H2020-JTI-EuroHPC-2019-1

Project no. 956748

ADAPTIVE MULTI-TIER INTELLIGENT DATA MANAGER FOR EXASCALE

D6.1
Report on the intelligent controller requirements

Version 1.0

Date: September 30, 2021

Type: Deliverable
WP number: WP6

Editor: Emmanuel Jeannot
Institution: INRIA

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

Change Log

Rev.	Date	Who	Site	What
1	21/04/21	Jesus Carretero	UC3M	Document creation.
2	04/05/21	Hamid Fard	TUDA	Redesign of ADMIRE architecture.
3	03/09/21	Clément Barthélemy	INRIA	RPCs study.
4	10/09/21	David E. Singh	UC3M	ADMIRE architecture overview.
5	13/09/21	Massimo Torquati	CINI	Applications user interface definition.
6	16/09/21	Jesus Carretero	UC3M	Added terminology.
7	16/09/21	Jesus Carretero	UC3M	Added introduction.
8	20/09/21	David E. Singh	UC3M	Intelligent Controller structure and workflow
9	20/09/21	Hamid Fard	TUDA	Inserting more details about ADMIRE architecture.
10	21/09/21	Clément Barthélemy	Inria	Databases comparison section
11	22/09/21	David E. Singh	UC3M	ADMIRE API, sections 4.1-4.6 and 4.8.
12	22/09/21	Hamid Fard	TUDA	Revising application life-cycle in the framework.
13	23/09/21	Jesus Carretero	UC3M	Added executive summary.
14	23/09/21	Massimo Torquati	CINI	Revised the introduction of ADMIRE architecture and the Application Manager section.
15	23/09/21	David E. Singh	UC3M	Updated images of ADMIRE architecture workflow and data flow.
16	27/09/21	David E. Singh	UC3M	UML diagram.
17	29/09/21	Alberto Miranda	BSC	Full document revision. Provided additional details on I/O Scheduler involvement. Added references to D2.1 and D4.1.

Executive Summary

The Intelligent Controller (IC) is a component in ADMIRE whose primary responsibility is coordinating the execution of applications in the ADMIRE framework with the goal of maximizing the usage of resources and the global performance of the system. To achieve that the IC will provide control mechanisms that, through suitable communication channels, will allow influencing the execution of all the components in the ADMIRE framework, that is, jobs and applications, ad hoc storage systems, the long-term parallel file system (PFS), and other system resources such as compute nodes and the network fabric.

The Intelligent Controller will interoperate with the monitor, applications, system job scheduler (such as Slurm), and the data plane through the control points defined for each. The main idea is to create a distributed control infrastructure that still provides a single system image through a distributed consistency protocol and to provide facilities transparent to the user, such as novel runtime analytics tools to tune I/O system and applications behaviour through control points of multi-criteria distributed control algorithms.

This deliverable includes several foundations for the Intelligent Controller, such as control and data plane architectural blocks for orchestrating system components, the definition of requirements, protocols and workflows of the Intelligent Controller, and the definition of the Application Programming Interface (API) of the Intelligent Controller with the different ADMIRE components.

Contents

Chapter 1

Introduction

To develop the concept proposed and to achieve the objectives, the ADMIRE project has started 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. We depart from our work in CLARISSE [9] and FlexMPI [13] to combine run-time data analytics, malleability and I/O control mechanisms to enhance the execution of multiple applications.

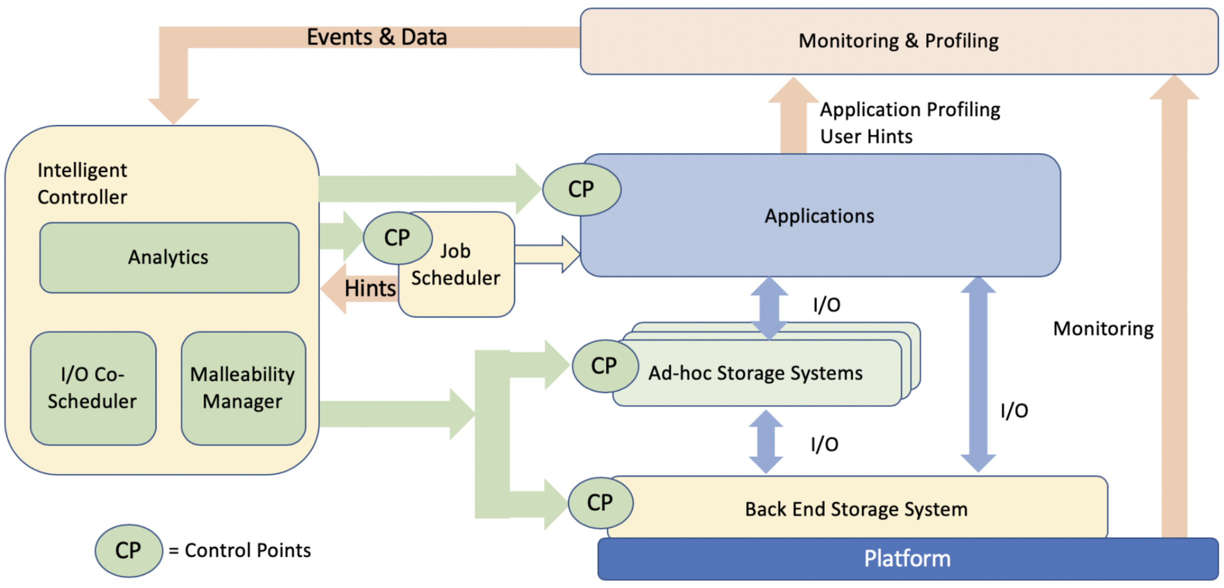

Figure 1.1: ADMIRE architectural blocks.

The heart of ADMIRE is the Intelligent Controller that allows joining cross-layer information from system, applications and users to optimise the throughput of the system and the performance of the applications. The Intelligent Controller will communicate with the other components of the system through control points that will be defined in the project.

The control plane defined will manage a replicated log for resilience. An API will be 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 including the job scheduler, back-end storage systems, and I/O scheduler. The data plane module is composed by ad-hoc storage systems developed in ADMIRE and the back-end storage systems existing in the HPC platforms (such as Lustre).

The Intelligent Controller will interoperate with the monitor, applications, system job scheduler (such as Slurm), and the data plane through the control points defined for each. The main idea is to create a distributed control infrastructure that still provides a single system image through a distributed consistency protocol and to provide facilities transparent to the user, such as novel runtime analytics tools to tune I/O system and applications behaviour through control points of multi-criteria distributed control algorithms.

Applications and system 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 analytics component, the malleability management component, and the I/O scheduler. This way, it integrates and analyses cross-layer system data to dynamically and intelligently steer the system components. The goal is to optimise at system-scale the data management and the I/O accesses of the running applications based on the input provided by the ecosystem (WP5) and to enforce policies (e.g., I/O scheduling (WP4)) through malleability (WP3) and I/O management (WP2). In order to make decisions it will rely on machine learning techniques to predict resource usage and application behaviour.

