EuroHPC-01-2019

IO-SEA

IO – Software for Exascale Architectures

Grant Agreement Number: 955811

D2.1

Ephemeral Data Access Environment
Concepts and Architecture

Final

Version:	1.0
Author(s):	A. Lopez (Atos), S. Valat (Atos), S. Narasimhamurthy (Seagate), M. Golasowski (IT4I)
Contributor(s):	S. Krempel (ParTec), M. Rauh (ParTec), P. Deniel (CEA), D. Vasiliauskas (Seagate)
Date:	26.01.2022

Project and Deliverable Information Sheet

IO-SEA Project	**Project ref. No.:**	955811
	Project Title:	IO – Software for Exascale Architectures
	Project Web Site:	https://www.iosea-project.eu/
	Deliverable ID:	D2.1
	Deliverable Nature:	Report
	Deliverable Level: PU *	**Contractual Date of Delivery:** 31 / January / 2022
		Actual Date of Delivery: 26 / January / 2022
	EC Project Officer:	Daniel Opalka

*– The dissemination levels are indicated as follows: **PU** - Public, **PP** - Restricted to other participants (including the Commissions 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).

Document Control Sheet

Document	**Title:** Ephemeral Data Access Environment	
	ID: D2.1	
	Version: 1.0	**Status:** Final
	Available at: https://www.iosea-project.eu/	
	Software Tool: LaTeX	
	File(s): IO-SEA_D2.1-report.pdf	
Authorship	**Written by:**	A. Lopez (Atos), S. Valat (Atos), S. Narasimhamurthy (Seagate), M. Golasowski (IT4I)
	Contributors:	S. Krempel (ParTec), M. Rauh (ParTec), P. Deniel (CEA), D. Vasiliauskas (Seagate)
	Reviewed by:	1. Max Holicki (FZJ) 2. Jacques-Charles Lafourcrière (CEA)
	Approved by:	Exec Board/WP7 Core Group

Document Status Sheet

Version	Date	Status	Comments
0.1	1/12/2021	Outline published	Done
0.5	15/12/2021	Ready for WP-internal review	Done
0.6	7/1/2022	Ready for internal review - 1st round	Done
0.7	14/1/2022	Ready for internal review - 2nd round	Done
0.8	20/1/2022	Ready for publishing	Done
1.0	26/1/2022	Published	Done

Section	Status
Executive summary	Proofread
Introduction	Proofread
Architecture	Proofread
Interfaces	Proofread
Software	Proofread
Summary	Proofread

Document Keywords

Keywords:	IO-SEA, HPC, Exascale, Software, Ephemeral, Services, Architecture

Contents

List of Figures

List of Listings

List of Tables

Executive Summary

This document presents the concepts and architecture of all the software defined in the work package 2 to implement the *Ephemeral Data Access Environment*. The objective of this work package is to provide an on-demand data access environment suitable for the needs of applications and workflows within the IO-SEA project.

The first chapter is an introduction to the work done in the work package. Chapter 2 describes the technical solution for the Ephemeral Data Access Environment and how it interacts with the other technical work packages. Chapter 3 describe the interfaces used and provided by the work package. An example using the LQCD use-case is provided in Chapter 4. Chapter 5 focuses on the software the participants will bring along to be included in the solution, how they will be modified to achieve that inclusion and the progress done so far. And finally the summary is provided in Chapter 6.

This report describes our evolved understanding of the work to be done, based on the discussion process and data sharing during the first ten months of the IO-SEA work package 2. This work is ongoing and will be refined over the project, which will be reflected in future devliverables.

1 Introduction

This deliverable exposes the design and the key elements for the ephemeral service feature in the IO-SEA. This feature is a keystone of the solution developed inside this project with strong dependencies on work to be done in other work packages (mostly work packages 4 and 5).

Today, as a compute job runs, it usually performs its I/O operations on a persistent I/O service and most of the time a parallel file system like Lustre or GPFS is used. In this scope, the I/O service is a “perpetual resource”, that is not dedicated to a specific compute job. This approach, widely used in petascale-capable compute center reaches its limits with the coming exascale era, in particular major scalability issues are expected.

The ephemeral service brings a new approach to fit the forthcoming exascale challenges. It basically relies on a very simple idea: as the compute job starts, an associated I/O server, dedicated to this compute job, will start too. This way, the compute job uses a dedicated service, it does not directly access a global storage service that can be overwhelmed by the heavy workload of other jobs.

During the run, I/O operations will go through this ephemeral service which acts as a proxy to a massive object-store that perpetually keeps the data, making the data flow independent from the data flow of other compute jobs. Ephemeral services bring an explicit proxy level that provides ways to solve many scalability issues.

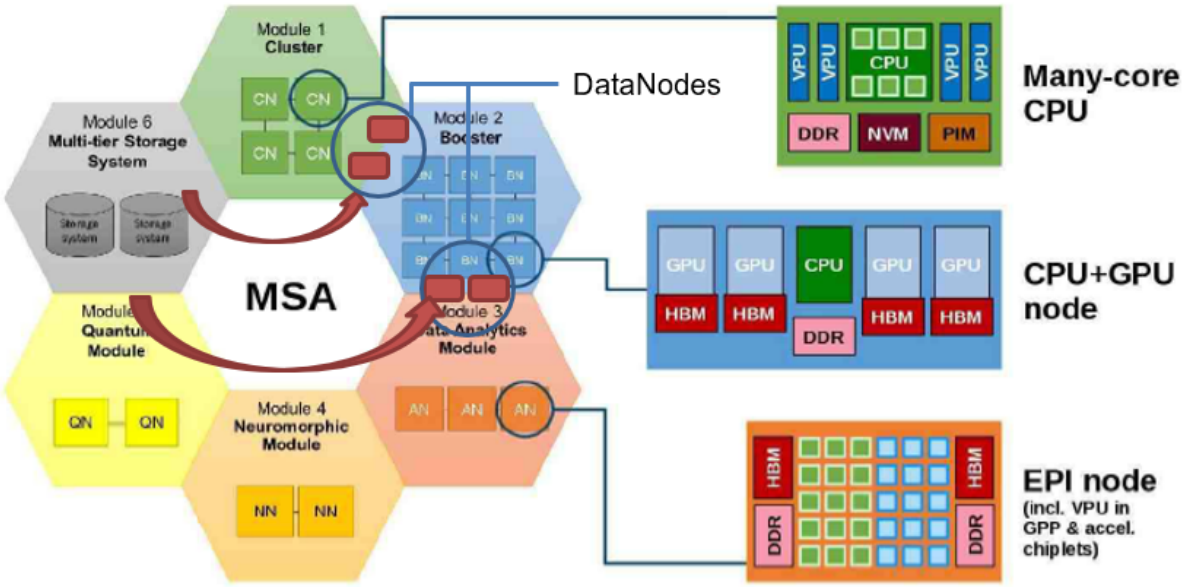


Figure 1: IO-SEA’s Modular Supercomputer Architecture.

The ephemeral service relies on the components of the Modular Supercomputer Architecture (MSA): compute jobs will run on compute nodes, while ephemeral I/O service will run on data nodes. The ephemeral service starts right before its clients (they are the applications running on compute nodes involved in the compute job) and it ends right after the compute job ends, after having flushed all data on persistent storage. Its lifecycle is related to the compute job’s lifecycle.

The ephemeral service is a central concept. On one side, it is system-oriented and is a keystone of the IO-SEA solution, but on the other side, users will be in "contact" with it as they use it to access their information.

Work package 2 will develop an *Ephemeral Data Access Environment*, running on data nodes in the modular supercomputing environment (Fig. 1), taking advantage of modern hardware such as NVMe and NVRAM and network-closeness.

The process leading to this report involved presentations to and from the other work packages. Several global cross-work-package workshops and direct meetings with other work packages allowed us to share information and reach consensus on several concepts.

This is the first major step for Work Package 2, although this is not the final word on the architecture. As our collaboration with the other work packages continues, some minor adaptations are expected to better manage any detail that we might have missed and thus provide the best solution for the IO-SEA project.

2 Architecture

2.1 Concepts

In the context of ephemeral services, two concepts are of key importance: the notion of a dataset and the notion of a namespace. As these concepts are relevant to the entire storage architecture of IO-SEA, they have been defined jointly, involving participants from all work packages (in so called "cross-work-package sessions").

The IO-SEA storage architecture plans to store data inside a massive object-store. Object-stores are known for being very scalable, they will fit the requirements in terms of performance and scalability for exascale storage, but this enhanced scalability has a cost: an object-store is a "*bag full of data*", it does not store files in a hierarchical name-space (stored in directories and subdirectories, using a classical POSIX tree-like architecture we are all familiar with), it stores objects in a flat name-space. Two objects have no connection, they are fully independent from each other, and each of them is managed independently. An object-store is very well named: it's a low level service that only stores data, but it does not organise the data.

Nevertheless, data organisation is very important and meaningful for end users: user data will be made of bunches of records (for example many files coming from experiments or scientific studies involving older simulation runs) where pieces of data explicitly depend on each other. Datasets and namespaces exist for providing such an organisation that does not exist inside the object-store.

