electronics

MDPI

Article

Distributed File System to Leverage Data Locality for Large-File Processing

Erico Correia da Silva *, Liria Matsumoto Sato and Edson Toshimi Midorikawa

Escola Politécnica, Universidade de São Paulo, São Paulo 05508-010, Brazil; liria.sato@usp.br (L.M.S.); emidorik@usp.br (E.T.M.)

* Correspondence: erico.silva@usp.br

Abstract: Over the past decade, significant technological advancements have led to a substantial increase in data proliferation. Both scientific computation and Big Data workloads play a central role, manipulating massive data and challenging conventional high-performance computing architectures. Efficiently processing voluminous files using cost-effective hardware remains a persistent challenge, limiting access to new technologies for individuals and organizations capable of higher investments. In response to this challenge, AwareFS, a novel distributed file system, addresses the efficient reading and updating of large files by consistently exploiting data locality on every copy. Its distributed metadata and lock management facilitate sequential and random I/O patterns with minimal data movement over the network. The evaluation of the AwareFS local-write protocol demonstrated efficiency across various update patterns, resulting in a performance improvement of approximately 13%, while benchmark assessments conducted across diverse cluster sizes and configurations underscored the flexibility and scalability of AwareFS. The innovative distributed mechanisms outlined herein are positioned to contribute to the evolution of emerging technologies related to the computation of data stored in large files.

Keywords: distributed file systems; HDFS; Big Data; distributed lock management; scientific data analysis; data locality

Citation: da Silva, E.C.; Sato, L.M.; Midorikawa, E.T. Distributed File System to Leverage Data Locality for Large-File Processing. *Electronics* **2024**, *13*, 106. https://doi.org/10.3390/electronics13010106

Academic Editors: Ioannis Yiannis Kompatsiaris, Stefanos Vrochidis, Giuseppe Amato and Sotiris Diplaris

Received: 7 November 2023
Revised: 8 December 2023
Accepted: 18 December 2023
Published: 26 December 2023

1. Introduction

Crucial emerging technologies, such as generative artificial intelligence, place significant demands on computational resources, including ample storage, potentially limiting access for companies with constrained budgets [1]. Additionally, organizations leverage information for decision making and to enhance competitiveness. Consequently, every digital action is treated as informational and systematically recorded, resulting in a significant volume of data. Business data analytics requires the adoption of advanced technologies like Big Data [2]. Over the last decade, an enormous number of companies have started using Big Data technologies in their businesses, facing several challenges [2]. The IDC predicts a significant surge in global data, expecting it to escalate from 33ZB in 2018 to 175ZB by 2025 [3]. Simultaneously, distributed file systems are widely used to store data related to physics experiments. In [4], Blomer mentions experiment collaborations at the Large Hadron Collider (LHC) that store over 1 billion files and hundreds of petabytes. Other scientific data, such as weather or seismic data, are also extensive in size [5]. Meanwhile, machine-generated data from sources like sensor devices and surveillance cameras are prominent data generators for Big Data, leading to an amplification of both volume and variety [5]. The unstructured nature of storage contributes to challenges in managing and analyzing data effectively [4,5]. Advances in storage systems are required to effectively read and write scientific data, corporate data, and machine data, aligning with the evolving applications designed to handle them over the years.

The processing of such large volumes of data requires big clusters of servers, utilizing commodity infrastructure to overcome associated costs. Due to their characteristics, the

use of these clusters involves assuming hardware failure as the norm [6]. In this scenario, Hadoop has become the most popular framework, especially because it can employ heterogeneous clusters of inexpensive hardware [7]. Owing to the volume of data, moving it to the computational resources takes precious time, so Hadoop takes advantage of data locality [7], with mechanisms and semantics closely related to its MapReduce processing methodology. With Hadoop MapReduce, data are processed in chunks on the same servers that store the data in a distributed file system called HDFS. Due to its 'write-once, read-many' semantics, it is not practical to use HDFS with processing strategies other than map and reduce, lacking better compatibility with scientific applications generally constructed over embarrassingly data-parallel patterns [8].

In contrast to data-centric frameworks like Hadoop and its HDFS, traditional scientific tools follow a compute-centric idea, also using clusters but moving the data to the required computational resources [8]. Widely used distributed file systems for scientific computing like Lustre [8,9] and PVFS [10] have architectures that are highly dependent on hardware resilience, taking failure as the exception [6]. This implies that using commodity hardware poses a challenge. Therefore, a significant redesign of such distributed file systems would be necessary to accommodate Big Data characteristics.

As we observe in Big Data workloads, scientific data processing may require the analysis of large datasets [8,11], leading to a high overload for data transfer in traditional compute-centric clusters with MPI-based applications [8]. Scientific computing may require more flexible I/O profiles, including random writes. MapR-FS, as a more sophisticated data-centric distributed file system, implements POSIX semantics [12], allowing for random I/O. However, with MapR-FS, data locality is not leveraged during write requests, as they are exclusively handled by the node with the primary copy, using a remote-write protocol [13]. Despite its popularity, HDFS also poses challenges when working with metadata, requiring more robust servers to manage the namespace with considerable complexity for redundancy [14]. In fact, data locality can be exploited in various domains. Wang et al. [15] emphasize the significance of leveraging data locality to effectively distribute tasks associated with data-intensive workloads in synergy with storage resources. In addition, the evolution of storage systems for Big Data urges novel metadata management strategies. Ceph [16] and GPFS [12] implement enhanced distributed and fail-safe metadata management, even though their architectures are not inherently data-centric, causing data movement to compute nodes for processing. In [11], Zou et al. propose a monolithic approach for a distributed storage system that avoids the computational costs imposed by layered architectures. However, this comes at the expense of compatibility with many well-known Big Data and HPC applications that rely on a distributed file system for storage.

The architecture enabling high-performance data analytics (HPDA) systems is a recent focus of research, aiming to integrate advancements from both Big Data and HPC technologies. Emphasizing data locality is crucial for ensuring efficiency, particularly concerning time and energy considerations [17]. Usman et al. emphasize in [17] that, to prepare for the upcoming exascale systems, it is essential to prioritize research in technologies capable of efficiently working with locality-aware scheduling. These aspects highlight the potential advantages of data locality for problems involving large datasets. However, traditional Big Data distributed file systems are often incompatible with MPI-based programs like mpiBLAST 1.6.0 and ParaView 3.14 [8], preventing the full realization of these benefits.

1.1. POSIX and Data Locality

The IEEE Computer Society specified the Portable Operating System Interface (POSIX) as a set of standards to maintain compatibility among operating systems, enabling the portability of software between them. Aligning a file system with POSIX facilitates software development and expands its adoption. Various distributed file systems such as Lustre [9], GPFS [12,17], Ceph [16], or BeeGFS [18] are largely POSIX-compliant but do not exploit data locality. This implies that, before processing can occur, all data must be moved to the location of computational resources (e.g., hosts with CPU and memory). On the other

hand, distributed file systems like GFS [12] and HDFS [14,19] enable the leveraging of data locality but are not fully POSIX-compliant. MapR-FS [12] is fully POSIX-compliant with random-write capabilities managed by a single static node where the primary copy resides, utilizing other copies solely for read purposes.

The HDFS relaxes certain POSIX requirements, restricting file access from allowing random writes. Its write-once/read-many semantics drove the development of other Hadoop components to alternative approaches in order to meet semantic constraints. A notable example is HBase [20,21], a Hadoop column-oriented nonrelational database management system. HBase introduced specific housekeeping procedures, known as compactions, to generate new files by merging valid data from original files and removing unnecessary entries, what would not be necessary with a rewritable storage. However, running compactions in HBase clusters can impact performance [22].

1.2. Our Contributions

In recent years, concentrated efforts in research have aimed to integrate the advantages of Big Data systems with those of scientific computing systems, combining these two approaches in the development of new tools to address their respective requirements [17]. In this study, we present AwareFS, an innovative data-centric distributed file system, empowered by a robust locking mechanism. The initial architecture of AwareFS was first introduced at the "2021 IEEE International Conference on Big Data", where the concept of the local-write protocol was initially presented, focusing solely on sequential I/O. Through subsequent updates, AwareFS now supports a broader set of I/O profiles, leveraging concurrent processing techniques to invalidate regions of chunks. This optimization significantly enhances the efficiency of random I/O operations. Tailored for applications in both MapReduce processing and MPI-based programs, AwareFS strategically employs data locality in read and write workloads. The system utilizes a local-write primary-based protocol to minimize data movement while ensuring data persistence across available copies. In contrast to HDFS, implemented in Java, AwareFS is entirely developed in C++, incorporating refined parallelism techniques to optimize performance. Deployable in heterogeneous clusters of commodity hardware, it provides a multilanguage client development interface. Engineered to mitigate single points of failure, AwareFS adopts a distributed horizontally scalable architecture for managing both data and metadata. Consistency is ensured through an advanced distributed locking management system. The evaluation of AwareFS encompasses a comprehensive analysis of its performance and scalability, highlighting the efficacy of the local-write protocol in enhancing distributed write performance. Furthermore, we demonstrate the read and write capabilities across various cluster configurations, block sizes, and random-access patterns—features crucial for scientific computing that are not supported by Hadoop and its HDFS.

2. Related Works

Many distributed file systems have been proposed so far [4], many of them widely used like HDFS [14] for Big Data processing frameworks and Lustre [9] for high-performance computing systems. With an architecture that considers hardware failure as a norm [6], which allows its use with commodity hardware, HDFS is the de facto distributed file system for Big Data applications, but it places significant challenges due to its write-once/read-many semantics that do not allow random write access [4,19]. On the other hand, Lustre permits a more flexible semantics with a robust locking mechanism [9]. The architecture of Lustre considers hardware failure as the exception [6] and is a better fit for HPC applications with more compute-centric characteristics [8]. Another robust compute-centric-focused file system is Ceph [16]. Built upon a distributed object store known as RADOS, Ceph distinguishes itself by decoupling data and metadata management, employing a pseudo-random data distribution algorithm named CRUSH. However, Ceph currently lacks a significant data-locality exploitation mechanism. With a similar approach to optimizing metadata management, BeeGFS [18] enables architectures with multiple metadata servers, distribut-

ing the workload on a per-file or per-directory basis. It is designed for compute-centric clusters without specific data-locality requirements. To leverage the use of commodity hardware, GlusterFS [23] provides different replication strategies, from simple replications to erasure coding, and eliminates the need for centralized metadata servers. Exploiting data locality with GlusterFS poses a challenge due to its design, which was not initially conceived for data-centric purposes. The latency introduced by replication may compromise I/O efficiency. Other distributed file systems like PVFS [10] and GPFS [12] are used in compute-centric clusters where the data must travel from storage hosts to where computation occurs. REHDFS [19] enhances HDFS by adding a lock/validation manager to permit fully random data access. Data locality plays a crucial role in ODDS [8], which introduces a data distribution monitor, a virtual I/O translation layer, and a data-locality scheduler. These components are integrated with HDFS to enhance storage management for data-centric computation. Instead of leveraging HDFS, MapR-FS [4,24] is another distributed file system, focused on data-centric Big Data processing, with an architecture that considers hardware failure as the norm. MapR-FS allows flexible semantics with random data access using a remote-write protocol, where the primary copy is located on the node that acts as the master of the replication chain, and it will not migrate to a different node. With this approach, exploiting data locality for write operations in MapR-FS is restricted to just one copy of the data. This limitation results in network traffic even for write operations initiated on a node with a valid copy that is not the primary copy. MapR-FS does not rely on locks to ensure data consistency and, if a conflict is detected, its lockless persistence mechanism will force a costly undo of the write process. Nowadays, Big Data frameworks rely on a layered architecture where the distributed file system is the persistence layer and is combined with other layers like memory management and job execution. Despite this layering strategy allowing for an easy combination of different systems with compatible interfaces, its adoption may impose an overhead that is eliminated by the monolithic approach proposed by Pangea [11]. Unlike AwareFS and other distributed file systems that support common interfaces, Pangea cannot be integrated with widely used frameworks such as Spark and other traditional workloads. Pangea is also incompatible with operational workloads due to its required write-once/read-many semantics.

3. AwareFS Architecture

The idea of having a distributed file system for data-centric processing that would also be a good fit for traditional scientific workloads, usually processed in HPC clusters, posed an enormous challenge while defining the AwareFS architecture. All components combined should be highly distributed to promote availability with no single point of failure. At the same time, components should be organized in a manner that facilitates scalability on top of clusters of commodity hardware, where we have failure as the norm. Because of these attention points, all of the AwareFS architecture was designed to work with distributed storage systems, where both data and metadata are stored and managed in the same servers that should be used to process the data chunks. A robust distributed lock management system was designed to guarantee data consistency with locks controlled as close to the data as possible, leveraging data locality also for the computation involved in the data access management.

3.1. Data Placement

All data stored in AwareFS [13] are divided in chunks, and each chunk is stored in a container (Figure 1). Besides these chunks, a file in AwareFS also has its metadata stored in an inode. In AwareFS, each container is a storage area that can be a compound of up to 256 entities that can be either chunks or inodes. Every container has at least three copies, each one stored in a different storage node called a DS, from "Data Service". A single DS can store the data of several containers. The available containers are managed by a centralized service called CS (Container Service). Every container created has a set of assigned DSs that will have copies of its data; this set of DSs is the container's replication chain. It is the CS's

responsibility to request the creation of new containers whenever needed and guarantee that all DSs have a valid copy of the container list, that is, a simple list with the replication chain of every available container.

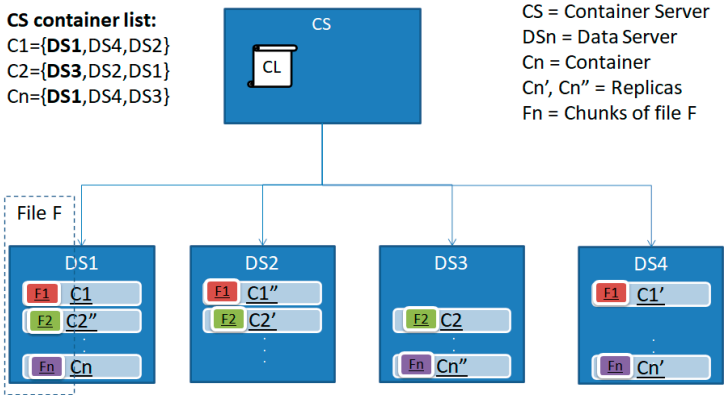


Figure 1. AwareFS data placement scheme: file chunk distribution (source [13]).

