Fragmented Objects: Boosting Concurrency of Shared Large Objects *

Antonio Fernández Anta † Chryssis Georgiou ‡ Theophanis Hadjistasi §
Nicolas Nicolaou§ Efstathios Stavrakis§ Andria Trigeorgi‡

December 5, 2020

Abstract

This work examines strategies to handle *large* shared data objects in distributed storage systems (DSS), while boosting the number of concurrent accesses, maintaining strong consistency guarantees, and ensuring good operation performance. To this respect, we define the notion of *fragmented objects:* concurrent objects composed of a list of fragments (or *blocks*) that allow operations to manipulate each of their fragments individually. As the fragments belong to the same object, it is not enough that each fragment is linearizable to have useful consistency guarantees in the composed object. Hence, we capture the consistency semantic of the whole object with the notion of *fragmented linearizability*. Then, considering that a variance of linearizability, *coverability*, is more suited for versioned objects like files, we provide an implementation of a distributed file system, called CoBFS, that utilizes coverable fragmented objects (i.e., files). In CoBFS, each file is a linked-list of coverable block objects. Preliminary emulation of the CoBFS demonstrates the potential of our approach in boosting the concurrency of strongly consistent large objects.

1 Introduction

In this paper we deal with the storage and use of shared readable and writable data in unreliable distributed systems. Distributed systems (composed of computers and networks that interconnect them) 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 [3]. 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. This preserves the same properties of the larger object (e.g., operations on individual linearizable registers do not violate the consistency of the 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 [3]) to distribute it consistently.

*This work is supported by the Cyprus Research and Innovation Foundation under the grant agreement POST-DOC/0916/0090.

†IMDEA Networks Institute, Madrid, Spain; antonio.fernandez@imdea.org

‡University of Cyprus, Nicosia Cyprus; {chryssis, atrige01}@cs.ucy.ac.cy

§Algolysis Ltd, Limassol, Cyprus; {theo, nicolas, stathis}@algolysis.com

Related work: Attiya, Bar-Noy and Dolev (ABD) [3], proposed an algorithm to emulate a shared distributed R/W register in an asynchronous environment where processes may crash. To provide availability, the object is replicated among a set of object hosts, and to provide operation ordering, a logical timestamp is associated with each written value. Given that less than the majority of hosts may fail, a write operation issued by the sole writer in the system, increments its local timestamp and propagates the value associated with the new timestamp to a majority of hosts. On the other hand, a read operation queries a majority of hosts for their latest timestamp-value pairs, discovers the maximum among them (ordered by timestamp), and before returning it propagates the largest timestamp-value to a majority of hosts. This algorithm was later extended for the multi-writer/multi-reader model in [21], and its performance was later improved by several works, including [11, 16, 17, 13, 15]. Those solutions considered small objects, and relied to the dissemination of the object values in each operation, imposing a performance overhead when dealing with large objects.

Fan and Lynch [12] attempted to reduce performance overheads by separating the metadata of large objects from their value. In this way, communication-demanding operations were performed on the metadata, and large objects were transmitted to a limited number of hosts, and only when it was "safe" to do so. Although this work improved the latency of operations, compared to traditional approaches like [3, 21], it still required to transmit the entire large object over the network per read and write operation. Moreover, if two concurrent write operations affected different "parts" of the object, only one of them would prevail, despite updates not being directly "conflicting."

Recently, Erasure-Coded (EC) approaches have gained momentum and have proved being extremely effective in saving storage and communication costs, while maintaining strong consistency and fault-tolerance [6, 7, 10, 19, 20, 8, 28, 23]. EC approaches rely on the division of a shared object into coded blocks and deliver a single block to each data server. While very appealing for handling large objects, they face the challenge of efficiently encoding/decoding data. Despite being subdivided into several fragments, reads and writes are still applied on the entire object value. Therefore, multiple writers cannot work simultaneously on different parts of an object.

Value continuity is also important when considering large objects, oftentimes overseen by distributed shared object implementations. In files, for example, a write operation should extend the latest written version of the object, and not overwrite any new value. *Coverability* was introduced in [24], as a consistency guarantee that extends linearizability and concerns versioned objects. An implementation of a coverable (versioned) object was presented, where ABD like read operations 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, preventing overwriting a newer version of the object.

Contributions: In this work 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. Consequently, transmission of smaller messages over the network allows for faster operations (message size causes delays in the communication medium) and can boost concurrency as it is possible now for multiple operations to modify different blocks concurrently. In particular, the contributions of this paper are as follows:

- We define two types of concurrent objects: (i) the *block* object, and (ii) the *fragmented* object. Blocks are treated as R/W objects, while fragmented objects are defined as lists of block objects (Section 3).
- We examine the consistency properties when allowing R/W operations on individual blocks of the fragmented object. This enables concurrent modifications on multiple blocks of the fragmented object. Assuming that each block is linearizable, we define the precise consistency that the fragmented object provides, termed *Fragmented Linearizability* (Section 4).
- We provide an algorithm that implements coverable fragmented objects (in particular coverable fragmented files). 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 adopts a modular architecture, separating the object fragmentation process from the shared memory service, which allows to follow different fragmentation strategies and shared memory implementations. We show

that COBFS preserves the validity of the fragmented object and satisfies *fragmented coverability* (Section 5).

- We describe an experimental development and deployment of COBFS on the Emulab testbed [1]. Preliminary results are presented, comparing our proposed algorithm to its non-fragmented counterpart. Results show clear evidence that a fragmented object implementation boosts concurrency while reducing the latency of operations (Section 6).

2 Model

We are concerned with the implementations of highly-available replicated concurrent objects that support a set of operations. 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. Let $\mathcal{I} = \mathcal{C} \cup \mathcal{S}$.

Processors communicate by exchanging messages via asynchronous point-to-point *reliable*[1] channels; messages may be reordered. Any subset of client processes and up to a minority of servers (less than $|\mathcal{S}|/2$), may crash at any time in an execution.

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 none precedes the other.

Consistency: We consider *linearizable* [18] read/write (R/W) objects. A complete history H_ξ is linearizable if there exists some total order on the operations in H_ξ, such that, it respects the real-time order $\rightarrow$ of operations, and is consistent with the semantics of operations.

Notice that we use read and write in a relaxed way: (i) write represents any operation that changes the state of the object, and (ii) read represents any operation that returns that state.

3 Fragmented Objects

A *fragmented object* is a concurrent object (i.e., an object that can be accessed concurrently by multiple processes) that is composed of a finite list of *blocks*. In Section 3.1 we formally define the notion of a block, and in Section 3.2 we give a formal definition of a fragmented object.

3.1 Block Object

A *block* b is a concurrent R/W object with a unique identifier from a set $\mathcal{B}$. A block has a value $val(b) \in \Sigma^*$, extracted from an alphabet Σ. For performance reasons it is convenient to bound the block length. Hence, we denote by $\mathcal{B}^\ell \subset \mathcal{B}$, the set that contains bounded length blocks, s.t. $\forall b \in \mathcal{B}^\ell$ the length of $|val(b)| \leq \ell$. We use $|b|$ to denote the length of the value of b when convenient. An *empty block* is a block b whose value is the empty string ε, i.e., $|b| = 0$.

Operation $\mathsf{create}(b, D)$ 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 $|D| \leq \ell$.

