H2020-JTI-EuroHPC-2019-1

Project no. 956748

ADAPTIVE MULTI-TIER INTELLIGENT DATA MANAGER FOR EXASCALE

D2.2
Design of the ad hoc storage systems

Version 1.0

Date: April 29, 2022

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	21/03/22	Jesus Carretero	UC3M	Document creation.
2	28/03/22	Marc-André Vef	JGU	Created structure.
3	12/04/22	Ramon Nou	BSC	Added GekkoFS changelog.
4	19/04/22	Marc-André Vef	JGU	Updated GekkoFS summary section.
5	19/04/22	Marc-André Vef	JGU	Added GekkoFS download and installation section.
6	19/04/22	Marc-André Vef	JGU	Added GekkoFS running section.
7	19/04/22	Marc-André Vef	JGU	Updated GekkoFS updates section.
8	20/04/22	Marc-André Vef	JGU	Added GekkoFS use cases section.
9	20/04/22	Marc-André Vef	JGU	Added GekkoFS application compatibility section.
10	20/04/22	Marc-André Vef	JGU	Added executive summary section.
11	20/04/22	Marc-André Vef	JGU	Updated appendix section.
12	20/04/22	Marc-André Vef	JGU	Added introduction section.
13	20/04/22	Marc-André Vef	JGU	Added conclusion section.
14	21/04/22	Diego Camarmas, Félix García-Carballeira, Alejandro Calderón	UC3M	Updated Expand section.
15	24/04/22	Nafiseh Moti	JGU	Added application section.
16	25/04/22	Javier García-Blas	UC3M	Added Hercules IMSS section.
17	25/04/22	Marc-André Vef	JGU	Updated application section.
18	25/04/22	Marc-André Vef	JGU	Updated ad hoc storage system section.
19	25/04/22	Marc-André Vef	JGU	Revised document; First draft finished.
20	27/04/22	Hamid Fard	TUD	Reviewed deliverable.
21	28/04/22	Marc-André Vef	JGU	Reviewed the review and finalized the document.

Executive Summary

Ad hoc storage systems represent a dynamic component within the ADMIRE project to accelerate I/O performance of scientific applications and significantly reduce I/O traffic to the general-purpose storage backends, i.e., shared *parallel file systems*, of *High-Performance Computing* (HPC) clusters. Ad hoc storage systems are usually of an ephemeral nature and live within a particular application context (e.g., an HPC compute job), usually using node-local storage devices (e.g., SSDs). In order to achieve better performance in HPC systems within the ADMIRE project, ad hoc storage systems are integrated and controlled by the *I/O scheduler* (respecting the malleable decisions taken by the *malleability manager*). Moreover, ad hoc storage systems should be malleable themselves and support longer-running workflows, such as *job campaigns*.

In Deliverable 2.1, we provided a detailed analysis of the ad hoc storage systems, their interfaces to the ADMIRE ecosystem, and an initial analysis of the ADMIRE applications. In this deliverable, we will focus on the deployment and the software updates of each ad hoc storage system since Deliverable 2.1. We will further present how to acquire, compile, test and run each ad hoc storage system, and we will discuss their limitations with regards to supporting the ADMIRE applications and how they can be overcome, if applicable. Finally, we will show further insights into the ADMIRE applications with regards to their I/O footprint, which we will continuously work on to understand the benefits of the ad hoc storage systems for each application.

Contents

1 Introduction

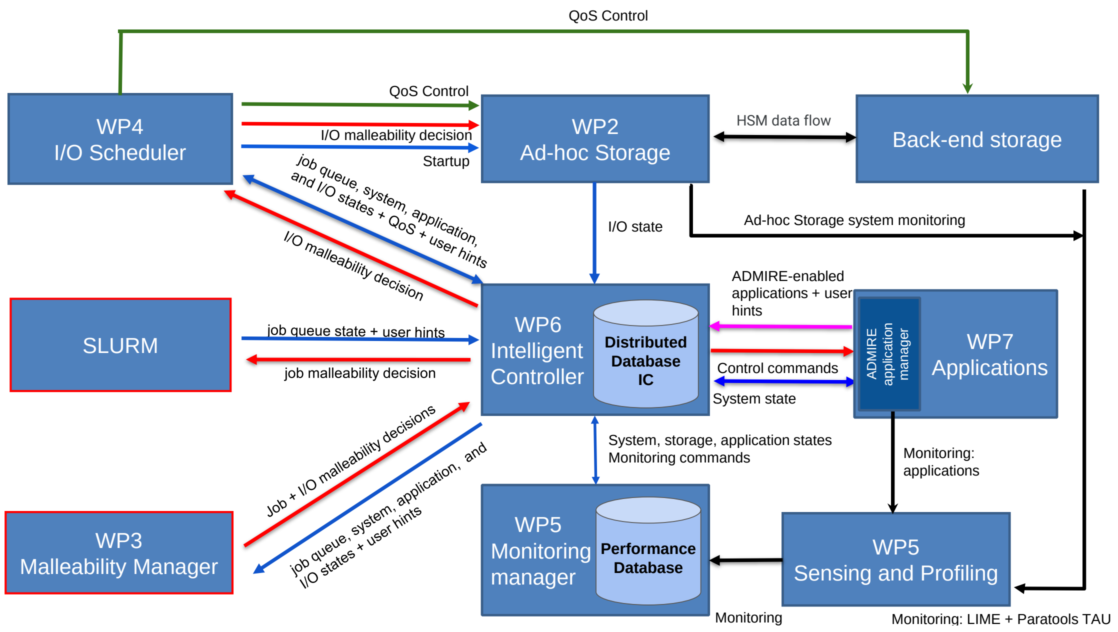

QoS Control
WP4
I/O Scheduler
QoS Control
I/O malleability decision
Startup
WP2
Ad-hoc Storage
HSM data flow
Back-end storage
job queue, system, application, and I/O states + QoS + user hints
I/O malleability decision
I/O state
Ad-hoc Storage system monitoring
ADMIRE-enabled applications + user hints
SLURM
job queue state + user hints
job malleability decision
WP6
Intelligent
Controller
Distributed Database IC
Control commands
System state
ADMIRE application manager
WP7
Applications
Job + I/O malleability decisions
job queue, system, application, and I/O states + user hints
System, storage, application states
Monitoring commands
Monitoring: applications
WP3
Malleability Manager
WP5
Monitoring manager
Performance Database
WP5
Sensing and Profiling
Monitoring
Monitoring: LIME + Paratools TAU

Figure 1.1: ADMIRE architecture overview. Each component developed in the project's scope has included the label of its related work package (WP).

Work package 2 considers several ad hoc storage systems to efficiently use node-local storage technologies, e.g., SSDs or future *non-volatile memories* (NVMs), to reduce the pressure on the main backend storage systems that are used in HPC systems, e.g., Lustre or GPFS. In general, WP2 focuses on three main tasks:

1. The used applications in ADMIRE are analysed to understand how ad hoc storage system should be modified so that applications can benefit from a malleable and dynamic storage system that can react to specific I/O requirements during runtime (see D2.1 for details).

2. The ad hoc storage systems are further developed and integrated into the ADMIRE ecosystem adding support for fast storage technologies and new data placement algorithms.

3. Resilience mechanisms are added to the ad hoc storage systems so that they can run for longer periods, such as in application workflows which consist of multiple consecutive compute jobs that can use the same shared namespace without moving data.

Figure 1.1 presents an overview of the ADMIRE ecosystem with the ad hoc storage system being connected

1. to the *I/O scheduler* (WP4) which controls and manages the ad hoc storage system instances;

2. to the *sensing and profiling* module (WP5) which receives the current ad hoc storage system state and performance counters;

3. to the *intelligent controller* (WP6) which receives malleability confirmations after forwarding malleability decisions from the *malleability manager* (WP3) to the ad hoc storage system via the I/O scheduler; and

4. to the *backend storage*, e.g., Lustre, in which an ad hoc storage system is used within the backend storage system's *hierarchical storage management* (HSM).

First, Chapter 2 of this prototype deliverable will focus on the usage of the ad hoc storage systems *GekkoFS*, *dataClay*, *Expand*, and *Hercules IMSS* with regards to where they can be acquired, how they can be built and deployed, how they can be tested, and how they can be practically used by any application. Next, we will discuss each ad hoc storage system's main use cases, their compatibility with the applications that are used in ADMIRE, and how theses limitations could be overcome, if applicable. Moreover, Chapter 2 will present the development and software updates of each ad hoc storage systems since D2.1. Chapter 3 provides new insights into our on-going application analysis concerning their I/O footprint. Finally, Chapter 4 concludes this deliverable.