This deliverable includes several foundations for the intelligent controller. Chapter 2 includes the control and data plane architectural blocks for orchestrating system components. Chapter 3 includes the definition of requirements, protocols and workflows of the Intelligent Controller. Chapter 4 includes the definition of the Application Programming Interface (API) of the Intelligent Controller with the different ADMIRE components. Chapter 5 shows the closing of the deliverable. Finally, Annex A shows a work done to evaluate communication alternatives such Mercury and Mochi.

Chapter 2

Architecture

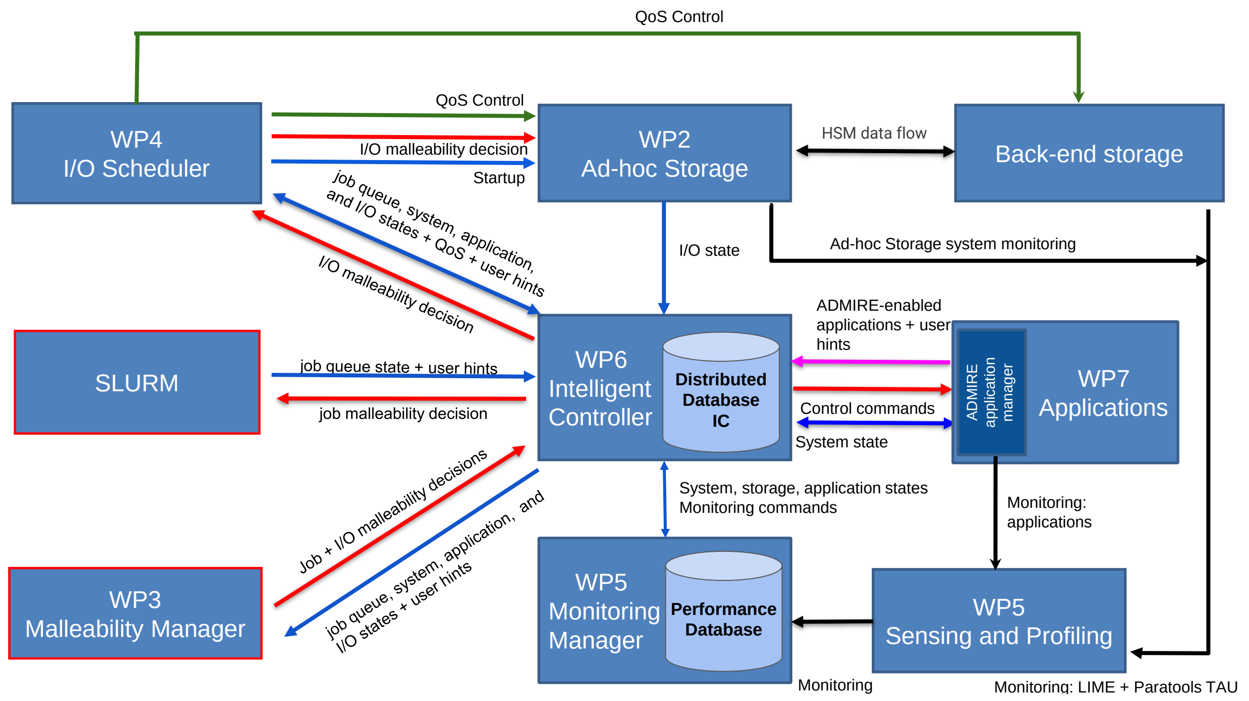
QoS Control
WP4
I/O Scheduler
QoS Control
I/O malleability decision
Startup
WP2
Ad-hoc Storage
HSM data flow
Back-end storage
job queue, system, application, and I/O states + QoS + user hints
I/O malleability decision
I/O state
Ad-hoc Storage system monitoring
ADMIRE-enabled applications + user hints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

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

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 on the upper part of the figure and consists of the ad-hoc and back-end storage systems. The former one is designed in 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 (back-end storage) represents the parallel file system used by the HPC platform (e.g., Lustre). Both storage systems 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, 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-right 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 malleable commands can be executed. Both applications and storage systems are monitored by the Sensing and Profiling component (lower-right corner of the figure), developed in WP5. This component is responsible for collecting system-wide performance metrics at the compute node level that will be stored in an internal database. Also, this component would generate a performance model to help the malleability manager for a more concrete schedule solution. The Monitoring Manager (lower-central part of the figure) will manage this database and will generate performance models related to each running application, storage system (both ad-hoc and back-end), and compute nodes. The Malleability Manager (lower-left corner of the figure) is responsible for determining the 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 to better balance the computation and I/O requirements. The Intelligent Controller (central part of the figure) has different roles. The first one is to collect the current system status using the information collected from Slurm, the Monitoring Manager, the storage systems and the applications. This will include combined information about the hardware status, the existing running applications and the storage. This information will be kept in an internal distributed database. The second role of the Intelligent Controller is to generate performance models of these components and use them to predict potential performance bottlenecks in the system. These models will also be provided to the Malleability Manager and I/O scheduler to support the decision-making of both of them. The other main role of the Intelligent Controller is to coordinate the actions taken by other ADMIRE's components. Examples of these actions are (1) to activate/deactivate each application monitoring and (2) to send the malleable decisions taken by the Malleability Manager to the I/O scheduler or the applications, in case of being I/O-related or application-related decisions, respectively.

2.1 WP6 focus

The Intelligent Controller (IC) is a multi-criteria distributed component that will integrate cross-layer data for providing a holistic view of the system. The IC will also make intelligent analysis to adapt dynamically the system to current and future workloads. It will enable the global orchestration of the exascale system components through an intelligent dynamic control plane. In this way, the IC interoperates with the Monitoring Manager, applications, Slurm job scheduler, malleability management, 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 behaviour 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 analysed using novel data-centric machine learning techniques to predict application and system behaviour. Additional information will be collected by the historical job records of previous runs. In this context, the analytics component of the IC will holistically integrate and analyse cross-layer system data to dynamically and intelligently steer the system to tune I/O performance at system-scale. The strategy is to train model classifiers that can handle jobs that have heterogeneous I/O patterns, detect congestion, perform resource provisioning, perform anomalies detection that will help to identify and predict potential problems and failures of applications.

- **Malleability management support.** ADMIRE project will leverage malleability to maximise system throughput by adjusting both the set of computational and I/O resources assigned to a job. In this context the IC will provide the aforementioned models and predictions to support the I/O and application malleability policies implemented in the Malleability Manager.

