H2020-JTI-EuroHPC-2019-1

Project no. 956748

ADAPTIVE MULTI-TIER INTELLIGENT DATA MANAGER FOR EXASCALE

D6.2
Report on the intelligent controller design

Version 1.0

Date: May 11, 2022

Type: Deliverable
WP number: WP6

Editor: Alberto Cascajo
Institution: UC3M

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	15/01/22	Jesus Carretero	UC3M	Document creation.
2	26/04/22	Alberto Cascajo	UC3M	Introduction updated. New architectural blocks included in Figure 1.1. Figures 2.1 3.1 4.1 updated.
3	27/04/22	Alberto Cascajo	UC3M	Architecture updated.
4	28/04/22	David E. Singh	UC3M	Review changes and new paragraph in Section 3.2
5	29/04/22	Ramon Nou	BSC	Review I/O Scheduler Functions, review API
6	29/04/22	Alberto Cascajo	UC3M	Review Intelligent controller subsections and new functions and descriptions have been updated in Chapter 4
7	02/05/22	Clément Barthélemy	Inria	Reorder IC sections & add resource manager interface paragraph
8	03/05/22	Javier Garcia-Blas	UC3M	Revised the document.
9	03/05/22	Alberto Cascajo	UC3M	Diagrams 3.4 and 3.5 included. New comments in architecture and Intelligent Controller sections.
10	04/05/22	Marc-André Vef	JGU	Review of ad-hoc storage system API (ID12) and changes to the corresponding section. Various minor modifications.
11	05/05/22	Hamid Fard	TUDA	Reviewed and revised the document as the external reviewer.
12	06/05/22	Alberto Cascajo	UC3M	Revised the document. Some suggested changes were applied.
13	08/05/22	Alberto Cascajo	UC3M	Figure 1.1 updated and other minor changes due to this modification. Figure 4.2 updated.
14	09/05/22	Alberto Cascajo	UC3M	Diagrams 3.4 and 3.5 were updated. The new reviewer's suggestions were applied.
15	09/05/22	Jean-Baptiste Besnard	PARATOOLS	Reviewed and revised the document as the external reviewer: style and typographic fixes, and general commentary.
16	10/05/22	Alberto Cascajo	UC3M	Diagrams 3.4 and 3.5 were updated. The new reviewer's suggestions were applied.

Executive Summary

In ADMIRE, WP6 defines the Intelligent Controller (IC) architecture, which is in charge of coordinating the rest of the components to optimize the performance of the applications and, therefore, the system throughput. he heart of ADMIRE is the Intelligent Controller that allows joining cross-layer information from sys- tems, applications and users to optimise the throughput of the system and the performance of the applications. The Intelligent Controller communicates with other system components (such as the monitor, applications, job scheduler, etc.) through control points defined for each. It is in charge of coordinating their tasks. In this way, the IC manages the system using a dynamic "control plane", composed of the Monitoring Manager, applications, Slurm job scheduler, malleability management, I/O scheduler, and accessing the information provided by the "data plane" through predefined control points and interfaces (the data plane is composed by the Ad-hoc Storage Systems and the Backend Storage System). An API is being designed to provide access to the control plane following the requirements of the ADMIRE components. The control plane will then steer the main active components.

This deliverable includes the design of the Intelligent Controller showing control and data plane architectural blocks for orchestrating system components, the final version of the definition of the Application Programming Interface (API) with the other of the Intelligent Controller.

Contents

Chapter 1

Introduction

In ADMIRE, WP6 defines the Intelligent Controller (IC) architecture, which is in charge of coordinating the rest of the components to optimize the performance of the applications and, therefore, the system throughput. To develop the concept proposed and to achieve the objectives, the project started with the design and the implementation of the open storage management framework, shown in Figure 1.1, consisting of several active modules working in cooperation with the architecture.

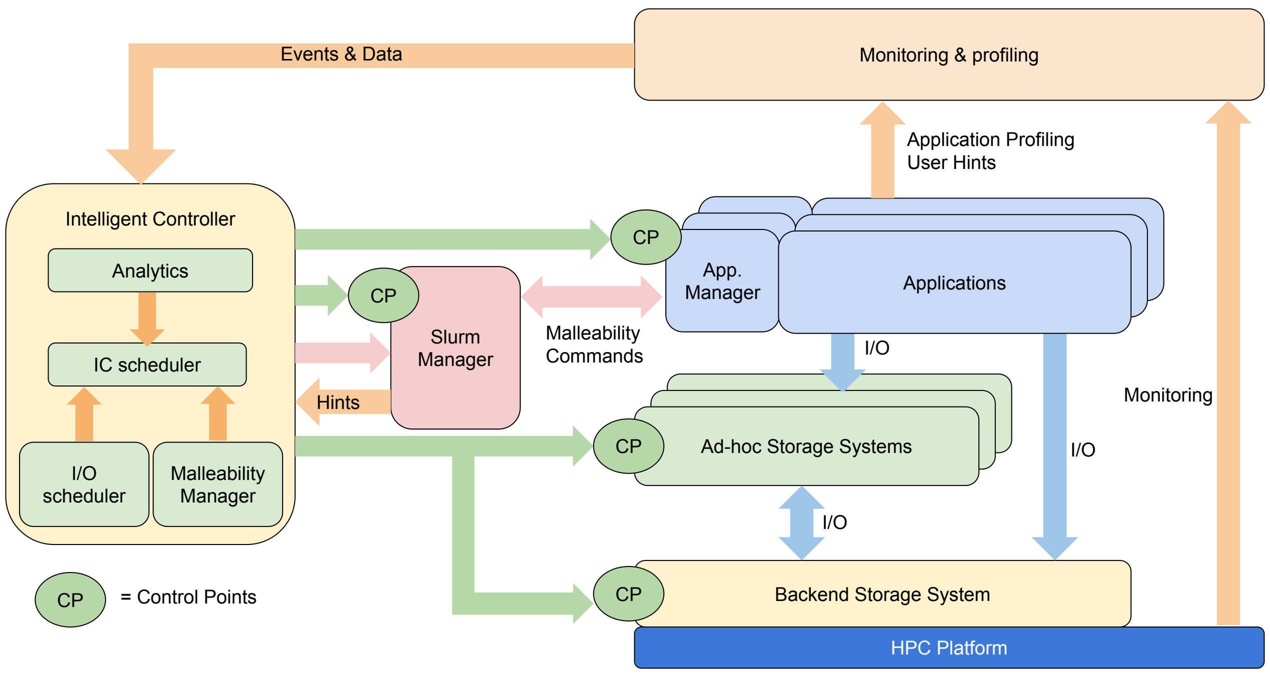


Figure 1.1: ADMIRE architectural blocks.

We departed from our work in CLARISSE [2] and FlexMPI [3] to combine runtime data analytics, malleability, and I/O control mechanisms to enhance the execution of multiple applications. All these features allow the Intelligent Controller to have a more detailed view of the applications and the system, improve the scheduling decisions, reconfigure the processes in runtime, and detect I/O contention between applications.

The heart of ADMIRE is the Intelligent Controller that allows joining cross-layer information from systems, applications and users to optimise the throughput of the system and the performance of the applications. The Intelligent Controller communicates with other system components (such as the monitor, applications, job scheduler, etc.) through control points defined for each. It is in charge of coordinating their tasks. In this way, the IC manages the system using a dynamic "control plane", composed of the Monitoring Manager, applications, Slurm job scheduler, malleability management, I/O scheduler, and accessing the information provided by the "data plane" through predefined control points and interfaces (the data plane is composed by the Ad-hoc

Storage Systems and the Backend Storage System). An API is being designed to provide access to the control plane following the requirements of the ADMIRE components. The control plane will then steer the main active components.

The Intelligent Controller inter-operates with the monitor, the Slurm Manager, the Application Manager, and the data plane through the control points defined for each one specifically. The main idea is to create a distributed control infrastructure that provides two features transparently: (1) a single system image through a distributed consistency protocol and (2) a set of facilities for the user. The second feature includes facilities such as novel runtime analytics tools to tune the I/O system and applications behavior (using control points of multi-criteria distributed control algorithms).

Applications and the HPC Platform provide information to the Intelligent Controller through a monitoring and profiling module. The Intelligent Controller offers the necessary global system services for developing the active components in this project: the Analytic Component, the Malleability Manager, the I/O scheduler, and the IC Scheduler (which is in coordination with the other components). This way, it integrates and analyses cross-layer system data to dynamically and intelligently steer the system components. The goal is to optimize at the system-scale the data management and the I/O access of the running applications based on the input provided by the ecosystem (WP5) and to enforce policies (e.g., I/O scheduling in WP4) through malleability management (WP3) and I/O management (WP2). To make decisions, the IC scheduler will rely on machine learning techniques to predict resource usage and application behavior.

This deliverable includes information about the developed features in the context of the Intelligent Controller and other foundations for the next steps. Chapter 2 describes the control and data plane architectural blocks for orchestrating system components. Chapter 3 includes the definition of the integrated components, communication protocols, and workflows of the Intelligent Controller. Chapter 4 defines the Application Programming Interface (API) of the Intelligent Controller with the different ADMIRE components. Chapter 5 shows the closing of the deliverable.

Chapter 2

ADMIRE Architecture

Figure 2.1 illustrates an overview of the ADMIRE architecture and shows how the information, data, and controls, are exchanged between the components. The storage subsystem is represented in the upper part of the figure and consists of the Ad-hoc and Backend Storage Systems. The former one is designed in the ADMIRE's WP2 and is responsible for providing to each application an ad-hoc high-performance storage system tailored to the application's characteristics. The latter one (Backend Storage) represents the parallel file system used by the HPC platform (e.g., Lustre). Both storage tiers are coordinated by the I/O scheduler (shown in the upper-left corner of the figure), which is responsible for the deployment and configuration of the Ad-hoc Storage System, the specification of Quality-of-Service metrics, and the implementation of I/O scheduling policies. The applications that are being executed in the platform are shown in the central part of Figure 2.1. ADMIRE-enabled applications can provide user-defined application-specific information to the system to aid the identification of I/O patterns and reconfiguration-safe states in which malleability commands can be executed.

Both applications and storage systems are monitored by the Sensing and Profiling component (central-right part of the figure), developed in WP5. This component is responsible for collecting system-wide performance metrics at the compute node level that will be stored in an internal database. This component is responsible for generating the performance model of each application, which will be used to support the Malleability Manager and Intelligent Controller in finding the most appropriate scheduling policy. The Monitoring Manager (lower-right part of the figure) leverages the information stored in the database to generate performance models related to (1) each running application, (2) all storage systems (both ad-hoc and backend), and (3) the compute nodes. The Malleability Manager (lower-left corner of the figure) determines and suggests malleable actions related to each running application and ad-hoc storage system. These actions may produce reconfiguration of processes/threads of a specific application or the deployment/removal of one or more instances of the Ad-hoc Storage System to better balance the computation and I/O requirements. The Intelligent Controller (central-left 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. It includes combined information about the hardware status, the existing running applications, and the storage. This information will be kept in an internal distributed database (Redis) that is described in section 3.2. 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 used to design a schedule solution, which will also consider the suggestions provided by the Malleability Manager and the I/O scheduler. The other main role of the Intelligent Controller is to coordinate the actions taken by other ADMIRE components. Examples of these actions are (1) to activate/deactivate each application monitoring and (2) to send the malleable decisions to the I/O scheduler or the applications in case of I/O-related or application-related decisions, respectively.