Both datasets and namespaces basically group data and both concepts are related. A dataset group pieces of data that have reasons to stay together, that will be used together: for example, weather data on a given time and location, a collection of microscopy pictures of a given organ, data representing a set of atomic particles involved in a quantum physics experiment, . . . Whatever their connection, they have to stay together and will be used as a whole. Keeping those pieces of data grouped is natural and this is required for keeping good performances and making optimisation possible. For example, if you manage thousands of files or objects, it's clearly wise to store them on the same tapes, otherwise accessing them will result in tens or hundreds of tapes being mounted by the tape robot, an operation that will waste a lot of time.

A namespace describes the way the data is seen by the end user, how the end user gives names to the pieces and data and how he organises them. For example, a NFS server exposes namespaces for the clients that mount it. The user will then see a classical hierarchical tree-based organisation, with directories containing files and subdirectories.

Datasets and namespaces work together, they are two sides of the same coin: a namespace is always built on top of a dataset, and the namespace will expose to the users an organised view of the data inside the dataset. From a stratospheric point of view and making an analogy with the filesystem logic, a namespace is mostly related to metadata (similar to the filesystem's metadata) and a dataset is closely connected to the content of the files.

These two concepts are very important in IO-SEA and they are connected to different IO-SEA work packages:

- Namespaces are mainly addressed by Work Package 5.

- Datasets are very important in Work Package 4 which is responsible for moving the data between storage tiers via Hierarchical Storage Management (HSM) feature.
- Metrics retrieved by Work Package 3 should take those concepts into account while designing its monitoring solution.

Coming back to ephemeral services, this feature deals with both concepts:

1. Before it starts, the ephemeral service acquires the dataset and it will possibly update it during the run.
2. The ephemeral service exposes the namespace built on top of this dataset to the end user during the run.
3. As it ends, the ephemeral service flushes all data to persistent storage, it closes the namespace, and it releases the dataset that is now available for another job associated with another instance of an ephemeral service.

For example, let's imagine that you are managing with your smartphone the pictures taken during your holidays. The list, or catalog of all your pictures, is a dataset, it tells you where the pictures are inside your phone's memory, but it shows no organisation. When you connect your smartphone to the computer a window appears that helps you browsing the pictures. They may be sorted in different folders/directories by data, or type (video and fixed pictures), or location (via a GNSS tag), . . . The system has started an ephemeral service to pop-up the window, the tree that you browse is a hierarchical namespace.

The dataset and namespace concepts are important to the ephemeral services, and the ephemeral service is the only feature to deal with both concepts. As said above, they are two sides of the same coin, and WP2 has to take care of these two faces of data management.

2.2 Overview

Applications will access data through some specific I/O services. These I/O services will be instantiated for a given *workflow* and destroyed when no longer needed. This is why they are referenced as *ephemeral*. The ephemeral I/O services have two main benefits:

1. Allow applications to access data stored in a format they do not know how to handle,
2. Accelerate the access to that data.

In the IO-SEA context, an application can be written to access data as files on a POSIX filesystem, or to access data stored as objects in an object-store such as Motr [1]. The first goal of ephemeral services is to allow an application to access data stored in a format it was not written for, on-the-fly converting the data between POSIX files and objects. This conversion requires that a mapping is established between the structure present in the storage and the structure presented to the application. In the case of an object-store, the structure is given in buckets and object IDs (OIDs); while for a POSIX filesystem, it will be given in terms of directories and file names. The structure used for the presentation is called a namespace.

Ephemeral I/O services run on the module's data nodes. A data node is a cluster node dedicated to provide I/O services and located, network-wise, close to the compute nodes (see Fig. 1). They are equipped with multiple NVMe [2] and NVRAM [3] devices used to locally store the workflow's data. Prior to launching the application, data is brought to the data nodes from the long-term storage module using the Hierarchical Storage Management (HSM) interface developed by Work Package 4. Using the fast local storage and helped by their closeness to compute nodes, data nodes achieve the second goal. Later, when the application has finished, new and modified data will be sent back to the long-term storage.

The ephemeral services life-cycles are directly associated to the steps' lifecycle and indirectly associated to the workflow's lifecycle. A workflow is a sequence of steps; each step being an application running on a computing module. An ephemeral service running on the module's data nodes is used by one or more (possibly consecutive) steps. Before starting a step, its ephemeral service is instantiated and the required data loaded in it from the long-term storage. After the step is finished, the next ephemeral service is instantiated, the data from the previous one is transferred to the new one and then the old service is terminated. Two consecutive steps can use the same ephemeral service, avoiding the need for data transfer. When the last step finishes, the data is transferred to the long-term storage. It is also possible to transfer data to the long-term storage during the intermediate steps.

Users will be able to express their requirements in terms of datasets, namespaces, data accessors and ephemeral services when launching their workflows. They can select the input, intermediate and output datasets, which ephemeral services will be launched and their size. These requirements will be completed with recommendations from the monitoring tools implemented by Work Package 3.

The work in Work Package 2 is divided into three tasks:

Task 2.1 Define the data accessors and I/O services we will focus on, adapt them to be ephemeral, define their lifecycle and provide a common interface allowing ephemeral instantiations. This is presented in Section 2.3.

Task 2.2 Adapt those same services to use NVMe and NVRAM[1] as local storage. The work in this task is presented in Section 2.4.

Task 2.3 Receive the user requirements, allocate the resources (data nodes, NVMe storage, NVRAM storage, etc.) and orchestrate the deployment of the required IO services. This work is presented in Section 2.5.

2.3 Task 2.1 - Ephemeral Data Access Environment

This section will discuss the global architecture we are considering to manage the ephemeral services. We split the discussion into three sub-sections. The first one focus on the ephemeral services that will be used. The second discusses the operations required to start, configure and use the ephemeral services. Finally, the third discusses, from an architectural aspect, how to deploy and manage the services based on user requests.

[1] Some services are already able to use NVMe storage, but none of them able to use NVRAM

2.3.1 Considered Ephemeral Services

The ephemeral I/O services considered are taken from this non-exhaustive list:

- *Atos Flash Accelerators* [4] is a suit of ephemeral services for accelerating user's jobs by using the NVMe storage available on the data nodes. It has three working modes:
  - *Smart Burst Buffer* (SBB) is an intelligent burst buffer providing a POSIX interface on top of an existing POSIX parallel filesystem or an S3-compatible object-store. It provides two levels of local cache: RAM and flash storage. A third level will be added to handle NVRAM.
  - With *Smart Bunch of Flash* (SBF) each compute node running a step of a workflow has a dedicated NVMe storage present on the data nodes, which is exposed to the compute nodes through NVMe-oF. It can be used as a fast temporary POSIX storage.
  - *Global Bunch of Flash* (GBF) deploys an ephemeral parallel POSIX filesystem on the data nodes using the NVMe devices. Compute nodes participating in the workflow's step will mount this filesystem.
- *Seagate's CORTX* [1] provides an S3 interface with access management used to access the Motr object-store.
- The *CEA's NFS Exporter* is used to expose Motr objects into a POSIX namespace.

2.3.2 Ephemeral Services Lifecycle

The instantiation of the ephemeral services requires several operations to be performed in order to get the environment ready for use.

The first operation consists of allocating the necessary resources. It requires finding the adequate data nodes with enough NVRAM and NVMe space, memory and CPU available to run the requested ephemeral services. The detailed actions of this operation are discussed in the Section 2.5 focused on the Yorc orchestrator [5] and possible alternative solutions.

Once the data nodes have been identified, the system needs to start the ephemeral services on the allocated data nodes and eventually make the link between the various services if they need to interact. The first stage consists of starting the ephemeral services used to instantiate the namespaces requested by the user. It relates to the CORTX NFS and S3 server to make the namespaces accessible via their protocol. Recall here that those services (NFS, S3) directly translate the incoming requests into Motr operations, as such, they do not offer any caching to accelerate the I/O.

In a second operation, we look at the possible interactions between the ephemeral services to boost the I/O performance on the data nodes as presented in the Fig. 2. Here, we consider adding caching with a local Motr to prefetch (via the HSM mechanism) the user data into the local NVMe or NVRAM. This use case is a novel approach to Motr which is described in the Section 2.4. A simpler scenario consists in adding a SBB instance on top of the namespace services (NFS or S3) to use the local RAM and flash memory as caches in order to provide a faster access to the data exposed by those

namespaces. Similarly, we can also copy a directory from the NFS namespaces to a local SBF or GBF service to make a local cache and flush out the data when the application ends.