- **I/O scheduling support.** The IC will provide decentralised 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 to handle critical stages such as the co-scheduling of many I/O intensive jobs that could cause I/O contention and under-utilisation of other resources, ultimately degrading performance. The idea is to combine the analytic component prediction models with decision-making mechanisms that will follow two directions. First, for scalability reasons, it will leverage on the I/O scheduling algorithm to provide a distributed version dealing with partial knowledge, failures and asynchronisity. Second, it will determine multi-criteria objective trade-offs such as response-time vs. throughput vs. energy. 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 though control points to the developers of control algorithms: dynamically enable/disable control domains by activating/deactivating control agents; provide an open API for implementation of new control algorithms; call the monitoring and profiling service for finding out system and applications I/O state information; and 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 at automatising the generation of low-level scripts/programs (e.g., Slurm scripts) used to run the different jobs composing the application workflow. Therefore, in order 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 the 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.

Additional IC features are:

- 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.

- 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 back-end storage system and to 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

3.1 Intelligent Controller design

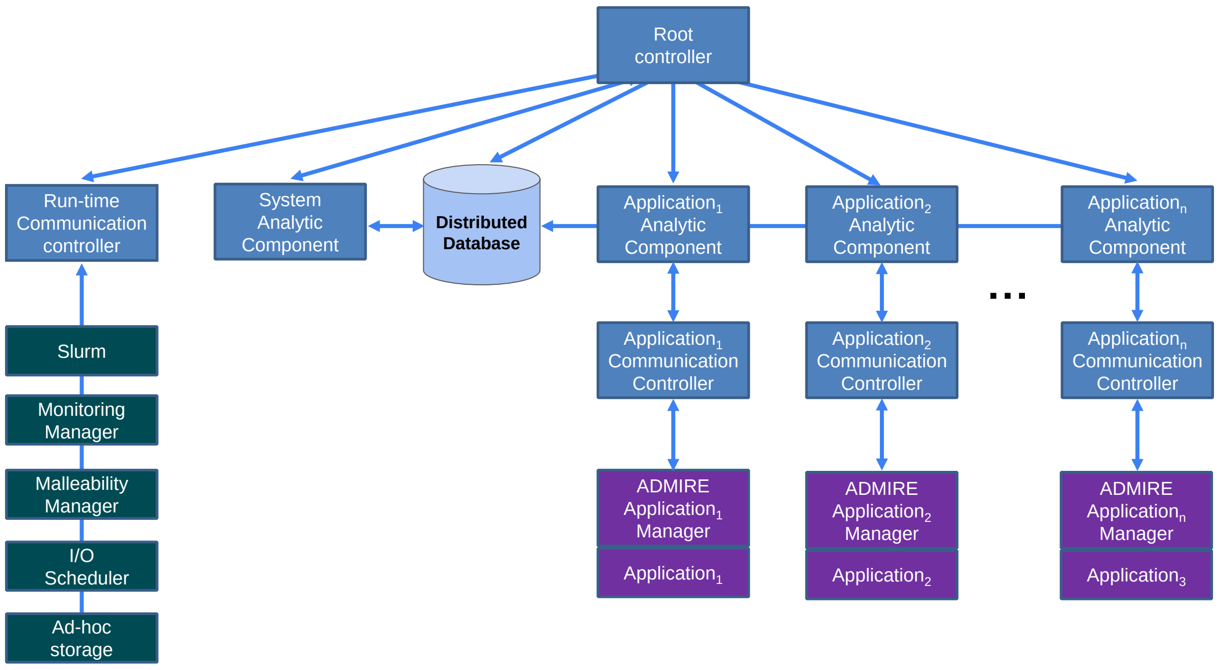
Root controller
Run-time Communication controller
System Analytic Component
Distributed Database
Application1 Analytic Component
Application2 Analytic Component
Applicationn Analytic Component
...
Slurm
Monitoring Manager
Malleability Manager
I/O Scheduler
Ad-hoc storage
Application1 Communication Controller
Application2 Communication Controller
Applicationn Communication Controller
ADMIRE Application1 Manager
ADMIRE Application2 Manager
ADMIRE Applicationn Manager
Application1
Application2
Application3

Figure 3.1: ADMIRE Intelligent Controller structure. 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.

Figure 3.1 shows the Intelligent Controller (IC) structure. It is organized as a hierarchical architecture in which a central controller (called *Root controller*) is responsible of creating and managing the rest of the IC components. All the information managed by the IC is stored in a *distributed database* (see section 3.4). The *Run-Time communication controller* is responsible of the communication with other ADMIRE's components like the Slurm application scheduler, the Monitoring Manager, the Malleability Manager, the I/O scheduler and the Ad-hoc storage. Note that this communication is bi-directional, involving the collection and dissemination between these components of metadata, commands and performance metrics. The *System Analytic component* will apply machine-learning techniques to model and predict the application performance. This will be carried out by combining the information provided by the Monitoring Manager, the application hints and the local performance records stored in the distributed database. The System Analytic output will be also stored in the distributed database. For each running application (shown in the figure as boxes in purple color) the root controller will execute an *Application Communication Component* and an *Application Analytic Components*. The former one is responsible of the communications between the application and the IC, while the latter one is responsible of collecting information about the application execution[1] and storing it in the IC's distributed database.

Note that this architecture will be replicated in several compute nodes in order to provide resilience capabilities to the framework.

3.2 ADMIRE Application Manager

The ADMIRE Application Manager is in charge of interfacing the IC with the single application running on the ADMIRE platform. It may be executed as a library attached to the running applications or as a process/threads spawned for interacting with the IC asynchronously[2] (see Figure 3.1). To reduce the related overhead on the application side, the communication with the IC will only involve a single component associated with the application root process. The main roles of the Application Manager will be:

- 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 will be 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 on the basis of control monitoring commands coming from the IC.
- Sending information upon request to the application about the platform status to guide application-dependent optimizations.

3.3 Communication protocol alternatives

The different components of the ADMIRE project will communicate using remote procedure calls (RPCs), a paradigm where local calls are executed on remote resources. The NFS filesystem for example makes heavy use of RPCs, Google gRPC and Apache Thrift are famous frameworks that support this paradigm.

The contenders for the project are Mercury [16] and Margo [14], 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 [15] 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.

Investigations were made into these two libraries. The results are shown in Appendix A. Table 3.1 presents a summary comparison between Mercury and Margo. Because Margo inherits the capabilities of Mercury, the two are very similar. The main selling point of Margo is the ease of programming when compared to Mercury, which has been confirmed by the GekkoFS developers who have first-hand experience with these libraries.

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

[2]This design decision will be made after carrying out an experimental evaluation of both alternatives at the implementation stage of the project.

The Mochi project also offers some interesting building blocks such as SSG that could be used for collective communication.

Finally, a word of warning: some versions of the lower-level libfabric library used by Mercury and Margo appear to have issues. This was independently confirmed by the GekkoFS and Mochi teams, who recommended that library versions be carefully tested and selected.