2.1 WP6 focus

The Intelligent Controller (IC) is a multi-criteria distributed component that will integrate cross-layer data to have a holistic view of the system. The IC will also make intelligent analyses to adapt dynamically the system to current and future workloads to increase its throughput. It will enable the global orchestration of the exascale

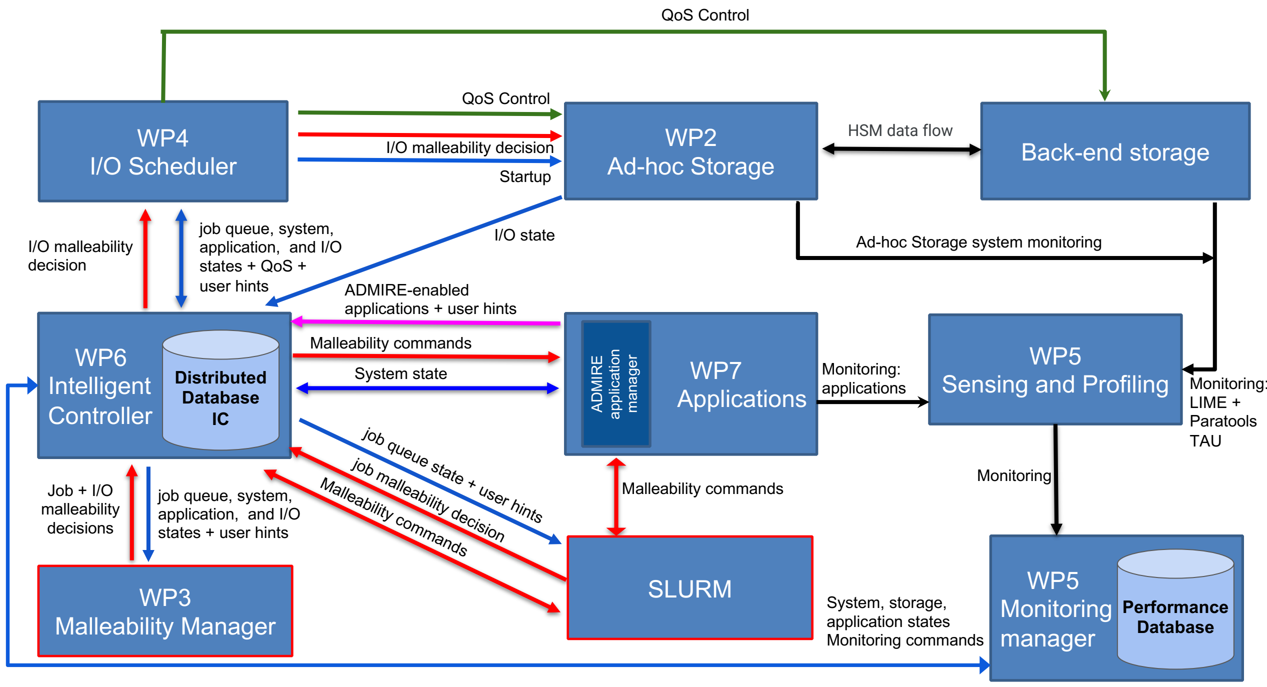

QoS Control
WP4
I/O Scheduler
QoS Control
I/O malleability decision
Startup
WP2
Ad-hoc Storage
HSM data flow
Back-end storage
I/O malleability decision
job queue, system, application, and I/O states + QoS + user hints
I/O state
Ad-hoc Storage system monitoring
ADMIRE-enabled applications + user hints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

Figure 2.1: ADMIRE architecture overview. Each component developed in the project's scope has included the label of its related Work Package (WP).

system components through an intelligent dynamic control plane. In this way, the IC interoperates with the Monitoring Manager, the Applications, the Slurm Manager, the Malleability Manager, the I/O scheduler, and the data plane through predefined control points and interfaces. The roles of the IC are:

- **Analytics.** The analytic component will improve I/O system behavior and facilitate anticipatory decisions for resource allocation. This will be achieved based on the knowledge collected from the I/O systems, applications, and the batch system. The IC will collect information provided by the monitoring system (WP5), which will be analyzed using novel data-centric machine learning techniques to predict application and system behavior. It also obtains additional information from the historical job records of previous runs. In this context, the analytics component of the IC will holistically integrate and analyze cross-layer system data to dynamically and intelligently steer the system to tune I/O performance at the system scale. The strategy of this component consists of training model classifiers that can handle jobs with heterogeneous I/O patterns, detect congestion, perform resource provisioning, and perform anomaly detection that will help the IC to identify and predict potential risks and failures of applications.

- **Malleability management support.** The Malleability Manager (MM) component provides optimal application scheduling solutions for the IC. The IC leverages those suggestions to improve its schedule solution and reconfigure the system in run-time. The MM solutions will maximize system throughput by adjusting both sets of computational and I/O resources assigned to a job. The MM solutions are based on applying I/O and application malleability policies to the models and predictions (previously given from the IC), resulting in an optimal scheduling solution for the current applications and resources.

- **I/O scheduling support.** The IC will provide decentralized decision-making mechanisms to support I/O scheduling based on information collected from the monitor and the applications profiler, together with the IC's analytic tools. This will support the development of global policies for balancing compute and I/O performance. In particular, it will support the design and implementation of end-to-end QoS policies through the I/O scheduler. In addition, the IC will support handling critical stages such as the co-scheduling of many I/O intensive jobs that could cause I/O contention and under-utilization of other resources, ultimately degrading performance. The idea is to combine the analytic component prediction models with decision-making mechanisms.For scalability reasons, it will leverage the I/O scheduling algorithm to provide a distributed version dealing with partial knowledge, failures, and asynchronicity.

- **IC scheduler.** This component leverages and combines (1) the performance models generated by the Analytic component, (2) the malleability suggestions provided by the Malleability Manager, and (3) the I/O scheduling solutions obtained from the I/O scheduler component. The IC will generate a new scheduling solution using multi-criteria objective trade-offs such as response-time vs throughput vs energy consumption (utilizing machine learning algorithms) and considering the current state of the platform, the processes, and their near future state. The multi-criteria goals will be provided either by the system administrator or by the job when submitted.

- **Application support.** The IC provides facilities to achieve an intelligent runtime tuning of parallel applications with control points. The following services will be provided through control points to the developers of control algorithms: dynamically enable/disable control domains by activating/deactivating control agents; provide an open API for the implementation of new control algorithms; call the monitoring and profiling service to find out system and applications I/O state information; allowing insertion of processing rules on incoming data such as filters for supporting dynamic in-situ/in-transit processing. Each active component will expose a series of parameters that are to be configured in a control point (e.g., the number of servers to stripe a file, the block size, etc.). The algorithm in each control point will perform a series of actions targeted to drive the system towards the desired goal. These actions include: finding out the system state based on the monitoring service, inserting processing rules on incoming data such as filters, communicating with control agents, and taking reconfiguration decisions.

- **Programming support.** The IC will include the design of a coordination language for describing I/O interactions among different parallel jobs composing a data-driven workflow. The coordination language will be used by the application programmer(s) to integrate cross-application data and to provide an aggregated view of the whole I/O system. The objective is twofold. On the one hand, to simplify the definition of different work data spaces and their interactions, and to gather useful information from the applications developers to feed the intelligent scheduler to make more informed decisions. On the other hand, we aim to automatize the generation of low-level scripts/programs (e.g., Slurm scripts) used to run the different jobs composing the application workflow. Therefore, to be scalable, the control points will be supported by a hierarchical architecture that will be in charge of collecting information and performing data transfers over the HPC interconnection fabric – effectively providing separate control and data planes.

- **Resiliency support.** ADMIRE will replicate data for resilience. IC will support storage resiliency from a malleability perspective. For example, failure of a storage node implies that the degree of redundancy temporarily decreases. By restoring the node, that is, its copy of the data, we increase it again after failure. Thus, this replicated data will also be taken into account by the I/O Scheduler to improve performance and guarantee QoS.

- **Scalable support.** It will be composed of a set of distributed agents for resilience and to provide scalability for large-scale HPC infrastructures. For performance reasons, the IC will run concurrently and interact through asynchronous communication.

- **Communication API support.** To develop these mechanisms, the intelligent controller will be accessed through a common API that will offer mechanisms to reserve bandwidth on the level of assigned remote procedure calls (RPCs) and which will also offer the possibility to link compute tasks to data transfers. The intelligent controller will forward QoS commands to the Backend Storage System and the network switches. Although it is conceptually possible to send commands on an arbitrarily fine-grained level, the anticipated control granularity can be, e.g., on the level of users, applications, files, network ids, or combinations of them.

Chapter 3

Intelligent controller

The Intelligent Controller (IC) is a multi-criteria distributed controller that integrates cross-layer data to provide a holistic view of the system for making intelligent analyses to adapt the system to current and future workloads dynamically. This chapter includes the design of the IC, a description of the selected distributed database (*Redis*), a description of the communication library (*libicc*) used by all of the components, and the IC workflow (explained through algorithms and flow diagrams).

3.1 Intelligent Controller design

Figure 3.1 shows the structure of the Intelligent Controller (IC) communication. It is organized as a hierarchical architecture in which a central controller (called *Root controller*) is responsible for creating and managing the rest of the IC components. Note that all these components run concurrently because each is executed in a separated thread. The IC stores its state in a Redis *distributed database* (see section 3.2).

The Intelligent Controller communication library *libicc*, provides all applications and the ADMIRE components with the required interfaces to interact with the IC. The communication can be bi-directional, meaning for example that an application can both contact and be contacted by the IC (see section 3.3).

The *System Analytic component* will apply machine-learning techniques to model and predict the performance of the applications. To generate those models and predictions, the algorithms will combine information from three different sources: (1) the performance metrics provided by the *Monitoring Manager*, (2) the application hints that come from the applications, and (3) the stored data in the Redis database. Once the *Analytic component* has generated its output, it will also be stored in the Redis database. For each running application in the system (shown in the figure 3.1 as boxes in purple color), the root controller will execute an *Application Analytic Component*. This component is responsible for collecting information about the application execution[1] and storing it in the Redis database.

3.2 Distributed database: Redis

NoSQL databases were implemented in reaction to the perceived rigidity of the relational data schema for applications with specialized needs, leading to various data models: key-value, document, graph, etc. The other impetus was a will to trade consistency and isolation for increased availability and horizontal scalability (i.e., performance scale when adding more machines). Thus most NoSQL databases have built-in support for replication and partition. However, they do not usually offer fully ACID transactions and restrict themselves to a weaker form of consistency, such as *eventual consistency*.