2 Ad hoc storage systems

This section will discuss all ad hoc storage systems that are used in the ADMIRE project concerning their overall design, their software updates since the last deliverable, and their supposed compatibility of the ADMIRE applications. Therefore, we will include information on what tasks are required to support the ADMIRE applications. Note, that we will include a brief summary of each ad hoc storage system, but we kindly refer to Deliverable 2.1 which introduces each ad hoc storage system in detail.

2.1 GekkoFS

In the following, we will briefly introduce GekkoFS's key points and its updates over the past months, and discuss its application compatibility.

2.1.1 Design summary

GekkoFS [7, 9, 10] is a highly scalable distributed file system for HPC clusters which runs entirely in user space. GekkoFS is capable of aggregating the local I/O capacity and performance of compute nodes to produce a high-performance storage space for applications. Using GekkoFS, HPC applications can run isolated from each other regarding I/O, which reduces interferences and improves performance. Further, 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 behaviour of HPC applications into account [5] to optimise 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 needs to be persisted beyond the lifetime of the job from GekkoFS to a 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 system call interception library that can be preloaded before calling these tools.

Figure 2.1 provides an overview of GekkoFS's architecture. Please refer to Deliverable 2.1 for a detailed introduction in the core design considerations.

2.1.2 Use cases

As described in Deliverable 2.1 in detail, ad hoc storage systems can significantly help in various I/O intensive workloads, such as *bulk-synchronous*, *checkpoint-restart*, or *Deep Learning* workloads [2]. In general, it depends on a given workload and the overall system how significant the runtime improvement can be when used with an ad hoc storage system compared with the main backend storage system, i.e., the *parallel file system* (PFS). One of the main goals of the ADMIRE project is to maximise the efficiency of a supercomputer environment, employing malleability techniques for computation and I/O. Therefore, an ad hoc storage system should be used over the PFS if it is beneficial to the applications runtime.

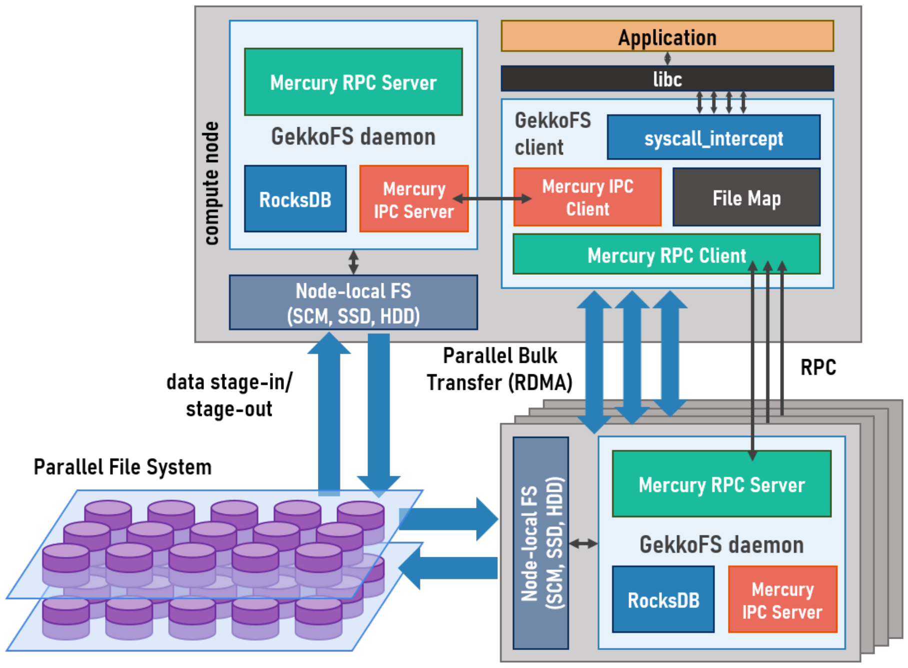


Figure 2.1: The GekkoFS architecture.

For GekkoFS, we target a wide range of distributed workloads that traditionally do not work well on PFSs, e.g., random I/O, small I/O requests, or the generation of many small files. As a result, the storage system can be especially beneficial when applications require a fast, global, and distributed namespace. For example, when many checkpoints need to be created, as it is the case with ADMIRE's *Quantum Espresso* and *Nek5000* applications, GekkoFS could allow for a much higher frequency of checkpoints that could then moved to the PFS asynchronously. Moreover, for Deep Learning (DL) workloads, we have already shown in our most recent publication "*Streamlining distributed Deep Learning I/O with ad hoc file systems*" [7] at the CLUSTER conference in 2021 how GekkoFS can benefit DL workloads with the Tensorflow [1] and Horovod [8] frameworks. This is particularly relevant for the *Remote Sensing* ADMIRE application which is also using Tensorflow and Horovod for distributed learning.

In the next months, we will further investigate the specific benefits of GekkoFS for each ADMIRE application.

2.1.3 Download and installation

GekkoFS is open-source and available at https://storage.bsc.es/gitlab/hpc/gekkofs. Further, a documentation wiki can be found at https://storage.bsc.es/projects/gekkofs/documentation. The wiki is continuously extended with advanced information, such as file system architecture, consistency information, installation and running details, code reference based on Doxygen, and more. In the following, we will present step-by-step instructions for download and installation. More detailed instructions and the required dependencies are available in the wiki.

1. Ensure the required system dependencies are available and at least GCC 8 is installed.

2. Clone the latest GekkoFS version 0.9.1 release[1]:

```
git clone --recurse-submodules --branch v0.9.1
↪  https://storage.bsc.es/gitlab/hpc/gekkofs.git
```

3. Set up the necessary environment variables where the compiled direct GekkoFS dependencies will be installed at (we assume the path `/home/foo/gekkofs_deps/install` in the following):

[1]The current deliverable presents the release of GekkoFS on the 29th of April of 2022 with its corresponding tag *v0.9.1* which can be directly accessed at https://storage.bsc.es/gitlab/hpc/gekkofs/-/tags/v0.9.1.

- `export LD_LIBRARY_PATH=/home/foo/gekkofs_deps/install/lib:`
  `↪ /home/foo/gekkofs_deps/install/lib64`

4. Download and compile the direct dependencies:
   - Download example: `gekkofs/scripts/dl_dep.sh /home/foo/gekkofs_deps/git`
   - Compilation example: `gekkofs/scripts/compile_dep.sh /home/foo/gekkofs_deps/git /home/foo/gekkofs_deps/install`
   - Consult `-h` for additional arguments for each script.

5. Compile GekkoFS and run optional tests:
   - Create build directory: `mkdir gekkofs/build && cd gekkofs/build`
   - Configure GekkoFS:

     ```
     cmake -DCMAKE_BUILD_TYPE=Release
     ↪  -DCMAKE_PREFIX_PATH=/home/foo/gekkofs_deps/install ..
     ```

     – Add `-DCMAKE_INSTALL_PREFIX=<install_path>` where the GekkoFS client library and server executable should be available.

     – Add `-DGKFS_BUILD_TESTS=ON` if tests should be build.

6. Build and install GekkoFS: `make -j install`

GekkoFS is now available at:

- GekkoFS daemon (server): `<install_path>/bin/gkfs_daemon`
- GekkoFS client interception library: `<install_path>/lib64/libgkfs_intercept.so`

2.1.4 Running GekkoFS

GekkoFS needs two steps to be used. First, we need to start the servers before any client can use 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).

The 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}/gkfs_mnt"

# Where we put the data files of GekkoFS
export GKFS_ROOT="${TMP_PATH}/gkfs_root"
# Sets a shared file to know which servers are available.
export GKFS_HOSTS_FILE=${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 (see WP 4).

2.1.5 Updates

The following list includes the updates and progress since the last deliverable in October 2021 for GekkoFS versions v0.9.0 and v0.9.1.

