H2020-JTI-EuroHPC-2019-1

Project no. 956748

ADAPTIVE MULTI-TIER INTELLIGENT DATA MANAGER FOR EXASCALE

D5.1
Definitions of the profiling and monitoring requirements

Version 1.0

Date: September 29, 2021

Type: Deliverable
WP number: WP5

Editor: Jesus Carretero
Institution: Uc3M

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

Change Log

Rev.	Date	Who	Site	What
1	21/04/21	Jesus Carretero	UC3M	Document creation.
2	06/09/21	Jean-Baptiste Besnard	PARA	Probes and sensing chapter (Part 1)
3	09/02/21	Jean-Thomas Acquaviva	DDN	Overview and architecture
4	09/09/21	Jean-Baptiste Besnard	PARA	Probes and sensing chapter (Part 2)
5	12/09/21	JT Acquaviva	DDN	Injection of the first version of the API
6	16/09/21	JT Acquaviva	DDN	Alpha version
7	17/09/21	Jean-Baptiste Besnard	PARA	Rework and complement API Chapter
8	20/09/21	Marc-André Vef	JGU	Added GekkoFS section
9	20/09/21	Ahmad Tarraf	TUDA	Added Extra-P related sections
10	20/09/21	JB Besnard, JT Acquaviva, M. Vef, A. Queralt	PARA, DDN, JGU, BSC	Beta version to be shipped for WP internal review
11	21/09/21	Ramon Nou	BSC	Internal Review
12	21/09/21	Ahmad Tarraf	TUDA	Internal Review
13	21/09/21	Barbara Cantalupo	CINI	Internal Review
14	23/09/21	JT Acquaviva	DDN	Consolidation based on internal reviews
15	25/09/21	Jesus Carretero	UC3M	External review #1
16	27/09/21	JB Besnard, JT Acquaviva	PARA, DDN	Addressed Review #1

Executive Summary

The ADMIRE project is defining a new holistic manner of coordinating Inputs and Outputs (I/Os) at the scale of an Exascale High-Performance Computing (HPC) system. The core objective is to design and implement a control and measure feedback-loop. The implementation of the feed-back loop will allow to both predict and learn optimal I/O configurations. To achieve this goal, the partners in ADMIRE, which are involved in all layers of the I/O stack ranging from the parallel application up to the low-level storage layer, are building a common infrastructure. In this context, the deliverable D5.1 addresses the envisioned Application Programming Interface(API) to implement performance feedback in the project. This API has been devised to provide as much information as possible while remaining lean and re-configurable to constantly adapt our measurement. The deliverable addresses the complexities of I/O measurements, review the existing state-of-the-art, and unfolds our proposed implementation. An added value of the approaches in ADMIRE is the combination of the all measurements layers leveraging their synergy. This document marks as well the reaching of Milestone 2: The API definition, with a Chapter (Chapter 4) dedicated to the presentation of the API to interface Performance Monitoring and the Intelligent Controller. Beside the presentation of the API, this deliverable sets the base for a versatile I/O measurement layer thanks to cross-layer collaborations. Moreover, due to the co-design process and the close collaboration in the project framework, we expect this new performance scheme to produce an unprecedented always-on versatile measurement layer for Exascale systems.

Contents

Chapter 1

Introduction

In the general ADMIRE architecture, as shown in Figure 1.1, WP5 is in charge of monitoring. The flow of information from other system components to WP5 is materialized by black arrows in the figure. The Monitoring Manager obtains information from multiple data sources: the ad-hoc file systems, the applications, the monitoring tool-set running on every nodes, and the main storage. All these data end-up stored in a dedicated Data Lake. An important architectural point is that data are flowing from multiple paths, but in term of control flow only a single path exists. Among all the components of the ADMIRE system only the Intelligent Controller can exchange commands and interact with the Monitoring System. Therefore, the Monitoring System can be seen as a data aggregator driven by the Intelligent Controller.

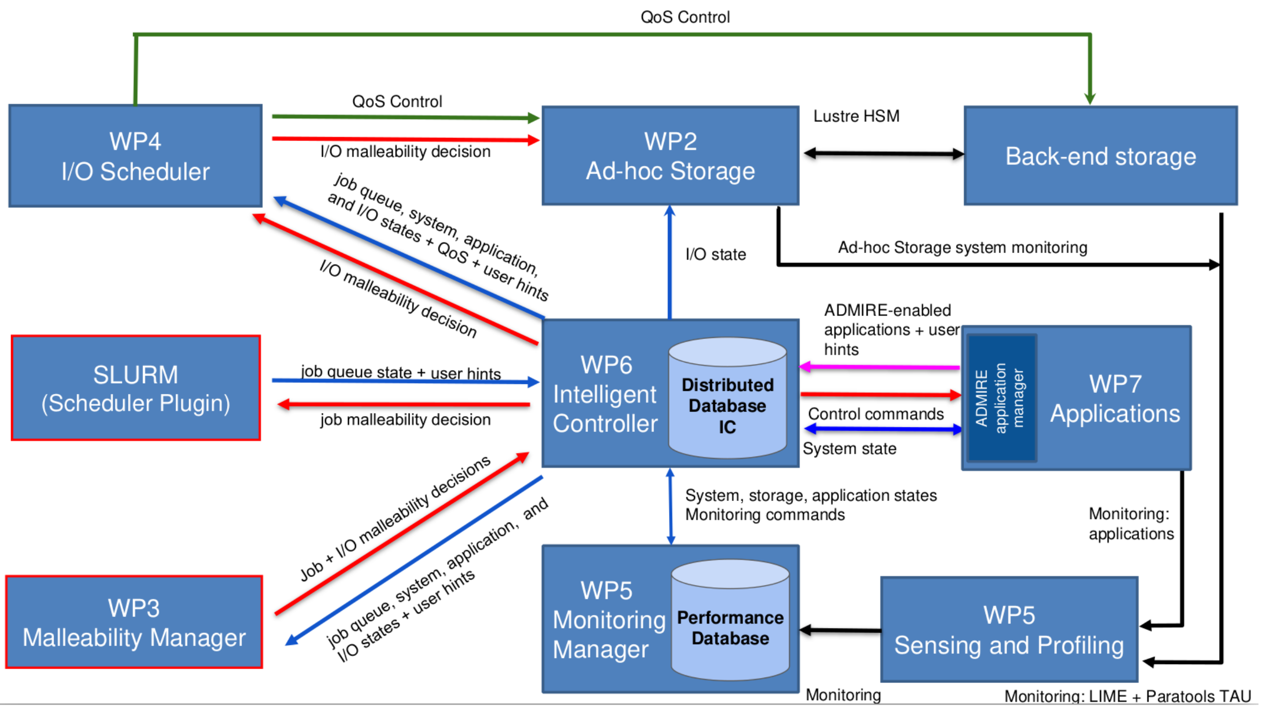
QoS Control
WP4 I/O Scheduler
QoS Control
I/O malleability decision
WP2 Ad-hoc Storage
Lustre HSM
Back-end storage
job queue, system, application, and I/O states + QoS + user hints
I/O malleability decision
I/O state
Ad-hoc Storage system monitoring
SLURM (Scheduler Plugin)
job queue state + user hints
job malleability decision
WP6 Intelligent Controller
Distributed Database IC
ADMIRE-enabled applications + 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
Monitoring: applications
WP3 Malleability Manager
WP5 Monitoring Manager
Performance Database
WP5 Sensing and Profiling
Monitoring
Monitoring: LIME + Paratools TAU

Figure 1.1: General Control Flow between ADMIRE components. The *WP5 Sensing and Profiling* probes and the *Monitoring Manager* in charge of managing data collection are referred jointly as the Monitoring System in the remainder of the document.

The purpose of this first Deliverable is to provide the current design status for the Monitoring System and to define a set of probes and their respective APIs. Publication of the API at this early stage of the project is important, as it has implication on the Intelligent Controller road map and implementation.

The internals of the Monitoring System will be iteratively refined during the lifetime of the project, We will start with the most generic probes and then refine and develop more advanced features. One example being an initial measurement of the CPU load and the I/O load and later-wise a measurement of the overlap between

CPU and I/O activities.

The probe mechanism envisioned in Chapter 4 is hierarchical, in the sense that a rule-based language is proposed to trigger additional measurements (e.g. if and only if a I/O activity reaches a threshold then a probing is triggered which is able to determine if this I/O activity is overlapped with computation). In the foreseen implementation of the Monitoring System, the notification capability of the Monitoring Manager is considered as a part of the Analytic Module. More details, including code samples, are provided in Section 2.4.6. Notification is now an expected feature in Cloud monitoring software that we intend to leverage within ADMIRE [1].

The general approach of WP5 is to favor a multi-components Monitoring System and avoid monolithic design. In terms of services, each compute node is observed by a dedicated process and the information is forwarded to a Data Lake under the orchestration of the Monitoring Manager. As several existing components already includes some self observation facilities, our approach is to integrate these capabilities and collect the data. After a review of these pre-existing monitoring capabilities in the software stack, we will present the outline and structure of the Deliverable.

1.1 Background Technologies

Prior to the project, the performance landscape is already an active field of research in the community. However, I/O is not well integrated and the most sophisticated solutions are devoted to the observation of the computational part.

Performances in ADMIRE are focused on I/O but not restricted solely to it. As we will discuss later in the document, there are some interactions between the I/O performance and the compute performance of an application. The ADMIRE consortium has some considerable expertise in performance analysis through the performance monitoring tool TAU. The I/O performance can be addressed either at the file system level, global and persistent such as Lustre, or ephemeral and at the local level as for GekkoFS. At last, performance can also be related to storage but not only to files, as illustrated by the Object Store dataClay.

The following section will depict the current status in terms of performance monitoring for each of these technologies.

1.1.1 TAU Performance Monitoring

TAU Performance System [26] is a portable profiling and tracing toolkit for performance analysis of parallel programs written in Fortran, C, C++, UPC, Java, or Python.

TAU (Tuning and Analysis Utilities) is capable of gathering performance information through instrumentation of functions, methods, basic blocks, and statements as well as event-based sampling. All C++ language features are supported, including templates and namespaces. The API also provides selection of profiling groups for organizing and controlling instrumentation. The instrumentation can be inserted in the source code using an automatic instrumenting tool based on the Program Database Toolkit (PDT), dynamically using DyninstAPI, at runtime in the Java Virtual Machine, or manually using the instrumentation API.

TAU's profile visualization tool, paraprof, provides graphical displays of all the performance analysis results, in aggregate and single node/context/thread forms. The user can quickly identify sources of performance bottlenecks in the application using the graphical interface. In addition, TAU can generate event traces that can be displayed with the Vampir, Paraver or JumpShot trace visualization tools.

1.1.2 Lustre

Lustre [3] is a fully POSIX-compliant parallel file system offering extreme performance and scalability. Lustre file system is ubiquitous in the HPC and is gaining footholds in the more generic enterprise market. The applications send their I/O requests to Lustre using a kernel module running on each node. The kernel module will direct the metadata operation to Lustre MetaData Server (MDS) and the data payload to the multiple Object Store Servers (OSS). Files are split in multiple chunks (named *stripe* in the Lustre terminology). The Object

[1] https://prometheus.io/docs/alerting/latest/overview/

Store Servers store these stripes as individual files on their local sequential file system. Therefore, Lustre can be seen as a parallel orchestrator on the top of multiple sequential file systems. The terminology to refer to such implementation of parallel file system is *overlay file system*. The number of active OSS defines the level of parallelism of the Lustre file system, hence the bandwidth performance. The second crucial aspect for the performance is related to the Metadata, here the number of active MDS is the main drivers for the performance. For such complex software stack performances depends on multiple criteria, some external such as the I/O pattern to handle and other related to some fine-tuning knobs, for instance the data-on-metadata or other optimization which are currently out of the scope of this document.

Lustre is an open-source project developed by the OpenSFS [2] with multiple commercial forks of the project, the most widely known ones being the ClusterStor [18] product from Cray / HPE and ExaScaler [29] from DDN.

Monitoring Capabilities

Lustre support multiple probes for logging, tracing, debugging, and monitoring performance.

Even though the logging mechanism of Lustre is powerful and extremely convenient for archiving (see RobinHood [17]), its usage for performance analysis remains limited only for metadata observation.

Lustre monitoring tools support a wide variety of metrics [3], however in the scope of ADMIRE the goal is to use specifically the most advanced Lustre monitoring interfaces. Originated from seminal works in Lime [31], the new performance monitoring interface of Lustre is named Barrel-eye[4]. This system allows not only to monitor the performance at node level or at the global file system level, but also at a key granularity in HPC: the job level. Figure 1.2 illustrates the typical measurements which can be obtained for (from?) a job on a

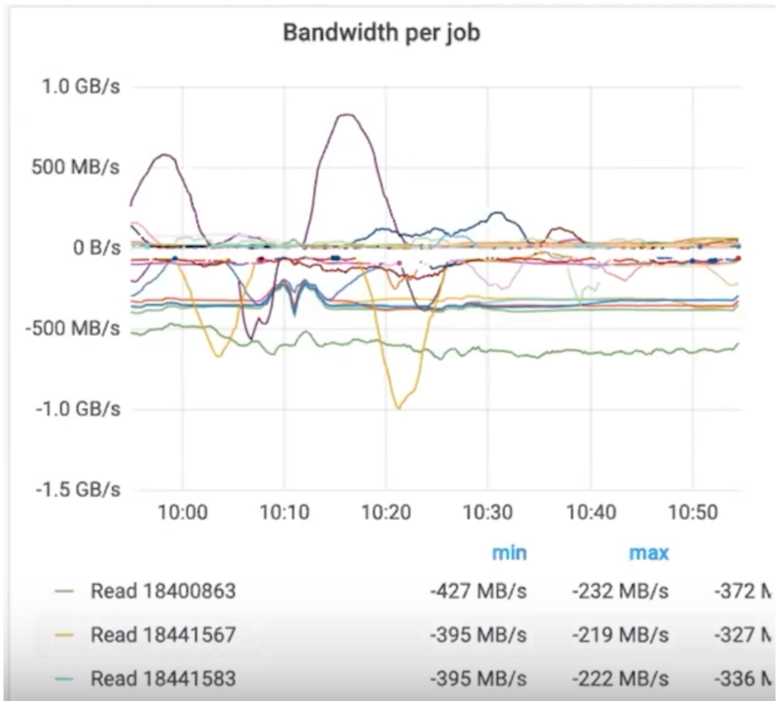

Figure 1.2: Lustre job monitoring capabilities as displayed by prototype version of Barrel-eye.

Lustre system. Barrel-eye is an open-source project available on GitHub.

A key feature in previously Lime and now Barrel-eye is the higher-level semantic brought to performance data. The file system can extract the job ID from each compute node and hence can rebuild the performance metrics on a per-job basis. This allows to provide performance information of all the jobs currently running on the system. The main limitation of Lime and Barrel-eye is that their sole focus is on I/O. No information about

[2] www.opensfs.org
[3] https://wiki.lustre.org/Lustre_Monitoring_and_Statistics_Guide
[4] https://github.com/LiXi-storage/barreleye

the memory pressure or the CPU/GPU activity is collected by the system. One of the goal of ADMIRE is to bridge this gap.

1.1.3 GekkoFS

GekkoFS is a highly scalable user-level distributed file system for HPC clusters. GekkoFS is capable of aggregating the local I/O capacity and performance of compute nodes to produce a high-performance storage space for applications. With GekkoFS, HPC applications and simulations can run isolated from each other regarding I/O, which reduces interferences and improves performance. Furthermore, GekkoFS has been designed with configurability in mind and allows users to fine tune several of the default POSIX file system semantics (e.g. support for symbolic links or strict bookkeeping of file access timestamps) that, even if useful, might not be required by their applications and hence negatively impact their I/O performance. Please refer to D2.1 for details on GekkoFS internals.