Redis is a key-value in-memory store implemented in the C language that offers storage support for multiple data structures. The concerned technology runs over a single thread in charge of the I/O communications and the data storage/retrieval operations. Besides, it implements two persistent possibilities: snapshotting and append-only logging. In terms of record indexing, Redis uses a dynamic hash-table to manage all key-value

[1]This information will include the application status, user hints provided through ADMIRE user interface and other application characteristics but it will not include the application performance metrics, which are collected by the *Sensing and Profiling component*

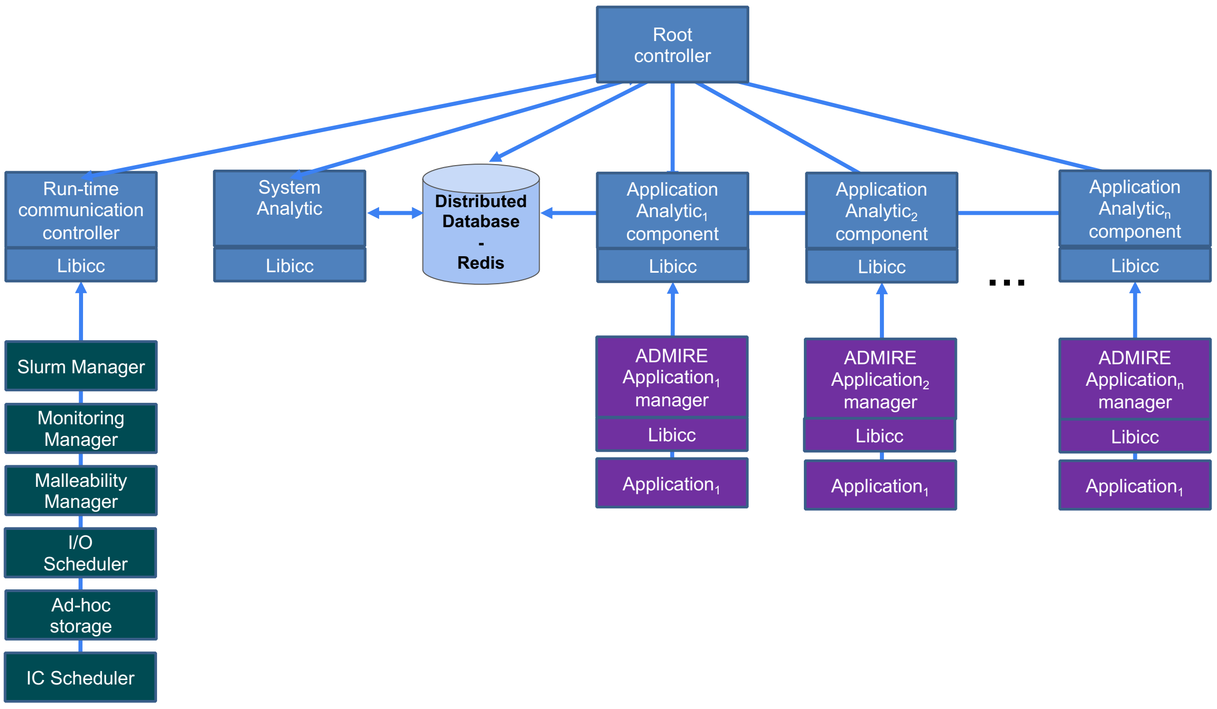

Root controller
Run-time communication controller
Libicc
System Analytic
Libicc
Distributed Database - Redis
Application Analytic$_1$ component
Libicc
Application Analytic$_2$ component
Libicc
...
Application Analytic$_n$ component
Libicc
Slurm Manager
Monitoring Manager
Malleability Manager
I/O Scheduler
Ad-hoc storage
IC Scheduler
ADMIRE Application$_1$ manager
Libicc
Application$_1$
ADMIRE Application$_2$ manager
Libicc
Application$_1$
ADMIRE Application$_n$ manager
Libicc
Application$_1$

Figure 3.1: Structure of the ADMIRE Intelligent Controller. The components of the Intelligent Controller are shown in blue color. Other ADMIRE's components are displayed in green color. Applications are displayed in purple color. Note that this architecture will be replicated in several compute nodes in order to provide resilience capabilities.

objects that grow as it is filled up through a rehashing strategy. Redis also provides a distributed version called Redis Cluster, which lacks any kind of centralized server achieving an absolute decentralization. Nevertheless, consensus must be reached between them to ensure a successful execution, which leads to significant overheads in the case of node arrival and departure. The Redis distributed version implements a hash slot partition strategy to find out which server within the deployment will deal with a certain record (a subset of all hash slots is assigned to each server).

Moreover, Redis is less a database than an in-memory data structure store. It has been used extensively as a shared cache and queue. Redis does offer high availability through passive (primary-secondary) replication.

Finally, the IC uses the database to store its state and the results of the generated application models, and we consider that Redis fulfills the previously defined requirements. Some of them are scalability and reliability, support for efficient varied queries, and a C application programming interface (API). The API used for connecting the framework and the Redis database is Hiredis. It is a minimalist C client library that adds support for the protocol, and it uses a high-level printf-alike API. Moreover, apart from supporting sending commands and receiving replies, it comes with a reply parser that is decoupled from the I/O layer. It can be used in higher-level language bindings for efficient reply parsing.

3.3 The libicc Intelligent Controller communication library

All communications between the IC, ADMIRE components, and applications are implemented in *libicc* with remote procedure calls (RPCs), a paradigm that local calls are executed on remote resources using Mercury [6] and Margo [4] frameworks, both well in use in the HPC community. Mercury is an RPC library specifically designed for high-performance computing (HPC) systems. Margo is a newer C library built on top of Mercury and the Argobots [5] user-level threading framework. Both Mercury and Margo are actively developed under the umbrella of the Mochi project, a collaboration between Argonne National Laboratory, Los Alamos National Laboratory, Carnegie Mellon University, and the HDF Group. These libraries have been chosen for the AD-

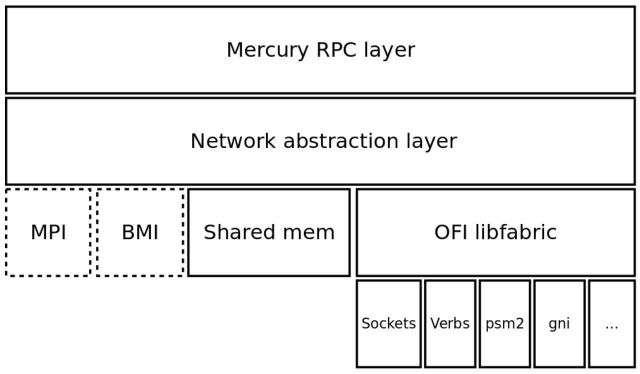


Figure 3.2: Mercury architecture. The RPC layer relies on the network abstraction layer and its plugins for transport. The main plugins are OFI libfabric and shared memory, BMI and MPI are legacy plugins. Adapted from [6].

MIRE project because they couple a low overhead with portability and abundant documentation. Both of them are also available under a free license.

3.3.1 Mercury

The Mercury library differs from other RPC frameworks by (1) not relying exclusively on the TCP/IP protocol and (2) allowing the efficient transfer of a large amount of data. This is achieved by using the remote direct memory access (RDMA) on HPC networking hardware that supports it.

From the bottom-up (see figure 3.2), Mercury consists of a *network abstraction layer* that abstracts lower-level network plugins such as shared memory or the OpenFabrics Interfaces (OFI) provided by libfabric [1]. Libfabric is a user-level networking library made of two parts: a hardware-independent client API against which applications program, and several hardware-specific *fabric providers*, such as Verbs, psm2, or gni. As of v1.0.0, Mercury uses libfabric as its main network plugin. Other legacy plugins include BMI and MPI.

The *RPC layer* offers serialisation and deserialisation of input and output parameters to the remote function calls into a buffer, and transport it to and from the remote target using the network abstraction layer. In addition, a *bulk layer* is dedicated specifically to the transfer of large amounts of data, using RDMA if possible to avoid costly copies. Applications are expected to restrict themselves to the RPC and bulk layers.

Mercury uses a callback style of programming. Callbacks are passed to non-blocking functions and queued in a 'completion queue' once the network operation completes. Network progress and callback execution has to be made explicitly via the `HG_Progress` and `HG_Trigger` functions, as described in figure 3.3.

To perform an RPC both the client ('origin' in Mercury parlance) and the server (or 'target') must:

1. initialize the Mercury interface. The first parameter is the network plugin and the second one indicate whether or not Mercury should listen for incoming connections (i.e. act as a server).

   ```
   hg_class_t *class = HG_Init("ofi+psm2", HG_TRUE);
   ```

2. Create a context of execution associated internally with the completion queue.

   ```
   hg_context_t *context = HG_Context_create(class);
   ```

3. Register the RPCs using an RPC name and argument structs. On the server, an associated callback function must be registered (here named `sum`) to be executed when and RPC request with this name arrives.

   ```
   hg_id_t rpc_id = MERCURY_REGISTER(class, "sum", sum_in_t, sum_out_t, sum);
   ```

4. The client will then need to get the network address of the server via an out-of-band method, such as a file on a shared filesystem, and create the RPC, this will return an RPC abstract handle.

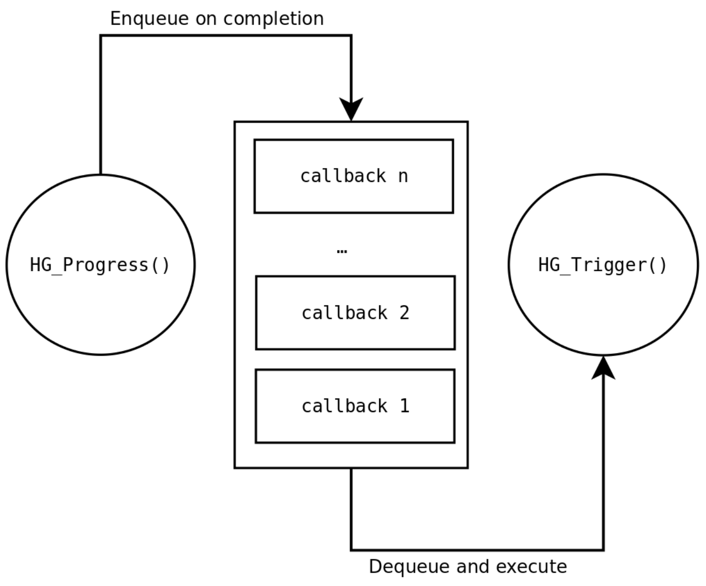


Figure 3.3: Mercury progress model. Both client and server have a completion queue of pending callbacks. `HG_Progress` makes explicit progress on network activity and on completion enqueues the corresponding callback. `Hg_Trigger` dequeues callbacks and executes them. Adapted from [6].

```
HG_Create(context, addr, rpc_id, &handle)
```

5. Finally, the client can forward the RPC to the server using the handle that was just initialized. The client also registers a callback, to be executed when the RPC response comes back from the server.

```
HG_Forward(handle, client_callback, NULL, &sum_in)
```

Progress at this point requires explicit intervention from the programmer. Both the client and the server must enter a progress loop calling alternatively `HG_Progress` and `HG_Trigger`:

```
do {
  unsigned int actual_count = 0;
  do {
    rc = HG_Trigger(hg_context, 0, 1, &actual_count);
  } while ((rc == HG_SUCCESS) && actual_count);

  rc = HG_Progress(hg_context, TIMEOUT_MS);
} while (rc == HG_SUCCESS);
```

3.3.2 Margo

Margo is based on Mercury and the Argobots [5] user-level threading library. Its goal is to wrap Mercury callbacks using Argobots user-level threads (ULTs). Internally, ULTs are used to invoke Mercury's non-blocking functions, suspended while network operations are in progress, thus saving resources, and resumed when the operations are complete. This means the programmer does not have to handle the Mercury progress loop. Margo also offers a way to spawn new ULTs to handle each RPC callback function.

Margo is used in a manner very similar to Mercury (see section 3.3.1), as shown partially below for the client code needed to create and forward an RPC. Importantly, no explicit progress loop needs to be written.

```
/* 1. initialize the Margo instance */
margo_instance_id mid = margo_init("ofi+sockets", MARGO_CLIENT_MODE, 0, 0);
```

```
/* 2. register the RPC and its argument structs */
hg_id_t rpc_id = MARGO_REGISTER(mid, "sum", sum_in_t, sum_out_t, NULL);

/* 3. create and forward the RPC */
margo_create(mid, addr, sum_rpc_id, &handle);
margo_forward(handle, &args);
```

3.4 ADMIRE Application Manager

The ADMIRE Application Manager is the interface between the applications and the IC. It runs in libicc with which applications are linked (see figure 3.1). To reduce overhead, interaction with the IC is generally restricted to the application root process. Because libicc uses Mercury and Margo, communications are handled asynchronously. The main roles of the Application Manager are:

- Receiving and using the user hints about the application's I/O behavior. These hints include information regarding application characteristics, specifically for what concerns I/O. Two classes of hints are considered. The first one contains static information about I/O behavior. This is provided via a kind of coordination language for I/O developed in the project, which describes coarse-grained I/O patterns in the application. Instead, the second class contains dynamic information provided directly by application developers during the execution of the application to the IC. These hints are more fine-grained, describing possible I/O patterns of a specific phase of the application flow.
- Sending the malleability commands received from the IC to the application.
- Receiving the application execution status (for instance, if a given malleable command has been successfully executed, or if the application is not in a reconfiguration-safe state).
- Activating/Deactivating different levels of monitoring granularity for the application based on control monitoring commands coming from the IC.
- Sending information upon request to the application about the platform status to guide application-dependent optimizations.

