H2020-JTI-EuroHPC-2019-1

Project no. 956748

ADAPTIVE MULTI-TIER INTELLIGENT DATA MANAGER FOR EXASCALE

D4.1
I/O Scheduler requirements and API

Version 1.4

Date: September 28, 2021

Type: Deliverable
WP number: WP4

Editor: Alberto Miranda
Institution: BSC

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	06/05/21	Ramon Nou	BSC	Structure and kickstart.
3	08/06/21	Marc Vef	JGU	Modified structure. Added job I/O activity information to API.
4	17/08/21	Yi Ju	MPCDF	Added content for in-situ API.
5	20/08/21	Alberto Miranda	BSC	Revised Executive Summary and Introduction.
6	30/08/21	Ramon Nou	BSC	Added content for NORNS.
7	31/08/21	Alberto Miranda	BSC	Integrate contributions from partners into broader I/O Scheduler chapter.
8	01/09/21	Hamid Fard	TUDA	Contributed to Malleability, Architecture and API sections.
9	03/09/21	Alberto Miranda	BSC	Editing and reordering.
10	06/09/21	Marc Vef	JGU	Added section for Slurm user interface.
11	07/09/21	Alberto Miranda	BSC	Editing. Section on in transit data operations.
12	12/09/21	Alberto Miranda	BSC	Editing API section.
13	13/09/21	Jesus Carretero	UC3M	Extended information on ADMIRE architecture and control flow.
14	14/09/21	Marc Vef	JGU	Added more content to Slurm user interface.
15	14/09/21	Ramon Nou	BSC	Added glossary.
16	15/09/21	Massimo Torquati	CINI	Added the possibility to specify streaming executions of in-situ transformations.
17	15/09/21	Jean-Thomas Acquaviva	DDN	Provided context for QoS and related APIs.
18	15/09/21	Alberto Miranda	BSC	First draft complete.
19	21/09/21	Hamid Fard	TUDA	Revision of the documents.
20	23/09/21	Javier Garcia-Blas	UC3M	Revision of the document.
21	25/09/21	Ariel Oleksiak	PSNC	Revision of the document.
22	27/09/21	Marc Vef	JGU	WP4 UML.
23	28/09/21	Alberto Miranda	BSC	Final version.

Executive Summary

The I/O Scheduler is a component in ADMIRE whose primary responsibility is controlling the movement of datasets between different storage backends, with the goal of reducing the overall I/O contention of an HPC cluster. These *data scheduling decisions* must take into account the particular mix of workloads currently in execution so that they respond to the needs of the cluster. To capture these needs, the I/O Scheduler must establish suitable communication channels with all 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.

As an example, the Job Scheduler could allow users of the system to provide a list describing the datasets consumed and produced by a job, which would be considered by the I/O Scheduler to schedule the required data transfers appropriately. Similarly, the Intelligent Controller could provide monitoring information about the state of the system, which would allow the I/O Scheduler to enforce QoS constraints for applications, ad-hoc storage systems and the network, thus reducing I/O congestion in the process. Finally, applications could directly ask the I/O Scheduler to transfer datasets from the PFS to node-local storage (or vice versa), and the I/O Scheduler would queue these transfers and decide the best moment to execute them.

To achieve this level of communication and the I/O Scheduler primary goals, it is very important that a flexible enough API is provided that allows capturing the I/O requirements derived from these interactions and transform them into useful work. Thus, this deliverable describes the work done during Tasks 4.1 and 4.3 to capture these requirements, it also proposes an initial version of the I/O Scheduler API to support these interactions as well as the necessary Slurm user interface.

Contents

1. Introduction

Traditionally, the design of large-scale HPC infrastructures has been focused on maximising parallel processing power while reducing energy consumption. Nevertheless, the advent of data-intensive computing applications in recent years, including high-performance data analytics (HPDA) and deep learning (DL), as well a huge increase in the data requirements in scientific computing is changing this compute-centric view. As a result, there is a growing agreement in the HPC community that large-scale supercomputers will act as key data processing nodes in the emerging distributed computing environment integrating HPC and data-intensive computing resources [4, 14].

This growing demand for data processing is accompanied by disruptive technological progress of the underlying storage technologies. As a result, upcoming exascale HPC systems are transitioning from a simple HPC storage architecture, consisting of a parallel backend file system and archives often based on tapes, towards a multi-tier storage hierarchy that includes node-local non-volatile main memory (NVMM) with a performance close to DRAM, NVMe-based SSDs inside compute nodes with a bandwidth of many GBytes/s, SSDs on I/O nodes, parallel file systems, campaign storage, and archival storage (see Figure 1.1). Unfortunately, there is a significant lack of appropriate interfaces for managing this I/O stack, which means that accessing the different storage tiers is not part of any scheduling decision. Thus, data transfers between the different storage tiers are managed explicitly by users, which leads to uncoordinated accesses to storage, redundant data movement, increased energy consumption, and delayed end-to-end performance. Furthermore, backend storage systems can be easily flooded by a few compute nodes performing many concurrent data or metadata requests. This means that it is necessary to control the applications' data and metadata flows by setting (and enforcing) QoS constraints that can prevent a single application from overloading the shared file system, while guaranteeing a predictable performance at the same time.

To address these issues, WP4 will develop an I/O Scheduler component that will coordinate and schedule the data flows between the shared backend parallel file system and the ad hoc storage systems, executing data transfers as necessary according to the information gathered by other components in the ADMIRE framework.

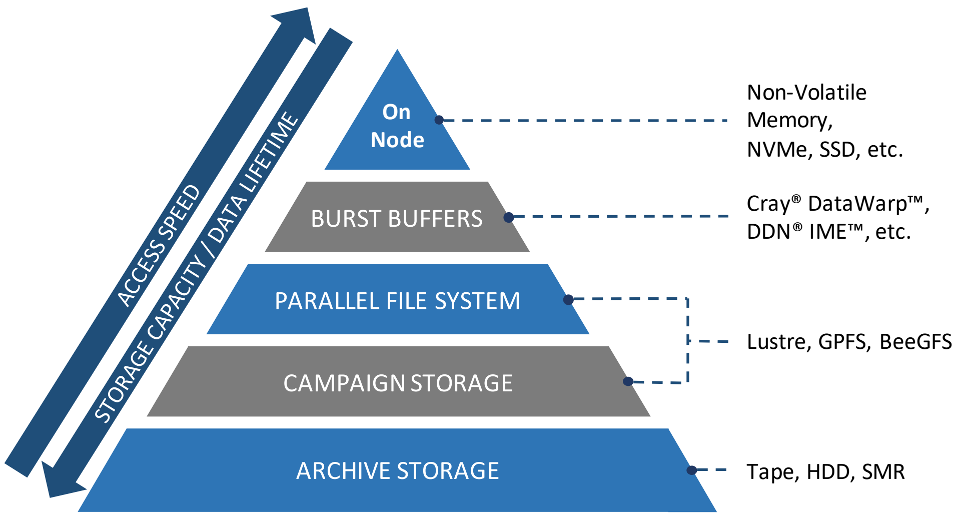

Figure 1.1: The complexity of the storage hierarchy in modern HPC systems.

2. The I/O Scheduler

WP4 will develop an I/O Scheduler with control point support that will coordinate direct and indirect inputs from the Intelligent Controller, the Job Scheduler, and the Malleability Manager to provide QoS-aware data scheduling. Functionalities to support in-situ/in-transit data transformations will be provided, and using low-power processors for such tasks will be researched. Given these requirements, and in order to provide context for Section 3, this section describes the principal responsibilities and functionalities of the I/O Scheduler, as well as its functioning context as defined in a co-design process with the other WPs.

2.1 General responsibilities

The main responsibility of the I/O Scheduler component is to control (and often execute) the movement of *datasets* (i.e. files or objects) between storage tiers with the goals of accelerating data processing by maintaining data locality as well as reducing I/O contention to the PFS. This kind of contention typically occurs due to HPC applications using the PFS for their normal I/O operation (e.g. reading configuration files, reading input data, writing results, checking file attributes, checkpointing opertions, etc.). Even if these operations often follow fairly well defined patterns *per application*, such patterns end up diluted when considering all applications as a whole and the PFS perception is that it is receiving a random mix of unrelated I/O operations that are very difficult to optimise. The I/O Scheduler will address this by appropriately scheduling data staging into Ad hoc Storage Systems, combined with defining and enforcing QoS constraints per application. Data staging changes PFS I/O from a random mix of unrelated operations to well-defined sequential streams of read-only or write-only operations, while keeping normal application I/O constrained within node-local storage. QoS enforcement makes it so that a single application cannot overwhelm the PFS and its I/O capabilities can be distributed fairly.

More concretely, the I/O Scheduler implementation offered by ADMIRE has the following requirements:

1. The I/O Scheduler needs to take into account how data is going to be accessed, and give a higher priority to data shared by several compute nodes in order to reduce contention as much as possible.

2. The I/O Scheduler must offer mechanisms to support the definition and enforcement of QoS constraints for the I/O of an application. This means that it must be possible to set limits on e.g. the amount of metadata operations, the number of I/Os per second (IOPS), the bandwidth consumed w.r.t. the shared PFS, or even the network congestion it generates.

3. The I/O Scheduler must support multiple storage tiers, including at least the backend parallel file system, node-specific flash storage, and Storage Class Memory attached to the CPUs' memory buses. It will also allow scheduling bulk data transfers between the different storage tiers, and will manage fine-grained accesses from one storage tier to another, following the aforementioned QoS guarantees.

4. The I/O Scheduler (possibly in cooperation with the Job Scheduler) must offer clients the possibility of executing arbitrary *in situ*/*in transit* codes on a job's input or output data. For instance, data transformation plugins may be included into the I/O workflow so that they are applied while a batch job input data is transferred to the selected compute nodes, or so that its output results are compressed in-place prior to staging them out to the shared backend storage system. *In situ* processing further allows data-intensive analysis tasks to co-exist and be co-scheduled with simulation tasks.

5. The I/O Scheduler must include several scheduling policies with different levels of aggressiveness, given that applications may benefit differently from each policy.

Thus, we consider that I/O scheduling is needed in scenarios where normal I/O usage ends up causing contention and causes HPC applications to run for longer than they would have if they had run in isolation. Additionally, it was also decided that the I/O Scheduler will be responsible for executing all tasks related to starting up and tearing down an Ad hoc Storage System in response to requests by the Job Scheduler. The rationale behind this decision is that since Ad hoc Storage Systems are ephemeral in nature (i.e. they live as long as the job/jobs that need them), they typically must to be started before a job starts and they also need to be kept running after it ends. This is required, firstly, so that it is possible to stage data into them. Secondly, the Ad hoc Storage System may be kept running for subsequent jobs in a workflow. And thirdly, it may also be necessary to complete any pending data transfers from the Ad hoc Storage System to the PFS once all jobs have completed. Since the I/O Scheduler will have first hand knowledge of when these transfer tasks complete, and will also have facilities to deploy and run code on-demand, it is simpler to let it take care of this.