Monitoring

GekkoFS uses a tracing subsystem to report valuable insights on the behavior of GekkoFS. In the context of ADMIRE, we will extend GekkoFS and integrate it with monitoring tools, e.g., Prometheus, allowing us to report a number of GekkoFS performance counters to the ADMIRE stack. Examples for such counters could be the average number of RPCs per second for each server or the current storage capacity information. The specifics of available performance counters, particularly w.r.t. low-level network information, remain to be defined. For sending the combined monitoring information, one server will act as a management role and will be responsible to gather this information from other servers that are part of the ad-hoc file system. We envision several modes to make this data available: On-request or at defined intervals.

It is important to note that a too tight time interval, or in general, too granular monitoring may lead to significant performance degradation for the application. In the worst case, this may also impact the behavior of the application itself, resulting in invalid monitoring information because it does no longer reflect the application workload without monitoring [28]. Therefore, one of our tasks will be to minimize monitoring overhead as much as possible to ensure accurate information.

1.1.4 DataClay

DataClay [19] is a distributed object store with active capabilities. Objects in dataClay have an associated semantics, which gives them an object-oriented structure, as well as the possibility to attach arbitrary user code to manipulate them. This feature allows applications to execute object methods within dataClay, instead of transferring the objects to the application space. In this way, data transfers are reduced and also new devices, such as NVMs, can be leveraged during the execution of these methods transparently to the application. dataClay is implemented at user-level, so it is visible to applications using its client library. Please refer to D2.1 for further details on dataClay and its architecture.

Monitoring Capabilities

dataClay is integrated with the Paraver[5] tool, which enables the performance analysis of applications and detect possible bottlenecks or other sources of inefficiencies. This is done through different kinds of visualizations of execution traces that can be analyzed under several metrics in a post-mortem manner. However, dataClay does not currently provide its metrics and tools to monitor the behavior of the system during execution.

Thus, one of the goals within ADMIRE is to integrate dataClay with a monitoring tool, such as Prometheus, to obtain on-line metrics that can be exported to other components in the ADMIRE stack. The specific set of metrics that will be provided is currently being defined. However, at the current time, we have already identified some metrics that can be relevant for the objectives of the ADMIRE architecture. All of them are defined within the scope of a node, as this is the kind of information that will be useful to make decisions, e.g. on malleability.

[5] https://tools.bsc.es/

The kind of metrics we are currently considering are, for instance, the total size of objects[6] in a node, the number of objects per node, or hits and misses in the object caches. In addition, as dataClay is an active object store, additional metrics related to execution might also be interesting. For instance, the execution time of a method, or the amount of concurrent executions, can also be useful inputs for the Intelligent Controller.

1.1.5 Hercules

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

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

Hercules provides multiple data distribution policies by design (local, buckets, hash, round-robin, and crc) in both deployment modes, increasing the number of data scattering possibilities among storage servers and enlarging versatility in terms of application's data management. Within the previous possibilities, a *LOCAL* policy should be highlighted as it will have the objective of exploiting data locality as much as possible: data requests will be forwarded to the storage server running in the same machine where the request was made. Finally, a non-POSIX *get-set* interface will be provided in order to manage *datasets*, which conform to a storage abstraction used by Hercules instances to manage data blocks (the smallest data unit considered within the storage system).

As such, Hercules does not provide its monitoring tools. However, in the scope of ADMIRE the goal is to generate probes and metrics to monitor the Hercules system specifically and to provide a monitoring interface compatible with the monitorization tools used in ADMIRE.

1.2 Document Structure

D5.1 is organized in four different chapters (in addition to the present Introduction). Chapter 2 describes our design principles and a general overview of the resulting architecture. Chapter 3 is a deep-dive and details our future technical implementation. To back-up our technical choices the chapter includes also discussions on the nature of profiling. The API itself, which is the reference part of the document is described in Chapter 4. Even if the conclusion in chapter 5 presents both a summary of our contribution and a road-map of our futures works, interested readers should not overlook the Bibliography and detailed Terminology provided at the end of this document.

[6]Object is data format to store data. Objects offer a much simpler API than files, as a result Object storage systems tends to offers better scalability than file systems.

Chapter 2

Sensing and profiling system architecture

In this section, we will first describe the challenges we faced in the design of our monitoring framework, then in the light of these constraints, we will detail our proposed architecture. Eventually, we will discuss telemetry and how precise node state will be monitored using common tools and additional plugins tailored for ADMIRE's needs.

2.1 ADMIRE Design Principles

The ADMIRE Performance Monitoring architecture follows the principle of separation of concern: within the Monitoring System, data flows and control flows are separated. Furthermore, the standalone Monitoring System in ADMIRE does not functionally depend on any other technical bricks. Of course, the system remains inter-operable and can be interfaced with other components in ADMIRE through high-level API. All the communications between ADMIRE malleability software stack and the Monitoring System are made through the interface between the Monitoring Manager and the Intelligent Controller.

An important ambition in ADMIRE is to leverage existing Open-Source project and avoid developing things from scratch. Before re-implementing a component, we will review available alternative with the right license model. Pushing code upstream to the right project is a more perennial investment than from scratch developments, specifically in respect of code maintenance beyond the lifespan of the project.

2.1.1 Control Plane

The control plane in ADMIRE is internal between the sub-components of the Monitoring System, and external between the Monitoring Manager and the other ADMIRE main software components. Figure 2.3 presents the flow of information between the different components of the Monitoring System: Monitoring Manager, Monitoring Daemons, performance Data Lake, Monitoring Display, and the Analytic Module. Within the Monitoring System, all the monitoring components reports solely and react only to commands from the Monitoring Manager. This simple approach allows to implement and to develop the Monitoring System in a standalone environment and drastically limits the complexity of testing.

The flow of control information within the Monitoring System will be detailed in Figure 2.3. The head of the system is the Monitoring Manager: in terms of control it will send messages to all the Monitoring Daemons running on each compute nodes. Additional control traffic will occur between the performance Data Lake and the Monitoring Display, as well as between the Data Lake and the Analytic Module.

The integration of the Monitoring System within the broader framework of ADMIRE stack is handled by the Monitoring Manager which will exchange control traffic with the Intelligent Controller. This control traffic is the core topic of the API presented in Chapter 4.

2.1.2 Data Plane

The data plane for the performance system is more complex than the control plane. Some elements emit a steady amount of information per second, while others have a less predictable flow. In ADMIRE, performance data can be collected from 3 different sources: the file system, the ephemeral file system, and the compute nodes. All these streams of data are forwarded to the performance Data Lake.These flows to the performance Data Lake are ingest flows, in terms of outgest flows ADMIRE Monitoring System supports 3 flows as well. One between the Data Lake and the Analytic Module, one between the Data Lake and the live monitoring module, and one from the Data Lake to the Monitoring Manager. In our view of the architecture, the Monitoring Manager will constantly interrogate and browse the performance database to take the relevant decision (including initiating control communications with the Intelligent Controller).

The exact implementation means of the data plane are discussed and presented in Chapter 3.

2.2 Problem Statement: On the Variable Nature of I/Os activities

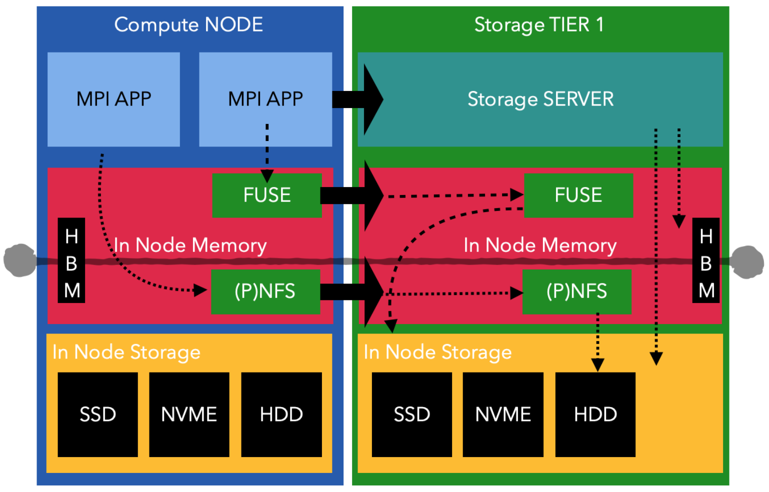

Figure 2.1: Illustration of an I/O system with various media and link kinds.

The starting point of the sensing and monitoring infrastructure in ADMIRE is to characterize I/Os activities in real-time on a whole system. Practically, it means our Monitoring System has to both instrument and map performance data to create an intelligible performance state. However, as far as I/Os are concerned and given the various devices present in an I/O subsystem, data-movement is a very general notion. As sketched in Figure 2.1, I/O encloses many-things and can be relatively complex to describe as a whole, and this is what makes ADMIRE challenging, requiring the controller to work at all levels. As far as measurement is concerned, I/O emitted by a single UNIX process are rapidly dissolved in a much larger stream of I/Os, crossing layers and storage caches. The consequence of this variability in I/O media, is that instrumentation – to be efficient – will need cooperation from the target programs. Such programs, not only encompass the actual scientific application, but also the associated I/O runtime. ADMIRE is an opportunity to cross these lines, bringing together parties which usually work side by side, in order to create a coherent but yet open monitoring infrastructure. Transversely, our goal in WP5 is to feed performance data inside a shared-database providing a representative performance state to help other work-packages and particularly WP6 featuring the Intelligent Controller (IC) and WP3 malleability.

2.3 Measurement Sources

Points discussed in the previous Section, combined with the variable I/O interfaces considered even in the project alone, advocates for a close collaboration with the I/O runtime to capture figures of interest. This will have the advantage of not requiring instrumentation to *guess* the I/O behavior but instead, a clear state will be derived thanks to dedicated instrumentation. Besides, some I/O subsystems are also relying on pre-loading to capture relevant I/O calls, resulting in potential conflicts in case a similar approach was used for profiling. Practically, it means that always-on measurements should be made available in a manner not impacting the applications. Indeed, compatibility with a wide-range of programs without modifications is a strong requirement.

Naturally, application-state is not something we gave up upon, and specific means of attaching and detaching from a running program will be developed to provide a wide performance spectrum to the Intelligent Controller.

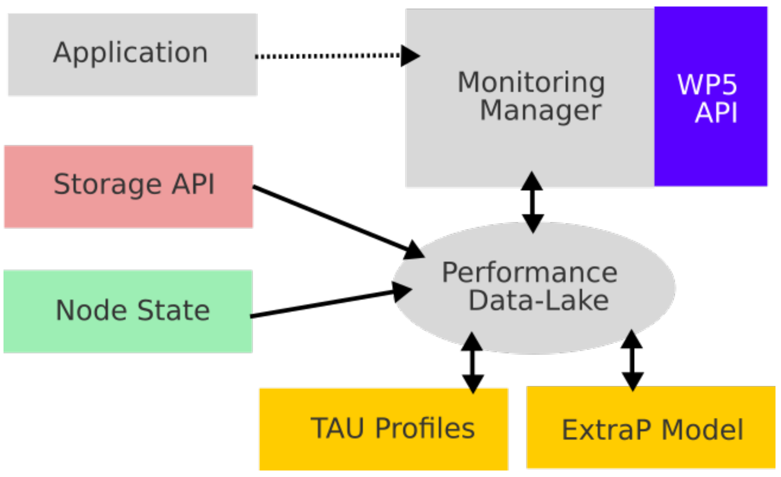

Figure 2.2: Overview of the instrumentation data-flow.

As presented in Figure 2.2, instrumentation data-flow is made of two main classes. First, as discussed in the previous paragraph, always-on measurements are managed separately and directed to a dedicated local storage, the remote Data Lake will be fed by filtered versions of these measurements. As we will detail further later in this document, Prometheus is the considered Time Series DataBase (TSDB) to handle such long-running measurements. This then covers both the node status in general (e.g. resources) and the I/O interfaces. For this later point, efforts will be dedicated to instrument involved I/O interfaces to report to the TSDB. A second kind of measurement is originating from the application itself. As it is critical not to interfere with it in terms of performance, we will develop a specific profiling tool capable of both attaching and detaching from an already running application. It means that we plan to provide dynamic profiling capabilities similar to what a debugger is capable of. Eventually, a more verbose information will be provided as profiles generated from the TAU performance system or models build with Extra-P in the performance database.

Overall, all this information will be both processed and exposed through the dedicated WP5 API as presented in Section 4. Thanks to this interface, the IC will be able to retrieve important performance events as they occur during the execution. Moreover, it is not excluded to allow the IC to directly access the Performance Data-Lake for some applications such as machine learning where the more data, the better.

2.4 Resulting Architecture

The monitoring infrastructure in ADMIRE has to provide 4 key services,

1. **Observability** Observe the I/O consumption of each application process running on each compute node. Data are collected on each compute node assigned to a peculiar job. The observability supports inter-node malleability. For instance, if a new compute node is assigned at runtime to a job or some compute nodes are de-allocated from a running job, the Monitoring System is able to tag dynamically the collected data and to keep the performance data in consistency with the evolution of the job perimeter.

2. **Scalable Telemetry** Even if a job is executed on many computes nodes (i.e thousands), potentially on the whole system, the transfer of the performance data is not altered, and the monitoring data are sent efficiently across the system.

3. **Persistence and archive** Performance data collected during the execution of a job survive the execution of the job itself. This means that the monitoring infrastructure is not limited to live observation, but can be used for post-processing analytic.

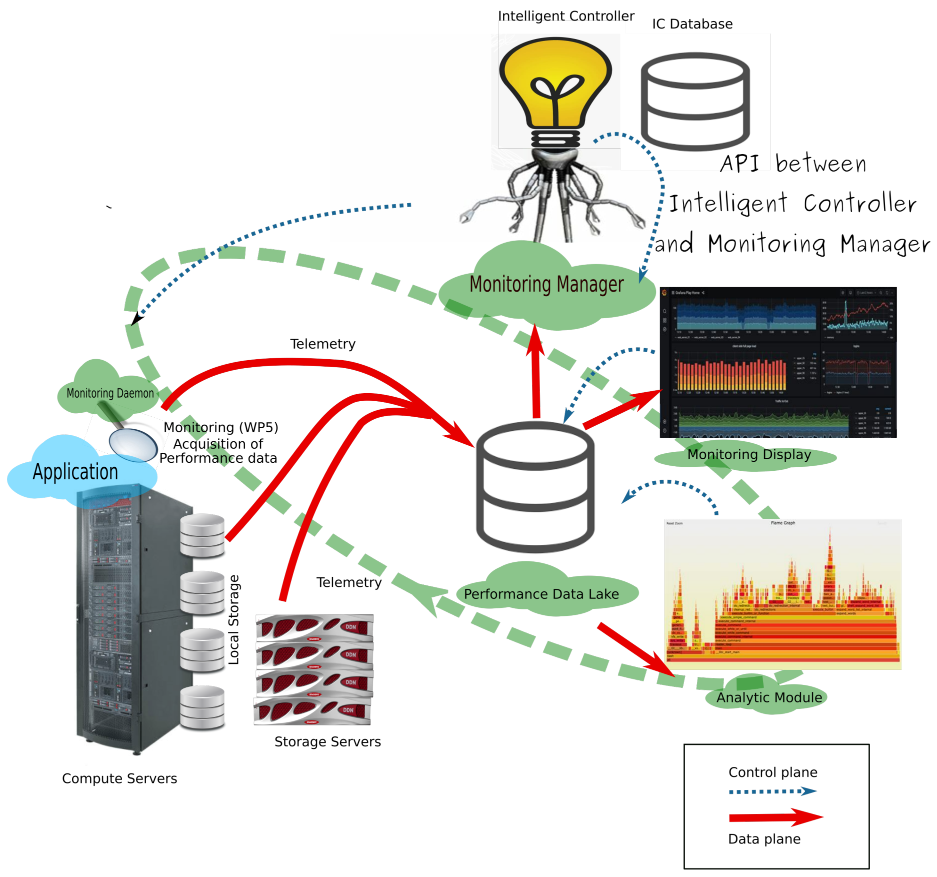

Figure 2.3: Overview of the monitoring infrastructure in ADMIRE.

4. **Exploitation** Data collected during the execution of the job are analyzed and meaning is extracted. The exploitation stage needs to be flexible enough in order to allow advanced users (tools) to code their analysis, and also to propose high-level metrics and analytic to detect and potentially solve performance issues.

