H2020-JTI-EuroHPC-2019-1

Project no. 956748

ADAPTIVE MULTI-TIER INTELLIGENT DATA MANAGER FOR EXASCALE

D2.1
Definition of requirements for ad hoc storage systems

Version 1.0

Date: September 29, 2021

Type: Deliverable
WP number: WP2

Editor: Marc-André Vef
Institution: JGU

Project co-funded by the European Union Horizon 2020 JTI-EuroHPC research and innovation programme and Spain, Germany, France, Italy, Poland, and Sweden		
Dissemination Level		
PU	Public	√
PP	Restricted to other programme participants (including the Commission Services)	
RE	Restricted to a group specified by the consortium (including the Commission Services)	
CO	Confidential, only for members of the consortium (including the Commission Services)	

Change Log

Rev.	Date	Who	Site	What
1	21/04/21	Jesus Carretero	UC3M	Document creation.
2	19/05/21	Ramon Nou	BSC	Adding chapters.
3	20/07/21	Anna Queralt	BSC	Adding content to 2.2.
4	07/09/21	Marc-André Vef	JGU	Modified structure
5	03/09/21	Marc-André Vef	JGU	Adding Introduction
6	14/09/21	Jesus Carretero	UC3M	Adding IMMS Description to 2.2
7	17/09/21	Felix Garcia	UC3M	Adding Expand Description to 2.2
8	15/09/21	Massimo Torquati	CINI	Software Heritage Anlytics Requirements
9	18/09/21	Marc-André Vef	JGU	Adding ad hoc storage system use cases
10	18/09/21	Ramon Nou	BSC	Terminology and conclusion
11	19/09/21	Marc-André Vef	JGU	Adding GekkoFS internals description
12	22/09/21	Marc-André Vef	JGU	Revising Chapter 3
13	22/09/21	Marc-André Vef	JGU	Restructuring document
14	22/09/21	Marc-André Vef	JGU	Adding ADMIRE data flow to Chapter 2
15	22/09/21	Marc-André Vef	JGU	Adding WP2 API and ADMIRE control flow to Chapter 3
16	24/09/21	Marc-André Vef	JGU	Revising Chapter 4
17	24/09/21	Marc-André Vef	JGU	Adding executive summary and conclusion
18	24/09/21	Marc-André Vef	JGU	Revising document; First draft finished
19	27/09/21	Adalberto Perez	KTH	Review of the Deliverable
20	28/09/21	Marc-André Vef	JGU	Addressing review comments and adding UML diagram
21	29/09/21	Marc-André Vef	JGU	Final version

Executive Summary

Ad hoc storage systems represent a dynamic component within ADMIRE to accelerate the I/O performance of scientific applications and significantly reduce I/O traffic to the *High-Performance Computing* (HPC) cluster's general-purpose storage backends, that is, shared parallel file systems (PFSs). Ad hoc storage systems are usually of an ephemeral nature and live within a particular application context, such as an HPC compute job, using (oftentimes) otherwise unused node-local storage devices, e.g., SSDs. Because ad hoc storage systems are often only accessed by a single application, they do not need to offer the same features as PFSs and, instead, can aggressively optimize towards the I/O needs of the application. Common examples for this are the alignment of an applications I/O size to the internally-used remote procedure call (RPC) bulk transfer sizes, or the relaxation of standard file system consistency guarantees to decrease inter-node communication and therefore increase I/O performance due to an higher amount of local I/O operations. Overall, this considerably reduces storage system complexity, resulting in huge amounts of I/O performance speed-ups (especially metadata) that linearly scale with the number of used compute nodes. Their performance is therefore mostly limited by the network speed and the node-local storage devices.

Nevertheless, existing ad hoc storage systems cannot react to different changing requirements of applications on their own. To optimize for the application's behavior during runtime and to use the HPC systems I/O resources efficiently, ad hoc storage systems need to be part of a more complex system, that is, ADMIRE, which not only controls them and their data movement, but also gives them commands on how to change their internal protocols and configurations to achieve these goals. Within the ADMIRE project, ad hoc storage systems are integrated into this system, controlled by the *I/O scheduler* taking malleable decisions (taken by the *malleability manager*) whenever they are seen as beneficial to the overall HPC system. This needs to involve intricate monitoring and analyzation mechanisms to understand how scientific applications work and how malleable decisions benefit them to improve their overall runtime and decrease their resource usage.

Thus, as part of T2.1, we explore in this deliverable (D2.1) the different ad hoc storage systems that are used in ADMIRE and discuss their features and their planned enhancements within ADMIRE. Moreover, we present in detail how the ad hoc storage systems are incorporated into the ADMIRE stack and what requirements must be fulfilled. This involves an in-depth description of an API on how to communicate with other ADMIRE components to achieve the above described goals. Finally, we introduce the scientific applications used in ADMIRE and their I/O requirements (which are detailed in D7.1) while briefly discussing how ad hoc storage systems can be used to improve the applications' runtimes.

Contents

1. Introduction

High-Performance Computing (HPC) applications have historically been computationally-bound, large-scale simulations with I/O access patterns that mostly performed sequential I/O operations on large files. As a result, the shared backend storage systems, that is, parallel file systems (PFS), e.g., Lustre or GPFS, were optimized for these access patterns. Yet, nowadays, HPC applications are often data-driven and generate, process, and analyze massive amounts of data, allowing many scientific fields to work on many unaddressable challenges of the past [18, 36]. These applications impose a different set of requirements than in the past on the overall system [31, 45] with workloads which include a huge number of small files, many metadata operations, random and non-contiguous I/O, and small I/O requests [8, 31]. Especially, access patterns that involve many small files and small I/O requests are on of the most challenging workloads for PFSs to handle efficiently [43]. What is more, the workloads of these data-driven applications can significantly disrupt other applications that are accessing the same PFS as well [10, 40]. As a result, all accessing applications suffer from extended latencies and drastically reduced bandwidths [41, 42].

For decades, magnetic disks have served as the storage backbone of HPC clusters, and their physical properties have significantly influenced the design of PFSs. Nowadays, the modern storage landscape offers significantly more options, filling the latency and bandwidths gaps between SSDs and DRAM. Although an increasing number of supercomputers also employ flash-based storage for handling metadata [46], faster storage devices, e.g., NVMe devices, are still too expensive for large-scale deployment. Many supercomputer sites have also started to equip their compute nodes with fast node-local SSD devices as burst-buffers with the goal to increase I/O performance of applications. The bandwidth of node-local SSDs typically exceeds the peak bandwidth of the attached parallel file system, while the maximum number of I/O operations (IOPS) can even be more than 10,000× higher than that of the parallel file system. Unfortunately, these devices are typically unused because applications depend on a much larger shared namespace whereas a single node only provides local storage.

One solution to increase the I/O performance of data-intensive applications and to reduce the load on the PFS caused by their workload's challenging access patterns, is to deploy *ad hoc storage systems* that use the fast node-local storage devices to form a shared namespace [5, 41]. Ad hoc storage systems are launched solely for a specific use case, workflow, or compute job and are destroyed afterwards. Therefore, input data is copied into the ad hoc storage system from the PFS and output data is copied back when the application finishes [29], henceforth called *stage-in* and *stage-out*, respectively. All intermediate data that is generated, such as temporary data or checkpoints, can then be placed on the ad hoc storage system and never reaches the PFS. However, ad hoc storage systems have not yet been fully integrated into the HPC storage stack in a dynamic and transparent fashion that makes them easy to use for the user and their applications. Further, because an ad hoc storage system can be launched within a job for a single application or for an application workflow, there is a tremendous opportunity to mold the ad hoc storage system to the needs of an application and even reacting to varying I/O requirements while the application is running – an optimization which is not possible with traditional PFSs that 1. are developed for the general use case to support all applications, and 2. are strictly POSIX-compliant making the highly distributed PFS behave like any other local file system. POSIX (Portable Operating System Interface) provides the standard semantics accepted by most application developers, but enforcing it in a distributed environment has shown to significantly reduce the PFS's peak performance [44], while many of POSIX's features are not required by most scientific applications [27].

To address these challenges, this WP will significantly extend several existing ad hoc storage systems to efficiently use fast node-internal storage technologies (e.g., NVMe and persistent memory), reducing the I/O pressure on the backend PFSs. By analyzing data-intensive applications (see WP7), we will define fast and

minimal storage system protocols (that may not necessarily follow the standard POSIX rules) to fully exploit today's and future storage devices. Further, we will develop extensive malleability options within the ad hoc storage systems to optimize their behavior to the requirements of the applications while being able dynamically extend or shrink the ad hoc storage systems based on the malleability decisions taken by the ADMIRE system. Finally, we expand the ad hoc storage system to support longer-running applications and workflows including error correcting codes.

2. Ad hoc storage systems

Parallel file systems such as Lustre [3, 34, 35] or GPFS [38] have already been serving as reliable backbones for HPC clusters for more than two decades. Meanwhile, a need for changes in the HPC storage architecture arose with the arrival of *data-intensive applications* in HPC. These applications shift the bottleneck from being computationally intensive, i.e., being restricted by the performance of the CPUs, to being bound by the quantity of data, its complexity, and how often it changes [24].

Node-local SSDs and burst buffers have been introduced into the HPC storage hierarchy to support the new application requirements. This hierarchy level can be used by ad hoc storage systems to share data and to provide better performance than that provided by general-purpose storage backends. Ad hoc storage systems utilize this temporary available storage within the HPC environment to form an ephemeral storage system that is created *ad hoc* for a particular use case or workflow. Ad hoc storage systems then form a global namespace by pooling all storage resources together. Because these kinds of storage systems are run for a limited use case, they do not need to provide all semantical guarantees or features that common PFSs offer. Instead, they can, in principle, only offer the features that a specific application requires and optimize for them to maximize I/O latencies and bandwidths. For example, if cache coherency is not needed, it can be disabled in the ad hoc storage system to improve write performance. It is therefore important to know application behavior to be able to decide on such optimization techniques.

Nevertheless, employing an ad hoc storage system does not come without cost when running them within a context of a compute job or workflow. This is because the ad hoc storage system provides a fresh environment, once launched, without any data. In other words, the *input* data needs to be copied from the PFS to the ad hoc storage system before the compute job can start, and the *output* data needs to be copied back from the ad hoc storage system to the PFS once the compute job finishes. These processes are known as *stage-in* and *stage-out*, respectively, and are handled by the I/O scheduler (WP4), among others.

Within the context of ADMIRE, we differentiate between ad hoc file systems (GekkoFS, IMSS, and Expand in ADMIRE) and ad hoc object stores (dataClay in ADMIRE) depending on the required semantics. We introduce each ad hoc storage system in the following sections, preceded by a discussion on the broad use cases of ad hoc storage systems. Lastly, we present the ADMIRE architecture and WP2's role in it.

2.1 Use cases

Ad hoc storage systems can be tailored for specific application semantics and can be applied, e.g., in the following use cases [5].

Big data workloads Data processing and analysis have always been important applications for smaller and mid-sized HPC clusters. Researchers of various fields have developed community-specific workflows and processing environments that often have been adapted to the specific properties of HPC backend storage systems [1, 13, 25, 47]. Using HPC backend storage as primary file systems, however, also restricted these big data applications to the drawbacks of centralized storage, for example that bandwidths and IOPS are shared between all concurrently running applications.

Ad hoc storage systems can help couple locality with a global namespace while additionally providing the random access rates of node-local SSDs. Nevertheless, in this case, the ad hoc storage systems must also support long-running campaigns, so that data staging between the backend storage and ad hoc storage system

can be reduced to a minimum. Specifically, the task to support long-running ad hoc storage systems is part of ADMIRE in T2.3 of WP2.

Bulk-synchronous applications Bulk-synchronous applications are the dominant workload seen on today's HPC systems. Here, applications run in a loosely synchronized fashion, generally synchronizing on major timestep boundaries. At these boundaries, the applications perform collective communication and I/O operations, for example, output or visualization dumps and checkpoint/restart. In the general case, the I/O operations per process are independent and written either to per-process files or to process-isolated offsets in a shared file. Additionally, read and write operations occur in bulk phases, without concurrent interleaving of reads and writes. In the context of ADMIRE, these kind of workloads are represented in the ADMIRE applications (see WP7 and Chapter 3.3).

These behaviors can easily be supported by ad hoc storage systems that can provide higher performance than general-purpose parallel file systems can. We know that the processes will not read and write concurrently and that each process will write to its own isolated offsets, the ad hoc storage system does not need to implement locking around write operations and hence can greatly improve performance.