3.2. Architecture Overview

Considering the aforementioned data-placement strategy, we created the AwareFS architecture in a manner that enables the distributed control of storage for both chunks and inodes. Functions were organized in services that can reside in hosts of different architectures, promoting the usage of heterogeneous clusters. All components were thought to be easily restarted or replaced by a different instance, in case of any unresponsiveness. Figure 2 shows a small distribution of AwareFS components, where each dashed space limits the responsibility of a different host, i.e., two data nodes and one metadata node. In this example, one of the data nodes also runs the AwareFS client.

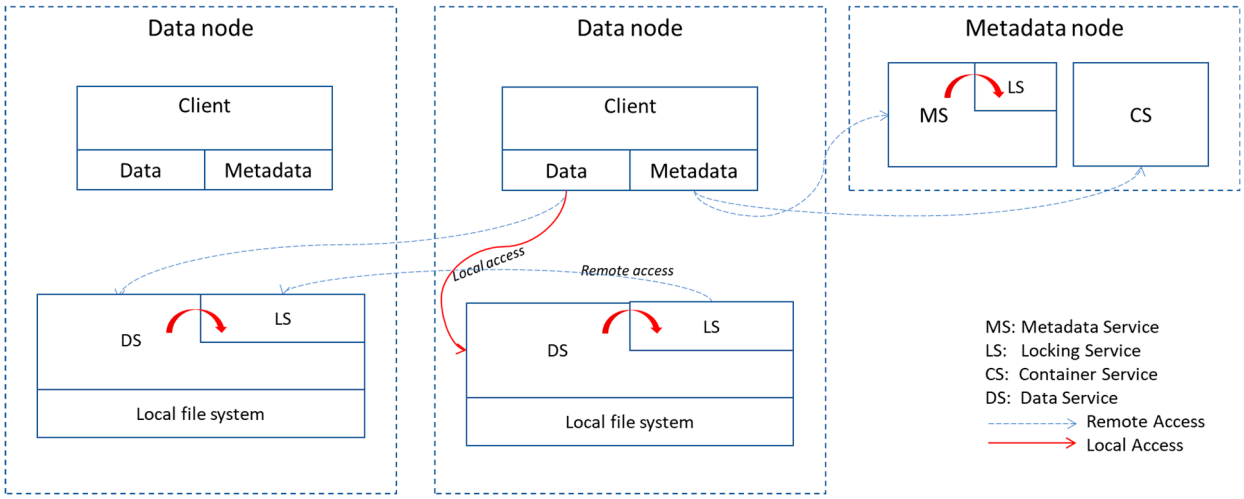


Figure 2. AwareFS architecture: example of component distribution across nodes.

The AwareFS components are as follows:

- DS: Data Service, responsible for storing and controlling access to both chunks and inodes organized in different containers. It is the DS's responsibility to answer concurrent I/O requests in a consistent manner;
- LS: Locking Service, the AwareFS service that manages the locking requests required to guarantee consistent read and write operations. Every DS has an associated LS instance, controlling lock requests to I/O operations on the stored chunks and inodes;
- MS: Metadata Service, will control the access point of each file, maintaining a lookup table from file path names to its inodes;

- CS: Container Service, is responsible for maintaining the container list and organizing the creation of new containers by the available DSs;
- Client: The AwareFS client is responsible for managing the interface between user requests and other AwareFS components like the DS and MS.

The components within the system engage in continuous message exchanges to signal their availability. It is the responsibility of the master Data Service (DS) within each replication chain to replace an unresponsive DS. This process is initiated by the master DS contacting the Container Service (CS) to obtain information about a new DS and subsequently initiating the inclusion of this new DS into the replication chain through a message exchange protocol.

In the event of a failure of the master DS within a replication chain, the addition of a new DS is preceded by the execution of an election protocol. During this protocol, the other DSs within the chain collaborate to determine the new master of the replication chain. Upon the inclusion of a new DS into the replication chain, all associated chunks and inodes are deemed invalid. The new DS then begins the process of receiving data from the other DSs within the chain, swiftly attaining a consistent state. This expedited convergence is facilitated by the relatively small size of a container. For any AwareFS cluster, singular instances of both the Metadata Service (MS) and Container Service (CS) are mandated. This necessity arises from the centralized and unique nature of the naming space governed by the distributed file system. Despite this singular configuration, the global availability of the cluster remains unaffected due to the architecture's design, which facilitates seamless continuity through a straightforward restart or replacement of both components by standby instances.

The restart or replacement process for both the Container Service and Metadata Service is uncomplicated, facilitated by the simplicity inherent in the container list and the pathname-to-inode lookup table. This design ensures the efficient restoration of these critical components, underscoring the resilience and ease of maintenance within the AwareFS architecture.

In this initial version of AwareFS, all data are stored in regular files of the DSs' underlying Linux file system. Also, the CS is a very lightweight service while the MS is based on a simple key/value table, while the inodes themselves are spread throughout the DSs.

3.3. Metadata Management

To facilitate scalable metadata management, AwareFS distributes the storage of metadata and the corresponding locking mechanisms necessary for maintaining consistency. Unlike HDFS, AwareFS achieves this scalability without relying on powerful hosts with larger RAM and fast disks [14]. For simplicity, metadata storage is delegated to Data Services, aligning with the approach used for regular file chunks. This design naturally increases metadata management capacity as storage capacity scales.

In standard Unix file systems, an inode is a data structure that defines a stored object, such as a file or directory. In AwareFS, the index associating pathnames with their respective inodes is initially managed by a centralized service, the Metadata Service (MS). Notably, most metadata reside within these inodes, which are stored and controlled in a distributed manner by Data Services (DSs), mirroring the approach used for chunks.

To facilitate distributed access to inodes and chunks, the Metadata Service (MS) maintains an index associating each pathname with a unique structure. This structure includes the identifier of the container used for storage, along with the inode number and a version number. Each entry stored in the MS database has the pathname as the key, and the corresponding values include the container identifier, inode identifier, and the inode version. The Container List (CL) serves as a crucial reference for identifying Data Services storing replicas of the inode. Much like the procedure for handling chunks, any Data Service with a valid copy of the inode can be employed for retrieving the inode content in a read operation. However, exclusive authority to alter attributes of the inode is reserved for the owner DS of the inode. To access data associated with the file described by the inode, the CL is once

again consulted to determine which DSs store copies of the chunk. Subsequently, read or write operations are redirected to these identified Data Services (Figure 3). In Figure 3, a file comprised of three chunks, namely, F1, F2, and F3, is distributed across three containers labeled C1, C2, and C3. Taking F1 as an example, its replicas, denoted as F1′ and F1″, are stored in containers C1′ and C1″. Data Services 1, 2, and 3 form the replica chain, as illustrated.

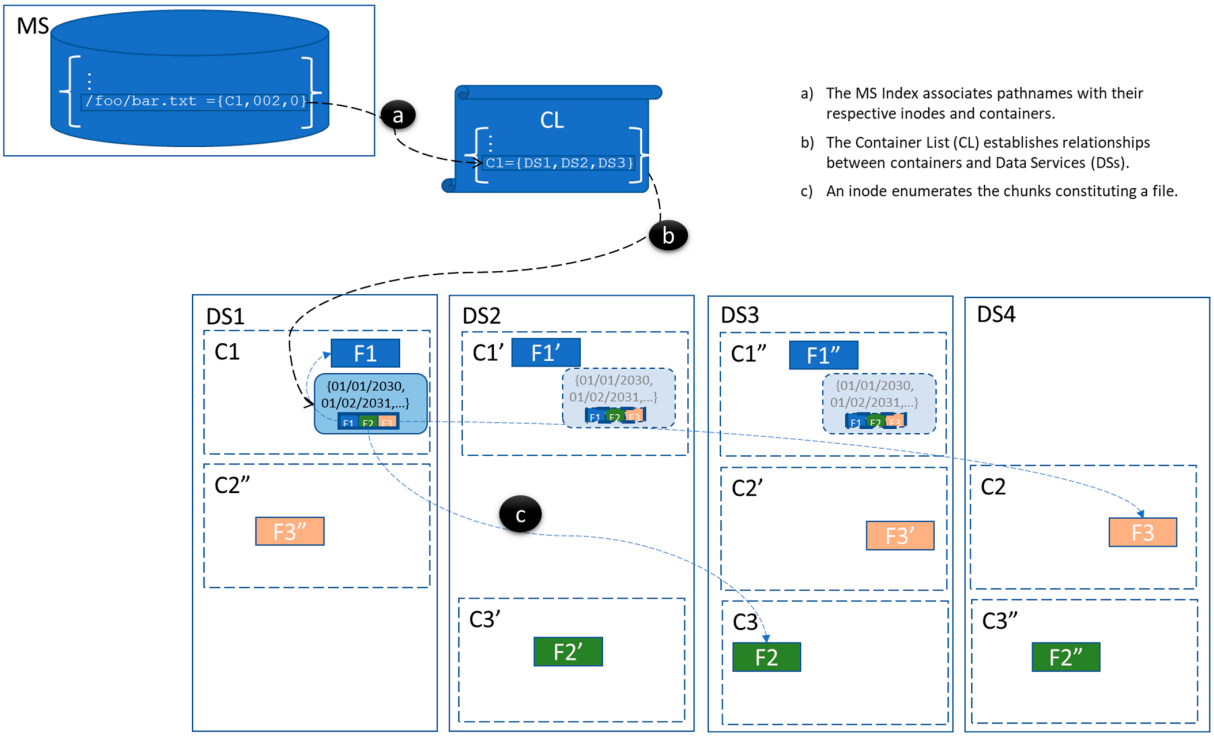


Figure 3. File access references: how the pathname-to-inode lookup table is utilized in conjunction with the container list to access inodes and chunks.

With the described mechanism, as locks are ultimately managed by the Locking Service associated with the Data Service storing the inode, all lock-related management for metadata stored in inodes is distributed across the cluster.

3.4. Replication and Checkpoints

To promote the redundancy of data storage, each chunk is replicated from its DS owner to at least two other DSs. The replication chain compound of at least three DSs, e.g., "{DS1, DS5, DS8}", will dictate what are the DSs that will be engaged to create copies of each chunk. Due to the primary-based local-write protocol of AwareFS [13,25], only one of the DSs in the replication chain will have the primary copy of a chunk at a given point in time; this DS is called the "owner DS" of the chunk. When data are initially written, the DS attending the write requests is the first owner of the chunk, but the ownership will move among the DSs in the replication chain whenever write requests are attended by other DSs in the chain. As soon as data are written to a chunk, the other DSs are requested to invalidate the chunk area being written, and then they will request and receive copies of the new data, keeping all copies identical.

The initial version of AwareFS [13] was of great help to evaluate the ownership migration strategy, although write mechanisms were efficient only for sequential I/O. Invalidating entire chunks for partial updates results in an inefficient replication approach, given the relatively large volume of data in chunks. To keep replicas identical, AwareFS underwent a comprehensive revision, now implementing a strategy of invalidating only the affected areas of a chunk. Each chunk carries information about its version, which is

updated in every update. Upon performing a write, an invalidation message containing the new version of the chunk, along with the offset and length of the affected area, is sent to other Data Services in the replication chain. After receiving the invalidation message, the DS sends a replication message requesting the data in the affected area. An entire chunk replication is requested, only if the version in the invalidation message indicates a late or out-of-order arrival. The same process applies to data replication messages.

Each container has a data structure called a "container content map" that stores the identification of all stored chunks, e.g., if it is a data chunk or an inode, if it is invalid, and which DS is the chunk owner, i.e., the DS with the primary copy of the chunk. The container content map is stored in a file in the same directory where the container data reside. Container data are stored in the underlying file system, using regular files. All files of a container are stored in a directory named as "<CID>.<version>", where CID is the container identification number and version is a sequential number of each declared version of the container.

Checkpoints

The container version is defined by a checkpoint process where all DSs in the replication chain exchange messages to assure that all the replicas have the same state of both data and metadata. Each version of a container is stored in a specific directory. After a given time, this checkpoint process is repeated, and a new directory for the new container version is created; then, writing to the previous container versions is blocked. When a new container version directory is created, hard links to the unchanged files are created, and new files are stored for every new chunk. As write requests will modify chunk files in directories for all versions, older versions will then have files to keep older data, touched by the new writes, in a copy-on-write scheme [26], and these data will be used to revert all to the state of a prior version.

With this approach, the container version is increased after a synchronization of the container state among the DSs in the replication chain, assuring consistency and enabling recovery to a previous valid state. Checkpoints will play an important role in recovery procedures due to DS crashes or any permanent failure. The readmission of a DS in a replication chain after transient failures will also take advantage of checkpoints to recover, requesting new copies only for chunks and inodes touched after the last available checkpoint.

3.5. The Data Service and the Primary Copy Ownership Management

At the core of AwareFS lies its Data Service (DS), a fundamental component tasked with overseeing replication, managing read and write requests, and facilitating resilience strategies. The DS communicates through messaging passing protocol or Remote Procedure Call (RPC) not only with other DS instances but also with additional components, including the Container Service (CS), clients, and Locking Service (LS). To concurrently manage various containers in different DS instances, a message passing processing protocol is employed.

Serving read and write requests is the most important activity that the DSs perform, but several other important procedures are dealt with by the DS, the most important ones being container creation, container checkpoint management, chunk replication, chunk region invalidation, and chunk primary copy ownership transference. When creating a container after a CS request, the master DS of the container replication chain, i.e., the first DS of the chain, will send a message for the DSs to create the container and another message to create a new communication context specific for exchanging messages related to this container among the DSs in the chain, starting the parallel processing of messages for synchronizing the container content map (Figure 4). The replication chain comprises at least three DSs, with the master DS overseeing the chain. The Container Service (CS) is responsible for managing the container list and initiating the container creation procedure. Subsequently, the master DS ensures the proper establishment of the replication chain. Entrusting the CS with initiating the container creation procedure is not an issue, as it

involves a lightweight procedure that should occur infrequently, given the container size and the pre-existence of containers across the cluster.