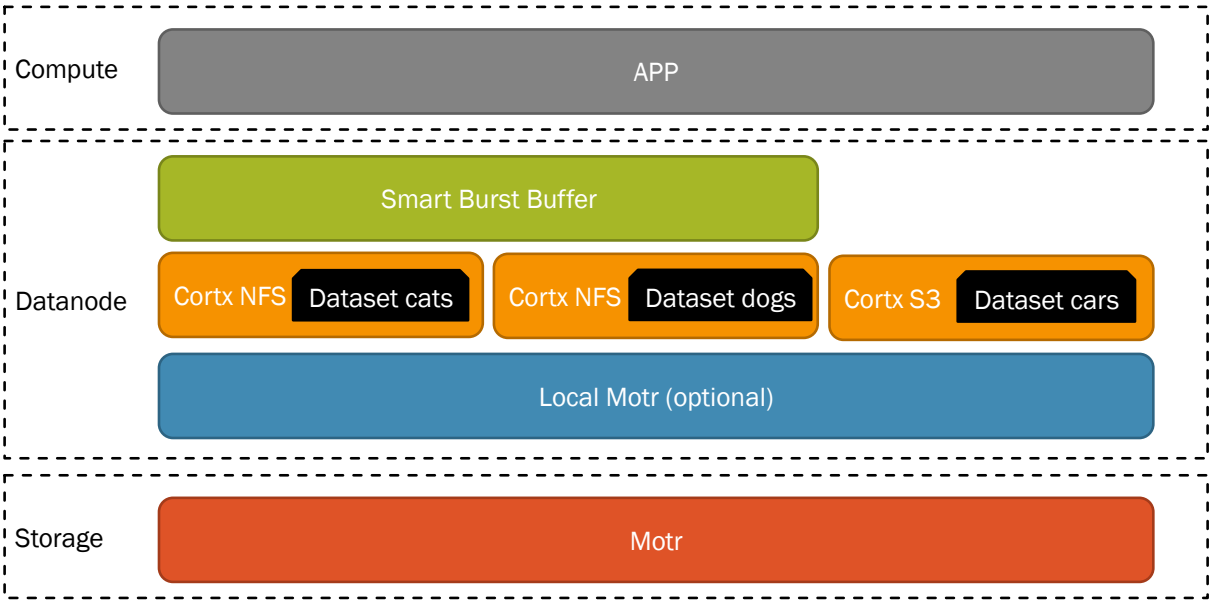


Figure 2: Interactions of services inside a data node to be stacked adding accelerator and local caches to the data access path.

Finally, the setup operation will then configure the compute nodes so that the user can use the requested ephemeral services. The global picture of all this chain is presented in Fig. 3 and shows the point of impact of each component and global interaction on the system. On the compute nodes the configuration will be done by setting environment variables and/or mounting the remote filesystems into the local filesystem. We can hence summarize the service operations as listed below:

- For SBB, the client-side configuration only consists of setting some environment variables. We first need to bind the client-side library to the client application via the special **LD_PRELOAD** environment variable. This library is called *iolib* and intercepts the POSIX I/O functions. The library can load modules to expand its behavior. The **IOLIB_MODULES** environment variable instructs the library which modules to load. In this case, we will load the *libsbb* module. Some extra environment variables are used to configure the SBB module:

  SBB_PROXY_RDMA to tell how to reach the SBB server.

  SBB_PROXIED_TARGETS which paths (targets) to capture and send through the accelerator.

  SBB_PARAMS to report its basic configuration to the *IO Instrumentation* tool (involved in Work Package 3).

- The configuration of the NFS client requires operations as a privileged user (usually root) to mount the remote filesystem.
- The GBF also requires mounting the filesystem on the client node in a manner very similar to the NFS approach.

- The SBF requires importing the remote NVMe device over the InfiniBand fabric and mounting the filesystem.
- The S3 server requires finding a convention in order to transmit the address and port exposed by the server. We use a simple URL that is to be stored in an environment variable, to be used by the user's application. We might consider several S3 servers (one per namespace) which results in several addresses to be transmitted to the user. In this case, we might provide a file in the environment variable in place of the address itself, and list the addresses in this file. We should, in this case, provide a key/value definition with a name for each address so the user can easily distinguish them.

As we consider using Slurm as the job manager in the IO-SEA cluster, these operations should be triggered by a Slurm plugin. For the implementation we may choose to extend the Slurm burst buffer plugin as already done at Atos for the Flash Accelerator products (SBB, SBF, GBF) and in the SAGE 2 European project regarding CORTX/Motr.

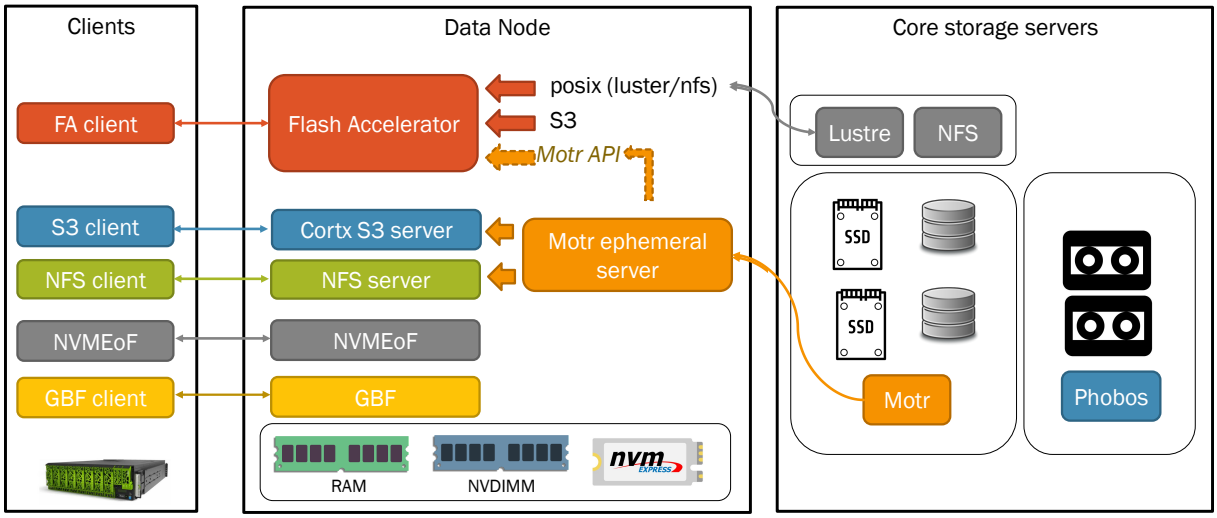


Figure 3: Overall interactions and places used to setup the ephemeral services on clients and data nodes.

2.3.3 Control Architecture

In order to control the life cycle of the ephemeral services we need an orchestrator which will be discussed in Section 2.5 to manage the data node resources and decide where to launch the ephemeral services. At this stage, we do not consider the client providing its requirements in the native format handled by this orchestrator. As the representation is too low level for most users, we do not consider exposing it. We prefer defining a YAML [6] file tuned for the IO-SEA semantics, which will be translated into the orchestrator language. The YAML file semantics will be discussed in Section 3.1 and should be simple to use and understand for the final users.

In order to perform this translation, we will add a REST API between the Slurm plugin and the orchestrator. This REST API will have the responsibility to:

- Translate user requests (the YAML file) into the orchestrator language.
- Track the state of the workflow progression to request the starting / ending of the ephemeral services.
- Report the ephemeral services status.
- Provide the necessary information to configure the client node for the given ephemeral services.

This architecture is summarized in Fig. 4. The usage of an intermediate REST API is also a way to make a clear role distinction and take most of the work done out of Slurm so we can consider at some point an integration into other job managers for the cluster. We can also consider changing the orchestrator in use by adapting the REST API implementation.

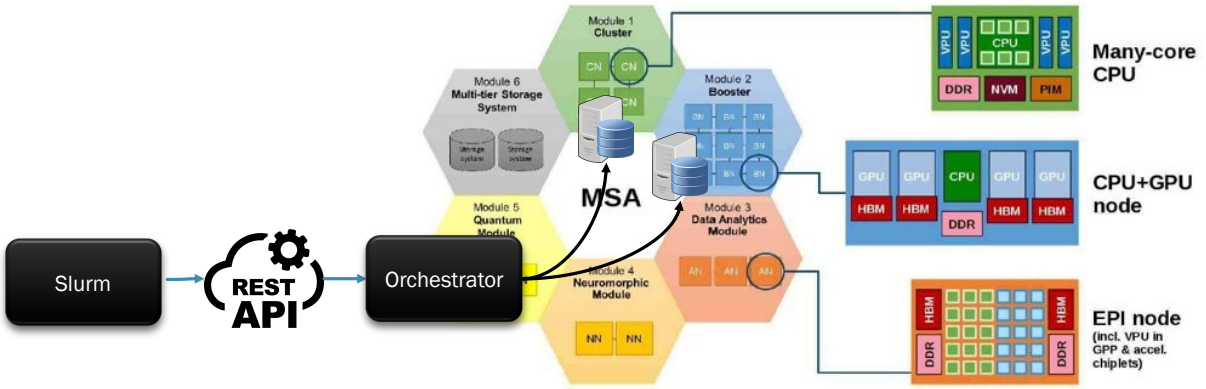


Figure 4: Global architecture showing the place of the REST API in the control path from Slurm to the services on the data nodes.

When making an implementation of this REST API we will need to take care of the authentication method so we can guaranty that only Slurm can call it. This can be done with a JSON Web Token (JWT) base authentication, as used by most web services.