Checkpoint/restart Checkpoint/restart is ubiquitous in scientific applications and, as such, also play a significant role in the ADMIRE applications. In general, HPC applications can survive failures by regularly saving their global state in checkpoints, which are often stored in the backend parallel file systems. The application can, in case of a failure, restart from the last checkpoint. Especially long-running applications benefit from the ability to restart failed simulation runs. However, the time to take a single application checkpoint increases linearly with the size of the application, and the overall checkpointing overhead increases with the checkpoint frequency. Older studies have shown that up to 65% of applications' runtimes were spent in performing checkpoints [14, 33], and studies indicate that up to 80% of HPC I/O traffic are induced by checkpoints [32].

The use of dedicated ad hoc checkpointing file systems, which can store the checkpoint either in main memory or on node-local SSDs are an interesting alternative to other techniques that employ dedicated checkpointing libraries [30], for example. Other solutions use compression or dedupication to reduce the checkpointing size [22, 23], which could even be coupled with ad hoc storage systems. If necessary, (some of the checkpoints) could be flushed to the backend PFS asynchronously. This allows first storing the checkpoint locally and later moving the data to the PFS that cannot be affected by a local failure [7].

Machine and deep learning workloads The desired input data sizes of machine and deep learning workloads are increasing rapidly. This is because of the use of small dataset sizes that can fail to produce adequately generalized models that can recognize real-world variations in input, such as poses, positions, and scales in images. The typical I/O workload for learning applications is that random samples from the full dataset are read repeatedly from the backend store during training. These random reads from a parallel file system can be a bottleneck, especially for learning frameworks being run on GPU clusters with very high computational throughput. For smaller datasets, the parallel file system cache, or perhaps node-local storage such as an SSD, can hold the entire dataset, and performance is not an issue. Larger datasets, however, may not fit in the file system cache or node-local storage, and the performance of the learning workload can suffer because of the I/O bottleneck [37, 48]. In the context of ADMIRE, such workloads are represented by the ADMIRE applications as well.

Learning workloads can benefit from ad hoc storage systems. They can distribute the very large datasets across the memory or storage on other compute nodes of a job and serve the randomly requested input to each process as needed. The learning workload can see vast improvements in I/O performance resulting in better training throughput. Further, the possibility to use a single global namespace of the ad hoc storage system guarantees that no learning biases are introduced as all nodes have access to the complete dataset. This is especially important as the deep learning community oftentimes partitions their datasets (e.g., with so-called *shards*, because they cannot fit into a single node. Even if a dataset was well partitioned, biases are not always trivial to detect and pose a significant challenge in the machine learning field – a challenge which can be circumvented by using a single global namespace as offered by ad hoc storage systems [37].

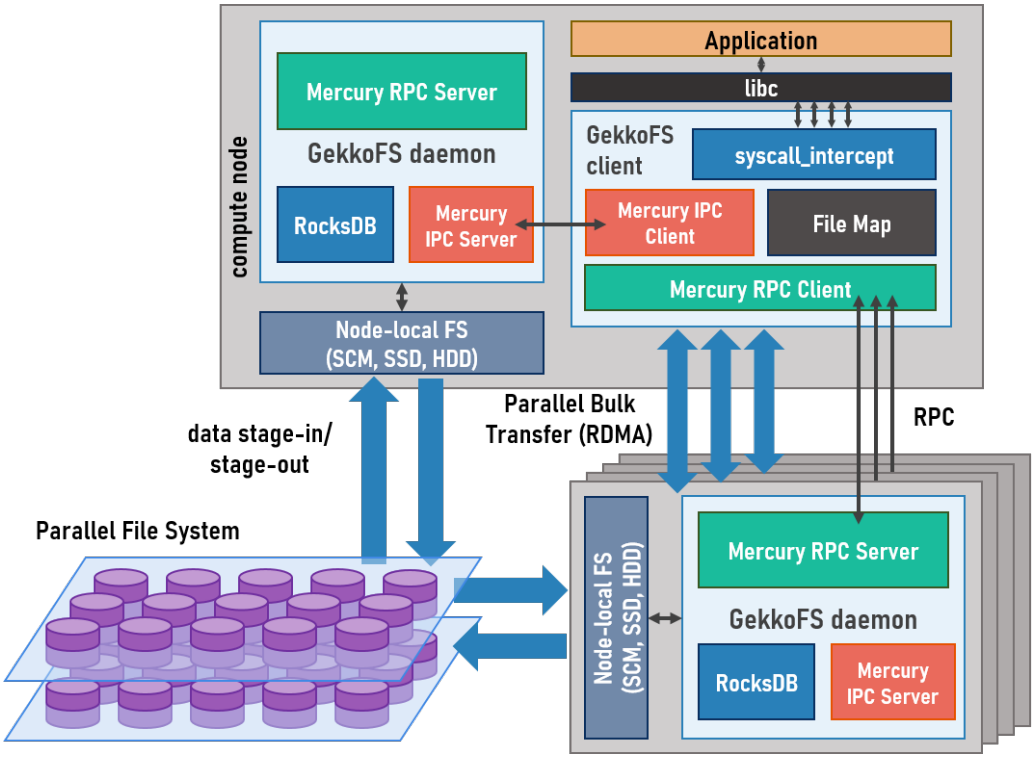


Figure 2.1: GekkoFS Architecture.

2.2 Ad hoc file systems

In ADMIRE, we will enhance and use GekkoFS, IMMS, and Expand to create ephemeral file systems using node-level resources or even external burst buffers if applicable. In the following sections, we will introduce each ad hoc file system in more detail concerning their architecture, their unique details, their enhancements within ADMIRE, and how they are used.

2.2.1 GekkoFS

GekkoFS is a highly scalable user-level distributed file system for HPC clusters. GekkoFS is capable of aggregating the local I/O capacity and performance of compute nodes to produce a high-performance storage space for applications. With GekkoFS, HPC applications and simulations can run in isolation from each other with regards to I/O, reducing interferences and improving performance. Furthermore, GekkoFS has been designed with configurability in mind, and allows users to fine-tune several of the default POSIX file system semantics, e.g., support for symbolic links, strict bookkeeping of file access timestamps and other metadata, or even modifying entire file system protocols. The overall design of GekkoFS takes previous studies on the behavior of HPC applications into account [27] to optimize the most used file system operations.

Contrary to general-purpose parallel file systems, a GekkoFS file system is ephemeral in nature. In other words, the lifetime of a GekkoFS file system instance is linked to the duration of the execution of its GekkoFS server processes, which are typically spawned when an HPC job starts and shut down when it ends. This means that users must copy any files that need to be persisted beyond the lifetime of the job from GekkoFS to a more permanent file system such as Lustre or GPFS. Moreover, because GekkoFS is implemented at user-level, the file system is only visible to applications using one of the GekkoFS client libraries. A consequence of this is that traditional file system tools (ls, cd, etc.) installed by system administrators will not be aware of files in a GekkoFS file system. To solve this, GekkoFS provides a client interception library that can be preloaded before calling these tools.

Architecture GekkoFS's architecture (see Figure 2.1) consists of two main components: a client library and a server process, called the *daemon*. GekkoFS runs entirely in user space and can therefore be used by any user without the need for administrative support. Because GekkoFS is running entirely in user-space, a GekkoFS file system cannot use the traditional mounting mechanism of kernel-based file systems. Instead, the file system uses user-defined directory path (e.g., `/mnt/gekkofs`) as a virtual mount point that is resolved during runtime. As a result, an application must use GekkoFS via the client interposition library through the `LD_PRELOAD` environment variable. The client library then intercepts all I/O calls and checks if it is within the virtual mount point. If this is not the case, the I/O calls is forwarded to the underlying operating system.

While a file is open, the client uses the file path within the virtual directory for each related file system operation and hashes it to determine the GekkoFS daemon node that should process it. The corresponding operation is then forwarded via an RPC message to the daemon where it is directly executed. By default, GekkoFS uses a pseudo-random distribution to spread data and metadata across all nodes (*wide-striping*). Note, however, that the data distribution can be chosen per application, and GekkoFS's modular approach makes it easy to add further data distributions, which is one of the goals of this work package. Because each client is able to independently resolve the responsible node for a file system operation, GekkoFS does not require central data structures that keep track of where metadata or data is located.

The daemons themselves run isolated (no communication between them) on each selected node, managing the node-local storage backend to store data and metadata. For metadata, each daemon operates a single local RocksDB key-value store which provides a high-performance embedded database for key-value data, based on a log-structured merge-tree (LSM) [9]. RocksDB is optimized for NAND storage technologies with low latencies and thus fits GekkoFS's needs as flash-based node-local storage is mainly used in today's HPC clusters. Data requests are split into equally sized chunks by the client before they are distributed across daemon nodes. During data transfers between client and GekkoFS daemons, the client exposes the relevant chunk memory region to the daemon which accesses it via *remote-direct-memory-access* (RDMA), if supported by the underlying network fabric protocol. The chunks are then persisted on the node-local storage device whereas each chunk corresponds to a single file on the local file system. Therefore, GekkoFS can use any node-local storage device as long as it is accessible by a path to the user.

For the remote procedure call (RPC) communication layer, we use on the *Mercury* RPC framework allowing GekkoFS to be network-independent. Mercury is an RPC communication library and focuses on HPC environments [39]. In contrast to other RPC frameworks, Mercury is able to use the native network transport layer and can, therefore, handle large data transfers efficiently. Mercury's *Network Abstraction Layer*, which provides a high-level interface on top of the lower level network fabrics, offers a wide variety of plugins to natively support common fabric protocols that are ubiquitous in HPC environments, e.g., InfiniBand or Omni-Path. This allows GekkoFS to efficiently transfer data within the file system.

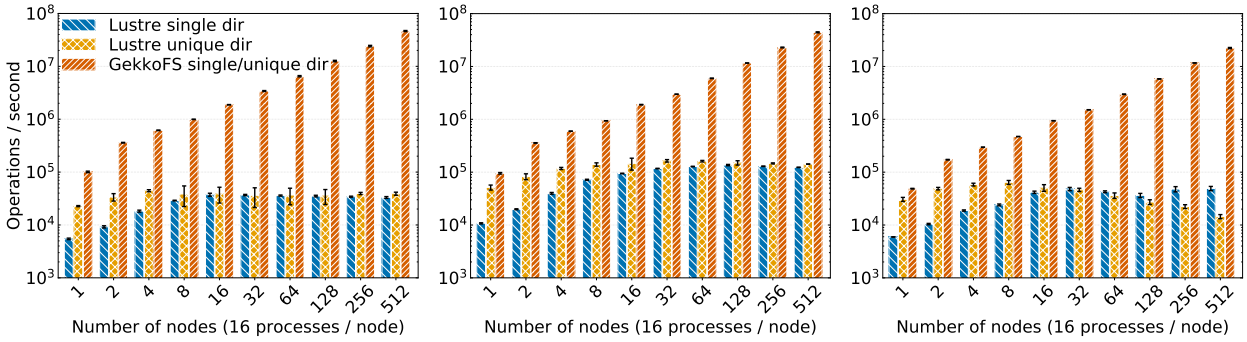
Overall, the highly decoupled architecture allows GekkoFS to scale linearly with the number of nodes while being highly configurable depending on the application's requirements. As a result, its performance is large dictated by the bandwidth and latency of the used network.

Performance We evaluated GekkoFS's performance with several benchmarks and real-life applications. For example, in the IO500 benchmark[1] which is suite of I/O benchmarks to compare facilities and storage systems driven by the storage community world-wide. During IO500, several metadata and data benchmarks, which have been shown to be difficult for file systems for handle, are performed via the *mdtest* and *ior* benchmarks. In the most recent edition, GekkoFS, with a limitation of 10 clients, reached the 7th position in the list using only 30 servers[2].

More specifically, GekkoFS is able to reach tens of millions metadata operations per second. On MOGON II at the Johannes Gutenberg University Mainz, we simulated common metadata intensive HPC workloads using the mdtest where it performs *create*, *stat*, and *remove* operations in parallel in a single directory. Concurrent metadata operations in a single directory are an important workload in many data-intensive HPC applications and are among the most difficult workloads for a general-purpose PFSs to handle efficiently [43]. Each operation was performed on GekkoFS using 100,000 zero-byte files per process with 16 processes used per node.

[1] https://www.vi4io.org/io500/list/20-11/10node

[2] The number of servers is not defined by IO500



(a) Create throughput (b) Stat throughput (c) Remove throughput

Figure 2.2: GekkoFS's file create, stat, and remove throughput for an increasing number of nodes compared to a Lustre parallel file system.

Figure 2.2 exemplary compares GekkoFS with Lustre in the three scenarios. GekkoFS's workload scaled with 100,000 files per process, while Lustre's workload was fixed to four million files for all experiments. This is because Lustre was detecting hanging nodes when scaling to too many files. Lustre experiments were run in two configurations: All processes operated in a single directory (`single dir`) or each process worked in its own directory (`unique dir`). Due to GekkoFS internally-kept flat namespace, both cases are handled indifferently. Compared to Lustre, GekkoFS achieved around 46 million creates per second (1,405x), 44 million stats per second (359x), and 22 million removes per second (453x) at 512 nodes.