As displayed in Figure 2.3, ADMIRE instantiates these 4 services with the following components:

- **Monitoring Daemon**, this process is always running on every computing nodes of the system. The Monitoring Daemon will probably be automatically started at boot time. This process is in charge of handling commands from the Monitoring Manager and potentially to trigger more accurate and intrusive performance monitoring.

- **Monitoring Manager** is the interface between the Monitoring System and the IC. The IC is the sole component from the ADMIRE Software stack able to directly interact with the Monitoring Manager. The purpose of this communication is twofold, first to analyze and detect potential issues in terms of I/O performance, and second to adapt the monitoring filter to potential variation of an application (malleability).

- **Performance Database**, or Data Lake. This database is in charge of logging all the performance information collected across the system. We can expect to be oriented toward time series due to the temporal nature of performance samples.

- **Telemetry** is the solution used to convey data from the compute node to the performance database. In principle Telemetry is a simple process (remote copy) but it has to offer strong properties related to scalability, reliability while still being low overhead and lightweight. Among the existing software tools, not that many offer an implementation fulfilling all these criteria.

- **Monitoring Display** is in charge of presenting key performance indicators extracted from the database to system administrators. Typically, it is a dashboard summarizing the status of the production and potentially handling notification for some specific events (components detected as dead peers and so on).
- **Analytic Module** is the module triggered on-demand to perform either prediction for the future performance of a peculiar job or at the opposite to analyze the past executions of a specific binary. The modeling will include Extra-P [24] and the analysis will provide clustering and other data science analysis.

2.4.1 Monitoring Daemon

The Monitoring Daemon will be always on and started during the boot process of all the compute nodes of the facility. This daemon is lightweight, resides in user space and provides the main communication chanel to the Monitoring Manager. The implementation of the Monitoring Daemon will be in two phases; an initial early prototyping for proof-of-concept and validation of the integration, followed by a full features and more complex implementation in C for efficiency purpose. Upon request from the Monitoring Manager, any Monitoring Daemon can attach a profiling process to an executing Application process to collect more performance data. The Monitoring Daemon can also increase or decrease the sampling rate, again upon request from the Monitoring Manager.

The monitoring Daemon will be developed from scratch for the ADMIRE project.

2.4.2 Monitoring Manager

The Monitoring Manager interacts with the Intelligent Controller. The Monitoring Manager is in-charge of sending commands and control message to all the Monitoring Daemon in the system. The Monitoring Manager is the interface of the Performance Monitoring System with the Intelligent Controller, therefore the Monitoring Manager implement an API (see Chapter 4) to communicate with the Intelligent Controller. The API will be implemented both as a Rest API and as a lower level RPC API. The Monitoring Manager will be hosted on a dedicated server or on a server consolidated with the Intelligent Controller. Notice that the Fault Tolerance features of the Monitoring Manager (redundancy) have yet to be addressed. The Monitoring Manager will be developed from scratch for the ADMIRE project.

2.4.3 Performance Database

Performance information need to be recorded in a Database. The database has to provide high ingest capabilities since some data streams will be always on. This database will be used for modeling and by the Intelligent Controller to take key decisions. Therefore, the database needs to support modern and efficient browsing capabilities. As any important component of the architecture, the database needs to be resilient when confronted to fault. We are evolving in HPC grade environment with potentially huge amount of compute servers to observe, and thus the database has to provide good scalability and at last we do expect a product mature enough to avoid too many deployments and maintenance issues. Table 2.1 captures these criteria and compare some solutions available on the market.

Some database solutions listed in the Table 2.1 have been designed specifically for performance record, they offer time series as default data format. We believe this is the right approach for ADMIRE system and therefore restricts ourselves among this sub-set of database. The most appealing technology in the current landscape of time-series database seems to be Prometheus [7]. Prometheus is a framework dedicated to performance data with a strong momentum, and it offers multiple plugins and bridges to other technologies. The query language of Prometheus database, PromQL (Prometheus Query Language), offers interesting features. Early prototyping should provide us more feedback on the strength and limitation of this query language. During the project implementation, WP5 should re-evaluate the database landscape to assess the relevance of this initial choice. At this stage, InfluxDB is considered as an interesting alternative to Prometheus.

Product	Data scheme	Scalability	Fault Tolerance	Code Maintenance
Cassandra	NoSql	++	+	Apache 2.0 license and strong community
HBase	NoSql	+	++	HBase tends to be less supported
MongoDB	NoSql	++	+	Maintenance of MongoDB at scale complex
ScyllaDB	NoSql	++	++	Emergering as replacement for MongoDB
RocksDB	NoSql	++	+	RocksDB is a storage library (used by Cassandra) will require integration
InFluxDB	Time Series	++	+	Free to start, pay for features
Prometheus	Time Series	++	++	++ emerging tech
LucidDB	Time Series	++	+	no longer maintained
File Systems	KV store	++	++	in-house development requires high maintenance
OpenTSDB	Time Series	+	+	Open Time series Database reliance on HBase increases maintenance [1]

Table 2.1: Strength and weakness of database candidates to store performance records.

2.4.4 Telemetry

Telemetry is the software component in charge of transmitting the data collected in situ to a remote location. In the ADMIRE case, the performance data harvested locally to the Performance Data Lake. Telemetry also used for the Lustre performance data and the GekkoFS metrics.

The technical requirements for the telemetry solution are:

- **Versatile**, to handle a large variety of output formats from the monitoring tools.
- **Scalable**, support strategies prevent congestion at the database level.
- **Resilient**, failure of one of the nodes should not prevent the whole data flow.

Possible candidates are:

- Nagios [2] is a well established telemetry solution in the community. Its technology foundations were laid in the 90s and the stack mostly administered via scripts and files. Nagios is mostly used to monitor distributed services and offer logging capability. However, using Nagios to transfer MBs of performance data every second is stretching out the product original specifications. Therefore, despite checking the boxes on resilience and versatility some uncertainties remain about the ability to adjust to the volume of performance information.

[2] https://github.com/NagiosEnterprises/nagioscore

- Collectd [3], open source, mature, well accepted in the community. Collectd is generic in terms of data payload and not specifically designed for time series data. Barrel-eye is using a forked version of Collectd for its telemetry layer. Collectd can scale and is resilient. Furthermore even if Collectd was not originally designed to transmit profiling information, we know from previous deployments in the field that it can adapt to transfer performance data.
- Telegraf [4], open source (MIT License) modern high performance written in Go. Telegraf is a more recent product with a good footprint in the Cloud market. Telegraf can be connected to many databases from MongoDB to OpenTSB or InfluxDB. We have hands-on experience with Telegraf, confirming its ease of deployment and management. The license model of Telegraf is open-source, free to use for upgrade. The pricing model may prevent us to operate at scale. Despite the complete coverage of the 3 key criteria, versatile, scalable and resilient, the license model may be challenging for ADMIRE.
- Prometheus [5], open source emerging standard, comes as a framework than as a single component. Prometheus is released under Apache 2.0. The architecture is decoupled with many plugins developed as add-on of the project. Prometheus includes the telemetry solution and the database system. Prometheus covers the three-main requirement, versatile with a large number of plugins, resilient and scalable. The license model is suitable for a Research project and it's tightly integrated with an existing database.

From this analysis, we conclude that the most promising technology to use is Prometheus. The software maintenance aspects are well covered, the legal issues are none existing with an Apache 2 license model, and all the technical requirements are met. Therefore, Telemetry will be handled by Prometheus. Initial experiment of integration of an existing solution relying on Telegraf have backed this initial choice. A simple translation between the existing Telegraph out and a Prometheus exporter has allowed to integrate at minimal cost the Telegraf communication system within the Prometheus set-up.

2.4.5 Monitoring Display

Live monitoring Grafana is the de facto standard for dashboard building with time series. We plan to have ADMIRE's Monitoring System to be interfaced with Grafana. Prometheus, Telegraph and the Collectd version used in Barrel-eye are fully operational with Grafana. In terms of license mode, Grafana provides a free of charge version and pay for feature extension. Initial tests with the free version of Grafana were positive, thus ADMIRE will support Grafana as the main choice for its display system. However, we will investigate alternative in case the licensing options of Grafana would become more restrictive. Netdata [6] is the main candidate to replace Grafana. Netdata is open source with an GPL v3 license model with support from a large community and state-of-the-art graphical features.

2.4.6 Analytic Module

Off-line Analysis Flame graph is a convenient representation for performance issues, it displays the call graph jointly with temporal information. This is an efficient mechanism to single out hotspots. We consider that Flame Graph should be a part of the available off-line analysis, but not the only possible representation. Specifically, clustering techniques to detect similarity in Performance data should have a graphical output.

Prometheus embeds its query language PromQL [7] which offers high-level data manipulation operations. PromQL is designed for time series and supports advanced commands. For instance:

```
    predict_linear(node_filesystem_avail_bytes{job="node"}[1h], 3600).
```

takes the metric and uses linear regression to extrapolate forward to its likely value in the future. Additionally to the data processing, alert and notification can be added to the query would the result reach a peculiar value.

[3] https://collectd.org/
[4] https://www.influxdata.com/time-series-platform/telegraf
[5] https://developer.lightbend.com/docs/telemetry/current/plugins/prometheus/prometheus.html
[6] https://www.netdata.cloud
[7] https://prometheus.io/docs/prometheus/latest/querying

```
  - name: node.rules
rules:
- alert: StorageCapacityExhaustedIn4Hours
  expr: predict_linear(node_filesystem_free{job="node"}[1h], 4*3600) < 0
  for: 5m
  labels:
    severity: page
```

The code displayed above implements an alert based on linear extrapolation. The natural language version of the code would be: Send an alert if based on the data rate observed during the last hour the storage will be filled up within the next 4 hours. The command avoids false positive, as test predict_linear will be called twice over a 5-minute interval before sending the alert.

Consequently, the Analytic Module will be in charge of providing the notification mechanism foreseen in ADMIRE. We do expect to deliver a library of functional block to ease the construction of more complex, tailored notification for ADMIRE end-users.

2.4.7 Empirical Performance Modeling

To predict the future performance of a particular job in terms of computational and I/O resources, the ADMIRE software stack will generate and utilize empirical performance models. In particular, the tool Extra-P (see Section 3.4.3) will be used to derive empirical performance models describing the performance behavior of a job, expressed in terms of a metric such as execution time and execution parameters (e.g. the number of processes) change. In general, human-readable performance models, such as those resulting from analytical reasoning, are one of the most powerful and insightful ways of describing and understanding the performance behavior of applications. However, the main obstacles of analytical approaches are the required expertise and the large amount of effort spent to gain insights. Extra-P automates this process by conducting a series of small-scale experiments, varying execution parameters, to obtain an empirical data basis to create performance models of each function of an application.

In the scope of the ADMIRE project and as a part of the Analytic Module, empirical performance models for both computational and I/O activities of an application will be generated on demand as soon as predictions are needed. This could be the case, for example, once the malleability manager from WP3 needs information about the scaling behavior of an application to take the best malleable decision regarding the scheduling goals (system throughput, energy consumption, etc.).

Chapter 3

Sensing probes definition

In this chapter, we present more extensively each probe in the system. We describe both the nature of collected data and how they are made persistent. We will first cover node-level monitoring and then move on to the applications' state. Moreover, we introduce extra components for online data reduction, improving measurement resolution on a timescale. Eventually, we will conclude this Chapter with more verbose and necessarily punctual measurements aimed at (1) describing internal application behavior and (2) providing a model for I/O performance projection.

3.1 Node-Level Tracking

Node-level tracking covers the overall system state and, more abruptly, everything which is outside the monitored application. As such data can easily become verbose as it encompasses a wide range of parameters that we wish to expose to the IC, we cannot provide data in the form of traces but instead, we need to do a form of data aggregation. It means that samples cannot have a very high resolution for every metric all the time. Instead, and it is the object of this Section, we need to have a representative overview of the resource usage on the system with an acceptable delay (tens of seconds). Still in some cases, for example, bandwidth metrics, the IC might be interested in more verbose data. In this latter case, the Tree-Based Overlay Network (TBON) infrastructure, that we will describe in Section 3.3 , is used to provide higher temporal resolution by sacrificing spatial resolution (space-time trade-off).

To rely on an open infrastructure, leveraging existing work addressing such issues, we propose to rely on the Prometheus [7, 27] Time Series Data-Base (TSDB). This database, used globally in cloud infrastructure, comes with a rich set of data-collector and a rich analysis tools corpus. It means that developing tools targeted at Prometheus and capable of meeting HPC standards and APIs is also opening these tools to a wider audience while focusing our efforts on actual data-collection issues instead of reinventing the wheel.

3.1.1 Storing Metrics in a Time-Series Data-Base (TSDB)

There are several means of monitoring a set of nodes such as Nagios [2], Collectd [6], Telegraf [8, 23], and Prometheus. What made this last option appealing, is that it features several data-collector and is associated with a dynamic ecosystem of tools and users. Moreover, in ADMIRE, we try to expose as much data as possible from the various components of the system to create awareness to allow a more informed decision. It means, we want data to be accessible in many forms to enable WP6's IC to perform some data mining. This advocates for a monitoring solution providing some data-analytic capabilities, and Prometheus is featuring PromQL which is a query-language targeting time-series. Overall, Prometheus is providing both a rich set of measurements and allows us to exploit data in a transverse manner inside the Monitoring Manager. Other approaches do not provide such flexibility and may require additional components to reach the same capabilities, for example, Nagios has to be coupled with InfluxDB (a TSDB) to generate temporal views. Retaining Prometheus is then choosing a well-known monitoring infrastructure that has the advantage of being relatively easy to deploy.

As presented in Figure 3.1, the Prometheus server relies on three main components. First, and at its core, we find a Time Series Data-Base (TSDB) in charge of persisting performance measurements over time. This

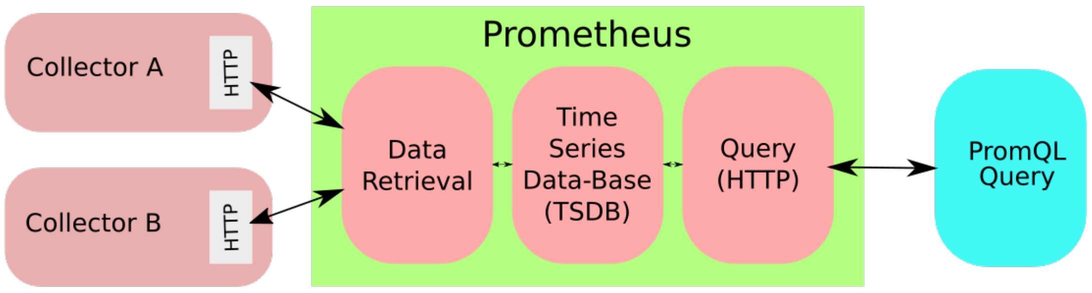

Figure 3.1: Overview of Prometheus architecture.

database, stored per node in the local file system, is fed thanks to a data scrapping module collecting the information from various *collectors* which are practically HTTP endpoints. It means that periodically, the server will query all data sources (*collectors*) to enrich the database, aggregating all entries in its TSDB. The last component in Prometheus is the query interface, exposed through its HTTP server and allowing arbitrary queries thanks to the PromQL language.

As we have seen, the node-level monitoring infrastructure can be addressed by Prometheus capabilities. However, we will need to feed data inside this TSDB, and this is where node-level instrumentation efforts will be deployed. Indeed, the project features multiple ad-hoc storages (GekkoFS, DataClay, Hercules), and we will have to export their metrics to Prometheus. This will materialize itself in the development of a Prometheus *exporter* featuring metrics of interest, based on the expertise from developers of each of the pre-cited ad-hoc storage solutions. What is appealing in this approach is that the *exporter* itself won't be tied to ADMIRE as it will be available for other monitoring infrastructure. Indeed, there is already an interest for Prometheus to monitor HPC machines and HPC oriented exporters are already available (see Table 3.1). Moreover, this will be a direct open-source contribution to the project's storage layers, enabling a new monitoring feature.