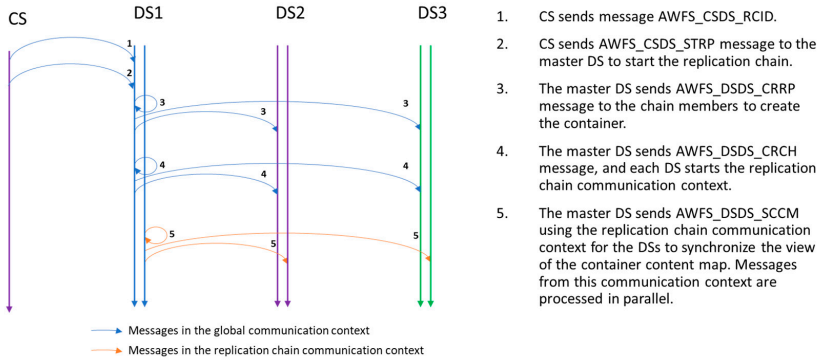


Figure 4. Container creation protocol: message exchange for container creation.

The checkpoint creation protocol is also initiated by the CS, with another lightweight procedure, and the master DS will make sure all DSs in the chain have a synchronized copy of the container content map before starting the checkpoint creation (Figure 5).

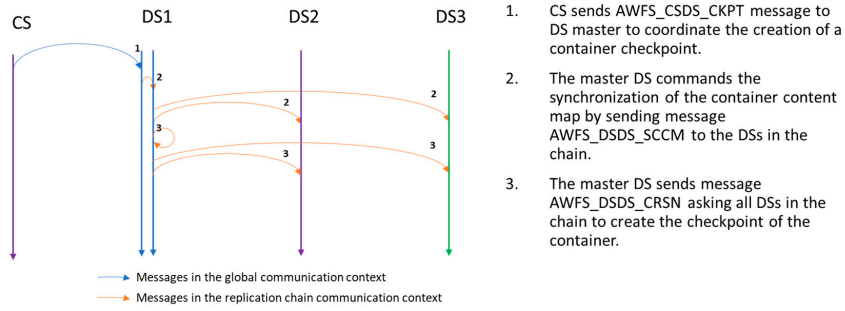


Figure 5. Container checkpoint creation protocol: message exchange for checkpoint creation.

The chunk replication is also guaranteed by a message passing protocol, where the DS with a chunk completely or partially invalidated will request data replication to the DS which owns the chunk. The replication protocol is more complex to accommodate MPI implementation requirements, where the receive part of the process must be ready when the send occurs. After a synchronous write request from the AwareFS client, the affected region undergoes invalidation either synchronously or through message passing. Following this, Data Servers (DSs) lacking the primary copy transmit messages to the DS designated as the owner of the chunk, requesting the replication of the altered data. The owner DS then dispatches a message to initiate the data reception process. Simultaneously, the DS seeking the new data commences parallel reception and notifies the owner DS of its readiness to receive the data. Subsequently, the owner DS initiates the transmission of the data (Figure 6).

Read and write requests are attended synchronously. Any DS in the replication chain is capable of answering read requests, as long as it has a valid copy of the required chunk. However, adhering to the primary copy write protocol, only the DS designated as the owner of the chunk will respond to write requests. Additionally, in accordance with a local-write primary-based protocol, AwareFS transfers the ownership of the primary copy to another DS if it resides on the same host as the client requesting the write. Considering the file access references schema in Figure 3, a write request is started by the client and may interact with more than one Data Service (Figure 7). To guarantee consistency of concurrent reads and writes, the DS must request and acquire locks for each operation. Such locks are managed by the LS attached to the owner DS of the chunk. The process of transferring chunk ownership is carried out synchronously, employing a two-phase

commit protocol. Initially, the former owner of the chunk conducts an inquiry among all associated Data Servers (DSs) within the chain to determine the feasibility of ownership transfer. If conditions permit, the former owner then instructs all DSs to officially recognize the newly designated DS as the rightful owner of the chunk going forward. Conversely, if the transfer is deemed unfeasible, the former owner directs all DSs to terminate the chunk ownership transfer process (Figure 8). It is crucial to note that the successful transfer of chunk ownership relies on the recipient DS possessing a valid copy or being able to obtain one from the relinquishing owner.

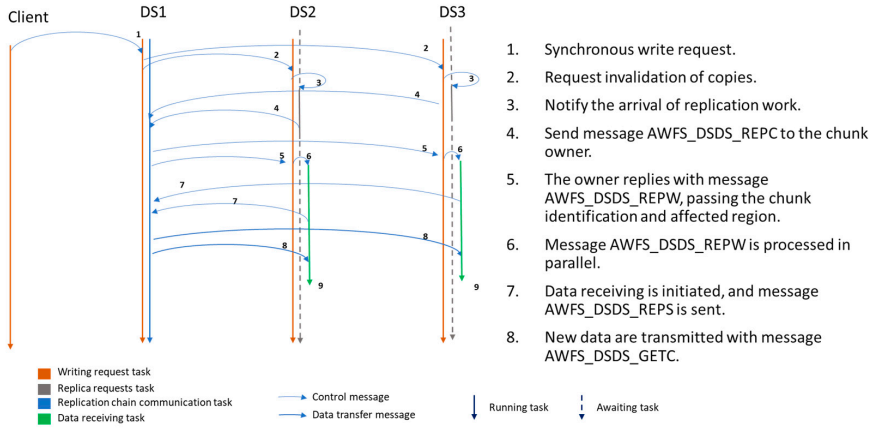


Figure 6. Chunk replication protocol: the exchange of messages to synchronize data among DSs in a replication chain.

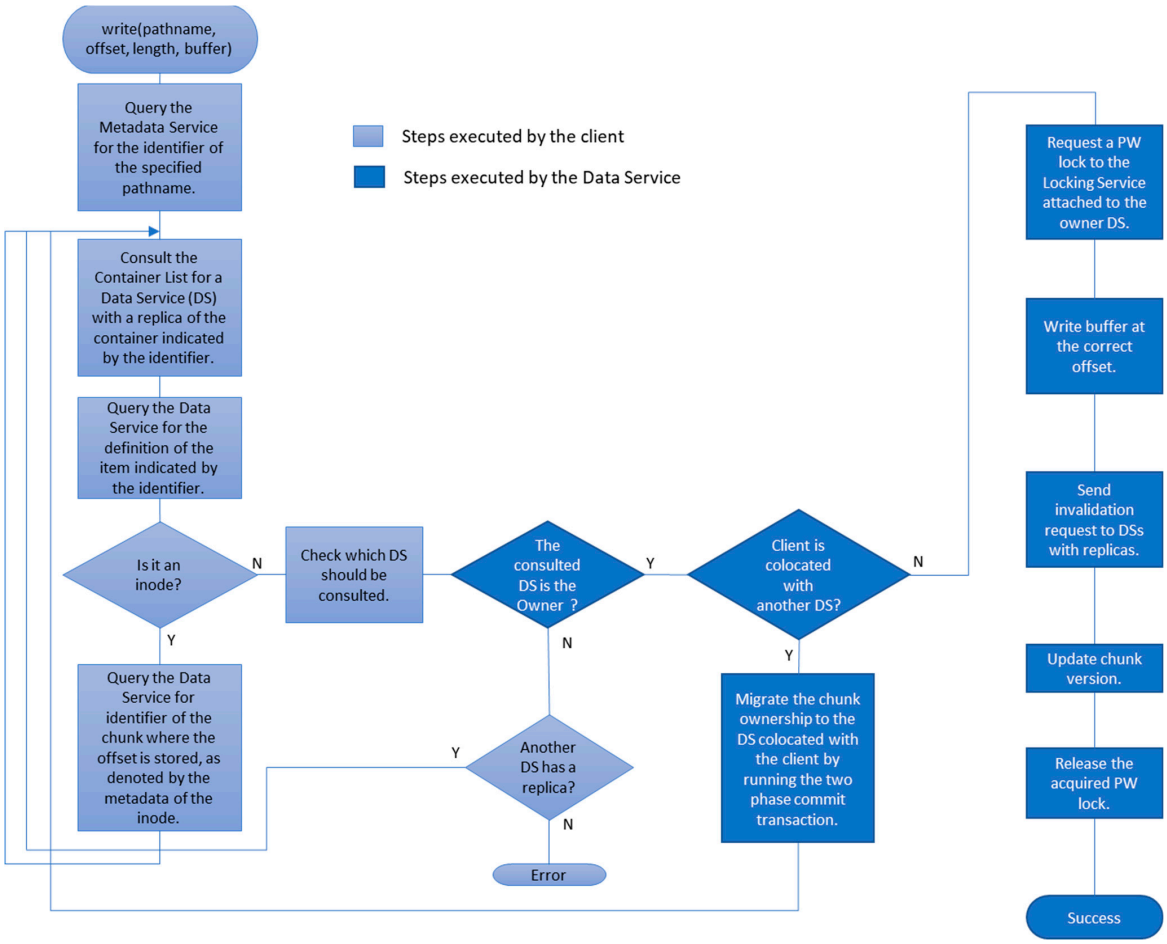


Figure 7. Steps for handling a write request from the client to the Data Service side.

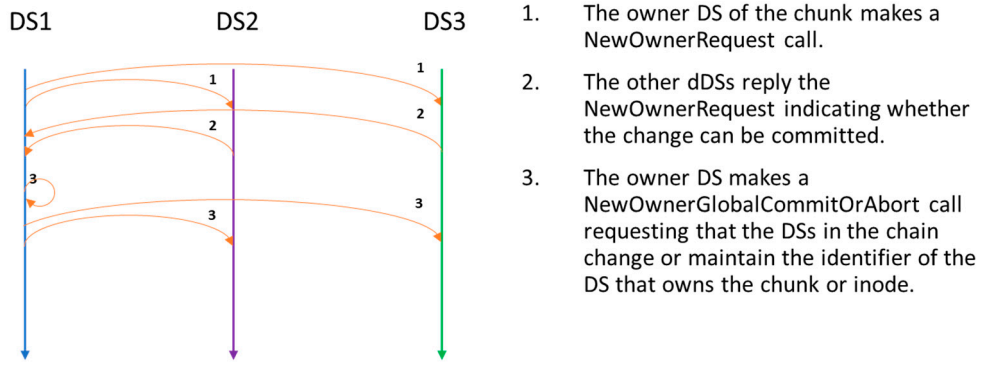


Figure 8. Chunk ownership transference protocol: the exchange of messages to migrate the primary copy ownership to a different Data Service.

4. Distributed Locking Management

As explained in [13], the AwareFS coherency is based on a sequential consistency model, where all writes are ordered by the DS which owns the primary copy of the affected chunk or inode. For consistently attending concurrent read and write requests, AwareFS provides coherency by a distributed lock management system. To manage locks for all the chunks it owns, each DS will have a local LS instance. Considering that files are divided in chunks, and chunks are spread throughout the several available DSs, lock management will be as distributed as the chunks primary copy ownership, spreading the considerable resource usage that such a workload requires and always leveraging data locality.

AwareFS lock types can be "file lock", "metadata lock", "path lock", or "chunk lock". File locks are the ones requested by the users using the Linux flock or POSIX fcntl function. Metadata locks are for metadata/inode manipulation. Path locks are for pathname-to-inode lookup table consistency. Chunk locks are used to guarantee data consistency and are treated by chunk region. Like Lustre [9], AwareFS lock management uses a strategy based on the VAX distributed lock manager, but the lock management distribution uses a different approach, as it is based on the inode or chunk primary copy ownership. As detailed in [13], AwareFS locks are controlled managing a structure called aw_lock_request, containing the lock mode, the identification of the entity associated with the lock (e.g., chunk or pathname), the requested lock type and mode, and the affected region of the file. The available lock modes are protective read/write (PR and PW), concurrent read/write (CR and CW), and exclusive (EX). Protective locks are used for coordinating data updates, and concurrent locks are especially for metadata updates that may happen concurrently without any consistency loss. Exclusive locks will cause all other accesses to be denied and are used for chunk ownership transferences. Table 1 has a column and a line per available lock mode, and it shows the compatibility between them. Locks are compatible with each other if the crossing cell for their modes has a 1 in Table 1, or if the respective regions of the file are not overlapping. For instance, considering Table 1, a protective read (lock PR) will be denied if requested for a chunk region for which a protective write lock (PW) was granted before because the PR/PW cell is 0 since these lock types are not compatible.

Table 1. Lock compatibility matrix: lock modes are compatible if the cell at the intersection has 1.

	CR	CW	PR	PW	EX
CR	1	1	1	1	0
CW	1	1	0	0	0
PR	1	0	1	0	0
PW	1	0	0	0	0
EX	0	0	0	0	0

The consistency of data operations is guaranteed by the DSs by using chunk locks, while other lock types are used for file lock and metadata consistency. Each LS controls all lock acquisition by manipulating a list of locks granted, named GrantedList, and a list of locks that were requested and are waiting to be granted, named WaitingList. When a lock is requested, the LS of the corresponding DS with the primary copy of the chunk or inode will first check if the lock lists have any other lock incompatible with the one being requested. The lock is granted if no other incompatible lock is registered. If an incompatible lock is present in the granted lock list, the lock being requested is added to the waiting list. When locks are released, they are removed from the granted list, and the waiting list is checked to see if any other lock can be finally granted.

5. Implementation Details

The AwareFS components follow a distributed processing organization where synchronization and communication are guaranteed through message passing or RPC. All development was conducted in C++ using Apache Thrift [26,27] and MPI [28].

Multithreading capabilities were enabled by both OpenMP [28,29] and native C++14 [30] thread control features. The initial number of threads in use by each AwareFS component can be configured and is scalable. OpenMP locking structures are largely used and play a key role along with C++ conditional variables to avoid costly busy waits while processing data replication, container checkpoints, and container creation.

The concurrent use of LS GrantedList and WaitingList require the use of semaphores for their thread safe management, and the DS has a C++ unordered_map container where each register associates each lock request (aw_lock_request structure) with a dynamically allocated OpenMP lock (i.e., omp_lock_t). All dynamic memory allocation was implemented using C++14 smart pointers, avoiding memory leaks without any garbage collection mechanism.