[1] Reliability is not necessary for the correctness of the algorithms we present. It is just used for simplicity of presentation.

A block object is linearizable if is satisfies the linearizability properties [22, 18] with respect to its create (which acts as a write), read, and write operations. Simply put, once created, a block object is an atomic register [22] whose value cannot exceed a predefined length ℓ.

3.2 Fragmented Object

A *fragmented object* f is a concurrent R/W object with a unique identifier from a set $\mathcal{F}$. Essentially, a fragmented object is a *sequence* of blocks from $\mathcal{B}$, with a value $val(f) = \langle b_1, b_2, \ldots, b_n \rangle$, where $b_i \in \mathcal{B}$, for $i \in [1, n]$. Initially, each fragmented object contains an empty block, i.e., $val(f) = \langle b_1 \rangle$ with $val(b_1) = \varepsilon$. We say that $f \in \mathcal{F}^\ell$ if $\forall b_i \in val(f)$, $b_i \in \mathcal{B}^\ell$. If $f \in \mathcal{F}^\ell$ and satisfies the latter condition then we say that f is *valid*; otherwise f is *invalid*.

Being a R/W object, one would expect that a fragmented object $f \in \mathcal{F}^\ell$, for any ℓ, supports the following operations:

- $\mathsf{read}()_f$ returns the list $\langle val(b_1), \ldots, val(b_n) \rangle$, where $val(f) = \langle b_1, b_2, \ldots, b_n \rangle$
- $\mathsf{write}(\langle D_1, \ldots, D_n \rangle)_f$, $|D_i| \leq \ell, \forall i \in [1, n]$, sets the value of f to $\langle b_1, \ldots, b_n \rangle$ s.t. $val(b_i) = D_i, \forall i \in [1, n]$.

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 $val(f) = \langle b_1, b_2 \rangle$, $val(b_1) = D_1$, $val(b_2) = D_2$, and assume that ξ contains two concurrent write operations by two different clients, one attempting to modify block b_1, and the other attempting to modify block b_2: $\pi_1 = \mathsf{write}(\langle D'_1, D_2 \rangle)_f$ and $\pi_2 = \mathsf{write}(\langle D_1, D'_2 \rangle)_f$, followed by a $\mathsf{read}()_f$ operation. By strong consistency (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_1 and π_2 updating the value of b_2. To this respect, we redefine the write operation to only update *one* of the blocks of a fragmented object. Since the update operation does not manipulate the value of the whole object, which would include also new blocks to be written, so it should allow the update of a block b with a value $|D| \geq \ell$. This essentially leads to the generation of new blocks in the sequence. More formally the update operation is defined as following:

- $\mathsf{update}(b_i, D)_f$ 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 [4]. 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. So, in the rest of the paper 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. In particular, the $\mathsf{read}()_f$ performs a sequence of read block operations (starting from the genesis block 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 $i = |f|$). We can now define the sequential specification of a fragmented object.

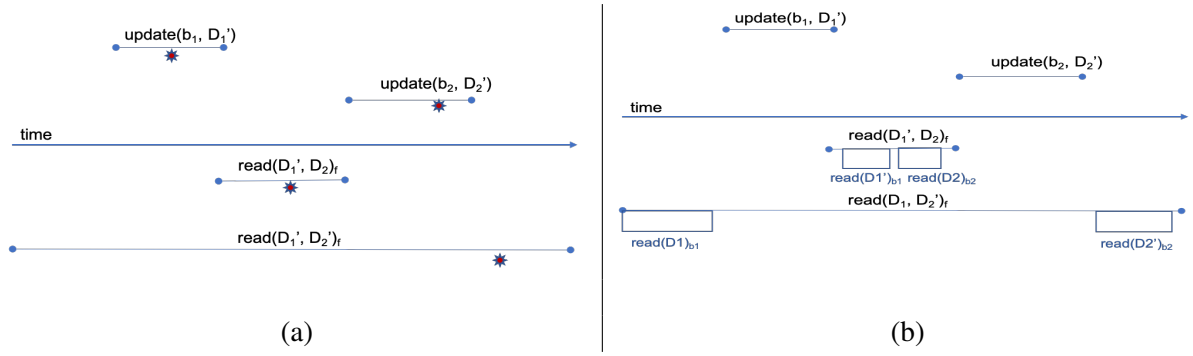


Figure 1: Executions showing the operations on a fragmented object. Figure (a) shows linearizable reads on the fragmented object (and the serialization points), and (b) shows reads on the fragmented object that are implemented with individual linearizable reads on blocks.

Definition 1 (Sequential Specification) *The* sequential specification *of a fragmented object $f \in \mathcal{F}^{\ell}$ over the complete sequential history H is defined as follows. Initially $val(f) = \langle b_1 \rangle$ with $val(b_1) = \varepsilon$. If at the invocation action of an operation π in H have $val(f) = \langle b_1, \ldots, b_n \rangle$ 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 $\langle (b_1, val(b_1)), \ldots, (b_n, val(b_n)) \rangle$. At the response action of π still holds that $val(f) = \langle b_1, \ldots, b_n \rangle$ 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 $|D| \leq \ell$: $val(f) = \langle b_1, \ldots, b_n \rangle$, $val(b_i) = D$;*
  - *if $|D| > \ell$: $val(f) = \langle b_1, \ldots, b_i, b_i^1, \ldots, b_i^k, b_{i+1}, \ldots, b_n \rangle$, such that $val(b_i) = D^0$ and $val(b_i^j) = D^j, \forall j \in [1, k]$, where $D = D^0|D^1| \cdots |D^k$ and $|D^j| \leq \ell, \forall j \in [0, k]$.*[2]

4 Fragmented Linearizability

A Fragmented Object is linearizable if it satisfies both the *Liveness* and *Linearizability* (Atomicity) properties [22, 18]. 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. *Liveness* property states that any read and update operation on the fragmented object should terminate. It remains to examine the consistency properties.

Linearizability: Let H_{ξ} be a sequential history of update and read invocations and responses on a fragmented object f. Linearizability [22, 18] provides the illusion that the fragmented object is accessed sequentially respecting the real-time order, even when operations are invoked concurrently[3]:

Definition 2 (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 every pair of 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 2, the operations must be totally ordered. Let us consider again the sample execution ξ from Section 3. Since we decided not to use write operations, the execution

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

[3]Our formal definitions of linearizability is adapted from [5].

changes as follows. Initially, $val(f) = \langle b_1, b_2 \rangle$, $val(b_1) = D_1$, $val(b_2) = D_2$, and then ξ contains two concurrent update operations by two different clients, one attempting to modify the first block, and the other attempting to modify the second block: $\pi_1 = \mathsf{update}(b_1, D'_1)_f$ and $\pi_2 = \mathsf{update}(b_2, D'_2)_f$ ($|D'_1| \leq \ell$ and $|D'_2| \leq \ell$), followed by a $\mathsf{read}()_f$ operation. In this case, since both update operations operate on different blocks, independently how π_1 and π_2 are ordered in the permutation σ, the $\mathsf{read}()_f$ operation will return $\langle D'_1, D'_2 \rangle$. Therefore, the use of these update operations has increased the concurrency in the fragmented object.