2.2 ADMIRE architecture and information flow

In order to fulfil the aforementioned responsibilities, the I/O Scheduler must collaborate with other components in the ADMIRE framework. To clarify these interactions, Figure 2.1 depicts how the consortium currently envisions the flow of information in the framework, and the place the I/O Scheduler has in it. The HPC storage subsystem is represented on the upper part of the figure and consists of the Ad hoc and Backend storage systems. The former provides each application a specialised high-performance storage tier tailored to the application's characteristics, while the latter represents the main parallel file system used by the HPC platform (e.g. Lustre, GPFS, etc.). Both storage tiers are coordinated by the I/O scheduler, 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. In order to make appropriate I/O scheduling decisions, the I/O Scheduler requires information about both the current state of the HPC system as well as the expected I/O that an application is going to perform. As shown by the figure, information about an application comes primarily from the Job Scheduler, the Malleability Manager, and the applications themselves:

- The Job Scheduler (e.g. Slurm) will be the front-end component with which the end users of the framework will interact in order to run their applications. As such, its main responsibilities in ADMIRE will be to keep track of the current workload of the system (e.g. *job queue state* in the figure), as well as to collect any *user hints* about application I/O and datasets usage that can help improve the overall performance of the system.

- The Malleability Manager (WP3) is a component in the ADMIRE framework that is responsible for providing elasticity in the allocation of resources for a job. As such, it will be responsible for making decisions that imply dynamically altering the amount of resources assigned to a job, which may heavily affect the proposed I/O scheduling solutions.

- Applications themselves will also be able to communicate additional information about an application I/O by making use of the APIs provided by the framework. This information can be really valuable for the framework since it will come from application developers themselves rather than from end users. Nevertheless, note that obtaining this information will require applications to be modified to use APIs, which means that the ADMIRE framework cannot rely on having this information always available in order to function properly.

Additionally, both applications and storage tiers are monitored by the Sensing and Profiling (WP5) component, which is responsible for collecting system-wide performance metrics at node level that will be processed and aggregated by the Monitoring Manager (WP5) in order to produce performance models for applications and storage tiers. All this information will be collected by the Intelligent Controller (WP6) that will use it to predict potential performance bottlenecks in the system. This means that the main source of information for

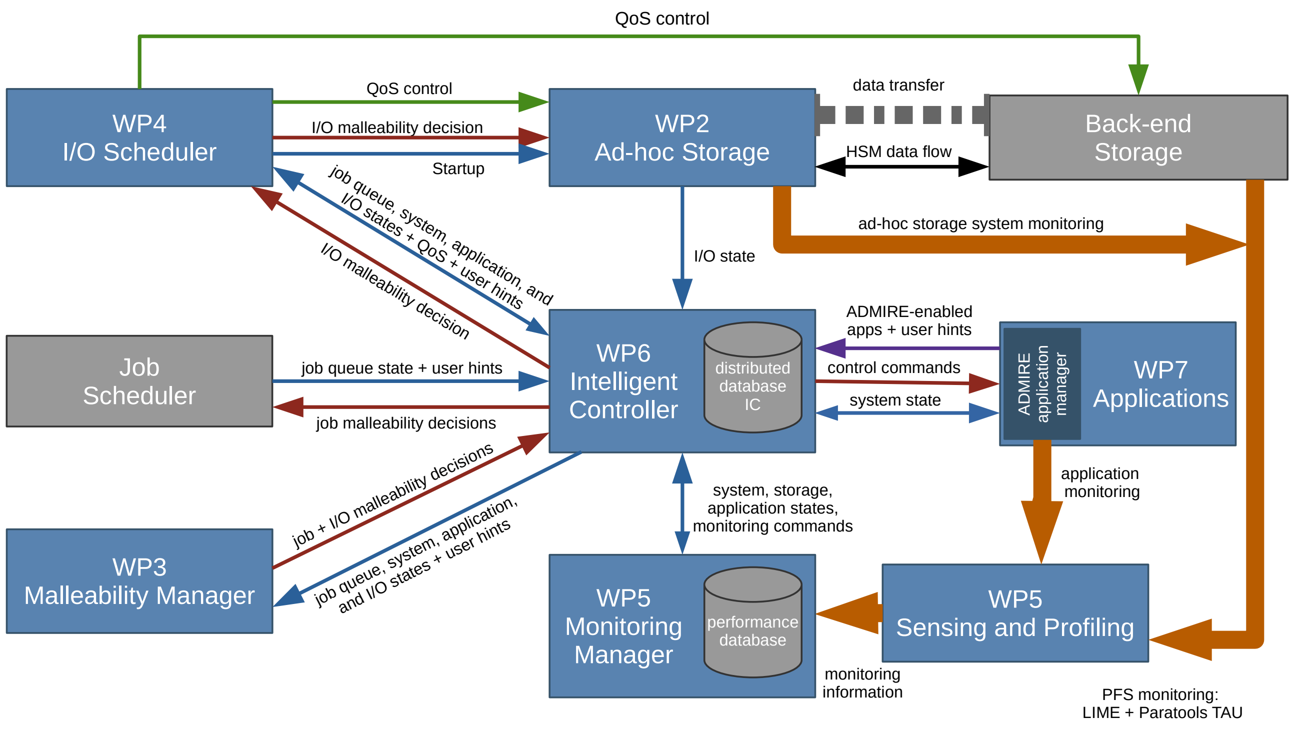
QoS control
WP4
I/O Scheduler
QoS control
I/O malleability decision
Startup
WP2
Ad-hoc Storage
data transfer
HSM data flow
Back-end
Storage
job queue, system, application, and I/O states + QoS + user hints
I/O malleability decision
ad-hoc storage system monitoring
I/O state
Job
Scheduler
job queue state + user hints
job malleability decisions
WP6
Intelligent
Controller
distributed database IC
ADMIRE-enabled apps + user hints
control commands
system state
ADMIRE application manager
WP7
Applications
job + I/O malleability decisions
job queue, system, application, and I/O states + user hints
system, storage, application states, monitoring commands
application monitoring
WP3
Malleability Manager
WP5
Monitoring
Manager
performance database
WP5
Sensing and Profiling
monitoring information
PFS monitoring:
LIME + Paratools TAU

Figure 2.1: Information flow within the ADMIRE framework. The Intelligent Controller acts as the nexus point for collecting, processing, and distributing all the information required for the proper functioning of the ADMIRE.

the I/O Scheduler to perform its tasks will be the Intelligent Controller. 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.

2.3 Specific requirements

In order for the I/O Scheduler to reduce congestion to the PFS and enable a better usage of the HPC cluster's I/O stack, it needs to be able to capture information from end users I/O usage, as well as to offer a set of particular functionalities to successfully play its part in the ADMIRE framework. This section describes these requirements in detail.

2.3.1 I/O hints

As mentioned, clients of the ADMIRE framework must be able to define the I/O requirements for their applications either when requesting resource allocations from the Job Scheduler or via the ADMIRE APIs in ADMIRE-enabled applications. The I/O Scheduler must then expose an API that allows the materialisation of I/O hints in the Job Scheduler. As part of the ADMIRE project, WP4 will ensure that the following I/O parameters can be communicated to the I/O Scheduler:

- **Stage-in datasets:** This parameter will allow users of the framework to precisely identify which datasets an HPC job will consume in order to produce meaningful results. This is helpful to the I/O Scheduler in order to kickstart the transfer process from the PFS into the desired Ad hoc Storage System, so that data can be on location before the job starts.
- **Stage-out datasets:** This parameter will allow users of the framework to precisely identify the expected datasets generated from an HPC job, as well as the location in the PFS where they should be stored.

This will be helpful to the I/O Scheduler in several ways: firstly, it will allow detecting dependencies between producer-consumer jobs; secondly, intermediate datasets produced by the job in an Ad hoc Storage System that had not been tagged for PFS storage will be considered as temporary by default, being deleted when the job completes; thirdly, it will allow the I/O Scheduler to determine what to do with the produced datasets, i.e., transfer them to the PFS for persistent storage or keep them in place/move them to other compute nodes for subsequent job executions.

- **Storage tiers:** The previous two arguments will include information that allows users to define the appropriate storage tiers where datasets reside.
- **Data distribution pattern:** This parameter will allow users of the framework to define how a dataset should be distributed between the requested compute nodes (e.g. one-to-one, one-to-many, many-to-many), as well as whether the dataset's data should be replicated in each target or spliced following a distribution pattern. This will allow the I/O Scheduler to choose the most efficient method for transferring data to its intended destination.
- **I/O access patterns and access mode:** This will allow users to define if a dataset is going to be used in a read-only, write-only, read-write, or read-modify-write manner. Additionally, it will also be possible to define whether a dataset is intended to be privately consumed or shared by several applications in a workflow.

2.3.2 QoS constraints

The goal of Quality of Service in ADMIRE's I/O performance management is to throttle I/O performance to prevent the excessive allocation of I/O resources to a single process/group of processes, with the purpose of avoiding diminished performance due to starvation for the remaining active process in the system. This interpretation of QoS remains less ambitious than a more strict definition of QoS which is to guarantee a level of performance or share of I/O resources to a given process or group of processes. As mentioned, the management of Resources in Job Schedulers (both Slurm or PBS) typically take care of a fair allocation of system resources such as computation and memory but typically exclude the I/O [18]. This situation is mitigated by the ability of some storage systems to cap the available capacity per user or per group/project using quotas, but this feature misses the control of the data and metadata traffic during a time frame.

This situation has been acknowledged by the Lustre community and a more complete QoS implementation is under development for it. This implementation, often referred as LIME [23], is based on the notion of a Token Bucket Filter (TBF). The key idea of TBF is to control the rate of network service from the storage servers based on the client's identity. As Lustre is a network attached file system, all demands both in terms of data and metadata are expressed by RPCs to storage servers from compute clients. Nevertheless, the lack of orchestration remains as one of the main difficulties for delivering QoS for a PFS. All traffic, both data and metadata, is dispatched between multiple servers, which means that each server only has a partial knowledge of the consumption of resources. This will be addressed in ADMIRE with WP7's Intelligent Controller. The panoptic vision of the system state offered by the Intelligent Controller allows implementing efficient QoS policies. With a centralised controller, it is possible to leverage existing throttling mechanisms in Lustre or other PFSs in order to guarantee a level of resource to specific processes. Thus, in order to implement these mechanisms effectively, it important for the I/O Scheduler to capture the QoS classes and constraints defined by the Intelligent Controller in order to 1) convey them to the Ad hoc Storage Systems, and 2) enforce them when transferring data between tiers.

Note that we consider that the target of a QoS class/constraint can be a dataset, a compute node or an application. We currently envision that the following QoS classes will need to be supported:

- **I/O Data Rate:** The data rate QoS class allows restricting the amount of data being transferred between storage devices (including RAM) during a certain unit of time. This QoS class allows setting upper limits to the I/O bandwidth used by an application, which is intended to prevent an application with high data production from saturating the I/O channel, which would reduce the effective I/O rate experienced by other applications, thus hindering overall system performance.