	Mercury	Mochi Margo
Portability	Good, thanks to libfabric	idem
Performance	Good	idem, no overhead
Project activity	Very active	idem
Documentation	Good, with examples	idem
Fault tolerance	No but primitive exists	idem
Dependencies	libfabric	Mercury, Argobots, json-c
Programming style	callback based	easier, more linear
Collective comm.	No	No but primitive exists

Table 3.1: Summary of Mercury and Margo characteristics.

3.4 Distributed database implementation alternatives

The IC needs a database to store its state and the results of its modeling of the applications running on the cluster. Some tentative requirements for the database management system (DBMS) are:

- Scalability and reliability.
- Support for efficient varied queries for modeling purposes.
- C application programming interface.

The first requirement points to *distributed* databases. The second requirement points away from simple key-value stores and towards DBMS that support secondary indexes and efficient joins. It is easier to rely on a standalone database that provides out-of-the-box consistency and access control, but for completeness's sake it should be noted that it is possible to build a distributed data store from simple, non-distributed, components. For example GekkoFS [18] and Mochi's SDSKV [14] are built on top of key-value store libraries designed to be embedded in other applications such as RocksDB and BerkeleyDB. Rqlite leverages SQLite and the Raft protocol for replication.

In this comparison, three types of databases will be considered: traditional relational (SQL) databases, NoSQL stores and the newer NewSQL databases [17]. All three offer different tradeoffs regarding (ACID) transactions, scalability, availability and query capabilities.

Relational databases In the traditional relational space, some databases are optimized for operational workload (*online transaction processing*, OLTP) that are row-oriented: a typical query fetches a small number of rows using the primary key and updates some of the fields. Other databases are optimized for analytical workload (*online analytical processing*, OLAP), more column-oriented: a typical query computes an aggregation for one column over all the rows in the database. Relational databases are associated with the rich SQL language, a data schema enforced at write time and strong ACID transactions. If they provide replication at all it is usually passive (primary-secondary) and done via log-shipping. Partitioning is not a common feature, meaning that vertical scalability (i.e using more powerful hardware) is usually favoured. Some examples include:

- For OLTP workload: PostgreSQL, MariaDB, SQLite (in-process), Oracle (proprietary).
- For OLAP workload: MonetDB, LucidDB (abandoned since 2014).

NoSQL databases 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 mostly restrict themselves to weaker form of consistency, such as *eventual consistency*. These databases introduced new ad-hoc query languages and favour insertion performance over queries efficiency.

NoSQL databases can be sorted according to their data model, by increasing order of complexity:

- Key-value: Dynamo (proprietary from Amazon), Redis (in memory), Apache Ignite.
- Wide-column: BigTable (proprietary from Google), Cassandra, Scylla, Hbase.
- Document: MongoDB, CouchDB.
- Graph: Neo4j, ArangoDB.

Note that this classification is oversimplified, Apache Ignite also provides an SQL interface on top of the key-value store, ArangoDB is really multimodel and offers both document and graph data models.

NewSQL databases The NewSQL databases seek to combine the advantages of the relational and NoSQL databases, they offer an attractive alternative to developers seeking horizontal scalability and fault tolerance without sacrificing consistency, ACID transactions and, to a lesser extent, the familiar SQL language. The NewSQL systems also promise to bridge the gap between OLTP and OLAP workload, allowing fast ingestion and easy relational queries. Example include Google Spanner, CockroachDB and FaunaDB (proprietary).

Table 3.2 compares a few databases from the SQL, NoSQL and NewSQL classes. Here are a few notes related to the systems described in the table.

SQLite SQLite is a library that implements a self-contained relational database engine. It is not supposed to be distributed, so there can be no consistency violation. However, any distributed system built on top of SQLite will need to handle consistency which represents a considerable amount of work to be done correctly. Other non-SQL in-process databases include BerkeleyDB, LevelDB and RocksDB.

Postgresql PostgreSQL is a mature relational DBMS. It offers ACID transaction but defaults to the *read commited* isolation level, while allowing higher isolation levels if necessary. PostgreSQL is not distributed and only offers passive (primary-secondary) replication for fault tolerance purpose, so again no consistency problem are to be expected if a client only interacts with the leader. Note that Citus develops an extension to distribute PostgreSQL tables across several nodes.

Redis Redis is less a database than an in-memory data-structure store. It has been used extensively as a shared cache and queue. It does offer high-availability through passive (primary-secondary) replication. Apache Ignite is similarly marketed as a cache, and offers an SQL interface on top of a key-value store.

Mongodb MongoDB is a document database. It offer horizontal scaling and high availability via partition and replication, but only supports a partial form of join, and queries using secondary indexes can be inefficient in highly distributed deployment. It is thus optimized for data model where all the information can be entirely contained in a document (i.e denormalized) and indexes restricted to a partition. The isolation and consistency

	SQLite	PostgreSQL	Redis	MongoDB	Cassandra	CockroachDB
License	Pub. domain	BSD-like	3-cl. BSD	Hybrid	Apache 2.0	Hybrid
First release	2000	1996	2009	2009	2008	2015
Latest release	2021-06	2021-08	2021-07	2021-07	2021-09	2021-06
Language	C	C	C	C++	Java	Go
Inspired by	/	/	/	/	DynamoDB	Spanner
Usage	In-process	Standalone	Standalone	Standalone	Standalone	Standalone
Data model	Relational	Relational	Key-value	Document	Wide-column	Relational
Query lang.	SQL	SQL	Ad-hoc	JS	SQL-like	SQL
Schema	On-write	On-write	/	On-read	On-write	On-write
Storage/Index	B-tree	B-tree	/	Hybrid	LSM-Tree	LSM-Tree
Replication	No	Passive	Leader	Leader	Leaderless	Leader
Partitioning	No	No	Yes	Yes	Yes	Yes
Scalability	No	Limited	Good	Good	High	High
Joins	Yes	Yes	/	Partial	No	Yes
2ndary index	Yes	Yes	/	Yes	Partial	Yes
Transactions	Serializable	Read commited	Single-obj.	Weak	No	Serializable
Consistency	/	/	Weak	Weak	Weak	Strong
Optimized for		OLTP	In-memory		Writes	OLTP
C API	Native	Yes	Yes	Yes	Yes	Yes

Table 3.2: Comparison of different data stores in rough order of complexity: SQLite, a self-contained SQL library; PostgreSQL, a full-featured relational database; Redis, a distributable in-memory data store; MongoDB, a document database; Cassandra, a highly distributed key-value store; and CockroachDB, a distributed relational database.

guarantee of MongoDB have improved since its inception, but the default settings still lead to various consistency and isolation violation (see e.g [11]). We note that the situation can be improved with careful tuning, at the cost of reduced performance. MongoDB is free to use, but the license adds conditions when run and sold as a service (SaaS) [8].