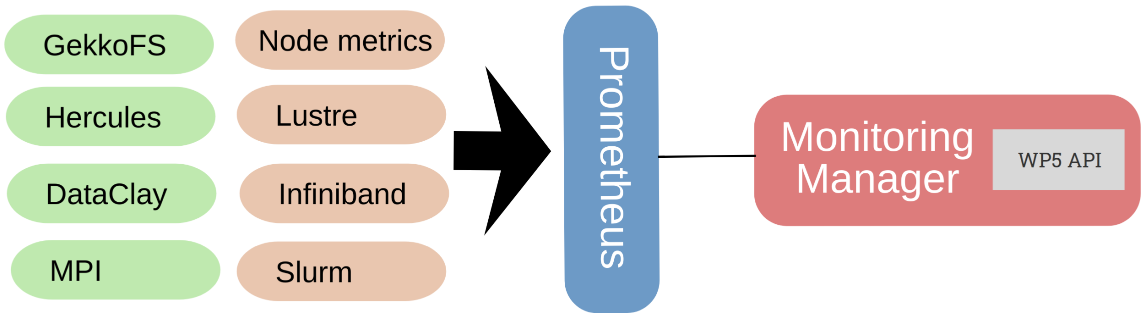

Figure 3.2: Prometheus' integration in ADMIRE.

ADMIRE will develop new *collectors* for the ad-hoc storage systems and runtimes present in the project. These new components are depicted green in Figure 3.2. Moreover, we will assess and leverage existing *exporters* (depicted in orange) to increase our performance coverage, targeting non-exhaustively, node-level metrics, GPU metrics, high-speed network metrics, etc. This rich set of measurements should be a facilitator when building performance models, both at the level of WP5 but also centrally inside the Intelligent Controller. Developing our exporters for ad-hoc storage layers will outline new metrics which were previously unreachable. For example, we will be interested in caches inside the I/O subsystem, anticipating their exhaustion. Similarly, we will correlate I/O request rate and network bandwidth saturation, trying to derive new combined performance indicators.

The Monitoring Manager, red in Figure 3.2 will then be able to aggregate data from the TSDB by issuing queries to respond to WP6's IC through its API. Such queries will be able to target varying time ranges, enabling retrospective analysis. This manager will then act as both a data-aggregator and a controller for the measurement process. Indeed, and as we will further detail in the next Sections we also plan to rely on more verbose instrumentation techniques.

3.1.2 I/O Runtimes State

As far as ad-hoc I/O runtimes are concerned, we will develop corresponding Prometheus exporters. These exporters shall, as aforementioned, eventually be part of the respective ad-hoc storage solutions. Overall, an exporter is a relatively simple code, particularly if we rely on one of the existing Prometheus client libraries to provide the glue with the TSDB. In fact, we estimate most of the design effort to be turned towards defining the actual instrumentation points in the code-base. Indeed, we would like to expose metrics that are usually out of reach, outside a holistic framework such as the one in ADMIRE. Therefore, we would like to track the following elements (non-exhaustive):

- Internal cache levels to predict potential exhaustion;
- Meta-data operation count and their nature as such operations are often the weak point in parallel file-systems;
- Estimated data-transfers reads and writes in various categories (local cache, back-end file-system, ...);
- Blaming of the corresponding media for data-transfers (to local SSD, to network, to pNFS, ...).

Naturally, it is of interest to converge towards a relatively unified set of counters between our ad-hoc systems, and the final goal is to be able to describe each of them. Still, the Monitoring Manager will act as an adaptation layer from WP5's API consumers' point of view to provide a unified interface, even if there are slight variations.

3.1.3 Node-Level State

As far as monitoring the node is concerned, we will rely on stock *exporters*. Indeed, this task is regular for Prometheus. Looking at the exporter list, we can already identify interesting data sources as we already illustrated in Figure 3.2.

Name	Description
Lustre	exporter for Lustre metrics
Slurm	exporter for Slurm status
Infiniband	exporter for Infiniband HCAs
Nvidia	exporter for Nvidia GPUs
GPFS	exporter for IBM GPFS
node_exporter	exporter for node-local metrics (hardware)

Table 3.1: List of exporters which will be considered for integration.

As presented in Table 3.1, Prometheus already provides a rich set of measurements without requiring specific developments from our side. It means that, for example, we may combine any of these existing metrics with the new ones we will develop to derive new metrics leveraging PromQL. This is then both a way to rely on existing proven technologies while ensuring our efforts are focused on what is missing in the node-level monitoring field. We will make HPC tools available in the Prometheus ecosystem, adding specifically designed probes. Moreover, it is not excluded that we use other collectors (not listed in Table 3.1) or that we develop other ones depending on future needs we may identify.

3.1.4 Application State

As previously mentioned, application state in ADMIRE is seen as a more verbose data-source. It means that we do not intend to generate time-series out of it. Still, we would be interested in – applying the same logic as what we do for I/O – extracting metrics from some runtimes. A common runtime is the Message-Passing Interface (MPI) runtime, which is used to exchange message between multiple nodes. MPI features a very dynamic tools infrastructure which has recently evolved with callback capabilities in MPI 4.0 [13]. Moreover, the MPI forum is currently devising QMPI [10] which is the successor of the PMPI interface which was previously based on weak-symbols for loader interposition. With QMPI in the near future, multiple tools will be able to attach to a given MPI runtime instance. This would circumvent the pre-loading limitations preventing multiple tools [25] or runtimes to intercept common calls.

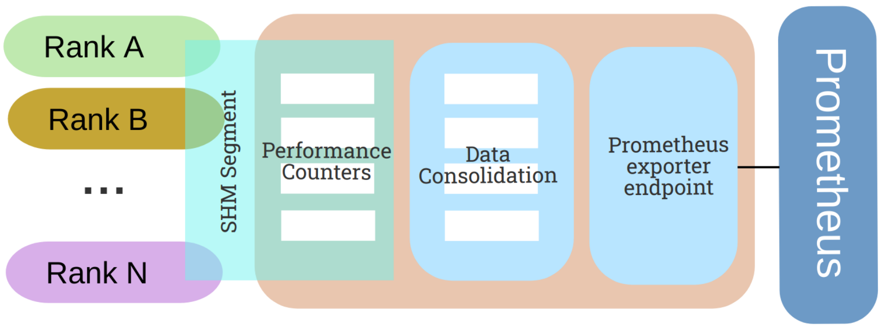

Figure 3.3: MPI collector for Prometheus.

As shown in Figure 3.3, we plan to develop an MPI *collector* for Prometheus, made of two components. First, libraries either preloaded (PMPI case) or loaded by the MPI runtime (QMPI case) and an aggregator acting as the *collector*. In Figure 3.3, libraries are referred as *Performance Counters* and the aggregator is in charge of performing *Data Consolidation*. Our goal is then to be able to track various MPI calls and their intensity on a per-node basis without interfacing in HTTP to individual MPI processes, which can be numerous. To limit performance overhead, we plan to rely on a shared-memory segment to exchange counters. In addition, we will explore means of exposing MPI-T variables [15, 22] – describing runtimes internals. This approach will then open MPI to the Prometheus monitoring world and will be compatible (if recompiled) with all MPI runtimes.

3.2 Tracking Applications

When a given program runs, it can sometimes be of interest to profile it in a more verbose manner, for example to determine the I/O interfaces which are in use or to blame sources of I/Os at the source-code level. For example, one may consider a cluster which faces adverse I/O patterns from some applications (i.e. creating thousands of files), it would be possible to attach and extract source line information to come back to the developer. The same may apply for inefficiencies in the I/O pipeline, such as random access patterns or small blocks. What will be observed macroscopically through the I/O server status, will have to be blamed to the application with as much precision as possible. Indeed, I/Os are transverse in an HPC system and it is not easy to pinpoint a library or a given line of code in a large corpus of code owned by multiple individuals.

This Section will present tracks to implement such monitoring tools. We will first explain why it is not practical to use standard tools (e.g. `LD_PRELOAD`) in this case, and why we opted for dynamically attached tools. In a second time, we will present our prototype tool, used to study the feasibility and conclude with multiple tracks that we would like to explore as the development goes.

3.2.1 On the Challenges of Always-on Monitoring

In ADMIRE, one of the main goal of WP5 is to maintain a control and measure feedback-loop between all of its components. When it comes to applications, the common instrumentation mean is to either pre-load the interception library or link/compile an instrumented version of the code. This naturally leads to (1) specific launch and compilation instructions and (2) potential overhead associated with the measurement. This second point could be mitigated using selective instrumentation (bypasses in the instrumentation wrappers) but the first point would remain.

Besides, some I/O frameworks rely on pre-loading to capture the I/O interface, it means that our tool should never compete with the I/O layer when trying to measure the I/O layer itself. It is the reason why inserting the tool inside the application was seen as a potential source of issues. Instead, information about the I/O layer would be retrieved at the I/O API level. Still, we do not want to give-up on metrics relative to the application state, and therefore, we had to devise alternative approaches as we will present in the next Section.

3.2.2 Dynamic Instrumentation

In ADMIRE, we would like the IC to order us to trace a given application, for example, as there is a strange I/O behavior on the node. Then the IC, will call the Monitoring System API and ask the Monitoring Manager to perform measurements. Practically, it means that an uninstrumented program will be instrumented for a short period of time. Naturally, this should be transparent for the end-user, indeed the code may slow-down temporarily but once detached, the program will resume its regular course.

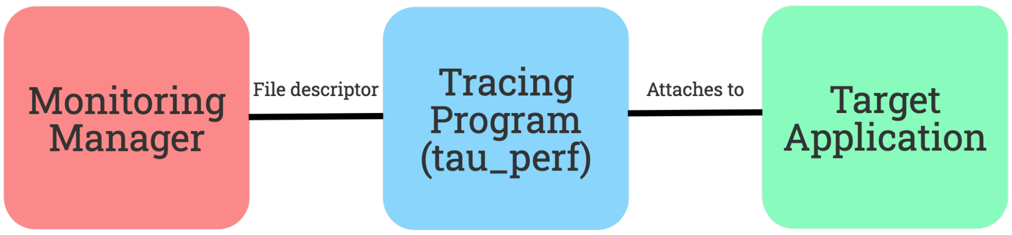

Figure 3.4: Illustration of dynamic instrumentation capabilities.

As presented in Figure 3.4, We would then like to behave as a debugger, attach to the process to monitor its doings, and detach when it is not relevant anymore. This would allow us to punctually inspect the program's behavior and disconnecting from it afterward. To experiment with this idea, we developed a prototype external I/O tracer with limited features. As a starting point, we considered syscall tracing with ptrace, a simple and relatively portable way of monitoring a target program. Doing so led to a behavior somewhat similar to what strace does, but only targetting I/Os. As soon as the tracing process will attach, it will register on syscalls and place accordingly breakpoint on functions of interest. The advantage of this model is that it does not require any form of previous knowledge or instrumentation in the binary, easing deployment. Then, all syscalls will be trapped and inspected. For functions, one could use either hardware or emulated breakpoints as GDB does to trace calls. In addition, again similarly to GDB, it is possible to capture the call-stack each time an event occurs, providing Event-Based Sampling capabilities outside the program in a trivial manner. In terms of implementation, as shown in Figure 3.4, we plan to have the Monitoring Manager forking an instance of the *tau_perf* tool specifically developed for the purpose. This tool then attaches to the target program. Dealing with communication back to the manager, we will provide a JSON data-stream through a preset file-descriptor (pipe) at the Monitoring Manager level.

Despite having the advantage of tracking every syscalls, this approach may lead to some overhead when attached as ptrace relies on signaling between the tracer and the tracee. This motivated us to look for more advanced interfaces that will be object of the next Section.

3.2.3 Foreseen Dynamic Instrumentation Interfaces

Linux systems have evolved in terms of measurement capabilities. In particular, the performance measurement interfaces available in the kernel multiplied and new features for syscall trapping were added. In particular, new instrumentation support, considering either Linux Perf or the Extended Berkeley Packet Filter (eBPF [5]) may provide valuable system level probes to feed our measurements. Moreover, Linux Perf has some attach and detach capabilities which would match what we achieved with ptrace but without the signaling overhead,

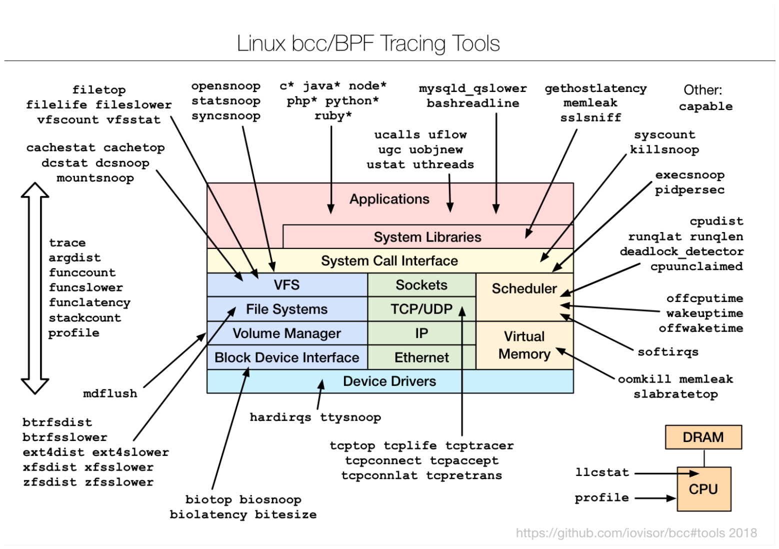

Figure 3.5: Observation points within the kernel as supported by Linux and its BCC tools suite.

samples being collected with the help of the kernel. As far as eBPF is concerned, it supports code injection to execute commands when certain rules are met, this is a bare metal manner of reacting to any system event. One can see how such model would be of interest to raise alerts for example. These more advanced interfaces and associated scenarios will be considered in the pursuit of the development of the dynamic application tracing tool to enrich its output while limiting its overhead.

Linux already supports a large variety of tools to observe the kernel stack, as illustrated in figure 3.5. Most of these tools are already built on the top of eBPF which offers as well the option to write code to fit more specific needs. The short code excerpt provided below is an excerpt of Brendan Gregg's blog [1].

```
BEGIN
{
    printf("Tracing block device I/O... Hit Ctrl-C to end.\n");
}

kprobe:blk_account_io_start
{
    @start[arg0] = nsecs;
}

kprobe:blk_account_io_completion
/@start[arg0]/

{
    @usecs = hist((nsecs - @start[arg0]) / 1000);
    delete(@start[arg0]);
}
```

The code will be executed within the Kernel. The execution is actually made within a Virtual Machine thus preventing the risk of kernel crash. The second security aspect is that the code is loop-free. The semantic of eBPF script does not support return edge in the control flow. Therefore, the code is guaranteed to end, thus preventing the risk of Denial of Service.

We plan to investigate eBPF or eBPF tools is order to retrieve performance metrics not only at the application level but as well within the Kernel.

3.3 Data Reduction Using a Tree-Based Overlay Network (TBON)

For now, we have presented (1) node-level metrics with Prometheus and (2) application-level metrics with the *TAU_perf* tool. Both these measurements were considered due to both their low-overhead nature and their descriptivity. Still, in some cases more verbose measurements might be needed as we will further elaborate on in this Section.

[1] https://www.brendangregg.com/blog/2019-01-01/learn-ebpf-tracing.html

3.3.1 Motivations

We now propose to consider the case when the system has to react to an abrupt change in performance. It means, that in complement of the medium resolution measurement at node-level, and of the punctual measurement inside the application, an intermediate set of metrics is needed. All our current metrics are spatially discrete, it means that we generate a set of points for individual components (nodes, applications). However, in this Section, we will cover a measurement which has *spatial reduction* at its basis. Thanks to this space-time trade-off, we will be able to have less spatial resolution while greatly improving the temporal resolution.