2.4 Task 2.2 - NVMe and Non-Volatile Memory usage on data nodes

This section presents work on allowing existing ephemeral services to use data node-local NVMe and NVRAM devices. We begin by reviewing flash memory technology and then discuss how it will be used by the ephemeral services.

The data nodes are assumed to have very fast, low-latency persistent storage resources, namely NVRAM and storage based on the NVMe access protocol. These resources temporarily manage objects during the ephemeral workflow execution. Objects are then down-migrated to lower-tier storage. The NVMe protocol as well as NVRAM technology offers improvements in latency and throughput. There has been research on various types of NVRAM technologies such as STT-MRAM [7], Memristor [8], PCM [9], etc. 3DXPoint technology [10] from Micron/Intel is the technology currently available in the market place, which is a kind of PCM. The following picture provides an idea of the performance improvements that can be availed by NVRAM technology and the NVMe protocol.

The NVRAM technology can both extend the available memory address space (and is available in DIMM form factor, called NVDIMMs) and can also be used as a block storage device (available as a PCI-E device). It is also possible to expose the NVDIMMs as a block device using software [11].

We next describe the overall architecture of the data nodes exposing these high-speed low-latency storage resources, which extends some of the concepts available with CORTX Motr.

2.4.1 Overall big picture of NVRAM/NVMe resources in data nodes

The overall architecture of NVRAM/NVMe exposed within data nodes is described in Fig. 5 below. The data nodes run ephemeral services as described earlier. Some ephemeral services interact with the Motr API instances on the data nodes, while others run by themselves.

Motr Services

Motr services corresponding to the Motr API instances are also assumed to ephemerally run within the data nodes. Motr hence can act as both a client and a server within these ephemeral nodes. The Motr services expose the data on the NVRAM/NVMe as objects. The Motr services on any data node can provide a unified object namespace across objects in other data nodes as well. NVRAM resources could either be in DIMM or PCI-E block device form factor. Such a unified addressing needs to be developed in the IO-SEA project.

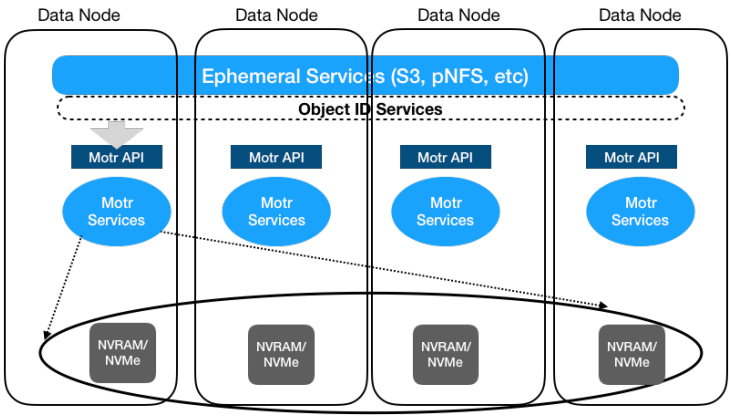


Figure 5: Overall Architecture: NVRAM/NVMe on data nodes exposed with Motr.

The Motr services are planned to be initiated and torn down similar to the other ephemeral services. More details on Motr are provided in Section 5.2.

HSM services are used to migrate the objects within NVMe/NVRAM in the data nodes to longer term persistent storage tiers at the end of the workflow. These are described separately.

We also show an object ID service that will be availed by the various ephemeral services to manage their name spaces. The object ID services make sure that OIDs allocated by Motr are uniquely provided to each of the services.

Object ID's and Object ID Service Motr has 128-bit object IDs. This means that there are 2^{128} OIDs to choose from. This is a vast ID space. On the other hand, during object creation, applications and ephemeral services need to be supplied with free OIDs, that is, OIDs that are not already taken by other applications nor ephemeral services. Guaranteeing an OID is free is impossible unless there is a centralised object ID manager service because a Motr cluster is a distributed environment and applications and services running on Motr are also completely distributed. Simple random ID generation methods are easily prone to collisions. Cryptographic hash-based methods are also not guaranteed to be collision free. Besides, random and hash-based methods definitely will not use the entire key space. Therefore, an ID service that can guarantee free object IDs, use entire ID space and also divide the ID space into mutually exclusive sub-spaces so that different kind of services can use keys from different sub-spaces is required.

Stand-Alone Services

Other ephemeral services such as Flash Accelerators do not require Motr to be running on the data node. These services will be modified to also take advantage of the NVMe and NVRAM devices available on the data nodes.

Atos will extend Flash Accelerators' Smart Burst Buffer (SBB) to integrate a remote mapping feature to a very large file, allowing to map a set of data from many compute nodes that could not fit into local memory.

mmap() is a system function allowing to memory-map a file on storage as if it was present in the system memory, and access it by reading and writing into that memory. Using *mmap()*-like semantics, it will be possible for applications running on the compute nodes to memory-map into NVRAM a file through the Smart Burst Buffer ephemeral service, taking advantage of the persistence provided by the NVRAM technology. This will be done leveraging technology developed in the Sage2 project [12].

More information on Flash Accelerators can be found in Section 5.1.

2.5 Task 2.3 - Data Operation Scheduling

The data operation scheduling can be divided in two consecutive tasks that have to be handled. The former being life-cycle handling of the ephemeral services and the latter task is optional staging of the input data to the resources exposed through the ephemeral services. In this section we describe a draft of the implementation of the first task.

As mentioned already in Section 2.3.3 the life-cycle of the ephemeral services will be handled by Yorc orchestrator through a Slurm plugin. It is a complex piece of software which can handle extensive number of use-cases. In this case, the orchestrator will be used to deploy a number of predefined topologies described in TOSCA/Terraform standard, where each topology corresponds to a particular type of ephemeral service.

Yorc is a software that provides abstraction over deployment topologies capable of handling various compute resources, including commercial cloud providers, bare-metal machines or HPC clusters.

Workflow execution is supported as well, which allows custom definition of application life-cycles. Ephemeral services in IO-SEA stack can be represented by applications in Yorc, which in turn are deployed in environments [5, 13].

The responsibility of the REST API in front of the Yorc will be to parse the YAML file and Data Access and Storage Interface (DASI) description supplied by the Slurm plugin to a set of calls to the Yorc and HSM API which will result in deployment of the requested ephemeral services and staging of the requested data on the allocated resources.

Resources for the services can be either bare-metal (data nodes) or virtualised (Cloud API, OpenStack). In the next phase of development we will explore both options and determine their viability. In both cases Yorc will handle their allocation and instantiation as part of the deployment process. This also means that Yorc will be responsible for tracking and selecting the available resources. Templates for the individual services are part of the REST API configuration.

The REST API of the scheduling service will be used by the SPANK plugin which will forward the YAML description and DASI description of the requested datasets. The lifecycle of the services will be handled within the context of a named session. The session is created upon start of the workflow execution and terminated at its end. Part of the YAML description of the requested services is the concept of steps, which are triggered as the workflow advances through its tasks. This concept is used to describe the dependencies between the workflow tasks and ephemeral services as one service can be reused by multiple tasks. The Yorc will be used together with other SPANK plugin such as burst buffer to handle provisioning of the storage resources for the ephemeral services.

Monitoring data from the REST API itself, the deployed services or the SPANK plugin can be provided to external services such as the ParaStation Healthchecker to provide health and performance auditing. The HealthChecker will especially need and use information about which ephemeral service shall run where and at what time to check if the services start and stop correctly and are available if they are expected to be by the jobs. This monitoring approach will be described in detail in the upcoming deliverable D3.1.

3 Interfaces

Users need to request the ephemeral services they require for their applications. At the same time, the ephemeral services need to interact with other components developed by other work packages. All these interactions happen through interfaces, which are described in the next paragraphs.

3.1 Ephemeral Services Lifecycle Control

As discussed in Section 2.3.3, the user will request its ephemeral environment by providing a YAML file listing the services he wants. This file will be provided to the REST API to be translated into operations to start and configure the ephemeral services. Here, we consider the use of a YAML file because there are too combinations of ephemeral services and namespaces that can be instantiated. In this section we will not provide a detailed final definition of this file format, but focus more on the semantics components it needs to take into account and on basic examples to understand its logic. The final detailed format will be fixed upon implementation.

The YAML file should first list all the services required to run the application. In practice, it will name of the ephemeral services, describe their type and their location on the cluster modules. This basic definition will be completed with a list of parameters used to implement the service or tune it. We recall here that there are various types of services:

Dataset and namespace instance: We consider here the NFS and S3 servers which are the key components to make the datasets and namespaces alive and accessible. In other words, these ares the services used to instantiate those namespaces.