Using linearizable read operations on the entire fragmented object can ensure the linearizability of the fragmented object as can be seen in the example presented in Figure 1(a). However, providing a linearizable read when the object involves multiple R/W objects (i.e., an atomic snapshot) can be expensive or impact concurrency [9]. 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. 1(b), two reads concurrent with two update operations may violate linearizability on the entire object. According to the real time ordering of the operations on the individual blocks, block linearizability is preserved if the first read on the fragmented object should return (D'_1, D_2), while the second read returns (D_1, D'_2). Note that we cannot find a permutation on these concurrent operations that follows the sequential specification of the fragmented object. More precisely, if we order $\mathsf{update}(b_1, D'_1)$ before $\mathsf{update}(b_2, D'_2)$, we cannot order $\mathsf{read}(D_1, D'_2)$ after $\mathsf{update}(b_2, D'_2)$. Similarly, if we order $\mathsf{update}(b_2, D'_2)$ before $\mathsf{update}(b_1, D'_1)$, we cannot order $\mathsf{read}(D'_1, D_2)$ after $\mathsf{update}(b_1, D'_1)$. Thus, the execution in Figure 1(b) violates linearizability. This leads to the definition of *fragmented linearizability* on the fragmented object, which relying on the fact that *each individual block is linearizable*, it allows executions like the one seen in Fig. 1(b). Essentially, fragmented linearizability captures the consistency one can obtain on a collection of linearizable objects, when these are accessed concurrently and individually, but under the "umbrella" of the collection.

In this respect, we specify each $\mathsf{read}()_f$ 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 $\langle (b_1, val(b_1)), \ldots, (b_n, val(b_n)) \rangle$ is specified by n individual read operations $\mathsf{read}()_{b_1}$,..., $\mathsf{read}()_{b_n}$, that return $val(b_1)$, ..., $val(b_n)$, respectively, where $\mathsf{read}()_{b_1} \rightarrow, \ldots, \rightarrow \mathsf{read}()_{b_n}$.

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 invocation 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 an execution $H|_f$ 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 $val(f) = \langle b_1, \ldots, b_n \rangle$ is the value returned by the read operation, then we replace the invocation of $\mathsf{read}()_{b_1}$ 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 3 (Fragmented Linearizability) *Let $f \in \mathcal{F}^{\ell}$ 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* [4]*, and*
- *for every pair of 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.*

In other words, fragmented linearizability guarantees that all concurrent operations on different blocks prevail, and only concurrent operations on the same blocks are conflicting. Furthermore, consider two read_f operations, r_1 and r_2, such that the response action of the last block read in r_1 is before the invocation action of the first block read in 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.

[4]The sequential specification of a block is similar to that of a R/W register [22], whose value has bounded length.

5 Implementing Files as Fragmented Coverable Objects

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

When manipulating files it is expected that a value update builds upon 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. In order to maintain this property in our implementation we utilize *coverable linearizable* blocks as presented in [24]. 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. Due to space limitations we refer the reader to [24] and in the optional Appendix for the exact coverability properties.

By utilizing coverable blocks, our file system provides *fragmented coverability* as a consistency guarantee (thus satisfying the properties in Definitions 3 and coverablity properties of Definition 9). In our prototype implementation we consider each object to be a plain text file, however the underlying theoretical formulation allows for extending this implementation to support any kind of large objects. Due to lack of space, any omitted proofs appear in the Appendix.

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

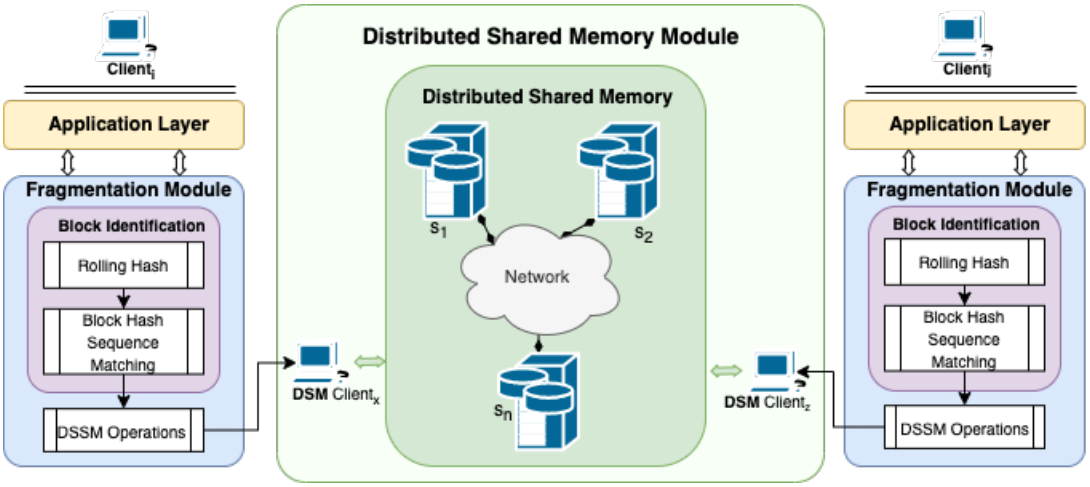


Figure 2: Basic architecture of CoBFS

Overview of the Basic Architecure: The basic architecture of CoBFS appears in Fig. 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 $f \in \mathcal{F}$ 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.

Algorithm 1 Distributed Shared Memory Module: Operations on a coverable block object b at client p

1: **State Variables:**
2: $ver_b \in \mathbb{N}$ initially 0; $val_b \in V$ initially $\perp$;

3: **function** dsmm-read( )$_{b,p}$
4: $\langle val_b, ver_b \rangle \leftarrow b$.cvr-read()
5: **return** val_b
6: **end function**

7: **function** dsmm-create$(val)_{b,p}$
8: $\langle val_b, ver_b \rangle \leftarrow b$.cvr-write$(val, 0)$
9: **end function**

10: **function** dsmm-write$(val)_{b,p}$
11: $\langle val_b, ver_b \rangle \leftarrow b$.cvr-write$(val, ver_b)$
12: **return** val_b
13: **end function**

Algorithm 2 Optimized coverable ABD (read operation)

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

4: **function** cvr-read( )
5: **send** $\langle$READ, $ver_b\rangle$ to all servers ▷ Query Phase
6: **wait until** $\frac{|\mathcal{S}|+1}{2}$ servers reply
7: $maxP \leftarrow \max(\{\langle tg', v' \rangle$ received from some server$\})$
8: **if** $maxP.tg > tg_b$ **then**
9: **send** (WRITE, $maxP$) to all servers ▷ Propagate Phase
10: **wait until** $\frac{|\mathcal{S}|+1}{2}$ servers reply
11: $\langle tg_b, val_b \rangle \leftarrow maxP$
12: **end if**
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 $\langle 0, \perp \rangle$; $val_b \in V$, initially $\perp$

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: **end if**
22: **if** $M.type =$ READ and $M.tg \geq tg_b$ **then**
23: send$(\langle tg_b, \perp \rangle)$ to q ▷ Reply without content
24: **else**
25: send$(\langle tg_b, val_b \rangle)$ to q ▷ Reply with content
26: **end if**
27: **end function**

In turn, a block $b \in \mathcal{B}$ of a file is identified by a triplet $\langle fid, cid, cseq \rangle \in \mathcal{F} \times \mathcal{C} \times \mathbb{N}$, where $fid \in \mathcal{F}$ is the identifier of the file in which the block belongs to, $cid \in \mathcal{C}$ 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 it 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 at its core implements a distributed R/W shared memory based on an *optimized coverable variant* of the ABD algorithm, we call CoABD, presented in [24]. 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, waits for that operation to complete, and returns the value received from that operation.