New

1. Added a new script for starting and stopping daemons on multiple nodes (beta version)
2. Added Statistic gathering on daemons
   - Stats output can be enabled with:
     - `--enable-collection` collects normal statistics
     - `--enable-chunkstats` collects extended chunk statistics
   - Statistics output to file is controlled by `--output-stats <filename>`.
3. Added *Prometheus* output
   - New option to define gateway `--prometheus-gateway <gateway:port>`
   - Prometheus output is optional with "GKFS_ENABLE_PROMETHEUS"
   - `--enable-prometheus` creates a thread to push the metrics
4. Added new experimental metadata backend: *Parallax*
5. Added support to use multiple metadata backends.
6. Added –clean-rootdir-finish argument to remove rootdir/metadir at the end when the daemon finishes.
7. GekkoFS now uses C++17.
8. Added a new dirents_extended function which can improve find operations.
9. Code coverage reports for the source code are now generated and tracked
10. Considerable overhaul and new features of the GekkoFS testing facilities
11. Namespace have been added to the complete GekkoFS codebase

12. The system call socketcall() is now supported
13. System call error codes are now checked in syscall_no_intercept scenarios in non x86 architectures.
14. GekkoFS documentation is now automatically generated and published.
15. Added a guided distributor mode which allows defining a specific distribution of data on a per directory or file basis.
16. A convenience library has been added for unit testing.
17. Code format is now enforced with the *clang-format* tool
18. A new script is available in `scripts/check_format.sh` for easy of use.
19. `GKFS_METADATA_MOD` macro has been added allowing the `MetadataModule` to be logged, among others.
20. A convenience library has been added for `path_util`

Changed

1. `-c` argument has been moved to `--clean-rootdir-finish` and is now used to clean rootdir/metadir on daemon shutdown.
2. Environment variable to change Daemon log levels was changed from `GKFS_LOG_LEVEL` to `GKFS_DAEMON_LOG_LEVEL`.
3. Update Catch2 to support newer glibc library.
4. Adding support for `faccessat2()` and `getxattr()` system calls.
5. GekkoFS license has been changed to *GNU General Public License version 3*
6. Create, stat, and remove operation have been refactored and improved, reducing the number of required RPCs per operation.
7. *Syscall_intercept* now supports *glibc* version 2.3 or newer.
8. All arithmetic operations based on block sizes, and therefore chunk computations, are now constexpr.
9. The CI pipeline has been significantly optimized.
10. The GekkoFS dependency download and compile scripts have been severely refactored and improved.
11. GekkoFS now supports the latest dependency versions.
12. -c argument has been moved to `--clean-rootdir-finish` and is now used to clean rootdir/metadir on daemon shutdown.

Removed

1. Removed old initialization code in the GekkoFS client.
2. Removed boost interval dependencies from guided distributor.
3. *Boost* is no longer used for the client and daemon. Note that tests still require *Boost_preprocessor*.
4. Unneeded sources in *CMake* have been removed.

Fixed

1. Documentation: Doxygen now includes private struct and class members.
2. Guided distributor tests are no longer run when they are turned off.
3. Building tests no longer proceeds if `virtualenv` creation fails.
4. An error where unit tests could not be found has been fixed.
5. The daemon can now be restarted without losing its namespace.
6. An issue has been resolved that required *AGIOS* even if it wasn't been used.
7. Several issues that caused docker images to fail has been resolved.
8. An *CMake* issue in `path_util` that caused the compilation to fail was fixed.
9. Fixed an issue where ls failed because newer kernels use `fstatat()` with `EMPTY_PATH`
10. Fixed an issue where `LOG_OUTPUT_TRUNC` did not work as expected.

2.1.6 ADMIRE application compatibility

GekkoFS is not considered a fully POSIX-compliant file system and relaxes some file system protocols to minimise performance bottlenecks, that are typically occurring in distributed file systems due to complicated global locking mechanisms, and therefore improves the overall I/O performance. Further, GekkoFS is not a kernel-based file system which allows users to easily deploy it without administrative support and uses an interposition library to intercept I/O calls before they are sent to the kernel. Based on studies [5, 11] which analysed HPC applications and their behaviour, GekkoFS aims to optimise for these use cases and therefore supporting most (but not all) applications. For instance, these studies have shown that some operations, e.g., a `rename()`, is not used by the tested HPC applications and it is only used by users directly in an interactive manner [2].

Currently, we are working on supporting `rename()` operations which is one of the use cases in ADMIRE's *Software Heritage* application. In addition, we will also support the `mmap()` and `mmsync()` operations which is required by the Software Heritage application. As GekkoFS supports both POSIX I/O and MPI I/O interfaces, we do not foresee other compatibility issues of ADMIRE applications at this time.

2.2 dataClay

In the following, we will present the dataClay object store design, deployment and its updates.

2.2.1 Design summary

dataClay [6] is a distributed object store with active capabilities. It is designed to hide distribution details while taking advantage of the underlying infrastructure, such as of 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. dataClay can be deployed in two different ways: as a service or as an ephemeral storage system, for which is currently being optimised. 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 assigned for a job, such that the data, which it contains, must be persisted after completing the job, if needed. In both cases, the active capabilities (in-store method execution) of dataClay minimise data transfers and copies,

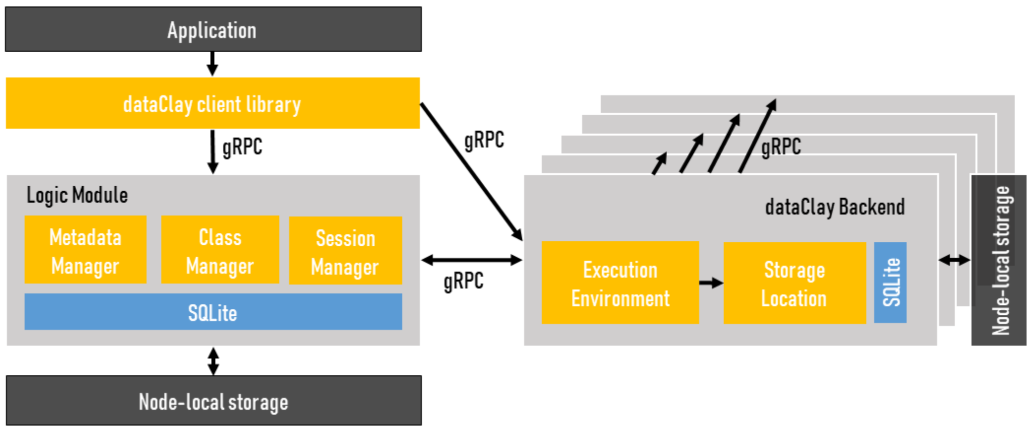


Figure 2.2: dataClay Architecture.

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 by means of an object cache, where objects are already instantiated and ready to serve execution requests.

The architecture of dataClay is depicted in Figure 2.2. Please refer to D2.1 for more details on its design.

2.2.2 Use cases

The goal of dataClay is twofold. First, it provides transparent access to storage, meaning that data can be accessed in the same format as it is managed in memory (i.e. objects). Second, it optimizes applications performance by means of in-store function execution, thus avoiding data transfers from data store to application. Although these are general aspects that can be applied to many kinds of applications, the ones that are most benefited from dataClay are those manipulating objects at a coarse granularity, especially when the results are much smaller than the input data, as much less data is transferred in this case.

2.2.3 Download and installation

dataClay is an open-source product, available at `https://github.com/bsc-dom/dataclay`. Additional material and documentation can be found at `https://www.bsc.es/dataclay`.

In the following, we will outline a brief step-by-step download and installation. More detailed instructions and the required dependencies are available in previous links.

1. Ensure the required system dependencies are available.
2. Clone the dataClay repository:
   ```
   git clone --recurse-submodules https://github.com/bsc-dom/dataclay
   ```
3. Package javaclay source code into a jar file:
   ```
   mvn -f javaclay/pom.xml package
   ```
4. (Optional) For Python applications, install dataClay in Python, following either of the following alternatives:
   - From the pyclay folder:
     ```
     python setup.py install
     ```
   - Using pip:
     ```
     pip install dataclay
     ```

2.2.4 Running dataClay

As detailed in D2.2, the architecture of dataClay is composed of two main components: the *Logic Module* and the *dataClay Backends*. The *Logic Module* is a central repository that handles object metadata and management information. Each *Backend* handles object persistence and execution requests.

In order to deploy dataClay on a cluster of N nodes, one possible setup is to assign 1 node for the *Logic Module* and $N - 1$ nodes for the *Backends* (this scenario can be easily extrapolated to more complex ones, for example, deploying a backend in the *Logic Module* node, or sharing one node for multiple backends.

Considering the proposed setup, the deployment would be as follows:

Deploy the *Logic Module* in one node:

1. Define the necessary environment variables (you may want to change the default values):
   - `export LOGICMODULE_PORT_TCP=11034`
   - `export LOGICMODULE_HOST=127.0.0.1`
   - `export DATACLAY_ADMIN_USER=admin`
   - `export DATACLAY_ADMIN_PASSWORD=admin`
2. Deploy the *Logic Module*:

   ```
   java -cp <jar_path> es.bsc.dataclay.logic.server.LogicModuleSrv
   ```

The rest of the nodes deploy a *Backend* (referred to as `DATASERVICE` in the environment variables):

1. Define the necessary environment variables (you may want to change the default values):
   - `export DATASERVICE_NAME=DS1`
   - `export DATASERVICE_JAVA_PORT_TCP=2127`
   - `export LOGICMODULE_PORT_TCP=11034`
   - `export LOGICMODULE_HOST=127.0.0.1`
2. Deploy the *Storage Location* and the *Java Execution Environment*:

   ```
   java -cp <jar_path> es.bsc.dataclay.dataservice.server.DataServiceSrv
   ```
3. For Python applications, deploy the *Python Execution Environment*:

   ```
   python -m dataclay.executionenv.server --service
   ```

In order to connect your applications with dataClay you need a client library for your preferred programming language. If you are developing a Java application, you can add the following dependency into your pom file to install the Java client library for dataClay version 2.6:

```
<dependency>
    <groupId>es.bsc.dataclay</groupId>
    <artifactId>dataclay</artifactId>
    <version>2.6.1</version>
</dependency>
```

Notice that this section presents the general installation and deployment instructions for dataClay, but most of these steps will be transparent to the user in the context of ADMIRE.

