i : cseq[i].status = F})
16:    ν ← |cseq|
       for i = μ : ν do
18:      ⟨τ, v⟩ ← max(cseq[i].cfg.get-data(), ⟨τ, v⟩)
       if version = τ then
20:      flag ← chg
         ⟨τ, v⟩ ← ⟨⟨τ.ts + 1, ω_i⟩, val⟩
22:    else
         flag ← unchg
24:    version ← τ
       done ← false
26:    while not done do
         cseq[ν].cfg.put-data(⟨τ, v⟩)
28:      cseq ←read-config(cseq)
         if |cseq| = ν then
30:        done ← true
         else
32:        ν ← |cseq|
       end while
34:    return ⟨τ, v⟩, flag
     end operation
36:  CVR-Read Operation:
     at each reader r_i
38:  State Variables:
     cseq[]s.t.cseq[j] ∈ C × {F, P}
40:  Initialization:
     cseq[0] = ⟨c_0, F⟩

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

The local variable $flag \in \{chg, unchg\}$, maintained by the write clients, is set to chg when the write operation is successful and to $unchg$ otherwise; initially it is set to $unchg$. The state variable $version$ is used by the client to maintain the tag of the coverable object. At first, in both cvr-read and cvr-write operations, the read/write client issues a read-config action to obtain the latest introduced configuration; cf. line Alg. 4:14 (resp. line Alg. 4:43).

In the case of cvr-write, the writer w_i finds the last finalized entry in $cseq$, say μ, and performs a $cseq[j].conf$.get-data() action, for $\mu \leq j \leq |cseq|$ (lines Alg. 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 Alg. 4:19 - 4:24 depict the main difference between the coverable cvr-write and the original one: if the maximum τ is equal to the state variable $version$, meaning that the writer w_i has the latest version of the object, it proceeds to update the state of the object ($\langle \tau, v \rangle$) by increasing τ and assigning $\langle \tau, v \rangle$ to $\langle \langle \tau.ts + 1, \omega_i \rangle, val \rangle$, where val is the value it wishes to write (lines Alg. 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 first phase (i.e., the write has become a read). No matter whether the state changed or not, the writer updates its $version$ with the value τ (line Alg. 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 Alg. 4:46–47). The propagation of $\langle \tau, v \rangle$ in both cvr-write (lines Alg. 4:26–33) and cvr-read (lines Alg. 4:49–56)) remains the same. Finally, the cvr-write operation returns $\langle \tau, v \rangle$ and the $flag$, whereas the cvr-read operation only returns ($\langle \tau, v \rangle$).

Correctness of CoAres**.** CoAres is correct if it satisfies *liveness* (termination) and *safety* (i.e., linearizable coverability) as indicated in Definitions 3 and 4. Termination holds since read, update and reconfig operations on the CoAres always complete given that the DAP completes. As shown in [16], 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 [16].
We prove the following theorem. The proof is given in [58, 65].

Theorem 9. CoAres *implements a linearizable coverable object, given that the DAPs implemented in any configuration c satisfy Property 1.*

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.

5.2 EC-DAP Optimization

In this section we present an optimization in the implementation of EC-DAP, to reduce the operational latency of the read/write operations in DSMM layer. We show that this optimized EC-DAP, which we refer to as EC-DAPopt, satisfies Property 1 of Section 4.1, and thus can be used by any algorithm that utilizes the DAPs, like any variant of Ares (e.g., CoAres and CoAresF). The specification of the optimized DAP is given in Alg. 5, and the servers' responses in Alg. 6.

Description of EC-DAPopt. The main idea of the optimization (stems from our previous work [54]) is to avoid unnecessary object transmissions between the clients and the servers. Specifically, we apply the following optimization: in the get-data primitive, each server sends only the tag-value pairs with a larger or equal tag than the client's tag. In the case where the client is a reader, it performs the put-data action (propagation phase), only if the maximum tag is higher than its local one. EC-DAPopt is presented in Alg. 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.

Algorithm 5 EC-DAPopt implementation

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

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

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

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

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

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

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

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

Following [16], each server s_i stores a state variable, $List$, which is a set of up to

$(\delta + 1)$ (tag, coded-element) pairs; δ is the maximum number of concurrent put-data operations. In EC-DAPopt, we need another two state variables, the tag of the configuration ($c.tag$) and its associated value ($c.val$). We now proceed with the details of the optimization. Note that the c.get-tag() primitive remains the same as the original.

Primitive c.get-data()***:*** A client, during the execution of a c.get-data() primitive, queries all the servers in $c.Servers$ for their $List$, and awaits responses from $\lceil \frac{n+k}{2} \rceil$ servers. Each server generates a new list ($List'$) where it adds every (tag, coded-element) from the $List$, 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 Alg. 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 Alg. 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 Alg. 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 Alg. 6:17).