Interface to the resource manager Malleability command sent by the IC may require an application to expand or shrink its resource allocation, requiring an interaction with the resource manager. The ADMIRE project uses the Slurm resource manager which provides ways to resize jobs in its API. Their use is not straightforward, but they have the merit of existing and mean that the project can rely on an unmodified version of Slurm. As said, libicc handles communications asynchronously in the root process of the application, meaning allocation shrinking and expansion can be done independently from the main computation in the application. But at some point, the application needs to be able to restrict its operation (in case of shrinking) or use the new resources (in case of expansion). This is handled in libicc by requesting the registration of a reconfiguration callback at initialization time, called by the library after a change in the underlying resources. In the interest of generality, the definition of the callback is left to the application developers. One way to handle the dynamic allocation of resources in an MPI application would be to use the standard `MPI_Comm_spawn` function, designed to spawn new processes at runtime.

3.5 Intelligent controller workflow

There are two different options to provide malleability in the ADMIRE project. The first one consists of communicating the Intelligent Controller with the Application Manager and letting the Application Manager and Slurm coordinate to (1) allocate the new resources and (2) perform the reconfiguration. Algorithm 1 represents this workflow – called Option 1. In the second option, the Intelligent Controller is in charge of (1) communicating with Slurm for the resource allocation and (2) communicating with the Application Manager to perform the malleability over the new resources. Algorithm 2 represents the second workflow – called Option 2. The ADMIRE project offers these two options because the suitability of each one depends on the application characteristics. Note that red arrows represent optional data transfers, and blue arrows (in Algorithm 2) represent differences with Algorithm 1 in the workflow.

Algorithm 1 describes the different lines and components involved in the execution of an application within ADMIRE framework and option 1. The following listing provides a more detailed description of these steps.

- In line 1 the user that plays the role of programmer or application owner optionally inserts ADMIRE hints into the application source code using the ADMIRE API. The application is compiled, and its related executable is created.
- In line 2 the user that is responsible for the job execution submits the job to Slurm. Additional application information can be optionally provided to Slurm via the Slurm CLI APIs described in D4.1.
- In line 3 Slurm sends the static user hints captured during the job submission to the IC.
- In line 4 the Intelligent Controller sends the new application to the Malleability Manager.
- In line 5 the Intelligent Controller generates its initial schedule solution based on the data provided by the Analytic component, the I/O co-scheduler, and the Malleability Manager.
- In line 6 the Malleability Manager provides an initial schedule solution based on the application characteristics.
- In line 7 the IC evaluates if the schedule solution provided by the Malleability Manager could produce better performance than its own generated solution.
- The IC creates in line 8 two new instances related to the application: the Application Analytic Component and the Application Communication Controller (see Figure 3.1 for more details).
- In line 9, the ADMIRE Application Manager associated with the new executed application creates a communication channel with the IC. Note that there is a single channel per application regardless of the application size (application number of processes),
- In line 10 the ADMIRE Application Manager sends the user hints and the current application status to the IC. This information is stored in the System Status Table managed by the IC.
- In line 11 the Intelligent Controller sends to the Application Manager the commands needed to execute the application.
- In line 12 the Application Manager sends the allocation commands to Slurm, waiting for the list of available resources.
- After a certain amount of time (during which the job is queued), in lines 13 to 17, Slurm allocates the resources requested by the user and starts executing the job. Before starting the job's execution, however, the I/O Scheduler uses the APIs described in D2.1 to initialize an ad-hoc storage system instance if requested during the job submission. Once this instance is running, it starts to asynchronously transfer any input datasets needed to start the job, making sure that any QoS constraints for the application are maintained.
- In line 18 the Intelligent Controller (IC) receives (via Slurm) the job execution notification and the hints that the user has provided to Slurm. Based on this information, the IC updates the System Status Table[2] located in the IC's database.

[2]The System Status Table contains a list of all the executing applications and resources (compute nodes, I/O nodes) associated with each running application.

- In line 19 the IC communicates with the ADMIRE Application Manager to activate the application monitoring.
- During the application execution, the Sensing and Profiling component (line 21) collects the application performance metrics; other metrics from the ad-hoc storage system related to the application; back-end storage performance metrics. These metrics are stored in the Monitoring Manager database.
- In line 22 the Monitoring Manager calculates and stores different performance indicators related to the running application. These indicators include information about the application characteristics like I/O bandwidth, CPU/memory utilization, etc. These indicators are also stored in the Monitoring Manager database and are accessible to the IC.
- In line 23 the IC creates and updates an application performance model using machine learning algorithms that use as inputs the following data sources: the application performance indicators (provided by the Monitoring Manager), the user hints (provided by Slurm and the user interface of ADMIRE applications), historical records of previous executions of the same application (stored in the IC's database), the application and I/O performance metrics (provided by the Monitoring Manager) and the I/O storage status (provided by the ad-hoc storage system). This model is stored in the Application Status Table located in the IC's database.
- In line 24 the IC elaborates a system-wide performance forecast considering the models of all running applications and combining this information with other system-wide performance metrics collected by the Sensing and Profile module. This forecast will include potential contention situations that may occur in the future as well as the prediction of future system performance indicators.
- In line 25 all the previous information (forecasts, performance models, and System Status Table) are sent to the Malleability Manager and the I/O scheduler to support the decision-taking process of these modules. The Monitoring Manager also receives this information for guiding the deployment and configuration of the monitoring components.
- As in the initial section, between lines 26 and 30 the Intelligent Controller and the Malleability Manager, may generate a reconfiguration commands addressed to the application or its related ad-hoc storage system. Note that these commands involve the creation or destruction of application processes for the former case or storage nodes for the latter one. The Malleability Manager suggests its solution, but the IC may optionally override these actions after taking into account other factors not considered by the Malleability Manager.
- Once Slurm reallocates the requested resources (line 31), the workflow depends on the application and if it uses an external framework for malleability. If the application uses an external framework (e.g., FlexMPI), the Application Manager will send it the malleability commands, and it will spawn/shrink the processes. Otherwise, the IC will manage the reconfiguration with Slurm. This behaviour is defined between lines 33 and 38.
- In line 40 the IC provides different kinds of information to the running application. This information includes the platform status -which can be used by the application to guide local optimization techniques- and processing rules and commands that will permit execution filters and in-situ/in-transit processing algorithms on the application side.
- In line 41 the I/O Scheduler may optionally transfer datasets asynchronously to persistent storage if requested by the application or the ad-hoc storage system instance. Again, QoS constraints for the application are enforced.
- When the application terminates (line 43), Slurm sends a notification to the IC.
- In line 44, a notification of the application termination is sent to the Malleability Manager, the I/O scheduler, and the Monitoring Manager.

- In line 45 the IC notifies the application termination to the Monitoring Manager and deactivates (via the application ADMIRE application controller) the application monitoring.
- In line 46 the IC updates the application status in the System Status Table, removes the application communication channel, and terminates the IC processes related to the application (Application Analytic Component and Application Communication Controller).
- In line 47, the I/O Scheduler executes any data transfers to the back-end storage system requested by the application/ad-hoc storage system, ensuring that QoS constraints for the application are enforced.
- Finally, in line 48 the ad-hoc storage system instance is destroyed unless other jobs in the Slurm queue have needed it.

Algorithm 2 shows the execution for Option 2, which introduces a few changes concerning the previous one (Option 1). In this case, the Intelligent Controller is in charge of communicating with Slurm for the resource allocation and the application deployment. This controller is also responsible for communicating with the Application Manager (to send the malleability commands and to receive application information). The main differences between both algorithms are shown in blue color.

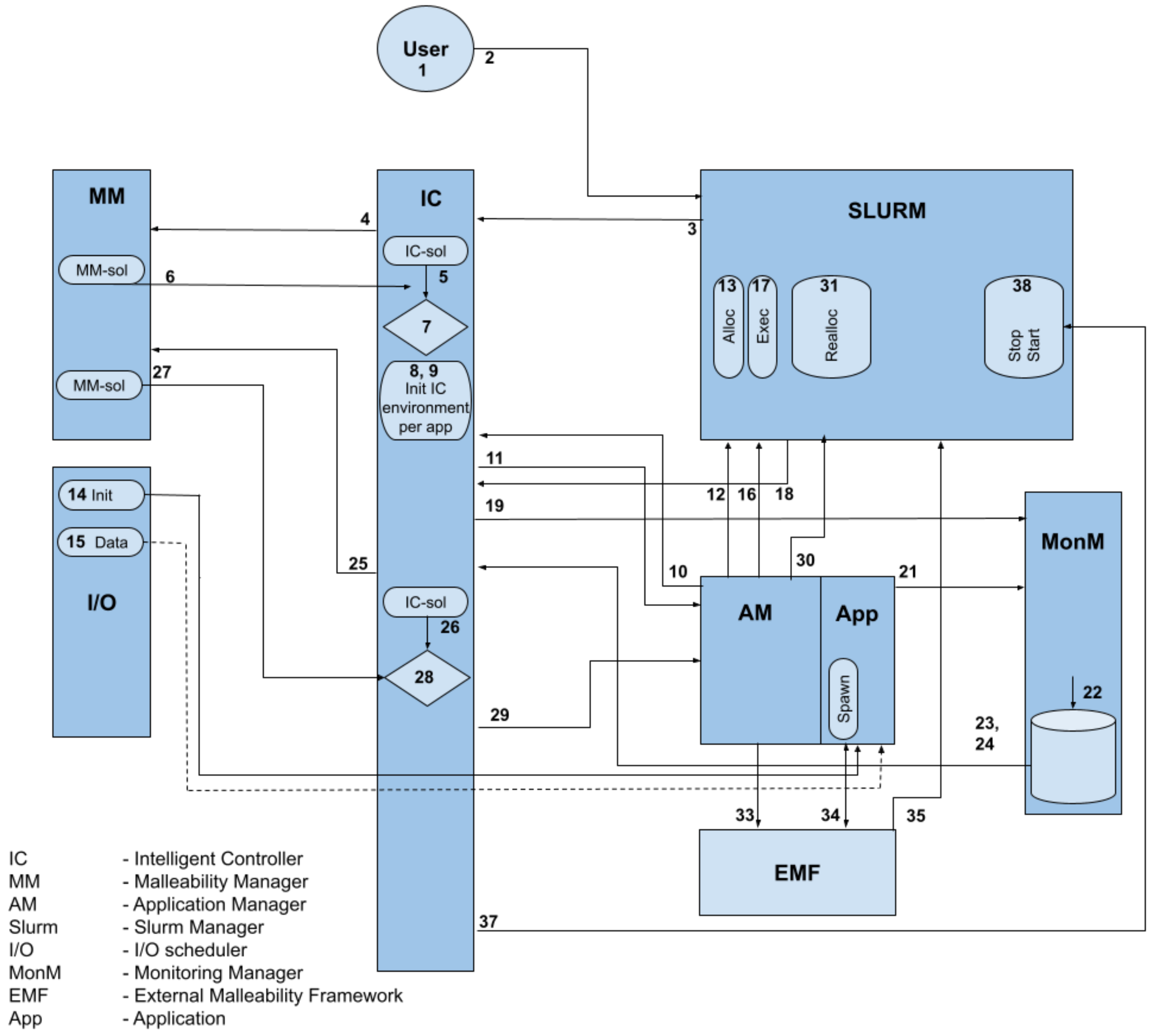
To show a graphical view of the proposed workflows, Figures 3.4 and 3.5 show a flow diagram of the Algorithm 1 (based on the coordination between IC and AM) and Algorithm 2 (based on the coordination between IC and Slurm) respectively. Each arrow is identified with a line number in the algorithm that it represents. Note that this is an overview: it does not include the whole interactions between the components. The objective is to provide a graphical view of the main components that participate in the execution/reconfiguration of the applications.