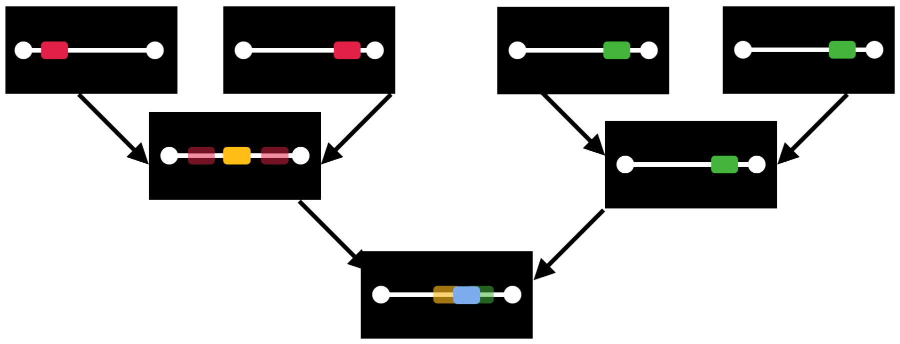
Figure 3.6: Sample Tree-Based Overlay Network averaging data-points.

To do so, and as presented in Figure 3.6 we propose to leverage what is called in the performance measurement field a Tree-Based Overlay Network (TBON) [4, 21]. Such network is simply a static reduction network overlaid atop a running process to return a constant stream of data-points. Considering the wide variety of probes we will have at node-level, we will spatially reduce some of them to obtain some system-wide measurement points with high temporal resolution.

3.3.2 Proposed Implementation

If we look at Figure 2.2, an instance of the Monitoring Daemon will run on each node. This daemon will be in charge of providing the WP5 API (see Section 4) by interfacing to previously mentioned measurement sources. We plan to leverage all these instances of the Monitoring Daemon on the whole supercomputer to implement the TBON. We will use a discovery mechanism (file-system or WP6 database) to interconnect all these instances in a tree-based topology. Data points will be sent at a regular pace to be reduced towards the tree root. Viewed from the root, we will obtain a time series of system-wide performance. The IP connection layer will be leveraged for this purpose. The main reason for this is that on most super-computers, the high-performance network is usually seconded by an administrative network relying on Ethernet. Using regular TCP connectivity is then a way to decouple traffic, trying to limit performance interference. Moreover, this traffic should be relatively low even compared to of-shelf interconnects thanks to its hierarchical nature. To illustrate this scalability, consider that we are willing to reduce n metrics of s bytes each, in a binary tree each node would have to receive two-stream of data (from children) and send up one stream consisting of incoming data contributions complemented with local metrics. We can easily model the resulting overall bandwidth (in/out) in bytes per second for a given measurement period τ as $B(n, s, \tau) = 3ns\frac{1}{\tau}$.

As shown in Figure 3.7, TBONs are very efficient in terms of bandwidth while allowing interesting temporal resolution. In this example, we consider 100 metrics of 8 bytes each (doubles or 64 bits integers). For example, a bandwidth budget of 100 kB/sec allows for a temporal resolution τ of 23.44 millisecond. If we now compare this figure to the default scraping period in Prometheus, 60 seconds[2], such infrastructure provides non-negligible improvements in terms of temporal resolution (at the cost of the spatial one).

Note that we may have to adapt the collectors implemented in the project to provide such high-frequency endpoints, it means that we may consider additional means of transferring data-points if the HTTP request is too expensive. Developments done in the context of the MPI collector and its local data-reduction framework

[2]We plan to use lower values in the context of the project (around 5 seconds).

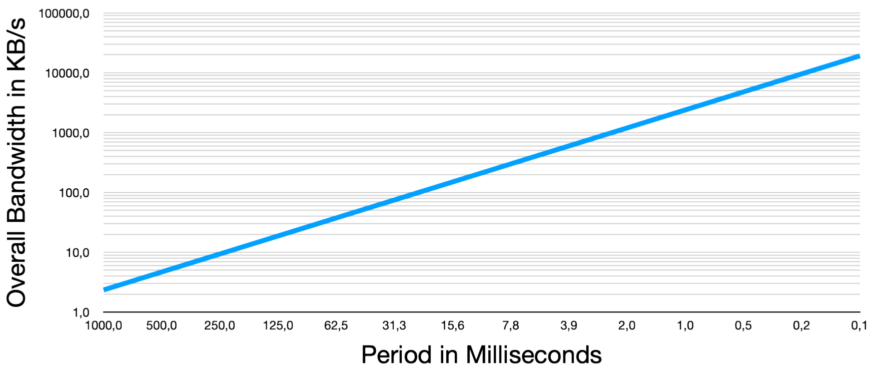

Figure 3.7: Resulting per-node (in/out) bandwidth in KB/s for reducing 100 entries of 8 Bytes in function of τ.

(see Section 3.1.4) may lead to a common infrastructure to share data-points, in particular alternative coupling methods with improved performance will be considered (shared-memory segments, Cross-Memory Attach (CMA), message queues, ...).

3.3.3 Resulting Metrics

The ability to generate high-resolution time series can be of importance when it comes to taking decisions in terms of overall performance. Indeed, as previously discussed, the TBON will only provide aggregated performance. Still, it will give insight on the peak system performance which is the main metric of interest in HPC. In machine learning it is of paramount importance not only to have many values to build the interference model but also to have some key fitting metrics to train a given model. The TBON has been devised to provide such key metrics, they will be relatively compact as we do not plan to have more than a hundred system-wide counters. These counters will give in close to real-time some key performance indicators, among which we find:

- Overall read and write bandwidth
- Total network bandwidth (derived from exporters)
- Number of active data-streams
- Overall used and free space on each storage tier
- Overall caching capabilities for each storage tier

These aggregated metrics and their meaningfulness will be the object of discussions with the modeling components part of WP6. Moreover, during our exploration leading to the development of the individual collectors we will determine what are the valuable metrics thanks to exchanges with the developers of each individual components.

3.4 Discrete Data-Sources

Up to now, we have covered mainly the always-on Monitoring System, first with Prometheus and then with sampling tools capable of attaching to a running program. Indirect means of measuring running applications are privileged not to interfere systematically with all binaries running on a given system. Indeed, doing so would be a source for portability issues as mentioned in Section 3.2.1. Still, it is interesting to punctually run full-fledged tools to capture the best picture possible of a given application – this is the object of the current section.

3.4.1 Articulating Punctual Measurements

Both the TAU performance system and the tool Extra-P will run in the background. TAU will be used punctually to derive more verbose performance profiles using its state-of-the-art instrumentation chain. And Extra-P will be leveraged in a more indirect manner, reading measurements from the performance database (including TAU profiles) and then generating performances models, also stored for later use.

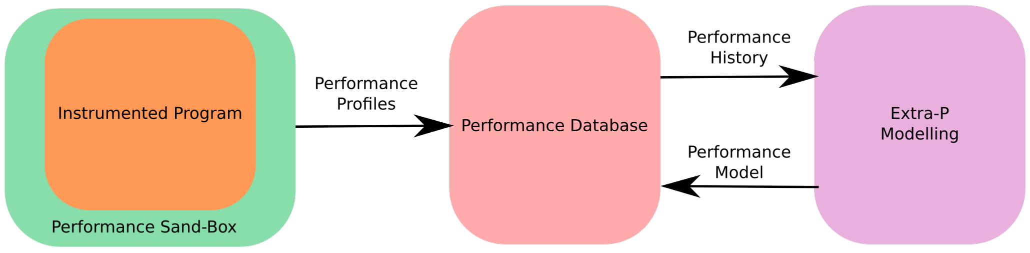

Figure 3.8: Sample data-flow for asynchronous measurements.

As shown in Figure 3.8, both TAU and Extra-P will interact with the performance database. Extra-P will be consuming performance data (of any kind) to generate a performance model. With TAU things are a bit more complex as TAU is a data-collector, meaning it runs with the source program to generate profiles. Practically, we may ask users to punctually run a TAU-instrumented binary to generate results in the performance database. However, we are willing to explore means of doing *performance sand-boxing*. Indeed, users generally spend time debugging their programs as otherwise, they may not work, still, performance measurement is often a side-concern as it is only a medium obstacle to producing the results. Yet, if we turn the table and look from a whole-machine point of view, running inefficient programs, for example, over-scaled with 5% efficiency, is simply a waste of resources and can artificially lengthen waiting queues. Therefore, for the particular case of TAU, we will leverage Linux name-spaces capabilities more widely known as containers to re-run instrumented jobs without issuing (even accidentally) artifacts to the permanent file system. Thanks to a coarse-grained selection using the aforementioned metrics, and then by rewiring paths in a temporary container, we should be able to generate off-line profiles without users-cooperation. This way, by running hashed binaries with the same command line, we can ensure that most applications are profiled with varying parameters even without the user noticing it.

3.4.2 Performance Profiles using TAU

TAU [26] is a profiling tool that needs to run alongside the application to generate performance results. It can rely on several instrumentation methods to capture the program's behavior. First, the application might be compiled with the TAU compilation wrappers, issuing an instrumented binary (source-level rewriting, library/linker interposition [1,22]). Despite potentially providing more precise measurements, including phases and potential source-to-source transformations (see how Opari [20] used to work for OpenMP) this will not be practical in our case as recompiling the program is generally not trivial.

Instead, we are willing to explore means of runtime interposition, which is done through library pre-loading. Moreover, measurements which were previously requiring source-level instrumentation are now feasible at runtime thanks to the development of tools-interface in common HPC runtimes such as MPI-T [16,22] for MPI and OMPT [9] for OpenMP, in addition, phase information in ADMIRE should be provided by the in-application interface featured in WP7. It means that privileging runtime instrumentation won't impede measurement accuracy. One can either wrap function calls using interposition, doing *direct* instrumentation, it is also possible to *sampling*, punctually interrupting the program to capture the program counter and associated stack. It is even possible to combine both doing sampling including when entering functions of interest through *event-based sampling* [12,30]. As of today, TAU is generally used in an Event-Based Sampling approach as it is providing relatively accurate measurements without causing too much overhead and without any form of recompilation (just adding `tau_exec` to the command line).

For ADMIRE, we want to avoid excluding any kind of measurement as TAU is capable of generating several outputs ranging from a compact performance profile to a full temporal event trace in OTF2 [11]. Depending on the interactions with Extra-P and other modeling components in WP6 and WP3, we will generate what is of interest as the project unfolds – adapting measurement verbosity. As far as the interfacing with the performance database is concerned, TAU already has such a database in the PerfExplorer tool [14] which can aggregate multiple profiles to generate a cross-experiment view, in this framework, it was also coupled with multiple standard data-bases. These existing capabilities will be leveraged and extended accordingly to fit in the ADMIRE framework in order to fill the performance database.

3.4.3 Performance Models using EXTRA-P

ADMIRE includes a component named Malleability Manager. This component is developed within WP3 and allows application processes to be dynamically re-shuffled, consolidated or dispatched on a different set of computing nodes. The core idea behind malleability is to adjust the topology of an application depending on nodes failures or performance fluctuations. To allow the Malleability Mananger to choose the right configuration of an application achieving thereby the scheduling goals, the malleability manager needs to predict the computational and I/O activities of the application running on a specific number of nodes. This is where Extra-P comes in. Extra-P models the performance behavior of an application as a function of its execution parameters e.g. the number of processes or the problem size, enabling the prediction of the application performance at different scales. For that, the tool requires a set of small-scale performance experiments. More precisely, to model a parameter, Extra-P needs at least five experiments with unique values for the parameter to be modeled while all remaining execution parameters stay constant. For modeling an application considering two application parameters, for example, Extra-P requires at least nine different measurement points. Ideally, several samples are collected at each measurement point. Subsequently, the median values of the measured performance metrics are computed and used for modeling. This reduces the influence of the system noise on the performance measurements and thus helps to the created the models. In general, the more measurement points and repetitions are performed, the more accurate the created models are.

Extra-P automatically creates performance models of all instrumented application functions, allowing an in-depth analysis of specific kernels as well as an overall estimation for the entire application. Hence, the generated performance models heavily rely on the application instrumentation and the sampling performed. If a function is not instrumented or profiled, Extra-P can not create a performance model for that function as the required data is missing.

Currently, Extra-P generates performance models describing the computation and communication of an application's functions. In the scope of this project, it is intended to extend the capabilities of the tool to allow for the generation of I/O requirement models. Doing so will allow the malleability manager to take the suitable decision based on both computational and I/O activities.

To generate the previous described performance models, the profiling data needed has been addressed in this WP. Moreover, the locality of the data has been as well agreed on. Using the profiling data stored in the database, Extra-P will be invoked by the intelligent controller at specific intervals to generate for a particular application its corresponding performance models. Whenever the application is scaled (expand/shrink) new sampling data will arrive at the database, refining the generated performance models. Finally, the intelligent controller forwards the generated models to the malleability manager, aiding the manager in its malleable decision.

Chapter 4

Sensing and profiling system API

In this Section, we will further detail the Application Programming Interface (API) associated with WP5. This interface is aimed at both providing profiling and hardware discovery capabilities. In the wider context of the project, as presented in the original proposal, we are willing to implement a feedback-loop implementing *control* and *measure*. Control is impersonated by the Intelligent Controller (IC), and measure is mostly provided by WP5's instrumentation layer. This central position in the project justifies that we make sure to provide as much metrics as possible, particularly if machine-learning methods are to be leveraged. This last point motivated some of our choices for more standards methodologies to make sure we quickly forward a rich set of measurements. We will now first provide an overview of the *Sensing and Profiling Interface* design, then after a brief discussion of its implementation, we will provide a description of the individual calls constituting the resulting interface.

4.1 General API Layout

We will now have a closer look at this API. The API between the Monitoring Manager and the Intelligent Controller is built in three different layers that we are now going to detail in following paragraphs.

As WP5 is planned to deploy a daemon on each node of the system to fulfill its always-on measurement duty, it has been judged reasonable to rely on it for **resource monitoring and discovery**. The purpose of this interface is twofold, (1) it enables the discovery of the hardware resources on each node (mount-points, network cards, ...) and (2) it allows for the listing of associated performance metrics (bandwidths, free space, file-descriptors, ...). This interface will extract its data from Prometheus and from both *proc* and *sys* file-systems. It can be classified as coarse-grained monitoring.

A second aspect, closer to the applications is **node and process discovery**, in WP5 we plan to work at Process IDentifier (PID) level to reach for any process, it allows us to build tools which are not dependent on the ADMIRE meta-data layer to operate. WP5 will allow a node listing, as each node will be able to list processes and their associated resources. We also plan on enabling process status monitoring to enable WP6 to inquire of the termination of a given process indirectly. On this particular interface, it is not excluded that WP6 may absorb it later on to mirror it in its state data-base, in this later case all these information would be derived from the *application manager* (see Figure 1.1 in the WP7 Applications box).

The third part is dedicated to **application performance monitoring**. Its goal is to enable dynamic application monitoring depending on the IC's needs. Thanks to this interface, it will be possible to generate a profile during a given time slice. We also plan on providing more lightweight functions for I/O profiling storing their results as a vector-space (application behavior, I/O patterns). Vector-space is a layout particularly convenient to fill real-time models such as neural networks classifiers which might be used in WP6. A second aspect, is providing the performance model as generated by Extra-P. This model built from previous application runs will be used by WP6 and forwarded to malleability manager in WP3 which in turn will project it to current machine state to yield the best configuration possible.

The last component that we can outline is the **control interface**. This interface is solely dedicated to the Intelligent Controller (WP6) to enable process registration and sampling rate fine-tuning. Before dwelling into more details in this interface, we propose to quickly discuss the articulation of these respective interfaces.

4.2 Implementation Considerations

Before describing each call turn-by-turn, it is important to describe how this interface will be deployed and exposed, which is the goal of this section. As mentioned at the beginning of this document, WP5 will deploy two main components, (1) the *Monitoring Daemon* and (2) the *Monitoring Manager*. The Monitoring Daemon will run on each node, whereas the Monitoring Manager will *drive* all instances of the Monitoring Daemon. The tree implementing the TBON will also be used for command, indeed, when data are reduced up, it is also possible to send data down using a trivial breadth-first search routing algorithm. Doing so will prevent bottlenecks between nodes, avoiding all-to-all connectivity patterns and supporting potential broadcast commands to all nodes at once. If there is a tree, it has a root, and this is one of the main reason for differentiating the monitoring daemon and the manager, the latter featuring the public WP5 interface.