2.2.5 Updates

Although dataClay can be used as an ephemeral data store, it is not optimised for this purpose, and some adaptations are required in order to use it effectively as an ad hoc storage system. An initial list of required changes was described in D2.1, including the optimisation of stage-in/stage-out, enabling malleability, and support for non-volatile memories. Since the last deliverable D2.1, some features in these directions have been incorporated into dataClay.

First, dataClay now incorporates support for non-volatile memories, taking advantage of their byte-addressability to further exploit data locality, avoiding data movements 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. The current implementation is based on `pynvm`[2], a Python implementation of bindings to the Persistent Memory Development Kit (PMDK)[3], and it allows to store and retrieve numpy arrays from a NVM device during a Python application execution. The implementation leverages the byte-addressable nature of NVM through the *Direct Access* (`dax`) feature, which enables direct load/store access to persistent memory by memory-mapping files on a persistent memory aware file system. According to our experiments evaluating this implementation, the benefits provided by the NVM support to dataClay depend on the application access patterns. The greater performance improvements are obtained in read-bound applications with data reuse.

Second, the ability to dynamically increase the number of backends is also supported. This functionality allows to add a new dataClay backend during application execution, which will be used to manage part of the objects created from that point on. At the moment, no redistribution of the previously existing objects is performed.

Finally, we have benchmarked the stage-in process in dataClay and identified that its main bottleneck is the way in which metadata is currently managed. In order to solve this problem and make dataClay appropriate as an ad hoc object store, we have designed a distributed metadata service based on the `etcd` distributed key-value store. This solution provides reliability as well as strong consistency guarantees and, according to our benchmarks, at the same time is much more efficient than the current metadata handling in dataClay. We are currently implementing this approach, which will not only improve the stage-in process but also accelerate other operations during application execution.

2.2.6 ADMIRE application compatibility

Although ADMIRE applications are not currently using object stores (see D7.1), we have identified some initial possibilities for the incorporation of dataClay in order to optimize their execution. A candidate application is *Software Heritage Analytics*, as it exhibits some characteristics that are suitable for dataClay. In particular, it is written in Java and Python (as well as Scala), which are the languages supported by dataClay. Also, the application performs in-situ operations, for which the active capabilities of dataClay can be leveraged.

Another candidate is ADMIRE's *Environment*[4] application. Although it is written in C++, which is currently not supported by dataClay, the fact that it uses NetCDF provides the possibility to abstract the dataClay API behind a NetCDF-compliant interface, making the application agnostic to the incorporation of dataClay, and providing the possibility to optimise its performance by executing part of the operations within the object store, also taking advantage of NVM devices.

2.3 Expand

In the following, we will briefly introduce the key points of Expand ad hoc parallel file system, its updates over the past months, and discuss its application compatibility.

[2]`https://github.com/pmem/pynvm`
[3]`https://github.com/pmem/pmdk`
[4]`https://github.com/ccmmma/wacommplusplus`

2.3.1 Design summary

Expand is a parallel file system based on standard servers, as described in Deliverable D2.1. This section describes the work developed to convert Expand into an ad hoc parallel file system.

Figure 2.3 shows the structure of Expand as an ad hoc parallel file system. This structure is based on a series of data servers running on the compute nodes that communicate with each other using MPI. The use of MPI facilitates the standardisation of Expand and its use in HCP environments.

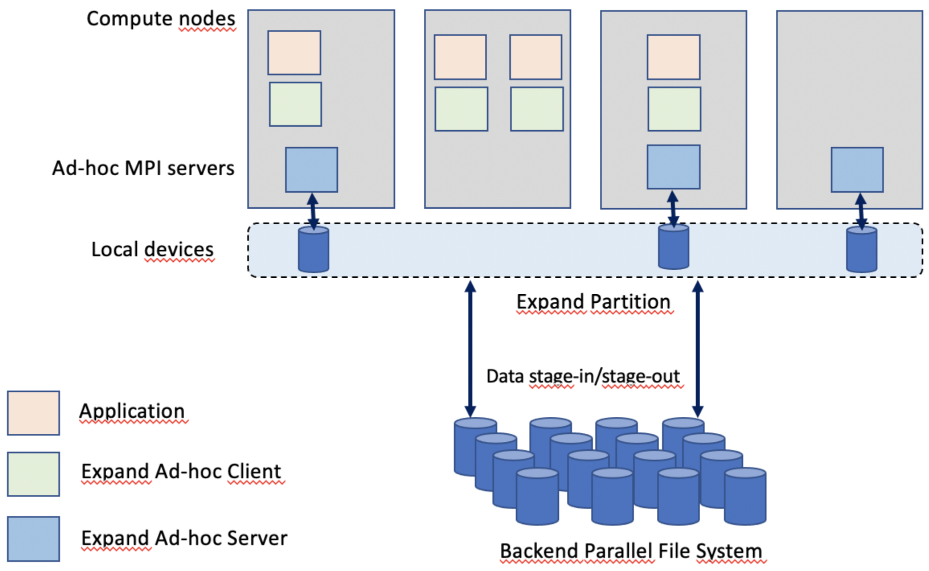


Figure 2.3: The Expand Ad hoc architecture.

Parallel data partitions are created on the ad hoc servers on the local storage devices (HDD or SSD) using the services provided by the local operating system. The applications are deployed on the compute nodes and communicate with the Expand servers also using MPI. As can be seen in the figure, applications can be deployed on nodes without ad hoc servers and servers can be deployed on nodes where no applications are running.

Figure 2.4 shows the internal details of the Expand as ad hoc parallel file system. Next section describes some aspects of Expand design.

2.3.1.1 Data distribution and files

Expand combines several Expand MPI ad hoc servers (see Figure 2.4) in order to provide a generic parallel partition. Each server provide one or more directories that are combined to build a distributed partition to use in compute nodes. All files in the system are striped across all ad hoc servers to facilitate parallel access, with each server storing conceptually a subfile of the parallel file. A file consists of several subfiles, one for each ad hoc server. All subfiles are fully transparent to the Expand users. On a parallel partition, the user can create stripped files with cyclic layout. In these files, blocks are distributed across the partition following a round-robin pattern. As shown in Figure 2.5, the user data file is the file stored in Expand and the block size is the capacity used in Expand for distributed the blocks among all servers. This block size is independent of the structure used in the backend parallel file system.

2.3.1.2 Naming and metadata management

Partitions in Expand are defined using a small configuration file. For example, the following configuration file defines a partition with two ad hoc servers and an block size of 512 KB. For each server, a directory must be specified. All files will be storage in this directory.

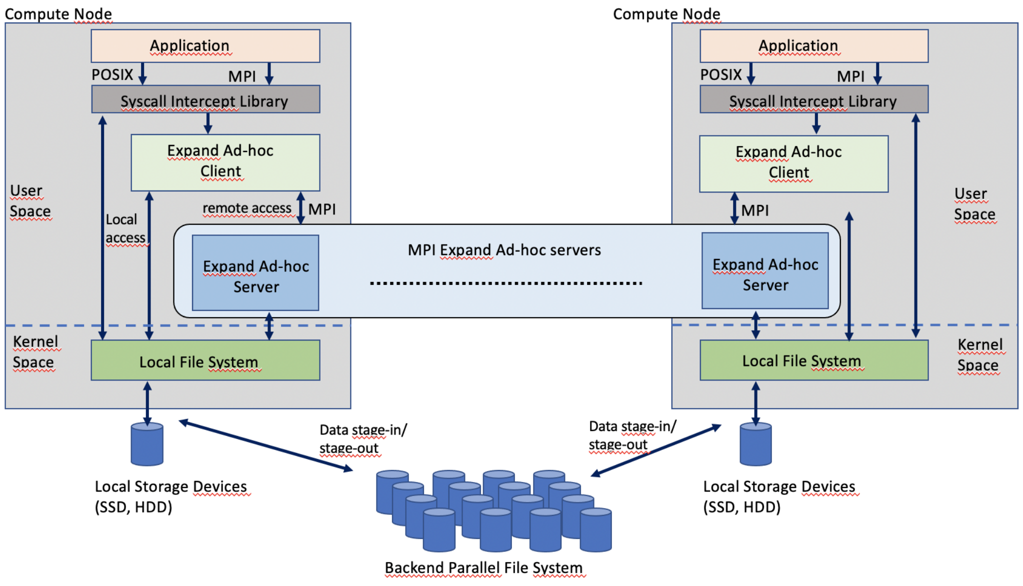


Figure 2.4: The Expand ad hoc detailed architecture.