Container management is highly parallelized by using an MPI communicator for dealing with each container in a different thread. The number of threads in use is configurable, and the same system thread will be reused for different containers avoiding costly thread creation processes. For example, during the container creation, the master DS of the container replication chain will send a message for the DSs to create the MPI communicator for all messages related to this container, and all communication about this new container will then be dealt in parallel, using another thread (Figure 4).

The use of MPI for message passing was an enabler, allowing the creation of a robust multiplatform distributed file system. By creating a replication protocol where MPI_Send starts only after its equivalent MPI_Recv, the efficient data transfer capabilities of MPI could be leveraged. Besides using an MPI communicator for each container, MPI communication functions are highly parallelized by using a special MPI tag as a hash of the related chunk identifier, the offset, and the length of the affected region.

Another important enabler for AwareFS implementation was using Thrift for all RPC communication and for data serialization. The powerful Thrift TNonblockingServer class was used in both the DS and MS, allowing the efficient use of several concurrent connections with a minimum configurable number of threads. Thrift being a multilanguage library means that new clients and all interaction with the AwareFS components can also be written in different languages like Java, Python, Ruby, Go, and many others.

The AwareFS prototype was built following the entire conceptual design, including check points and data replication. Failure detection and component resilience were not developed in this initial version since their absence would not affect points being evaluated now, like the benefit of taking advantage of data locality with the primary copy local-write protocol.

5.1. Easy Deployment with Configuration Files

AwareFS is easily deployable by initializing the Metadata Service and Container Service, followed by the deployment of Data Services on each node within the cluster.

Parameterization of all services can be accomplished either through the command line or by specifying parameter values in configuration files.

The Metadata Service offers the option to specify the TCP port for the service, the quantity of threads allocated for service I/O processing, and the number of threads dedicated to concurrently processing metadata management requests. The Container Service requires parameters to include the TCP port number, the initial count of Data Services in the cluster, and the default chunk size. The Data Service permits different configurable parameters, allowing users to determine whether writes should be directly sent to disks, and defining the number of threads engaged in the concurrent processing of replication, checkpoints, and replica invalidation. Furthermore, the Data Service configuration file enables the definition of the underlying Linux file system directory for data storage.

The deployment process is very simple using any scripting language for constructing the configuration files or through commonly used automation tools.

5.2. Fuse Client

Considering the importance of POSIX compliance, the most important AwareFS interface is its FUSE-based [31,32] client. FUSE or "file system in user space" is a library that allows the development of a file system interface without the need of dealing with delicate kernel adaptations. As with many other distributed file systems [16,23,24], AwareFS uses FUSE to implement functions for reading and writing data and for metadata manipulation. This easiness provides a regular Unix mountable file system interface (e.g., running commands like cat, cd, ls, rm, mkdir) and interaction with POSIX-compliant applications including proven benchmark applications such as IOR [33] and fio [34].

Each AwareFS FUSE client instance works with multiple threads, allowing the concurrent manipulation of files. Whenever a file is opened, it is created a data structure with all information required in a context for interacting with the different available DSs and their LSs, the CS, and the MS. Such contexts are reusable, meaning that the overhead required in its creation is minimized. When files are closed the resources associated with the context are released if not required anymore [13].

Read requests are accelerated with a prefetch algorithm where data are read from the DS starting at the required offset up to the end of the chunk. Subsequent read operations will return right away if the requested block was already brought from the DS. Any DS with a valid copy of the required chunk is an option for being contacted.

When receiving a write request for creating chunks, the AwareFS FUSE client may use any DS in an equivalent container replication chain; the first step is deciding which DS to call, and the first option will be the DS hosted in the same server of the client, and if this is not possible, the first DS of the replication chain will then be used.

For rewrites of existing chunks, the DS to be used will be the one specified as the last known owner (i.e., lko) in the chunk's metadata. Write requests of existing chunks can only be honored by the DS that is the actual owner of the chunk. If the lko information is outdated and a DS that is not the actual owner is requested to rewrite a chunk, it replies with the information of the actual owner, which is then called and the lko information is then updated.

Buffering for write requests is used to minimize client–DS costly communications. Whenever a write request occurs, it is first stored in an AwareFS FUSE client local buffer, which is sent over to the equivalent DS only if the next block being written is not in sequence. With this mechanism, write requests are grouped before being sent over to the DS.

Metadata operations like unlink, rename, readdir, or rmdir are dealt with by the AwareFS FUSE client which translates the request in equivalent MS RPC requests.

6. Results and Discussion

Different tests evaluated various aspects of the AwareFS I/O characteristics and efficiency. We divide the AwareFS evaluation into two different approaches, as follows:

The first approach was to compare the performance gain of its local-write protocol to the performance loss of a distributed file system that uses a remote-write protocol. The distributed file system used with a remote-write protocol was MapR-FS 6.1, which is a commercial distributed file system that does not take advantage of local copies while writing data.

The aim of the second approach was to evaluate the scalability of AwareFS and how the local-write protocol would improve the performance, especially in random-write workloads, using different cluster sizes and block sizes used for writing data. For all tests, containers were created with up to 256 chunks of 64 MB each. Confidence intervals were calculated with a 5% significance level with different sample sizes for each set of tests.

6.1. I/O Evaluation

To understand the correct behavior of I/O operations in AwareFS, we used two well-known benchmark tools: IOR [33], a parallel I/O benchmark tool that can be used to test distributed file systems, and fio [34], another benchmark tool with several options to configure different I/O profiles. IOR is particularly useful due to its ability to run read and write operations in different hosts in a synchronized fashion, but for testing random access on large files, fio is best due to its ability to run operations during a specified amount of time [23].

6.2. Comparing the Local-Write Protocol with a Commercial File System Remote-Write Protocol

To guarantee consistency, directing all write requests to just one of the available copies may be a strategy to enforce the sequence of write operations in distributed file systems, and this approach is referred to as a primary-based write protocol [25]. In [25], the authors divide the primary copy protocols into two categories:

- Remote-write protocols: the write operation is forwarded to the node with the primary copy;
- Local-write protocols: the primary copy migrates to the node that initiated the write operation.

Considering distributed file systems used for Big Data, the write operations divide every file into chunks that are replicated throughout the cluster. In HDFS, the distributed file systems most widely used for Big Data processing; the semantics do not even allow programs to rewrite data. On the other hand, for MapR-FS, another commercially used Big Data file system, data can be rewritten using a primary copy remote-write protocol, where just one node manages the primary copy of each chunk, meaning that just one fixed node will attend every write request, causing nodes with a secondary copy to redirect write requests to the node with the primary copy, causing data to travel over the network.

Big Data processing frameworks like Hadoop spread the processing tasks throughout the cluster considering data locality, which means that tasks involving rewrite operations should be located in nodes capable of writing data to physical disks. Tasks involving rewrite operations will exploit data locality only on nodes with the primary copy, especially in file systems employing remote-write protocols.

For this set of evaluation tests, the objective was to reproduce the comparison showed in [13], where the authors proposed a test where a set of files were created using a single node, then rewritten in parallel by different nodes, forcing the situation where file systems using remote write protocols have to send all data over the network to be written to disk using the node with the primary copy, which does not happen for AwareFS and its local-write protocol.

6.2.1. The Experimental Setup

For evaluating how the I/O performance can benefit of a local-write protocol, we created a cluster of six virtual machines spread throughout three different physical hosts. The hardware used was a Dell EMC vxRail cluster with four nodes running VMware vSphere 7.0.3 (Table 2).

Table 2. Hardware and software specifications.

Component	Hardware Characteristics	Software Characteristics
Physical Hosts	vxRail P570 nodes with 1×Intel Xeon Silver 4110 @2.10 GHz, 8 cores (16 logical processors), 127.62 GB RAM, 4× 1.09TB HDD SAS Disks, and 2× 10GbE network interfaces	VMware ESXi version 7.0.3. Vmware vSphere vSAN 7.0.3 storage system
Switches	Cisco Nexus 3000 with 48× 10GbE	NX-OS version 6.0(2)U3(1)
Virtual Machines	6 vCPUs, 32 GB RAM, 1 vNIC and 300 GB virtual disks	CentOS 7.7.1908 using XFS

The six virtual nodes were distributed throughout the servers as depicted in Figure 9 with Data Services and Locking Services in all nodes. The Metadata Service and Container Service were installed only in Virtual Node 1.

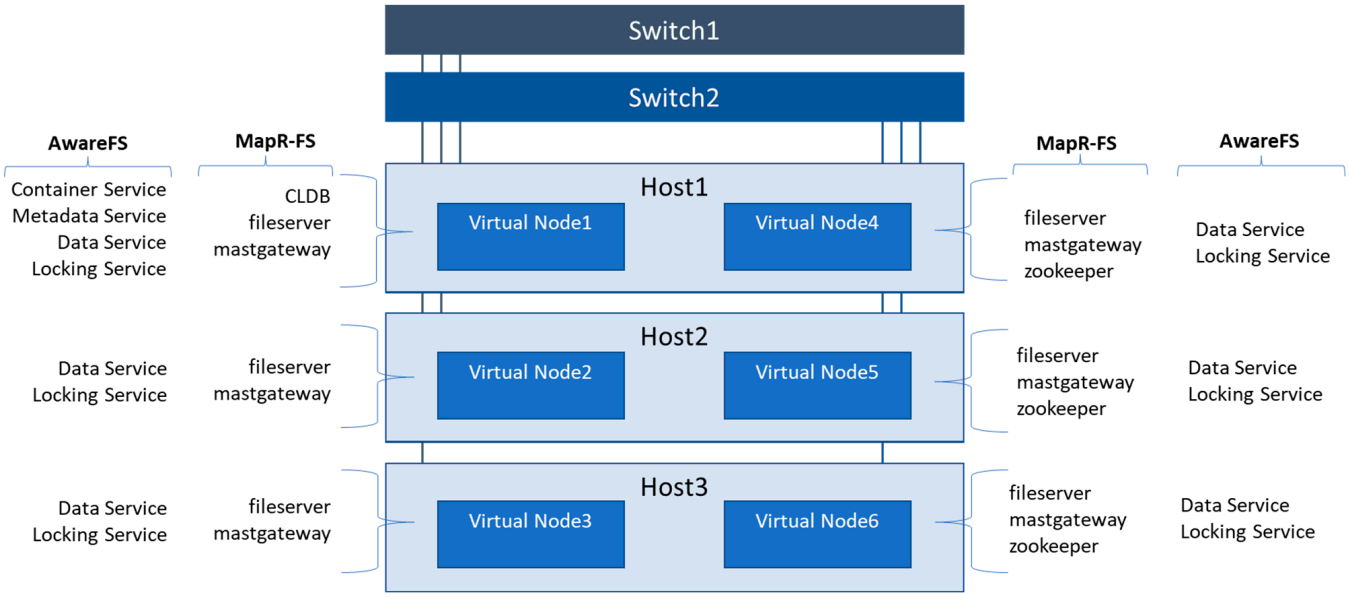


Figure 9. Host disposition and connection along with virtual machine and service distribution.

The bandwidth observed for communication among virtual nodes is shown in Table 3, measures obtained with the iperf benchmark. The communication between virtual nodes collocated in the same physical host is naturally higher as it does not go through the physical network.

The utilized distributed file systems were AwareFS and "MapR-FS version 6.1.1". Both were configured to drive reasonable comparisons. The strategy was to use some cautions:

- Compression was disabled for MapR-FS directories;
- Since AwareFS writes to regular Linux files, MapR-FS was installed using block-based storage on top of Linux regular files;
- The client writeback cache was disabled to make AwareFS and MapR-FS acknowledge write requests only after a date is written to storage servers;
- Chunk size was configured to 64 MB for both AwareFS and MapR-FS.

The IOR benchmark (version 3.3.0) was configured to use a 1 GB file per process without using client page caching. Write requests had 128 KB, running fsync upon POSIX write close.

Table 3. Iperf benchmark obtained for different node communications.

Source	Target	Data Transferred (GB)	Transfer Rate (Gbps)
Virtual Node 5	Virtual Node 3	2.74	4.71
Virtual Node 5	Virtual Node 4	5.49	9.43
Virtual Node 5	Virtual Node 2	15.50	26.70
Virtual Node 4	Virtual Node 3	5.49	9.44
Virtual Node 4	Virtual Node 2	5.49	9.43
Virtual Node 3	Virtual Node 2	5.49	9.43
Virtual Node 6	Virtual Node 2	5.18	8.89
Virtual Node 6	Virtual Node 4	5.48	9.42
Virtual Node 6	Virtual Node 3	15.10	25.90
Virtual Node 6	Virtual Node 1	5.48	9.42
Virtual Node 6	Virtual Node 5	5.48	9.41
Virtual Node 1	Virtual Node 4	15.10	26.00
Virtual Node 1	Virtual Node 3	5.49	9.43
Virtual Node 1	Virtual Node 5	5.49	9.43
Virtual Node 2	Virtual Node 1	3.03	5.28

6.2.2. About the Tests

For these tests, MapR-FS played the role of the file system with remote-write protocol, while AwareFS assumed the role of the file system that uses a local-write protocol. As proposed in [13], we used the POSIX clients of both file systems to rewrite six files in two different forms, using different nodes of the cluster:

- Concentrated write: All six files were created and rewritten using just one node of the cluster, with chunk replicas being sent to different nodes in the cluster. The purpose was to show sequential write performance when using a single node as the gateway for the cluster.
- Distributed write: All six files previously created on the concentrated write test were rewritten by different nodes (i.e., DS1 writes the file F1, DS2 writes F2, . . ., DS6 writes F6). Since primary copies were all stored in DS1, this time, write requests were all initiated in parallel in different nodes, many of them storing secondary copies of the modified chunks. In the case of using a remote-write protocol, every write request causes data to be sent to DS1, while write requests are attended to locally whenever possible when using a local-write protocol. The goal here was to demonstrate how beneficial writing directly to the local copies can be, as it avoids the need to send data through the network.