Cassandra Cassandra is an example of the distributed *column-family* databases based on Google BigTable which pioneered the idea of improving locality and thus query performance by grouping related columns together. Cassandra also uses a leaderless replication scheme reminiscent of DynamoDB [5] allowing for high write throughput. It is highly scalable and can be replicated, but does not offer advanced query capability across partitions (like MongoDB, limited secondary indexes and no aggregates), nor ACID transactions. Scylla is a rewrite of Cassandra in C++ that claims performance improvements. Apache Hbase is another alternative with similar characteristics, but that relies on the HDFS file system and the external Zookeeper coordination service.

CockroachDB CockroachDB is a distributed relational database, it can be queried with SQL (or at least a subset of it), and is compatible with the PostgreSQL wire protocol. It is based on Google Spanner [2], but lacks its atomic and GPS synchronised clocks, so it cannot provide the same guarantee of strict consistency

Algorithm 1: Application life-cycle when executed within ADMIRE framework. 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: Malleability Manager: providing the initial schedule solution
 5: Slurm: resource allocation
 6: I/O Scheduler: ad hoc storage initialization
 7: I/O Scheduler: transfer of required input datasets (optional)
 8: Slurm: job execution
 9: Slurm: job execution notification
10: Intelligent Controller: creation of the IC components related to the application
11: Intelligent Controller: application communication channel creation
12: ADMIRE Application Manager: dynamic user hints notification
13: Intelligent Controller: application monitoring activation
14: while application is running do
15:     Monitoring Manager: monitoring metric collection
16:     Monitoring Manager: application performance indicator update
17:     Intelligent Controller: application performance model update
18:     Intelligent Controller: system-wide performance forecast update
19:     Intelligent Controller: system status notification
20:     Malleability Manager: application and I/O malleable decision
21:     Intelligent Controller: application notification
22:     I/O Scheduler: background transfers of datasets (optional)
23: end while
24: Slurm: application termination
25: Intelligent Controller: application termination notification
26: Intelligent Controller: application monitoring deactivation
27: Intelligent Controller: application removal
28: I/O Scheduler: transfer of required output datasets (optional)
29: I/O Scheduler: ad hoc storage termination (optional)
```

(i.e linearizability) [10]. It does however provide serializable transactions and allows complex queries while maintaining good scalability. CockroachDB licenses newer versions under MariaDB Business Source License [3], which grants free use of the software but prohibits selling it in a SaaS fashion.

As a final remark we note that all these databases are designed for “cloud” hardware (commodity machines using IP networks) and do not take advantage of the highly parallel HPC hardware and fast interconnects. It appears that some research is being led in this area: see for example [1, 12, 19].

3.5 Intelligent controller workflow

Algorithm 1 describes the different lines and components involved in the execution of an application within ADMIRE framework. 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 of the jobs 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 Malleability Manager provides an initial schedule solution based on the application characteristics.
- After a certain amount of time (during which the job is queued), in lines 5 to 8, 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 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 9 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[3] located in the IC's database.
- The IC creates in line 10 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 11, the ADMIRE Application Manager associated to 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 12 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 13 the IC communicates with the ADMIRE Application Manager to activate the application monitoring.
- During the application execution, the Sensing and Profiling component (line 15) collects: the application performance metrics; other metrics from the ad-hoc storage related to the application; back-end storage performance metrics. These metrics are stored in the Monitoring Manager database.
- In line 16 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 17 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). This model is stored in the Application Status Table located in the IC's database.
- In line 18 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 19 all the previous information (forecasts, performance models and System Status Table) are sent to the Malleability Manager and the I/O scheduler in order 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.

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

- In line 20 the Malleability Manager may generate a reconfiguration commands addressed to the application or to its related ad-hoc storage. Note that these commands involve the creation or destruction of application processes/threads for the former case, or storage nodes for the latter one. The commands are sent to the IC and then are forwarded to the other components that may be involved in the reconfiguration (applications, I/O scheduler, Slurm, etc.). The IC may optionally override these actions after taking into account other factors not considered by the Malleability Manager.

- In line 21 the IC provides different kinds of information to the running application. This information includes the platform status -that can be used by the application to guide local optimization techniques- and processing rules and commands that will permit to execute filters and in-situ/in-transit processing algorithms on the application side.

- In line 22 the I/O Scheduler may optionally transfer datasets asynchronously to persistent storage if requested by the application or the Ad hoc Storage instance. Again, QoS constraints for the application are enforced.

- When the application terminates (line 24), Slurm sends a notification to the IC.

- In line 25, a notification of the application termination is sent to the Malleability Manager, the I/O scheduler and the Monitoring Manager.

- In line 26 the IC notifies the application termination to the Monitoring Manager and deactivates (via the application ADMIRE application controller) the application monitoring.

- In line 27 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 28, the I/O Scheduler executes any data transfers to the back-end storage system requested by the application/Ad hoc Storage, ensuring that QoS constraints for the application are enforced.

- Finally, in line 29 the Ad hoc Storage System instance is destroyed unless other jobs in the Slurm queue have need of it.

Chapter 4

Application Programming Interface

4.1 API description

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, 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.

4.2 Interface of Malleability Manager

There are two communication channels between the Malleability Manager and the Intelligent Controller. The first one (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 (function ADM_suggestScheduleSolution, ID2 in

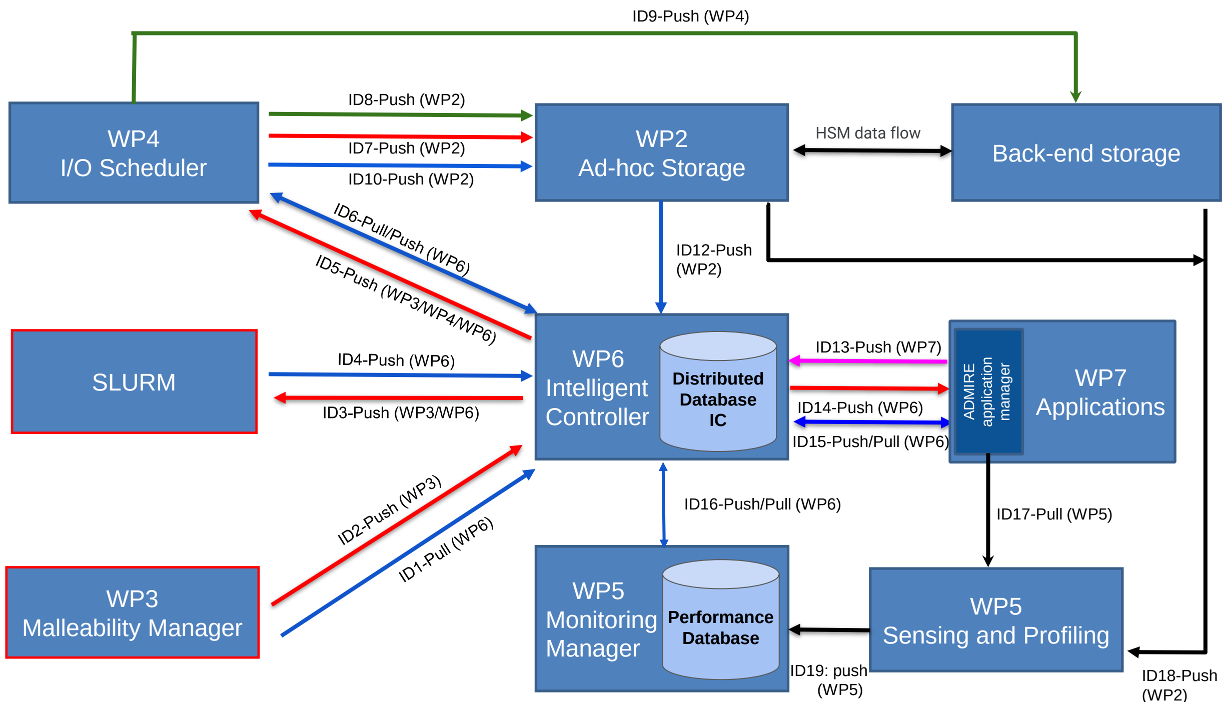

Figure 4.1: ADMIRE control flow between the components.

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.