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

Theorem 10 (Safety + Liveness)**.** EC*-DAPopt satisfies both conditions of Property 1, and given that no more than δ write operations are concurrent with a read they guarantee that any operation terminates.*

Proof challenges. The complete proof is given in [58, 65]. The **main challenge** of the proof is to show that reducing the values returned by the servers does not violate linearizability, and at the same time, it does not prevent operations from reconstructing the written values, preserving liveness. We prove safety by showing that EC-DAPopt satisfies both conditions of Property 1 in Section 4.1. Particularly, we prove that the tag returned by a get-data() operation is larger than or equal to the tag written by any preceding put-data() operation, and the value returned by a get-data() operation is either written by a put-data() operation or it is the initial value of the object. Liveness is proven by showing that any put-data and get-data operation defined by EC-DAPopt terminates. In the proof, we assume an $[n, k]$ MDS code, $|c.Servers| = n$ of which no more than $\frac{n-k}{2}$ may crash, and that δ is the maximum number of put-data operations concurrent with any get-data

operation. Without this assumption on δ, a get-data operation may not be able to discover a decodable value, and hence fail.

5.3 CoAresF: Integrate CoAres with CoBFS

In this section we describe how CoAres can be integrated with CoBFS to obtain what we call CoAresF, thus yielding a dynamic distributed memory suitable for large objects. Furthermore, this enables to combine the fragmentation approach of CoBFS with a second level of striping when EC-DAP is used, making storage efficient at the servers. A particular challenge of this integration is how the fragmentation approach should invoke reconfiguration operations, since CoBFS 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 Alg. 7, while the specification of reconfig on a fragmented object is given in Alg. 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 the last block by following the next block ids (Alg. 3). The $reconfig$ operation executes the $block\ reconfig$ operations on the shared memory (Alg. 7) using dsmm-reconfig operations.

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
```

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