- **I/O Operation Rate:** The operation rate QoS class allows restricting the amount of operations per unit of time being emitted. This, for instance, allows restricting the metadata IOPs emitted by an application, which is often the cause of increased I/O latency for HPC applications.
- **Network Data Rate:** This QoS class is similar to the I/O Data Rate class, but applied to the network resource rather than to I/O. This class allows restricting the amount of network bandwidth used by a compute node, or an application for data transfers.

Though this is the minimum set of classes that the I/O Scheduler needs to perform its work effectively, the API we propose in Section 3.4 has been designed in a generic manner, so that the implementation can be easily extended with future QoS classes without having to alter the API.

2.3.3 Asynchronous data movement

As described, the I/O Scheduler is responsible for executing and tracking data transfer tasks to move datasets between storage tiers. For efficiency's sake, it makes sense to perform these data transfers in an asynchronous manner, so that the initiators (i.e. applications and other ADMIRE components) don't need to wait for the transfers to be completed and may carry on with other tasks. To this end, ADMIRE will integrate a data transfer service for HPC clusters called NORNS [15] that already fulfils most of the necessary requirements.

NORNS[1] is an infrastructure service developed by BSC that allows moving data between different storage backends in a one-to-one fashion, including file-to-file data transfers via POSIX I/O, as well as file-to-memory and memory-to-memory transfers via RDMA. It was developed in the context of the NEXTGenIO [2] European project to provide a middleware to Slurm for the orchestration of asynchronous data transfers between the different storage layers in an HPC cluster. Its design goals are the following:

- To simplify the management of the increasingly heterogeneous I/O stack by offering interfaces for abstracting and controlling the different storage tiers.
- To hide the complexity of each specific tier by providing a unified API for asynchronously transferring data, so that NORNS clients (Slurm and applications in NEXTGenIO) do not need to bother with the technical details to execute such transfers efficiently.
- To allow clients to start, monitor, and manage transfers between storage tiers, so that it is possible to control and account such transfers.
- To execute data transfers as efficiently as possible, taking advantage of fast interconnects and native APIs where available.

NORNS provides facilities to system administrators to expose the cluster's storage architecture to end users and applications, and also offers APIs for creating and monitoring asynchronous data transfers between these local and remote layers. With an appropriate Slurm job description file, the user can create I/O tasks to stage-in/stage-out data from/to the PFS and into another storage tier (e.g. an Ad hoc Storage System in ADMIRE) when the workflow starts/ends, keep persistent data on node-local storage to feed upcoming phases or move data directly between compute nodes to match future job schedules.

Since its main goal is to keep track of any data staging required to run a job, its architecture has been designed to be tightly coupled with job schedulers. As shown in Figure 2.2, the NORNS service is currently composed of the following components:

- A resource control daemon (`urd`) which runs at each compute node. Its responsibilities are to coordinate with any administrative clients (e.g. the Job Scheduler or the Intelligent Controller) to manage and track the storage tiers spaces defined for each job. It is also the component in charge of actually accepting, validating, executing, and monitoring all I/O tasks affecting this compute node.

[1] `https://storage.bsc.es/gitlab/hpc/norns`
[2] `https://www.nextgenio.eu`

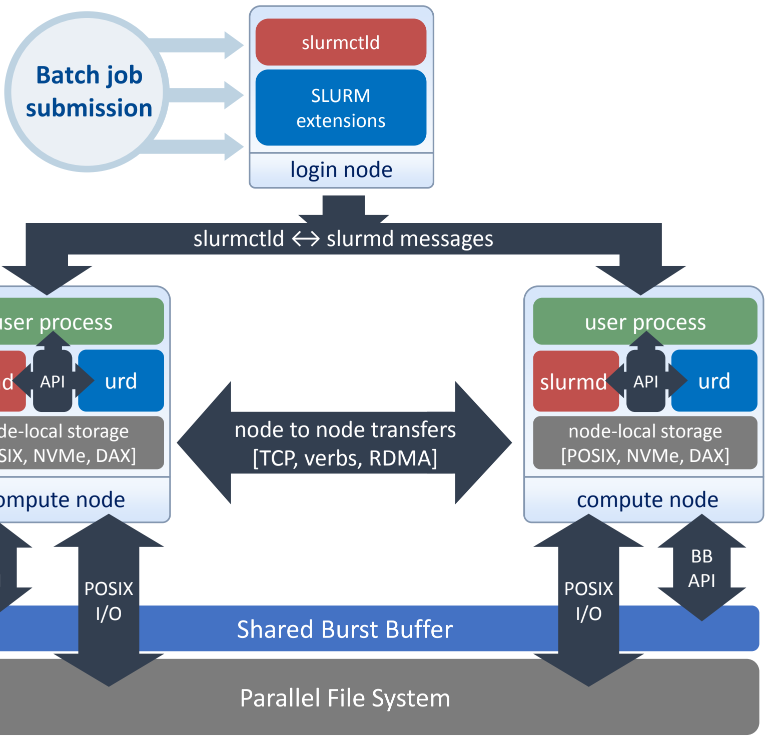

Figure 2.2: NORNS service architecture and component interaction.

- A shared library `libnornsctl.so` implementing a control API (`nornsctl`). This API offers administrative interfaces so that administrative clients can control the transfer daemon, query its state, define the appropriate storage tiers accessible by each job, and submit/control I/O tasks.
- A shared library `libnorns.so` implementing a user-level `norns` API. This API allows parallel applications to query information about the storage tiers defined for them, and also offers interfaces so that they can submit and control I/O tasks between such tiers.

Another advantage of the service is that it is built around the ANL's Mercury[3], a highly scalable RPC library that offers a common interface for several HPC fabrics such as TCP, InfiniBand, Verbs, or Omni-Path. NORNS, thus, provides a solid framework for the development of scalable data-aware scheduling algorithms to arbitrate data transfers and it makes sense to use it as the workhorse for the I/O Scheduler, as well as to leverage its APIs in the definition of the APIs proposed in this deliverable.

2.3.4 *In situ* and *in transit* data operations

ADMIRE's I/O Scheduler must also include facilities that allow the execution of *in situ* and *in transit* transformations on data, and its API must reflect this. Traditionally, data visualisation, analysis and transformation has been *post hoc* processed, i.e., simulations save data through the I/O system to permanent storage and the visualisation, analysis or transformation programs load this data, again using the I/O system. With *in situ* processing, the load on the I/O system can be drastically reduced, as data ends up being processed in the same compute nodes where it is generated, without having to involve any I/O tiers for permanent storage. With *in transit* processing, however, operations are applied on data when a simulation transfers it over the network to a separate set of visualisation nodes for processing.

While the *in transit* model may seem as less efficient when compared to the *in situ* model, this kind of data processing allows having a set of dedicated visualisation nodes on the same machine as the simulation, which allows visualisation routines to run concurrently with simulation code, but not impacting its runtime as with *in situ* methods. Moreover, it has been shown that the effects of storage latency can be hidden by streaming data

[3] `https://mochi.readthedocs.io`

writes to the separate allocation of staging nodes, and letting them run while the simulation continues [1,16,17]. Nevertheless, before *in transit* processing can take place, the data must be sent from the simulation to a resource for processing. This data dump can saturate the network, and could even cause a slowdown in the simulation while it sends the data off over the network. Even so, this data dump has the potential to end up moving far less data, in total vs. using an *in situ* approach for data processing. The reason is that visualisation allocations are typically much smaller than simulation node allocations, meaning that communication takes place over a much smaller domain.

Furthermore, the *in situ* paradigm is constrained to use the concurrency of the allocated resource, which typically means the allocation for the entire simulation. This level of concurrency can actually be a bottleneck for data processing routines that require significant communication (e.g. particle tracking, etc), or algorithms that don't scale up to the level of simulation codes (e.g. hundreds of thousands of cores). Conversely, in a *in transit* paradigm, the concurrency of the data processing resource can be appropriately configured for the tasks to be performed. Algorithms that require significant synchronisation and communication will generally perform much better at lower levels of concurrency, and this can be used to optimise the performance. Thus, it is not clear whether one model is superior to the other and it makes sense to support both in ADMIRE for different tasks.

2.3.4.1 Requirements

With the increasing interest in this kind of processing, a number of frameworks have been developed, and thus it is sensible to reuse their concepts in the I/O Scheduler API, or even integrate one of such frameworks in ADMIRE if at all possible. Interestingly, frameworks for *in situ*/*in transit* data processing can be classified along the six axes depicted by Figure 2.3 depending on how they function, and hence it makes sense to use the same axes to describe the kind of data processing framework that we want the I/O Scheduler to support:

1. The *integration type* axis refers to how visualisation and analysis routines are integrated into the simulation code. In most implementations, the simulation code is aware of the integration and makes calls in support of data marshalling, but it is also possible to integrate *in situ* routines without the simulation being aware (e.g. application-aware vs. application-unaware).

2. The *proximity type* axis characterises how close the visualisation and analysis routines are to the data (e.g. on-node vs. off-node).

3. The *access* axis refers to how the simulation makes data available to visualisation and analysis routines. The main options are direct access (where *in situ* routines share the same logical memory space as simulation code) and indirect access (where *in situ* routines run in a distinct logical memory space from simulation code).

4. The *division of execution* axis considers how compute resources are shared between simulation and *in situ* routines. The two main options are space division (a subset of compute resources is exclusively dedicated to *in situ* routines) and time division (some or all compute resources alternate between advancing the simulation and visualisation).

5. The *operation controls* axis describes the mechanism for selecting which operations are executed during run-time (e.g. automatic vs. human-in-the-loop).

6. The *output type* axis describes which operations the framework performs on simulation data before it is output. Three major categories can be inferred for this axis: subset (a subset of the output data is selected, and the rest is discarded), transform (one or more operations are performed on each element of the data), and derived (operations generate new data of a different nature than the input).

Taking these six axes into consideration, the I/O Scheduler features for *in situ*/*in transit* processing that the consortium wants to achieve in ADMIRE, as well as the rationale for each decision, are summarised by Table 2.1.

	Features	**Rationale**
Integration Type	application-aware, application-unaware	The *in situ* facilities provided by the I/O Scheduler must be offered through the ADMIRE APIs for ADMIRE-enabled applications. Nevertheless, legacy applications that don't make explicit use of the APIs should still be able to benefit from the functionalities to a certain extent through the extensions for the Job Scheduler.
Proximity	on-node, off-ode	In order to reduce contention as much as possible, the I/O Scheduler will execute *in situ* routines in the same nodes where the data is generated. For *in transit* models, routines may be executed in either source or destination nodes.
Access	direct access	Direct access is preferred since on-node execution is more aligned with ADMIRE objectives.
Division of Execution	time, space	Dedicating subsets of compute nodes for data processing routines or using all compute nodes to alternate between simulation and data processing are both acceptable strategies for ADMIRE. During the implementation phase it will be evaluated whether it is possible to support both within the scope of the project.
Operation Controls	automatic	For performance sake, how and when data processing routines are executed should be a decision made by the ADMIRE framework.
Output Type	subset, transform, derived	At this moment it doesn't seem to make sense to limit the scope of what data processing routines could do with the data. Thus all options should be supported.

Table 2.1: Desired features for the data processing framework in ADMIRE.