Algorithm 1: Application life-cycle when executed within ADMIRE framework (**Option 1: Application Manager is in charge of communicating with the Slurm Manager**). For each line, the entity (programmer, Slurm, etc.) that initiates the action is shown. Then a brief description of the action is provided.

```
 1: User: application creation
 2: User: job submission
 3: Slurm: static user hints notification
 4: Intelligent Controller: sending the new jobs and their related data to the Malleability Manager
 5: Intelligent Controller: generating the initial schedule solution (IC-sol)
 6: Malleability Manager: providing its initial schedule solution (MM-sol)
 7: Intelligent Controller: evaluating the best schedule solution between IC-sol and MM-sol
 8: Intelligent Controller: creation of the IC components related to the application
 9: Intelligent Controller: application communication channel creation
10: Application Manager: dynamic user hints notification
11: Intelligent Controller: sending the starting commands to the Application Manager
12: Application Manager: sending allocation command to Slurm
13: Slurm: resource allocation
14: I/O Scheduler: ad-hoc storage system initialization
15: I/O Scheduler: transfer of required input datasets (optional)
16: Application Manager: sending job execution command to Slurm
17: Slurm: job execution
18: Slurm: job execution notification
19: Intelligent Controller: application monitoring activation
20: while application is running do
21:     Monitoring Manager: monitoring metric collection
22:     Monitoring Manager: application performance indicator update
23:     Intelligent Controller: application performance model update
24:     Intelligent Controller: system-wide performance forecast update
25:     Intelligent Controller: system status notification
26:     Intelligent Controller: generating the new schedule solution (IC-sol)
27:     Malleability Manager: providing a new schedule solution (MM-sol)
28:     Intelligent Controller: evaluating the best schedule solution between IC-sol and MM-sol
29:     Intelligent Controller: sending the malleability commands to the Application Manager
30:     Application Manager: sending re-allocation command to Slurm
31:     Slurm: resource allocation
32:     if ∃ External_Malleability_Framework then
33:         Application Manager: sending reconfiguration command to External Framework (e.g. FlexMPI
34:         External Framework: spawning the processes and collecting their PIDS
35:         External Framework: PIDs are sent to AM to annotate them for SLURM management
36:     else
37:         Intelligent Controller: sending reconfiguration command to SLURM
38:         Slurm: stopping and restarting the application with the new number of processes
39:     end if
40:     Intelligent Controller: application notification
41:     I/O Scheduler: background transfers of datasets (optional)
42: end while
43: Slurm: application termination
44: Intelligent Controller: application termination notification
45: Intelligent Controller: application monitoring deactivation
46: Intelligent Controller: application removal
47: I/O Scheduler: transfer of required output datasets (optional)
48: I/O Scheduler: ad-hoc storage system termination (optional)
```

Algorithm 2: Application life-cycle when executed within ADMIRE framework (**Option 2: The Intelligent Controller manages the communication regarding the malleability with the Slurm Manager and the Application Manager**). For each line, the entity (programmer, Slurm, etc.) that initiates the action is shown. Then a brief description of the action is provided.

```
 1: User: application creation
 2: User: job submission
 3: Slurm: static user hints notification
 4: Intelligent Controller: sending the new jobs and their related data to the Malleability Manager
 5: Intelligent Controller: generating the initial schedule solution (IC-sol)
 6: Malleability Manager: providing its initial schedule solution (MM-sol)
 7: Intelligent Controller: evaluating the best schedule solution between IC-sol and MM-sol
 8: Intelligent Controller: creation of the IC components related to the application
 9: Intelligent Controller: application communication channel creation
10: Application Manager: dynamic user hints notification
11: Intelligent Controller: sending allocation commands to Slurm
12: Slurm: resource allocation
13: I/O Scheduler: ad-hoc storage system initialization
14: I/O Scheduler: transfer of required input datasets (optional)
15: Intelligent Controller: sending job execution command to Slurm
16: Slurm: job execution
17: Slurm: job execution notification
18: Intelligent Controller: application monitoring activation
19: while application is running do
20:     Monitoring Manager: monitoring metric collection
21:     Monitoring Manager: application performance indicator update
22:     Intelligent Controller: application performance model update
23:     Intelligent Controller: system-wide performance forecast update
24:     Intelligent Controller: system status notification
25:     Intelligent Controller: generating the new schedule solution (IC-sol)
26:     Malleability Manager: providing a new schedule solution (MM-sol)
27:     Intelligent Controller: evaluating the best schedule solution between IC-sol and MM-sol
28:     Intelligent Controller: sending the malleability commands to Slurm
29:     Slurm: resource allocation
30:     Intelligent Controller: sending reconfiguration command to the Application Manager
31:     if $\exists$ External_Malleability_Framework then
32:         Application Manager: sending reconfiguration command to External Framework (e.g. FlexMPI
33:         External Framework: spawning the processes and collecting their PIDS
34:         External Framework: PIDs are sent to AM to annotate them for Slurm management
35:     else
36:         Application Manager: sending reconfiguration command to Slurm
37:         Slurm: stopping and restarting the application with the new number of processes
38:     end if
39:     Intelligent Controller: application notification
40:     I/O Scheduler: background transfers of datasets (optional)
41: end while
42: Slurm: application termination
43: Intelligent Controller: application termination notification
44: Intelligent Controller: application monitoring deactivation
45: Intelligent Controller: application removal
46: I/O Scheduler: transfer of required output datasets (optional)
47: I/O Scheduler: ad-hoc storage system termination (optional)
```



IC	- Intelligent Controller
MM	- Malleability Manager
AM	- Application Manager
Slurm	- Slurm Manager
I/O	- I/O scheduler
MonM	- Monitoring Manager
EMF	- External Malleability Framework
App	- Application

Figure 3.4: ADMIRE workflow diagram (Option 1).

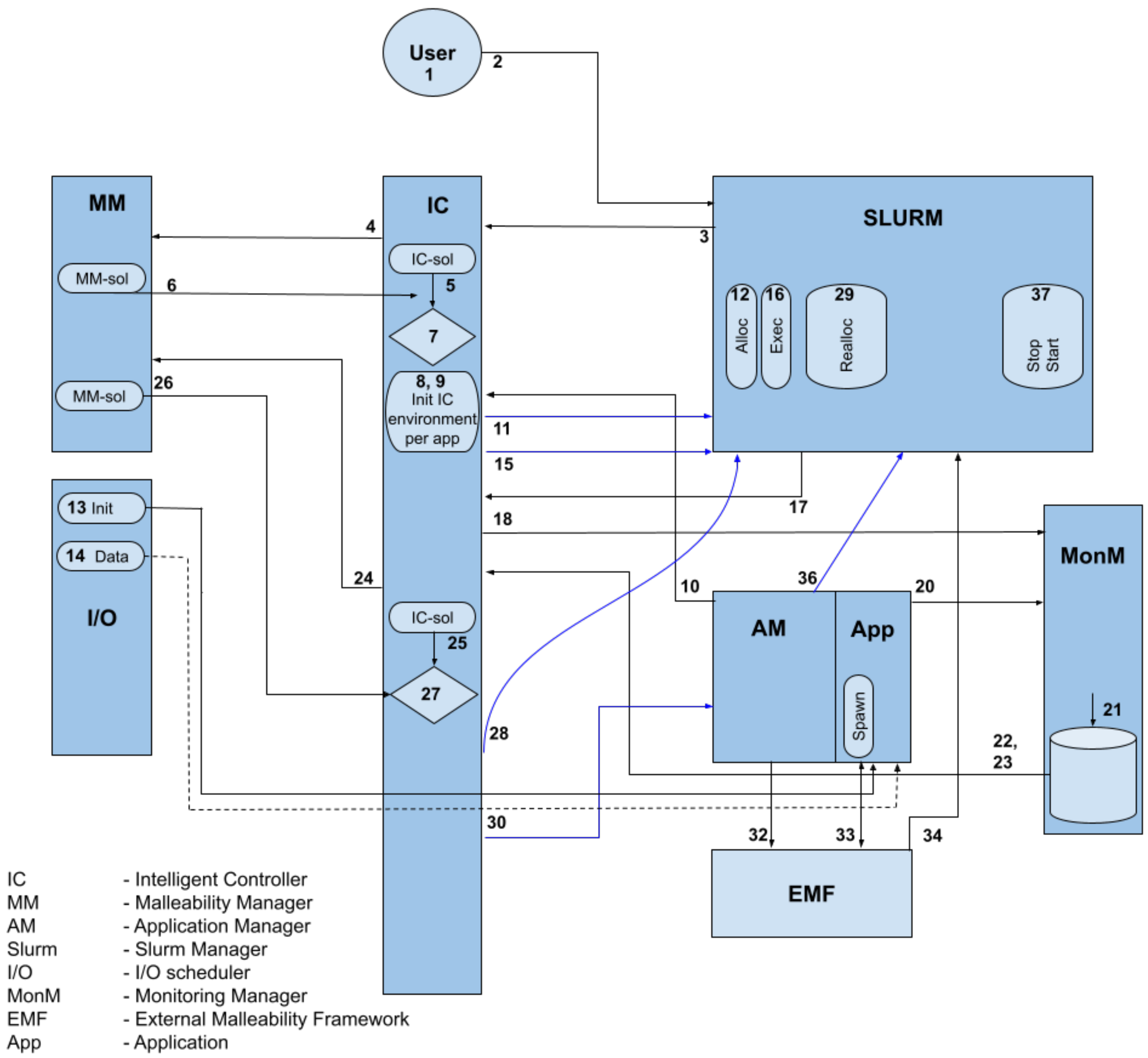


Figure 3.5: ADMIRE workflow diagram (Option 2).

Chapter 4

Application Programming Interface

This section describes the API definition of the communications related to the Intelligent Controller. Figure 4.1 shows the control flow diagram of the overall ADMIRE architecture. We can observe that the Intelligent Controller communicates with the Malleability Manager, SLURM, I/O Scheduler, Ad-hoc Storage system, the Monitoring Manager, and applications (both the application itself and the ADMIRE application manager). The following sections describe the APIs related to each one of these components taking into account the functionalities offered by the Intelligent Controller.

4.1 Interface of Malleability Manager

There are two communication channels between the Malleability Manager and the Intelligent Controller. The first one (the function ADM_getSystemStatus, ID1 in Figure 4.1), provides the Malleability Manager with information collected and processed by the Intelligent Controller. This information includes platform and application status information, performance models, and user hints. It offers a global view of the current state of the system as well as a forecast of future platform states. The second one (the function ADM_suggestScheduleSolution, ID2 in Figure 4.1) includes the job and I/O malleable decisions taken by the Malleability Manager for certain running applications and ad-hoc storage systems, respectively. This function is depicted in the Deliverable D3.1.