```
1: State Variables:
2: L_f a sequence of blocks, initially ⟨b_0⟩;
3: function fm-reconfig(c)_{f,p}
4:     b ← val(b_0).ptr
5:     L_f ← ⟨b_0⟩
6:     while b not NULL do
7:         dsmm-reconfig(c)_{b,p}
8:         b ← val(b).ptr
9:     end while
10: end function
```

Correctness of CoAresF. We conclude with the main result of this section.

Theorem 11. CoAresF *implements a* linearizable coverable fragmented object.

Proof Challenges. 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. Also, we show that *fragmented coverability* (see Section 3.2) cannot be violated.

5.4 Higher-Level Experimental Evaluation

We now overview the experimental evaluation we conducted for evaluating our approach. For a more extensive exposition of our experimental evaluation and obtained results, see [65]. The collected data are available in [66], so one could validate our analysis.

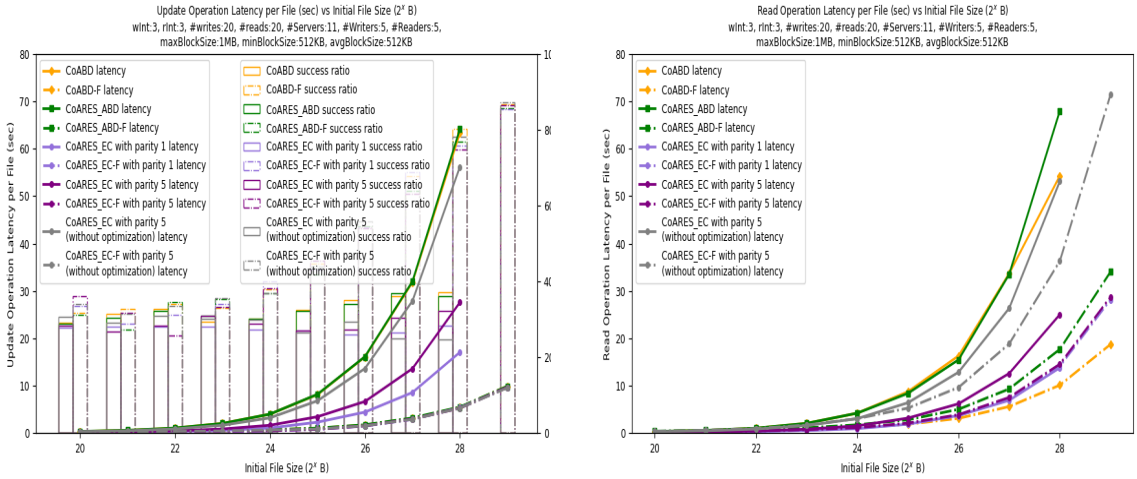
Evaluated Algorithms. We have implemented and evaluated the performance of the following algorithms: CoABD, CoABDF, CoAresABD, AresABDF, AresEC, AresECF.

Distributed Experimental Setup. The experiments were executed on both Emulab [40] [40], and Amazon Web Services (AWS) EC2 [41]. On Emulab we used physical nodes with one 2.4 GHz 64-bit Quad Core Xeon E5530 "Nehalem" processor and 12 GB RAM. While on AWS we used a cluster with 8 nodes of type t2.medium with $4\,\mathrm{GB}$ RAM, 2 vCPUs and $20\,\mathrm{GB}$ storage. For each experiment on Emulab we reported the average over five runs, while AWS experiments run only once. For the evaluation we generate a text file with random byte strings whose size increases as the writers keep updating it.

Experimental Scenarios and Results. Each client in Emulab performs 20 operations and in AWS 50 operations. We used a *stochastic* invocation scheme in which clients pick a random time between the interval $[1...3sec]$ to invoke their next operations. The Emulab results are compiled as averages over five samples per each scenario. However, the AWS results for the scenario presented below were run only once.

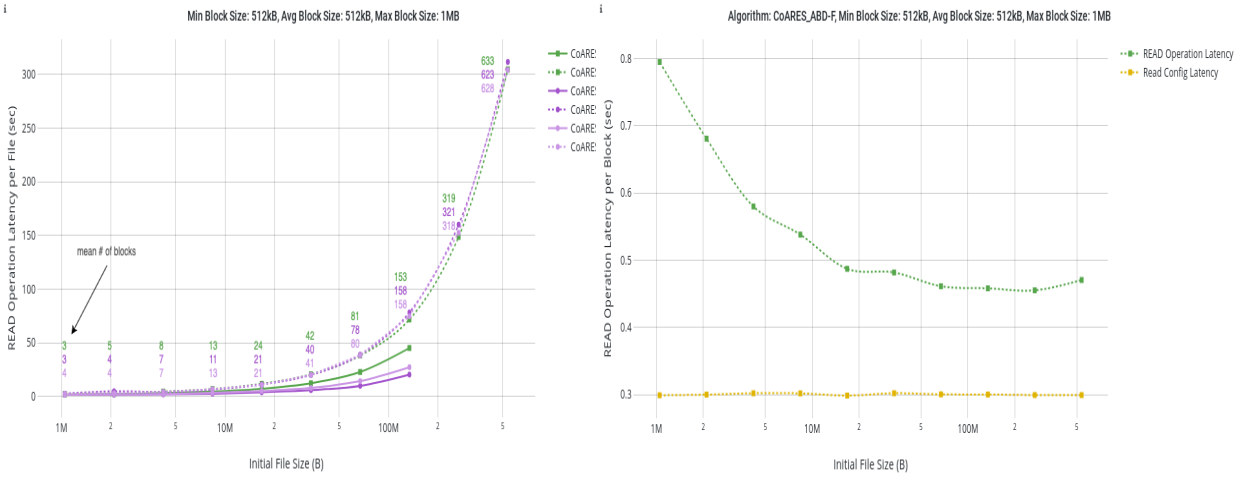
Performance VS. Initial File Sizes. We varied the f_{size} from $1\,\mathrm{MB}$ to $512\,\mathrm{MB}$ by doubling the file size in each experimental run. The performance of some experiments is missing as the non-fragmented algorithms crashed when testing larger file sizes due to an out-of-memory error. For Emulab we used $|\mathcal{W}| = 5, |\mathcal{R}| = 5, |\mathcal{S}| = 11$, while for AWS we used $|\mathcal{W}| = 1, |\mathcal{R}| = 1, |\mathcal{S}| = 6$.

Results. As shown in Fig. 5.1a, the fragmented algorithms on Emulab achieve significantly smaller write latency, since the FM writes only the new and modified



(a) (b)

Figure 5.1: Emulab results for File Size experiments.



(a) (b)

Figure 5.2: AWS results for File Size experiments.

blocks. Also, their success ratio is higher as the file size increases, since the probability of two writes to collide on a single block decreases. The corresponding AWS findings show similar trends.

As shown in Fig. 5.1b, all the fragmented algorithms on Emulab have smaller read latency than the non-fragmented ones. This happens since the readers in the shared memory level transmit only the contents of the blocks that have a newer version. On the contrary, the read latency of CoARES on AWS (Fig. 5.2a has not improved with the fragmentation strategy. The CoARESF operations perform at least two ad-

ditional rounds (compared to COABDF), in order to read the configuration before each of the two phases. Thus, when the FM module sends multiple read block requests, has a significant stable overhead for each block request in the real network conditions of AWS (Fig. 5.2b).
We can also observe from the Figs. 5.1a-5.1b, 5.2a that the further increase of the parity (m) of ARESEC and ARESECF algorithms (and thus higher fault-tolerance) the larger the latency. In addition, the read and write latency of these algorithms when used with EC-DAP are double than of the ones when our optimized DAP (EC-DAPopt) is used.

5.5 Conclusions

In this chapter we have presented COARESF, the first dynamic distributed shared memory that utilizes coverable fragmented objects and enables the use of erasure coding. To achieve this, we developed a coverable version of ARES and integrated it with COBFS. When COARESF is used with an (optimized) Erasure Coded DAP we obtain a two-level striping dynamic and robust distributed shared memory system providing strong consistency and high access concurrency to large objects (e.g., files). We have complemented our development with an overview of our extensive experimental evaluation over the Emulab and AWS testbeds. Compared to the approach that does not use the fragmentation layer of COBFS (COARES), COARESF is optimized with an efficient access to shared data under heavy concurrency.

Chapter 6

Remaining Work