Accelerators: We consider here the services like SBB, GBF, SBF or the local Motr which are responsible of making a local copy of the data to accelerate the data transfers. In the file description we might consider two subcategories. First the accelerators which run by themselves to accelerate the access to an existing storage in the cluster. We can give as an example an instance of SBB to access files stored into the cluster Lustre filesystem. Second, the accelerator used to accelerate the accesses to an IO-SEA namespace. In this case we need to take care of linking the accelerator to the related namespace service so we can make the local setup in the data node to access it.

```yaml
---
  services:
    - name: cats
      location: cpu_module
      cortx_nfs:
        namespace: my_cats_namespace_id
        mountpoint: /media/myuser/cats
    - name: sbb_on_cpu_module
      location: cpu_module
      service_dependencies:
        - cats
```

```
    smart_burst_buffer:
      targets: /nfs/myuser/dir1,/nfs/myuser/dir2,/media/myuser/cats
      datanodes: 2
      datanode_cores: 16
      datanode_mem: 4GB
      datanode_flash: 32GB
```

Listing 3.1: Basic example describing a dataset instance and a SBB service instance.

Looking at the applications presented by Work Package 1, it appears that many of them are used to run various steps to progress in a global workflow. Each of these steps might require special services which might not be required by other steps. Hence, we propose to add a step definition in the YAML file to describe the lifecycle or the services over the workflow. At the moment, we consider describing the steps of an application and leave it to the user to move from one step to another. Each step will be described by a name and a list of services to be enabled. An example of step definition is shown here:

```
---
  services:
    - name: cats
      //...
    - name: sbb_on_cpu_module
      //...
    - name: sbb_on_gpu_module
      //...

  steps:
    - name: step_1
      services:
        - cats
        - sbb_on_cpu_module
    - name: step_2
      services:
        - cats
        - sbb_on_gpu_module
```

Listing 3.2: Example of step definition in the YAML service file.

The given YAML file describes a service implementing the *cats* namespace and being enabled for the two steps. In addition, we enable a SBB instance on top of this cat service on the local module,CPU for the `step_1` and GPU for the `step_2`. The REST API will be responsible for stopping the first SBB and starting the second one while moving from one step to another. Notice that if both steps are running in parallel we will have the two SBB instances also running at the same time for the two jobs requiring them.

We note here that we decided to define the service location directly in the service definition such that the users controls explicitly where they want the service to be deployed. We have discussed an alternative which consists of moving this declaration into the step giving the REST API the responsibility to find the right location for a services if the service is used in multiple steps. One

counter argument against this approach is that the REST API does not have information on the relative weight of each step so it might have difficulties in making the right decision if a server should be used by steps in various locations.

We described the philosophy of the YAML file to be used by the user to express their requirements in terms of ephemeral services. In practice, this file should be transmitted to the REST API via Slurm such that it is used during the instantiation of the listed services. In terms of the command line, this might be done by using the srun or sbatch commands, giving an option with the YAML file and other options specifying which steps the job is running.

The semantics are not yet strictly defined, but they can translate to something inspired by what is done in the Atos Flash Accelerator product with the Slurm burst buffer plugin and comes to:

```
# launch a single-step workflow
srun --ephemeral "services-single.yaml" --bb "EPH" ./myapp

# start a workflow for a multi-step run
srun --ephemeral "services-session.yaml" --bb "EPH NAME=my_workflow START" /bin/true
# or can be wrapped in a more friendly way
io-sea-eph-session start --name "my_workflow" --yaml "services.yaml"

# run a first step
srun --bb "EPH NAME=my_workflow STEP=step_1" ./my_app_step_1

# run a second step
srun --bb "EPH NAME=my_workflow STEP=step_2" ./my_app_step_2

# terminate a workflow for a multi-step run
srun --bb "EPH NAME=my_workflow END" /bin/true
# or can be wrapped in a more friendly way
io-sea-eph-session stop --name "my_workflow"
```

Listing 3.3: Example of desired semantics on the Slurm command side.

We had an interesting exchange with Work Package 1 with respect to how the semantics should be used by their applications. One point is that it might be interesting to have several steps used in parallel because the workflow of their application is not necessarily linear. We might want to also go back from one step to a previous one without considering a specific order. We notice here that it does not change the definition we made before but just requires to be flexible enough in the implementation of the REST API. If we want a service running during the whole workflow, we might list it in every step for it not to be stopped when moving from one step to another.

Based on recent discussions regarding non-linear workflow support, we are currently converging on a slight adaptation of the step service list definition in the YAML file so we can optionally annotate the service name with a start/stop keyword to declare what is the desired operation to apply when the step starts. This might answer the question of the service end of life in the non-linear scenario. Consider:

```
---
services:
  - name: cats
```

```
steps:
  - name: step_1_start_cats
    services:
      - cats:start
  - name: step_2_stop_cats
    services:
      - cats:stop
```

Listing 3.4: Example of step definition in the YAML service file.

3.2 REST API proof of concept

During the past months a small proof of concept (POC) was quickly developed to try and play with this semantics. This was useful in discussing with the partners to converge on a global picture of the way we will interact with the user. It was done as a REST API based on Python's Flask-RESTPlus framework and Yorc as orchestrator configured to be able to launch a fake SBB script. Its implementation currently considers the linear workflow semantics and does not allow multiple steps active in parallel. The POC is installed in a small Docker cluster composed of:

- A container for the REST API.
- A container for Yorc.
- A container to be scaled for a fake SBB datanode.

The use of Docker Compose permits one to quickly setup the POC environment on any development machine and play with it. Although this might be usable for the SBB server, we will have to think how to extend it if we at some point want to have real services including Motr which requires a virtual machine instead of a container. Fig. 6 show the topology of the POC mini cluster built by docker-compose.

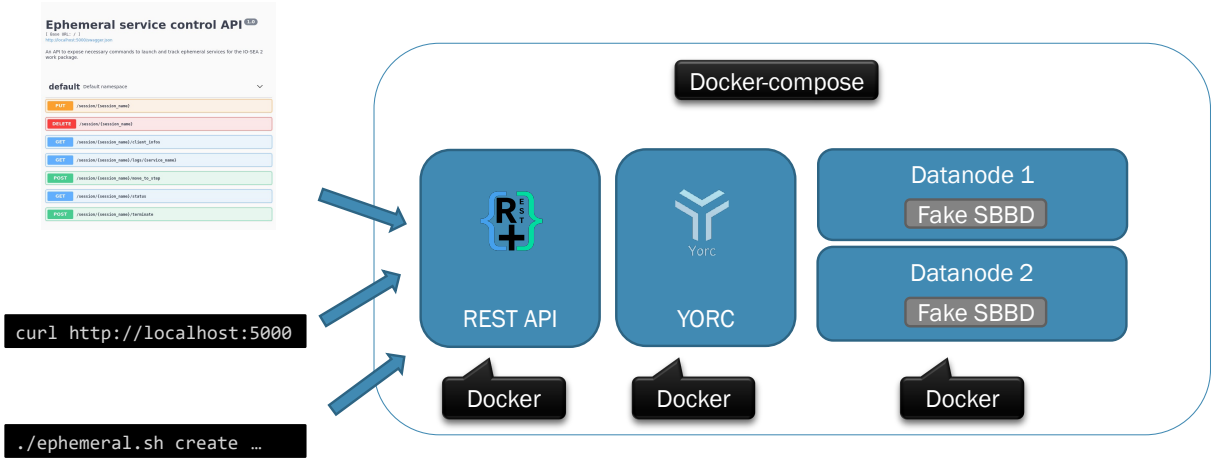


Figure 6: Mini cluster built by docker-compose to run the REST API proof of concept.

This REST API will mostly be controlled by Slurm, so for most operations it will not be directly exposed to the user, but it shows how Slurm will interact with it. The commands currently considered are:

```
# register a new session giving the YAML file
./ephemeral.sh create {SESSION_NAME} {YAML_FILE}

# move to the given step
./ephemeral.sh step {SESSION_NAME} {STEP_NAME}

# Get status of the services attached to the session
./ephemeral.sh status {SESSION_NAME}

# Get the services information needed to configure the client
./ephemeral.sh info {SESSION_NAME}

# Get the logs for the given service in the session
./ephemeral.sh logs {SESSION_NAME} {SERVICE_NAME}

# Terminate the running services
./ephemeral.sh terminate {SESSION_NAME}

# Purge all information of the service
./ephemeral.sh purge {SESSION_NAME}
```

Listing 3.5: List of commands available to manipulate the REST API POC.

3.3 Data Movement

When the ephemeral services have been spawned in the data nodes, we might want to prefetch some of these objects into those services so the user can get a faster access to these objects. This prefetch operation should be done by the HSM mechanism from Work Package 4 when considering a local Motr instance.

For the Flash Accelerator product (SBB, SBF and GBF) we might, in a first stage, consider the user calling Slurm to run a job dedicated to read the data or make a copy to the local storage.

A more advanced feature would be to make the HSM commands from Work Package 4 interacting with the ephemeral services and being able to move to data not only inside Motr, but also into the Flash Accelerator services.

As an even more advanced approach, we might be able to put the prefetch requests in the YAML file for the burst buffer plugin moves the data before running a job on the related ephemeral services. This approach might call the HSM command as a backend.