2.3.4.2 Existing frameworks

While designing the API presented in Section 3.5, several existing *in situ* data processing frameworks were considered, as integrating an already established and well known API would facilitate the adoption of the ADMIRE framework. In the following, we describe the frameworks considered for integration, some of which are more generic in nature than others:

- **VisIt (+ LibSim):** VisIt [7, 22] is an open source visualisation and analysis software for end users. VisIt is designed to work as a distributed system: it has a server that utilises parallel compute capabilities coupled with the client running as the user interface. In addition, VisIt can run *in situ* via the LibSim library, enabling simulation code to transfer the data (in VTK format) to the VisIt Server, which the VisIt Clients then contact to visualise the data. VisIt has been shown to scale effectively to tens of thousands of cores, but it only allows on-node proximity with direct access and is also fairly heavy weight, which may cause problems when performing different types of *in situ* integrations.
- **ParaView (+ Catalyst):** ParaView [2] is another tool for the visualisation of large data, while Catalyst is an adaptor to enable *in situ* data analysis and visualisation. The simulation code passes the data (again in VTK format) through an adaptor built with Catalyst to ParaView to visualise the data. ParaView and Catalyst share the same characteristics as VisIt + LibSim. Also similarly, ParaView is also a heavy weight

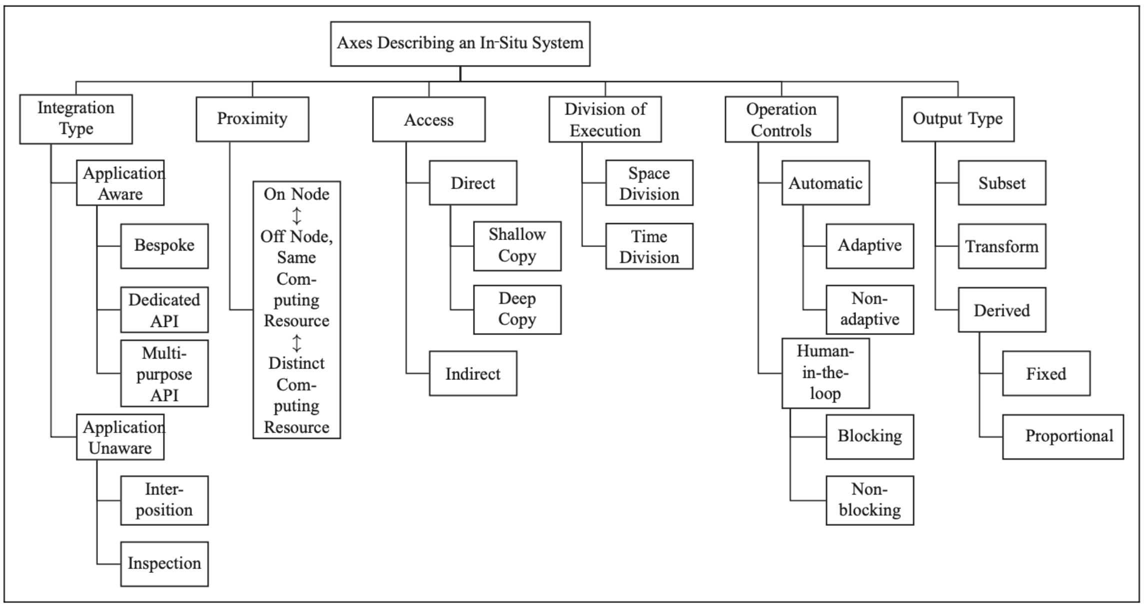

Figure 2.3: Categorisation of *in situ* data processing frameworks. Taken from Bauer's *et al. "A terminology for in situ visualization and analysis systems"* [6].

visualisation tool.

- **SENSEI:** SENSEI [3] is an effort to both streamline the *in situ* instrumentation of scientific code and allow for flexibility in the choice of analysis infrastructure. This flexibility is achieved by offering adaptors to simulation code that allow it to connect to underlying technologies such as VisIt+LibSim or Paraview+Catalyst for visualisation and GLEAN, HDF5 or ADIOS for data staging. SENSEI also offers support for data transfer in VTK format. SENSEI supports proximity on the same computing resource in addition to on-node, both direct and indirect access.

- **Omniverse:** Omniverse [10] is an *in situ* system developed by NVIDIA with the target of allowing and *in situ* visualisation using GPUs. One example is that the ParaView with Catalyst transfers the VTK data through Omniverse Bridge in USD format to Omniverse. It supports both on-node and off-node access.

- **Damaris/Viz:** Damaris/Viz [8] is an *in situ* framework built on top of MPI that uses a client-server pattern to enable data processing. Damaris/Viz was developed with the goals of having low impact on simulation runtime, low impact for in situ integration, and high adaptability. The data formats allowed in Damaris are VTK and HDF5. Damaris/Viz supports both direct and indirect accesses, but does not support distinct remote computing resources. It can operate in either an *in situ* approach, utilising a subset of cores on each simulation node, or *in transit*, by using a dedicated set of visualisation nodes.

- **ALPINE:** ALPINE [11] is an *in situ* infrastructure using the proxy pattern. Like Damaris, ALPINE also supports VTK and HDF5 data formats. It only supports on-node proximity and direct access.

- **GLEAN:** GLEAN [21] is a non-intrusive framework for real time data analysis and I/O acceleration that uses the proxy pattern to allow *in situ* processing, which allows it to be be integrated with the simulation unaware of the *in situ* processing. GLEAN allows for custom data analyses to be performed on both the compute and staging resources, mitigating the overall data saved to disk and improving application performance. In transit workflows are supported when GLEAN asynchronously moves simulation data to a separate allocation of staging nodes though standard I/O libraries like HDF5.

- **ADIOS I & II:** The Adaptable I/O System (ADIOS) [9, 12, 13] provides the highest level of synchronous I/O performance. ADIOS is a componentisation of the I/O layer used by high-end simulations and/or for high-end scientific data management, providing an easy-to-use programming interface. ADIOS abstracts the API away from implementation, allowing users to compose their applications without detailed knowledge of the underlying software and hardware stack. ADIOS I uses the layer pattern and ADIOS II uses the factory pattern to allow *in situ* analysis. ADIOS II is more flexible since it allows configuration during run time and all also supports all the data types offered by MPI, such as arrays of double-precision floating-point data. To address the growing imbalance between computational capability and I/O performance, ADIOS introduced the concept of data staging where rather than writing data directly to shared backend storage devices, a staging pipeline moves data to a transient location [5]. Since ADIOS supports this location to be on separate physical nodes and/or on memory resources on the same node where data is generated, ADIOS supports both the *in transit* and *in situ* data processing paradigms.

For a generic framework as envisaged in ADMIRE, the support for several proximity and access mechanisms as well as flexibility in the data formats required are important. Interestingly, many of the frameworks considered require specific data formats like the Visualization Toolkit (VTK) format [19], which offer a useful data format for visualisation but typically require data transformation to use it, incurring an unnecessary overhead. VisIt/LibSim, ParaView/Catalyst, and SENSEI are based on VTK pipelines and do not support distinct, remote computing resources proximity, which makes them too specific as a general framework as the one we are targeting in ADMIRE. On the contrary, ADIOS offers generic abstractions to link *in situ* routines with simulation code, supporting distinct computing resources proximity and providing different possibilities for *in situ* processing. It can also handle normal MPI data formats, which allows to use it directly from a simulation without the need for data transformations. ADIOS can further use the I/O system and can stage the data via RDMA or MPI. ADIOS, then, is a good candidate for implementing the core of ADMIRE's data processing framework, and the abstractions used in its API have heavily influenced the ADMIRE interfaces proposed in Section 3.5.

3. The I/O Scheduler API

In ADMIRE, the framework components can ask the I/O Scheduler (via the Intelligent Controller) for the execution of several I/O related actions in the context of a HPC job. These actions can be differentiated in actions that take place before/after the job starts/ends (i.e. pre/post-job actions), and actions taking place during the job run time. In both cases, these actions will be executed by an appropriate target storage system, be it and Ad hoc Storage System (e.g. GekkoFS or Hercules), the shared PFS (e.g. Lustre), or an Object Store (e.g. dataClay). Figure 3.1 shows the control flow between the different ADMIRE components as well as the general IDs for each interface and whether it "pushes" or "pulls" information to/from the recipient.

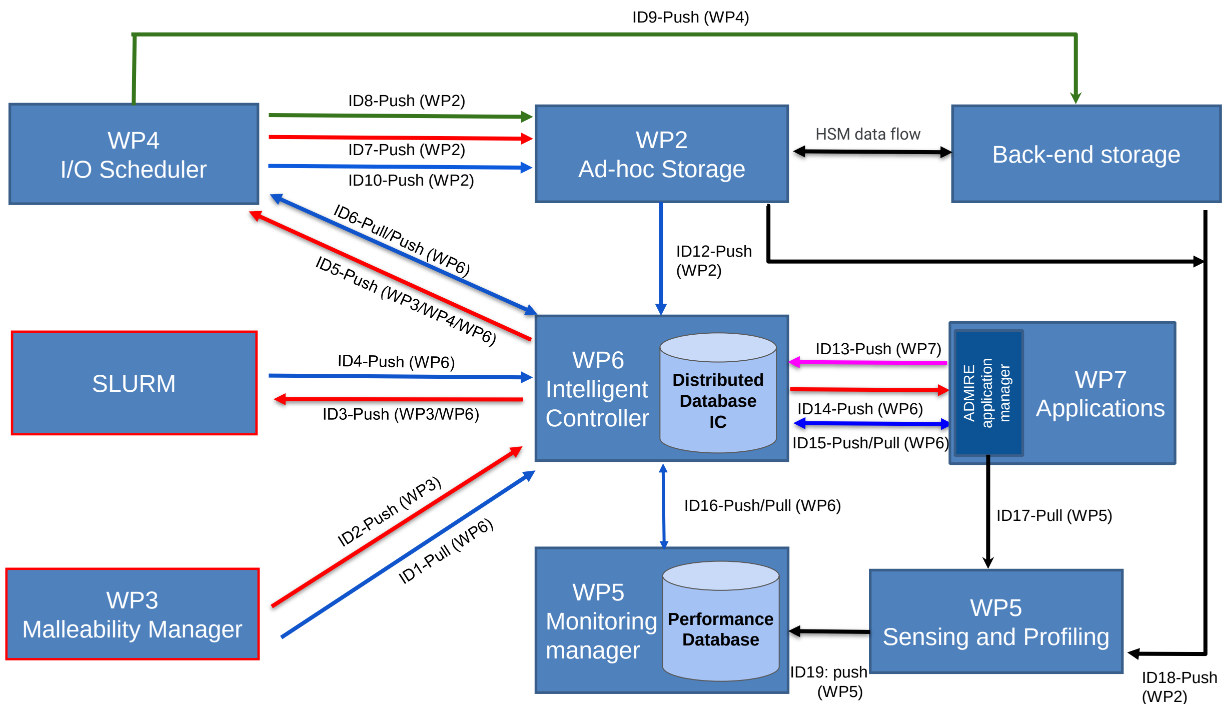

Figure 3.1: ADMIRE control flow between the components.