While this work has achieved several promising outcomes, there are still areas that require improvement. In this section, we will explore the choices made during the development of this work and address the challenges. This ongoing research should also be considered part of the remaining work for this thesis. As described in Section 1.2, the remaining work of the thesis is divided into two parts: (i) the identification of bottlenecks in our DSS and exploring proposed solution (Stage 8 in Methodology of Section 1.2), and (ii) the design of a user-friendly interface to assist users in utilizing our DSS (Stage 9 in Methodology of Section 1.2). These steps correspond to Section 6.1 and Section 6.2 respectively.

6.1 Enhance the Performance of our DSS

We introduce several design features in CoBFS: versioning, data striping, high performance, scalability, high access concurrency, and fault-tolerance. In this work, we perform a series of experiments that put these principles to test. We show that even when highly concurrent, mixed read-write workloads are present, CoBFS even in static (CoABDF) or dynamic setting (CoAresF) can sustain a low latency and remains scalable in spite of increasing the number of concurrent clients and the number of servers. 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. However, the more extensive monitoring information we have, the better we can understand the behavior of each system component. A simple approach to know exactly how long each request took, the services and components it used, and how much latency was produced at each phase is to use *Distributed Tracing* [67, 68, 69].

Distributed Tracing is a technique used to monitor and profile distributed systems by tracing individual requests or transactions as they move across multiple components and systems. It involves adding code to the distributed system to gather detailed data on the flow of requests and the behavior of each component. This data is then combined and analyzed to gain a better understanding of the sys-

tem's overall performance and to identify any problems or bottlenecks. In particular, the distributed tracing method creates *traces*, which are records of the activity of individual requests as they pass through various microservices in a distributed system. These traces can be used to diagnose and debug problems in the system, as well as to gain insights into how the system is operating. Also, the individual units of work within a trace are called *spans*. Each span corresponds to a specific piece of work that is performed as a request passes through a microservice. Spans can be used to track the performance of individual components of a system and to identify bottlenecks or other issues that may be impacting system performance.
There are several distributed tracing tools available, including OpenTelemetry [67], Jaeger [68] and Zipkin [69]. OpenTelemetry is an open-source observability framework that provides a standard way to collect telemetry data from distributed systems, including tracing, metrics, and logs. It provides integrations with various backends, such as Jaeger and Zipkin. Both collectors (Jaeger and Zipkin) can store the collected data in-memory, or persistently with a supported backend such as Apache Cassandra or Elasticsearch. For the visualization of collected data into meaningful charts and plots, a proposed tool is called *Grafana Jaeger* [70]. Grafana is a popular open-source platform for visualizing and analyzing data. It has built-in support for Jaeger, allowing for visualization of Jaeger tracing data through Grafana.

Our proposed approach. We chose to instrument our system with OpenTelemetry and send the collected traces to Jaeger for analysis. Additionally, we chose Cassandra as the backend storage for Jaeger in order to persistently store data. This decision was based on our familiarity with Cassandra and our understanding of how to set it up, as presented in Section 4.4. In particular, tracing code will be embedded in our implementation (CoAresF) in order to collect performance data (traces) on experiments conducted on the Emulab testbed. We need to monitor specific procedures of interest (for example, communication overhead or overhead of collecting a set of messages). The data collected will be visualized using Grafana Jaeger, as explained above.
Once bottlenecks are detected, we will propose solutions for eliminating (or at least reducing) the identified performance bottlenecks. These developed solutions should preserve the correctness conditions of the original algorithms. The main objective 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. As mentioned previously, during the experiments we identified several performance bottlenecks. We observed the stable overhead of read-config operations in the experiment sets of Sections 4.4, 5.4. We must find a way to avoid unnecessary read-config operations. Another possible issue of the Ares algorithms is the old or even unviable configurations that exist in $cseq$ in any variant of Ares (cf. Section 4.1). A good solution, retrieved from RAMBO [17], would be to develop a *garbage-collection (CG)* mechanism which is responsible for removing old configurations, by ensuring linearizability, that is, the users observe successively newer values even when the set of nodes that store the object change over time.