To reduce the number of blocks transmitted per read operation, we apply a simple yet very effective optimization (see Algorithm 2): a read operation 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 the local tag of the server; otherwise the server replies with its local tag without the block content (as the reader knows a more recent version of the block). The reader waits for a majority of servers to reply. It detects the maximum tag among the replies, and checks if that tag is higher than the local known tag. If it is then it forwards (as in regular ABD) 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. Notice that 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 operation will be sent over to the servers, resulting in significant reductions in case of large objects.

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, while the write uses the incremented local version in an attempt to write a new version of b. Notice that the write in create will always succeed as it will introduce a new, never been 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 [24] for the implementation of cvr-read and cvr-write, which are simple variants of the corresponding implementations of ABD [3]. Thus we may conclude with the following lemma:

Lemma 4 *The DSMM implements R/W coverable block objects.*

Proof.[Proof Sketch] When both the read and write operations perform two phases the correctness of the algorithm is derived from Theorem 10 in [24]. It is easy to see that the optimization does not violate linearizability. The second phase of a read is omitted when all the servers reply with a tag smaller or equal to the local tag of the reader r. Since however, a read propagates its local tag to a majority of servers at every tag update, then every subsequent operation will observe (and return) the latest value of the object to be associated with a tag at least as high as the local tag of r. □

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 FM contains a *Block Identification (BI) module* that utilizes known approaches for data fragmentation and diff extraction. Next, we explain the process for *BI* and present the operations offered by the FM in more detail.

Algorithm 3 Fragmentation Module: Block Identification (BI) and Operations on a file f at client p

```
1:  State Variables:
2:  H initially ∅; ℓ ∈ ℕ;
3:  L_f a linked-list of blocks, initially ⟨b_g⟩;
4:  bc_f ∈ ℕ initially 0;

5:  function fm-block-identify( )_{f,p}
6:      ⟨newD, newH⟩ ← RabinFingerprints(f, ℓ)
7:      curH = hash(L_f)
8:                          ▷ hashes of the data of the blocks in L_f
9:      C ← SMatching(curH, newH)
10:                                                  ▷ modified
11:     for ⟨h(b_j), h_k⟩ ∈ C.mods s.t. h(b_j) ∈ curH, h_k ∈ newH do
12:         D ← {D_k : D_k ∈ newD ∧ h_k = hash(D_k)}
13:         fm-update(b_j, D)_{f,p}
14:     end for
15:                                                  ▷ inserted
16:     for S ∈ C.inserts s.t. h_i ∈ S are in sequence do
17:         D ← {D_i : h_i ∈ S ∧ D_i ∈ newD ∧ h_i = hash(D_i)}
18:         b ← b_j s.t. ∀h_i ∈ S inserted after h(b_j)
19:         fm-update(b, D)_{f,p}
20:     end for
21: end function

22: function fm-read( )_{f,p}
23:     b ← val(b_g).ptr
24:     L_f ← ⟨b_g⟩                                  ▷ reset L_f
25:     while b not NULL do
26:         val(b) ← dsmm-read()_{b,p}
27:         L_f.insert(val(b))
28:         b ← val(b).ptr
29:     end while
30:     return Assemble(L_f)
31: end function

32: function fm-update(b, D = ⟨D_0, D_1, ..., D_k⟩)_{f,p}
33:     for j = k : 1 do
34:         b_j ← ⟨f, p, bc_f++⟩                     ▷ set block id
35:         val(b_j).data = D_j                      ▷ set block data
36:         if j < k then
37:             val(b_j).ptr = b_{j+1}               ▷ set block ptr
38:         else
39:             val(b_j).ptr = val(b).ptr
40:                                                  ▷ point last to b ptr
41:         end if
42:         L_f.insert(val(b_j))
43:         dsmm-create(val(b_j))_{b_j}
44:     end for
45:     val(b).data = D_0
46:     if k > 0 then
47:         val(b).ptr = b_1                         ▷ change b ptr if |D| > 1
48:     end if
49:     dsmm-write(val(b))_b
50: end function
```

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. Furthermore, by drawing

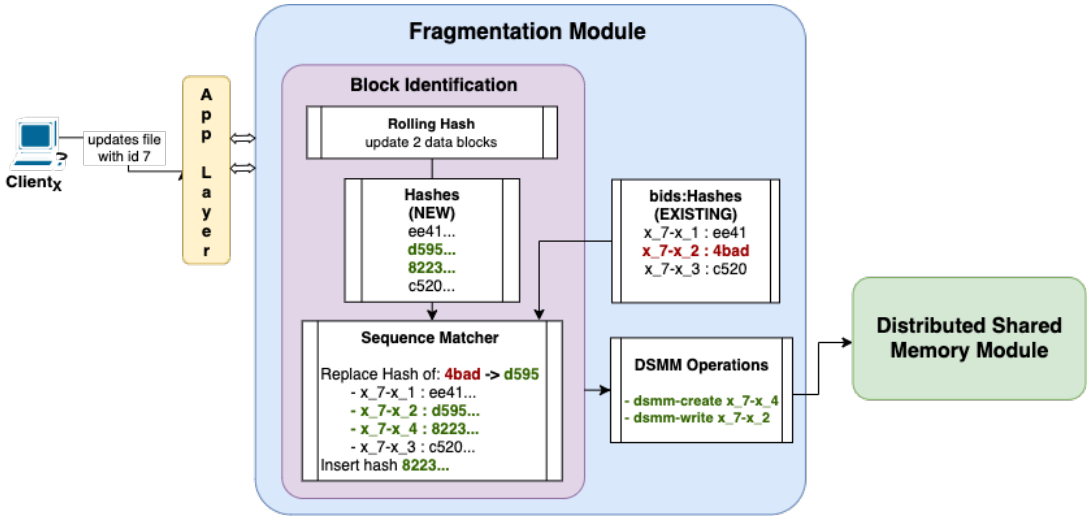


Figure 3: An example of text insertion which changes the block structure.

ideas from the RSYNC (Remote Sync) algorithm [26], 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 [25], and (2) it compares the hashes of the blocks of the current and the previous version of the file using a string matching algorithm [2] to determine the modified/new data blocks. The role of BI within the architecture of CoBFS and its process flow appears in Fig. 2, while its specification is provided in Algorithm 3 (function fm-block-identify()). 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 algorithm identifies the block boundaries and it performs content-based chunking by calculating and returning the fingerprints (block hashes) over a sliding window, and guarantees that each block identified has a bounded size of no more than ℓ.