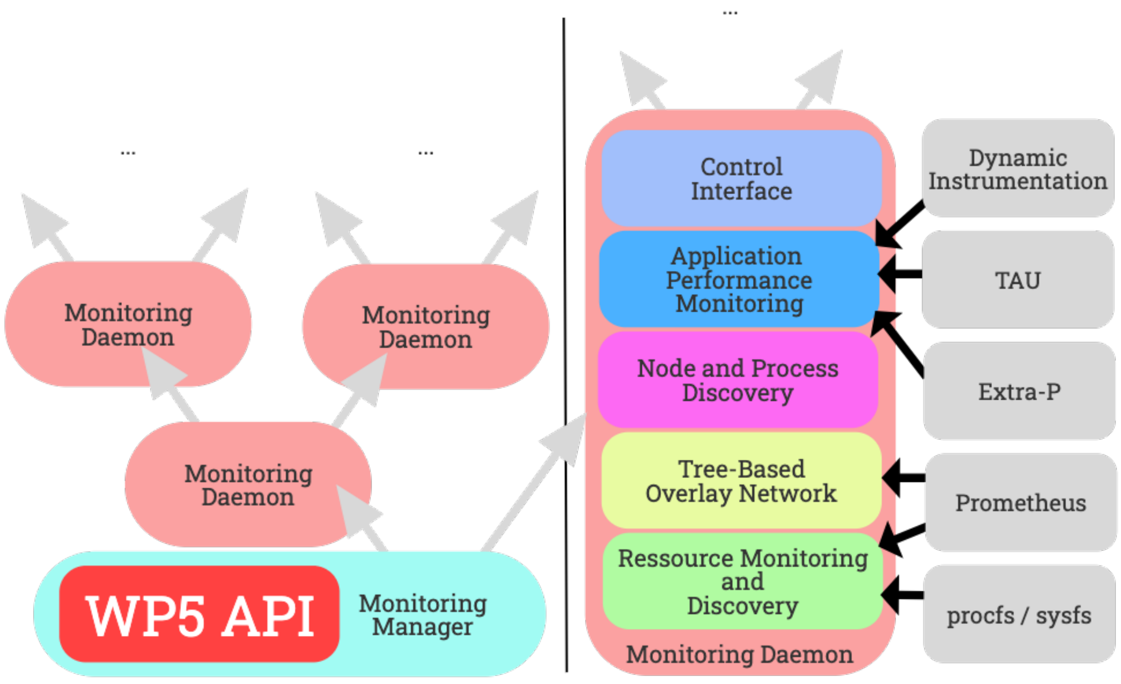

Figure 4.1: Functional diagram for the measurement facilities in WP5.

As shown in Figure 4.1, instrumentation daemons will be connected in a tree-shaped layout. At the root we will find the monitoring manager which will issue commands on the tree. Dealing with the communication between daemons we will leverage HPC class RPCs (through the Mercury RPC stack), same thing for the WP5 API which will also be made available through this high-performance conduit. However, we consider as important to also provide a TCP exposure, particularly given the reduced amount of data which will circulate the tree. Moreover, for fault-tolerance it is critical to have a connected protocol in place to be able to identify quickly and reliably failed nodes. Indeed, in HPC network stacks involving OS bypass capable devices, failure is often only partially handled, making fail-over more difficult. In our case, as we implement an always-on daemon, it is crucial to stay up and running including in some adverse cases such as a node promptly rebooting – the IP layer will play an important role in mitigating such cases. As a side effect, having IP to initially build the system (while considering an alternative in the design) will allow faster prototyping and also portability to most systems which generally feature an IP layer. Eventually, this opens the systems status to punctual query through IP if needed, for example, directly addressing a given node instead of using the whole tree, the monitoring manager may even implement such bypass. Overall, despite this document mainly discussed the scalability of our measure, scalability of control has been considered in how monitoring will be deployed.

4.3 Resource Labeling

In the rest of this Chapter, the various functions will be targeted generally at a given PID running on a given node, it has been considered that such identifier could be summarized in a single union targeting a 64bits integer, with the most significant bits, addressing the node and the least significant bits the PID. This has the advantage of simplifying our interface gathering two parameters in a single one. Moreover, these two parameters are strictly positive, meaning that we decide by convention to consider zero as a wildcard. It has the drawback of breaking the C counting convention, but conversely has the advantage of using the full 32 bits dynamic associated with each entry by dropping the sign bit. Therefore, we can outline the `ADM_profile_id_t` node identifiers using the following code excerpt:

```c
/**
 * @brief Descriptor for a given Node or PID
 * @note 0 is a wildcard
 *
 */
typedef union
{
    struct
    {
        unsigned int node_id;   /**< Target Node-Id */
        unsigned int pid;       /**< Target PID */
    }members;
    uint64_t id;                /** Encoded Node + PID */
}ADM_profile_id_t;
```

Ih the following sections, the *Doxygen* code will be presented jointly with an algorithmic pseudo-code of each API functions.

4.4 Resource Monitoring and Discovery

All the Resource Discovery functions are to be *c*alled from the Intelligent Controller to the Monitoring Manager. All these functions are of type *P*ull: information to be extracted to the Intelligent Controller from the Monitoring Manager.

Functions dedicated to Persistent Memory Resources (Storage)

```c
/** @brief this is the system-wide handle for a storage resource */
typedef uint64_t ADM_profile_storage_res_id_t

/**
 * @brief List storage devices on a given Node
 *
 * @warning Returned values are static arrays not to be freed
 *
 * @param[in] id target Process/Node ID
 * @param[out] count number of output elements
 * @param[out] res_ids array of storage resources ID
 * @param[out] bandwidth array of bandwidth
 * @param[out] mountpoints array of mountpoints
 * @param[out] fstype array of mountpoints types (as in fstab)
 * @return int non null on error
 */
int ADM_resStorList(ADM_profile_id_t id,
                    size_t * count,
                    const ADM_profile_storage_res_id_t **res_ids,
                    const uint64_t ** bandwidth,
                    const char ** mountpoints,
                    const char ** fstype);

/**
 * @brief Return current state of a given storage device
 *
 * @param[in] id target Process/Node ID
 * @param[in] res_id resource identifier to query
 * @param[out] used_space
 * @param[out] overall_space
 * @param[out] timestamp
 * @return int non null on error
 */
int ADM_resStorStat(ADM_profile_id_t id,
                    ADM_profile_storage_res_id_t res_id,
                    uint64_t *used_space,
                    uint64_t *overall_space,
                    uint64_t *timestamp);
```

Name: *ADM_resStorList*
input : $ADM\_profile\_id\_t$ id
InOut : $size\_t*$ count
InOut : $ADM\_profile\_storage\_res\_id\_t**$ res_ids
InOut : $String*$ bandwidth
InOut : $String*$ mountpoint
InOut : $String*$ FS type (ramfs, nvme ...)
output: int error_code
Description:
This function lists all the storage resources available on a given node, the resources are detailed by their key attributes. In case of failure an error code is returned.

Name: *ADM_resStorStat*
input : $ADM\_profile\_id\_t$ id
input : $ADM\_profile\_storage\_res\_id\_t$ res_ids
InOut : $uint64\_t*$ used_space
InOut : $uint64\_t*$ overall_space
InOut : $uint64\_t*$ timestamp
Description:
This function returns capacity related information for a given storage resource on a specific node. The function fills up all the data structures passed as argument to describe the current state of the storage device. A call to this function returns 0 for a success or an error code.

Functions dedicated to Memory Resources

```
/** @brief this is the system-wide handle for a memory resource */
typedef uint64_t ADM_profile_memory_res_id_t

/**
 * @brief List memory devices on a given Node
 *
 * @warning Returned values are static arrays not to be freed
 *
 * @param[in] id target Process/Node ID
 * @param[out] count number of output elements
 * @param[out] res_ids array of memory resources ID
 * @param[out] kind array of memory device kinds
 * @param[out] is_numa array of booleans indicating NUMA effects
 * @param[out] attached_core_ids array of arrays listing attached cores
 * @return int non null on error
 */
int ADM_resMemList(ADM_profile_id_t id,
                   size_t * count,
                   const ADM_profile_memory_res_id_t **res_ids,
                   const uint64_t ** bandwidth,
                   const char ** kind,
                   const int * is_numa,
                   const int ** attached_core_ids);

/**
 * @brief Return current state of a given memory device
 *
 * @param[in] id target Process/Node ID
 * @param[in] res_id resource identifier to query
 * @param[out] used_space
 * @param[out] overall_space
 * @param[out] timestamp
 * @return int non null on error
 */
int ADM_resMemStat(ADM_profile_id_t id,
                  ADM_profile_memory_res_id_t res_id,
                  uint64_t *used_space,
                  uint64_t *overall_space,
                  uint64_t *timestamp);
```

Name: *ADM_resMemList*
input : $ADM\_profile\_id\_t$ id
InOut : $size\_t*$ count
InOut : $ADM\_profile\_memory\_res\_id\_t**$ res_ids
InOut : $uint64\_t**$ bandwidth
InOut : $String*$ Kind
InOut : $boolean*$ is_numa
InOut : $int**$ attached_core_ids
output: int error_code
Description:
List all memory devices and precise their nature for the node passed in parameter. Where the nature of a memory device could be either HBM (High Bandwidth Memory), swap, RAM or other type to be defined. The function returns 0 for success or an error code otherwise.

Name: *ADM_resMemStat*
input : $ADM\_profile\_id\_t$ id
input : $ADM\_profile\_memory\_res\_id\_t$ res_ids
InOut : $uint64\_t*$ used_space
InOut : $uint64\_t*$ overall_space
InOut : $uint64\_t*$ timestamp
output: int error_code
Description:
Returns the current state of the specified memory device on a given node. The state is provided by the pass-through pointer arguement. The function returns 0 for sucess or an error code in case of failure.

Functions dedicated to Networking Resources

```
/** @brief this is the system-wide handle for a network resource */
typedef uint64_t ADM_profile_network_res_id_t

/**
 * @brief List network devices on a given Node
 *
 * @warning Returned values are static arrays not to be freed
 *
 * @param[in] id target Process/Node ID
 * @param[out] count number of output elements
 * @param[out] res_ids array of storage resources ID
 * @param[out] bandwidth array of network device bandwidth
 * @param[out] kind array of network device kinds
 * @param[out] address array of underlying network address
 * @param[out] netmask array of underlying network masks
 * @return int non null on error
 */
int ADM_resNetlist(ADM_profile_id_t id,
                   size_t * count,
                   const ADM_profile_network_res_id_t **res_ids,
                   const uint64_t ** bandwidth,
                   const char ** kind,
                   const char ** address,
                   const char ** netmask );

/**
 * @brief Return current state of a given network device
 *
 * @param[in] id target Process/Node ID
 * @param[in] res_id ressource identifier to query
 * @param[out] total_received number of bytes received
 * @param[out] total_sent number of bytes sent
 * @param[out] estimated_bandwidth estimated current bandwidth
 * @param[out] timestamp measurement timestamp
 * @return int non null on error
 */
int ADM_resNetStat(ADM_profile_id_t id,
                   ADM_profile_network_res_id_t res_id,
                   uint64_t *total_received,
                   uint64_t *total_sent,
                   uint64_t *estimated_bandwidth,
                   uint64_t *timestamp);
```

Name: *ADM_retNetList*
input : $ADM\_profile\_id\_t$ id
InOut : $size\_t*$ count
InOut : $ADM\_profile\_network\_res\_id\_t**$ res_ids
InOut : $uint64\_t**$ bandwidth
InOut : $string*$ kind
InOut : $string*$ address
InOut : $string*$ netmask
output: int error_code
Description:
List networking devices available on a given node. The networking devices can be of several kinds among HCA (InfiniBand), BXI, OFA, ETH. In case of success the function returns 0, otherwise an error code is returned.

Name: *ADM_resNetStat*
input : $ADM\_profile\_id\_t$ id
input : $ADM\_profile\_network\_res\_id\_t$ res_ids
InOut : $uint64\_t*$ Total data received
InOut : $uint64\_t*$ Total data sent
InOut : $uint64\_t*$ Estimated Current Bandwidth
InOut : $uint64\_t*$ timestamp
output: int error_code
Description:
Returns current usage on the given network device. This status function is, as all status functions of the API, a resource description function not to be used for performance measurement. For instance the estimated bandwidth is based on initial tests at the booting of the node and are not reflecting potential congestion occurring the network at a given time.

4.5 Process Discovery

```
/**
 * @brief Retrieve the list of tracked PIDs running on a given node
 *
 * @param[in] node_id Target node (PID component is ignored)
 * @param[out] pids PIDs running on the target node
 * @return int non null on error
 */
int ADM_pmList(ADM_profile_id_t node_id, ADM_profile_id_t * pids);

/**
 * @brief Retrieve the list of tracked nodes
 *
 * @param[out] nodes List of nodes (PID being 0)
 * @return int non null on error
 */
int ADM_pmNodeList(ADM_profile_id_t * nodes);

/**
 * @brief Check if a node or PID is still running
 *
 * @param[in] target which PID or node to target
 * @param[out] alive true if the node/PID is still running
 * @return int non null on error
 */
int ADM_pmAlive(ADM_profile_id_t * target, bool *alive);

/**
 * @brief Retrieve general information from a given process
 *
 * @param[in] pid target PID (no wildcards)
 * @param[out] argc size of the argument array
 * @param[out] argv passed command line arguments
 * @param[out] cpuset current process CPUset
 * @param[out] attributes list of detected attributes for this process (GPU, MPI, OpenMP, ...)
 * @param[out] mpi_rank MPI rank attached with this process
 * @param[out] job_id JOB Identifier for this process
 * @return int non null on error
 */
int ADM_pmInfo(ADM_profile_id_t pid,
               const int * argc,
               const char *** argv,
               const cpuset_t *cpuset,
               const char ** attributes,
               const uint32_t mpi_rank,
               uint64_t job_id);

/**
 * @brief Estimate memory resources associated with a given target
 *
 * @param[in] target targeted nodes or PIDs (wildcard accepted)
 * @param[out] virtual_memory total virtual memory in use
 * @param[out] estimated_phys_mem total estimated physical memory in use
 * @param[out] percentage_of_mem percentage of memory used
 * @param[out] estimated_space_left estimated space left
 * @return int non null on error
 */
int ADM_pmStat(ADM_profile_id_t target,
               uint64_t * virtual_memory,
               uint64_t * estimated_phys_mem,
               double * percentage_of_mem,
               uint64_t * estimated_space_left);
```

Name: *ADM_pmList*
input : $ADM\_profile\_id\_t$ node_id
InOut : $ADM\_profile\_id\_t*$ PIDs
output: int error_code
Description:
List all processes either being tracked globally or from a given node_id. The parameter PIDs is filled up with the corresponding list of processes. This function returns 0 on success or an error code.

Name: *ADM_pmNodeList*
InOut : $ADM\_profile\_id\_t*$ nodes
output: int error_code
Description:
The node parameter is returned as a list of the nodes monitored in the system. The function returns 0 in case of success or an error code.

Name: *ADM_pmAlive*
InOut : $ADM\_profile\_id\_t*$ target
InOut : $boolean*$ alive
output: int error_code
Description:
Set $alive$ to true if the target is alive. Where target could either a node or a PID. The function returns 0 in case of success or an error code.

Name: *ADM_pmInfo*
input : $ADM\_profile\_id\_t$ pid
InOut : $int*$ argc
InOut : $char**$ argv
InOut : $cpuset\_t*$ cpuset
InOut : $char**$ attributes
InOut : $uint32\_t*$ mpi_rank
InOut : $uint64\_t*$ job_id
output: int error_code
Description:
This functions returns a description of a given process passed as argument. All the information are sent back using parameters passed pointer. These information includes the attributes of the process, if the process belongs to a MPI or to a job. Additionally the type of hardware the process is running on, such as CPU or GPU. A success value is 0 otherwise an error code is returned.