Once the proposed enhancements are implemented, we will again deploy on Em-

ulab and utilize distributed tracing. The collected data will be used to generate informative visualizations and essentially compare the performance of the algorithms to the original ones.

6.2 Design a User Interface

Another important requirement for a DSS is the design of a User Interface (UI) though which users can interact with the provided storage service. To create an effective UI for our DSS, we need to consider the users' needs. The UI should be easy to use and navigate, and it should provide all the features of our DSS. Finally, it is important to test our UI with actual users through usability tests and gather feedback to identify any areas for improvement.
It is worth mentioning that in order to enable user interactivity with DSS, we have already developed (i) an in-house command-line interface, and (ii) a Notify API to track file system events. Our Notify API, uses Watchdog, Python API and shell utilities, to monitor a user's specific directory for any new files or file modifications. Based on the user's undertaken actions (e.g., file creation, modification, deletion, renaming, or moving of files from one folder to another), the Notify API triggers the corresponding events. However, in order to allow users to interact with the DSS directly, a user interface (UI) is necessary.

Our proposed approach. To develop our UI, we decided to utilize Laravel [71], which is a cost-less and open-source PHP web application framework employed for constructing web applications. Laravel provides a simple API for working with files and directories. We can use these functions to perform operations on our DSS. To allow the UI to interact with our DSS, we aim to extend our implementation with an API. Thus, the UI can send the actions a user wants to perform on our DSS, such as creating a new file, uploading a file, previewing a file, renaming a file, or deleting it. As we mentioned earlier, we have also the option to perform actions through our command-line interface or our Notify API. So these tools will need to interact will the UI either to retrieve or send data through HTTP requests.
Moreover, Laravel will enable us to ensure security in our DSS by leveraging its built-in authentication system. This system will enable users to create accounts, log in, and access only those files that they are authorized to view. We could define a set of roles (such as "admin", and "user"), and assign each user to one or more of these roles. Then, we could use Laravel's middleware to restrict access to certain parts of our UI based on the user's role. For example, we could define a middleware that only allows users with the "admin" role to access the admin dashboard, or a middleware that restricts access to certain features of our UI based on the user's role. The admin dashboard will provide full control over the UI, allowing an admin to manage files, user accounts, and other important settings related to the system. On the other hand, the user dashboard provides limited control over the DSS, allowing users to access and manage only the files they are authorized to access, based on their permissions. So we also have to define custom permissions for individual files, such as "read-only", "read-write", or "delete". This would allow us to give users more specific access to individual files.

Also, we will need to use a database to manage the metadata associated with the files that are stored in the DSS. For example, we have to create a database table to store information about each file, including its name, path, size, owner, permissions, and other attributes. We could then use Laravel's funtionality, to interact with this database and retrieve or update the metadata associated with each file. We also want to store other data related to the system in the database, such as user accounts, and access permissions.

Chapter 7

Conclusions and Future Work

The current 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 under concurrency. The development of CoBFS, a highly efficient distributed data storage service, 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 proposal. This chapter will focus on highlighting the usefulness of the contributions presented in this work and providing 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.