- **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 $\langle b_0, \ldots, b_n \rangle$. More precisely, we produce the vector $\langle h(b_0), \ldots, h(b_n) \rangle$ where $h(b_i) = hash(val(b_i).data)$ from the current blocks of f, and using a string matching algorithm [2] we compare the two hash vectors to obtain one of the following statuses for each vector 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 $\mathsf{update}(b_j, \{D_i\})_{f,p}$ action is then issued to modify the data of b_j to D_i (Lines 10:13). 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 $\mathsf{update}(b_j, \{val(b_j).data, D_i, D_k\})_{f,p}$ is performed to create the new blocks after b_j (Lines 15: 19). In our formulation block deletion is treated as a modification that sets an empty data value (e.g., an empty string); thus, in our implementation *no blocks are deleted.*

The schematic in Fig. 3 shows an example of a writer process with id x writing new text at the beginning of the second block of a text file with id $f_{id} = 7$. Notice how 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, extending the second block. Lastly, the modified block with b_{id} = x_7-x_2 and the new block with b_{id} = x_7-x_4 are sent to the DSM.

FM Operations: The FM's external signature includes the two main operations of a fragmented object: read_f, and update_f. The specifications of the two operations appear in Algorithm 3.

Read operation - $\mathsf{read}()_{f,p}$**:** To retrieve the value of a file f, a client p may invoke a $\mathsf{read}_{f,p}$ request to the fragmented object. Upon receiving such a request the FM issues a series of read operations 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 $\mathsf{Assemble}()$ function. Notice that the reader p issues a read operation for all the blocks in the file. This is done to ensure the property stated in the following lemma:

Lemma 5 *Let ξ be an execution of* CoBFS *with two read operations $\rho_1 = \mathsf{read}_{f,p}$ and $\rho_2 = \mathsf{read}_{f,q}$ from clients p and q on the fragmented object f, s.t. $\rho_1 \rightarrow \rho_2$. If ρ_1 returns a list of blocks $\mathcal{L}_1$ and ρ_2 a list $\mathcal{L}_2$, then $\forall b_i \in \mathcal{L}_1$, then $b_i \in \mathcal{L}_2$ and $version(b_i)_{\mathcal{L}_1} \leq version(b_i)_{\mathcal{L}_2}$.*

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*. This is captured by the following lemma:

Lemma 6 *In any execution ξ of* CoBFS*, if ξ contains an $\pi = \mathsf{update}(b, D)_{f,p}$, then π is successful iff the operation b.*cvr-write *called within* dsmm-write$(val(b))_{b,p}$*, is successful.*

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

Lemma 7 *In any execution ξ of* CoBFS*, if a $\mathsf{read}_{f,p}$ operation returns a list $\mathcal{L} = \langle b_g, b_1, \ldots, b_n \rangle$ for a file f, then $val(b_g).prt = b_1$, $val(b_i).ptr = b_{i+1}$, for $1 \leq i < n - 1$, and $val(b_n).ptr = \perp$.*

This leads us to the following:

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

Proof. By Lemma 4 every block operation in CoBFS satisfies coverability and together with Lemma 5 it follows that CoBFS implements a coverable fragmented object satisfying the properties presented in Definitions 3 and 9. Also, the BI ensures that the size of each block is limited under a bound ℓ and Lemma 7 ensures that each operation obtains a connected list of blocks. Thus, CoBFS implements a *valid* fragmented object. □

To enhance the practicality of our prototype and to bring it closer to a file system, we have implemented additional operations, which are all framed around the two main operations of the FM. Due to lack of space, a high-level description of these operations can be found in the Appendix.

6 Preliminary Evaluation

To further appreciate the potential of the proposed approach from an applied/practical point of view, we performed a preliminary evaluation of CoBFS against the CoABD. Notice that 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 as discussed in Algorithm 2 analysis.

Both approaches were implemented and deployed on *Emulab,* [27], a network testbed with tunable and controlled environmental parameters.

Experimental Setup: Across all experiments, three distinct types of distributed nodes are defined and deployed within the emulated network environment as listed below. Communication between the distributed nodes is via point-to-point bidirectional links implemented with a DropTail queue.

- **writer** $w \in W \subseteq C$ **:** a client that dispatches update requests to servers.
- **reader** $r \in R \subseteq C$**:** a client that dispatches read requests to servers
- **server** $s \in S$**:** listens for reader and writer requests and is responsible for maintaining the object replicas according to the underlying atomic shared memory they implement.

Performance Metrics: We measured performance using two metrics: (i) *operational latency*, and (ii) *the update success ratio*. The operational latency is computed as the sum of communication and computation delays. In the case of CoBFS, computational latency encompasses the time necessary for the FM to fragment a file object and generate the respective hashes for its blocks. The update success ratio is the percentage of update operations that have not been converted to reads (and thus successfully changed the value of the indented block). In the case of CoABD, we compute the percentage of successful updates on the whole file over the number of all update requests by all writers. Based on the same method, we compute the equivalent percentage of successful block updates for the CoBFS.

Scenarios: Both algorithms are evaluated under the following experimental scenarios:

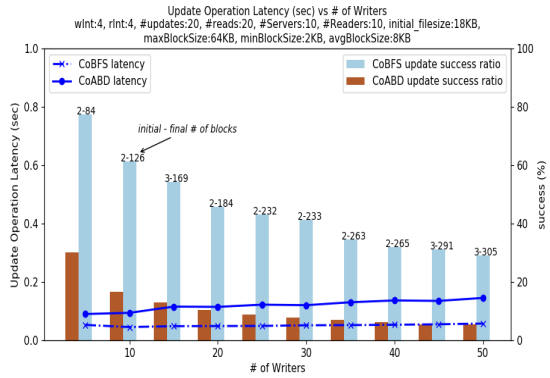
- **Scalability:** examine performance under various numbers of readers/writers/servers (CoABD, CoBFS)
- **File Size:** examine performance when using different initial file sizes (CoABD, CoBFS)
- **Block Size**: examine performance under different block sizes (CoBFS only)

We use a *stochastic* invocation scheme in which reads are scheduled randomly from the intervals $[1...rInt]$ and updates from $[1..wInt]$, where $rInt, wInt = 4sec$. Each experiment was repeated multiple times for better estimations. During all the experiments of each scenario, as the writers kept updating the file, its size increased.

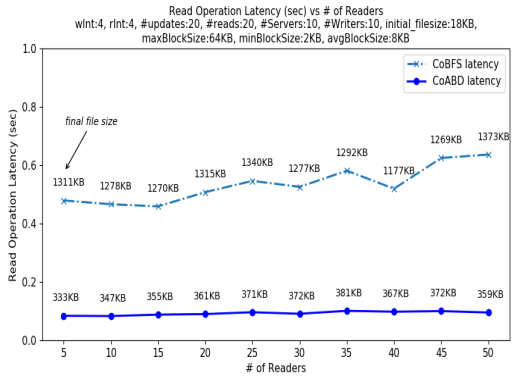
Scalability Experiments: We varied the number of readers $|R|$, the number of writers $|W|$, and the number of servers $|S|$ in the set $\{5, 10, 15, 20, 25, 30, 35, 40, 45, 50\}$. While testing for readers' scalability, the number of writers and servers was kept constant, $|W|, |S| = 10$. Using the same approach, scalability of writers, and in turn of servers, was tested while preserving the two other types of nodes constant (i.e. $|R|, |S| = 10$ and $|R|, |W| = 10$ respectively). In total, each writer performed 20 updates and each reader 20 reads. The size of the initial file used was set to $18\,\mathrm{kB}$, while the maximum, minimum and average block sizes (*rabin fingerprints* parameters) were set to $64\,\mathrm{kB}$, $2\,\mathrm{kB}$ and $8\,\mathrm{kB}$ respectively.