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 and back-end storage systems. Information include the list nodes involved in each ad-hoc storage, the I/O load and different I/O performance metrics.
- `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.

4.3 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 creation/removal 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)

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.

4.4 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, ID4 in Figure 4.1) is the scheduling solution obtained by the Malleability Manager, that 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 (ID5 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.

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

- `adhocStorageID`. Ad-hoc storage subjected to be reconfigured.
- `jobSchedule`. I/O scheduling policy for the application.
- `assignedBandwidth`. Assigned I/O bandwidth for the considered application.
- `ahocStorageNodes`. List of compute nodes involved in the ad-hoc storage reconfiguration.
- `ad-hocStorageMalleabilityRequest`. 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 4.2 for more details about this interface).

4.5 User interface of Ad-hoc Storage

There is a single communication channel between the ad-hoc storage and the Intelligent Controller (function ADM_IOStatus, ID 12 in Figure 4.1). It is designed to provide the Intelligent Controller with an updated status information of the Ad-hoc Storage. Note that the ad-hoc storage is subjected to carry out malleable actions as well as to change performance metrics (available bandwidth, QoS, etc.) according to the Malleable Manager and the I/O scheduler policies. Every time that these is a change in the AD-hoc Storage configuration, the Intelligent Controller receives a notification by means of this channel. Deliverable D2.1 depicts the interface of this function.

4.6 User 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 involves the creation/destruction of application threads (functions ADM_SpawnThread and ADM_RemoveThread, ID 4 in Figure 4.1) or processes (functions ADM_SpawnProcess and ADM_RemoveProcess, ID 4 in Figure 4.1). The monitoring commands include the activation / deactivation of the application monitoring service (function ADM_MonitoringService, ID 4 in Figure 4.1)

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 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.

4.7 User 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.

4.7.1 User interface for I/O patterns identification

The interface described in this section allows HPC applications 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 updated some times 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.

4.7.2 User 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 applications 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.

4.7.3 User interface for collecting system information

The ADMIRE-enabled application may ask the Intelligent Controller to receive information related to some pre-defined 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.

4.8 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 4.2 for more details about this interface).
Description:
This function is the same as the one described in the interface with Malleability Manager

4.9 UML diagram

Figure 4.2 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.

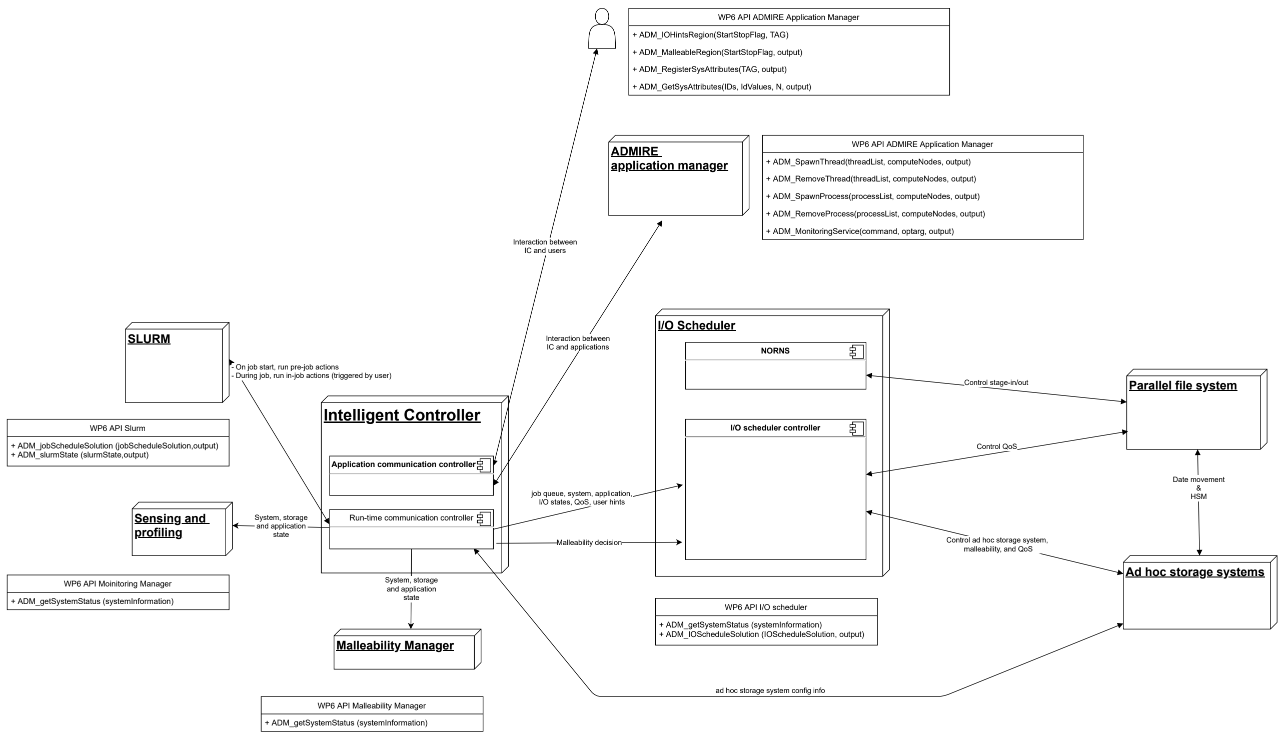
WP6 API ADMIRE Application Manager
+ ADM_IOHintsRegion(StartStopFlag, TAG)
+ ADM_MalleableRegion(StartStopFlag, output)
+ ADM_RegisterSysAttributes(TAG, output)
+ ADM_GetSysAttributes(IDs, IdValues, N, output)
ADMIRE application manager
WP6 API ADMIRE Application Manager
+ ADM_SpawnThread(threadList, computeNodes, output)
+ ADM_RemoveThread(threadList, computeNodes, output)
+ ADM_SpawnProcess(processList, computeNodes, output)
+ ADM_RemoveProcess(processList, computeNodes, output)
+ ADM_MonitoringService(command, optarg, output)
Interaction between IC and users
Interaction between IC and applications
SLURM
- On job start, run pre-job actions
- During job, run in-job actions (triggered by user)
WP6 API Slurm
+ ADM_jobScheduleSolution (jobScheduleSolution,output)
+ ADM_slurmState (slurmState,output)
Intelligent Controller
Application communication controller
Run-time communication controller
Sensing and profiling
System, storage and application state
WP6 API Moinitoring Manager
+ ADM_getSystemStatus (systemInformation)
System, storage and application state
Malleability Manager
WP6 API Malleability Manager
+ ADM_getSystemStatus (systemInformation)
job queue, system, application, I/O states, QoS, user hints
Malleability decision
I/O Scheduler
NORNS
I/O scheduler controller
WP6 API I/O scheduler
+ ADM_getSystemStatus (systemInformation)
+ ADM_IOScheduleSolution (IOScheduleSolution, output)
Control stage-in/out
Control QoS
Control ad hoc storage system, malleability, and QoS
Parallel file system
Date movement & HSM
Ad hoc storage systems
ad hoc storage system config info

Figure 4.2: UML diagram related to WP6 API. The Sensing and profiling component also contains the Monitoring Manager.

Chapter 5

Conclusion

This deliverable covered the Intelligent Controller component that will integrate cross-layer data for providing a holistic view of the system for making intelligent analysis. Also, this deliverable described 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.

From the studies and thework made to elaborate this deliverable, we have arrived to several important design decisions, such as having a distributed Intelligent Controller, using RPCs as communications mechanisms, or using a distributed database to decouple the IC with monitoring and profiling tools.

Appendix A

A comparison between Mercury and Mochi Margo

A.1 Introduction

It was decided that different components of the ADMIRE project would communicate using remote procedure calls (RPCs), a paradigm where local calls are executed on remote resources. The NFS filesystem for example makes heavy use of RPCs. Google gRPC and Apache Thrift are famous frameworks that support this paradigm. The contenders for the project are Mercury [16][1] and Margo [14]. Mercury is an RPC library specifically designed for high-performance computing (HPC) systems. Margo is a C library built on top of Mercury and the Argobots [15] user-level threading framework. Both Mercury and Margo are actively developed under the umbrella of the Mochi project[2], a collaboration between Argonne National Laboratory, Los Alamos National Laboratory, Carnegie Mellon University, and the HDF Group.

Next, we report the results of investigations into these two libraries.

A.2 Mercury

A.2.1 Architecture

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 remote direct memory access (RDMA) on HPC networking hardware that support it.

[1]See also the good project documentation at mercury-hpc.github.io

[2]https://mochi.readthedocs.io/en/latest/

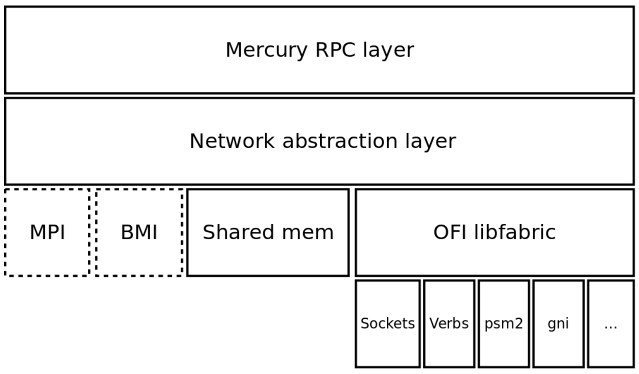

Figure A.1: Mercury architecture. The RPC layer relies on the network abstraction layer and its plugins for transport. The main plugins are OFI libfabric and shared memory, BMI and MPI are legacy plugins. Adapted from [16].

From the bottom-up (see figure A.1), 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 [7]. 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.

A.2.2 High-level RPC layer

Mercury also provides a set of macros based on the BOOST preprocessor to ease the generation of serialisation/deserialisation functions of RPC arguments (this is the *high-level RPC layer* in the Mercury documentation). This significantly reduces the amount of boilerplate code needed for simple functions.

For example, an RPC (named `acc`) that takes an integer as argument would simply require the following macro