As described in [13], to illustrate the test concept, Figure 10 displays the arrangement of chunks for a test involving four files and four nodes. In this representation, each cell corresponds to a chunk, each column constitutes a set of chunks composing a file, and each group of four columns represents the chunks in a DS. Solid-colored cells denote primary copies, while light-colored cells denote replicas. As mentioned in [13], the arrows on top indicate the Data Service generating the write requests.

Upon closer examination of the behavior of write requests imposed on the first chunk, for instance, the first test (concentrated write) is intended to assign ownership of the chunks of all files to the first node, where all primary copies were initially situated. Figure 11 illustrates where write requests are processed for a single chunk, depicting the placement of primary copy ownership and nodes acting as replica sources. In Figure 11a, we see that all files are written on the first node, and each chunk is replicated to two other nodes with the first node as the source. Since the example depicts only a single container with two replicas, the fourth node does not have any copies, as the copies were distributed among the first three nodes. In Figure 11b, write requests are initiated in all four nodes. All data must travel to the first node because the primary copy ownership is fixed, and all primary copies are located in the first node. The first node serves as the source for all replication workloads. In Figure 11c, we see the behavior of write requests with a local-write protocol,

where the primary copy ownership moves to the node requesting the write, avoiding data movement between nodes by leveraging the use of local copies whenever available. For the situation depicted in Figure 11c, only requests initiated by the fourth node cause data to be sent over the network, as there is no local replica of the required container. Also, replication workloads use different nodes as sources for the situation depicted by Figure 11c.

(a)

Files written by DS1.
DS1 owns the primary copy.

			DS1				DS2				DS3				DS4			
			F1 ↓	F2 ↓	F3 ↓	F4 ↓	F1	F2	F3	F4	F1	F2	F3	F4	F1	F2	F3	F4
C1	1,2,3	chunk1																
C2	1,3,4	chunk2																
C3	1,2,4	chunk3																
C4	1,3,2	chunk4																
C5	1,4,2	chunk5																
C6	1,2,3	chunk6																
C7	1,2,4	chunk7																
C8	1,3,4	chunk8																

(b)

Each file rewritten by each DS.
Ownership of the primary copy relocated in local-write protocols.

			DS1				DS2				DS3				DS4			
			F1 ↓	F2	F3	F4	F1	F2 ↓	F3	F4	F1	F2	F3 ↓	F4	F1	F2	F3	F4 ↓
C1	1,2,3	chunk1																
C2	1,3,4	chunk2																
C3	1,2,4	chunk3																
C4	1,3,2	chunk4																
C5	1,4,2	chunk5																
C6	1,2,3	chunk6																
C7	1,2,4	chunk7																
C8	1,3,4	chunk8																

Figure 10. (**a**) Chunk distribution on concentrated write test, initial primary copy placement; (**b**) chunk distribution on distributed write test, primary copy disposition after ownership relocated.

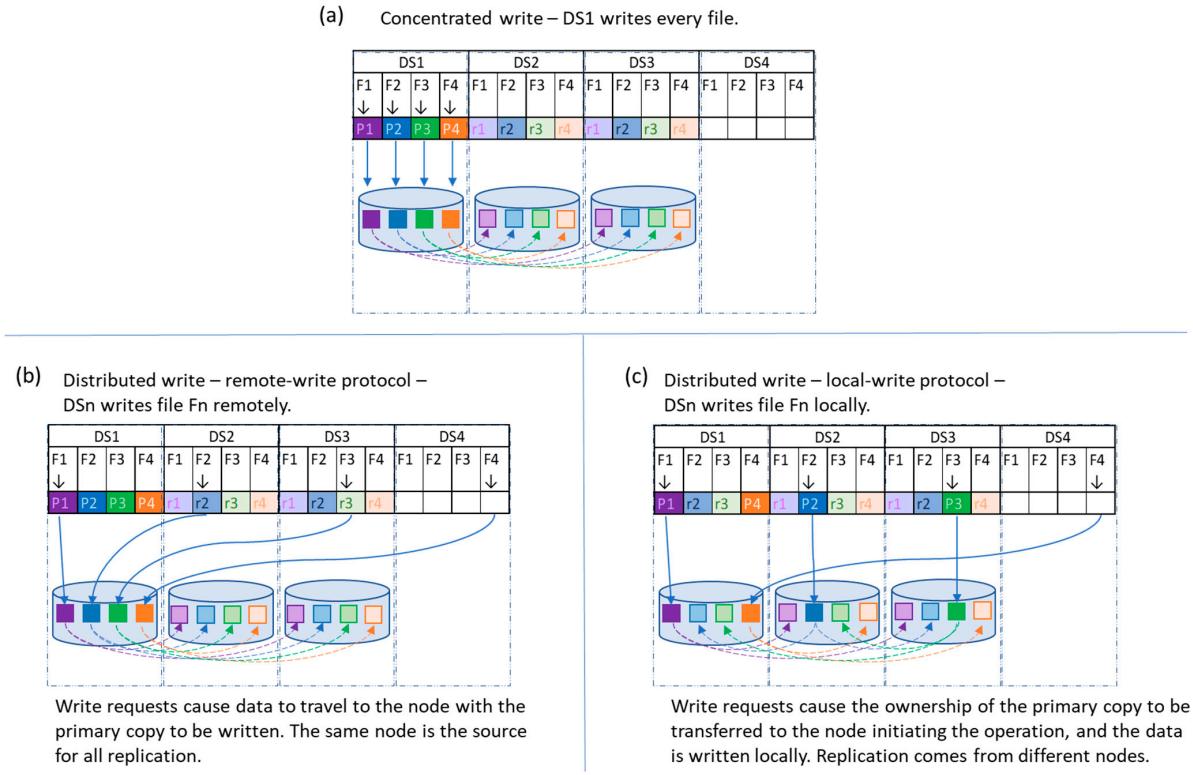


Figure 11. Write request behavior during tests: (**a**) initial primary copy disposition; (**b**) final primary copy disposition and write behavior for remote-write protocol; (**c**) final primary copy disposition and write behavior for local-write protocol.

Analyzing Figure 11, for write-based workloads, data locality would be leveraged only for the first node if a remote-write protocol is used, while three nodes would leverage data locality if a local-write protocol is used.

Similar to Figures 10 and 11, in our tests, we used a set of six files to have a better vision of how primary copy ownership migration would influence the sequential write performance on a six-node cluster. As proposed in [13], the evaluation was performed in two steps: first, running the concentrated write test using IOR to create and fully rewrite the six 1 GB files only on the first node, and then, running the distributed write test using IOR to fully rewrite the same six 1 GB files using all six nodes in parallel, each node dealing with a specific file.

A default MapR-FS install has different characteristics that would not permit us to compare its remote-write protocol with the AwareFS local-write protocol. To guarantee a reasonable comparison, we disabled the MapR POSIX client writeback cache and data compression. Also, the persistence mechanism used in MapR-FS writes data straight to block devices, while AwareFS in its initial implementation still uses regular files on top of the Linux XFS file system, so the comparison was made possible by configuring MapR-FS to write to Linux loop devices (pseudo devices accessible as block devices but persist data to regular files). With these specific configurations, it was possible to compare the performance differences for both MapR-FS and AwareFS caused by the write workloads over files created remotely.

For tests ran using AwareFS, Node 1 was also used for running MS and CS, which are, respectively, the Metadata Service and the Container Service. Similarly, for tests ran using MapR-FS, Node 1 was also used for the MapR container database (CLDB).

6.2.3. Test Results

After running the IOR tests with both the local-write protocol and remote-write protocol, we can observe the performance difference between the concentrated and distributed write tests in Figure 12 and Table 4. Each test was executed 100 times, measuring the write speed, and then calculating the arithmetic average, the standard deviation, and the 95% confidence intervals, using a normal distribution.

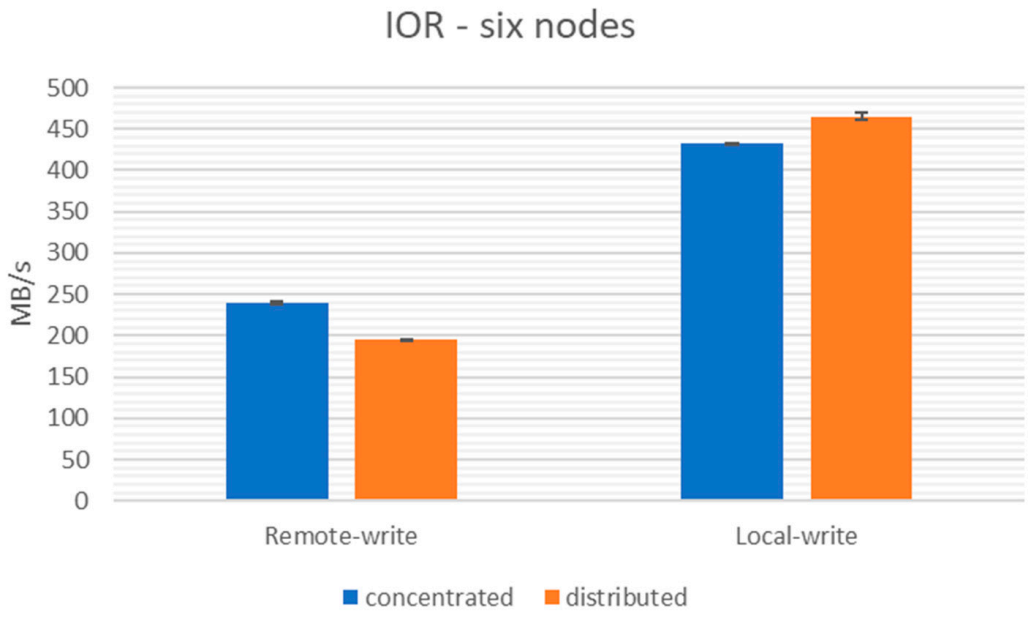


Figure 12. IOR write test results: performance loss with remote-write in distributed writing and performance gain with local-write in distributed writing.

As anticipated, upon examining the results of the remote-write protocol, it becomes apparent that, despite the focused write test utilizing only DS1 for the entire workload,

the overall performance is 18% lower when all six Data Services are concurrently engaged for the same purpose. This discrepancy is attributed to the substantial data movement occurring from other Data Services to DS1 when write requests are processed in parallel across all Data Services.

Table 4. Average transfer rate, standard deviation, and 95% confidence intervals (normal distribution).

Test	Write Protocol	Transfer Rate (MB/s)	Std. Dev.	Conf. Int.
Concentrated	Remote-write	238.978586	10.16399	[236.9865: 240.9707]
Concentrated	Local-write	431.830808	7.730645	[430.3156: 433.3460]
Distributed	Remote-write	195.54899	2.975474	[194.9658: 196.1322]
Distributed	Local-write	465.216869	20.08817	[461.2797: 469.1541]

In a real-world scenario involving Big Data processing, if data persistence predominantly relies on a single DS, especially when this workload coincides with data processing, the heightened utilization of DS resources for handling write requests can significantly impact the final performance of the Big Data workload. Conversely, when write requests are distributed across all six DSs, the observed behavior of the remote-write protocol underscores that the requisite data movement negatively influences performance, thereby impacting the final output of real-world Big Data workloads.

Focusing on the right side of Figure 12, the observed behavior for AwareFS and its local-write protocol goes counterclockwise to what is explained about the results observed for the remote-write protocol. We observed a performance improvement of about 8% when comparing the concentrated write test performance to the distributed write test, meaning that if we spread the Big Data processing and its write requests throughout the cluster by using a local-write protocol, the resulting performance would improve accordingly by leveraging the use of local copies in all DSs as primary copies, not raising the data movement between nodes. It is reasonable to expect higher rates of performance improvement in real-life situations than the observed 8% rate, if we consider that the tests used virtual machines in a VMware cluster with a 10GbE network, where every two DSs were running on a same host not even requiring the use of any network physical hardware.

Considering the influence of the write protocol on common Big Data workloads, it is important to compare the distributed write performance of both local-write and remote-write protocols. The AwareFS local-write protocol improved the distributed write performance by about 140% compared to the performance achieved by the MapR-FS remote-write protocol. Also, the difference of both protocols, observed for the concentrated write tests, exposes a performance gain of 80% when using the two different distributed file systems with the same environment under the same constraints, although the MapR-FS setup was not the one commercially used, e.g., avoiding writing directly on block devices.

The local-write protocol of AwareFS caused the relocation of the primary copies of 34 chunks out of the 96 in total, which means that it was not necessary to send the data to the DS1 to rewrite 35% of the chunks, writing locally in other DSs with valid copies, avoiding the network traffic that would be created by a remote-write protocol.

After the initial assessment of AwareFS as outlined in [13], focusing exclusively on sequential writes with 128 KB write requests, a comprehensive reengineering effort was undertaken, optimizing all components. The current observations indicate that the enhancements in Data Service concurrency, coupled with refinements in lock management, have yielded substantial benefits. The performance improved by approximately 56% in the distributed write test when compared to the metrics reported in [13]. Additionally, there was a 2.5-fold reduction in the statistical uncertainty of the results.

6.3. Random I/O and Scalability Evaluation

The evaluation tests described so far were all using sequential writes on a fixed-size relatively small cluster, but different applications may use I/O requests of different block

sizes and may require different cluster sizes, especially Big Data workloads. To best evaluate AwareFS and its local-write protocol, it is important to use different cluster sizes and other write profiles. This section describes a set of evaluation tests where AwareFS is used with different I/O profiles, going from reading to writing with different block sizes, i.e., with different numbers of bytes per write operation and with cluster sizes ranging from 8 to 36 nodes. With this approach, it will be possible to evaluate how the overall performance changes, varying the number of bytes transferred per operation along with changes in the number of nodes working. Since the used infrastructure would not allow the installation of any other commercial distributed file system, we observed the advantages of using the AwareFS local-write protocol by running tests with the primary copy ownership migration disabled, then repeating the tests after it was reenabled and comparing the obtained results.

To evaluate the time spent while reading and writing to disks, we configured the DSs to not acknowledge I/O operations before writing data to disks. In all tests, FUSE AwareFS clients had to wait until the data to be written on disks before resuming operation in each I/O thread. This is key to understanding how AwareFS can behave in situations where I/O operations must be committed to disks to guarantee data persistence.