CoBFS **improves performance of Distributed Shared Memory.** The efficiency of CoBFS is mainly affected by the storage layer. Thank 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 by extensive experimental evaluation. To validate the theoretical principles of each developed algorithms in this work, 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 trade-offs 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 trade-off between splitting the object into smaller blocks, for improving the concurrency in the system, and paying for the cost of sending these blocks in a distributed fashion. Therefore, it is crucial to discover the "golden" spot with the minimum communication delays (while having a large block size) that will ensure a small expected probability of collision (as a parameter of the block size and the delays in the network).

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

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

Future Work

Our DSS may not have been fully developed, highlighting the need for improvement in future studies. In fact, new features can be leveraged to enhance the system's capabilities.

Design Reconfiguration Orchestration Strategies for dynamic DSS. Utilizing the reconfiguration mechanism offered by CoAresF, the 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.
More precisely, this Reconfiguration Orchestration Module (ROM) will function as an external service to CoAresF. Integration of the two services will be accomplished by implementing an API that exposes the reconfiguration operation of CoAresF. An API will also be used for the interaction of the ROM service with the different monitoring tools, which will be deployed to collect the data necessary for the ROM to generate reconfiguration decisions. The integrated service have to be evaluated against three (3) characteristics: (i) scalability in terms of concurrent users and object sizes the service may support, (ii) operation speed (especially compared with our previous implementation), and (iii) fault-tolerance and live-

ness, mainly testing the reaction of the ROM service during harsh environmental conditions.

Server Failure Prediction. The main question that arises in ROM is "when" it is appropriate to modify an established configuration. Infrequent reconfigurations may affect the liveness and thus, availability of the service, while frequent reconfigurations may hinder the service performance. Ideally, the problem could be solved if there could be a way to detect imminent failures of the components in the system. One approach to decide whether servers decline and might fail is by using a storage monitoring service. A storage monitoring service is a tool that provides continuous monitoring of a storage environment, including the performance, capacity, availability, and health of storage systems and devices. In addition, Machine Learning would be used to identify correlations on the metrics so to predict drive failures. This failure prediction must be integrated with service with the Reconfiguration Orchestration Module.

Fully-functional Distributed Storage System. Evolve our system into a distributed storage service imposing strong security guarantees. Security is a crucial feature of any storage service that is intended to facilitate data sharing between untrusted parties. While this aspect is beyond the scope of this work, we acknowledge its importance. Also, it is important to investigate solutions to enforce the proposed security and authentication policies and provide secure access with permissions on dedicated memory cells on the shared memory.

Extensive experimental evaluation. We have already conducted an in-depth experimental comparison of ARES with two established open-source distributed storage solutions. This comparison was fair, as neither of these solutions employed any striping method before the use of shared memory. After optimizing our DSS, it is a good opportunity to compare the final version of COBFS with ARES storage (COARESF) against commercial solutions that employ a striping method, or try to integrate any commercial solution into the fragmentation module of COBFS.

Develop, Deploy, and Evaluate an Enhanced Web Platform. Another future direction involves the design and implementation of a web platform to be used as a portal for the deployment, management, and access of our DSS. The platform will work as a wrapper allowing the users through an intuitive User Interface (UI), compelling to a wide class of users, to: (i) deploy new configurations of the DSS (e.g., a different DAP or a different set of servers) with ROS support by specifying a set of hosting devices; (ii) manage the existing configuration by getting an overview of various service parameters (e.g., hosts, memory size, memory objects, etc.); and (iii) get access to existing configuration for reading and writing data objects either through the platform directly or through a third party application using appropriate security tokens. The platform will be evaluated in terms of the number of supported users, UI scalability as the number of users and configurations grows, and UI usability.

Bibliography

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

[2] P. Viotti and M. Vukolic, "Consistency in Non-Transactional Distributed Storage Systems," *ACM Computing Surveys (CSUR)*, vol. 49, pp. 1 – 34, 2016.