We also conducted a number of data experiments which are available in detail in [41]. In summary, GekkoFS is able to efficiently use the available node-local storage devices, providing linear scaling using small and large I/O sizes for sequential and random access patterns in both shared file and file-per-process use cases.

We have done an initial approximation using GekkoFS to run a NEK5000 (ADMIRE application) workload. The results, with 16 nodes, generated an improvement of a 20% in the running time.

Enhancements inside ADMIRE GekkoFS will be enhanced in several dimensions to provide the requirements needed for each application inside the ADMIRE project.

1. **Malleability**: GekkoFS will provide an API to request the increase or decrease in the number of nodes available to the servers. This will produce a change in the number of IOPS, bandwidth, as well as space for the job. However, a change in the number of servers will produce extra data movements to redistribute the data correctly. Moreover, GekkoFS will offer a number of runtime malleability options that, for instance, can change the strictness of caching algorithms or can entirely change certain file system protocols in order to optimize not only for specific applications but also for their I/O phases that use varying access patterns. Lastly, GekkoFS should be able to minimize network and cpu usage and therefore minimize interference for the application that is running on the same node when it requires the node's resources [16].

2. **Resilience**: As part of T2.3, GekkoFS will provide resilience mechanisms. Resilience is the ability to recover from node failures [15]. While resilience is not as crucial for short-running ad hoc storage systems that are living within the context of a compute job, it becomes more important for longer-running workflows where ad hoc file system campaigns are running for several weeks or months. There is also the possibility to restrict resilience mechanisms to per-directory or per-file basis via I/O hints, reducing the overall managing overhead.

3. **New backends**: GekkoFS can use any file system to store its data as long as it is accessible to the user via a path. However, other interfaces, such as, persistent memory devices require other storage backends, e.g., based on the PMDK[3] library, to be able to fully exploit the capabilities of these fast

[3] https://pmem.io/pmdk/

storage devices. The Storage Performance Development Kit (SPDK) provides a set of tools and libraries for writing high performance, scalable, user-mode storage applications. The key techniques it uses to achieve high-performance involve avoiding syscalls by moving drivers into user-space or avoiding locks in the I/O path to improve overall latencies. We will investigate if SPDK is worthwhile for ADMIRE.

4. **Data distribution**: GekkoFS uses a hashing function to distribute data to the different nodes. This allows GekkoFS to avoid management nodes or inter-node locking mechanisms that have shown to be one of the main scalability culprits in general-purpose PFSs. However, wide-striping is not always the best distribution for an application. For example, they could prefer that the data is split in different ways so that access is always local. This could be done by enhancing the API, for instance, providing a mapping file to all the nodes specifying where to place files or directories. Nevertheless, this needs to be analyzed in detail as a single node, even with local access, may not overcome the bandwidth or IOPS that multiple parallel access to several servers can produce.

5. **System call intercept and file system interfaces**: Although system call intercept, which GekkoFS uses to intercept I/O calls, has the benefit to allow any application to access GekkoFS facilities without modification, it can cause certain challenges too. For example, different architectures (i.e. non-x86) are not compatible yet. For this reason, a library mode, so an application can directly call open/read/write functions in GekkoFS will be made available. Such library will also produce an increase in the performance as the syscall intercept introduces several jumps when a syscall is executed (although it is hidden by the increase in I/O performance). Because, such a library requires application modifications, we will also investigate other possibilities based on new innovations within the storage community, e.g., a file system client based on XFuse [20].

Example GekkoFS needs two steps to be used. First, we the servers before any client uses the ad hoc file system (although one improvement in ADMIRE will be the addition or removal of servers once the client is running, at least one server should be activated).

Next lines include an excerpt of a job description file (SLURM based) to launch the servers and the clients.

```
module load gekkofs

# Where do we put the data in the computation node
# Can be any directory (you can use /tmp as it is virtual)

export TMP_PATH=$TMPDIR
export GKFS_MNT="${HOME}/mntgkfs"

# Where we put the data files of GekkoFS
export GKFS_ROOT="${TMP_PATH}/agkfs_root"
# Sets a shared file to know which servers are available.
export GKFS_HOSTS_FILE=${HOME}/test/gkfs_hosts.txt
export LIBGKFS_HOSTS_FILE=${HOME}/test/gkfs_hosts.txt

rm $GKFS_HOSTS_FILE

CMD="${GKFS_DAEMON} --mountdir=${GKFS_MNT:?} --rootdir=${GKFS_ROOT:?}"

# We clean the system

srun \
    -n ${SLURM_JOB_NUM_NODES:?} \
    -N ${SLURM_JOB_NUM_NODES:?} \
    bash -c "rm -rf ${TMP_PATH} ; mkdir -p ${GKFS_MNT} ${GKFS_ROOT}"

echo "Starting GEKKOFS_DAEMON " $SLURM_JOB_NUM_NODES

srun \
    -N ${SLURM_JOB_NUM_NODES:?} \
```

```
    -n ${SLURM_JOB_NUM_NODES:?} \
    --cpus-per-task=1 \
    -m cyclic \
    --export="ALL" --oversubscribe \
    /bin/bash -c "echo Starting Daemon \${SLURMD_NODENAME}; ${CMD} " &
```

Then we can use it with the clients :

```
LD_PRELOAD=$GKFS_PRELOAD userapp
srun -N 4 -n 48 --oversubscribe --export="ALL",LD_PRELOAD=${GKFS_PRLD} \
    /bin/bash -c "parallel_userapp ${GKFS_MNT}/datadir"
```

It is important to note that these excerpts are presenting the status quo to use GekkoFS within an HPC job. In the context of ADMIRE, most parts of the require setup will be transparent to the user, including the staging of data between the PFS and the GekkoFS instance. Please refer to *D4.1 – Section 3.1 Slurm user interface* for more information.

2.2.2 Hercules IMSS (in-memory storage system

Nowadays, storage systems are one of the main bottlenecks in high performance systems and this challenge is expected to continue in next generation exascale systems. Current research works depict that emerging high-speed networks outperform physical disk performance, reducing the relevance of disk locality. Thus, one of the solutions proposed to this problem is to create in-memory storage systems by organizing the memory spaces in a complex hierarchy, moving data to local cache as fast as possible and throwing it to slower devices in an asynchronous way. This structure should be constructed with decoupling in mind, which consists on splitting and isolating compute nodes, storage nodes, and services (network, admin, etc.) as much as possible.

ARCOS has long experience creating and deploying in-memory ad hoc storage systems. AHPIOS was proposed as an ad hoc I/O solution for MPI applications [21]. Hercules is another solution proposed for HPC applications and workflows [11, 12]. Hercules has been enhanced in the ASPIDE project (https://www.aspide-project.eu/) to increase capacity and scalability. Below, we show a brief description of the IMSS system built.

Hercules IMSS has a set of well-defined objectives. Firstly, IMSS should provide flexibility in terms of deployment. To achieve this, the IMSS's API provides a set of deployment methods where the number of servers conforming the instance, as well as their locations, buffer sizes, and their coupled or decoupled nature, can be specified. IMSS follows a multi-threaded design architecture. Each server conforming an instance counts with a dispatcher thread and a pool of worker threads. The dispatcher thread distributes the incoming workload between the worker threads with the aim of balancing the workload in a multi-threaded scenario. Main entities conforming the architectural design are IMSS clients (front-end), IMSS server (back-end), and IMSS metadata server. Finally, the IMSS offers a set of distribution policies at dataset level to the application. As a result, the storage system will increase awareness in terms of data distribution at the client side, providing benefits such as data locality exploitation and load balancing.

Furthermore, to deal with the IMSS dynamic nature, a distributed metadata server entity was included in the design step. The metadata servers are in charge of storing the structures representing each IMSS and dataset instances. Consequently, clients are able to join an already created IMSS as well as accessing an existing dataset among other operations. The metadata server follows the chosen IMSS deployment strategy and will be exclusively accessed in metadata-related operations, such as: *create_dataset* or *open_imss*. Note that no metadata requests will be performed in data-oriented operations, such as *get_data* or *set_data*, reducing both the overhead of the I/O accesses and the risk of contention at the metadata server.

IMSS provides multiple data distribution policies by design (local, buckets, hash, round-robin, and crc) in both deployment modes, increasing the number of data scattering possibilities among storage servers and enlarging versatility in terms of application's data management. Within the previous possibilities, a *LOCAL* policy should be highlighted as it will have the objective of exploiting data locality as much as possible: data requests will be forwarded to the storage server running in the same machine where the request was made.

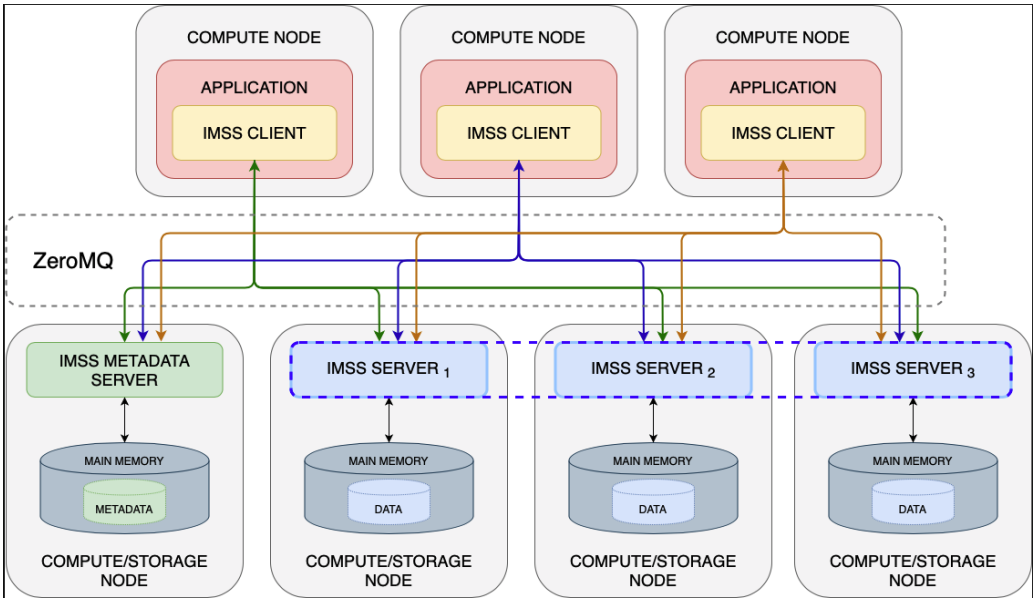


Figure 2.3: Hercules IMSS Architecture.

Finally, a non-POSIX *get-set* interface will be provided in order to manage *datasets*, which conform a storage abstraction used by IMSS instances in order to manage data blocks (smallest data unit considered within the storage system).

Deployment Strategies Hercules IMSS provides two deployment strategies to adapt the storage system to the application's requirements.

- *Application-detached* strategy, consisting on deploying Hercules IMSS clients and servers as process entities on decoupled nodes. IMSS clients will be deployed in the same computing nodes as the application, using them to take advantage of all available computing resources within an HPC cluster, while IMSS servers will be in charge of storing the application datasets and enabling the storage's execution in application's offshore nodes. In this strategy, IMSS clients do not store data locally, as this deployment was thought to provide an application-detached possibility. In this way, persistent IMSS storage servers could be created by the system and would be executed longer than a specific application, so as to avoid additional storage initialization overheads in execution time.

- *Application-attached* strategy seeks empowering locality exploitation constraining deployment possibilities to the set of nodes where the application is running, so that each application node will also include an IMSS client and an IMSS server, deployed as a thread within the application. Consequently, data could be forced to be sent and retrieved from the same node, thus maximizing locality possibilities for the data. In this approach each process conforming the application will invoke a method initializing certain in-memory store resources easing future deployments. Flexibility aside, as main memory will be used as storage device, an in-memory store will be implemented so as to achieve faster data-related request management.

Hercules IMSS layers Hercules IMSS has two major architectural layers, as shown below. The architecture of Hercules IMSS is depicted in Figure 2.3.

- *Client*. Client applications handle IMSS and dataset instances through an IMSS client library. The API provides a set of operations to *create*, *join*, *get*, *set*, and *release* data, datasets, and IMSS instances. Along any session, clients create and join multiple *IMSS instances*. An IMSS instance is defined as an ephemeral dedicated storage entity conformed by multiple servers distributed along a set of user-defined machines that use main memory in order to store datasets. An IMSS instance is identified by a unique

Uniform Resource Identifier (URI) and it is represented by a data structure storing parameters such as the number of servers conforming the instance and their respective location. Moreover, a dataset entity corresponds to a collection of data elements with a constant size that are distributed among the storage servers of a single IMSS instance following a certain data distribution policy. As IMSS instances, datasets are identified by a unique URI, which reflects the storing IMSS entity. A data structure representing the dataset abstraction is created per instance, gathering parameters such as the distribution policy assigned to the dataset, the number of data elements conforming the dataset, and the replication factor, among others.