6.3.1. The Effects of Page Cache

As explained in [33], if write operations are followed by reads of the same files using the same hosts, all read performance is dramatically raised due to the cache capabilities present in the host operating systems. Usually, different operating systems have means to keep in their memory the data being written to disk in a way that modifications applied to the same region will be made in memory, taking advantage of memory access speed that is much faster than accessing the disk. The use of such page cache capability is very beneficial and will make read measures observed in benchmarks usually much higher than writes. Since the page cache is beneficial if the data being read were put into the cache by a previous read/write operation, such a benefit is not observed for files written in other nodes. So, the chosen strategy used to evaluate the underneath file system performance is accessing the files created by a neighbor node [33].

6.3.2. The Experimental Setup

We ran AwareFS on top of up to 72 nodes of an HPC cluster (Table 5) without enabling the InfiniBand low-latency high-performance connectivity. Nodes were connected to a "top of rack" switch, and racks were connected at 10 Gbps through an aggregation switch.

Table 5. Hardware and software specifications—HPC cluster.

Component	Hardware Characteristics	Software Characteristics
Physical Hosts	Dell Technologies PowerEdge C6620 servers with 2× Intel Xeon Platinum 8480+ processors of 2.0 GHz, 56 cores per socket (8 NUMA regions), 512 GB RAM at 4800 MHz, 1× 1GbE network card, NVMe disks.	RedHat Linux 8.6 operating system with 407 GB XFS file system
Aggregation Switch	Dell Technologies Z9100-ON with 32× 10GbE	OS9 operating System
Top of Rack (TOR) Switches	Dell Technologies S3048-ON with 48× 1GbE	OS9 operating System

The 1GbE network interface of every node was advertising full duplex mode with auto-negotiation on. The detected speed was 1000 Mb/s, and iperf showed a 0.96 Gbps throughput for all nodes.

The deployed MPI version was OpenMPI 4.0.2, along with OpenMP version 201511. The Thrift version was 0.12.0.

OpenMP was initialized with the following parametrization:

- OMP_NUM_THREADS = 39
- OMP_PROC_BIND = spread
- OMP_PLACES = cores(39)

- GOMP_CPU_AFFINITY = 1–39
- GOMP_DEBUG = 1

SLURM was the job scheduling system, and all AwareFS parametrization files were created automatically by the submission script, always assigning the Container Service and Metadata Service to the first node, while all other nodes were dedicated for running the Data Service and Locking Service.

6.3.3. Sequential I/O Leveraging Data Locality

For evaluating the performance of AwareFS while reading and writing large files, we used the fio benchmark tool to read a 1 GB file sequentially from its start to its end and then rewriting the file the same way. Every node in the cluster has its own file, created by itself right before the tests, and it is the file used with fio. At the same time, all nodes ran the fio benchmark for reading their own files, allowing the observation of how AwareFS could be capable of managing several I/O operations in parallel. This was followed by another fio execution to rewrite the files, using the same approach. Since each DS was responsible for reading then writing a file created by itself, all caches available in FUSE and in the operating system were used at their full potential, leveraging the advantages gained with data locality. This was done to evaluate a real scenario where the data is created and manipulated by each node of a cluster using a Big Data distributed processing framework such as Hadoop. To evaluate the scalability of AwareFS, we repeated the same tests increasing the nodes available in the cluster while increasing the number of files and total number of containers proportionally. Each node ran four threads of fio tests, pushing the number of files simultaneously managed by AwareFS up to a number equal to the number of nodes in the cluster multiplied by four. After speed measurements, the arithmetic average, the standard deviation, and the 95% confidence intervals were calculated using a normal distribution. Measurements were taken in each half-second interval. Sample sizes varied from 3833 measurements for the 8-node cluster up to 77,697 measurements for the 36-node cluster.

Figure 13 shows that the transfer rate raises considerably if we compare the same tests using 4 KB and 128 KB blocks, but the difference is quite small if we compare results for the 128 KB block size with the results for the 1 MB block size. This best behavior observed when using 128 KB as the block size is expected as the default configuration of FUSE is optimized for read and write operations of 128 KB blocks [31,32]. It is also clear that performance increases linearly as the cluster size increases along with the number of files and threads used for running fio. Since files were created locally, i.e., files were created by the same node running the DS and the FUSE client, the better read performance is expected due to the page cache effects. This shows how AwareFS improves sequential read performance by leveraging local OS cache features in a common situation of a cluster node consuming data created by itself.

6.3.4. Random Reads and Mixed Random Reads and Writes

Besides the large-volume reads and writes, some applications may need to read and/or write to distinct locations of Big Data files in a random fashion. Measuring the I/O operations per unit of time is an option to evaluate the performance for this kind of I/O profile. For evaluating random I/O performance in terms of IOPS, fio is also a valuable tool because of its ability to measure the total number of I/O operations after a specific amount of time. With that said, we measured the AwareFS random I/O performance by starting simultaneously a set of fio threads on each node and aggregating the obtained results [23]. We configured all fio executions performing random I/O operations to run reads or writes varying the offset randomly in a 64 MB area of the file (the size of a chunk), changing to a different area only after 64 MB of data was read or written. We chose this control of random offsets in areas because in real-life Big Data workloads, jobs will process files in chunks [14], which makes it less likely the random access jumping from one chunk to another. After obtaining IOPS measures, the arithmetic average, the standard deviation,

and 95% confidence intervals were calculated using a normal distribution. For random I/O evaluations, the duration of tests started at 60 s for an 8-node cluster, rising to 270 s for the 36-node cluster, and further to 540 s for the 72-node cluster. Measurements were taken in each half-second interval. Sample sizes varied from 442 measurements for random read tests with the 8-node cluster to up to 68,912 measurements for the random write tests with the 36-node cluster.

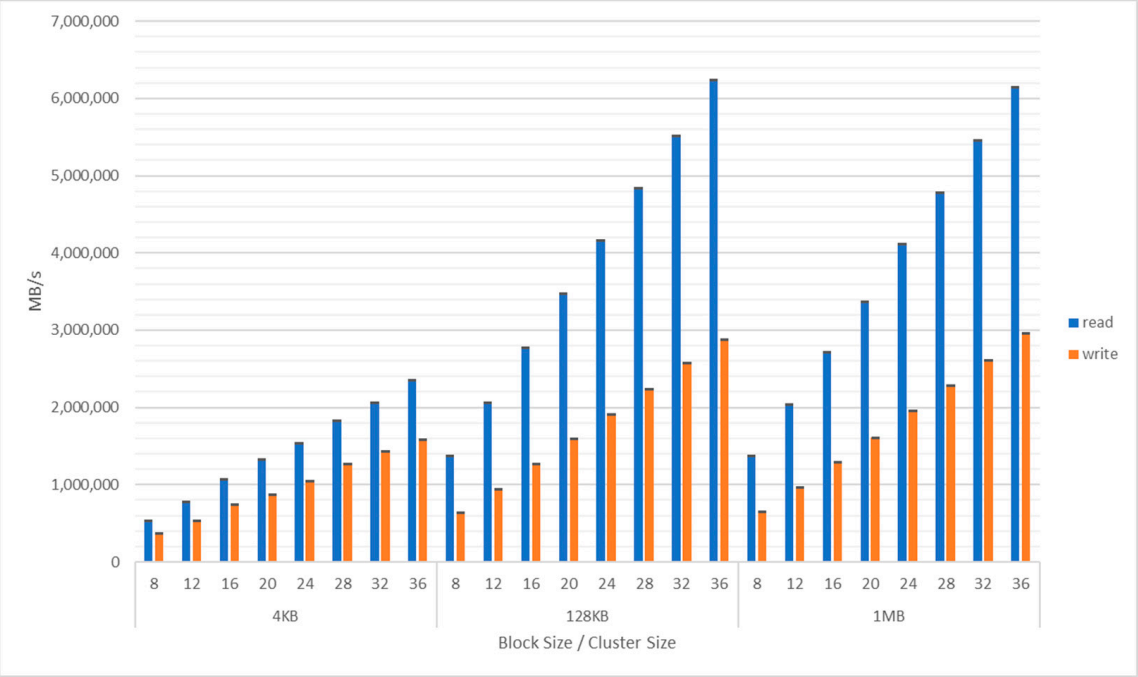


Figure 13. Sequential write transfer rate with 4 threads per node: transfer rates for different block and cluster sizes.

Random read operations were evaluated by running four fio threads simultaneously in each node but reading the file created in its neighbor node. The idea of not reading a chunk locally created is to cancel the effects of the page cache by reading a file written by a neighbor node.

The results observed in Figure 14 show how the number of random read requests served scales close to linearly, as denoted by the linear-trending dotted lines. It is clear in Figure 14a that the linearity is less evident when using 4 KB blocks, as the number of files used and nodes in the cluster grows. This is due to the randomness of read requests affecting the effectiveness of buffering in the AwareFS FUSE client. In its initial configuration, the AwareFS FUSE client uses a "read ahead" buffering strategy where all data from the required read offset to the end of the file chunk are read and buffered, making subsequent read requests to be fulfilled without communicating with any DS if the required data are already locally buffered. The number of IOPS observed for random reads grow as the cluster size increases from 8 nodes to 36 nodes, considering smaller block sizes like 4 KB (Figure 14a) and larger block sizes like 1 MB (Figure 14b). Additionally, for an intermediary block size like 128 KB, Figure 14c demonstrates the growing behavior for the number of IOPS, continuing up to the 72-node cluster.

Another important I/O profile involves alternating random reads with random writes. This I/O profile was observed using fio, similar to the approach for random reads but specifying the "randrw" pattern that mixes random reads and writes. As done for other I/O patterns, the duration of each test in seconds was the cluster size multiplied by 7.5. The offset for read/write requests varied randomly in a 64 MB area of the file. Clusters ranging from 8-nodes to 72-nodes were benchmarked for reading/writing locally created files (Figure 15). Error bars were added to indicate a statistical significance with 95% confidence

intervals, calculated using a normal distribution. As fio records measurements every half second, the number of samples varied from 2055 for the 8-node cluster, to 292,884 for the 72-node cluster.

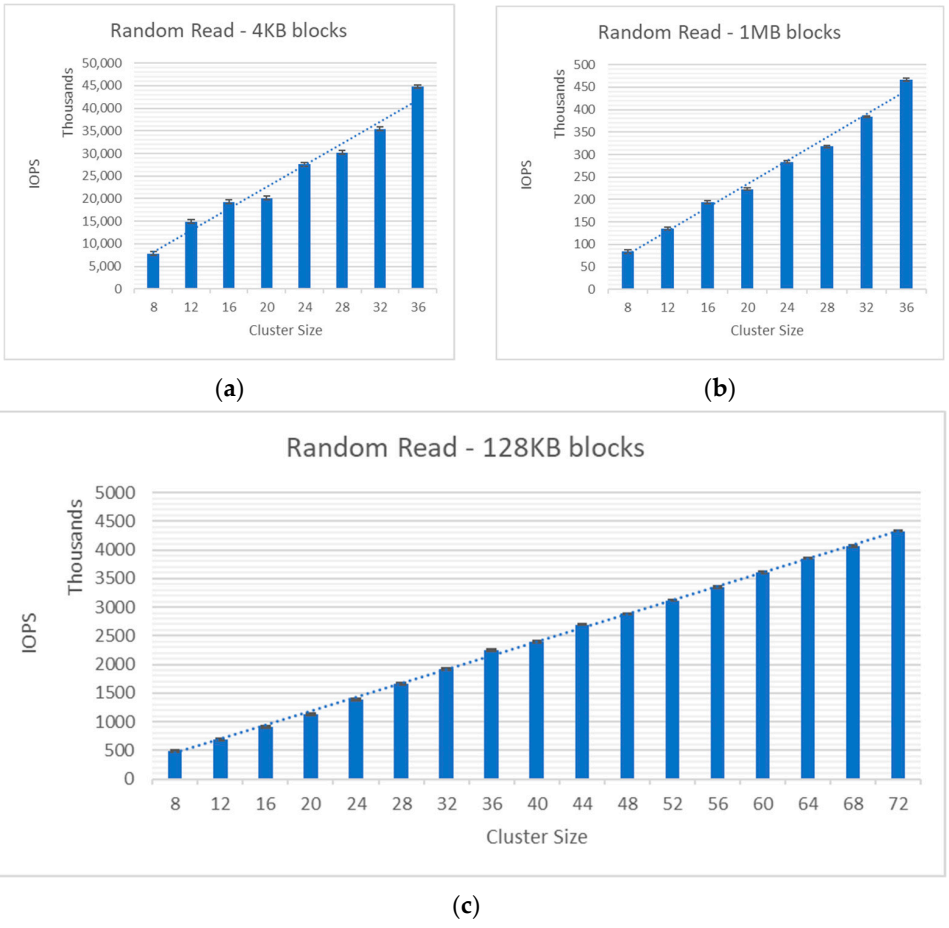


Figure 14. Random read of a file created by a neighbor node (**a**) using 4 KB blocks; (**b**) using 1 MB blocks; (**c**) and using 128 KB blocks (up to 72 nodes).

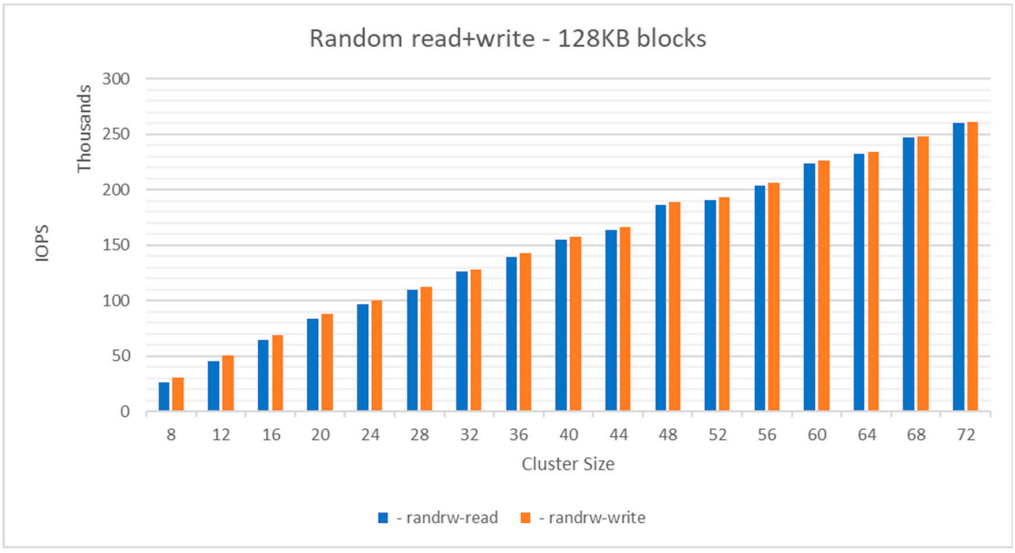


Figure 15. Mixed random reads and writes with 128 KB blocks and different cluster sizes: better IOPS rate was observed for writes.