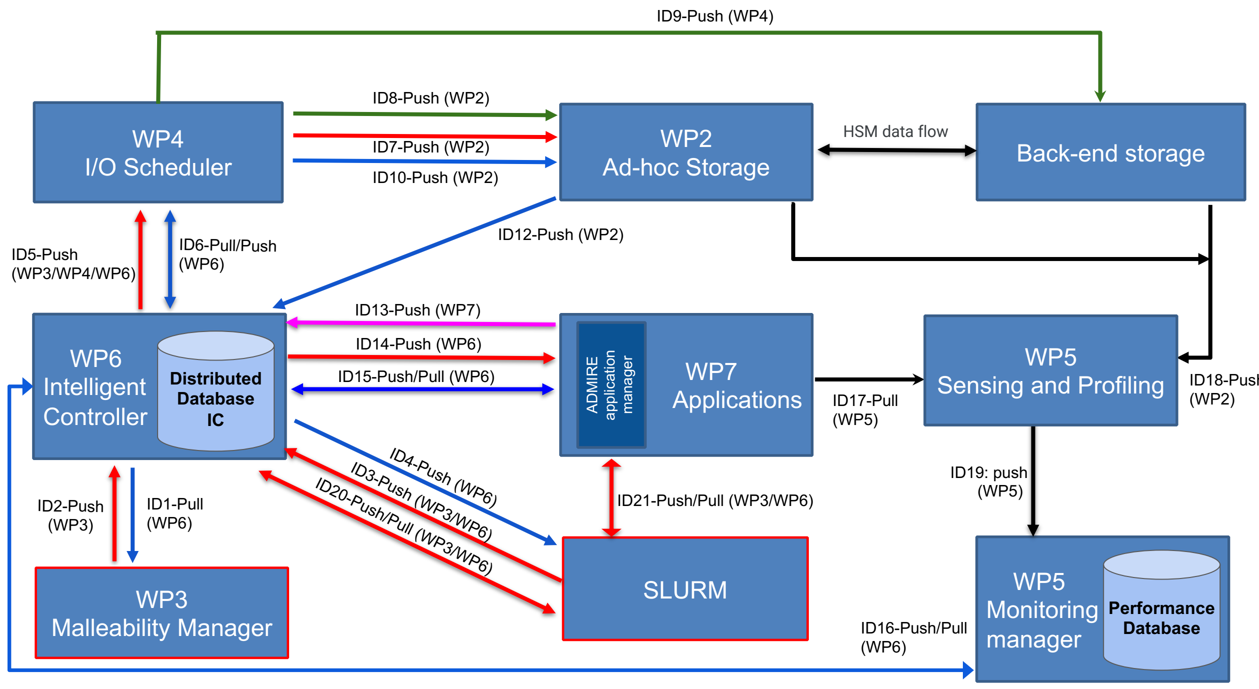


Figure 4.1: ADMIRE control flow between the components.

The Malleability Manager accesses the required information to provide malleable-aware scheduling of the existing running application. The information needed to produce this scheduling is obtained through the Redis database. This strategy reduces the amount of data that has to be exchanged between the components (Malleability Manager, Intelligent Controller, etc.), and allows the Malleability Manager to access the information when needed (on-demand access), instead of waiting for receiving it from the Intelligent Controller. This strategy reduces the communication protocol complexity as well as the Malleability Manager response time.

Name: *ADM_getSystemStatus*
input : None.
output: struct systemStatus. Structure with the system information. It contains the following fields:

- `jobQueueState`. Status of each job that is in the queue of Slurm Workload Manager. The information provided will be related to Slurm.
- `userHints`. Collection of user hints, per running application, provided by the user via Slurm or the ADMIRE application API.
- `applicationState`. Status of each application being executed in the platform. The information provided will include information collected from all ADMIRE components: previous malleable reconfigurations made by the application, I/O scheduling policies that are being applied, user hints, etc.
- `applicationPerformanceModels`. Performance metrics and models related to each executing application.
- `systemState`. Status of each compute node of the platform. It includes metrics of resource utilization.
- `IOState`. Status of the ad-hoc storage system. Information includes the list of nodes involved in each ad-hoc storage system, their applied configurations, and used performance metrics (the actual performance metrics are sent to the Sensing and Profiling module of WP5).
- `QoS`. Quality of Service metrics related to the ad-hoc and back-end storage systems.
- `systemPerformanceForecast`. Prediction of the future platform state including system use and potential contention hazards.

Description:
This function provides system information by the intelligent controller to the malleability manager about the system state, also including application models and future contention hazards.

Name: *ADM_slurmMallRequestOP1*
input : struct slurmMallRequestOP1. Structure of the reconfiguration that Slurm has to perform. The operations can be allocating new nodes or removing some of them. It contains the following fields:

- `Job`. Job ID.
- `Operation`. Expand/Shrink/Move
- `nNodes`. Number of nodes to add/remove.

output: List of new allocated resources
Description:
This function allocates the resources that the Application Manager requires. This step is followed by (1) the execution of an new application or (2) the (malleable) reconfiguration of a running one.

4.2 Interface of Slurm

The information exchanged between Slurm and the Intelligent Controller includes the scheduling solution obtained by the Malleability Manager —that is forwarded to Slurm—. This solution comprises the requests for the allocation/deallocation of certain compute nodes (function ADM_JobScheduleSolution, ID3 in Figure 4.1) when a malleable operation of expand/shrink of processes is going to be carried out for a certain application. The second communication data is the application status information collected by Slurm, which includes the existing unscheduled and running jobs and the user hints provided via Slurm. This information is sent to the Intelligent Controller (function ADM_slurmState, ID4 in Figure 4.1) Finally, when the Intelligent Controller implements the workflow described by the Algorithm 1, then the Intelligent Controller delegates the communication with the Slurm to the Application Manager (ID21 in Figure 4.1). However, if the workflow corresponds to the one described in Algorithm 2, then, the Intelligent Controller is in charge of sending the application allocation information to Slurm, and then, Slurm returns to it the available resources to run the application or to perform malleable operations (ID20 in Figure 4.1).

Name: *ADM_jobScheduleSolution*
input : struct jobScheduleSolution. Structure of the job reconfiguration solution. It contains the following fields:

- `applicationID`. Application subjected to be reconfigured.
- `assignedNodes`. List of compute nodes involved in the job reconfiguration.
- `assignedProcessesPerNode`. Number of processes that have to be created (if positive) or removed (if negative) for each one of the previous nodes.

output: int exitValue: 0 success, -1 failure
Description:
This function forwards the final job schedule and computational requirements (optionally overridden by the Intelligent Controller) to the resource manager.

Name: *ADM_slurmState*
input : struct slurmState. Structure of the information collected by Slurm about the running and unscheduled jobs. It contains the following fields:

- `uJobs`. List of unscheduled jobs including requirements and user hints
- `rJobs`. List of running jobs
- `sJobs`. List with the state of running jobs.
- `sResources`. List with the state of system resources
- `uHints`. List with user hints provided for each job

output: int exitValue: 0 success, -1 failure
Description:
This functions forwards the overall system state available to Slurm including user hints of jobs to the Intelligent Controller.

Name: *ADM_slurmMallRequest*
input : struct slurmMallReq. Structure of the reconfiguration that Slurm has to perform. The operations can be allocating new nodes or removing some of them. It contains the following fields:

- `Job`. Job ID.
- `Operation`. Expand/Shrink/Move
- `nNodes`. Number of nodes to add/remove.

output: List of new allocated resources
Description:
This function allocates the resources that the IC needs. This step is followed by (1) the execution of an new application or (2) the (malleable) reconfiguration of a running one.

Name: *ADM_slurmMallRequestOP1*
input : struct slurmMallRequestOP1. Structure of the reconfiguration that Slurm has to perform. The operations can be allocating new nodes or removing some of them. It contains the following fields:

- `Job`. Job ID.
- `Operation`. Expand/Shrink/Move
- `nNodes`. Number of nodes to add/remove.

output: List of new allocated resources
Description:
This function allocates the resources that the Application Manager requires. This step is followed by (1) the execution of an new application or (2) the (malleable) reconfiguration of a running one.

4.3 Interface of I/O Scheduler

There are two sources of information exchanged between the I/O Scheduler and the Intelligent Controller. The first one (function ADM_IOScheduleSolution, ID6 in Figure 4.1) is the scheduling solution obtained by the

Malleability Manager, which is now forwarded to the I/O Scheduler. This information includes the requests for the allocation/deallocation of storage instances, the assignment of certain I/O bandwidth to a given application, and the suggestion of I/O scheduling policies —also related to a certain application—. It will also include the definition of policies for allowing the insertion of processing rules on data for supporting dynamic in-situ/in-transit processing. Each active component will expose a series of parameters that are to be configured in a control point (e.g., the number of servers to stripe a file, the block size).

The second one (function ADM_getSystemStatus, ID5 in Figure 4.1), is the same as the one related to the Malleability Manager (ID1 in Figure 4.1): it provides information collected and processed by the Intelligent Controller, including platform and application status information, performance models, and user hints.

Besides, it is important to note that the I/O scheduler leverages the NORNS data staging service. It is a component that gets information about I/O operations and I/O movements from different tiers (e.g. local NVM and PFS) and can apply various transfer scheduling options.

Name: *ADM_IOScheduleSolution*
input : struct IOScheduleSolution. Structure of the ad-hoc storage system reconfiguration solution. It contains the following fields:

- `adhocStorageID`. Ad-hoc storage system subjected to be reconfigured.
- `jobSchedule`. I/O scheduling policy for the application.
- `assignedBandwidth`. Assigned I/O bandwidth for the considered application.
- `adhocStorageNodes`. List of compute nodes involved in the ad-hoc storage system reconfiguration.
- `adhocStorageMalleabilityRequest`. Number of I/O storage instances that have to be created (positive values) or removed (negative values) for the previous list of nodes.

output: int exitValue: 0 success, -1 failure
Description:
This function forwards the final job schedule and I/O requirements (optionally overridden by the Intelligent Controller) to the I/O Scheduler. It also includes the ad-hoc storage malleability commands.

Name: *ADM_getSystemStatus*
input : None.
output: struct systemStatus. Structure with the system information.
Description:
This function is the same as the one described in the interface with Malleability Manager (see Section 4.1 for more details about this interface).

4.4 Interface of the ad-hoc storage system

There is a single communication channel between the ad-hoc storage and the Intelligent Controller (function ADM_IOStatus, ID 12 in Figure 4.1). This interface allows ad-hoc storage systems to send a confirmation message to the Intelligent Controller that a specific malleable request has been applied by the storage system. In general, when malleable decisions are made by the Malleable Manager, the decision is passed to the I/O scheduler (which manages the ad-hoc storage systems) via the Intelligent Controller. The I/O scheduler then sends the malleable request to the corresponding ad-hoc storage system. Such requests not only include malleable actions but also changing performance metrics (available bandwidth, QoS, etc.) according to the Malleable Manager and the I/O scheduler policies. Kindly refer to D2.1 for additional information on this interface.

Name: *ADM_IOStatus*
input : struct IOStatus. Structure of the ad-hoc storage status. It contains the following fields:

- `adhocStorageIDs`. Storage identification
- `adhocStorageNodes`. (only for ad-hoc storage system), list of compute nodes used by the ad-hoc storage system.
- `configurationParameters`. List with different configuration parameters related to the ad-hoc storage system that have been applied.

output: int exitValue: 0 success, -1 failure
Description:
This function provides the status of the I/O system including the nodes where it is deployed (in the case of the ad-hoc storage system) and different configuration parameters related to the storage system.