- *Back-end Layer*. Each IMSS instance is conformed by multiple IMSS storage servers. Each one stores multiple data blocks of different datasets. Each IMSS server deploys a *dispatcher* thread that distributes and balances client connection requests among worker threads following a round-robin policy. In addition, worker threads belonging to the same server associate data blocks' identifiers to memory locations in a map-based memory container. In order to handle *get* and *set* requests, each worker thread exclusively accesses the map container for the provided data block location. Afterwards, the requested data block is wrapped into a message and is sent back to the client in case of a *get* operation. If the requested data block is not found, an error code is returned. If the operation is a *set*, the worker thread overwrites the concerned block if it was already stored. Otherwise, the data block is written and a new key-value pair representing the previous block is added to the map.

Data Distribution Policies Dataset distribution policies included in Hercules IMSS define the distribution of each dataset in the instance deployed. The policy determines the back-end server in charge of storing data blocks. The IMSS front-end layer handles the policy assignment whenever a dataset is created. The IMSS metadata server maintains the dataset's data structure, annotating the distribution policies of each one. The following policies have been developed: ROUND_ROBIN, BUCKETS, HASHED, CRC16bits, and LOCAL.

With those policies, IMSS enables the possibility to tune the dataset distribution. These distribution policies aim to increase performance. The LOCAL policy experimentally obtains the greatest performance due to the exploitation of locality. In the current prototype, the distribution policy is established at creation time and it can not be modified. In the future, we plan implement a dynamic distribution policy that enables to adapt the behaviour in terms of system metrics (CPU, memory consumption, etc).

Communications In order to optimize both intra and inter process communication, ZeroMQ library [19] was opted, handling communications between the different entities conforming an IMSS instance. ZeroMQ has been qualified as one of the most efficient libraries for creating distributed applications [26]. ZeroMQ provides multiple communication patterns across various transport layers, such as inter-threaded, inter-process, TCP, UDP, and multicast. ZeroMQ conforms a performance-friendly API with an asynchronous I/O model that promotes scalability. In addition, ZeroMQ library provides support for zero-copy messages, avoiding further overheads due to data displacements.

However, ZeroMQ must be used with datagrams to provide performance and the possibility of message losses increases with the traffic, increasing the complexity of communication managament to avoid those errors. An RPC interface is most suitable for programmers. Thus, we will consider to use RPCs for Hercules in ADMIRE project.

Hercules enhancements Hercules IMSS will be enhanced in several dimensions inside the ADMIRE project:

1. Malleability. Currently, Hercules has functions in the API to increase or decrease the number of servers and the number of clients. While extending the system is not a problem, reducing the number of servers is challenging as we have to copy data to other nodes.

2. Persistence. Hercules is a pure in-memory system. We plan to use it as frontend layer for other backend file systems and thus providing persistence of data. Data will be most probably stored in large chunks resulting of flushing the in-memory stores to the backends. The goal is to minimize data movements.

3. Data distribution. Even if Hercules support several policies for data distribution, we could include more or better shaping for some of them, depending on the profile of the applications on top. The idea is to have an adaptive data distribution policy even per dataset.

4. Communications. Currently Hercules uses ZeroMQ, but we plan to change the communications to Mercury RPCs, that are currently well in use in the HPC world. This way, the communication mechanism will be the same that in GekkoFS and we will get the good performance and reliability proved by Mercury in HPC environments.

2.2.3 Expand

Expand is a parallel file system based on standard servers. Expand provides a user-space library for multiple file system interfaces such as POSIX, MPI-IO, and Java applications [2, 6, 17]. The first prototype developed was based on NFS servers and a later version was extended to other servers back-ends such as HTTP, GridFTP, and ad hoc file systems developed by UC3M. The main advantage of Expand over other parallel file systems is its ease to install and use.

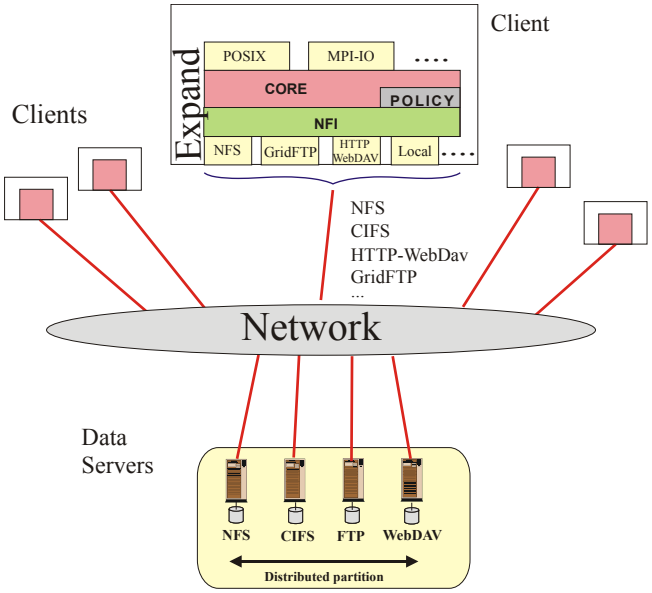


Figure 2.4: Architecture of *Expand*.

The Expand library is responsible for contacting the servers using the protocols associated with them. To this purpose, the client is designed as shown in Figure 2.4 and it is divided into four layers. The top layer is responsible for providing various access interfaces to parallel applications. The layer named *core* is responsible for defining the basic algorithms used for data access. Inside this layer the module *policy* is located, that enables the possibility to define what are the actions to take in the operations of metadata location, selection of servers involved in every operation, etc. Both *core* and *policy* layers use the services of *NFI* layer, which facilitates for Network File Interface. This layer provides an interface to the basic operations of a file system. The lower layer is responsible for implementing the interface provided in the *NFI* layer for the different access protocols to a file system.

Data is scattered by Expand through all servers using blocks as distribution units. The processes of a parallel application are clients that use the Expand library to access a distributed partition.

Using the outlined approach for the design of Expand, we have the following advantages:

- No changes are required on the servers (NFS, WebDAV, FTP, etc.) to benefit from Expand. All aspects of the operations of Expand are deployed in the client space.

- Expand is independent of the operating system used on clients. All operations are implemented using standard protocols.

- It enables the use of servers running on different architectures and operating systems, since the use of standard protocols hides these differences.
- Building a file system is significantly simplified because all operations are implemented in the front-end side. This approach is completely different to that used in many of the current parallel file systems.
- The configuration of the system is very simple as standard servers, like NFS, are very familiar to users. The server only needs to export the appropriate directories, and clients only need a small configuration file detailing the information of the distributed partition.

Expand can be used as ad hoc parallel file system and can be integrated with different server technologies by implementing the appropriate file system driver.

The parallel file systems most commonly used in both HCP and Big Data environments are characterized by providing ad hoc servers that need to be deployed in the clusters. This increases the effort that IT staff have to make for their installation. Expand provides the following features that facilitates its deployment:

- Expand partitions are defined using a small configuration file in XML format. Applications can deploy partitions as needed, thus fitting data distribution, locality, and bandwidth.
- Files are distributed among all servers used in the distributed partition.
- Provides different data distributions models for each file: cyclic distribution, random distribution, and user-driven distribution.
- Expand does not have any metadata server, all metadata management are managed independently by the servers and directly by the expand library. This feature reduces possible bottlenecks in the system.
- Dynamic reconfiguration of partitions. Expand allows to dynamically reconfigure a partition by adding new server nodes to it. Therefore, partitions size can increase and new resources can be included dynamically.
- Access control and security depend on the individual protocol used.

The main interface provided by Expand is POSIX, which provides direct access to data stored in Expand partitions. This interface is provided through a user level library. Parallel applications can also use Expand with MPI-IO. Expand has been integrated in ROMIO and can be used with MPICH.

Expand partitions are defined using a small configuration file in XML format [4] with the following structure:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xpn_conf>
 <partition name="<partition1>" type="<options>" bsize="<block_size>">
     <data_node id="id1" url="<protocol>://<server>/path/"/>
     ...
 </partition>
  ...
 <partition name="<partitionN>" type="<options>" bsize="<block_size>">
     <data_node id="id1" url="<protocol>://<server>/path/"/>
     ...
 </partition>
</xpn_conf>
```

Where the label *partition* enables the definition of the structure of a storage partition. For each partition, we define each one of the servers used to form the partition. The servers are defined by a URL which is unique within a partition, not being able to use the same URL more than once per partition.

2.3 Ad hoc object store

In ADMIRE, we will enhance and use dataClay to create an object store using node-level storage resources.

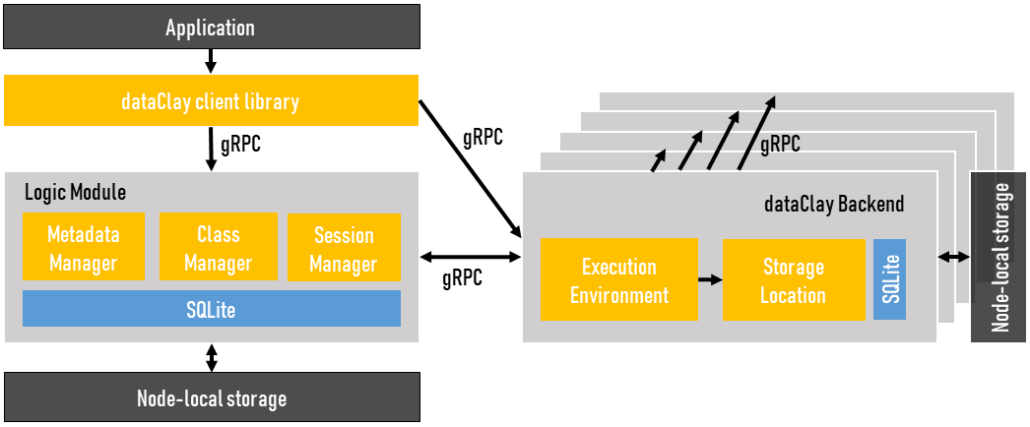


Figure 2.5: dataClay Architecture.

2.3.1 dataClay

DataClay [28] is a distributed object storeare. It is designed to hide distribution details while taking advantage of the underlying infrastructure, be it an HPC cluster or a highly distributed environment such as edge-to-cloud.

Objects in dataClay are enriched with semantics, including the possibility to attach arbitrary user code to them. In this way, dataClay enables applications to store and access objects in the same format they have in memory, also allowing them to execute object methods within the store to exploit data locality. In this way, only the results of the computation are transferred to the application instead of the whole object.

DataClay is implemented at user-level, so it is visible to applications using its client library. It can be deployed in two different ways: as a service or as an ephemeral storage system. In the first case, it is deployed as a long-lived service in a dedicated set of nodes, and objects can be shared by multiple jobs or applications. In the second case, dataClay runs in the compute nodes within a job, and the data it contains must be persisted after the job finishes if needed. In both cases, the active capabilities (in-store method execution) of dataClay minimize data transfers and copies, either between nodes when the application and dataClay run independently or between the application and the dataClay processes when both of them live in the same node. Additionally, regardless of the kind of deployment, disk accesses are avoided as much as possible via an object cache, where objects are already instantiated and ready to serve execution requests.

DataClay has been designed with flexibility in mind, with the aim to provide each application only with the required functionality, not paying extra overheads for any functionalities that the application does not need. Thus, dataClay has no embedded replication, consistency, or fault tolerance policies, but these can be customized according to the application requirements. This feature makes dataClay very suitable as an ad hoc storage system, as not only can it temporarily manage the application data, but it can also do it in the most appropriate way for that application.

Additionally, the abstraction layer it provides on top of the infrastructure allows using different storage solutions as backend, while applications keep managing objects only. This has the potential to incorporate new devices, such as non-volatile memories, and make the most out of them transparently to the application, hiding the details of the interaction with these devices.

Architecture The architecture of dataClay is depicted in Figure 2.5. The main components in dataClay are the *Logic Module* and the *Backends*.

The Logic Module is the central authority for metadata. It holds information about the location of the objects and their identifiers (*Metadata Manager*), which are internal, unique and automatically generated by the system. Additionally, objects can be identified by means of a user-defined name or alias known by the applications. The Logic Module also manages semantic information about objects, that is, their structure and associated user code (*Class Manager*). This information is communicated to dataClay during a registration process, previous to application execution, where the Logic Module ships the code to the Backends. The Logic Module is also in charge of managing the user sessions (*Session Manager*).

Each backend is composed of a *Storage Location*, where objects are stored, and an *Execution Environment*, which handles execution requests on the subset of objects managed by the backend. Objects are cached in their native representation in the Execution Environment, so that execution requests can be served without contacting the Storage Location. Both components can live in the same or in independent nodes, and a Storage Location can have one or more associated Execution Environments.

In order to avoid accessing the Logic Module to locate objects during execution, each backend caches metadata for the objects it creates or accesses so that it can directly communicate with other backends storing the objects that are required for an execution. When the information is not cached or invalid because the object has been moved, the Logic Module is contacted to get the updated information. Also, the metadata of a new object is sent to the Logic Module from the Backends when the Execution Environment sends the object to the Storage Location to free memory.

Finally, the client library communicates with dataClay on behalf of the application, issuing creation, deletion, or execution requests directly to the backends. Some requests are sent to the Logic Module from the client library, namely those regarding class or session management and those involving user-defined aliases.

Enhancements inside ADMIRE Although dataClay can effectively be deployed within a job and be used to manage data only within the lifetime of an application, it is not optimized for being used as an ad hoc storage system. Thus, some aspects of the architecture should be re-designed, and also some functionalities should be added, in order to support the requirements in ADMIRE:

1. **Stage-in and stage-out:** The in-storage processing capabilities of dataClay can be exploited during stage-in/out phases, as they enable a pre/post-processing of the data. This can be useful to adapt the data to the format of the ad hoc or backend storage system. However, as dataClay was not initially designed as an ad hoc object store, some of its functionalities or architectural decisions must be revised. For instance, the way in which metadata is currently managed in dataClay is not optimal in an ad hoc context, as it optimizes performance in execution but penalizes ingestion, which may result in losing the performance benefits provided by the ad hoc storage system when taking into account the whole process. Thus, metadata should be distributed across all backends, not only cached, to accelerate ingestion and minimize communications during execution and stage-in/out phases.

2. **Malleability and data distribution:** dataClay does not currently provide any mechanisms to dynamically increase or decrease the number of nodes according to the application needs. The distribution of metadata required for efficient stage-in/out will also enable the efficient implementation of new malleability mechanisms. Depending on the application, increasing/decreasing the number of nodes may or may not require a redistribution of the data, so the proposed malleability mechanisms should come with the possibility to choose an appropriate redistribution policy (for instance, to favor load balancing or to favor data locality by placing together those objects that are accessed together). The active features of dataClay will enable it to adapt its behavior to the particularities of each application.

3. **Support new kinds of storage devices:** The abstraction layer that dataClay adds on top of storage allows to manage different kinds of storage devices transparently to the application. This gives us the possibility to incorporate support to new kinds of devices, such as NVMe or NVM, to take advantage of their characteristics. For instance, the byte-addressability of NVM can further increase data locality, not only between the object store and the application but also within the object store itself, eliminating the need to copy data to main memory for execution. This optimization should provide a performance increase in those applications that fit the I/O characteristics of these devices.

Example dataClay must be orchestrated before being used. This task is performed with the help of the `dataclaysrv` script which automatizes this process. An example of this process can be seen in the following script:

```
1 module load DATACLAY
2
```

```
BACKENDS_PER_NODE=4
HOSTS="node01 node02 node03"
dataclaysrv start --hosts "$HOSTS" \
--pyclay-path $PYCLAY_PATH \
--javaclay-path $DATACLAY_JAR --python-ee-per-sl $BACKENDS_PER_NODE
```

The DATACLAY module loads a default value for most variables shown in the script (e.g. $PYCLAY_PATH) which are adequate for most execution scenarios.

Once dataClay has been deployed, the application can be started. Then, the client library will be able to connect to the dataClay instance to operate on objects on behalf of the application.

2.4 ADMIRE architecture

In this section, we will present the overall ADMIRE architecture and its information flow between its components. Then we will discuss WP2's role within this architecture and how it is connected with the other components.

2.4.1 ADMIRE information flow

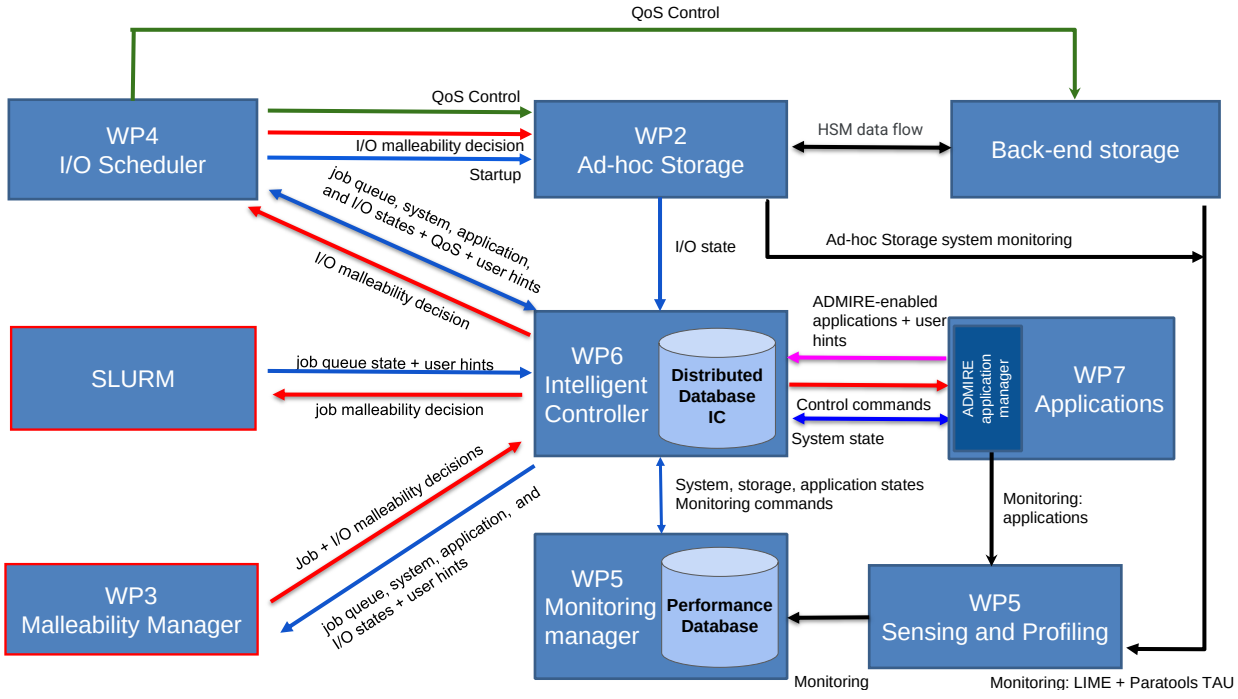


Figure 2.6: ADMIRE architecture overview. Each component developed in the project's scope has included the label of its related Work Package (WP).

Figure 2.6 illustrates an overview of ADMIRE architecture and shows how the information is exchanged between the components.

1. **Storage systems:** The storage subsystem is represented on the upper part of the figure and consists of the ad hoc and backend storage systems. The former is designed in ADMIRE's WP2 and is responsible of providing to each application an ad hoc high-performance storage system tailored to the application's characteristics. The latter one (backend storage) represents the main filesystem used by the HPC platform (Lustre for example).

2. **I/O scheduler:** Both storage systems are coordinated by the I/O scheduler (shown in upper-left corner of the figure) which is the responsible of the deployment and configuration of the ad hoc storage, the specification of Quality-of-Service metrics and the implementation of I/O scheduling policies.

3. **ADMIRE applications:** The application that are being executed in the platform are shown in the right-central part of Figure connected to the intelligent controller.

4. **Monitoring:** Both applications and storage systems are monitored by the Sensing and Profiling component (lower-right corner of the figure), developed in WP5. This component is responsible of collecting system-wide performance metrics at node level that will be stored in an internal performance database. The Monitoring Manager (lower-central part of the figure) will manage this database and will generate performance models related to each existing application, storage system (both ad hoc and backend) and compute nodes.

5. **Malleability:** The malleability manager (lower-left corner of the figure) is responsible of determining the malleable actions related to each running application and ad hoc storage system. These decisions include the increase/decrease of the number of processes or threads of a certain application or to deploy/remove the number of instances of a given ad hoc storage.

6. **Intelligent controller:** The intelligent controller (central part of the figure) has different roles. The first one is to collect the current system status using the information collected from Slurm, the Monitoring Manager, the storage systems and the applications. This will include combined information about the hardware status, the existing running applications and the storage. This information will be kept in an internal distributed database. The second role of the intelligent controller is to generate performance models of these components and use them to predict potential performance bottlenecks in the system. These models will be also provided to the malleability manager and I/O scheduler in order to support the decision-making of both of them. Other main role of the intelligent controller is to coordinate the actions taken by other ADMIRE's components. Examples of these actions are (1) to activate/deactivate each application monitoring and (2) to send the malleable decisions taken by the malleability manager to the I/O scheduler or the applications, in case of being I/O-related or application-related decisions, respectively.

2.4.2 WP2 in ADMIRE

In the context of ADMIRE, WP2 offers several ad hoc storage systems to be used by HPC applications. In contrast to the other ADMIRE components, an ad hoc storage system is no single or permanent component within the ADMIRE stack. Instead, the number of ad hoc storage systems and on how many nodes they run is highly dynamic. As they are ephemeral in nature, their number and used system resource allocations is changing over time depending on the cluster requirements. Moreover, the ad hoc storage system instances are controlled by the I/O scheduler. Therefore, how an ad hoc storage system behaves is decided by the ADMIRE system and by the user via hints through the Slurm API (See *D4.1 – Section 3.1 Slurm user interface* for details).

The following paragraphs discuss the connections of the ad hoc storage systems to the ADMIRE components in-depths.

I/O scheduler The I/O scheduler represents the primary connection between ADMIRE and the ad hoc storage system and manages them. It has the authority to 1. start and stop an ad hoc storage system on a set number of nodes, 2. move (stage) data between the backend storage and the ad hoc storage system, 3. forward malleability requests to them, and 4. control QoS mechanisms that influence their resource (CPU, network, or I/O) usage.

Intelligent controller Whenever a change occurrs within an ad hoc storage system, such as a malleability request that changes internal configurations, the ad hoc storage system reports them to the intelligent controller which also keeps track of their configurations.

Monitoring (sensing and profiling) One of WP2's task is to extend the ad hoc storage systems with a monitoring subsystem so that they are able to report various performance counters, e.g., used network bandwidth or the used storage capacity, to the ADMIRE stack allowing to learn about an applications I/O footprint during runtime. As such, the WP2 partners agreed to extend the ad hoc storage systems with a Prometheus[4] monitoring subsystem that is able to report performance-related information to the Prometheus framework, processed in WP5.

Backend storage The direct interaction between the backend storage and an ad hoc file system is a special case in which WP2 aims to investigate the benefits of incorporating an ad hoc file system, such as GekkoFS, directly into Lustre's *Hierarchical Storage Management* (HSM) mechanisms. As a result, the GekkoFS namespace transparently becomes part of Lustre's global namespace as a distributed Lustre client cache, while using Lustre's QoS mechanisms [35] to steer I/O traffic. This interaction will be build on previous works authored by both ADMIRE partners JGU and DDN [34].

[4] https://prometheus.io/

3. The ad hoc storage system API

This chapter introduces and discusses the ad hoc storage system API. The API is split into two parts: the startup API and the runtime API. The former refers to a *command-line interface* (CLI) API that the I/O scheduler interacts with when starting an ad hoc storage system on a set of compute nodes. The runtime API refers to an RPC interface that is called by the I/O scheduler to send the ad hoc storage system malleability and Quality-of-Service (QoS) requests.

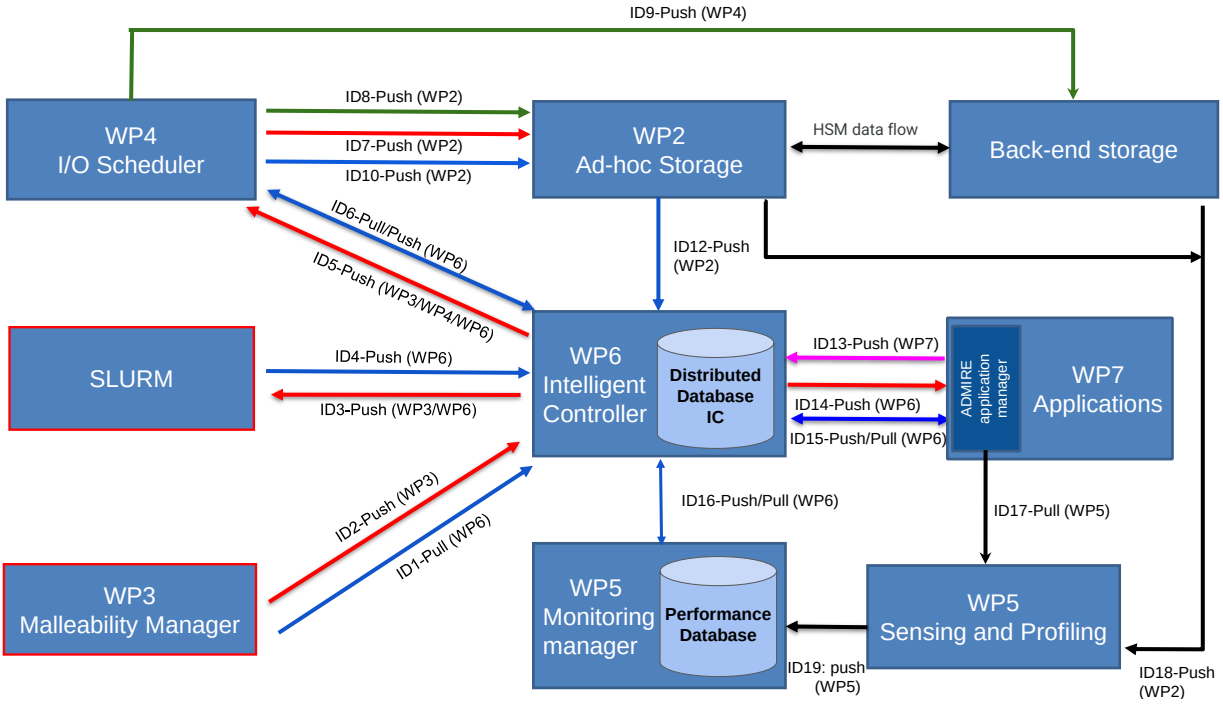


Figure 3.1: ADMIRE controlflow between the components.

3.1 Startup API

The following functions are the minimum required set of arguments to work with the ADMIRE system. They are implemented by all ad hoc storage systems. The main goal is to be able to 1. define the backend storage they use for storage system data, 2. specify how users can access the storage system, that is, the mountdir path,

3. set the number of nodes each storage system instance is run on, and 4. configure each storage system.

Algorithm 1: Definition for the ad hoc storage system's `data_dir` interface

Name: *data_dir*
ID: ID10
Type: Push
Input: The *path* for the ad hoc storage system's used storage backend
Description:
Specify the path where the ad hoc storage system should place its data, e.g., the node-local SSD.
Example: *gekkofs_daemon* `--data_dir=/mnt/ssd`

Algorithm 2: Definition for the ad hoc storage system's `mount_dir` interface

Name: *mount_dir*
ID: ID10
Type: Push
Input: The *path* where the ad hoc storage system is mounted
Description:
Specify the path at which users access the ad hoc storage system namespace.
Example: *gekkofs_daemon* `--mount_dir=/mnt/gekkofs`

Algorithm 3: Definition for the ad hoc storage system's `hostfile` interface

Name: *hostfile*
ID: ID10
Type: Push
Input: The *hostfile* containing the node for the ad hoc storage system
Description:
Specify the hostfile which contains a number of nodes (one node per line) that should run a ad hoc storage system server.
Example: *gekkofs_daemon* `--hostfile=/tmp/job_123456/adhoc_hostfile`

Algorithm 4: Definition for the ad hoc storage system's `config` interface

Name: *config*
ID: ID10
Type: Push
Input: The path to a *config* file containing ad hoc storage system configurations
Description:
Specify the path to a config file which contains specific initial configurations when the ad hoc storage system is first started. The configurations are defined in JSON format and specified in the coming months.
Example: *gekkofs_daemon* `--config=/tmp/job_123456/adhoc_hostfile`

3.2 Runtime API

The runtime API to be implemented by the ad hoc storage systems to be able to contact them during operation and to modify their behavior according to the application's needs. Nevertheless, they also report configuration changes within the file system (such as, a change of used nodes) to the intelligent controller, and they report performance counters and other collected events to ADMIRE's monitoring component.

3.2.1 Interaction with the I/O scheduler

The following functions represent the ad hoc storage system API offered to the I/O scheduler.

Algorithm 5: Definition of the `ADM_change_node_number` runtime interface

Name: *ADM_change_node_number*
ID: ID7
Type: Push + Async
Input: A path to *hostfile* defining the new ad hoc storage system configuration
Output: A *err* code determining whether the operation was successful
Description:
Specify via a hostfile (containing one node per line) which nodes are part of the new storage system configuration. Depending on the number of nodes, the ad hoc storage system will be extended or shrunk. Depending on the data distribution of the storage system, data will need to be redistributed.

Algorithm 6: Definition of the `ADM_malleability` runtime interface

Name: *ADM_malleability*
ID: ID7
Type: Push
Input: A path to *description_file* containing a number of malleability commands
Output: A *err* code determining whether the operation was successful
Description:
Specify via a description_file containing one or more malleability commands that need to be executed on the ad hoc storage system. Potential malleability options could be the change of: file system block sizes, the data distribution, the storage backend, file system cache consistency, or file system cache strictness, among others. We aim to identify more malleability options that have beneficial effects to an application once we continuously analyze the ADMIRE applications and other data-driven applications that run in HPC environments.

Algorithm 7: Definition of the `ADM_flush_{file,object}` runtime interface

Name: *ADM_flush_{file,object}*
ID: ID7
Type: Push
Input: A path to a *file* or *object* in the ad hoc storage system that should be flushed in the background
Input: A path to a *file* in the backend PFS which should receive the file
Input: The maximum allowed *transfer_limit*
Output: A *err* code determining whether the operation was successful
Description:
Specify a specific file or object in the ad hoc storage system that should be flushed out to the PFS in the background. To reduce potential interference a transfer limit can be given that applies to the maximum read and write speed of the file. Use cases for this function include the flushing of intermediate results or checkpoint files.

Algorithm 8: Definition of the `ADM_flush_{directory,container}` runtime interface

Name: *ADM_flush_{directory,container}*
ID: ID7
Type: Push
Input: A path to a *directory* or *container* in the ad hoc storage system whose contents should be flushed in the background
Input: A path to a *directory* in the backend PFS which should receive the contents
Input: The maximum allowed *transfer_limit*
Input: If *parallel* execution is allowed
Output: A *err* code determining whether the operation was successful
Description:
Specify a specific directory or container in the ad hoc storage system that should be flushed out to the PFS in the background. To reduce potential interference a transfer limit can be given that applies to the maximum read and write speed of the file. Further, the parallel flag can be enabled allowing multiple files or objects to be flushed out in parallel. Note, however, that the transfer_limit must still be honored. Use cases for this function include the flushing of intermediate results or several checkpoint files within a directory.

Algorithm 9: Definition of the `ADM_network_bandwidth_limit` runtime interface

Name: *ADM_network_bandwidth_limit*
ID: ID8
Type: Push
Input: The maximum allowed network throughput in *mbps* per ad hoc storage server process.
Input: The maximum allowed number of *rpc* per second per ad hoc storage server process.
Output: A *err* code determining whether the operation was successful
Description:
A QoS option that limits the network usage by specifying the maximum allowed number of bandwidth or RPCs per second for each ad hoc storage system server process. This setting allows to control the network usage of the ad hoc storage system to minimize interference with the network traffic of the same application if it is running on the same node.

Algorithm 10: Definition of the `ADM_io_limit` runtime interface

Name: *ADM_io_limit*
ID: ID8
Type: Push
Input: The maximum allowed I/O bandiwdth to the node-local storage in *mbps* per ad hoc storage server process.
Output: A *err* code determining whether the operation was successful
Description:
A QoS option that limits the node-local storage device usage by specifying the maximum allowed I/O bandwidth for each ad hoc storage system server process. This setting allows to control the usage of the node-local storage device if, for instance, an application also relies on the node-local storage device in certain I/O phases. Note that this setting implicitly effects the used network bandwidth as well.

See Deliverable D4.1 for I/O scheduler information.

3.2.2 Interaction with the intelligent controller

The following function represent the ad hoc storage system API to send its state to the intelligent controller.

Algorithm 11: Definition of the `ADM_adhoc_config_report` runtime interface

Name: *ADM_adhoc_config_report*
ID: ID12
Type: Push
Input: An ad hoc storage system *configuration* string in JSON format
Output: A *err* code determining whether the operation was successful
Description:
When a malleability request is received by the ad hoc storage system, it may require some time to fully apply these changes (e.g., if the number of nodes change and data need to be redistributed). The ad hoc storage system sends this RPC to the intelligent controller to keep it updated about the changes. Moreover, this RPC is send with the initial ad hoc storage system configurations once it initially launched.

See Deliverable D6.1 for intelligent controller information.

3.2.3 Interaction with sensing and profiling

Depending on the events and performance counters monitored (may differ per ad hoc storage system), the ad hoc storage systems will integrate the Prometheus subsystem to send performance-related information to ADMIRE's sensing and profiling module.

Algorithm 12: Definition of the `ADM_adhoc_resource_state` runtime interface

Name: *ADM_adhoc_resource_state*
ID: ID18
Type: Push
Input: Prometheus monitoring data
Output: A *err* code determining whether the operation was successful
Description:
Uses the Prometheus exporter to send monitoring information to the Prometheus database.

See Deliverable D5.1 for sensing and profiling information.

3.3 UML diagram

Figure 3.2 presents the UML diagram from WP2's perspective. It includes only the components the ad hoc storage systems interact with, that is, the parallel file system, the sensing and profiling module (WP5), the intelligent controller (WP6), and the I/O scheduler (WP4) which controls the ad hoc storage systems. For a full UML diagram, refer to D6.1.

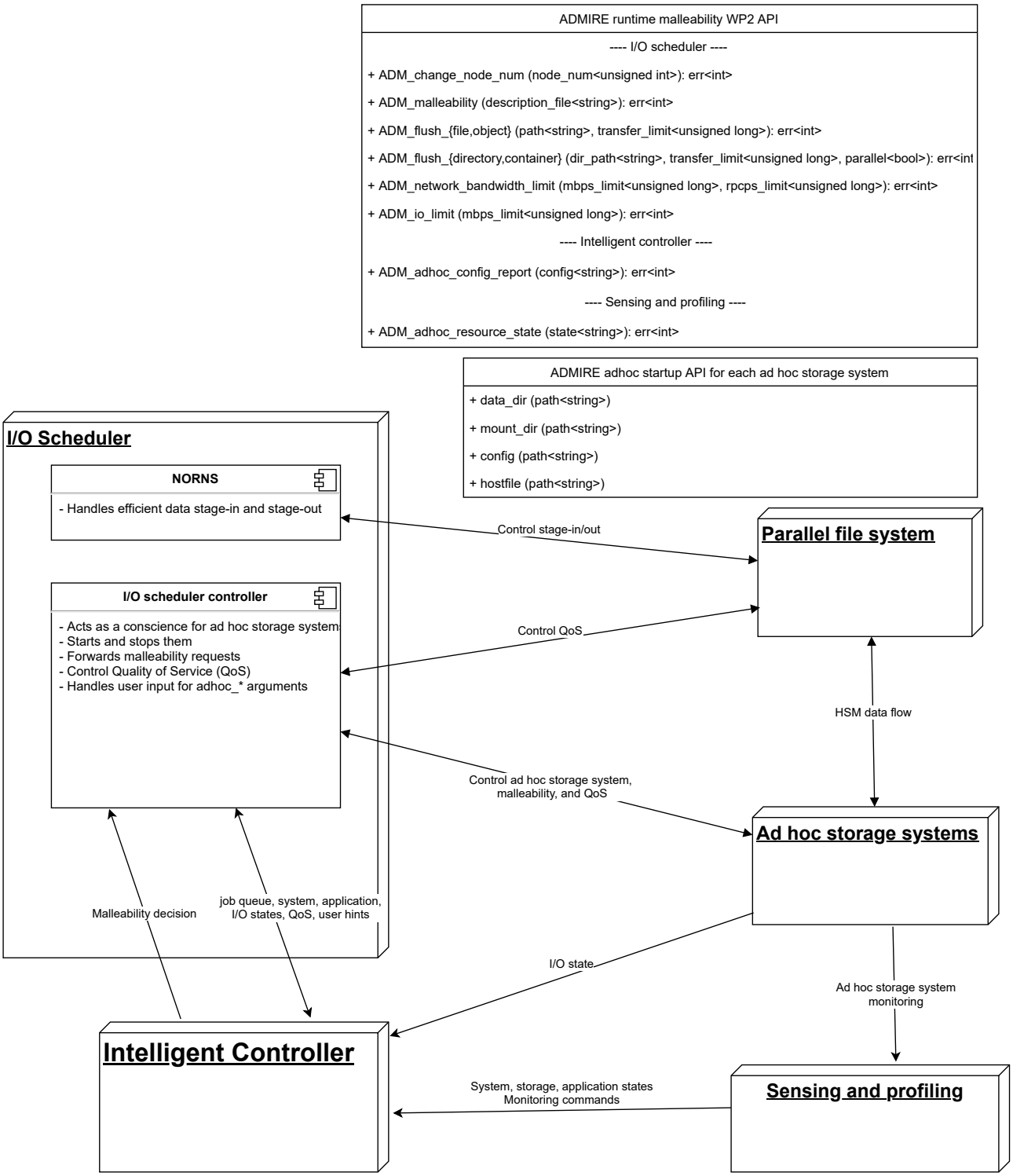


Figure 3.2: ADMIRE UML diagram from WP2 perspective.

4. Application requirements

In this chapter, we will summarize what we found with a preliminary analysis of the applications in I/O terms (understanding I/O as persisting data to the PFS). The detailed information about each application can be found in the deliverable of WP7: D7.1.

4.1 Summary

In general, all the applications use small data at the start, although there are exceptions: Checkpoint files (restarting a job) or DeepLearning / Machine learning. Then applications start the computation phase and start writing data: Checkpoints, intermediate steps, or small logs.

These computations and I/O phase are the ones that will benefit from ADMIRE by, for example, reducing the PFS congestion and using the I/O in the local storage using the ad hoc storage systems. Finally, at the end of the job, the output files will be needed to be moved to the PFS. This will be an extra step that it is not included in the original job as the files are generated directly (thus generating congestion) to the PFS, but it is the one that will be scheduled accordingly with the I/O Scheduler.

Deep Learning (DL) applications, have a different behavior. They use a continuous load of information (i.e., images) on all the phases. In those cases, deciding if we move data to a local storage (ad hoc storage system) or to leave in the PFS will be a decision of the scheduler to reduce system overload and interferences. It seems reasonable that the data will be still being loaded from the PFS if there is no data reuse, but with a coordinated co-design, the application can inform to the I/O Scheduler and wait until the system will not be affected by the operation.

In the context of DL, we have published a conference paper [37], authored by the ADMIRE partners JGU and BSC, to the CLUSTER 2021 conference on how DL I/O can be streamlined with ad hoc file systems. Among others, we proposed a transparent DL workflow using GekkoFS which does not rely on data packing to ensure unbiased data samples since the ad hoc file system creates a shared global namespace across all nodes. Data packing is a technique in DL to split large training data, that can not fit within the storage of a single node, into many parts. Nevertheless, data packing is a challenging and difficult process while biases are not immediately obvious. With an ad hoc file system, this is no longer an issue making DL application significantly easier to use while, in addition, the file system can provide scalable I/O performance. For the DL applications in ADMIRE, we will build on these insights to further improve DL applications performance by employing ad hoc storage systems.

In the following, we briefly present the ADMIRE applications, but we refer to D7.1 for details. In addition, we provide an itemized summary of the particular I/O-related characteristics for each application.

4.2 Environmental Sciences

WaComM++ is an enhanced version of WaComM – Water quality Community Model. WaComM++ supports distributed memory (MPI), shared memory (OpenMP), and GPU (CUDA) parallelization.

Water Community Model (WaComM) uses a particle-based Lagrangian approach relying on tridimensional marine dynamics produced by coupled Eulerian atmosphere and ocean models. WaComM has been developed to match the hierarchical and heterogeneous parallelization design requirements.

Before the simulation is started, all the NetCDF files containing the ocean state (produced by an ocean dynamics model) must be available to all computing nodes that will be used for distributed memory computation.

For each simulated time unit a NetCDF file with the concentration of the particles is produced. A restart data file is produced at the end of each simulation and read at the start of the next simulation.

All the input and output is performed using POSIX I/O without use of MPI I/O or other APIs.

- Needs a distributed PFS input/output
- NetCDF format via POSIX I/O
- Bursty during initialization and finish

4.3 Molecule Simulation

Quantum ESPRESSO mainly causes a bursty I/O behavior and is oriented to intensive writing to file (mainly checkpoint files and data files related to specific physical quantities, e.g., a wave function) with a specific frequency that is every n steps (this can be changed in input files). Reading from file is minimal and focused in the initial setup when reading from input file and pseudo-potential files. The I/O access is sequential and managed by a single rank (typically the zeroth) in a MPI run. The application makes use of POSIX I/O with standard Fortran read and write commands.

All the I/O occurs in the global shared file system and there is no multi-staging. Input data, apart from configuration run file, require only few files. They are all read at once and they are of small file sizes (600 MiB on average). The output data size depends on the simulated case system and the input parameters provided in the configuration run file.

- Bursty I/O
- Intensive writing to file