```
MERCURY_GEN_PROC(sum_in_t, ((int32_t)(x))
                           ((int32_t)(y)))
```

that will produce an input structure named `sum_in_t` and a generic function for serialisation/deserialisation:

```
typedef struct {
    int32_t x;
    int32_t y;
} sum_in_t;

static hg_return_t hg_proc_sum_in_t(hg_proc_t proc, void *data) {
  hg_return_t ret = HG_SUCCESS;
  sum_in_t *struct_data = (sum_in_t *)data;

  ret = hg_proc_int32_t(proc, &struct_data->x);

  ret = hg_proc_int32_t(proc, &struct_data->y);

  return ret;
}
```

Complex structs, consisting of pointers for example, can also be serialised, at the cost of writing the serialisation/deserialisation function explicitely. Mercury also offers the possibility to use the XDR [6] data serialisation format, but it is disabled by default because, to quote [16] ‘practically all HPC systems are homogeneous’.

Note that this means that the native style of Mercury is to have one function per RPC, not a couple of high-level function (e.g PUT, GET) with the semantic of the call passed as argument, although the second style certainly is possible as well.

A.2.3 Mercury usage

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 A.2.

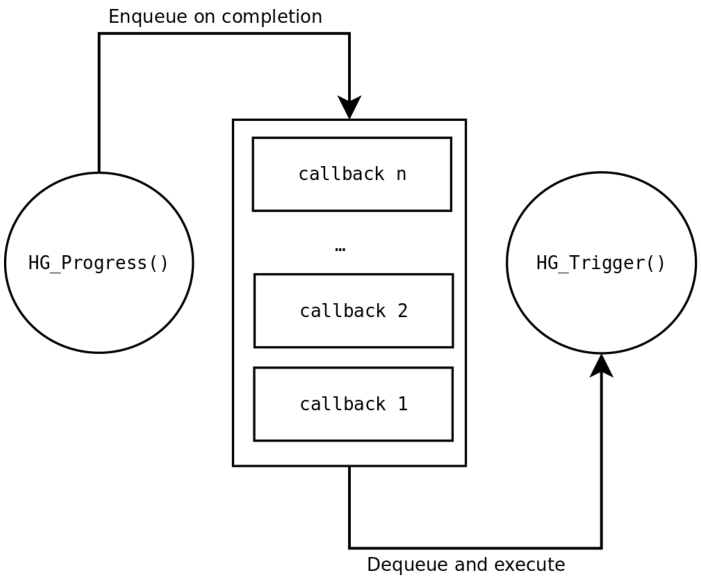

Figure A.2: Mercury progress model. Both client and server have a completion queue of pending callbacks. `HG_Progress` makes explicit progress on network activity and on completion enqueues the corresponding callback. `Hg_Trigger` dequeues callbacks and executes them. Adapted from [16].

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.

   ```
   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);
```

A.3 The Mochi project

The Mochi project [14] aims to provide a framework for building data services for HPC environment out of a small set of core building blocks that include Mercury and Margo. More complex microservices built from this core are also maintained by the project. Margo is introduced first, and some example services are presented next.

A.3.1 Margo

Margo is actually based on Mercury and the Argobots [15] user-level threading library. Its goal is to wrap Mercury callback model into a simpler, more linear programming model using Argobots user-level threads (ULTs). Internally, ULTs are used to invoke Mercury non-blocking functions, suspended while network operations are in progress, thus saving resources, and resumed when the operations complete. This means the programmer does note have to handle 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 A.2.3), as shown partially below for the client code needed to create and forward an RPC. Importantly, no callback need to be defined, and no explicit progress loop 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);
```