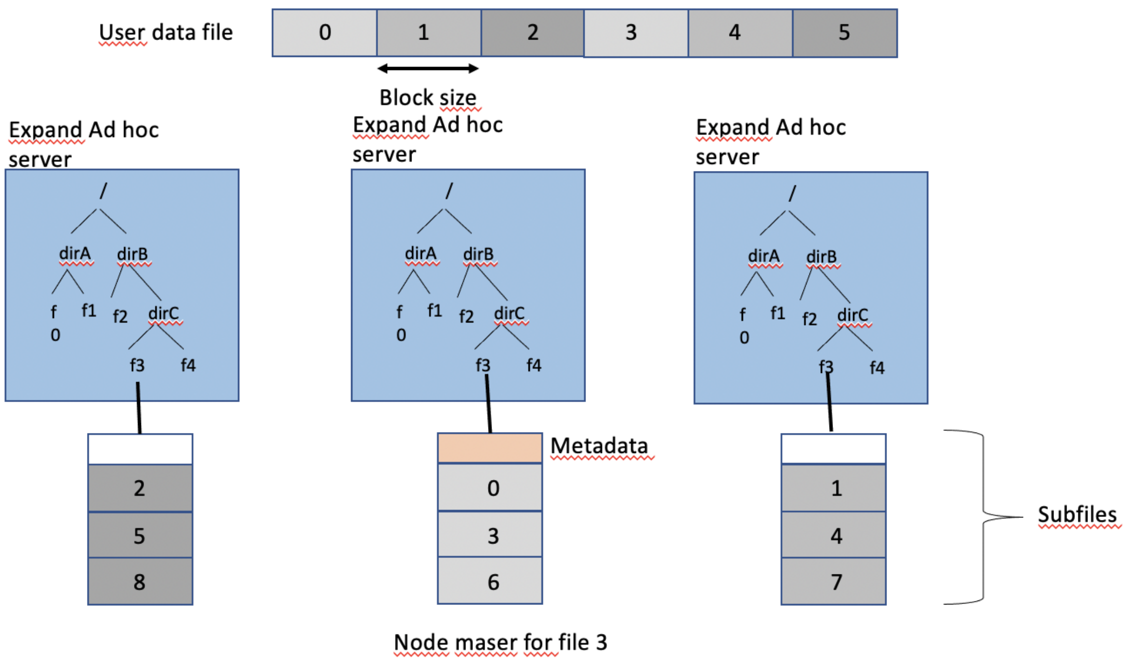


Figure 2.5: The file structure and directory mapping in Expand.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xpn_conf>
  <partition name="xpn" type="NORMAL" bsize="512k"  >
    <data_node id="<node ID>" url="mpiServer://<server name or IP>/<storage path1>"/>
    <data_node id="<node ID>" url="mpiServer://<server name or IP>/<storage path2>"/>
  </partition>
</xpn_conf>
```

Each subfile of a Expand file (see Figure 2.5) has a small header at the beginning of the subfile, which stores the file's metadata. This metadata includes the following information: stride size, base node, that identifies the ad hoc server where the first block of the file resides and the file distribution pattern used. Currently, we only use files with cyclic layout.

All subfiles have a header for metadata, although only one node, called master node stores the current metadata. The master node can be different from the base node. To simplify the naming process and reduce potential bottlenecks, Expand does not use any metadata manager. Figure 2.5 shows how directory mapping is made in Expand.

The metadata of a file resides in the header of a subfile stored in an ad hoc server. This server is the master node of the file. To obtain this node, the file name is hashed into the number of node.

Because the determination of the master node is based on the file name, when a user renames a file, the master node for that file changes. To rename a file, the Expand applies the following algorithm:

```
rename(oldname, newname) {
   oldmaster = hash(oldname)
   newmaster = hash(newname)
   move the metadata subfile from oldmaster to newmaster
}
```

2.3.1.3 Parallel Access

All file operations in Expand use a virtual filehandle. This virtual filehandle is the reference used in Expand to reference all operations. When Expand needs to access to a subfile, it uses the appropriated filehandle. To enhance I/O, user requests are split by the Expand library into parallel subrequests sent to the involved servers. When a request involves k ad hoc servers, Expand issues k requests in parallel to the servers, using threads to parallelise the operations. The same criteria is used in all Expand operations. A parallel operation to k servers is divided into k individual operations that are provided by ad hoc servers.

2.3.1.4 Data stage-in/stage-out operations

Data stage-in operations are performed in parallel from the backend parallel file system to the ad hoc servers without the intervention of any client application. Each server builds the corresponding subfile in its local storage space, reading the data from the file stored on the backend file system. This operation is performed in parallel in all ad hoc servers.

In the same way, data stage-out are performed in parallel. Each server writes the blocks stored in his local subfile to the final file in the backend parallel file system.

Although these operations could be performed using a normal POSIX application, we have developed two new system calls that allow these operations to be performed directly from the servers with improved performance. This system calls are:

- *Preload*, which copies a file from the backend file system to the Expand partition;
- *Flush*, which writes a file from the Expand partition to the final backend file system.

2.3.2 Use cases

Expand provides POSIX interface that is appropriate for a wide range of applications. Nevertheless, in the next months, we will further investigate the workloads more appropriate for Expand.

2.3.3 Download and installation

The Expand File System repository is available at:

- Source code: `https://github.com/xpn-arcos/xpn`
- Documentation: `https://xpn-arcos.github.io/arcos-xpn.github.io/`

The Expand repository is continuously extended with advanced information, such as file system architecture, installation and running details, etc.

In order to deploy Expand, you have to follow the next steps:

1. **Install required software** (pre-requisites).
2. **Download and configure** Expand.
3. **Compile and Install** Expand.

Install the required software

To install the required software, please follow the next steps:

1. Please ensure you have already installed the following system software:

   ```
   sudo apt-get install -y autoconf automake sysutils gcc g++ make libmxml-dev
   ```

2. Install MPICH (at least version 3.4.2 or higher):

   ```
   sudo apt-get install -y mpich libmpich-dev mpich-doc
   ```

Download and configure Expand

1. Clone the last version of the Expand source code:

   ```
   git clone https://github.com/xpn-arcos/xpn.git
   ```

2. Access into the installation directory, which from now on we will denote as `<xpn_path>`:

   ```
   cd xpn
   ```

3. Generate the installation files:

   ```
   ./autogen.sh
   ```

4. Configure Expand, usually:

   ```
   ./configure --prefix=<install_path> --enable-mpiserver=<mpich_path>
   ```

   The `--prefix` is used to change the default directory in which Expand will be installed. This switch is optional, and it common used if your UNIX user do not have enough permissions on the default directory. The `--enable-mpiserver` switch is used to enable the mpiserver module of Expand. If MPICH has not been installed in its default installation path (e.g.: `/usr/bin`), it is necessary to specify the path in which it has been installed (`--enable-mpiserver=<mpich_path>`).

Compile and Install Expand

1. Compile Expand:

   ```
   make -j
   ```

2. Install Expand:

   ```
   make install
   ```

3. Compile the interception library:

   ```
   cd <xpn_path>/bypass && make -j
   ```

4. Compile the mpiserver:

```
cd <xpn_path>/external-utils/mpiServer && make -j
```

The key elements of Expand File System are now available in the following paths:

- Expand client library:
  - `<install_path>/include/xpn.h`
  - `<install_path>/lib/libxpn.a`
- Expand client interception library: `<xpn_path>/bypass/xpn_bypass.so`
- Expand MPI server: `<xpn_path>/external-utils/mpiServer/mpiServer.exe`

2.3.4 Running Expand

The Expand File System is based on the client-server paradigm, such that we need to run the server first, then run the client(s).

Running Expand mpiServer

To start the mpiServers the following steps must be followed:

1. If the MPICH hydra nameserver is not running in the current host, please execute:

   ```
   HYDRA_HOSTNAME=$(hostname)
   [<mpich_path>/bin/]hydra_nameserver &
   sleep 1
   ```

   If you install MPICH with the `--prefix` switch then you need to use the `<mpich_path>/bin/`, otherwise it is not needed.

2. List the machines to be used in the `<hostfile_server>` file.

3. Start the mpiServer (with N processes distributed on the machines listed in the `<hostfile_server>` file) by running:

   ```
   [<mpich_path>/bin/]mpiexec -hostfile <hostfile_server> -np <N> -nameserver
   ↪  ${HYDRA_HOSTNAME} <xpn_path>/external-utils/mpiServer/mpiServer.exe
   ```

Running Expand client

1. Create the configuration file for the Expand Client where you specify the mpiServer nodes (`mpiServer://<server>/<path>`).

2.a Run the existing client binary (with M processes distributed on the machines in the `<hostfile_client>` file) by using:

   ```
   [<mpich_path>/bin/]mpiexec -hostfile <hostfile_client> -np <M> -nameserver
   ↪  ${HYDRA_HOSTNAME} -genv LD_PRELOAD=<xpn_path>/bypass/xpn_bypass.so>
   ↪  <client_binary>
   ```

2.b If the client binary use the Expand API then you can execute (without intercepting the POSIX system calls) by running:

   ```
   [<mpich_path>/bin/]mpiexec -hostfile <hostfile_client> -np <M> -nameserver
   ↪  ${HYDRA_HOSTNAME} <client_binary>
   ```

One example of configuration file for the mpiServer module of Expand Client is:

```xml
<?xml version="1.0" encoding="ISO-8859-1"?>
<xpn_conf>
  <partition name="expand1" type="NORMAL" bsize="512k"  >
    <data_node id="dn1" url="mpiServer://hostname_1/mnt/xpn"/>
    <data_node id="dn2" url="mpiServer://hostname_2/mnt/xpn"/>
  </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. The current prototype only allows the definition of one only partition.