- Checkpoint files
- Specific frequency (can be predicted)
- Minimal reading
- I/O Access by node master, sequential
- POSIX I/O - Fortran

4.4 Turbulence Simulation

In general, Nek5000 is cyclic, each x computational cycles (defined by the user) writes data for the results and checkpoint/restart. The application in general is write-oriented. The writes are done to a shared file (MPI I/O Collective) with its format directly in the PFS.

- MPI I/O
- PFS involved directly during output

Initial experiments with GekkoFS have already shown a runtime improvement of around 20% for a 16 node experiment.

4.5 Remote Sensing

The remote sensing application trains a ResNet CNN on BigEarthNet, a large remote sensing dataset. The training of the model is performed with TensorFlow. We scaled our application on multiple GPUs with Horovod. The dataset is loaded from the *tfRecord* files – a file type containing a sequence of binary records.

The application makes use of the Tensorflow Dataset API for data loading. For each iteration during the training of the DL model, the batches are loaded by multiple workers from the tfRecord training and validation files. In the future, we would like to understand whether the data loading scales linearly w.r.t. the number of workers and if this operation is a bottleneck.

The checkpoints are written only by one worker every 10 *epochs*[1] and should not impact significantly the training time.

- Python application
- HDF5, read oriented, multiple workers
- Uses Tensorflow and Horovod

4.6 Life Sciences

NanoJ-SRRF (pronounced as *surf*) is a novel open-source and high-performance analytical approach for Live-cell Super-Resolution Microscopy, provided as a fast GPU-enabled ImageJ plugin. The general I/O behaviour of SRRF should be as follows: The program reads the sequence of images (all at once). At this point hundreds or even thousands of images can be read. Next, computations on these images are performed (preferably on GPU), but no checkpoints are saved in the meantime The result is a single output image, which is saved.

Further experiments are required to understand the I/O footprint of this application.

4.7 Software Heritage Analytics

Software Heritage (SH) is a software stack that exposes an API to memorize and retrieve software repositories worldwide. The mission of SH is to preserve all the open-source code ever produced by humans. It collects and preserves software in source code form to avoid the risk of losing it. The SH storage is 400 TiB+ and continuously growing, and the median file size is about 3 KiB. The read access pattern is random, whereas file writing is done based on the hashing of the file, thus no spatial locality is present. Software Heritage is a service and not an HPC parallel application. Therefore, it has no stage-in/out phases. SH is designed to store data and not to analyze them. In its interactive usage, files are directly retrieved from the remote server via the web API.

The Software Heritage Analytics (SHAnalytics) builds on top of SH APIs. It exploits a local copy of the SH data and metadata, implements a batch extraction of the files to be analyzed, and then produces a stream of them towards one or more data analytics applications implemented in Spark Streaming. The SHAnalytics framework can run user-defined data analysis applications and enable Stream Data Analytics applications to execute on data files stored in the SH repository. The SHAnalytics architecture leverages CAPIO (Cross-Applications Programmable I/O) components. Among the other features, it implements a cross-application system-wide file cache of a predefined size that stores the most recent files extracted from the SH repository. The files cache exploits the potential temporal and spatial locality in accessing the same files from different data analytics applications running at the same time.

The shared files cache implementation based on standard POSIX API, can potentially exploit the features of ADMIRE's ad hoc storage systems to improve the overall performance of the SHAnalytics framework.

[1] In a single epoch each training sample is applied once to the model

5. Conclusion

In this deliverable, we have first presented all ADMIRE ad hoc storage systems that we will leverage on within ADMIRE. GekkoFS, dataClay, IMMS, and Extend will provide a wide range of options to the users and applications from traditional file systems, library interfaces and object store systems. Developed in former EU projects, with ADMIRE we have an opportunity to include new features that are needed, e.g., malleability to react to an application's I/O needs. The ability to extend or shrink entire storage systems will provide a unique opportunity to use an HPC systems I/O resources much more efficiently that can benefit both the HPC system and the applications. Nevertheless, resizing storage system is not a trivial task and likely involves the redistribution of at least some amounts of data. Moreover, the ad hoc storage system should be able to remain accessible to the application (albeit temporarily with less I/O capabilities due to the redistribution). This will be one of the major challenges and a great opportunity to research new data distribution (and redistribution) schemes. Other exciting research directions within the context of ADMIRE involve 1. optimizing the storage of data based on their type and access (read-only data can have different consistency requirements than read-write data); 2. batching I/O operations for more efficient communication particularly when handling huge amounts of small files and metadata; or 3. offering granular resilience mechanisms and cache consistency protocols that only apply to some parts of the storage system's namespace.

In addition, we have presented the ad hoc storage system APIs to incorporate them into the ADMIRE stack involving APIs for the startup and runtime malleability. Finally, we have provided an overview of the I/O requirements of the ADMIRE applications and how ad hoc storage systems can benefit them.

Appendix A

Terminology

- Ad hoc Storage System, ephemeral storage system that only exists in a determined period, i.e. during a job's execution.
- API, Application Programming Interface, a mechanism that enables an application or service to access a resource within another application or service. The application or service doing the accessing is called the client, and the application or service containing the resource is called the server.
- CLI, command line interface.
- DRAM, dynamic random-access memory.
- Disaggregated Storage, storage systems where all the storage capabilities are centralized in dedicated network attached storage servers. This approach allows connected compute nodes to access a storage capacity without constraints related to the capacity of a single storage device.
- EBNF, Extended Backus–Naur Form is a family of metasyntax notations, any of which can be used to express a context-free grammar. EBNF is used to make a formal description of a formal language such as a computer programming language. They are extensions of the basic Backus–Naur form (BNF) metasyntax notation.
- Ephemeral Storage, file systems which are making persistent (surviving across system reboot) but which are designed to be deployed and destroyed over a limited period of time, from few hours up to few months.
- In situ data, processing the data where it is originated.
- In transit data, processing the data when it is moved.
- Intelligent Controller,
- MDS, MetaData Server.
- MDT, MetaData Target.
- Monitoring Manager,
- Node Local Storage, ability for a compute server to store persistent data on physically local storage devices.
- NORNS, data transfer service for HPC developed at BSC.
- NVM, non-volatile memory.
- Object store, persistent storage system where data are stored not as file but as objects. In its canonical implementation Object are immutable and the API is limited to PUT, GET and DELETE. More sophistical object store have been developed on the ground of these concepts such as ADMIRE Data Clay.

- OSS, an Object Store Server in the Lustre terminology is a computing server in charge of managing the ingest of data, including generation of the data protection, and ship these data to the correct Object Store Target.
- OST, Object Store Target in the Lustre terminology is a storage server accommodating potentially a large number of hard drives and/or NMVes. The OST write the data received from the OSS and make them persistent.
- PFS, Parallel File System, type of distributed file system supporting a global namespace and spread across multiple storage servers.
- POSIX, Portable Operating System Interface, family of standardized functions.
- PromQL, the query language supported by the Prometheus database. Syntax, documentation and examples are available here: `https://prometheus.io/docs/prometheus/latest/querying`.
- QoS, Quality of Service.
- Rest API, such APIs can be developed without constraint and the programming language and support a variety of data formats. The only requirement is that they align to the following six REST design principles - Uniform interface, Client-server decoupling, Statelessness, cacheability, code on demand (optional).
- RDMA, remote direct memory access.
- RPC, remote procedure call.
- Slurm, job submission system widely used.
- SSD, solid state drive.
- Stripe, an elementary chunk of data according to the Lustre terminology. A large file is split in multiple stripes and each stripe is sent to an individual OST. The higher is the number of stride, the higher is the parallelism.
- TBON, Tree Based Overlay Network,
- Monitoring Daemon,

Bibliography

[1] Ilka Antcheva, Maarten Ballintijn, Bertrand Bellenot, Marek Biskup, Rene Brun, Nenad Buncic, et al. ROOT - A C++ framework for petabyte data storage, statistical analysis and visualization. *Computer Physics Communications*, 182(6):1384–1385, 2011.

[2] Borja Bergua, Felix García Carballeira, Alejandro Calderón, Luis Miguel Sánchez, and Jesús Carretero. Comparing Grid Data Transfer Technologies in the Expand Parallel File System. In *Proceedings of the 16th Euromicro Conference on Parallel, Distributed and Network-Based Processing (PDP 2008)*, PDP '08, pages 110–114, 2008.

[3] Peter J Braam and Philip Schwan. Lustre: The intergalactic file system. In *Ottawa Linux Symposium*, page 50, 2002.

[4] Tim Bray, Jean Paoli, C Michael Sperberg-McQueen, Eve Maler, François Yergeau, and John Cowan. Extensible markup language (xml) 1.0, 2000.

[5] André Brinkmann, Kathryn Mohror, Weikuan Yu, Philip H. Carns, Toni Cortes, Scott Klasky, Alberto Miranda, Franz-Josef Pfreundt, Robert B. Ross, and Marc-Andre Vef. Ad hoc file systems for high-performance computing. *J. Comput. Sci. Technol.*, 35(1):4–26, 2020.