3.1 Managing storage tiers and naming data resources

While using the facilities provided by the ADMIRE framework, users of the API often need a way to refer to both data resources and the different storage tiers (i.e. the parallel file system, ad hoc storage systems, object stores, local storage, etc.). In order to support this in a flexible manner, the API will support a naming schema that reuses the internal IDs provided by each storage tier (e.g. POSIX paths for file systems and keys for object stores), but adds a specific `Tier_ID` as a colon-separated prefix that identifies the storage tier from which data comes from (e.g. `lustre`, `gekkofs`, `hercules`, etc.). Using such a naming schema has several advantages:

1. The actual resource name does not change between ADMIRE and the corresponding storage tier. It is

just tagged with additional relevant information that can easily be extracted if needed.

2. Users can refer to data resources within a storage tier without specifically knowing how to access the tier. This is specially important for user-level storage systems that do not always rely on the kernel to setup a real mount point for them, or that even need specific libraries to be `LD_PRELOAD`ed before accessing them.
3. Using Tier IDs allows the ADMIRE framework to dynamically configure the storage tiers visible to an application, by creating namespaces as necessary. This allows job scripts and application code to be written in a generic manner, without users having to concern themselves about where data is coming from.
4. This naming schema has been previously used by well known parallel I/O libraries used in HPC such as ROMIO [20], which links nicely with what the consortium wants to do in ADMIRE.

Furthermore, using a naming schema also allows specifying additional options when referring to data resources, such as a specific range of bytes to transfer for a POSIX file, specific objects in a collection stored in a KV-store, or a multi-dimensional array in an HDF5 file. Therefore whenever there is a need to refer to data resources both in the Slurm interface or when directly invoking functions of the ADMIRE API, expressions following the EBNF grammar below shall be supported:

$$\begin{aligned}
\langle\text{data resource ID}\rangle &::= \langle\text{tier ID}\rangle\ \texttt{':'}\ \langle\text{item ID}\rangle\ [\ \texttt{'@'}\ \langle\text{options}\rangle\ ] \\
\langle\text{tier ID}\rangle &::= \textit{an alphanumeric string assigned by the ADMIRE framework} \\
\langle\text{item ID}\rangle &::= \textit{a POSIX path}\ \mid\ \textit{an object ID} \\
\langle\text{options}\rangle &::= \langle\text{option name}\rangle\ \texttt{'='}\ \langle\text{option value}\rangle
\end{aligned}$$

Note that the hostname may be included in order to differentiate between node-local storage tiers with the same name (e.g. `node42[tmp]:/path/to/file`). The listing below demonstrates some examples, assuming that the ADMIRE framework has previously defined the `lustre`, `gekkofs`, and `dataclay` Tier IDs, respectively, for a running installation of Lustre PFS, an instance of GekkoFS and an instance of dataClay:

```
# referencing a file in the Lustre PFS
lustre:/path/to/file

# referencing a specific range or ranges of a file in GekkoFS
gekkofs:/path/to/file@range=0+4096
gekkofs:/path/to/file@ranges=8192+8192,24576+8192,32768+8192

# referencing a local file at node 84
node84[tmp]:/path/to/file

# accessing a slice from matrix b01_12 in dataClay
dataclay:mathlib.matrix.Block/b01_12@slice([:2, :2])
```

3.2 Slurm user interface

As discussed in Section 2.2, to simplify gathering I/O requirements from end-users, the ADMIRE framework will allow specifying such requirements when asking for resources from the batch scheduler, i.e., Slurm. Therefore, in order to allow this interaction between end-users and the I/O Scheduler, it is necessary to extend the existing Slurm API with the necessary facilities, so that it is possible to invoke them using a *command line* *interface* (CLI)[1]. This would allow users to specify storage tiers, *in transit* and *in situ* data operations, as well as several Ad hoc Storage System options depending on the I/O requirements of their jobs. Thus, Slurm will become the main point of interaction for end-users to provide requirements for ADMIRE's I/O Scheduler. Each Slurm argument is considered a *user hint* and is sent to the Intelligent Controller [ID4-Push] from where it is forwarded to the I/O scheduler [ID6-Push] (see Figure 3.1). This section discusses the interfaces in this Slurm user API in detail.

3.2.1 Specifying locations for data resources

The `ADM_input` and `ADM_output` interfaces (see Interface 3.1 and Interface 3.2, respectively) allow users to specify the datasets that should be transferred between Ad hoc Storage Systems and the shared backend storage system. We can distinguish the following relevant locations:

1. The location for an input dataset in the PFS storage tier.
2. The corresponding location where such dataset should be placed in the target Ad hoc Storage System.
3. The location of a dataset in an Ad hoc Storage System where the output from a simulation is being/has been stored.
4. The corresponding output location in the PFS where such data should be placed after the simulation completes.

Note that items 1 and 2 refer to the stage-in process and need to at least be partially completed before the compute-job starts. Similarly, items 3 and 4 refer to the stage-out process. As discussed in Section 3.1, the corresponding storage tiers are defined by specifying the Tier ID for each path, which allows the ADMIRE framework to determine the specific tiers involved, as well as their particularities. In addition, we define an `ADM_inout` interface (see Interface 3.3) which allows users to easily define input and output paths for use cases in which the output path is equal to the input path. Therefore, the former overwrites the latter.

Interface 3.1: Definition of the Slurm `ADM_input` interface.

Name: *ADM_input*
Control ID: ID4 and ID6 (user hints)
Control Type: Push
Input: An $origin$ location for the source dataset
Input: A $target$ location for the destination dataset
Description:
Specifes the $origin$ *location in a storage tier where input is located, as well as the* $target$ *location where it should be placed in a different storage tier.*
Example: `sbatch --ADM_input="lustre:/some/dir gekkofs:/some/other/dir"`

Interface 3.2: Definition of the Slurm `ADM_output` interface.

Name: *ADM_output*
Control ID: ID4 and ID6 (user hints)
Control Type: Push
Input: An $origin$ location for the source dataset
Input: A $target$ location for the destination dataset
Description:
Specifies the $origin$ *location in a storage tier where output is located, as well as the* $target$ *location where it should be placed in a different storage tier.*
Example: `sbatch --ADM_output="gekkofs:/some/dir lustre:/some/other/dir"`

[1]Which in Slurm also allows including them in a job script.

Interface 3.3: Definition of the Slurm `ADM_inout` interface.

Name: *ADM_inout*
Control ID: ID4 and ID6 (user hints)
Control Type: Push
Input: An $origin$ location for the source dataset
Input: A $target$ location for the destination dataset
Description:
Specifies both the input and output locations in a storage tier. This combines both `ADM_input` and `ADM_output` for user convenience: the input data provided by $origin$ is overwritten by the output data generated at $target$.
Example: `sbatch --ADM_inout="lustre:/some/dir gekkofs:/some/other/dir"`

3.2.2 Options for Ad hoc Storage Systems

The interfaces described in this section allow users to control to a certain extent the behaviour of the target Ad hoc Storage System that will be used to run their HPC applications.

Lifetime and resource allocation: The lifetime of an Ad hoc Storage System is inherently linked to an application lifetime and, hence, to the batch job where it is executing. Nevertheless, it also makes sense to keep an Ad hoc Storage System running to use it for a workflow of interrelated jobs. Furthermore, while in the first case the resources that the Ad hoc Storage System instance uses to do its work will be co-located with the application (i.e. they would share the same set of compute nodes), in the second case the Ad hoc Storage System instance should run in a separate allocation with a separate set of resources (i.e. a separate set of compute nodes) to facilitate reuse. To support these use cases, the `ADM_adhoc_context`, `ADM_adhoc_nodes`, and `ADM_adhoc_walltime` interfaces are provided (Interfaces 3.4, 3.6, and 3.7, respectively).

Thus, the `ADM_adhoc_context` interface offers the following options:

- `in_job:shared`: The Ad hoc Storage System instance shares the lifetime of the job and all its resources.
- `in_job:dedicated`: The Ad hoc Storage System instance shares the lifetime of the job and uses some of its resources in a dedicated manner.
- `separate:new`: The Ad hoc Storage System instance runs independently from the job in a separate set of resources.
- `separate:existing`: The Ad hoc Storage System used is a separate instance setup in a different job. This option requires a valid context identifier to be provided with the `ADM_adhoc_context_id` interface (Interface 3.5).

The number of nodes used for the Ad hoc Storage System will be specified using the `ADM_adhoc_nodes` interface. Additionally, when running in a `separate:new` mode, a *context identifier* will be returned that subsequent jobs may use with the `separate:existing` and the `ADM_adhoc_context_id` interfaces to get access to this Ad hoc Storage System instance (see Interface 3.5). How long this separate instance lives can be controlled with the `ADM_adhoc_walltime` interface (Interface 3.7).

Interface 3.4: Definition of the Slurm `ADM_adhoc_context.` interface

Name: *ADM_adhoc_context*
Control ID: ID4 and ID6 (user hints)
Control Type: Push
Input: A valid $execution\_mode$ describing how the Ad hoc Storage System should behave.
Output: $adhoc\_context\_id$
Description:
Specifies the $execution\_mode$ an Ad hoc Storage System should use. Valid options:

1. `in_job:shared`*: run while sharing the application's compute nodes;*
2. `in_job:dedicated`*: run using a subset of the application's compute nodes;*
3. `separate:new`*: ask the system to allocate a separate job with separate runtime and number of nodes;*
4. `separate:existing`*: ask the system to reuse an already running Ad hoc Storage System instance.*

The number of nodes assigned for the Ad hoc Storage System must be specified with `ADM_adhoc_nodes`. In the `separate:new` $execution\_mode$, the lifetime of the Ad hoc Storage System will be controlled with `ADM_adhoc_walltime`. In the `separate:existing` $execution\_mode$, a valid context ID must be provided with `ADM_adhoc_context_id`.

Example:

```
# full co-location:  40 nodes fully shared
sbatch --nodes=40 --ADM_adhoc_nodes=40 --ADM_adhoc_context=in_job:shared

# partial co-location:  40 nodes for app.  + 20 nodes for I/O
sbatch --nodes=40 --ADM_adhoc_nodes=20 --ADM_adhoc_context=in_job:shared

# partial co-location, dedicated:  20 nodes for app.  + 20 nodes I/O
sbatch --nodes=40 --ADM_adhoc_nodes=20 --ADM_adhoc_context=in_job:dedicated

# separate allocation:  40 nodes for app.  + 20 nodes I/O
#  (Returns Context ID 42)
sbatch --nodes=40 --ADM_adhoc_nodes=20 --ADM_adhoc_context=separate:new

# reuse allocation with ID 42:  30 nodes for app.  + existing 20 nodes I/O
sbatch --nodes=30 --ADM_adhoc_context=separate:existing
       --ADM_adhoc_context_id=42
```

Interface 3.5: Definition of the Slurm `ADM_adhoc_context_id` interface.

Name: *ADM_adhoc_context_id*
Control ID: ID4 and ID6 (user hints)
Control Type: Push
Input: A valid $context\_id$ for a separate instance of an Ad hoc Storage System
Description:
Specifies an existing Ad hoc Storage System to use via its ID.
Example:

```
sbatch --nodes=30 --ADM_adhoc_context=separate:existing
        --ADM_adhoc_context_id=42
```

Interface 3.6: Definition of the Slurm `ADM_adhoc_nodes` interface.