Using this configuration file, the name used for a file in Expand is, for example:

`xpn://expand1/dirA/filename`

We use URI for naming. This file is distributed in the partition using two servers: `hostname_1` and `hostname_2`. In each server the subfile used for this Expand file is:

`/mnt/xpn/dirA/filename`

In the Expand configuration file, the type label identifies the type of file:

- `NORMAL`, which is a normal partition with cyclic layout;
- `FT`, which is a partition with fault tolerant support.

In the current prototype, only NORMAL partitions are supported.

2.3.5 Updates

The following list includes the updates and progress since D2.1 in October 2021 for ad hoc Expand version.

1. New version of the system call interception library.
2. New ad hoc Expand server developed.
3. New ad hoc Expand client developed.
4. New installation manual.

2.3.6 ADMIRE application compatibility

Expand, as GekkoFS, is not a kernel-based file system and requires a interposition library to intercept I/O system calls before they are sent to the kernel. Expand provides a POSIX-compliant for read and write operations. Nevertheless, some operations, such as the renaming of the same file, may cause problems when is performed in parallel by several clients. However, this is not considered a typical operation in an HPC environment.

2.4 Hercules IMSS (in-memory storage system)

In the ADMIRE project, we designed and implemented Hercules IMSS that can be considered a fully POSIX-compliant file system.

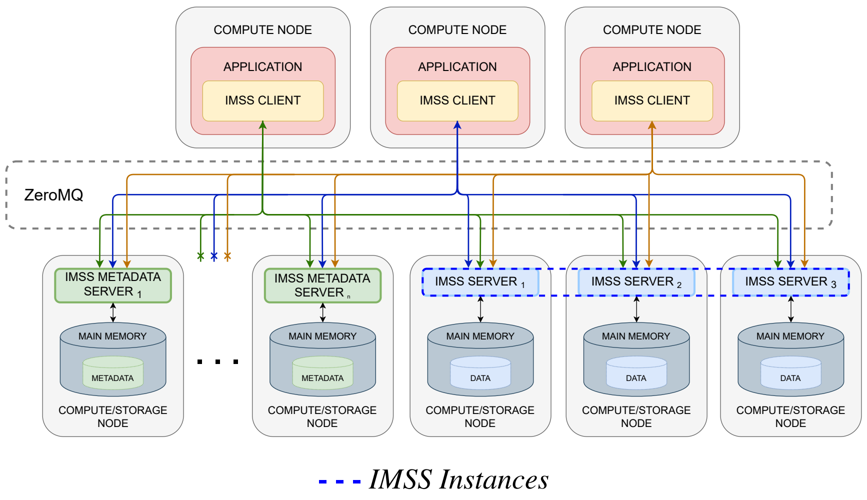


Figure 2.6: Representation of an IMSS deployment.

2.4.1 Design summary

As, shown in Figure 2.6, the architectural design of IMSS follows a client-server design model where the client itself will be responsible of the server entities deployment. We propose an application-attached deployment constrained to application's nodes and an application-detached considering offshore nodes.

The development of the present work was strictly conditioned by a set of well-defined objectives. Firstly, IMSS should provide flexibility in terms of deployment. To achieve this, the IMSS 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. Second, parallelism should be maximised. To achieve this, 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. Addressing the interaction between these components, the IMSS client will exclusively communicate with the IMSS metadata server whenever a metadata-related operation is performed, such as: *create_dataset* and *open_imss*. Data-related operations (*get_data* & *set_data*) will be handled directly by the corresponding storage server. Finally, IMSS offers to the application a set of distribution policies at dataset level increasing the application's awareness about the location of the data. 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.

Two of the most suitable network interfaces are sockets and Remote Procedure Calls (RPCs). To choose the best one, we made a comparison between several communication mechanisms sockets, gRPC, and we chose ZeroMQ [3] in order to handle communications between the different entities conforming an IMSS instance[5]. ZeroMQ has been qualified as one of the most efficient libraries for creating distributed applications [4]. ZeroMQ provides multiple communication patterns across various transport layers, such as inter-threaded, inter-process, TCP, UDP, and multicast. ZeroMQ provides a performance-friendly API with an asynchronous I/O model that promotes scalability. In addition, ZeroMQ library offers zero-copy messages, avoiding further overheads due to data displacements.

Furthermore, to deal with the IMSS dynamic nature, a distributed metadata server, resembling CEPH model [12], was included in the design step. The metadata server is 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.

[5](https://gitlab.arcos.inf.uc3m.es/mandres/imss/blob/master/Middleware_Comparison.pdf)

2.4.2 Use cases

Two strategies were considered so as to adapt the storage system to the application's requirements. On the one hand, the *application-detached* strategy, consisting of deploying 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 initialisation overheads in execution time. Figure 2.7 (left) illustrates the topology of an IMSS application-detached deployment over a set of compute and/or storage nodes where the IMSS instance does not belong to the application context nor its nodes.

On the other hand, the *application-attached* deployment 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 maximising locality possibilities for data. In this approach each process conforming the application will invoke a method initialising certain in-memory store resources preparing for future deployments. However, as the attached deployment executes in the applications machine, the amount of memory used by the storage system turns into a matter of concern. Considering that unexpectedly bigger memory buffers may harm the applications performance, we took the decision of letting the application determine the memory space that a set of servers (storage and metadata) executing in the same machine shall use through a parameter in the previous method. This decision was made because the final user is the only one conscious about the execution environment as well as the applications memory requirements. 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. Figure 2.7 (right) displays the topology of an IMSS application-attached deployment where the IMSS instance is contained within the application.

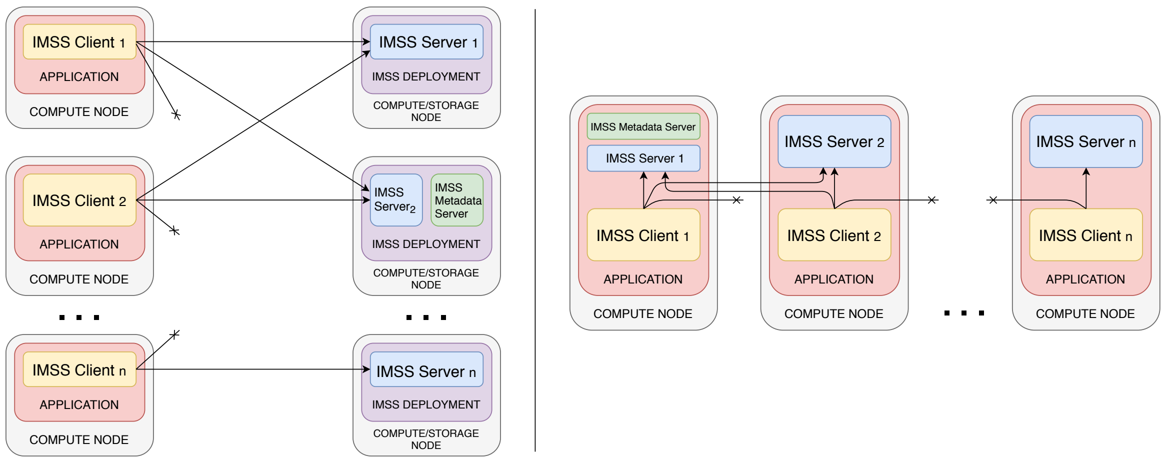


Figure 2.7: IMSS application-detached deployment (left side) vs IMSS application-attached deployment (right side).