Name: *ADM_pmStat*
input : $ADM\_profile\_id\_t$ target
InOut : $uint64\_t*$ Virtual_mem_in_use
InOut : $uint64\_t*$ Estimation_physical_mem
InOut : $double*$ memory_pressure
InOut : $uint64\_t*$ Estimated_mem_space_left
output: int error_code
Description:
This function provides statistics for a given target, where target could be node or a PID. The statistics are returned via the pointers passed in argument. The memory state is described with a differentiation between physical and virtual memory. The memory pressure corresponds to fraction (expressed a a percentage) of total (virtual and physical) memory consumed. This function may be adjusted slightly in future iterations of the API as HBM is not specifically addressed while it could be an important performance factor. The function returns 0 in case of success and an error code otherwise.

4.6 Performance Monitoring

```
/**
 * @brief Opaque definition of a sampling profile
 *
 */
struct ADM_profile_s;

/**
 * @brief Typedef of a sampling profile
 *
 */
typedef struct ADM_profile_s ADM_profile_t;

/**
 * @brief Unique identifier for a given application
 *
 */
typedef uint64_t ADM_app_id_t;

/**
 * @brief Perform dynamic profiling on one or multiple processes
 *
 * @param[in] targets list of processes or nodes to track
 * @param[in] duration time during which the profile should take place
 * @param[in] do_backtrace should the profile include backtraces
 * @param[in] region_string (optionnal) should profiling target a single region
 * @param[out] profile handle to a newly allocated profile result
 * @param[out] begin profile start date
 * @param[out] end profile end date
 * @return int non null on error
 */
int ADM_perfSample(ADM_profile_id_t targets,
                   double duration,
                   bool do_backtrace,
                  const char * region_string,
                  ADM_profile_t *profile,
                  uint64_t *begin,
                  uint64_t *end);


/**
 * @brief Perform dynamic profiling on one or multiple processes focussing on IOs (EBS)
 *
 * @param[in] targets list of processes or nodes to track
 * @param[in] duration time during which the profile should take place
 * @param[in] do_backtrace should the profile include backtraces
 * @param[in] region_string (optionnal) should profiling target a single region
 * @param[out] profile handle to a newly allocated profile result
 * @param[out] begin profile start date
 * @param[out] end profile end date
 * @return int non null on error
 */
int ADM_perfIO(ADM_profile_id_t targets,
              double duration,
              bool do_backtrace,
              const char * region_string,
              ADM_profile_t *profile,
              uint64_t *begin,
              uint64_t *end);

/**
 * @brief Attemp to classify one or multiple processes in terms of overall behavior
 *
 * @param[in] targets one or multiple processes to classify
 * @param[out] size size of the resulting vector space
 * @param[out] coordinates newly allocated array descriping processes behavior
 * @param[out] descriptions static list of labels for each coordinates
 * @return int non null on error
 */
int ADM_perfClass(ADM_profile_id_t targets,
                 uint64_t *size,
                 uint64_t ** coordinates,
                 const char ** descriptions);

/**
 * @brief Attemp to classify one or multiple processes in terms of IO behavior
 *
 * @param[in] targets one or multiple processes to classify
 * @param[out] size size of the resulting vector space
 * @param[out] coordinates newly allocated array descriping processes behavior
 * @param[out] descriptions static list of labels for each coordinates
 * @return int non null on error
 */
int ADM_perfIOClass(ADM_profile_id_t targets,
                 uint64_t *size,
                 uint64_t ** coordinates,
                 const char ** descriptions);

/**
 * @brief Opaque structure for a performance model
 *
 */
struct ADM_perf_model_s;

/**
 * @brief Type definition for a performance model
 *
 */
typedef ADM_perf_model_s ADM_perf_model_t;

/**
 * @brief Retrieve the performance model of a given application
 *
 * @param[in] app target application ID
 * @param[out] model returned model from Extra-P
 * @return int non null on error
 */
int ADM_perfModel(ADM_app_id_t app,
                 ADM_perf_model_t * model);
```

Name: *ADM_perfSample*
input : $ADM\_profile\_id\_t$ targets
input : int duration
input : $boolean$ do_backtrace
input : $String$ region_name
InOut : $ADM\_profile\_t*$ profile
InOut : $uint64\_t$ begin
InOut : $uint64\_t$ end
output: int error_code
Description:
This function is the main knob to increase both the scope and the accuracy of the performance measurements. The length of the sampling profile is set to $duration$ seconds. The amount of information and more generally the accuracy / frequency of the sampling can be configured, $do\_backtrace$ allows to retrieve a weighted call graph in the profiling information. The parameters $targets$ specifies either a node or a PID, or a set of nodes or a set of PIDs. The profiling window can be specified not only using the time information ($begin$ and end) but also using a specific code region tag defining a region name. A region name would be provided, the function will set up the $begin$ and end parameters to the value corresponding to the length of this code region. Both parameters $begin$ and end corresponds to time stamps. The profile information generated are generic profile, i.e. including CPU information and other none-I/O values. The function returns 0 upon success or an error code.

Name: *ADM_perfIO*
input : $ADM\_profile\_id\_t$ targets
input : int duration
input : $boolean$ do_backtrace
input : $String$ region_name
InOut : $ADM\_profile\_t*$ profile
InOut : $uint64\_t$ begin
InOut : $uint64\_t$ end
output: int error_code
Description:
This function is the I/O focused derivation of the function ADM_perf_sample. Only I/O information are attached to the profile. The functions returns 0 in case of success and an error code otherwise.

Name: *ADM_perfClass*
input : $ADM\_profile\_id\_t$ targets
InOut : $uint64\_t$ size
InOut : $uint64\_t**$ coordinate
InOut : $String*$ descriptions
output: int error_code
Description:
This functions attempts to produce a classification for a set of processes or nodes. The classification is returns via filled-in pointers parameters. The classification is instantiated as vector description, of $size$ elements, where the values are set in $coordinate$ and $descriptions$ holds the textual label of each field of the classification. The function returns 0 upon success or an error code.

Name: *ADM_perfIOClass*
input : $ADM\_profile\_id\_t$ targets
InOut : $uint64\_t$ size
InOut : $uint64\_t**$ coordinate
InOut : $String*$ descriptions
output: int error_code
Description:
Similar function than $ADM\_perfClass$ but limited to the I/O behavior.

Name: *ADM_perfModel*
input : $ADM\_app\_id\_t$ app
InOut : $ADM\_perf\_model\_t*$ model
output: int error_code
Description:
This function returns, via a passed-through data structure, the result of Extra-P modeling for a given application app. The full specification of model returned is not yet fully defined, thus the API relies on a opaque data structure. In case of success 0 is returned, otherwise an error code is returned.

4.7 IC control plane

New call for IC control does ON/OFF plus REGISTER/UNREGISTER

```
/**
 * @brief Actions associated with @ref ADM_monitoringEnabled
 *
 */
typedef enum
{
    ADM_MONITORING_REGISTER, /**< Register a new process (arg is application ID) */
    ADM_MONITORING_DELETE,   /**< Delete a given process (arg is ignored) */
    ADM_MONITORING_SUSPEND,  /**< Suspend all monitoring for arg seconds */
    ADM_MONITORING_LIGHT,    /**< Set light monitoring for arg seconds */
    ADM_MONITORING_MEDIUM,   /**< Set medium monitoring for arg seconds */
    ADM_MONITORING_VERBOSE   /**< Set verbose monitoring for arg seconds */
}ADM_monitoring_action_e;

/**
 * @brief Do actions for a given PID
 *
 * @param[in] pid target PID
 * @param[in] action what is to be done for target pid (see @ref ADM_monitoring_action_e)
 * @param[in] arg optionnal argument (see @ref ADM_monitoring_action_e)
 * @return int non null on error
 */
int ADM_monitoringEnabled(ADM_profile_id_t pid,
                          ADM_monitoring_action_e action,
                          uint64_t arg);
```

Name: *ADM_Monitoring_enable*
input : $ADM\_profile\_d\_t$ PID
input : $ADM\_monitoring\_action\_e$ action
input : $(uint64\_t)$ arg
output: int success (0) or error code
Description:
Set the level of monitoring, either switch it off completely or any value between light (low intrusiveness) to full blow monitoring (heavy weight)

See Deliverable D6.1 for more information on Intelligent Controller.

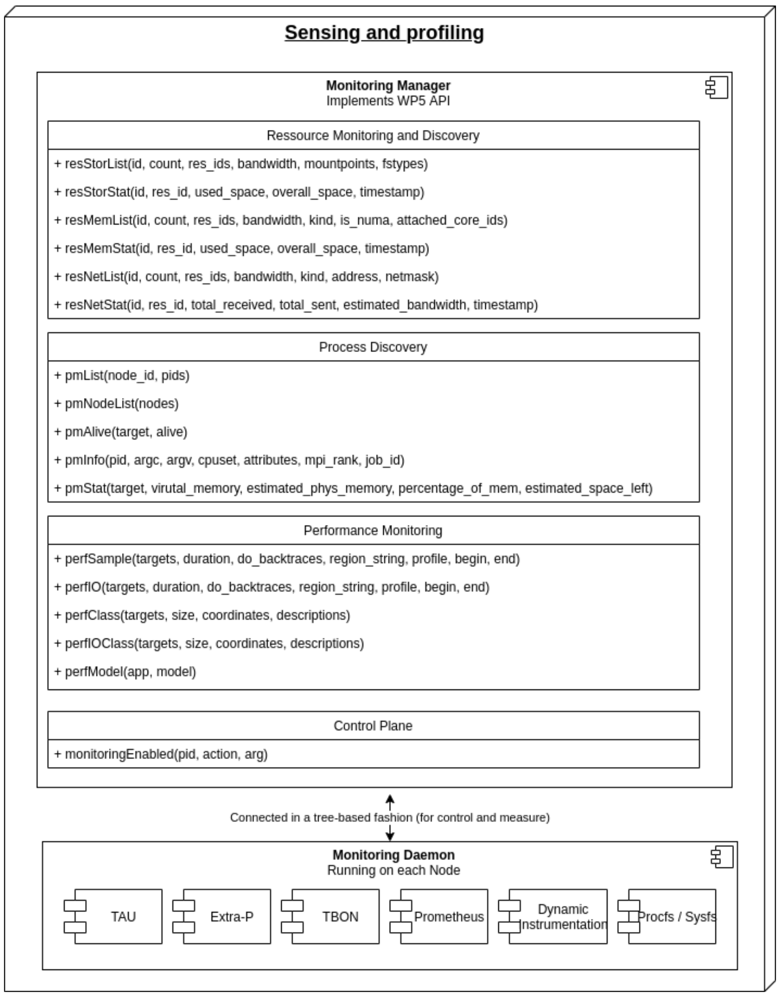
Sensing and profiling
Monitoring Manager
Implements WP5 API
Ressource Monitoring and Discovery
+ resStorList(id, count, res_ids, bandwidth, mountpoints, fstypes)
+ resStorStat(id, res_id, used_space, overall_space, timestamp)
+ resMemList(id, count, res_ids, bandwidth, kind, is_numa, attached_core_ids)
+ resMemStat(id, res_id, used_space, overall_space, timestamp)
+ resNetList(id, count, res_ids, bandwidth, kind, address, netmask)
+ resNetStat(id, res_id, total_received, total_sent, estimated_bandwidth, timestamp)
Process Discovery
+ pmList(node_id, pids)
+ pmNodeList(nodes)
+ pmAlive(target, alive)
+ pmInfo(pid, argc, argv, cpuset, attributes, mpi_rank, job_id)
+ pmStat(target, virutal_memory, estimated_phys_memory, percentage_of_mem, estimated_space_left)
Performance Monitoring
+ perfSample(targets, duration, do_backtraces, region_string, profile, begin, end)
+ perfIO(targets, duration, do_backtraces, region_string, profile, begin, end)
+ perfClass(targets, size, coordinates, descriptions)
+ perfIOClass(targets, size, coordinates, descriptions)
+ perfModel(app, model)
Control Plane
+ monitoringEnabled(pid, action, arg)
Connected in a tree-based fashion (for control and measure)
Monitoring Daemon
Running on each Node
TAU
Extra-P
TBON
Prometheus
Dynamic Instrumentation
Procfs / Sysfs

Figure 4.2: UML diagram associated with WP5 API.

4.8 UML Diagram

As presented in the UML diagram of Figure 4.2 and illustrated in Figure 4.1, from an interfacing point of view, we expose public calls solely in the *Monitoring Manager*. On each node, a *Monitoring Daemon*, connected to the *Monitoring Manager* in a scalable manner (tree-based) proceeds to measurements using previously described components while issuing commands as requested per the tree-root.

4.9 Data Velocity

In this section, we will discuss our choices with respect to the data footprint associated with the measurements previously covered in this report. Indeed, we do not solely want to imagine a system, but also want to make our approach practical and efficient, particularly considering production and constraints associated with such long-running services. For this purpose, we will first summarize our data-management plan, explaining which are the verbosities at play, and where they play a role. Second, and in the light of this introductory discussion, we will estimate the individual data-footprints to draw an overall measurement budget. Eventually, this measurement storage requirement will be contrasted with HPC-class storage capabilities.

4.9.1 Design Choices and Data-Management

What is both challenging and highly interesting in the ADMIRE project, is the transverse nature of the measurements we have to carry on. Indeed, performance tools generally excel at collecting a precise kind of metric with a reduced span, either temporally or spatially. Here, we want everything at once: a global view (e.g. time series) and the ability to *zoom-in*, being able to blame measurements up to the correct line number. In all these cases, there must be a constant space-time trade-off. More precisely, the more spatial resolution you get, the less temporal resolution you have and conversely. To better illustrate this consideration, let's quickly recall the measurements avenues we have described in Chapter 3.

We began with the Prometheus infrastructure combined with the new exporters associated with respective ad-hoc I/O layers that we are going to unfold in the project. This is the always-on measurement system, fully distributed on all the nodes it generates measurements with a relatively low frequency (≈ 5 seconds) on a wide range of local counters. This spatial definition (each node is tracked) is then necessarily constraining our ability to increase the sampling rate as data volumes would otherwise become unmanageable. We plan to (1) carefully select metrics to be tracked in each collector and (2) study the sampling rate to ensure the practicability of our back-end Monitoring System – what is important is that these levers will be available to dimension the system.

Our second measurement layer is the opposite, here we give up on spatial resolution to privilege the temporal one. To achieve this, we rely on a Tree-Based Overlay Network as presented in Section 3.3. Thanks to a reduction tree interconnecting the monitoring daemons, we plan to provide sub-second measurements for the whole system. As sketched in the dedicated section, considering one hundred counters which seem to be a reasonable system-state snapshot, this generates only a limited amount of data. In fact, and as shown in the dedicated section, reducing 100 counters every 23.44 milliseconds would only yield 33.33 KB per second of performance data. Again, in this measurement layer, we can easily change the sampling rate or limit the number of counters to head for a precise data-capping budget as shown in Figure 3.7.

The last measurement layer is associated with the application itself and applies to a larger spectrum. Indeed, as presented in Section3.2.2, our "standard" measurement scheme will rely on sampling to collect both the program counter and associated call-stacks to generate a collated performance profile of the running application. We do not anticipate this profile to be larger than a few megabytes in the most adverse case. However, as discussed in Section 3.4, we have not excluded punctual extensive measurements either with TAU (profiles, traces) or Extra-P (performance models). Given the discrete nature of such instrumentation, we do not rule out, if required, the collection of full temporal performance traces up to several Gigabytes. Considering Event-Based Sampling (EBS) as done in most state-of-the-art performance tools, the output should not be larger than a few megabytes. This approach enables direct instrumentation on the target interfaces – unlike with dynamic instrumentation. As far as our mitigation approach is concerned, it is relatively straightforward. Indeed, as we decide how and what to track we can naturally limit the resulting data footprint. For example, it might be unwise to instrument a two-week run as an application running only a few minutes. Consequently, this more verbose instrumentation will be driven by measurements at coarser grains (previous layers). Moreover, these unsupervised and asynchronous measurements allow us to (1) cancel a profile generating too much data and (2) put up with increased measurement overheads which would be intolerable in production.