The current implementation of the AwareFS FUSE client employs a single buffer for both reads and writes, ensuring consistency by discarding the buffer when transitioning between read and write operations. The AwareFS read-ahead strategy optimistically requests more data than initially requested, anticipating the next operation to be read. However, this strategy loses its benefit when switching to a write operation, as the buffer is discarded. On the other hand, write requests are buffered and transmitted to the Data Service in 64 MB chunks. The buffers used in write operations are flushed either upon receiving a write request with an out-of-order offset or when transitioning to a read operation. The observed behavior in Figure 15, where read operations show a lower IOPS measurement compared to write operations, can be attributed to this single-buffer approach. Nevertheless, it is evident that performance improves as the cluster size increases.

6.3.5. Strategy for Evaluating the Local-Write Protocol with Random Operations

With AwareFS, all write operations are consistently managed by a primary-based write protocol [13]. With this strategy, the order of write operations is assured by concentrating all requests in only one of the copies a piece of data may have [25]. As stated in [13], AwareFS uses a primary-based local-write protocol where the primary copy role migrates to the node that initiated the write operation, instead of redirecting the write requests to the node which currently owns the copy acting as the primary. With this protocol, write operations will require less network traffic, as they will be done locally in a DS if it has a valid copy of the chunk being written. To fully evaluate the efficiency and performance of random writes with AwareFS, as a first step, it is important to take performance measures forcing the use of a remote-write protocol, simply by writing in chunks created by a neighbor node, without the primary copy role migration capability. Then, as a second step, the same operations can be repeated using the local-write protocol, and the new performance measures can be compared to the first observation, to identify the advantages of migrating the primary copy role and then leveraging the local copy for the write. Since the primary copy role had migrated in the second step, the same operations can be repeated once more as a third step, still using the local-write protocol, and this will show the performance of write requests done leveraging the local copy without the overhead required for changes of primary copy ownership. This way, to better evaluate the scalability of AwareFS, the performance of random write operations was observed with three different write block sizes, 4 KB, 128 KB, and 1 MB, and these observations were a compound of three steps:

1. Move disabled: Chunks created on a neighbor node are randomly written after disabling the primary copy ownership migration capability;
2. Move enabled: After reenabling primary copy ownership migration, chunks created on a neighbor node are written randomly;
3. Move enabled—2nd round: Chunks created on a neighbor node are randomly written, and the ownership of the primary copy may have been migrated to the local node during the previous test.

To enable comparison, the same write operations randomly created in Step 1 are repeated in Steps 2 and 3.

The following sections will describe these three steps initially in a 12-node cluster using 128 KB blocks, then using 4 KB blocks and 1 MB blocks, and finally the three steps are used to measure performance in different clusters sizes.

6.3.6. Random Write Performance with 128 KB Blocks

The first evaluated block size was 128 KB, the default size used by FUSE. We ran fio benchmarks for random writes of 128 KB blocks for 90 s, using the mentioned approach of the three consecutive steps. The 128 KB writes were done randomly in a 64 MB area of the file (the size of a chunk), changing to a different area only after 64 MB of data was written. This write profile was chosen because partitioning the data is a common practice for applications that may take advantage of data locality for improving execution

performance. Hadoop workloads, for example, will use 64 MB as the size of input, split to match the size of a block of the underlying distributed file system [14].

Figure 16 shows the performance gain for both set of measures taken with the primary copy ownership migration enabled. Even with the overhead imposed by migrating the primary copy role, thanks to the reduction in the volume of data sent over the network in a write request, an improvement of 9% is noticed with the first measures taken right after enabling the primary copy ownership migration. Also, we observed an improvement of 13% for the number of achieved IOPS, when comparing the higher measure obtained with the local-write protocol with values observed when it is disabled. The extra improvement with the third step, also performed with the primary copy ownership migration enabled, is due to the write being performed leveraging the local copy without the overhead required for changing primary copy ownership, since most of the ownership migration had already happened.

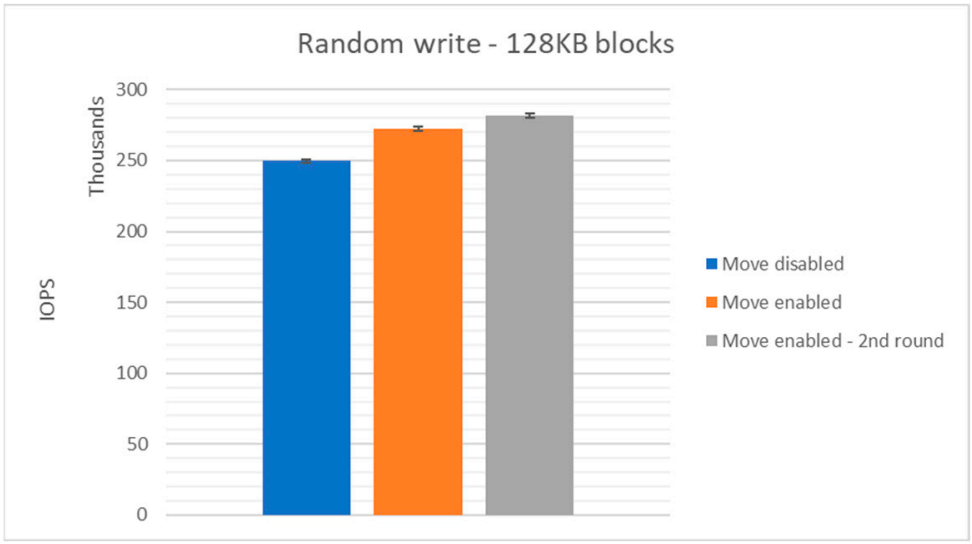
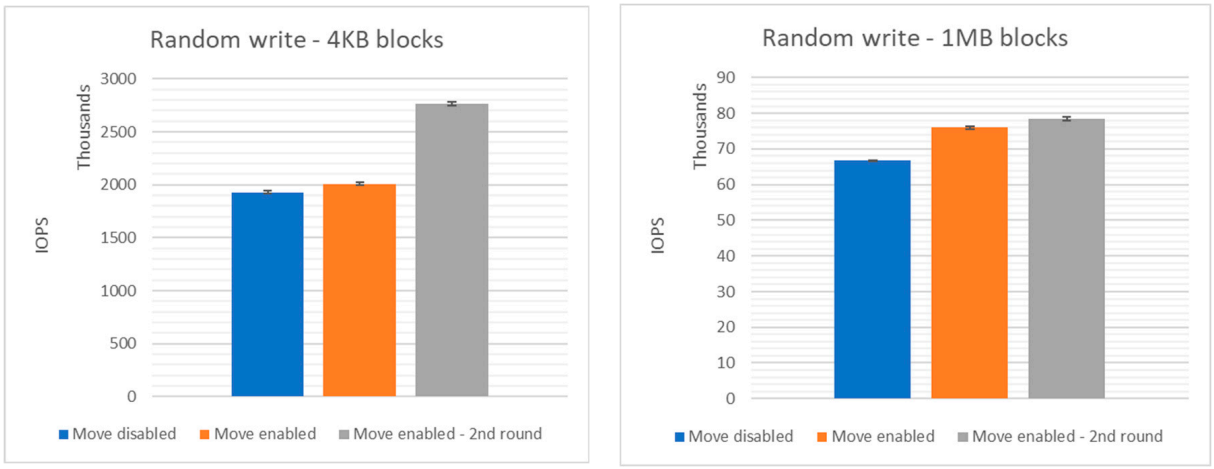


Figure 16. Primary copy ownership migration influence in random writes for 128 KB blocks—12-node cluster: better IOPS rate observed with the local-write protocol enabled.

6.3.7. Random Write Performance with Different Block Sizes

Even though the configured FUSE block size is 128 KB, write requests of different block sizes can be made depending on the Big Data application. If the application updates the files randomly in smaller pieces in a concentrated chunk, the number of IOPS may increase considerably due to cache hits. When larger blocks (e.g., 1 MB) are updated, the observed IOPS decreases due to longer disk write times and increased communication with a DS, resulting in fewer operations per second.

This time, the same three-step approach proposed and used with 128 KB blocks was repeated by running fio for 90 s with 4 KB blocks and 1 MB blocks. Figure 17 shows that while using a 12-node cluster, the observed increase in IOPS was of around 43% for 4 KB blocks and 17% for the 1 MB block size, while it was around 13% for 128 KB blocks (Figure 16). The obtained measures show that the local-write protocol is beneficial for the three evaluated block sizes with the 12-node cluster. The improvement observed with 4 KB blocks is more significant because the number of operations performed in 90 s is higher with smaller blocks, and the local-write protocol improves response rate per operation. Comparing results for 128 KB blocks and 1 MB blocks, the latter showed a greater improvement. This is due to more expensive data movement over the network being avoided when larger blocks are used, in addition to the active local-write protocol.



(**a**) (**b**)

Figure 17. Primary copy ownership migration influence in random writes for different block sizes—12-node cluster: (**a**) using 4 KB blocks; (**b**) using 1 MB blocks.

The 12-node cluster setup serves as a meaningful example to illustrate the statistical significance of observed performance differences. Table 6 summarizes all values used to create the charts in Figures 16 and 17, including error bars. The uncertainty estimation involved computing 95% confidence intervals using established statistical techniques for normal distributions.

Table 6. Performance gain of local-write protocol for a 12-node cluster in terms of IOPS, with 95% confidence intervals calculated using a normal distribution.

Observation Step	Block Size	IOPS Average	Samples	Standard Deviation	95% Confidence Interval (Normal Distribution)
Move disabled	4 KB	1,929,046	6537	605,447.00	[1,914,369.078:1,943,722.922]
Move enabled	4 KB	2,005,977	6970	571,088.00	[1,992,569.903:2,019,384.097]
Move enabled 2nd round	4 KB	2,762,293	7399	732,518.00	[2,745,602.095:2,778,983.905]
Move disabled	128 KB	249,789	5942	51,390.00	[248,482.347:251,095.653]
Move enabled	128 KB	272,589	5932	58,474.00	[271,100.975:274,077.025]
Move enabled 2nd round	128 KB	281,622	5989	63,949.00	[280,002.411:283,241.589]
Move disabled	1 MB	66,678	5505	8913.00	[66,442.553:66,913.447]
Move enabled	1 MB	75,944	5377	15,512.00	[75,529.384:76,358.616]
Move enabled 2nd round	1 MB	78,316	5473	19,774.00	[77,792.122:78,839.878]

6.3.8. Random Write Performance with Different Cluster Sizes

To show how the cluster size affects random write performance, we ran the evaluation with different block sizes but also varying the number of nodes used in the cluster and increasing the duration of the tests proportionally. Like it was done to evaluate the sequential I/O and the random read performances, we started four fio threads simultaneously in each node, randomly writing the file created in its neighbor node, causing AwareFS to manage an intense workload comprised of a total number of files equal to the number of nodes in the cluster multiplied by four, all of them randomly updated concurrently.

The chart in Figure 18 shows that the overall performance of random writes increases along with the cluster size. A similar behavior to the one described for the three block sizes (4 KB, 128 KB, and 1 MB) with a 12-node cluster is observed for clusters ranging from an 8-node cluster to a 36-node cluster. Additionally, the evaluations for the 128 KB block size were done increasing the cluster size up to 72 nodes, and the same behavior was observed even with larger clusters.

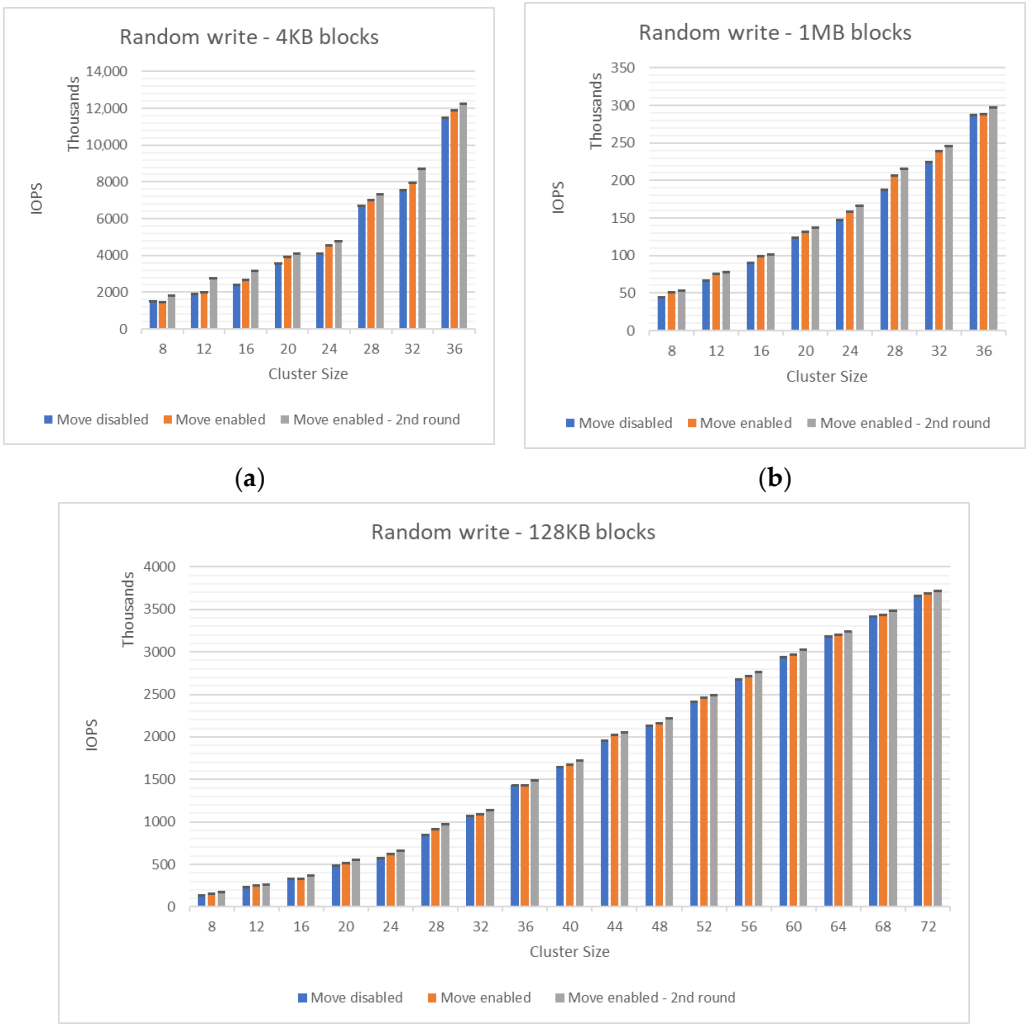


Figure 18. Primary copy ownership migration influence in random writes for different block sizes and different cluster sizes: (**a**) using 4 KB blocks; (**b**) using 1 MB blocks; (**c**) and using 128 KB blocks (up to 72 nodes). Sustained scalability and a better IOPS rate are observed with the local-write protocol enabled for all cluster and block sizes.