2.4.3 Download and installation

Hercules IMSS repository is available at: `https://gitlab.arcos.inf.uc3m.es/admire/imss`. The repository contains the following folder structure. The following software packages are required for the compilation of Hercules IMSS:

- CMake

- ZeroMQ
- Glib
- tcmalloc
- FUSE
- MPI (MPICH or OpenMPI)

Hercules IMSS is a CMAKE-based project, so the compilation process is quite simple:

```
mkdir build
cd build
cmake ..
make
```

As a result the project generates the following outputs:

- `mount.imss`: run as daemons the necessary instances for Hercules IMSS. Later, it enables the usage of the interception library with execution persistency.
- `umount.imss`: umount the file system by killing the deployed processes.
- `libimss_posix.so`: dynamic library of intercepting I/O calls.
- `libimss_shared.so`: dynamic library of IMSS's API.
- `libimss_static.a`: static library of IMSS's API.
- `imfssfs`: application for mounting HERCULES IMSS at user space by using FUSE engine.

2.4.4 Usage

The current prototype of Hercules IMSS enables the access to the storage infrastructure in three different ways: API library, FUSE, and LD_PRELOAD by overriding symbols. In the following subsections, we describe the characteristics of each alternative.

2.4.4.1 API

Hercules IMSS is defectively accessible by using C-based API. This API includes the following calls:

- **hercules_init:** initializes the infrastructure required to deploy an IMSS attached server within the calling client. Besides, it deploys an attached metadata server if requested.
- **hercules_release:** releases infrastructure resources of an attached deployment.
- **stat_init:** creates a communication channel with every metadata server. Besides, the former method declares additional resources that will be required throughout the client session.
- **stat_release:** releases resources required to communicate with the metadata servers and additional session parameters.
- **init_imss:** deploys an IMSS detached instance or initializes an IMSS attached one.
- **open_imss:** joins to an existing IMSS instance enabling access to the stored datasets.
- **release_imss:** releases resources required to communicate with an IMSS instance as well as the every IMSS instance server if requested.

- **create_dataset:** creates a new dataset within a previously created or joined IMSS instance.
- **open_dataset:** subscribes to an existing dataset within a previously created or joined IMSS instance.
- **release_dataset:** frees resources required to read and write blocks of a certain dataset.
- **get_data:** retrieves a certain block of a previously created or opened dataset from one of the IMSS servers conforming the instance.
- **set_data:** stores a certain block of a previously created or opened dataset into a set of IMSS servers part of the same instance.
- **get_type:** given a certain URI, the former method returns if it corresponds to an IMSS instance or dataset.

The below code excerpt depicts an usage example of Hercules IMSS by using the proposed API. The example creates a determined number of datasets in-memory.

```
char metadata[]  = "./metadata";
char localhost[] = "localhost";
char imss_test[] = "imss://berries";
char hostfile[]  = "./hostfile";

//Hercules init -- Attached deploy
if (hercules_init(rank, 2048, 5555, 5569, 1024, metadata) == -1) exit(-1);

//Metadata server
if (stat_init(localhost, 5569, rank) == -1) exit(-1);

//Imss deploy
if (init_imss("imss://berries", hostfile, 1, 5555, 1024, ATTACHED, NULL) == -1)
↪  exit(-1);

//Dump data -- Remark: DATA MUST BE IN DYNAMIC MEMORY
for(int i = 0; i < NUM_DATASETS; ++i){
    int datasetd_;
    char dataset_uri[32];
    sprintf(dataset_uri, "imss://berries/%d", i);

    //Create dataset, 1 Block of 1 Kbyte
    if ((datasetd_ = create_dataset(dataset_uri, "RR", 1, 1024, NONE)) < 0) exit(-1);

    char * buffer = (char *) malloc(1024 * 1024 * sizeof(char));

    //Fill the buffer with \n
    memset((void*) buffer, 0, 1024 * 1024);

    //Copy the used data
    char const * testdata = "\na\nab\nabc\nabcd\nabcde\nabcdef\nabcdefg"
                            "\na\nbb\nccc\ndddd\neeeee\nffffff\nggggggg"
                            "\n1\n22\n\n333\n";
    memcpy(buffer, testdata, strlen(testdata));

    //Set the data in 2 Blocks
    int32_t data_sent = set_data(datasetd_, 0, (unsigned char*)buffer);
    release_dataset(datasetd_);
}
```

2.4.4.2 FUSE

We have constructed a file system layer on top of the previously described library. The in-memory file system currently supports both data and metadata operations, such as file permissions, ownership, folders, and namespaces. Files and folders are presented as datasets inside Hercules IMSS, conforming a hierarchical representation supported by URIs, which univocally identifies each dataset. Data is partitioned into multiple blocks, reserving the first block for storing metadata. We distinguish between inner metadata, which represents the classical POSIX-like metadata mainly represented by the *struct stat* representation and outer metadata, which depicts the metadata related to the data blocks location and the applied distribution policy. Inner metadata is stored as a data block on each data set. Outer metadata is maintained by a separate metadata server. Hercules IMSS supports attached and detached metadata servers.

Using the call above, we have to mount the Hercules IMSS file system using FUSE.

```
./imssfs -p 5555 -m 5569 -M ./metadata -h ./hostfile -b 1000000 -r imss:// -a
↪  ./stat_hostfile -S 10000000000 -d 0 -B 1048576000 -l /mnt/imss/
```

The mount call requires the following parameters:

- `-p`: determines the listening port number of the I/O servers.
- `-m`: indicates the port number of the external/internal metadata server.
- `-h`: requires a file containing the hostnames of all I/O server involved.
- `-b`: specifies the block size employed for all network data transfers used by ZeroMQ.
- `-r`: determines the default dataset root for the deployed file system.
- `-a`: requires a file containing the hostnames of all metadata servers involved.
- `-s`: the maximum capacity in bytes of the storage system.
- `-d`: determines the deployment mode. 0 indicates an attached deployment strategy (metadata and data services are instantiated as a FUSE process) and 1 indicates a completely detached deployment, is such a way, both data and metadata servers have to be executed as independent processes.
- `-l`: indicates the mount point path.

2.4.4.3 LD_PRELOAD

The project repository provides support for running Hercules IMSS overriding I/O calls by using the LD_PRELOAD environment variable. Both data and metadata calls are currently intercepted by the implemented dynamic library.

```
LD_PRELOAD=libimss_posix.so ls -l /mnt/imss/
```

2.4.5 Updates

The current version of Hercules IMSS have been tested and evaluated by using IO500 benchmark[6]. This benchmark executes both data and metadata micro-benchmarks in order to measure the efficiency of the target file system. Current prototype successfully passed the full version of IO500, which includes IOR and MDBENCH benchmarks. These benchmarks test and evaluate the performance by using different I/O calls of the POSIX interface.

```
IO500 version io500-sc21-scc_v1-14 (standard)
[RESULT]       ior-easy-write        0.124110 GiB/s : time 30.017 seconds
[RESULT]    mdtest-easy-write        0.660125 kIOPS : time 2.524 seconds
[      ]            timestamp        0.000000 kIOPS : time 0.000 seconds
[RESULT]       ior-hard-write        0.072691 GiB/s : time 30.007 seconds
[RESULT]    mdtest-hard-write        0.470201 kIOPS : time 31.063 seconds
[RESULT]                 find        3.032690 kIOPS : time 4.990 seconds
[RESULT]        ior-easy-read        0.120736 GiB/s : time 30.854 seconds
```

[6]https://github.com/IO500/io500.

```
[RESULT]     mdtest-easy-stat        2.714802 kIOPS : time 1.370 seconds
[RESULT]        ior-hard-read        0.121106 GiB/s : time 18.011 seconds
[RESULT]     mdtest-hard-stat        2.664673 kIOPS : time 6.305 seconds
[RESULT]   mdtest-easy-delete        0.505189 kIOPS : time 2.986 seconds
[RESULT]     mdtest-hard-read        1.402222 kIOPS : time 11.079 seconds
[RESULT]   mdtest-hard-delete        0.639612 kIOPS : time 23.109 seconds
[SCORE ] Bandwidth 0.107170 GiB/s : IOPS 1.151233 kiops : TOTAL 0.351251
```

2.4.6 ADMIRE application compatibility

Given the cooperation with the WP 7, Hercules IMSS fully accomplishes the requirements of ADMIRE's use cases. As discussed in the regular WP 2 meetings, Hercules IMSS is compatible with the I/O services required by all use cases. For this reason, we have extended the initial prototype to two novel file access systems (FUSE and I/O interception). Hercules IMSS enables multiple parameter configuration for adapting tits behaviour to different use case scenarios such as the internal block size, number of I/O nodes, the maximum memory capacity, etc.

3 Applications analysis

This section will discuss new insights in the on-going analysis of the HPC applications (here, the molecule simulation and turbulence simulation applications) in the ADMIRE project and how studying the demonstrated I/O pattern of these applications can lead to design better ad hoc storage system semantics. With this analysis, we aim to understand the application I/O behaviour in a realistic scientific environment setting and the real-world use cases. In this section, we will show the results by running the applications on the MOGON II [1] cluster at the Johannes Gutenberg University Mainz (JGU).