As shown in Figure 4.3, the various instrumentation methodologies we have unfolded in this document are covering a wide spectrum. Indeed, thanks to the multi-scale approach we have outlined we can change data-verbosity according to both our needs and current machine capacity. We plan to rely on this re-configurable instrumentation scheme to collaborate with WP6's Intelligent Controller to constantly adapt our monitoring (and transitively its cost) to what is needed to enable ADMIRES' improved I/Os.

4.9.2 Estimated Data-Budget

A legitimate concern is the intrusiveness of the observation and the overhead induces to production operations. Here we provide an estimate of the data volume to transit on the data plane. We consider that the amount of data transiting through the control plane will remain limited. Control messages are short commands, where latency is the key issue, for such messages the most important hurdle is scalability.

It is why, as discussed in the previous section, we ensured each layer of ADMIRE's instrumentation chain is re-configurable to allow us to dynamically manage instrumentation overhead which is tightly bound to data velocity. As show in Table 4.1, each layer of instrumentation exposes possibilities of mitigating its verbosity, including dynamically in some cases. Thanks to this approach, we will be able to experimentally dimension our measurement system to find the right trade-off between descriptively and overhead. As far as the control layer

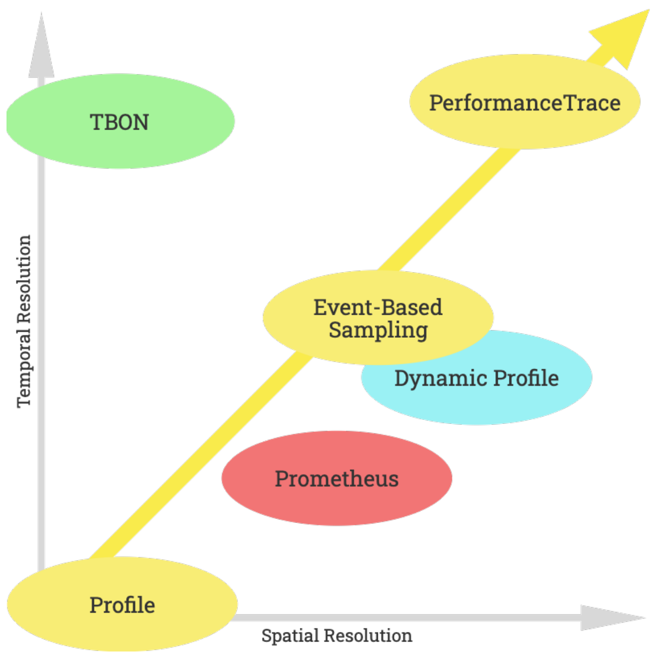

Figure 4.3: Illustration of space-time trade-offs in the ADMIRE instrumentation chain.

Instrumentation Layer	**Mitigation Techniques**
TBON	Dynamic sampling rate, number of collected entries
Prometheus	Sampling rate, retention duration, number of collected entries
Dynamic Instrumentation	Selective instrumentation (on-demand, per phase), optional back-traces
Discrete Instrumentation	On demand, fully-configurable, asynchronous and unsupervised, can be canceled

Table 4.1: Recall of anticipated data velocity mitigation strategies in ADMIRE's instrumentation chain.

is concerned, we will rely on a tree-based approach to avoid centralizing data-exchanges, yet we anticipate the control bandwidth to be negligible.

If we had to do a rough estimation of what we are intending to measure constantly, we could assume it relies on the following parameters (1) associated sampling rate, (2) the number of retrieved data-points per node and (3) the number of nodes. Therefore, considering a typical sampling rate is in the range of 5 seconds, and thousands of nodes which is a good assumption for most HPC centers. Besides, regarding the number of data points to retrieve, we estimate that 1 KB (> 1 hundred of 64-bit numbers) provides an accurate description of the performance landscape. Therefore, we can deduce the following data velocity estimate:

$$\frac{Nb\ Nodes \times 1KB}{sampling\ rate} = \frac{10000 \times 1KB}{5} = 2MB/sec$$

Considering the bisection bandwidth of a 10.000 nodes HPC system, an overhead of 2 MB/s is insignificant. Eventually, a performance database operating for three years at full speed will culminate below 200 TB. The mitigation schemes in ADMIRE will be combined with bookkeeping techniques (archiving, compression, filtering), drastically reducing the amount of hot data, yet 200 TB on cold storage is affordable in respect of current storage capacities deployed in HPC systems. As far as the instrumentation layer state is concerned, the modeling capabilities of Extra-P will not generate an additional need for data. Besides, accurate models will be generated out of these 2 MB/sec of data. Dealing with the forwarding of the IO hints yielded by WP7, we will read them directly from the Intelligent Controller's application manager inside the target application. Their storage will be associated with corresponding measurements (profiles, phase outlining, on-demand profiling), and their data velocity should then share the same upper bound.

Overall, as covered in this section, we have seen that ADMIRE transverse nature enables measurements usually out of reach solely from the application side – the most common instrumentation technique. Indeed, this exclusive ability of *zooming* into the performance data is one of the difficulties when tracking HPC applications. In our case, we will do it *automatically* without even the users noticing it, the reason why, for example, we avoided having a direct control flow with WP7. We anticipate that the upcoming discussions with other WPs, particularly WP6, will focus on deciding what is needed and when. The main goal is to enable both malleable I/O services and dynamic instrumentation in a constant feedback loop.

Chapter 5

Conclusion

A first contribution of this Deliverable is the description of the general architecture of ADMIRE's Monitoring System. The design principles of this system are based on separation of concern, with a clear distinction between the control and data planes. Furthermore, an effort is made to re-use existing open software solution if and only if they offer added value in respect of home grown solution. Since the inception of the project, an effort has been made to review candidate technology and assess the most suitable open-source projects for ADMIRE could rely on. As a project side-result of ADMIRE, we expect to push upstream patches to these selected open-source projects.

Overall the Monitoring System of ADMIRE will be composed of:

- Monitor Manager, this component will be developed in the framework of ADMIRE. the Monitoring manager is in charge of controlling all the Monitoring Daemons spread over the system. The Monitoring Manager is the interface with the Intelligent Controller. The Monitor Manager can also browse the Performance Data Lake to extract information.
- Monitoring Daemon, this component will be developed in the framework of ADMIRE. A Monitoring Daemon is active on each compute node of the system. The monitoring daemon is in charge of collection performance information, and upon request from the Monitoring Manager, to trigger more intrusive profiling. This additional profiling will be based on TAU and PStack based prototype tool.
- Telemetry solution based on the open-source project Prometheus. We will develop a node exporter compatible with Prometheus and reverse it to the Prometheus project.
- Live monitoring interface, base on the open-source project Grafana. The resulting Dashboard will help ADMIRE end-user to understand the I/O behavior and observe the resulting performance.
- Analysis, will be based on the modeling tool Extra-P, which is one of the ADMIRE assets, as well as on external open-source components if required (e.g Elk), besides pre-define database analysis written for this project.

Additionally to the components listed above, the Monitoring System includes a performance Data Lake. The Data Lake is in charge of storing the performance data. The Data Lake is expected to be mostly hosted by a Prometheus database.

As our effort has not been limited to the design of an architecture, the deliverable includes as well additional details on the performance probes mechanism. We have presented technical discussion on the implementation of the probing mechanism, where the goal is to support an open system. Initial implementation will support Lustre and GPFS parallel file systems, CPU and GPU, Infiniband network interface. But should a new accelerator technology arise within the lifespan of the project the Monitoring Mechanism design is flexible enough to accommodate it at minimal cost. The specific challenges of dynamic instruction are not overlooked, and we even foresee some interesting capabilities with the development of eBPF. About implementation we have also detailed the possible realization of a Tree-based overlay network to performance data reduction.

This deliverable defines the first iteration of the API between the Monitoring Manager and the Intelligent Controller. The sole interface of the Monitoring System with the other ADMIRE software is this API. An important contribution of this first deliverable is the specification of the main call to be supported by the Monitoring Manager to interact with the Intelligent Controller. We plan to implement the API is two different versions, first as a REST API and secondly as a low-latency RPC API. The REST API will allow us to decoupled the development of the Monitoring System from the constraints of the other WP in ADMIRE. It offers ease of integration in test and validation frameworks. The low latency RPC API will offer tighter integration with the Intelligent Controller and the level of performance align with the expectations in the market of HPC. ADMIRE will support state-of-the art notification mechanisms based on its modeling component as well as from its Analytic Module.

The next deliverable in WP5 is due by M14, at this date the 3 main implementations tasks will be active, the initial design task T5.1 which has been the main contributor of this document will be completed. We do expect by the time of D5.2 to have a prototype implementation of the Monitoring System with full connectivity among the Monitoring System components.

Bibliography

[1] Dieter an Mey, Scott Biersdorf, Christian Bischof, Kai Diethelm, Dominic Eschweiler, Michael Gerndt, Andreas Knüpfer, Daniel Lorenz, Allen Malony, Wolfgang E Nagel, et al. Score-p: A unified performance measurement system for petascale applications. In *Competence in High Performance Computing 2010*, pages 85–97. Springer, 2011.

[2] Wolfgang Barth. *Nagios: System and network monitoring*. No Starch Press, 2008.

[3] Peter Braam. The lustre storage architecture. *arXiv preprint arXiv:1903.01955*, 2019.

[4] Michael J Brim, Luiz DeRose, Barton P Miller, Ramya Olichandran, and Philip C Roth. Mrnet: A scalable infrastructure for the development of parallel tools and applications. *Cray User Group*, 2010.

[5] Cyril Cassagnes, Lucian Trestioreanu, Clement Joly, and Radu State. The rise of ebpf for non-intrusive performance monitoring. In *NOMS 2020-2020 IEEE/IFIP Network Operations and Management Symposium*, pages 1–7. IEEE, 2020.

[6] collectd developers. collectd – The system statistics collection daemon , 2021. https://collectd.org/.

[7] Prometheus developers. Prometheus Webpage, 2021. https://prometheus.io/.

[8] Telegraf developers. Telegraf webpage, 2021. https://www.influxdata.com/time-series-platform/telegraf/.

[9] Alexandre E Eichenberger, John Mellor-Crummey, Martin Schulz, Michael Wong, Nawal Copty, Robert Dietrich, Xu Liu, Eugene Loh, and Daniel Lorenz. Ompt: An openmp tools application programming interface for performance analysis. In *International Workshop on OpenMP*, pages 171–185. Springer, 2013.

[10] Bengisu Elis, Dai Yang, Olga Pearce, Kathryn Mohror, and Martin Schulz. Qmpi: A next generation mpi profiling interface for modern hpc platforms. *Parallel Computing*, 96:102635, 2020.

[11] Dominic Eschweiler, Michael Wagner, Markus Geimer, Andreas Knüpfer, Wolfgang E Nagel, and Felix Wolf. Open trace format 2: The next generation of scalable trace formats and support libraries. In *Applications, Tools and Techniques on the Road to Exascale Computing*, pages 481–490. IOS Press, 2012.

[12] Todd Gamblin, Martin Schulz, Bronis R de Supinski, Felix Wolf, Brian JN Wylie, et al. Reconciling sampling and direct instrumentation for unintrusive call-path profiling of mpi programs. In *2011 IEEE International Parallel & Distributed Processing Symposium*, pages 640–651. IEEE, 2011.

[13] Marc-André Hermanns, Nathan T Hjlem, Michael Knobloch, Kathryn Mohror, and Martin Schulz. Enabling callback-driven runtime introspection via mpi_t. In *Proceedings of the 25th European MPI Users' Group Meeting*, pages 1–10, 2018.

[14] Kevin A Huck and Allen D Malony. Perfexplorer: A performance data mining framework for large-scale parallel computing. In *SC'05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, pages 41–41. IEEE, 2005.

[15] Tanzima Islam, Kathryn Mohror, and Martin Schulz. Exploring the capabilities of the new mpi_t interface. In *Proceedings of the 21st European MPI Users' Group Meeting*, pages 91–96, 2014.

[16] Tanzima Islam, Kathryn Mohror, and Martin Schulz. Exploring the mpi tool information interface: features and capabilities. *The International Journal of High Performance Computing Applications*, 30(2):212–222, 2016.

[17] Thomas Leibovici. Taking back control of hpc file systems with robinhood policy engine. *arXiv preprint arXiv:1505.01448*, 2015.

[18] Glenn K Lockwood, Alberto Chiusole, Lisa Gerhardt, Kirill Lozinskiy, David Paul, and Nicholas J Wright. Architecture and performance of perlmutter's 35 pb clusterstor e1000 all-flash file system. 2021.

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

[20] Bernd Mohr, Allen D Malony, Sameer Shende, and Felix Wolf. Design and prototype of a performance tool interface for openmp. *The Journal of Supercomputing*, 23(1):105–128, 2002.

[21] Aroon Nataraj, Allen D Malony, Alan Morris, Dorian Arnold, and Barton Miller. A framework for scalable, parallel performance monitoring using tau and mrnet. In *International Workshop on Scalable Tools for High-End Computing (STHEC 2008)*. Citeseer, 2008.

[22] Srinivasan Ramesh, Aurèle Mahéo, Sameer Shende, Allen D Malony, Hari Subramoni, Amit Ruhela, and Dhabaleswar K DK Panda. Mpi performance engineering with the mpi tool interface: the integration of mvapich and tau. *Parallel Computing*, 77:19–37, 2018.

[23] Prapaporn Rattanatamrong, Yoottana Boonpalit, Siwakorn Suwanjinda, Ayuth Mangmeesap, Ken Subraties, Vahid Daneshmand, Shava Smallen, and Jason Haga. Overhead study of telegraf as a real-time monitoring agent. In *2020 17th International Joint Conference on Computer Science and Software Engineering (JCSSE)*, pages 42–46. IEEE, 2020.

[24] Marcus Ritter, Alexander Geiß, Johannes Wehrstein, Alexandru Calotoiu, Thorsten Reimann, Torsten Hoefler, and Felix Wolf. Noise-resilient empirical performance modeling with deep neural networks. In *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 23–34. IEEE, 2021.

[25] Martin Schulz and Bronis R de Supinski. Pnmpi. Technical report, Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States), 2008.

[26] Sameer Shende, Allen D Malony, Wyatt Spear, and Karen Schuchardt. Characterizing i/o performance using the tau performance system. In *Applications, Tools and Techniques on the Road to Exascale Computing*, pages 647–655. IOS Press, 2012.

[27] Nitin Sukhija and Elizabeth Bautista. Towards a framework for monitoring and analyzing high performance computing environments using kubernetes and prometheus. In *2019 IEEE SmartWorld, Ubiquitous Intelligence Computing, Advanced Trusted Computing, Scalable Computing Communications, Cloud Big Data Computing, Internet of People and Smart City Innovation (SmartWorld/SCALCOM/UIC/ATC/CBDCom/IOP/SCI)*, pages 257–262, 2019.

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

[29] Jeffrey S Vetter. *Contemporary high performance computing: from Petascale toward exascale*. CRC Press, 2013.

[30] Michael Wagner and Andreas Knüepfer. Automatic adaption of the sampling frequency for detailed performance analysis. In *2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, pages 973–981. IEEE, 2017.

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

Appendix A

Terminology

- Ad hoc Storage System, ephemeral storage system that only exists in a determined period, i.e. during a job's execution.
- CLI, command line interface.
- DRAM, dynamic random-access memory.
- EBNF, Extended Backus–Naur Form is a family of metasyntax notations, any of which can be used to express a context-free grammar. EBNF is used to make a formal description of a formal language such as a computer programming language. They are extensions of the basic Backus–Naur form (BNF) metasyntax notation.
- In situ data, processing the data where it is originated.
- In transit data, processing the data when it is moved.
- NORNS, data transfer service for HPC developed at BSC.
- NVM, non-volatile memory.
- PFS, parallel file system.
- POSIX, Portable Operating System Interface, family of standardized functions.
- QoS, Quality of Service.
- RDMA, remote direct memory access.
- RPC, remote procedure call.
- Slurm, job submission system widely used.
- SSD, solid state drive.
- Object store, persistent storage system where data are stored not as file but as objects. In its canonical implementation Object are immutable and the API is limited to PUT, GET and DELETE. More 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`.