As it is an advanced feature and can be considered as an extension, we can work on it when all the other features are available. In the YAML file, it could take the following form:

```
---
services:
  - name: cats
```

```
    //...
  - name: sbb_on_cpu_module
    //...

steps:
    //...

prefetch:
  - service: sbb_on_cpu_module
    members:
      - cats/black-*.jpg
      - cats/gray-*.jpg
```

Listing 3.6: Possible skeleton YAML file semantics to prefetch files.

This part is still in preliminary discussions but we can think of an extension providing a more fine grain control via data movers which can be called from the steps.

```
---
services:
  - name: nfs_lqcd
    cortx_nfs:
      namespace: my-run-2021-12-1-lqcd
      mountpoint: /mnt/USER/lqcd

data-movers:
  - name: lqcd
    service: nfs_lqcd
    prefetch-on-step-load:
      level: flash
      elements:
        - gauges/*.hdf5
    archive-on-step-finish:
      level: tapes
      elements:
        - gauges/*.hdf5

steps:
  - name: default
    module: cpu
    services:
      - nfs_lqcd
    data-movers:
      - lqcd
```

Listing 3.7: Longer proposal from the YAML file semantics to prefetch files.

4 The LQCD Use-Case

This section will describe some exchanges with the Work Package 1 about the ephemeral services usage by applications. The goal is to start thinking on how applications can use the semantics we have presented in the previous chapter. Here we will focus on the LQCD use case [14] from Work Package 1.

4.1 LQCD Application Phases And Files

As shown in Fig. 7, the use case operates in four phases. A first phase (A) will take the input config file and will generate a large number of files containing the gauge fields computed by the Markov chain use by this step. We then use 10% of those files as input for the phase B which reads the input file and and a configuration file in order to produce propagator files. Phase C take those propagators files to aggregate the results in the final output file which ends the campaign and phase D will run post-analysis on the generated output file. This cycle is executed several times, phase A producing around 400 files per run, which are used by the next phases. In other words, the user runs phases A, B, C and D and and then check the results. If the obtained results do not have the desired precision, the user will start again from phase A to get more data in order to improve the final precision. When the results have the desired precision, the workflow ends.

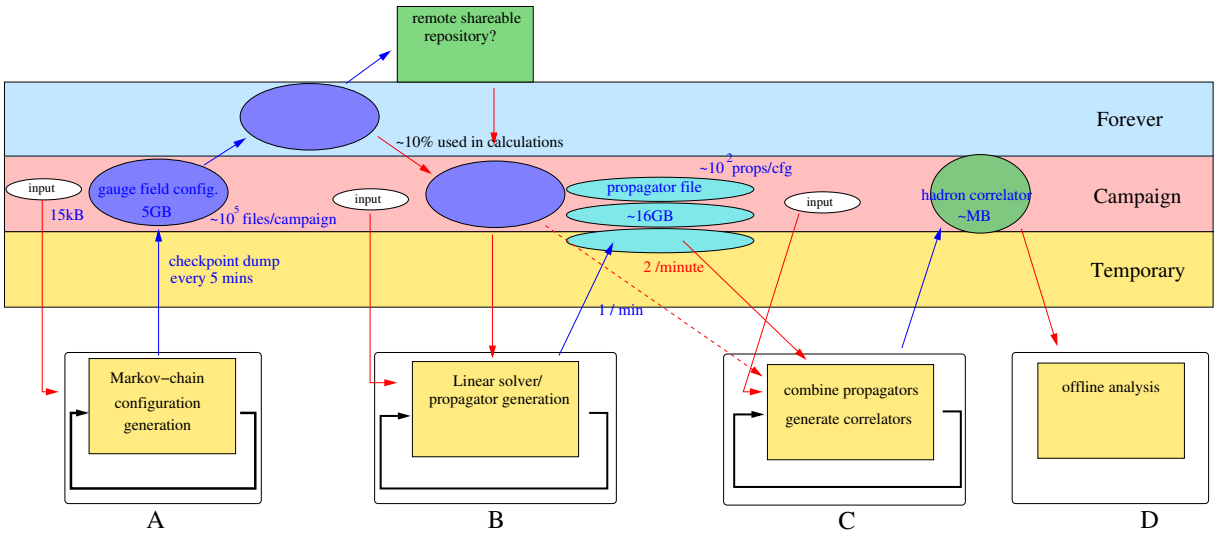


Figure 7: Description of the workflow of a campaign for the LQCD application.

This description shows that there are several categories of files to be used or generated by the application:

- The configuration files used by phase A and B.
- The gauge field files generated by A and read by B.

- The propagator files generated by B read by C which can be considered as temporary files in the workflow.
- The final output file aggregating the results from C and used by D.

Note here that the files from phase A can be archived for a long-term storage.

4.2 Mapping to Datasets And Namespaces

In the IO-SEA project we had several cross-work-package exchanges where the concepts of dataset and namespace were discussed to organize the data stored into the object-store. These concepts will in practice have consequences on the ephemeral services as we will need to instantiate the namespaces via the NFS or S3 server to make them effectively accessible by the user.

Considering the list of files used by the LQCD application, we can distinguish three different approaches to organize them into datasets. This can be done by going from the simpler one to the finer grain one:

- We can first consider an approach close to the current way of using a parallel file system. It consists of creating a unique dataset and POSIX namespace to store all the files generated and used by the campaign.
- The gauge-field files might be archived for long term storage, while the rest of the files are temporary. To express this, we might distinguish their storage and create three datasets. One for the gauge-field files from phase A, another one to store the propagator files and a last one to store the final results and all the configuration files used to run each phase.
- We can ultimately consider splitting the datasets in a finer grain strategy by creating a dataset for each call of phase A run and its following steps. This means creating a dataset for the 400 gauge-fields generated by a unique run of the phase A application. Again creating a dataset for each run of the phases B and C.

The coming sections will describe how we see the implementation of these three approaches.

4.3 User Semantics with a Unique Namespace

As described, the first scenario consists of creating a unique dataset and POSIX namespace for the whole campaign. It gives freedom to the application to handle the files in this namespace. The drawback of this approach is the difficulty that arises when we would like to archive the files from phase A as they are mixed with the temporary files from phase B in the dataset. This might require the removal of the files we do not want to archive or to copy the gauge-field files to another namespace.

The service YAML file will be written as listed here:

```
--
services:
  - name: lqcd-nfs
    location: cpu-module
    cortx_nfs:
      namespace: my-run-2021-12-1-lqcd
      mountpoint: /media/USER/lqcd

steps:
  - name: default
    services:
      - lqcd-nfs
```

Listing 4.1: YAML file for the first scenario of LQCD.

As discussed in the previous sections, we see here the appearance of the NFS service to instantiate the LQCD namespace which must have been previously created by calling a specific command. We have now a unique step in terms of storage with a unique service to setup at start-up and to destroy at the end of the campaign. We chose to name it "default" and it lists all the available services in it (a single service in this case). Here we can consider a simplification which allows the user to not define steps when there is only one. We chose to make it explicit here to understand the logic compared to the coming approaches.

As we get the YAML file we now need to run commands to run the application and exploit the described ephemeral services. Here, we need to:

- Create the unique dataset and namespace which will be initially empty.
- Upload the initial configuration file into the namespace.
- Register the ephemeral service YAML description file to start the storage session in the REST API.
- Run the phase A.
- Produce the configuration file for phase B and push it to the namespace.
- Run the phase B
- Run the phase C.
- Run the phase D.
- Eventually loop again on step A, B, C, D to improve the final precision
- Terminate the ephemeral storage session.

We notice here that the cross-work-package meetings did not yet defined precisely the commands used to handle the datasets and namespaces so we propose here a possible solution which might be adapted in future discussions. We list them here under a form which shows the interesting concepts they introduce. Here we mostly consider the availability of an *io-sea-ns* command which allows one to create a namespace and to copy a file from the local filesystem to said namespace. The namespace is considered to be identified by a string. Here, we consider executing these commands from an

interactive login node which leaves open some questions about the way the commands authenticates themselves when accessing the CORTX infrastructure and interacting with the namespace. Those questions will be treated during the on-going cross-work-package discussions.

```
# create namespaces
io-sea-ns create --auto-create-dataset my-run-2021-12-1-lqcd

# fill config NS with the input config file
io-sea-ns put my-run-2021-12-1-lqcd ./gauge.cfg

# register the ephemeral session
srun --eph-services "services.yaml" --bb "EPH START NAME=LQCD_1" true

# Run step A
srun --bb "EPH NAME=LQCD_1 STEP=default" ./lqcd_step_A /mnt/USER/lqcd/gauge.cfg

# prepare run of step B with its config file
io-sea-ns put my-run-2021-12-1-lqcd ./${ID}/propagator-run.cfg

# run step B & C & D
srun --bb "EPH NAME=LQCD_1 STEP=default" ./lqcd_step_B \
     /mnt/USER/lqcd/${ID}/propagator-run.cfg
srun --bb "EPH NAME=LQCD_1 STEP=default" ./lqcd_step_C \
     /mnt/USER/lqcd/${ID}/propagator-run.cfg
srun --bb "EPH NAME=LQCD_1 STEP=sdefault" ./lqcd_step_D

# POSSIBLY LOOP ON PREVIOUS STEPS (A/B/C/D) ?

# finish the session
srun --bb "EPH NAME=LQCD_1 END" true
```

Listing 4.2: Running the LQCD campaign with the single namespace approach.