File Size Experiments: 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 (*rabin fingerprints* parameters) were set to $1\,\mathrm{MB}$, $512\,\mathrm{kB}$ and $512\,\mathrm{kB}$ respectively.

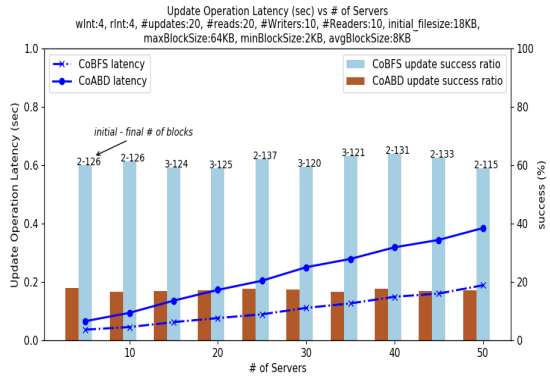
Block Size Experiments: We varied the minimum and average b_{sizes} of CoBFS from $1\,\mathrm{kB}$ to $64\,\mathrm{kB}$. The number of writers, readers and servers was fixed to 10. In total, each writer performed 20 updates and each reader 20 reads. The size of the initial file used was set to $18\,\mathrm{kB}$, while the maximum block size was set to $64\,\mathrm{kB}$ (*rabin fingerprints* parameter).



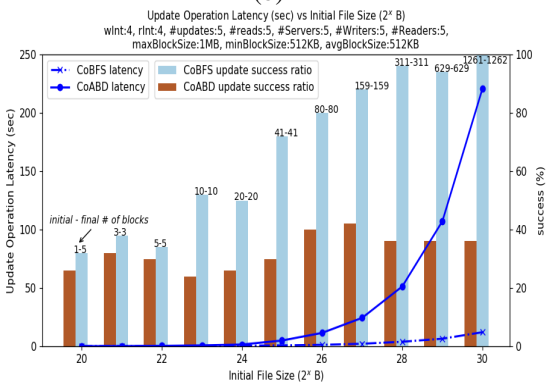
(a)



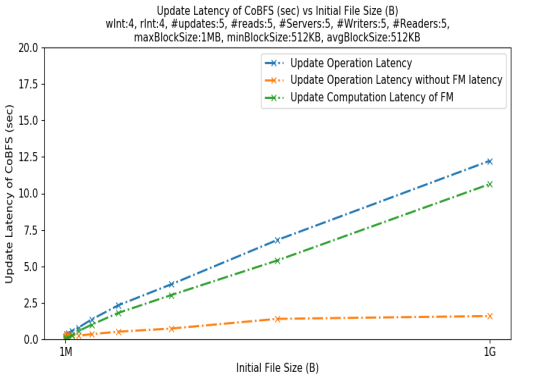
(b)



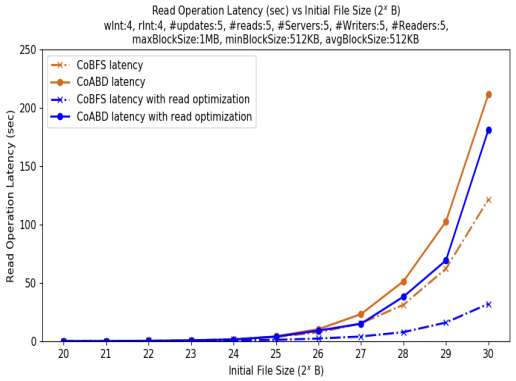
(c)



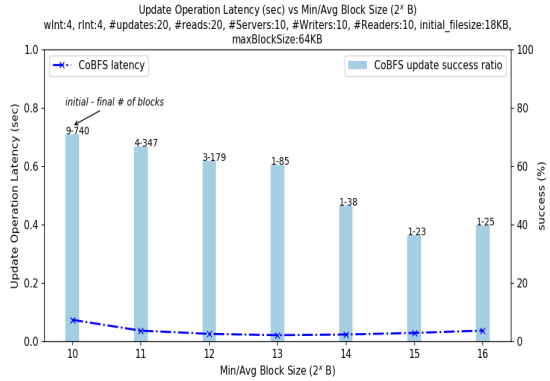
(d)



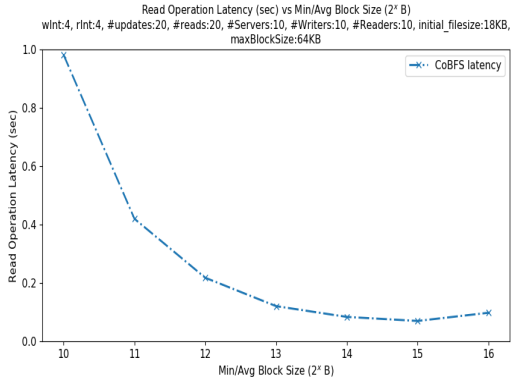
(e)



(f)



(g)



(h)

Figure 4: Simulation results for algorithms CoABD and CoBFS.

Results: Overall, our results suggest that the efficiency of CoBFS is inversely proportional to the number of block operations, rather than the size of the file. This is primarily due to the individual block-processing nature of CoBFS. We can reach the following observations based on the plots in Fig. 4.

Scalability: In Fig. 4(a), the operational latency of updates in CoBFS remains almost unchanged and smaller than the one of CoABD. This is mainly due to the fact that each CoABD writer updates a rather small file (18 kB), while each CoBFS writer updates a subset of blocks which are modified or created. The computational latency of FM in CoBFS is negligible, when compared to the total update operation latency, because of the small file size. In Fig. 4(c), we observe that the update operation latency in CoABD increases even more as the number of servers increases.

As more updates are successful in CoBFS, reads may transfers more data compared to reads in CoABD, explaining their slower completion as seen in Fig. 4(b). Also, readers send multiple read block requests of small sizes, waiting each time for a reply, while CoABD readers wait only one message containing a small file (18 kB). It would be interesting to examine whether the multiple read block requests in CoBFS could be sent in parallel, reducing the overall communication delays.

Concurrency: The percentage of successful file updates achieved by CoBFS are significantly higher than those of CoABD. This holds for both cases where the number of writers increased (see Fig. 4(a)) and the number of servers increased (see Fig. 4(c)). This demonstrates the boost of concurrency achieved by CoBFS. Furthermore, in Fig. 4(a) 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 like in Fig. 4(d). The probability of two writes to collide on a single block decreases, and thus CoBFS eventually allows all the updates (100%) to succeed. On the contrary, CoABD does not experience any improvement as it always manipulates the file as a whole.

File Size: Figure 4(d) 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, as shown in Fig. 4(e). It is worth mentioning that 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 initial 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 operation latencies of the two algorithms with and without the read optimization can be found in Fig. 4(f). The CoABD read latency increases sharply, even when using the optimized read operations. This is inline with our initial hypothesis, as CoABD requires read operations 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 read latency is close of that of CoABD. Notice that each read operation that discovers a new version of the file needs to request and propagate the content of each individual block. On the other hand, read optimization decreases significantly the CoBFS read latency, as reads transmit only the contents of the blocks that have changed.