Name: *ADM_adhoc_nodes*
Control ID: ID4 and ID6 (user hints)
Control Type: Push
Input: The desired $number\_of\_nodes$
Description:
Specifies the number of nodes for the Ad hoc Storage System. If the `ADM_adhoc_execution_mode` is `shared`, the number cannot exceed the number of allocated nodes within the compute job. If the `ADM_adhoc_execution_mode` is `dedicated`, the number of nodes is not restricted.
Example: `sbatch --ADM_adhoc_nodes=10`

Interface 3.7: Definition of the Slurm `ADM_adhoc_walltime` interface.

Name: *ADM_adhoc_walltime*
Control ID: ID4 and ID6 (user hints)
Control Type: Push
Input: The desired $walltime$ in minutes
Description:
Specifies for how long the ad hoc storage system should run before should down. Only relevant in the context of the `ADM_adhoc_context` function.
Example:

```
sbatch --nodes=40 --ADM_adhoc_nodes=10 --ADM_adhoc_context=separate:new
       --ADM_adhoc_walltime=60
```

Usage: Knowing the intended usage of an Ad hoc Storage System in advance allows the I/O Scheduler to make assumptions on how data will be used. It may also allow the Storage System itself to perform some optimisations that would not be possible without this information. The `ADM_adhoc_access` interface (Interface 3.8) allows users to provide this information from the Slurm command line. Note that this is just a hint on how the file system will be used and though it will be passed along to the Ad hoc Storage System instance, the ADMIRE framework will not enforce this usage.

Interface 3.8: Definition of the Slurm `ADM_adhoc_acccess` interface.

Name: *ADM_adhoc_access*
Control ID: ID4 and ID6 (user hints)
Control Type: Push
Input: The desired $access$ method
Description:
Specifies access to the ad hoc storage system: `write-only`, `read-only`, `read-write`. Cannot be used when using an existing Ad hoc Storage System instance.
Example: `sbatch --ADM_adhoc_access=read-only`

Data distribution: Instead of always using a fixed pre-defined data distribution, some Ad hoc Storage Systems allow users to choose between several data distributions that might be more beneficial performance-wise for their applications. The `ADM_adhoc_distribution` interface (Interface 3.9) allows choosing this option and passing it to the Ad hoc Storage System instance required by the application.

Interface 3.9: Definition of the Slurm `ADM_adhoc_distribution` interface.

Name: *ADM_adhoc_distribution*
Control ID: ID4 and ID6 (user hints)
Control Type: Push
Input: The desired $data\ distribution$
Description:
Specifies the data distribution within the ad hoc storage system, e.g., `wide-striping`, `local`, `local-data-global-metadata`.
Example: `sbatch --ADM_adhoc_distribution=wide-striping`

Background flush: The `ADM_adhoc_background_flush` interface (see Interface 3.10) allows users to enable the background movement of data from a specified output location in an Ad hoc Storage System to the shared PFS while the application is running. Note that this option may not be supported by all Ad hoc Storage Systems, in which case an error will be returned. Also note that this option requires an output target location in the PFS to have previously been defined using the interfaces presented in Section 3.2.1.

Interface 3.10: Definition of the Slurm `ADM_adhoc_background_flush` interface.

Name: *ADM_adhoc_background_flush*
Control ID: ID4 and ID6 (user hints)
Control Type: Push
Input: A $boolean$ enabling or disabling the option
Description:
Specifies if data in the output location should be moved to the shared backend storage system in the background (default `false`).
Example: `sbatch --ADM_adhoc_background_flush=true`

3.2.3 *In situ/in transit* data operations

These Slurm interfaces allow users to specify that they need *in situ* or *in transit* data operations to be performed on datasets, and provide a configuration file describing such operations (see Interfaces 3.11 and 3.12, respectively). The actual format of the configuration file will depend on how ADMIRE's *in situ/in transit* framework ends up being implemented, but will allow applying the APIs defined in Section 3.5.

Interface 3.11: Definition of the Slurm `ADM_in_situ_ops` interface.

Name: *ADM_in_situ_ops*
Control ID: ID4 and ID6 (user hints)
Control Type: Push
Input: A $path$ to the configuration file
Description:
In situ data operations specified in a given configuration file.

Interface 3.12: Definition of the Slurm `ADM_in_transit_ops` interface.

Name: *ADM_in_transit_ops*
Control ID: ID4 and ID6 (user hints)
Control Type: Push
Input: A $path$ to the configuration file
Description:
In transit data operations specified in a given configuration file.

3.3 Internal interfaces for ADMIRE components

While the Slurm interface offers the main point of entry for end users to interact with ADMIRE's I/O Scheduler, the internal components of the framework (most notably the Intelligent Controller) also need appropriate interfaces to do so. This section describes such interfaces.

3.3.1 Requesting data transfers

Data transfers are described as the movement of datasets between storage tiers. As discussed in Section 3.1, the dataset IDs will follow a naming schema that will allow determining the appropriate storage tiers involved.

Interface 3.13: Definition of the `ADM_transfer_dataset` internal interface.

Name: *ADM_transfer_dataset*
Control ID: ID5
Control Type: Push/Pull + Async
Input: A $source\_location$ identifying the source dataset/s in the source storage tier
Input: A $destination\_location$ identifying the destination dataset/s in its desired location in a storage tier
Input: A list of $qos\_constraints$ that must be applied to the transfer. These may not exceed the global ones set at node, application, or resource level (see Section 3.4)
Input: A $distribution$ strategy for data (e.g. `one-to-one`, `one-to-many`, `many-to-many`)
Input: A job_id identifying the originating job
Output: A $transfer\_handle$ allowing clients to interact with the transfer (e.g. wait for its completion, query its status, cancel it, etc.
Description:
Transfers the dataset identified by the $source\_name$ *to the storage tier defined by* $destination\_name$*, and apply the provided* $constraints$ *during the transfer. This function returns a handle that can be used to track the operation (i.e., get statistics, or status).*

Note that even if the interface definition presented in Interface 3.13 is intended for internal use, a reduced version will be made available for ADMIRE-enabled applications. This limited interface will allow applications using the API to request data transfers directly, but will not be able to influence the QoS constraints controlling such transfers. Note that a `transfer_handle` is returned by the interface. This $transfer\_handle$ allows clients to further interact with the transfer. We currently envision that it should be possible to wait for an asynchronous transfer to complete, to query its status (i.e. did it complete successfully?) and to cancel it if it has not started yet. For the sake of brevity, we are omitting the definitions for these extra interfaces.

3.3.2 Specifying information about datasets

The `ADM_set_dataset_information` (Interface 3.14) interface allows clients to provide extra information to the I/O Scheduler about data resources. For now, we consider that it is important for the I/O Scheduler to have information about a dataset's lifespan (i.e. `temporary` vs. `persistent`), its intended usage (i.e. `input` vs. `output` vs. `inout`), and how it will be accessed (i.e. `read` vs. `write` vs. `read-write` vs. `read-modify-write`). Nevertheless, since this list may be expanded in the future, the interface is designed in a generic manner with an opaque $info$ parameter. Note that even though this information will arrive via the Intelligent Controller, Slurm is intended to be the main source of this information, since it will be able to process the extra options attached to a data resource providing I/O hints (see Section 3.1).

Interface 3.14: Definition of the `ADM_set_dataset_information` internal interface.

Name: *ADM_set_dataset_information*
Control ID: ID5
Control Type: Push
Input: A $resource\_id$ identifying the dataset of interest
Input: An opaque $info$ argument containing information about the dataset (e.g. its lifespan, access methods, intended usage, etc.)
Input: A job_id identifying the originating job
Output: A $status$ code determining whether the operation was successful
Description:
Sets information for the dataset identified by $resource\_id$.

3.3.3 Malleability

As discussed in D3.1, malleability is the ability to shrink or expand resources on demand, which also includes the resources related to the I/O capabilities of an application. As such, the I/O Scheduler offers the `ADM_set_io_resources` interface (Interface 3.15) to support altering this in a malleable manner. For now, we only envision changing the number of I/O nodes assigned to an Ad hoc Storage System but, nevertheless, the interface receives an opaque parameter `resources` that will allow expanding this requirement.

Interface 3.15: Definition of the `ADM_set_io_resources` internal interface.

Name: *ADM_set_io_resources*
Control ID: ID5
Control Type: Push
Input: A $tier\_id$ specifying the target storage tier
Input: An opaque $resources$ argument containing information about the I/O resources to modify (e.g. number of I/O nodes.)
Input: A job_id identifying the originating job
Output: A $status$ code indicating whether the operation was successful
Description:
Changes the I/O resources used by a storage tier, typically an Ad hoc Storage System.

3.3.4 Influencing pending transfers in the I/O Scheduler

In order to provide more precise control for the Intelligent Controller, the I/O Scheduler offers a set of interfaces that allow influencing how transfers are managed. Interfaces 3.16, 3.17, 3.18, and 3.19 have been designed for this purpose.

Interface 3.16: Definition of the `ADM_get_transfer_priority` internal interface.

Name: *ADM_get_transfer_priority*
Control ID: ID5
Control Type: Pull + Sync
Input: A $transfer\_id$ identifying a pending transfer
Output: The $priority$ of the pending transfer or an error code if it didn't exist or is no longer pending
Description:
Returns the $priority$ *of the pending transfer identified by* $transfer\_id$

Interface 3.17: Definition of the `ADM_set_transfer_priority` internal interface.

Name: *ADM_set_transfer_priority*
Control ID: ID5
Control Type: Push + Sync
Input: A $transfer\_id$ identifying a pending transfer
Input: A positive or negative number n for the number of positions the transfer should go up or down in its scheduling queue
Output: A $status$ code indicating whether the operation was successful
Description:
Moves the operation identified by $transfer\_id$ up or down by n positions in its scheduling queue.

Interface 3.18: Definition of the `ADM_cancel_transfer` internal interface.

Name: *ADM_cancel_transfer*
Control ID: ID5
Control Type: Push + Sync
Input: A $transfer\_id$ identifying a pending transfer
Output: A $status$ code indicating whether the operation was successful
Description:
Cancels the pending transfer identified by $transfer\_id$.

Interface 3.19: Definition of the `ADM_get_pending_transfers` internal interface.

Name: *ADM_get_pending_transfers*
Control ID: ID5
Control Type: Pull + Sync
Output: A list of $pending\_transfers$
Description:
Returns a list of pending transfers. Each operation will include a $transfer\_id$ as well as information about the involved resources and tiers

3.4 QoS constraints

QoS constraints will be provided to the I/O Scheduler by other ADMIRE components (notably the Intelligent Controller), and also need to be conveyed from the I/O Scheduler to the Ad hoc Storage System instances and the PFS. Thus, we define the `ADM_set_qos_constraints` interface for this particular purpose (Interface 3.20) that will be supported by these particular components. Additionally, we also provide the `ADM_set_qos_constraints` for querying purposes (Interface 3.21).