We might at some point want to add a burst buffer to accelerate the data access to the namespace. In this case, we will only have to change the YAML file to obtain the following example. Notice the addition of the burst buffer description and its dependency on the NFS instance allowing the ephemeral service manager to know that it also needs to mount this service on the data node hosting the burst buffer.

```
---
services:
  - name: lqcd-nfs
    location: cpu-module
    cortx_nfs:
      namespace: my-run-2021-12-1-lqcd
      mountpoint: /media/USER/lqcd
  - name: lqcd-burst-buffer
    location: cpu-module
    service_dependencies:
      - lqcd-nfs
    smart_burst_buffer:
```

```
        targets: /nfs/USER/lqcd
        datanodes: 2
        datanode_cores: 16
        datanode_mem: 4GB
        datanode_flash: 32GB

  steps:
    - name: default
      services:
        - lqcd-nfs
        - lqcd-burst-buffer
```

Listing 4.3: YAML file for the first scenario of LQCD with a burst buffer layer.

4.4 User Semantics With Phase Dedicated Namespaces

As discussed in Section 4.2, the second approach consists of using dedicated datasets and namespaces for every phase of the application. This means we will create a separate namespace for phase A, one for phases B and C, and a third one to store the configuration files and the final results file. In this case, the service YAML file will look like:

```
---
  services:
    - name: nfs_gauge_fields
      cortx_nfs:
        namespace: my-run-2021-12-1-gauge-fields
        mountpoint: /mnt/USER/gauge-fields
    - name: nfs_propagators
      cortx_nfs:
        namespace: my-run-2021-12-1-propagators
        mountpoint: /mnt/USER/propagators
    - name: nfs_config_and_out
      cortx_nfs:
        namespace: my-run-2021-12-1-config-and-out
        mountpoint: /mnt/USER/config-and-out

  steps:
    - name: step_A
      module: cpu
      services:
        - nfs_gauge_fields
        - nfs_config_and_out
    - name: step_B
      module: cpu
      services:
```

```
      - nfs_gauge_fields
      - nfs_propagators
      - nfs_config_and_out
  - name: step_C
    module: cpu
    services:
      - nfs_propagators
      - nfs_config_and_out
  - name: step_D
    module: cpu
    services:
      - nfs_config_and_out
```

Listing 4.4: YAML file for the second scenario of LQCD.

On the command line, it should look very similar to the previous approach mostly changing the creation of several namespaces instead of a single one, and the use of the step definition in the phase calls.

```
# create namespaces
io-sea-ns create --auto-create-dataset my-run-2021-12-1-gauge-fields
io-sea-ns create --auto-create-dataset my-run-2021-12-1-propagators
io-sea-ns create --auto-create-dataset my-run-2021-12-1-config-and-out

# fill config NS with input config file
io-sea-ns put my-run-2021-12-1-config-and-out ./gauge.cfg

# register EPH session
srun --eph-services "services.yaml" --bb "EPH START NAME=LQCD_1" true

# Run step A
srun --bb "EPH NAME=LQCD_1 STEP=step_A" ./lqcd_step_A /mnt/USER/lqcd/gauge.cfg

# prepare run of step B with its config file
io-sea-ns put my-run-2021-12-1-lqcd ./${ID}/propagator-run.cfg

# run step B & C & D
srun --bb "EPH NAME=LQCD_1 STEP=step_B" ./lqcd_step_B \
    /mnt/USER/lqcd/${ID}/propagator-run.cfg
srun --bb "EPH NAME=LQCD_1 STEP=step_C" ./lqcd_step_C \
    /mnt/USER/lqcd/${ID}/propagator-run.cfg
srun --bb "EPH NAME=LQCD_1 STEP=step_D" ./lqcd_step_D

# POSSIBLY LOOP ON PREVIOUS STEPS (A/B/C/D) ?

# finish the session
srun --bb "EPH NAME=LQCD_1 END" true
```

Listing 4.5: Running the LQCD campaign with the per-phase namespace approach.

4.5 User Semantics With Many Per-Step Namespaces

This section will now describes the per-step namespaces scenario using dedicated datasets for each run of the B/C phases. The interest for this approach is the finer grain tracking of the accessed files. This tracking can be used to provide hints to the lower storage manager to better know how to move the data from one layer to another. It also permits to use smaller namespaces which might be easier to handle in the storage hierarchy.

As previously, we first build the YAML file to declare our services. Contrary to the previous examples, we consider here that we do not start a session for all the steps but use a single YAML file for each job. We first define the file for step A.

```yaml
---
  services:
    - name: nfs_gauge_fields
      cortx_nfs:
        namespace: my-run-2021-12-1-gauge-fields
        mountpoint: /mnt/USER/gauge-fields
    - name: nfs_config
      cortx_nfs:
        namespace: my-run-2021-12-1-config
        mountpoint: /mnt/USER/config

  steps:
    - name: default
      module: cpu
      services:
        - nfs_gauge_fields
        - nfs_config
```

Listing 4.6: YAML file for the step A of the third scenario of LQCD.

For step B we create a namespace with a unique gauge-field file in it. We consider cloning the complete namespace used by A being backed by the same dataset with a filter exporting a single file. We notice here that this filter-based approach is currently considered as an advanced feature we might not want to deliver; although we wanted to expose it here. In the non-available case we can just use the same namespace as for step A.

Note that we now introduce some kind of template-based format with variables in it (${ID}) to distinguish the various namespaces to be used. We consider here just replacing the variable by its value in the template before giving the file to Slurm and the REST API.

```yaml
---
  services:
    - name: nfs_gauge_fields
      cortx_nfs:
        namespace: my-run-2021-12-1-gauge-field-${ID}
        mountpoint: /mnt/USER/${ID}/gauge-fields
    - name: nfs_propagators
```

```
      cortx_nfs:
        namespace: my-run-2021-12-1-propagators-${ID}
        mountpoint: /mnt/USER/${ID}/propagators
    - name: nfs_config
      cortx_nfs:
        namespace: my-run-2021-12-1-config
        mountpoint: /mnt/USER/config

  steps:
    - name: default
      module: cpu
      services:
        - nfs_gauge_fields
        - nfs_propagator
        - nfs_config
```

Listing 4.7: YAML file for the step B of the third scenario of LQCD.

Step C this follows the same philosophy:

```
---
  services:
    - name: nfs_propagators
      cortx_nfs:
        namespace: my-run-2021-12-1-propagators-${ID}
        mountpoint: /mnt/USER/${ID}/propagators
    - name: nfs_hadron_correlator
      cortx_nfs:
        namespace: my-run-2021-12-1-hadron-correlator-${ID}
        mountpoint: /mnt/USER/${ID}/hadron-correlator
    - name: nfs_config
      cortx_nfs:
        namespace: my-run-2021-12-1-config
        mountpoint: /mnt/USER/config

  steps:
    - name: default
      module: cpu
      services:
        - nfs_propagator
        - nfs_hadron_correlator
        - nfs_config
```

Listing 4.8: YAML file for the step C of the third scenario of LQCD.

We terminate with the file for step D:

```
---
  services:
```

```
  - name: nfs_hadron_correlator
    cortx_nfs:
      namespace: my-run-2021-12-1-hadron-correlator-${ID}
      mountpoint: /mnt/USER/${ID}/hadron-correlator
  - name: nfs_config
    cortx_nfs:
      namespace: my-run-2021-12-1-config
      mountpoint: /mnt/USER/config

steps:
  - name: default
    module: cpu
    services:
      - nfs_hadron_correlator
      - nfs_config
```

Listing 4.9: YAML file for the step D of the third scenario of LQCD.

We remark here that we defined a file per step but we might want to think about handling the variable semantics inside the REST API and injecting the variable value from the Slurm commands when calling the steps. Again this can be seen as an advanced feature if we already have an operational platform with the basic mode.

On the command line it we use something similar the the other approaches. Notice that we do not start a session here but use separated jobs. As we just discussed this can be improved with an advanced template semantics in the REST API.

```
# create namespaces
io-sea-ns create --auto-create-dataset my-run-2021-12-1-gauge-fields
io-sea-ns create --auto-create-dataset my-run-2021-12-1-propagators
io-sea-ns create --auto-create-dataset my-run-2021-12-1-hadron-correlator
io-sea-ns create --auto-create-dataset my-run-2021-12-1-config

# fill config NS with input config file
io-sea-ns put my-run-2021-12-1-config ./gauge.cfg

# Run step A
srun --eph-services "services-phase-A.yaml" --bb "EPH STEP=default" ./lqcd_step_A
    /mnt/USER/lqcd/gauge.cfg

# prepare run of step B with its config file
io-sea-ns put my-run-2021-12-1-lqcd ./${ID}/propagator-run.cfg

# create the namespace for step B & C
io-sea-ns create --clone my-run-2021-12-1-gauge-fields \
                --filter gauge-field-${ID}.hdf5
                my-run-2021-12-1-gauge-field-${ID}
io-sea-ns create --auto-create-dataset my-run-2021-12-1-propagator-${ID}

# run step B & C & D
```

```
srun --eph-services "services-phase-B.yaml" --bb "EPH STEP=default -DID=${ID}"
    ./lqcd_step_B \
    /mnt/USER/lqcd/${ID}/propagator-run.cfg
srun --eph-services "services-phase-C.yaml" --bb "EPH STEP=default -DID=${ID}"
    ./lqcd_step_C \
    /mnt/USER/lqcd/${ID}/propagator-run.cfg
srun --eph-services "services-phase-D.yaml" --bb "EPH STEP=default -DID=${ID}"
    ./lqcd_step_D

# POSSIBLY LOOP ON PREVIOUS STEPS (A/B/C/D) ?
```

Listing 4.10: Running the LQCD campaign with the per job namespace approach.