A.3.2 Mercury and Margo latency comparison

The main concern using Margo is that it will add some overhead compared to raw Mercury. Given that communication between ADMIRE modules will mainly consist of small messages, tests were devised to compare the latency of a simple RPC that accumulate an integer from the client into a counter on the server.

The tests were run on Inria's PlaFRIM cluster, with the client and the server on separate nodes interconnected with both an Intel OmniPath network and a Ethernet network. The RPC is repeated 100000 times, and the mean time over 10 runs is reported in table A.1.

Network provider	Mercury (s/RPC)	Margo (s/RPC)
ofi+psm2	6.21e-5	5.01e-5
ofi+tcp;ofi_rxm	8.20e-5	9.55e-5
ofi+sockets	7.54e-5	2.08e-2 (!)

Table A.1: Latency of simple RPCs for Mercury and Margo, using different network providers.

A few things to note: first, the order of magnitude is almost always the same, whether using the native OmniPath fabric or an ethernet network, with raw Mercury or Margo. The sole exception is the sockets provider when used with Margo where the latency increases more than 250 times. This is due to an inefficiency in Margo implementation of the progress loop with this provider, and goes to show that progress loops are hard to design well.

Surprisingly, Margo actually performs better than raw Mercury with the psm2 provider. This is probably due to the fact that in this case, the progress loop implementation is more efficient. These results show that Margo, when used with the right provider, does not introduce overhead and can abstract away a difficult part of Mercury design.

A.3.3 Example of services based on Mochi components

The Mochi project also maintains the scalable service groups (SSG), a microservice based on Margo. It allows managing groups of Mercury processes using the SWIM protocol [4] to detect failures. This service could probably be used as a primitive for collective communication.

SDSKV is a higher-level implementation of a distributed key-value database based on Margo and node-local key-value stores (BerkeleyDB or LevelDB). It uses a simple partitioning scheme to distribute the keys amongst the Mercury servers.

GekkoFS [18] is a burst-buffer filesystem for HPC applications based on Margo. It uses a partioning scheme similar to SDSKV. It is developed by members of the ADMIRE project.

A.4 Conclusion

First a word of warning: it appears that some libfabric versions are known to cause issue, this was independently confirmed by the GekkoFS and Mochi team, so this is something that should be carefully tested.

Table A.2 presents a summary comparison between Mercury and Margo. Because Margo inherits the capabilities of Mercury, the two are very similar. The main selling point of Margo is the ease of programming when compared to Mercury, which has been confirmed by the GekkoFS developers who have first-hand experience with these libraries. The Mochi project also offers some interesting building blocks such as SSG that we could use for collective communication. This investigation is left for future work.

	Mercury	Mochi Margo
Portability	Good, thanks to libfabric	idem
Performance	Good	idem, no overhead
Project activity	Very active	idem
Documentation	Good, with examples	idem
Fault tolerance	No but primitive exists	idem
Dependencies	libfabric	Mercury, Argobots, json-c
Programming style	callback based	easier, more linear
Collective comm.	No	No but primitive exists

Table A.2: Summary of Mercury and Margo characteristics.

Appendix B

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] Claude Barthels, G. Alonso, and Torsten Hoefler. Designing databases for future high-performance networks. *IEEE Data Eng. Bull.*, 40:15–26, 2017.

[2] J. Corbett, J. Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. Furman, S. Ghemawat, Andrey Gubarev, Christopher Heiser, Peter H. Hochschild, Wilson C. Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, S. Melnik, David Mwaura, D. Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, M. Szymaniak, Christopher Taylor, Ruth Wang, and D. Woodford. Spanner: Google's globally-distributed database. *ACM Trans. Comput. Syst.*, 31(3):1–22, 2013.

[3] MariaDB corporation Ab. Business source license 1.1, 2017. Last retrieved 2021-09-21.

[4] A. Das, I. Gupta, and A. Motivala. SWIM: scalable weakly-consistent infection-style process group membership protocol. In *Proceedings International Conference on Dependable Systems and Networks*, pages 303–312, 2002.

[5] Giuseppe deCandia, Deniz Hastorun, M. Jampani, Gunavardhan Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, Peter Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. In *SOSP*, 2007.

[6] Ed. M. Eisler. XDR: External data representation standard. RFC 4506, Internet Requests for Comments, 05 2006.

[7] 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.

[8] MongoDB Inc. Server side public license 1, oct 2018. Last retrieved 2021-09-21.

[9] 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.

[10] Kyle Kingsbury. CockroachDB beta-20160829. Technical report, Jepsen, feb 2017.

[11] Kyle Kingsbury. MongoDB 4.2.6. Technical report, Jepsen, may 2020.

[12] Xiaoyi Lu, D. Shankar, and D. Panda. Scalable and distributed key-value store-based data management using RDMA-Memcached. *IEEE Data Eng. Bull.*, 40:50–61, 2017.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] P. Valduriez, R. Jiménez-Peris, and M. Özsu. Distributed database systems: The case for NewSQL. *Trans. Large Scale Data Knowl. Centered Syst.*, 48:1–15, 2021.

[18] Marc-André Vef, Nafiseh Moti, Tim SÃMÃ, Markus Tacke, Tommaso Tocci, Ramon Nou, Alberto Miranda, Toni Cortes, and André Brinkmann. Gekkofs: A temporary burst buffer file system for HPC applications. *Journal of Computer Science and Technology*, 35:72–91, 01 2020.

[19] Erfan Zamanian, Xiangyao Yu, M. Stonebraker, and Tim Kraska. Rethinking database high availability with RDMA networks. *Proc. VLDB Endow.*, 12:1637–1650, 2019.