Comparing the initial primary copy ownership with the primary copy ownership after all evaluation steps, the ownership change rate decreased as shown in Table 7. Nevertheless, the performance for fio using the local-write protocol is still higher than observed when it is disabled, even for bigger clusters, showing how write performance can benefit from data locality with the primary copy ownership migration process of AwareFS.

Table 7. Total number of chunks vs. chunks whose primary copy ownership changed (1 MB blocks).

Cluster Size	Test Duration (s)	Total Chunks	Changed Owner	Change Rate
8	60	512	87	17.0%
12	90	768	104	13.5%
16	120	1024	113	11.0%
20	150	1280	80	6.3%
24	180	1536	76	4.9%
28	210	1792	69	3.9%
32	240	2048	54	2.6%
36	270	2304	49	2.1%

7. Conclusions and Future Works

Keeping data close to computational resources is key for Big Data challenges as moving large amounts of data from storage can take precious time. While high-performance computing storage systems have reached their maturity, Big Data distributed file systems have emerged enabling the exploitation of data locality for parallelizing computation with minimum data movement over a network. Now, advanced computation like scientific modeling or generative artificial intelligence systems require more robust storage, pushing data management technologies further ahead. Different distributed file systems have been developed to efficiently manage different I/O patterns. In most cases, storage resources are decoupled, and data are transferred to nodes specifically designated for processing. The Hadoop Distributed File System (HDFS), serving as the most popular distributed file system for Big Data, adopts a data-centric approach that exploits data locality, ensuring the proximity of processing to storage resources. However, its design imposes more constrained semantics. More robust distributed file systems for Big Data have been developed using primary-based protocols, enabling the rewriting of large files in just one of the available copies. This demands data movement even for writes over the replicas. In this work, we describe AwareFS, a distributed file system that facilitates the efficient use of large files across various I/O patterns. It employs a primary-based local-write protocol to exploit data locality, even for updates, and features a robust distributed lock management system to ensure consistency in a distributed manner. The efficiency of AwareFS was initially assessed by comparing its local-write protocol with the remote-write protocol of another distributed file system. In a small cluster with limited resources, the results indicated a performance improvement of about 8% after migrating the primary copy role and using the local copy for writing. In contrast, measurements with the remote-write protocol showed a decline of 18% in transfer rates. The AwareFS local-write protocol, complemented by its distributed lock management capabilities, facilitated random writing in various block sizes. It efficiently updated areas ranging from 4 KB to megabytes, ensuring a consistent and efficient process while handling requests from multiple clients simultaneously. The advantages of exploiting data locality during writes were demonstrated across clusters of varying sizes, resulting in a 43% improvement in measurements of IOPS for writes with a 4 KB size, underscoring the significance of minimizing data movement. The AwareFS client currently supports integration with traditional high-performance computing workloads, extending the advantages of a data-centric architecture to MPI-based scientific applications and emerging technologies for processing data stored in large files. Fault tolerance is yet to be fully developed in AwareFS, along with block-based storage management. Enhancements to the POSIX interface and refined client-side cache management will cater to other compatibility and performance requirements. Additionally, the integration of an HDFS protocol will position AwareFS as a storage system for commercial Big Data frameworks.

Author Contributions: Conceptualization, E.C.d.S. and L.M.S.; methodology, E.C.d.S. and L.M.S.; software, E.C.d.S.; validation, E.C.d.S., L.M.S. and E.T.M.; investigation, E.C.d.S.; resources, E.C.d.S.; writing—original draft preparation, E.C.d.S.; writing—review and editing, E.C.d.S., L.M.S. and E.T.M.; visualization, E.C.d.S.; supervision, L.M.S.; project administration, L.M.S.; funding acquisition, E.C.d.S. All authors have read and agreed to the published version of the manuscript.

Funding: This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior-Brasil (CAPES)-Finance Code 001.

Data Availability Statement: The data presented in this study are available on request from the corresponding author. The data are not publicly available as it is recommended to rerun benchmark tools for every utilized environment.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Bandi, A.; Adapa, P.V.S.R.; Kuchi, Y.E.V.P.K. The Power of Generative Ai: A Review of Requirements, Models, Input–Output Formats, Evaluation Metrics, and Challenges. *Future Internet* **2023**, *15*, 260. [CrossRef]

2. Baig, M.I.; Shuib, L.; Yadegaridehkordi, E. Big Data Adoption: State of the Art and Research Challenges. *Inf. Process. Manag.* **2019**, *56*, 102095. [CrossRef]
3. Rydning, D.R.-J.G.-J.; Reinsel, J.; Gantz, J. The Digitization of the World from Edge to Core. *Fram. Int. Data Corp.* **2018**, *16*, 1–28.
4. Blomer, J. A Survey on Distributed File System Technology. *J. Phys. Conf. Ser.* **2015**, *608*, 012039. [CrossRef]
5. Patgiri, R.; Ahmed, A. Big Data: The V's of the Game Changer Paradigm. In Proceedings of the 2016 IEEE 18th International Conference on High Performance Computing and Communications; IEEE 14th International Conference on Smart City; IEEE 2nd International Conference on Data Science and Systems (HPCC/SmartCity/DSS), Sydney, NSW, Australia, 12–14 December 2016. [CrossRef]
6. Thanh, T.D.; Mohan, S.; Choi, E.; Kim, S.; Kim, P. A Taxonomy and Survey on Distributed File Systems. In Proceedings of the 2008 Fourth International Conference on Networked Computing and Advanced Information Management, Gyeongju, Republic of Korea, 2–4 September 2008; Volume 1, pp. 144–149. [CrossRef]
7. Lee, S.; Jo, J.-Y.; Kim, Y. Hadoop Performance Analysis Model with Deep Data Locality. *Information* **2019**, *10*, 222. [CrossRef]
8. Wang, J.; Han, D.; Yin, J.; Zhou, X.; Jiang, C. ODDS: Optimizing Data-Locality Access for Scientific Data Analysis. *IEEE Trans. Cloud Comput.* **2020**, *8*, 220–231. [CrossRef]
9. Wang, F.; Oral, H.S.; Shipman, G.M.; Drokin, O.; Wang, D.; Huang, H. *Understanding Lustre Internals*; Oak Ridge National Lab. (ORNL): Oak Ridge, TN, USA, 2009.
10. Carns, P.; Lang, S.; Ross, R.; Vilayannur, M.; Kunkel, J.; Ludwig, T. Small-File Access in Parallel File Systems. In Proceedings of the 2009 IEEE International Symposium on Parallel & Distributed Processing, Rome, Italy, 23–29 May 2009; pp. 1–11.
11. Zou, J.; Iyengar, A.; Jermaine, C. Architecture of a Distributed Storage That Combines File System, Memory and Computation in a Single Layer. *VLDB J.* **2020**, *29*, 1049–1073. [CrossRef]
12. Rao, T.R.; Mitra, P.; Bhatt, R.; Goswami, A. The Big Data System, Components, Tools, and Technologies: A Survey. *Knowl. Inf. Syst.* **2019**, *60*, 1165–1245. [CrossRef]
13. Da Silva, E.C.; Sato, L.M.; Midorikawa, E.T. Distributed File System for Rewriting Big Data Files Using a Local-Write Protocol. In Proceedings of the 2021 IEEE International Conference on Big Data (Big Data), Orlando, FL, USA, 15–18 December 2021; pp. 3646–3655.
14. White, T. *Hadoop: The Definitive Guide*; O'Reilly: Springfield, MO, USA, 2015; ISBN 978-1-4919-0163-2.
15. Wang, K.; Zhou, X.; Li, T.; Zhao, D.; Lang, M.; Raicu, I. Optimizing Load Balancing and Data-Locality with Data-Aware Scheduling. In Proceedings of the 2014 IEEE International Conference on Big Data (Big Data), Washington, DC, USA, 27–30 October 2014; pp. 119–128.
16. Weil, S.A.; Brandt, S.A.; Miller, E.L.; Long, D.D.E.; Maltzahn, C. *Ceph: A Scalable, High-Performance Distributed File System*; USENIX Association: Berkeley, CA, USA, 2006; pp. 307–320.
17. Usman, S.; Mehmood, R.; Katib, I.; Albeshri, A. Data Locality in High Performance Computing, Big Data, and Converged Systems: An Analysis of the Cutting Edge and a Future System Architecture. *Electronics* **2022**, *12*, 53. [CrossRef]
18. Chowdhury, F.; Zhu, Y.; Heer, T.; Paredes, S.; Moody, A.; Goldstone, R.; Mohror, K.; Yu, W. I/O Characterization and Performance Evaluation of BeeGFS for Deep Learning. In Proceedings of the 48th International Conference on Parallel Processing, Kyoto, Japan, 5–8 August 2019; p. 10, ISBN 978-1-4503-6295-5.
19. Chandakanna, V. REHDFS: A Random Read/Write Enhanced HDFS. *J. Netw. Comput. Appl.* **2017**, *103*, 85–100. [CrossRef]
20. Sharma, A.; Singh, G. A Review on Data Locality in Hadoop MapReduce. In Proceedings of the 2018 Fifth International Conference on Parallel, Distributed and Grid Computing (PDGC), Solan, India, 20–22 December 2018; pp. 723–728. [CrossRef]
21. George, L. HBase—The Definitive Guide: Random Access to Your Planet-Size Data. 2011. Available online: https://learning.oreilly.com/library/view/hbase-the-definitive/9781449314682/ (accessed on 21 December 2023).
22. Yadav, V. Working with HBase. In *Processing Big Data with Azure HDInsight*; Apress: Berkeley, CA, USA, 2017; pp. 123–142.
23. Lee, J.-Y.; Kim, M.-H.; Raza Shah, S.A.; Ahn, S.-U.; Yoon, H.; Noh, S.-Y. Performance Evaluations of Distributed File Systems for Scientific Big Data in FUSE Environment. *Electronics* **2021**, *10*, 1471. [CrossRef]
24. Srivas, M.C.; Ravindra, P.; Saradhi, U.; Pande, A.; Sanapala, C.; Renu, L.; Kavacheri, S.; Hadke, A.; Vellanki, V. Map-Reduce Ready Distributed File System. U.S. Patent 20110313973A1, 22 December 2011.
25. Tanenbaum, A.S.; van Steen, M. *Distributed Systems: Principles and Paradigms*; Pearson Prentice Hall: Hoboken, NJ, USA, 2007; ISBN 978-0-13-613553-1.
26. Pate, S.; Van Den Bosch, F. *UNIX Filesystems: Evolution, Design and Impemenation*; John Wiley & Sons, Inc.: Hoboken, NJ, USA, 2003; ISBN 978-0-471-16483-8.
27. Abernethy, R. Programmer's Guide to Apache Thrift. 2019. Available online: https://learning.oreilly.com/library/view/programmers-guide-to/9781617296161/ (accessed on 21 December 2023).
28. Gabriel, E.; Fagg, G.E.; Bosilca, G.; Angskun, T.; Dongarra, J.J.; Squyres, J.M.; Sahay, V.; Kambadur, P.; Barrett, B.; Lumsdaine, A.; et al. Open MPI: Goals, Concept, and Design of a Next Generation MPI Implemen Tation. In Proceedings of the Recent Advances in Parallel Virtual Machine and Message Passing Interface: 11th European PVM/MPI Users' Group Meeting, Budapest, Hungary, 19–22 September 2004; Springer: Berlin/Heidelberg, Germany, 2004; pp. 97–104.
29. OpenMP Architecture Review Board. OpenMP Application Programming Interface Specification, Version 5.0. 2019. Available online: https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.0.pdf (accessed on 21 December 2023).
30. Meyers, S. *Effective Modern C++*; O'Reilly Media: Sebastopol, CA, USA, 2014; ISBN 978-1-4919-0399-5.

31. Bijlani, A.; Ramachandran, U. *Extension Framework for File Systems in User Space*; USENIX Association: Berkeley, CA, USA, 2019; pp. 121–134.
32. Vangoor, B.K.R.; Agarwal, P.; Mathew, M.; Ramachandran, A.; Sivaraman, S.; Tarasov, V.; Zadok, E. Performance and Resource Utilization of FUSE User-Space File Systems. *ACM Trans. Storage* **2019**, *15*, 15. [CrossRef]
33. Shan, H.; Shalf, J. *Using IOR to Analyze the I/O Performance for HPC Platforms*; Lawrence Berkeley National Laboratory: Berkeley, CA, USA, 2007.
34. Axboe, J. Fio-Flexible Io Tester. 2014. Available online: https://github.com/axboe/fio (accessed on 21 December 2023).