4.5 Interface of ADMIRE Application Manager

The ADMIRE Application Manager is executed with the applications and is responsible for applying the reconfiguration – malleable – actions and monitoring commands. The reconfiguration actions involve the creation/destruction of application threads (functions ADM_SpawnThread and ADM_RemoveThread, ID 14 in Figure 4.1) or processes (functions ADM_SpawnProcess and ADM_RemoveProcess, ID 14 in Figure 4.1). The monitoring commands include the activation/deactivation of the application monitoring service (function ADM_MonitoringService, ID 14 in Figure 4.1). This component also shares a communication function, which has been already defined in Section 4.2, called ADM_slurmMallRequestOP1 (ID20 in Figure 4.1), which is in charge of sending the new reallocation to Slurm.

Name: *ADM_SpawnThread*
input : int *threadList: list of the number of threads created in each compute node
input : char **computeNodes: list of the compute nodes where the threads are created
output: int exitValue: 0 success, -1 failure
Description:
This function sends a command to the application's ADMIRE application manager for creating one or several new MPI threads. The input arguments include two lists, one with the number of processes that are created in each compute node and other with the related compute node ids. Note that a zero value in processList means that no threads have to be created. Negative values of processList are not allowed, and it will return -1 (failure) code. Not including a certain compute node will not change the application number of threads running on the node. It is only possible to create threads in compute nodes already in use by the application.

Name: *ADM_RemoveThread*
input : int *threadList: list of the number of threads removed in each compute node
input : char **computeNodes: list of the compute nodes where the threads are destroyed
output: int exitValue: 0 success, -1 failure
Description:
This function sends a command to the application's ADMIRE application manager for removing one or several new MPI threads. The input arguments include two lists, one with the number of threads that are destroyed in each compute node and other with the related compute node ids. Note that a zero value in threadList means that no threads have to be removed. Negative values of threadList are not allowed, and it will return -1 (failure) code. Not including a certain compute node will not change the application number of threads running on the node. Destroying a number of threads greater than the existing ones for a certain compute node will return -1 (failure) code.

Name: *ADM_SpawnProcess*
input : int *processList: list of the number of processes created in each compute node
input : char **computeNodes: list of the compute nodes where the processes are created
output: int exitValue: 0 success, -1 failure
Description:
This function sends a command to the application's ADMIRE application manager for creating one or several new MPI processes. The input arguments include two lists, one with the number of processes that are created in each compute node and other with the related compute node ids. Note that a zero value in processList means that no processes have to be created. Negative values of processList are not allowed, and it will return -1 (failure) code. Not including a certain compute node will not change the application number of processes running on the node.

Name: *ADM_RemoveProcess*
input : int *processList: list of the number of processes removed in each compute node
input : char **computeNodes: list of the compute nodes where the processes are destroyed
output: int exitValue: 0 success, -1 failure
Description:
This function sends a command to the application's ADMIRE application manager for removing one or several new MPI processes. The input arguments include two lists, one with the number of processes that are destroyed in each compute node and other with the related compute node ids. Note that a zero value in processList means that no processes have to be removed. Negative values of processList are not allowed, and it will return -1 (failure) code. Not including a certain compute node will not change the application number of processes running on the node. Destroying a number of processes greater than the existing ones for a certain compute node will return -1 (failure) code.

Name: *ADM_MonitoringService*
input : int command: Action to be done on the given target nodes
input : int optarg: Optional argument attached to the given command
output: int exitValue: 0 success, -1 failure
Description:
This function sends a command to the application's ADMIRE application manager for enabling or disabling the application monitoring service. Despite only one command being sent, this command will be broadcast (by the ADMIRE application manager) to all application processes. Consequently, the monitoring of all the application processes will be affected by command. Supported commands will cover both process attach and detach along with monitoring verbosity. The optional argument will enable finer tuning by the Intelligent Controller.

4.6 Interface of ADMIRE applications

ADMIRE's application developers may provide hints at run-time to the Intelligent Controller (IC), for example, related to I/O patterns of their application code, or about safe regions of code into which malleable commands coming from the IC can be safely executed. Such information, provided directly by the programmer through an API, might be beneficial to the IC to make informed decisions for optimizing applications' execution on the ADMIRE platform. For example, based on user-defined hints, the IC may decide to optimize I/O operations in an annotated code region by employing local or shared burst buffers or enforcing aggressive buffering or prefetching at the file system level. Nevertheless, it is worth remarking that the insertion of API calls in the application code aiming at providing hints to the IC is not a mandatory task for the developers. Notwithstanding, developers' hints may be of foremost importance to obtain those pieces of information challenging to deduce from system monitoring at run time.

We have identified three distinct classes of API. The first one is to provide the IC with hints related to I/O behavior in the applications. The second one is for identifying code regions into which it is safe to reconfigure the application (i.e., receive and execute malleable commands, provided that the application is malleable). Finally, the third class contains commands to obtain system information (e.g., the current I/O bandwidth available) in the form of aggregated metrics directly from the IC.

4.6.1 Interface for I/O patterns identification

The interface described in this section allows HPC application developers to provide the Intelligent Controller (function ADM_IOHintsRegion, ID 13 in Figure 4.1) with information related to I/O patterns present in their applications. I/O patterns within the application are identified by defining some regions of code, possibly nested. A region of code is a piece of code included between two distinct API calls. The first one identifies the beginning of the region and, through some pre-defined tags, the kind of I/O behavior of the instructions enclosed in the region. The second call identifies the end of the region, and in the case of nested regions, which tags are no longer valid.

The IC will profitably use such information provided by the application developers to optimize the execution of such applications and the entire system behavior.

Name: *ADM_IOHintsRegion*
input : $START/STOP$ flag identifying the begin ($START$), and the end ($STOP$) of the code region.
input : bitwise-or of tags each one providing a hint to the Intelligent Controller. The tag codification include:

- `IO_INTENSIVE`. An I/O intensive phase is starting (if the first parameter is $START$) or has been completed (if the first parameter is $STOP$).
- `LOCAL_ONLY_IO`. The files produced are temporary files not supposed to be read by other application processes/nodes. They will be read and/or sometimes updated in the future, and then they will be destroyed.
- `READ_MODIFY_WRITE`. This tag announces a read-modify-write I/O access pattern, meaning that read operations will be followed by seek and write operations aiming at updating data in the files.
- `SMALL_IO`. Read/Write operations will be of small size (up to a few kilos).
- `LARGE_IO`. Read/Write operations will be of large size.
- `TEMPORAL_DATA`. The files produced will have a limited time span.
- `SEQUENTIAL_READS`. A sequence of reads is starting (if the first parameter is $START$) that may benefit from aggressive prefetching of data.
- `SEQUENTIAL_WRITES`. A sequence of writes is starting (if the first parameter is $START$) that may benefit from aggressive buffering of data.
- `RESET_ALL_TAGS`. Meaningful if the first parameter is $STOP$ and if this is the only tag provided as the second parameter. It resets to default all I/O tags previously set by all calls containing $START$ as the first parameter.

output: int exitValue: 0 success, -1 failure
Description:
This function allows the application programmer to send "hints/information" to the Intelligent Controller (IC) related to the I/O phases of the application. Hints are provided to the IC through a bitwise-or of pre-defined tags, each with a specific meaning. The list of tags might be further extended in the future if needed.

4.6.2 Interface for malleability

The Malleability Manager is a component in the ADMIRE software architecture that is responsible for providing elasticity in the allocation of resources for a given application or application component. It is also responsible for making decisions related to the dynamic adjustment of the number of resources assigned to an application, which, in turn, may significantly affect the I/O behavior.

The interface described in this section (function ADM_MalleableRegion, ID 13 in Figure 4.1) allows HPC application developers to provide the Intelligent Controller with information about reconfiguration-safe sections of their applications in which malleable commands coming from the IC may be safely executed by the application. It is worth noting that, extracting such information without the help of the application developers, for example, by executing a static analysis of the source code or by analyzing profiling information collected at run-time, is extremely difficult.

Name: *ADM_MalleableRegion*
input : $START/STOP$ flag identifying the begin ($START$), and the end ($STOP$) of the code region.
output: int exitValue: 0 success, -1 failure
Description:
With this function, the application developers tell the Intelligent Controller when it is safe to execute malleable commands. The code region annotated with this API identifies the reconfiguration-safe section of the application in which malleable commands may be accepted and executed by the application. All commands that are in transit before/after the safe code region annotated by this API will be discarded.

4.6.3 Interface for collecting system information

The ADMIRE-enabled application may ask the Intelligent Controller to receive information related to some predefined performance metrics or the system state and actual configurations of some components comprising the ADMIRE software architecture. The interface described in this section (functions ADM_RegisterSysAttributes and ADM_GetSysAttributes, ID 15 in Figure 4.1) allows the HPC application to obtain such system information from the IC. First, the application must register with the IC declaring which metrics or information it would like to ask for in the future. Then, the application can ask for all or a subset of metrics previously registered, as needed.

Name: *ADM_RegisterSysAttributes*
input : bitwise-or of IDs each one identifying a given system attribute provided by the Intelligent Controller. The fields will code the following information:

- `IO_BANDWIDTH`. Aggregate I/O system bandwidth currently available expressed in Gb/s.
- `APP_MANAGER_STATUS`. Status of the application manager.
- `SLURM_MALLEABLE_CAPABILITIES`. Reports whether the *SLURM* component has malleable features enabled.

output: int exitValue: 0 success, -1 failure
Description:
This function notifies the Intelligent Controller (IC) that the application is interested in receiving one (or a few) system attribute(s) upon explicit requests (see also `ADM_GetSysAttributes`*) among those registered with the IC. A system attribute can be a given performance metric (e.g., I/O bandwidth), the status and the capabilities of a given ADMIRE component, and, in general, any predefined system information that might be relevant for the application. The list of attributes is predefined and might be further extended in the future if needed.*

Name: *ADM_GetSysAttributes*
input : bitwise-or of IDs previously registered with the IC.
input : Array of pairs <ID, value> having as many entries as the number of attributes requested.
input : Number of entries of the array.
output: The array is filled out with the information related to each attribute.
Description:
This functions receives the information related to the list of attribute IDs provided as first parameter (IDs are bitwise-or'd). The attributed requested must be previously registered with the IC by using the ADM_RegisterSysAttributes.

4.7 Interface of Monitoring Manager

On one hand, the Intelligent Controller provides the Monitoring Manager with all the information about the system, including platform and application status information, performance models, and user hints. This is implemented in function ADM_getSystemStatus, ID16 in Figure 4.1 which is the same as the one related to the Malleability Manager (ID1 in Figure 4.1). On the other hand, the Monitoring Manager includes different functionalities for accessing the monitoring data. The related functions are described in Deliverable D5.1.

Name: *ADM_getSystemStatus*
input : None.
output: Structure with the system information (see Section 4.1 for more details about this interface).
Description:
This function is the same as the one described in the interface with Malleability Manager

4.8 UML diagram

Figure 4.2 and 4.3 shows the UML diagram of the functions provided by the Intelligent Controller. The diagram illustrates the API definitions related to each one of the ADMIRE architecture components: SLURM, Sensing and Profiling, Malleability Manager, I/O scheduler, ADMIRE application manager, and user. Note that the diagram has been divided into two sub-figures to make it more readable.