3.1 Molecule simulation

The system requires to set up the computation according to the input information in `cp.in`. The name list gives information about the number of position of the ions. In the *Quantum Espresso* application used for molecule simulation, the master process reads all the information as an input and broadcast the information to the all processors for computation. Each computations contains fourier transform and matrix multiplication operations. Computations are done in loop phases and after each phase some parameters are updated. There is a possibility to checkpoint after each loop to restart the simulation. In the end, the information is being aggregated and written to the storage using one process. Kindly refer to D7.1 for more general information on the Quantum Espresso application.

3.1.1 I/O behaviour

The behaviour of Quantum Espresso depends on the problem and its performance greatly depends on the physical cases. Fortran I/O and XML files are used to write data and log files. At the end of each phase of Car-parinello algorithm, an explicit synchronisation barrier is performed. Therefore, this strict MPI hierarchy composes the majority of the execution time. The I/O consists of read and write operations which are issued by the master processes initiating the data broadcast and synchronisation by the end of each loop. Therefore, it demonstrates a bursty I/O behaviour. However, most of the execution time is spent in MPI barriers. We ran the Quantum Espresso Car-parinello application and Covid protein data set. The simulation was ran using `MPI+OMP` and the input control parameters are the following:

```
&control
   calculation='cp',
   ...
   nstep=50,                # number of Car-Parrinello steps performed in this run
   iprint=1,                # band energies are written every iprint iterations
   isave=1,                 # number of steps between successive saving of information
   dt=1.0d0,                # time step for molecular dynamics.
   tstress = .true.         # write stress tensor to standard output each "iprint" steps.
   tprnfor = .true.         # automatically calculate forces
   disk_io = 'high',        # XML data file I/O size
   verbosity = 'high',      # verbosity of the output

```

[1] `https://hpc.uni-mainz.de`, Apr. 2022

We ran the simulation using 1200 and 2400 processor cores on 32 and 64 nodes, respectively. Each node is equipped with two 16-core Skylake processors (Xeon Gold 6130) and connected via OmniPath 100 gbps and consists of 192 GiB ram.

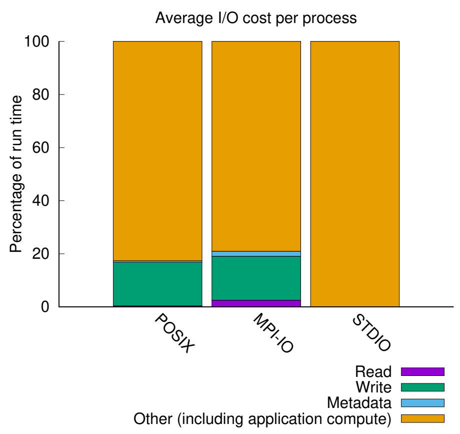
The runtime for the entire covid protein simulation with the above setting was 16 hours for the 32 node setup and 9 hours for the 64 core setup. The resulted outputs were written by four worker processors. The total execution time in our setup is bounded by memory and strict MPI hierarchy and barriers. The I/O intensity increases by activating checkpoints in the simulation. Many use cases of the molecular dynamic simulation can benefit from checkpoints for further data extraction and also to have the possibility of restarting the simulation with a different number of MPI processes. The size of the checkpoint files can be up to several terabytes and therefore these systems can benefit from faster storage systems, e.g., ad hoc storage systems, to decrease I/O overhead. Since the Quantum Espresso simulation performs intensive writes through master processes, these simulations can benefit from local writes on the file system as well.

3.2 Turbulence simulation

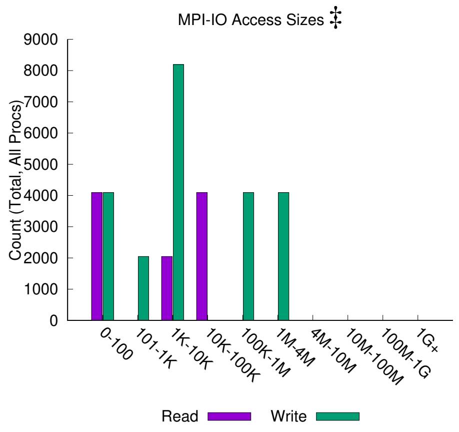
I/O behaviour in the turbulence simulation by running the *Nek5000* application depends on the simulation cases. We analysed the Nek5000 workload by running the provided example of *bentpipe* introduced in D7.1. We ran the example using a smaller setting (for more reasonable runtimes) and using the *Darshan*[2] I/O profiler. Figure 3.1 shows the Darshan profiling of the bentpipe simulation. As shown in Figures 3.1b and 3.1c, the I/O accesses consist of small sequential reads and writes to a binary file. Each rank writes on a different chunk of a shared binary file using MPI-IO.

I/O occurs at regular intervals and the divisions to when and how to write data is taken at the beginning of the simulation. These steps need to be chosen carefully as it can affect the performance, i.e., runtime, of the simulation. The I/O percentage of the execution time drastically increases as the sampling and checkpointing frequency increases. The higher sampling frequency can help with achieving a higher accuracy of the simulation. It also can be used for post-processing of the unprocessed data. The Nek5000 simulation can therefore benefit from a faster I/O system, e.g., via ad hoc storage systems, to store more sampling data points for later analyses. However, data size is also a decisive factor in increasing the checkpointing frequency. In addition to a faster storage, online lossless compression techniques can be useful to sample more data in the duration of the simulation run.

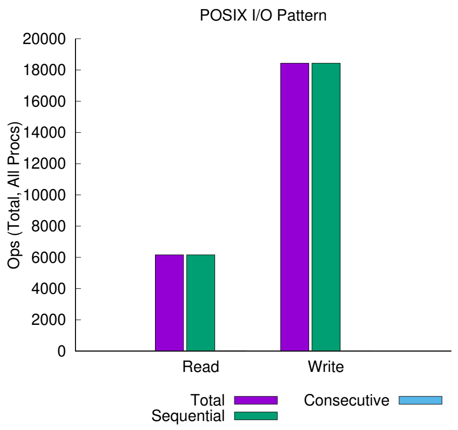
[2]https://www.mcs.anl.gov/research/projects/darshan



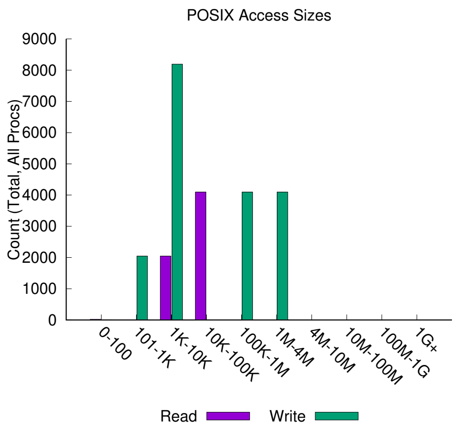
(a)



(b)



(c)



(d)

Figure 3.1: Darshan profiling of the bentpipe simulation

4 Conclusion

In this prototype deliverable, we have presented how to download, build, test, and run each of the four ad hoc storage systems that are considered in the ADMIRE project. We have therefore included detailed instructions to download, build and run these storage systems. We have further discussed the use cases and limitations of each ad hoc storage system concerning their compatibility with the ADMIRE applications, and we have described the software progress since the last deliverable. Finally, we have included new insights in the on-going application analysis concerning the application's I/O footprint.

Among others, the next steps will include learning from these insights and applying new techniques to the ad hoc storage systems to offer even higher I/O performance for HPC applications.

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.
- 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.
- Node-local Storage, ability for a compute server to store persistent data on physically local storage devices.
- 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.
- QoS, Quality of Service.
- RDMA, remote direct memory access.
- RPC, remote procedure call.
- Slurm, job submission system widely used.
- SSD, solid state drive.

Bibliography

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek Gordon Murray, Benoit Steiner, Paul A. Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: A system for large-scale machine learning. In Kimberly Keeton and Timothy Roscoe, editors, *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016*, pages 265–283. USENIX Association, 2016.

[2] 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.

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

[4] 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.

[5] 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.

[6] 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.

[7] 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).

[8] Alexander Sergeev and Mike Del Balso. Horovod: fast and easy distributed deep learning in tensorflow. *CoRR*, abs/1802.05799, 2018.

[9] 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.

[10] 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.

[11] Chen Wang, Kathryn Mohror, and Marc Snir. File system semantics requirements of HPC applications. In Erwin Laure, Stefano Markidis, Ana Lucia Verbanescu, and Jay F. Lofstead, editors, *HPDC '21: The*

30th International Symposium on High-Performance Parallel and Distributed Computing, Virtual Event, Sweden, June 21-25, 2021, pages 19–30. ACM, 2021.

[12] Sage A Weil, Scott A Brandt, Ethan L Miller, Darrell DE Long, and Carlos Maltzahn. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 307–320, 2006.