Interface 3.20: Definition of the `ADM_set_qos_constraints` internal interface.

Name: *ADM_set_qos_constraints*
Control ID: ID5/ID8/ID9
Control Type: Push + Sync
Input: The $scope$ it should be applied to: `dataset`, `node`, or `job`
Input: A QoS $class$ (e.g. "badwidth", "iops", etc.)
Input: A valid id for the element that should be constrained, i.e. a resource ID, a node hostname, or a Job ID
Input: An appropriate $value$ for the selected $class$
Output: A status code indicating whether the operation succeeded
Description:
Registers a QoS constraint defined by $class$, $scope$, and $value$ for the element identified by id

Interface 3.21: Definition of the `ADM_set_qos_constraints` internal interface.

Name: *ADM_get_qos_constraints*
Control ID: ID6
Control Type: Pull + Sync
Input: The $scope$ being queried: `dataset`, `node`, or `job`
Input: A valid id for the element of interest, i.e. a resource ID, a node hostname, or a Job ID
Output: A list of QoS constraints that includes all the $classes$ currently defined for the element as well as the $values$ set for them
Description:
Returns a list of QoS constraints defined for an element identified for id

3.5 In-situ/In-transit data operations

The I/O Scheduler needs to provide an API that allows deploying code for both *in situ* and/or *in transit* data operations, but also to define when they will be used. The interfaces `ADM_define_data_operation`, `ADM_connect_data_operation`, and `ADM_finalize_data_operation` fulfil this purpose (see Interfaces 3.22, 3.23, and 3.24). Alternatively, rather than connecting a data operation directly to a data resource, it is possible to link it to a pending data transfer so that it is applied when the transfer is executed. The interface `ADM_link_transfer_to_data_operation` covers this use case (see Interface 3.25). Note that in this case, the operation does not need to be explicitly finalised.

Interface 3.22: Definition of the `ADM_define_data_operation` internal interface.

Name: *ADM_define_data_operation*
Control ID: ID5
Control Type: Push
Input: A valid $path$ for the operation code
Input: A user-defined $operation\_id$ for the operation
Input: A list of $arguments$ for the operation
Output: A $status$ code indicating whether the operation was successful
Description:
Defines a new operation, with the code found in $path$. *The code will be identified by the user-provided* $operation\_id$ *and will accept the* $arguments$ *defined, using the next format "arg0, arg1, arg2, . . . "*

Interface 3.23: Definition of the `ADM_connect_data_operation` internal interface.

Name: *ADM_connect_data_operation*
Control ID: ID5
Control Type: Push
Input: The $operation\_id$ of the operation to be connected
Input: An $input$ data resource for the operation
Output: An $output$ data resource where the result of the operation should be stored
Input: A $stream$ boolean indicating if the operation should be executed in a streaming fashion
Input: The values for the $arguments$ required by the operation
Input: A job_id identifying the originating job
Output: An $operation\_handle$ for the operation that allows clients to further interact with the operation (e.g query its status, cancel it, etc.)
Description:
Connects and starts the data operation defined with $operation\_id$ *and with the* $arguments$, *using the* $input$ *and* $output$ *data storage (i.e., files). If the operation can be executed in a streaming fashion (i.e., it can start even if the input data is not entirely available), the* $stream$ *parameter must be set to* `true`.

Interface 3.24: Definition of the `ADM_finalize_data_operation` internal interface.

Name: *ADM_finalize_data_operation*
Control ID: ID5
Control Type: Push
Input: The $operation\_id$ of the operation to be connected
Output: A $status$ code indicating whether the operation was successful
Description:
Finalises the operation defined with $operation\_id$.

Interface 3.25: Definition of the `ADM_link_transfer_to_data_operation` internal interface.

Name: *ADM_link_transfer_to_data_operation*
Control ID: ID5
Control Type: Push
Input: The $operation\_id$ of the operation to be connected
Input: The $transfer\_id$ of the pending transfer the operation should be linked to
Input: A $stream$ boolean indicating if the operation should be executed in a streaming fashion
Input: The values for the $arguments$ required by the operation
Input: A job_id identifying the originating job
Output: An $operation\_handle$ for the operation that allows clients to further interact with the operation (e.g query its status, cancel it, etc.)
Description:
Links the data operation defined with $operation\_id$ with the pending transfer identified by $transfer\_id$ using the values provided as $arguments$. If the operation can be executed in a streaming fashion (i.e., it can start even if the input data is not entirely available), the $stream$ parameter must be set to `true`.

Similarly to Interface 3.13, some of these interfaces return an `operation_handle` to allow clients further interaction with the operation. We currently envision that it should be possible to query the status of an operation (i.e. did it complete successfully?) and to disconnect it/unlink it if it has not started yet. Again, for the sake of brevity, we are omitting the definitions for these extra interfaces.

3.6 Job I/O activity information

Finally, in order to provide information for the Monitoring Manager in WP5, the I/O Scheduler needs to provide an interface that allows monitoring the I/O behavior on a per-job basis. This includes collecting key metrics, such as information about read and write operations or the number of metadata requests. The `ADM_get_statistics` use case covers this case (see Interface 3.26).

Interface 3.26: Definition of the `ADM_get_statistics` internal interface.

Name: *ADM_get_statistics*
Control ID: ID6
Control Type: Pull
Input: job_id
Input: job_step
Output: $job\_statistics$
Description:
Returns the current I/O statistics for a specified job_id and an optional corresponding job_step. The information will be returned in an easy-to-process format, e.g., JSON (see Listing 3.1).

Listing 3.1: Example JSON output for job I/O activity.

```
{"job_statistics": {
  "job_id": "132544",
  "job_step": "1",
  "runtime": "180s",
  "number_of_operations": {
    "metadata": [
      {"type": "create", "ops": "30000"},
      {"type": "stat", "ops": "50000"},
      {"type": "remove", "ops": "500"},
      {"type": "readdir", "ops": "23"}
    ],
    [...]
  }
}}
```

3.7 UML diagram

For the sake of completeness, Figure 3.2 shows the UML diagram of the functions provided by the I/O Scheduler. The diagram illustrates the API definitions related to each one of the ADMIRE architecture components: SLURM, Sensing and Profiling, Malleability Manager, Intelligent Controller, Adhoc Storage and parallel file system.

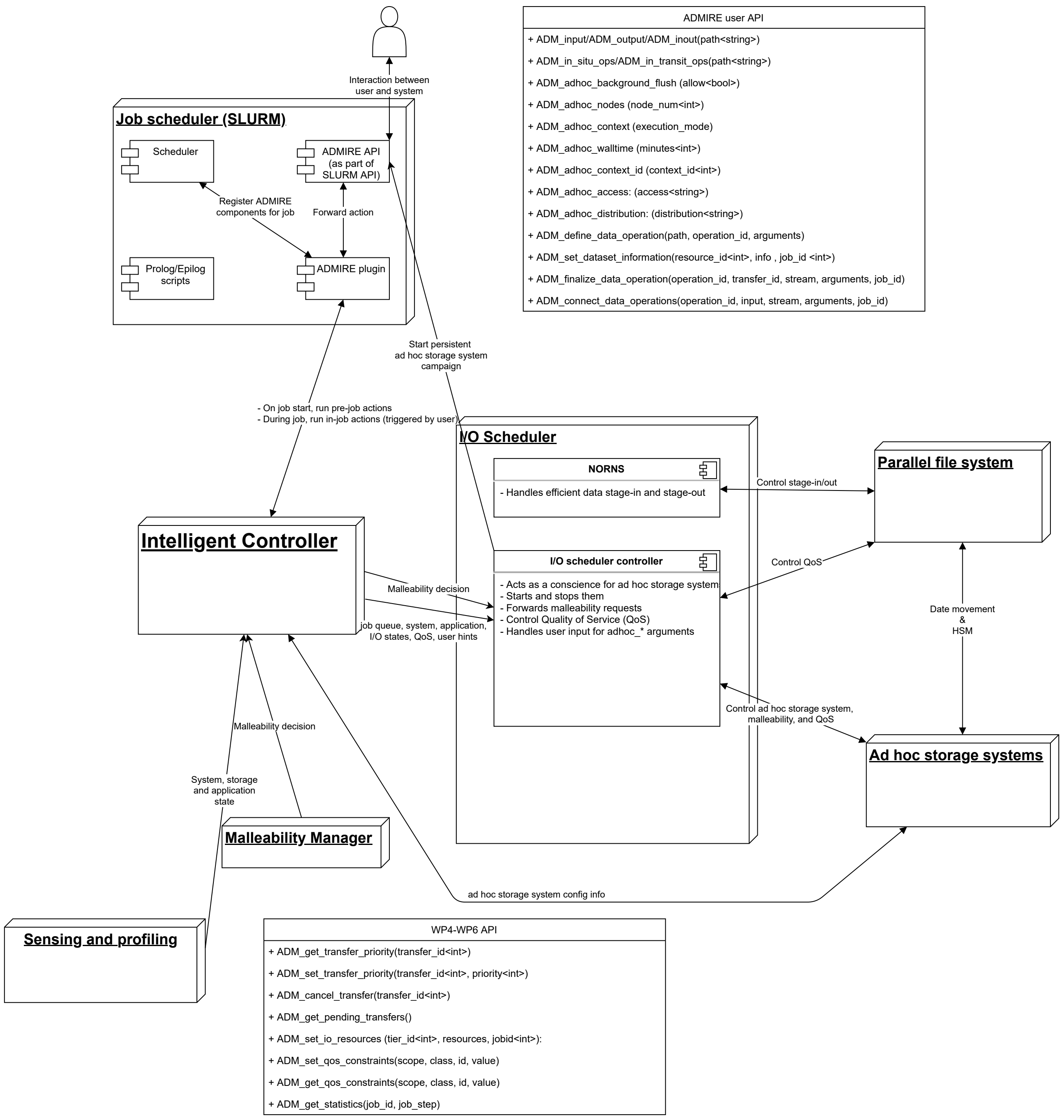
Interaction between user and system
ADMIRE user API
+ ADM_input/ADM_output/ADM_inout(path<string>)
+ ADM_in_situ_ops/ADM_in_transit_ops(path<string>)
+ ADM_adhoc_background_flush (allow<bool>)
+ ADM_adhoc_nodes (node_num<int>)
+ ADM_adhoc_context (execution_mode)
+ ADM_adhoc_walltime (minutes<int>)
+ ADM_adhoc_context_id (context_id<int>)
+ ADM_adhoc_access: (access<string>)
+ ADM_adhoc_distribution: (distribution<string>)
+ ADM_define_data_operation(path, operation_id, arguments)
+ ADM_set_dataset_information(resource_id<int>, info , job_id <int>)
+ ADM_finalize_data_operation(operation_id, transfer_id, stream, arguments, job_id)
+ ADM_connect_data_operations(operation_id, input, stream, arguments, job_id)
Job scheduler (SLURM)
Scheduler
ADMIRE API (as part of SLURM API)
Register ADMIRE components for job
Forward action
Prolog/Epilog scripts
ADMIRE plugin
Start persistent ad hoc storage system campaign
- On job start, run pre-job actions
- During job, run in-job actions (triggered by user)
I/O Scheduler
NORNS
- Handles efficient data stage-in and stage-out
Control stage-in/out
Parallel file system
Intelligent Controller
I/O scheduler controller
- Acts as a conscience for ad hoc storage system
- Starts and stops them
- Forwards malleability requests
- Control Quality of Service (QoS)
- Handles user input for adhoc_* arguments
Control QoS
Malleability decision
job queue, system, application, I/O states, QoS, user hints
Date movement & HSM
Control ad hoc storage system, malleability, and QoS
Malleability decision
Ad hoc storage systems
System, storage and application state
Malleability Manager
ad hoc storage system config info
Sensing and profiling
WP4-WP6 API
+ ADM_get_transfer_priority(transfer_id<int>)
+ ADM_set_transfer_priority(transfer_id<int>, priority<int>)
+ ADM_cancel_transfer(transfer_id<int>)
+ ADM_get_pending_transfers()
+ ADM_set_io_resources (tier_id<int>, resources, jobid<int>):
+ ADM_set_qos_constraints(scope, class, id, value)
+ ADM_get_qos_constraints(scope, class, id, value)
+ ADM_get_statistics(job_id, job_step)

Figure 3.2: UML diagram related to WP4 API.