Block Size: From Figs. 4(g)(h) we can infer that when smaller blocks are used, the update and read operation 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. 4(g), the small b_{size} leads to the generation of more new blocks during `update` operations, resulting in more update block operations, and hence higher operational latencies. As the minimum and average b_{sizes} increase, lower number of blocks need to be added when an `update` operation is taking place. Unfortunately, smaller number of blocks leads to a lower success rate. Similarly, in Fig. 4(h), 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. Therefore, 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 it does not give better (or worse) latencies.

7 Conclusions

In this work we have introduced the notion of linearizable and coverable fragmented objects and proposed an algorithm to implement coverable fragmented files. This algorithm is 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 show that our implementation 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 and an evaluation platform. The evaluation conducted demonstrates the potential of our approach in boosting the concurrency and improving the efficiency of read/write operations on strongly consistent large objects.

Our future plans include devising more algorithms to implement linearizable and coverable fragmented objects, some of which can be integrated in CoBFS. It would be also interesting to examine the performance of CoBFS when using more efficient algorithms to implement the distributed shared memory object. We also want to do a comprehensive and in more depth experimental evaluation of CoBFS (that will go beyond simulations, e.g., full-scale, real-time, cloud-based experimental evaluations), as well as many optimizations and extensions, in an effort to unlock in full the potential of our approach.

References

[1] Emulab network testbed. `https://www.emulab.net/`.

[2] String matching algorithm. `https://xlinux.nist.gov/dads/HTML/ratcliffObershelp.html`.

[3] H. Attiya, A. Bar-Noy, and D. Dolev. Sharing memory robustly in message passing systems. *Journal of the ACM*, 42(1):124–142, 1996.

[4] 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. `arXiv:2003.07787`.

[5] H. Attiya and J. L. Welch. Sequential consistency versus linearizability. *ACM Trans. Comput. Syst.*, 12(2):91–122, 1994. `doi:http://doi.acm.org/10.1145/176575.176576`.

[6] C. Cachin and S. Tessaro. Optimal resilience for erasure-coded byzantine distributed storage. pages 115–124, Los Alamitos, CA, USA, 2006. IEEE Computer Society. `doi:http://doi.ieeecomputersociety.org/10.1109/DSN.2006.56`.

[7] V. R. Cadambe, N. A. Lynch, M. Médard, and P. M. Musial. A coded shared atomic memory algorithm for message passing architectures. *Distributed Computing*, 30(1):49–73, 2017.

[8] Yu Lin Chen Chen, Shuai Mu, and Jinyang Li. Giza: Erasure coding objects across global data centers. In *Proc. of USENIX ATC '17*, pages 539–551, 2017.

[9] 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.*, 29(9):2033–2045, 2018. `doi:10.1109/TPDS.2018.2809551`.

[10] P. Dutta, R. Guerraoui, and R. R. Levy. Optimistic erasure-coded distributed storage. In *DISC '08*, pages 182–196, Berlin, Heidelberg, 2008. Springer-Verlag. `doi:http://dx.doi.org/10.1007/978-3-540-87779-0_13`.

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

[12] R. Fan and N. Lynch. Efficient replication of large data objects. In *DISC 2003*, pages 75–91.

[13] Antonio Fernández Anta, Theophanis Hadjistasi, and Nicolas Nicolaou. Computationally light "multi-speed" atomic memory. In *Proc. of OPODIS16*, 2016.

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

[15] C. Georgiou, T. Hadjistasi, N. Nicolaou, and A. A. Schwarzmann. Unleashing and speeding up readers in atomic object implementations. In *Proc. of NETYS 2018*, pages 175–190. doi:10.1007/978-3-030-05529@-5_12.

[16] C. Georgiou, N. C. Nicolaou, and A. A. Shvartsman. Fault-tolerant semifast implementations of atomic read/write registers. *Journal of Parallel and Distributed Computing*, 69(1):62–79, 2009. doi:http://dx.doi.org/10.1016/j.jpdc.2008.05.004.

[17] T. Hadjistasi, N. C. Nicolaou, and A. A. Schwarzmann. Oh-ram! one and a half round atomic memory. In *Proc. of NETYS 2017*, pages 117–132. doi:10.1007/978-3-319-59647-1_10.

[18] M. P. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM TOPLAS*, 12(3):463–492, 1990. doi:http://doi.acm.org/10.1145/78969.78972.

[19] K. M. Konwar, N. Prakash, E. Kantor, N. Lynch, M. Médard, and A. A. Schwarzmann. Storage-optimized data-atomic algorithms for handling erasures and errors in distributed storage systems. In *Proc. of IPDPS16*, pages 720–729, May 2016.

[20] K. M Konwar, N Prakash, N. Lynch, and M. Médard. Radon: Repairable atomic data object in networks. In *The International Conference on Distributed Systems (OPODIS)*, 2016.

[21] N. Lynch and A. A. Shvartsman. Robust emulation of shared memory using dynamic quorum-acknowledged broadcasts. In *Proc. of Symposium on Fault-Tolerant Computing*, pages 272–281, 1997.

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

[23] N. Nicolaou, V. Cadambe, N. Prakash, K. Konwar, M. Medard, and N. Lynch. Ares: Adaptive, reconfigurable, erasure coded, atomic storage. In *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*, pages 2195–2205, 2019.

[24] N. Nicolaou, A. Fernández Anta, and C. Georgiou. Coverability: Consistent versioning in asynchronous, fail-prone, message-passing environments. In *Proc. of IEEE NCA 2016*, pages 224–231.

[25] M. O. Rabin. Fingerprinting by random polynomials. http://www.xmailserver.org/rabin.pdf.

[26] A. Tridgell and P. Mackerras. The rsync algorithm. https://rsync.samba.org/tech_report/.

[27] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar. An integrated experimental environment for distributed systems and networks. In *OSDI02*, pages 255–270, Boston, MA, December 2002. USENIX Association.

[28] H. Zhang, M. Dong, and H. Chen. Efficient and available in-memory kv-store with hybrid erasure coding and replication. In *FAST 16*, pages 167–180, Santa Clara, CA, 2016. USENIX Association.

Appendix

A Fragmented Objects with Coverable Blocks

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 [14].

To this respect, in [24] the notion of *coverability* was introduced to leverage the solvability of R/W object implementations, while providing a *weak* RMW object. Informally, 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.

More formally, coverability uses a *totally ordered* set of *versions*, say $Versions$, and introduces the notion of *versioned (coverable) objects*. A *coverable object* is a type of R/W object where each value written is assigned with a version from the set $Versions$. The *coverable* R/W object X offers two operations: (i) $X.\mathsf{cvr\text{-}write}(val, ver)_p$, and (ii) $X.\mathsf{cvr\text{-}read}()_p$. A process p invokes a $\mathsf{cvr\text{-}write}(val, ver)_p$ operation when it performs a write operation that attempts to change the value of the object. The operation returns the value of the object and its associated version, along with a flag informing whether the operation has successfully changed the value of the object or failed. A write is *successful* if it changes the value of the register; otherwise the write is *unsuccessful*. The read operation $\mathsf{cvr\text{-}read}()_p$ involves a request to retrieve the value of the object. The response of this operation is the value of the register together with the version of the object that this value is associated with. Denoting a successful write $\mathsf{cvr\text{-}write}(v, ver)(v, ver', chg)_p$ as $\mathsf{tr\text{-}write}(ver)[ver']_p$ (updating the object from version ver to ver'), and $\mathsf{cvr\text{-}write}(v, ver)(v', ver', unchg)_p$ as $\mathsf{tr\text{-}write}(ver)[ver', unchg]_p$, a coverable implementation satisfies the following properties (for the formal definition see [24]).