[3] L. Lamport, "On Interprocess Communication, Part I: Basic Formalism," *Distributed Computing*, vol. 1, no. 2, pp. 77–85, 1986.

[4] L. Lamport, "On Interprocess Communication - Part II: Algorithms," *Distributed Computing*, vol. 1, no. 2, pp. 86–101, 1986.

[5] M. P. Herlihy and J. M. Wing, "Linearizability: A Correctness Condition for Concurrent Objects," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 12, no. 3, pp. 463–492, 1990.

[6] S. Gilbert and N. Lynch, "Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services," *SIGACT News*, vol. 33, p. 51–59, jun 2002.

[7] L. Kuper and P. Alvaro, "Toward Domain-Specific Solvers for Distributed Consistency," *Leibniz International Proceedings in Informatics, LIPIcs*, vol. 136, no. 10, pp. 1–10, 2019.

[8] W. Vogels, "Eventually Consistent," *Queue*, vol. 6, no. 6, pp. 14–19, 2008.

[9] "HDFS." `https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html`. Accessed: [08/10/2022].

[10] M. J. Fischer, N. A. Lynch, and M. S. Paterson, "Impossibility of Distributed Consensus with One Faulty Process," *J. ACM*, vol. 32, no. 2, p. 374–382, 1985.

[11] H. Attiya, A. Bar-Noy, and D. Dolev, "Sharing Memory Robustly in Message-Passing Systems," *Journal of the ACM (JACM)*, vol. 42, no. 1, pp. 124–142, 1995.

[12] N. Lynch and A. Shvartsman, "Robust Emulation of Shared Memory Using Dynamic Quorum-Acknowledged Broadcasts," *In Proc. of FTCS*, pp. 272–281, 1997.

[13] C. Georgiou, T. Hadjistasi, N. Nicolaou, and A. A. Schwarzmann, "Implementing Three Exchange Read Operations for Distributed Atomic Storage," *J. Parallel Distributed Comput.*, vol. 163, pp. 97–113, 2022.

[14] P. Dutta, R. Guerraoui, R. Levy, and A. Chakraborty, "How Fast can a Distributed Atomic Read be?," *In Prof. of PODC*, pp. 236–245, 2004.

[15] C. Georgiou, N. Nicolaou, and A. Shvartsman, "Fault-tolerant Semifast Implementations of Atomic Read/Write Registers," *Journal of Parallel and Distributed Computing*, vol. 69, no. 1, pp. 62–79, 2009.

[16] N. Nicolaou, V. Cadambe, N. Prakash, A. Trigeorgi, K. M. Konwar, M. Medard, and N. Lynch, "ARES: Adaptive, Reconfigurable, Erasure coded, Atomic Storage," *ACM Transactions on Storage (TOS)*, 2022. Accepted. Also in `https://arxiv.org/abs/1805.03727`.

[17] S. Gilbert, N. A. Lynch, and A. A. Shvartsman, "RAMBO: A Robust, Reconfigurable Atomic Memory Service for Dynamic Networks," *Distributed Comput.*, vol. 23, no. 4, pp. 225–272, 2010.

[18] M. Aguilera, I. Keidar, D. Malkhi, and A. Shraer, "Dynamic Atomic Storage Without Consensus," in *Proceedings of the 28th ACM Symposium on Principles of Distributed Computing (PODC '09)*, (New York, NY, USA), pp. 17–25, ACM, 2009.

[19] A. Trigeorgi, "A Survey: Robust and Strongly Consistent Distributed Storage Systems." Department of Computer Science, University of Cyprus, 2022.

[20] S. Adve and K. Gharachorloo, "Shared Memory Consistency Models: a Tutorial," *Computer*, vol. 29, no. 12, pp. 66–76, 1996.

[21] R. Fan and N. Lynch, "Efficient Replication of Large Data Objects," *In Proc. of DISC*, pp. 75–91, 2003.

[22] T. Hadjistasi, N. Nicolaou, and A. Schwarzmann, "Oh-RAM! One and a Half Round Atomic Memory," *In Proc. of NETYS*, 2016.

[23] N. Nicolaou, A. Fernández Anta, and C. Georgiou, "Coverability: Consistent Versioning in Asynchronous, Fail-Prone, Message-Passing Environments," in *Proc. of IEEE NCA 2016*, IEEE, 2016.

[24] P. Kuznetsov and A. Tonkikh, "Asynchronous Reconfiguration With Byzantine Failures," *Leibniz International Proceedings in Informatics, LIPIcs*, vol. 179, no. 27, pp. 1–17, 2020.

[25] E. Gafni and D. Malkhi, "Elastic Configuration Maintenance via a Parsimonious Speculating Snapshot Solution," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 9363, pp. 140–153, 2015.

[26] L. Jehl, R. Vitenberg, and H. Meling, "Smartmerge: A New Approach to Reconfiguration for Atomic Storage," in *International Symposium on Distributed Computing*, pp. 154–169, Springer, 2015.

[27] K. Birman, *Reliable Distributed Systems: Technologies, Web Services, and Applications*. Springer, 01 2005.

[28] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The Google File System," *The Google File System*, vol. 53, no. 1, pp. 79–81, 2003.

[29] "Colossus." `https://cloud.google.com/blog/products/storage-data-transfer/a-peek-behind-colossus-googles-file-system`. Accessed: [08/10/2022].

[30] S. Pan, T. Stavrinos, Y. Zhang, A. Sikaria, P. Zakharov, A. Sharma, S. S. P, M. Shuey, R. Wareing, M. Gangapuram, G. Cao, C. Preseau, P. Singh, K. Patiejunas, J. Tipton, E. Katz-Bassett, and W. Lloyd, "Facebook's Tectonic Filesystem: Efficiency from Exascale," in *19th USENIX Conference on File and Storage Technologies (FAST 21)*, pp. 217–231, USENIX Association, Feb. 2021.

[31] "Cassandra." `https://cassandra.apache.org/_/index.html`. Accessed: [08/10/2022].

[32] "Redis." `https://redis.io`. Accessed: [08/10/2022].

[33] A. Carpen-amarie, *BlobSeer as a Data-Storage Facility for Clouds: Self-Adaptation, Integration, Evaluation*. PhD thesis, Université de Rennes, France, 2012.