Conclusion

This deliverable describes the responsibilities of the I/O Scheduler and how it interacts with other components in ADMIRE. The I/O Scheduler is the component in ADMIRE responsible for orchestrating the different data movements between the Back-end and Ad hoc Storage Systems. It is also responsible for conveying and enforcing QoS constraints to the involved storage tiers in response to the system-wide information collected by the other components of the framework and processed by the Intelligent Controller. Moreover, it is also required to provide facilities for users to deploy and execute data operations that can be applied to datasets both *in situ* and *in transit*.

This deliverable also proposes a set of interfaces for the I/O Scheduler that (1) allow capturing the aforementioned I/O requirements from end-users; (2) enable other components to define and communicate QoS constraints for the I/O Scheduler to apply; and (3) allow describing and executing the necessary data transfers required for the execution of job workflows. The interfaces proposed have been designed on the one hand to allow end-users to provide these requirements via the Slurm Workload Manager (and/or specific APIs if developers wish to modify their applications), and on the other hand to be flexible enough for other ADMIRE components to communicate enough information for the I/O Scheduler to produce and apply useful I/O scheduling decisions.

Appendix A

Annex I: 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 sophistical object store 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 and the programming language and support a variety of data formats. The only requirement is that they align to the following six REST design principles - Uniform interface, Client-server decoupling, Statelessness, Cacheability, Code on demand (optional).
- OSS, an Object Store Server in the Lustre terminology is a computing server in charge of managing the ingest of data, including generation of the data protection, and ship these data to the correct Object Store Target.
- OST, Object Store Target in the Lustre terminology is a storage server accommodating potentially a large number of hard drives and/or NMVes. The OST write the data received from the OSS and make them persistent.
- 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] Hasan Abbasi, Matthew Wolf, Greg Eisenhauer, Scott Klasky, Karsten Schwan, and Fang Zheng. Datastager: scalable data staging services for petascale applications. *Cluster Computing*, 13(3):277–290, 2010.

[2] Utkarsh Ayachit, Andrew Bauer, Berk Geveci, Patrick O'Leary, Kenneth Moreland, Nathan Fabian, and Jeffrey Mauldin. Paraview catalyst: Enabling in situ data analysis and visualization. In *Proceedings of the First Workshop on In Situ Infrastructures for Enabling Extreme-Scale Analysis and Visualization*, pages 25–29, 2015.

[3] Utkarsh Ayachit, Brad Whitlock, Matthew Wolf, Burlen Loring, Berk Geveci, David Lonie, and E. Wes Bethel. The sensei generic in situ interface. In *2016 Second Workshop on In Situ Infrastructures for Enabling Extreme-Scale Analysis and Visualization (ISAV)*, pages 40–44, 2016.

[4] BDVA. BDVA Strategic Research and Innovation Agenda V4. Technical report, BDVA, 2017.

[5] David A Boyuka, Sriram Lakshminarasimham, Xiaocheng Zou, Zhenhuan Gong, John Jenkins, Eric R Schendel, Norbert Podhorszki, Qing Liu, Scott Klasky, and Nagiza F Samatova. Transparent in situ data transformations in ADIOS. In *2014 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pages 256–266. IEEE, 2014.

[6] Hank Childs, Sean D. Ahern, James Ahrens, Andrew C. Bauer, Janine Bennett, E. Wes Bethel, Peer-Timo Bremer, Eric Brugger, Joseph Cottam, Matthieu Dorier, Soumya Dutta, Jean M. Favre, Thomas Fogal, Steffen Frey, Christoph Garth, Berk Geveci, William F. Godoy, Charles D. Hansen, Cyrus Harrison, Bernd Hentschel, Joseph Insley, Chris R. Johnson, Scott Klasky, Aaron Knoll, James Kress, Matthew Larsen, Jay Lofstead, Kwan-Liu Ma, Preeti Malakar, Jeremy Meredith, Kenneth Moreland, Paul Navrátil, Patrick O'Leary, Manish Parashar, Valerio Pascucci, John Patchett, Tom Peterka, Steve Petruzza, Norbert Podhorszki, David Pugmire, Michel Rasquin, Silvio Rizzi, David H. Rogers, Sudhanshu Sane, Franz Sauer, Robert Sisneros, Han-Wei Shen, Will Usher, Rhonda Vickery, Venkatram Vishwanath, Ingo Wald, Ruonan Wang, Gunther H. Weber, Brad Whitlock, Matthew Wolf, Hongfeng Yu, and Sean B. Ziegeler. A terminology for in situ visualization and analysis systems. *The International Journal of High Performance Computing Applications*, 34(6):676–691, 2020-11.

[7] Hank Childs, Eric Brugger, Brad Whitlock, Jeremy Meredith, Sean Ahern, and Kathleen Bonnell. Visit: An end-user tool for visualizing and analyzing very large data. In *In Proceedings of SciDAC*, 2011.

[8] Matthieu Dorier, Gabriel Antoniu, Franck Cappello, Marc Snir, Robert Sisneros, Orcun Yildiz, Shadi Ibrahim, Tom Peterka, and Leigh Orf. Damaris: Addressing performance variability in data management for post-petascale simulations. *ACM Transactions on Parallel Computing*, 3(3):1–43, 2016-12-26.

[9] William F Godoy, Norbert Podhorszki, Ruonan Wang, Chuck Atkins, Greg Eisenhauer, Junmin Gu, Philip Davis, Jong Choi, Kai Germaschewski, Kevin Huck, Axel Huebl, Mark Kim, James Kress, Tahsin Kurc, Qing Liu, Jeremy Logan, Kshitij Mehta, George Ostrouchov, Manish Parashar, Franz Poeschel, David Pugmire, Eric Suchyta, Keichi Takahashi, Nick Thompson, Seiji Tsutsumi, Lipeng Wan, Matthew Wolf, Kesheng Wu, and Scott Klasky. ADIOS 2: The adaptable input output system. a framework for high-performance data management. *SoftwareX*, page 9, 2020.

[10] Mathias Hummel and Kees van Kooten. Leveraging NVIDIA omniverse for in situ visualization. In Michèle Weiland, Guido Juckeland, Sadaf Alam, and Heike Jagode, editors, *High Performance Computing*, volume 11887, pages 634–642. Springer International Publishing, 2019. Series Title: Lecture Notes in Computer Science.

[11] Matthew Larsen, James Ahrens, Utkarsh Ayachit, Eric Brugger, Hank Childs, Berk Geveci, and Cyrus Harrison. The ALPINE in situ infrastructure: Ascending from the ashes of strawman. In *Proceedings of the In Situ Infrastructures on Enabling Extreme-Scale Analysis and Visualization*, pages 42–46. ACM, 2017-11.

[12] Qing Liu, Jeremy Logan, Yuan Tian, Hasan Abbasi, Norbert Podhorszki, Jong Youl Choi, Scott Klasky, Roselyne Tchoua, Jay Lofstead, Ron Oldfield, Manish Parashar, Nagiza Samatova, Karsten Schwan, Arie Shoshani, Matthew Wolf, Kesheng Wu, and Weikuan Yu. Hello ADIOS: the challenges and lessons of developing leadership class i/o frameworks: HELLO ADIOS. *Concurrency and Computation: Practice and Experience*, 26(7):1453–1473, 2014-05.

[13] Jay F Lofstead, Scott Klasky, Karsten Schwan, Norbert Podhorszki, and Chen Jin. Flexible io and integration for scientific codes through the adaptable io system (ADIOS). In *Proceedings of the 6th international workshop on Challenges of large applications in distributed environments*, pages 15–24, 2008.

[14] N. Mathur and R. Purohit. Issues and challenges in convergence of big data, cloud and data science. *IJCA*, 160:7–12, 02 2017.

[15] Alberto Miranda, Adrian Jackson, Tommaso Tocci, Iakovos Panourgias, and Ramon Nou. NORNS: Extending Slurm to support data-driven workflows through asynchronous data staging. In *2019 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 1–12, 2019.

[16] Kenneth Moreland, Ron Oldfield, Pat Marion, Sebastien Jourdain, Norbert Podhorszki, Venkatram Vishwanath, Nathan Fabian, Ciprian Docan, Manish Parashar, Mark Hereld, et al. Examples of in transit visualization. In *Proceedings of the 2nd international workshop on Petascal data analytics: challenges and opportunities*, pages 1–6, 2011.

[17] Arifa Nisar, Wei-keng Liao, and Alok Choudhary. Scaling parallel I/O performance through I/O delegate and caching system. In *SC'08: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, pages 1–12. IEEE, 2008.

[18] Yingjin Qian, Xi Li, Shuichi Ihara, Lingfang Zeng, Jürgen Kaiser, Tim Süß, and André Brinkmann. A configurable rule based classful token bucket filter network request scheduler for the lustre file system. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12, 2017.

[19] Will Schroeder, Kenneth M Martin, and William E Lorensen. *The visualization toolkit an object-oriented approach to 3D graphics*. Prentice-Hall, Inc., 1998.

[20] Rajeev Thakur, William Gropp, and Ewing Lusk. A case for using MPI's derived datatypes to improve I/O performance. In *Proceedings of SC98: High Performance Networking and Computing*. ACM Press, November 1998.

[21] Venkatram Vishwanath, Mark Hereld, Vitali Morozov, and Michael E Papka. Topology-aware data movement and staging for I/O acceleration on Blue Gene/P supercomputing systems. In *SC'11: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, page 11, 2011.

[22] Brad Whitlock, Jean M. Favre, and Jeremy S. Meredith. Parallel in situ coupling of simulation with a fully featured visualization system. In *Proceedings of the 11th Eurographics Conference on Parallel Graphics and Visualization*, EGPGV '11, page 101–109, Goslar, DEU, 2011. Eurographics Association.

[23] Li Xi and Zeng Lingfang. Lime: A framework for Lustre global QoS management. In *Lustre Administrator and Developer Workshop*, 2018.