Definition 9 (Coverability [24]) *A valid execution ξ is **coverable** with respect to a total order $<_\xi$ on all successful write operations,$\mathcal{W}_{\xi,succ}$, in ξ if:*

- *(***Consolidation***) If a $\mathsf{tr\text{-}write}(ver_j)[*] \in \mathcal{W}_{\xi,succ}$ then ver_j is larger than any version written by a preceding successful write operation.*
- *(***Continuity***) if $\mathsf{tr\text{-}write}(ver)[ver_i] \in \mathcal{W}_{\xi,succ}$, then ver was written by a preceding write operation or $ver = \perp$ the initial version*
- *(***Evolution***) The version of the object is incrementally evolving and thus for two version 'chains' formed by concurrent writes on a single initial version ver, the last version of the longest chain is larger than the latest version on the shorter chain.*

If a fragmented object utilizes coverable blocks, instead of linearizable blocks, then Definition 3 provides what we would call ***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.

B Omitted Proofs

Proof.[Proof of Lemma 6] It is easy to see that if $\pi = \mathsf{update}(b, D)_{f,p}$ is successful, then all the $\mathsf{dsmm\text{-}write}$ operations invoked within π, including $\mathsf{dsmm\text{-}write}(val(b))_{b,p}$, are successful. It remains

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

Lemma 10 *In any execution ξ of* CoBFS*, if a $\rho =$ read$_{f,p}$ operation returns a list $\mathcal{L}$ then for any block $b \in \mathcal{L}$ there exists successful* update$(*)_{f,*}$ *operation that either precedes or is concurrent to ρ and invokes* sm-create$(val(b))_b$ *operation.*

Proof.[Proof of Lemma 10] According to our protocol it is clear that a block with id b appears in the list of f only if that is created and written during an update$_{f,*}$ operation. Also, if the block is created by an update that precedes ρ, then no other block in the list will point to b, ρ will not invoke a sm-read$_b$ operation for b, and thus $b \notin \mathcal{L}$.

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

If case (i) is true, then p will invoke a sm-write$(val(b))_b$ as b is the block that is updated. However, since we assume that the update was not successful, then by Lemma 6, the write operation is not successful. Thus, according to the coverable DSM, b was never written and this contradicts the assumption that p obtain $b \in \mathcal{L}$.

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

Proof.[Proof of Lemma 7] Assume by contradiction that there exist some $b_i \in \mathcal{L}$, s.t. $val(b_i).ptr \neq b_{i+1}$ (or $val(b_g).prt \neq b_1$). By Lemma 10, a block b_i may appear in the list returned by a read operation only if it was created by a successful update operation, say w.l.o.g. $\pi =$ update$(b, D)_{f,*}$. Let $D = \langle D_0, \ldots, D_k \rangle$ and $\mathcal{B} = \langle b_1, \ldots, b_k \rangle$ be the set of $k-1$ blocks created in π, with $b_i \in \mathcal{B}$. By the design of the algorithm we create a single linked path from b to b_k, by pointing b to b_1 and each b_j to b_{j+1}, for $1 \leq j < k$. Block b_k points to the block pointed by b at the invocation of π, say b'. So there exists a path $b \rightarrow b_1 \rightarrow \ldots \rightarrow b_i$ that also leads to b_i. According again to the algorithm, $b_{j+1} \in \mathcal{B}$ is created and written before b_j, for $q \leq j < k$. So when the b_j.cvr-write is invoked, the operation b_{j+1}.cvr-write has completed, and thus when b is written successfully all the blocks in the path are inserted successfully in f. So, if now b_i is different than b_k by the construction of the update then both b_i and b_{i+1} are in the list with $val(b_i).ptr = b_{i+1}$ contradicting our assumption.

If now $b_i = b_k$, then $val(b_i).ptr = b'$. Since b was pointing to b' at the invocation of π then b' was either (i) created during the update operation that also created b, or (ii) was created before b. In case (i), by Lemma 6, the update operation that created b was successful and thus b' must be created and inserted in f as well. In case (ii) it follows that b is the last inserted block of an update and is assigned to point to b'. With a simple induction one may show that the update operation that created b' must precede the update that created b. Since no block is deleted, then b' remains in $\mathcal{L}$ when b_i is created and thus b_i points to an existing block. Furthermore, since π was successful, then it successfully written b and hence only the blocks in $\mathcal{B}$ were inserted between b and b' at the response of π. So b' must be the next block after b_i in $\mathcal{L}$ at the response of π and there is a path between b and b'. This completes our proof. □

C Additional Operations Supported by the Prototype

To enhance the practicality of our prototype we have equipped it with additional operations, which are all framed around the two main operations of the FM.

Besides updating the contents of a file, reading a file and managing blocks, the *FM* supports a number of other useful operations, such as creating a file, renaming a file, deleting a file, obtaining a list of the existing files and an advanced list operation.

To store information about the files that the *FM* manages, internally the *FM* maintains a dictionary D. In more detail, a key entry is a file path f_{path} of f_{id}, and the corresponding value is a tuple consisting the b_{id} of the genesis block b_g of f_{id} and the file id f_{id} of the fragmented file f. That is, $D : \{key, value\} = \{f_{path}, \langle b_g, f_{id} \rangle\}$.

The *FM* uses f_{path} as key for this dictionary, in order to be able to monitor the changes that take place for each file. However, in the level of the Atomic Shared Object Algorithm, all the information about a file is stored based on its f_{id}.

It is worth mentioning that, the format of a block that sending to the Atomic Shared Object Algorithm, is a dictionary containing the header and the literal data of the block. The header includes some information about the block, i.e. the hash value, a boolean value that indicates if the block is the genesis one, the next b_{id}, the block size and the modification time of the block. If the block is the genesis block, the header it also contains the f_{path}.

- **Create Operation**: When a new file is created on the client's filesystem, the *FM* fragments it into its respective blocks (including the genesis block), and writes them on the servers by invoking a sequence of write operations for the entirety of the blocks comprising the file.
- **Rename Operation**: When a file is renamed on the client, the *FM* executes a special write request, where it writes the genesis block of the file that includes the new f_{path} in its header.
- **Delete Operation**: When a file is deleted on the client, the *FM* discards the f_{id} entry from its dictionary and sends a special write request to the servers, with the genesis bid b_{gen} of the file. The servers set the tag of the b_{gen} to -1, in order to notify that the file is deleted in case another client tries to have access to it before the delete operation is completed. As a result, no further operations can be performed on the deleted file, since the *FM* and the servers do not have access to its genesis block.
- **List Operation**: To obtain the list of existing files, the *FM* contacts the servers and obtains the f_{id}, the f_{path} and the genesis block id b_{id} of each file, which then allows for further read operations to be issued.
- **Advanced List Operation**: The advanced list operation, is similar to the simple list one, giving some additional information about each file. At first, the *FM* requests a simple list operation. Then for each file in the resulted list, it requests a series of block list operations. Each block list operation informs the *FM* about the size and the modified size of the block. As a result, the *FM* can calculate the size of the whole file and the maximum modified time that a block of the file has changed.