[34] B. Li, M. Wang, Y. Zhao, G. Pu, H. Zhu, and F. Song, "Modeling and Verifying Google File System," in *2015 IEEE 16th International Symposium on High Assurance Systems Engineering*, pp. 207–214, 2015.

[35] Y. Kalmukov, M. Marinov, T. Mladenova, and I. Valova, "Analysis and Experimental Study of HDFS Performance," *TEM Journal*, vol. 10, pp. 806–814, 2021.

[36] G. Donvito, G. Marzulli, and D. Diacono, "Testing of Several Distributed File-Systems (HDFS, Ceph and GlusterFS) for Supporting the HEP Experiments Analysis," *Journal of Physics: Conference Series*, vol. 513, no. 4, p. 042014, 2014.

[37] L. Beernaert, P. Gomes, M. Matos, R. Vilaça, and R. Oliveira, "Evaluating Cassandra as a Manager of Large File Sets," in *Proceedings of the 3rd International Workshop on Cloud Data and Platforms*, CloudDP '13, (New York, NY, USA), p. 25–30, Association for Computing Machinery, 2013.

[38] "Dropbox." `https://www.dropbox.com/`. Accessed: [08/10/2022].

[39] A. Fernández Anta, T. Hadjistasi, N. Nicolaou, A. Popa, and A. A. Schwarzmann, "Tractable Low-Delay Atomic Memory," *Distributed Computing*, 2020.

[40] "Emulab Network Testbed." `https://www.emulab.net/`. Accessed: [08/10/2022].

[41] "AWS EC2." `https://aws.amazon.com/ec2/`. Accessed: [04/05/2023].

[42] "Fed4fire." `https://www.fed4fire.eu`. Accessed: [04/05/2023].

[43] H. Attiya, "Robust Simulation of Shared Memory: 20 Years After," *Bulletin of the EATCS*, vol. 100, pp. 99–114, 2010.

[44] M. Vukolic, *Quorum Systems: With Applications to Storage and Consensus*. Synthesis Lectures on Distributed Computing Theory, Morgan & Claypool Publishers, 2012.

[45] H. Attiya and J. L. Welch, "Sequential Consistency Versus Linearizability," *ACM T.C.S.*, 1994.

[46] "jFed." `https://jfed.ilabt.imec.be`. Accessed: [04/05/2023].

[47] "ZeroMQ." `https://zeromq.org`. Accessed: [04/05/2023].

[48] "Ansible." `https://www.ansible.com/overview/how-ansible-works`. Accessed: [04/05/2023].

[49] H. Attiya, A. Bar-Noy, and D. Dolev, "Sharing Memory Robustly in Message Passing Systems," *Journal of the ACM*, vol. 42(1), pp. 124–142, 1996.

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

[51] H. Attiya, S. Kumari, A. Somani, and J. L. Welch, "Store-Collect in the Presence of Continuous Churn with Application to Snapshots and Lattice Agreement," 2020.

[52] C. Delporte-Gallet, H. Fauconnier, S. Rajsbaum, and M. Raynal, "Implementing Snapshot Objects on Top of Crash-Prone Asynchronous Message-Passing Systems," *IEEE Trans. Parallel Distrib. Syst.*, vol. 29, no. 9, pp. 2033–2045, 2018.

[53] M. J. Fischer, N. Lynch, and M. Paterson, "Impossibility of Distributed Consensus with one Faulty Process," *Journal of ACM*, vol. 32, no. 2, pp. 374–382, 1985.

[54] A. Anta, C. Georgiou, T. Hadjistasi, E. Stavrakis, and A. Trigeorgi, "Fragmented Object : Boosting Concurrency of Shared Large Objects," *In Proc.of SIROCCO*, pp. 1–18, 2021.

[55] A. Tridgell and P. Mackerras, "The Rsync Algorithm," 1996.

[56] M. O. Rabin, "Fingerprinting by Random Polynomials," 1981.

[57] P. Black, "Ratcliff Pattern Recognition," *Dictionary of Algorithms and Data Structures*, 2021.

[58] C. Georgiou, N. Nicolaou, and A. Trigeorgi, "Fragmented ARES: Dynamic Storage for Large Objects," in *Proceedings of the 36th International Symposium on Distributed Computing (DISC)*, 2022. to appear. Also at arXiv:2201.13292.

[59] I. S. Reed and G. Solomon, "Polynomial Codes Over Certain Finite Fields," *Journal of The Society for Industrial and Applied Mathematics*, vol. 8, pp. 300–304, 1960.

[60] D. Ongaro and J. Ousterhout, "In Search of an Understandable Consensus Algorithm," in *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC'14, (Berkeley, CA, USA), pp. 305–320, USENIX Association, 2014.

[61] "PySyncObj." `https://github.com/bakwc/PySyncObj`. Accessed: [04/05/2023].

[62] "PyEClib." `https://github.com/openstack/pyeclib`. Accessed: [04/05/2023].

[63] A. Trigeorgi, N. Nicolaou, C. Georgiou, T. Hadjistasi, E. Stavrakis, V. Cadambe, and B. Urgaonkar, "Invited Paper: Towards Practical Atomic Distributed Shared Memory: An Experimental Evaluation," in *Stabilization, Safety, and Security of Distributed Systems: 24th International Symposium, SSS 2022, Clermont-Ferrand, France, November 15–17, 2022, Proceedings*, (Berlin, Heidelberg), p. 35–50, Springer-Verlag, 2022.

[64] "Data Repository." `https://github.com/nicolaoun/ngiatlantic-public-data`. Accessed: [04/05/2023].

[65] C. Georgiou, N. Nicolaou, and A. Trigeorgi, "Fragmented ARES: Dynamic Storage for Large Objects," *CoRR*, vol. abs/2201.13292, 2022.

[66] "Data Repository." `https://github.com/atrigeorgi/fragmentedARES-data.git`.

[67] "Opentelemetry." `https://opentelemetry.io`. Accessed: [04/05/2023].

[68] "Jaegertracing." `https://www.jaegertracing.io`. Accessed: [04/05/2023].

[69] "Zipkin." `https://zipkin.io`. Accessed: [04/05/2023].

[70] "Grafana." `https://grafana.com`. Accessed: [04/05/2023].

[71] "Laravel PHP Framework." `https://laravel.com/`, 2011. Accessed: [04/05/2023].