[6] Félix García Carballeira, Alejandro Calderón, Jesús Carretero, Javier Fernández, and José María Pérez. The design of the expand parallel file system. *Int. J. High Perform. Comput. Appl.*, 17(1):21–37, 2003.

[7] Giuseppe Congiu, Sai Narasimhamurthy, Tim Süß, and André Brinkmann. Improving collective I/O performance using non-volatile memory devices. In *IEEE International Conference on Cluster Computing (CLUSTER), Taipei, Taiwan, September 12-16*, pages 120–129, 2016.

[8] Phyllis Crandall, Ruth A. Aydt, Andrew A. Chien, and Daniel A. Reed. Input/output characteristics of scalable parallel applications. In *Proceedings Supercomputing '95, San Diego, CA, USA, December 4-8, 1995*, page 59, 1995.

[9] Siying Dong, Mark Callaghan, Leonidas Galanis, Dhruba Borthakur, Tony Savor, and Michael Strum. Optimizing space amplification in rocksdb. In *CIDR 2017, 8th Biennial Conference on Innovative Data Systems Research, Chaminade, CA, USA, January 8-11, 2017, Online Proceedings*, 2017.

[10] Matthieu Dorier, Gabriel Antoniu, Robert B. Ross, Dries Kimpe, and Shadi Ibrahim. Calciom: Mitigating I/O interference in HPC systems through cross-application coordination. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium, Phoenix, AZ, USA, May 19-23, 2014*, pages 155–164, 2014.

[11] Francisco Rodrigo Duro, Javier Garcia Blas, and Jesus Carretero. A hierarchical parallel storage system based on distributed memory for large scale systems. In *Proceedings of the 20th European MPI Users' Group Meeting*, pages 139–140, 2013.

[12] Francisco Rodrigo Duro, Javier Garcia Blas, Florin Isaila, Jesus Carretero, JM Wozniak, and Rob Ross. Exploiting data locality in swift/t workflows using hercules. In *Proc. NESUS Workshop*, 2014.

[13] Robert C. Edgar. Search and clustering orders of magnitude faster than BLAST. *Bioinformatics*, 26(19):2460–2461, 2010.

[14] Kurt Ferreira, Rolf Riesen, Ron Oldfield, Jon Stearley, James Laros, Kevin Pedretti, Ron Brightwell, and Todd Kordenbrock. Increasing fault resiliency in a message-passing environment. Technical Report SAND2009-6753, Sandia National Laboratories, 2009.

[15] Alvaro Frank, Manuel Baumgartner, Reza Salkhordeh, and André Brinkmann. Improving checkpointing intervals by considering individual job failure probabilities. In *35th IEEE International Parallel and Distributed Processing Symposium, IPDPS 2021, Portland, OR, USA, May 17-21, 2021*, pages 299–309. IEEE, 2021.

[16] Alvaro Frank, Tim Süß, and André Brinkmann. Effects and benefits of node sharing strategies in HPC batch systems. In *2019 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2019, Rio de Janeiro, Brazil, May 20-24, 2019*, pages 43–53. IEEE, 2019.

[17] LM Sánchez García, Florin D Isaila, Félix García Carballeira, Jesús Carretero Pérez, Rolf Rabenseifner, and Panagiotis Adamidis. A new i/o architecture for improving the performance in large scale clusters. In *International Conference on Computational Science and Its Applications*, pages 108–117. Springer, 2006.

[18] Tony Hey, Stewart Tansley, and Kristin M. Tolle, editors. *The Fourth Paradigm: Data-Intensive Scientific Discovery*. Microsoft Research, 2009.

[19] Pieter Hintjens. *ZeroMQ: messaging for many applications*. " O'Reilly Media, Inc.", 2013.

[20] Qianbo Huai, Windsor Hsu, Jiwei Lu, Hao Liang, Haobo Xu, and Wei Chen. XFUSE: an infrastructure for running filesystem services in user space. In *2021 USENIX Annual Technical Conference, USENIX ATC 2021, July 14-16, 2021*, pages 863–875, 2021.

[21] Florin Isaila, Javier Garcia Blas, Jesus Carretero, Wei-keng Liao, and Alok Choudhary. Ahpios: An mpi-based ad hoc parallel i/o system. In *2008 14th IEEE International Conference on Parallel and Distributed Systems*, pages 253–260. IEEE, 2008.

[22] Tanzima Zerin Islam, Kathryn Mohror, Saurabh Bagchi, Adam Moody, Bronis R. de Supinski, and Rudolf Eigenmann. Mcrengine: a scalable checkpointing system using data-aware aggregation and compression. In *Conference on High Performance Computing Networking, Storage and Analysis (SC), Salt Lake City, UT, USA - November 11 – 15*, 2012.

[23] Jürgen Kaiser, Ramy Gad, Tim Süß, Federico Padua, Lars Nagel, and André Brinkmann. Deduplication potential of HPC applications' checkpoints. In *IEEE International Conference on Cluster Computing (CLUSTER), Taipei, Taiwan, September 12-16*, pages 413–422, 2016.

[24] Martin Kleppmann. *Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems*. O'Reilly UK Ltd., 2017.

[25] Johannes Köster and Sven Rahmann. Snakemake - a scalable bioinformatics workflow engine. *Bioinformatics*, 34(20):3600, 2018.

[26] Joel Lauener and Wojciech Sliwinski. Jacow: How to design & implement a modern communication middleware based on zeromq. In *16th International Conference on Accelerator and Large Experimental Physics Control Systems. 8 - 13 Oct*, 2017.

[27] Paul Hermann Lensing, Toni Cortes, Jim Hughes, and André Brinkmann. File system scalability with highly decentralized metadata on independent storage devices. In *IEEE/ACM 16th International Symposium on Cluster, Cloud and Grid Computing (CCGrid), Cartagena, Colombia, May 16-19*, pages 366–375, 2016.

[28] Jonathan Martí, Anna Queralt, Daniel Gasull, Alex Barceló, Juan José Costa, and Toni Cortes. Dataclay: A distributed data store for effective inter-player data sharing. *Journal of Systems and Software*, 131:129–145, 2017.

[29] Alberto Miranda, Adrian Jackson, Tommaso Tocci, Iakovos Panourgias, and Ramon Nou. NORNS: extending slurm to support data-driven workflows through asynchronous data staging. In *2019 IEEE International Conference on Cluster Computing, CLUSTER 2019, Albuquerque, NM, USA, September 23-26, 2019*, pages 1–12. IEEE, 2019.

[30] Adam Moody, Greg Bronevetsky, Kathryn Mohror, and Bronis R. de Supinski. Design, modeling, and evaluation of a scalable multi-level checkpointing system. In *Conference on High Performance Computing Networking, Storage and Analysis (SC), New Orleans, LA, USA, November 13-19, 2010*, 2010.

[31] Nils Nieuwejaar, David Kotz, Apratim Purakayastha, Carla Schlatter Ellis, and Michael L. Best. File-access characteristics of parallel scientific workloads. *IEEE Trans. Parallel Distrib. Syst.*, 7(10):1075–1089, 1996.

[32] Fabrizio Petrini. Scaling to thousands of processors with buffer coscheduling. In *Scaling to New Height Workshop*, 2002.

[33] Ian R. Philp. Software failures and the road to a petaflop machine. In *Proceedings of the 1st Workshop on High Performance Computing Reliability Issues (HPCRI)*, 2005.

[34] Yingjin Qian, Xi Li, Shuichi Ihara, Andreas Dilger, Carlos Thomaz, Shilong Wang, Wen Cheng, Chunyan Li, Lingfang Zeng, Fang Wang, Dan Feng, Tim Süß, and André Brinkmann. LPCC: hierarchical persistent client caching for lustre. In Michela Taufer, Pavan Balaji, and Antonio J. Peña, editors, *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2019, Denver, Colorado, USA, November 17-19, 2019*, pages 88:1–88:14. ACM, 2019.

[35] Yingjin Qian, Xi Li, Shuichi Ihara, Lingfang Zeng, Jürgen Kaiser, Tim Süß, and André Brinkmann. A configurable rule based classful token bucket filter network request scheduler for the lustre file system. In Bernd Mohr and Padma Raghavan, editors, *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2017, Denver, CO, USA, November 12 - 17, 2017*, pages 6:1–6:12. ACM, 2017.

[36] Robert Ross, Rajeev Thakur, and Alok Choudhary. Achievements and challenges for i/o in computational science. In *Journal of Physics: Conference Series*, volume 16, page 501, 2005.

[37] Frederic Schimmelpfennig, Marc-André Vef, Reza Salkhordeh, Alberto Miranda, Ramon Nou, and André Brinkmann. Streamlining distributed deep learning i/o with ad hoc file systems. In *IEEE International Conference on Cluster Computing, CLUSTER 2021, Portland, USA, September 07-10, 2021*. IEEE, 2021. (Accepted for publication).

[38] Frank B. Schmuck and Roger L. Haskin. GPFS: A shared-disk file system for large computing clusters. In Darrell D. E. Long, editor, *Proceedings of the FAST '02 Conference on File and Storage Technologies, January 28-30, 2002, Monterey, California, USA*, pages 231–244. USENIX, 2002.

[39] Jérome Soumagne, Dries Kimpe, Judicael A. Zounmevo, Mohamad Chaarawi, Quincey Koziol, Ahmad Afsahi, and Robert B. Ross. Mercury: Enabling remote procedure call for high-performance computing. In *2013 IEEE International Conference on Cluster Computing, CLUSTER 2013, Indianapolis, IN, USA, September 23-27, 2013*, pages 1–8, 2013.

[40] Sagar Thapaliya, Purushotham Bangalore, Jay F. Lofstead, Kathryn Mohror, and Adam Moody. Managing I/O interference in a shared burst buffer system. In *45th International Conference on Parallel Processing, ICPP 2016, Philadelphia, PA, USA, August 16-19, 2016*, pages 416–425, 2016.

[41] Marc-Andre Vef, Nafiseh Moti, Tim Süß, Markus Tacke, Tommaso Tocci, Ramon Nou, Alberto Miranda, Toni Cortes, and André Brinkmann. Gekkofs - A temporary burst buffer file system for HPC applications. *J. Comput. Sci. Technol.*, 35(1):72–91, 2020.

[42] Marc-Andre Vef, Nafiseh Moti, Tim Süß, Tommaso Tocci, Ramon Nou, Alberto Miranda, Toni Cortes, and André Brinkmann. Gekkofs - A temporary distributed file system for HPC applications. In *IEEE International Conference on Cluster Computing, CLUSTER 2018, Belfast, UK, September 10-13, 2018*, pages 319–324. IEEE Computer Society, 2018.

[43] Marc-André Vef, Vasily Tarasov, Dean Hildebrand, and André Brinkmann. Challenges and solutions for tracing storage systems: A case study with spectrum scale. *ACM Trans. Storage*, 14(2):18:1–18:24, 2018.

[44] Murali Vilayannur, Partho Nath, and Anand Sivasubramaniam. Providing tunable consistency for a parallel file store. In *Proceedings of the FAST '05 Conference on File and Storage Technologies, December 13-16, 2005, San Francisco, California, USA*, 2005.

[45] Feng Wang, Qin Xin, Bo Hong, Scott A Brandt, Ethan Miller, Darrell Long, and T McLarty. File system workload analysis for large scale scientific computing applications. Technical report, Lawrence Livermore National Laboratory (LLNL), Livermore, CA, 2004.

[46] Brent Welch and Geoffrey Noer. Optimizing a hybrid SSD/HDD HPC storage system based on file size distributions. In *IEEE 29th Symposium on Mass Storage Systems and Technologies, MSST 2013, May 6-10, 2013, Long Beach, CA, USA*, pages 1–12. IEEE Computer Society, 2013.

[47] Zhao Zhang, Kyle Barbary, Frank Austin Nothaft, Evan R. Sparks, Oliver Zahn and Michael J. Franklin, David A. Patterson, and Saul Perlmutter. Scientific computing meets big data technology: An astronomy use case. In *2015 IEEE International Conference on Big Data (Big Data), Santa Clara, CA, USA, October 29 - November 1*, pages 918–927, 2015.

[48] Yue Zhu, Fahim Chowdhury, Huansong Fu, Adam Moody, Kathryn Mohror, Kento Sato, and Weikuan Yu. Entropy-aware I/O pipelining for large-scale deep learning on HPC systems. In *26th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS), Milwaukee, WI, USA, September 25-28, 2018*, pages 145–156, 2018.