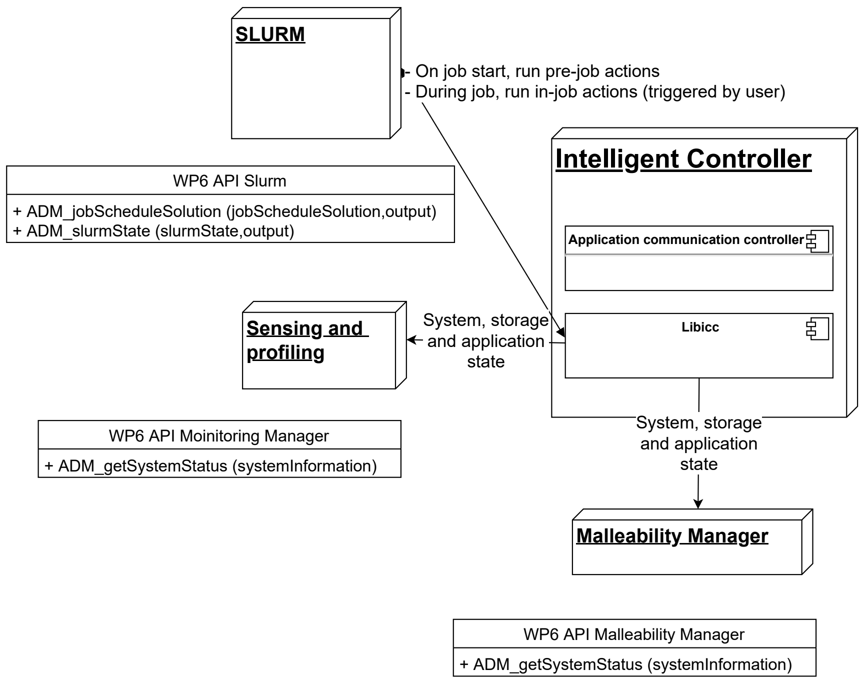

SLURM
- On job start, run pre-job actions
- During job, run in-job actions (triggered by user)
Intelligent Controller
WP6 API Slurm
+ ADM_jobScheduleSolution (jobScheduleSolution,output)
+ ADM_slurmState (slurmState,output)
Application communication controller
Sensing and profiling
System, storage and application state
Libicc
WP6 API Moinitoring Manager
+ ADM_getSystemStatus (systemInformation)
System, storage and application state
Malleability Manager
WP6 API Malleability Manager
+ ADM_getSystemStatus (systemInformation)

Figure 4.2: UML diagram (I) related to WP6 API. The "Sensing and Profiling" component also contains the Monitoring Manager.

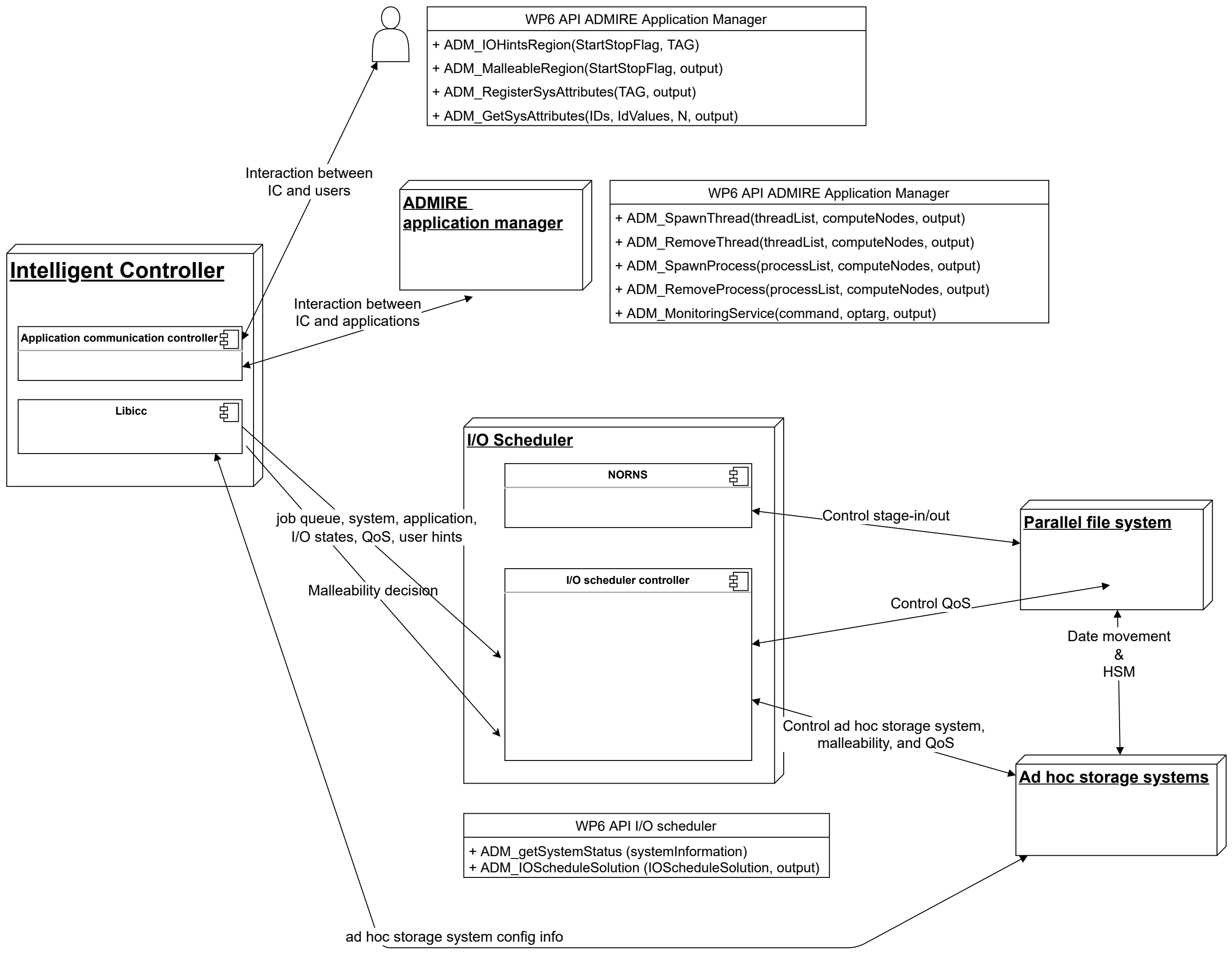

WP6 API ADMIRE Application Manager
+ ADM_IOHintsRegion(StartStopFlag, TAG)
+ ADM_MalleableRegion(StartStopFlag, output)
+ ADM_RegisterSysAttributes(TAG, output)
+ ADM_GetSysAttributes(IDs, IdValues, N, output)
Interaction between IC and users
ADMIRE application manager
WP6 API ADMIRE Application Manager
+ ADM_SpawnThread(threadList, computeNodes, output)
+ ADM_RemoveThread(threadList, computeNodes, output)
+ ADM_SpawnProcess(processList, computeNodes, output)
+ ADM_RemoveProcess(processList, computeNodes, output)
+ ADM_MonitoringService(command, optarg, output)
Intelligent Controller
Interaction between IC and applications
Application communication controller
Libicc
I/O Scheduler
NORNS
Control stage-in/out
Parallel file system
job queue, system, application, I/O states, QoS, user hints
Malleability decision
I/O scheduler controller
Control QoS
Date movement & HSM
Control ad hoc storage system, malleability, and QoS
Ad hoc storage systems
WP6 API I/O scheduler
+ ADM_getSystemStatus (systemInformation)
+ ADM_IOScheduleSolution (IOScheduleSolution, output)
ad hoc storage system config info

Figure 4.3: UML diagram (II) related to WP6 API. The "Sensing and Profiling" component also contains the Monitoring Manager.

Chapter 5

Conclusion

This deliverable describes our proposed architecture of the Intelligent Controller component that integrates cross-layer data for providing a holistic view of the system for making intelligent analysis. Also, this deliverable describes the detailed API for managing the Intelligent Controller at different layers, such as the malleability manager, Slurm, the I/O scheduler, monitoring, the storage layer, and applications.

We described the multiple alternative implementations that the ADMIRE run-time currently supports. These alternatives have been originally assessed and evaluated in deliverable D6.1. From the previous studies and evaluations made, we have reached several conclusions related to multiple design decisions, such as implementing a distributed Intelligent Controller, using RPCs as communications mechanisms, and using a Redis database to decouple the IC and the monitoring and profiling tools. We have also implemented the Intelligent Controller workflow, which includes two design alternatives, and the malleability protocol, which is partially implemented in the Algorithms 1 and 2 of this deliverable.

The next step in this working packet consists of integrating other components with the IC to increase, step by step, the functionality and the control plane previously described.

Appendix A

Terminology

- Ad hoc Storage System, ephemeral storage system that only exists in a determined period, i.e. during a job's execution.
- CLI, command line interface.
- DRAM, dynamic random-access memory.
- 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.
- In situ data, processing the data where it is originated.
- In transit data, processing the data when it is moved.
- NORNS, data transfer service for HPC developed at BSC.
- NVM, non-volatile memory.
- PFS, parallel file system.
- 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.
- 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 sophisticated object stores have been developed on the ground of these concepts such as ADMIRE Data Clay.
- 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.
- PFS, Parallel File System, type of distributed file system supporting a global namespace and spread across multiple storage servers.

- Node Local Storage, ability for a compute server to store persistent data on physically local storage devices.
- 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.
- 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.
- Rest API, such APIs can be developed without constraint of 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).
- OSS, 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.
- MDS, MetaData Server.
- MDT, MetaData Target.
- 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.
- Monitoring Manager,
- Intelligent Controller,
- Monitoring Daemon,
- TBON, Tree Based Overlay Network,
- PromQL, the query language supported by the Prometheus database. Syntax, documentation and examples are available here: https://prometheus.io/docs/prometheus/latest/querying.

Bibliography

[1] Paul Grun, Sean Hefty, Sayantan Sur, David Goodell, Robert D. Russell, Howard Pritchard, and Jeffrey M. Squyres. A brief introduction to the openfabrics interfaces. In *2015 IEEE 23rd Annual Symposium on High-Performance Interconnects (HOTI)*, pages 34–39, 2015.

[2] Florin Isaila, Jesus Carretero, and Rob Ross. Clarisse: A middleware for data-staging coordination and control on large-scale hpc platforms. In *2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, pages 346–355. IEEE, 2016.

[3] Gonzalo Martín, David E Singh, Maria-Cristina Marinescu, and Jesús Carretero. Enhancing the performance of malleable mpi applications by using performance-aware dynamic reconfiguration. *Parallel Computing*, 46:60–77, 2015.

[4] Robert B. Ross, George Amvrosiadis, Philip Carns, Charles D. Cranor, Matthieu Dorier, Kevin Harms, Greg Ganger, Garth Gibson, Samuel K. Gutierrez, , Robert Latham, Bob Robey, Dana Robinson, Bradley Settlemyer, Galen Shipman, Shane Snyder, Jerome Soumagne, and Qing Zheng. Mochi: Composing data services for high-performance computing environments. *Journal of Computer Science and Technology*, 35(1):121–144, 01 2020.

[5] Sangmin Seo, Abdelhalim Amer, Pavan Balaji, Cyril Bordage, George Bosilca, Alex Brooks, Philip Carns, Adrian Castello, Damien Genet, Thomas Herault, Shintaro Iwasaki, Prateek Jindal, Laxmikant V. Kale, Sriram Krishnamoorthy, Jonathan Lifflander, Huiwei Lu, Esteban Meneses, Marc Snir, Yanhua Sun, Kenjiro Taura, and Pete Beckman. Argobots: A lightweight low-level threading and tasking framework. *IEEE Transactions on Parallel and Distributed Systems*, 29(3), 03 2018.

[6] Jerome Soumagne, Dries Kimpe, Judicael Zounmevo, Mohamad Chaarawi, Quincey Koziol, Ahmad Afsahi, and Robert Ross. Mercury: Enabling remote procedure call for high-performance computing. In *2013 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 1–8, 2013.