Here we can criticize that it requires much more actions from the user and also that it will require the user to take care of the book-keeping of the many created namespaces to destroy them at some point which is not shown in the example. Mitigation actions will be analysed at the project level.

5 Software

5.1 Flash Accelerators

The Flash Accelerators product is part of the Smart Data Management Suite developed by Atos. It currently has three working modes (also called accelerators) which take advantage, in different ways, of the NVMe devices installed on the data nodes.

Integrated with Slurm, upon a user's request, one of the three working modes is activated, providing ephemeral services to the job. The user also expresses their requirements in terms of sizing, mountpoints, files to handle, etc.

Each working mode (also called accelerator) can be associated to a single Slurm job, or be created independent of them. In the first case, the accelerator is provisioned before the job enters the "*running*" state, and removed when the job is completed. No other job can use that accelerator. In the second case, the accelerator is created by launching a special initialization Slurm job, but, instead of destroying it when the job is completed, the accelerator is kept for further use. Other jobs launched by the same user can attach to that accelerator and access the data stored in it. When no longer needed, the accelerator can be destroyed by launching a special destruction job.

Smart Burst Buffer

The first working mode is called *Smart Burst Buffer*. SBB is an intelligent burst buffer providing a POSIX interface on top of an existing POSIX parallel filesystem or an S3-compatible object-store. When the back-end is an S3-compatible object-store, it also takes care of mapping the objects to provide the user a POSIX interface. It provides two levels of local cache: RAM and flash storage.

It provides a library that is linked at runtime (using LD_PRELOAD) in order to intercept the POSIX operations of the application. By doing so, it can reroute operations to the SBB server running on the data nodes in a way transparent to the application and allowing them to remain unchanged. Fig. 8 depicts this.

To maximize the InfiniBand network bandwidth, several SBB servers can be launched on the data nodes, each serving on a separate InfiniBand interface, a separate subset of the files.

All NVMe devices are grouped in a Logical Groups (VG) using the Logical Volume Manager (LVM). This achieves more flexibility than just assigning raw devices, in particular in terms of size. Each SBB server has an associated Logical Volume (LV) that will be used as a local cache.

The principal use-case for this accelerator is to catch write bursts such as check-points, but can also be seen as a more generic cache.

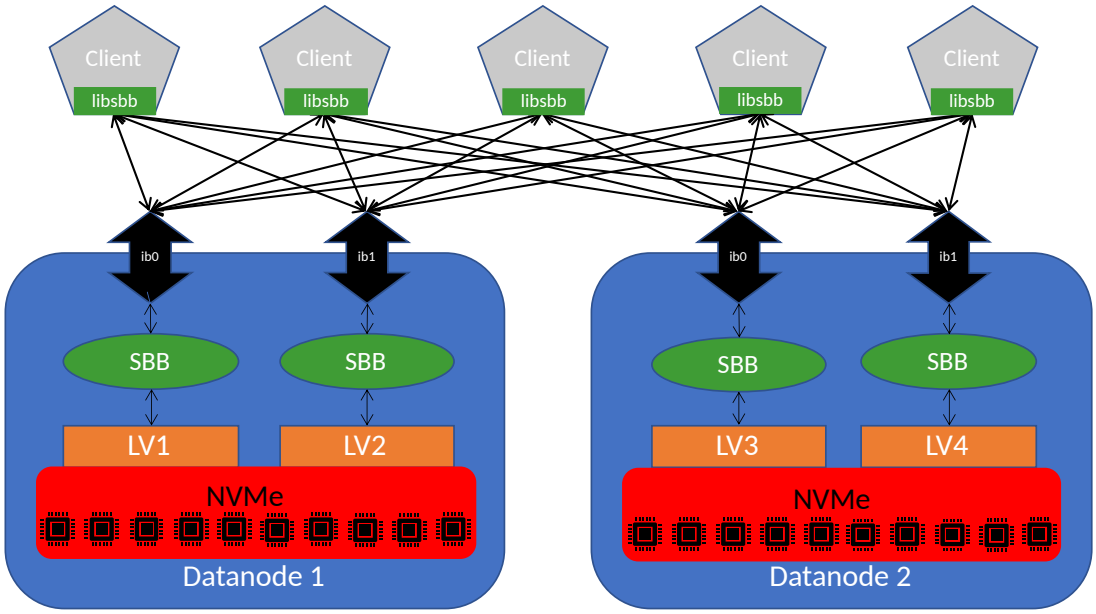


Figure 8: Smart Burst Buffer.

Smart Bunch of Flash

With SBF each compute node running a step of a workflow has a dedicated NVMe storage present on the data nodes exported to the compute nodes through NVMe-oF. It can be used as a fast, temporary POSIX storage.

For each compute node, a LV is created on the data nodes and an XFS filesystem created on it. The LVs are then exported using the NVMe-oF protocol. On the compute nodes, the exported LV are seen as local NVMe devices and can thus be mounted as a normal filesystem. The fact that the actual device is located in the data nodes is transparent to the applications.

Each compute node has it own "virtual" NVMe device and no other compute node can access it, making it suitable for situations where a fast local, non-shared storage is required to keep temporary files. This is illustrated in Fig. 9.

Global Bunch of Flash

GBF instantiates an ephemeral parallel BeeGFS filesystem on the data nodes using the NVMe devices. Compute nodes participating to the workflow's step, will mount this filesystem.

The server part of the filesystem is instantiated on the data nodes. The Management and Metadata services are placed on the same data node and associated to a single InfiniBand interface; while the

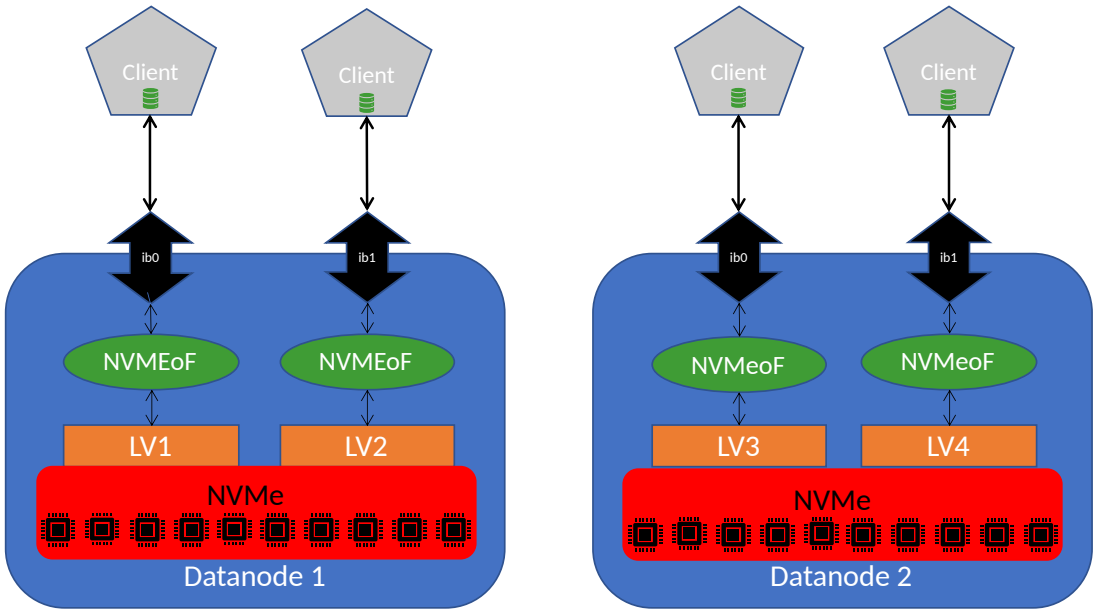


Figure 9: Smart Bunch of Flash.

storage is distributed among all the data nodes associated, each one, to a single InfiniBand interface, allowing the system to maximise the use of the network bandwidth. This can be seen in Fig. 10.

On the compute nodes, the client service is configured to connect to the corresponding filesystem instance and mount it locally.

Its use case is similar to that of SBF, but all the compute nodes can access the data. The filesystem is shared.

5.1.1 Improvements for the IO-SEA Project

Integration into the IO-SEA Project

The Flash Accelerators product is currently integrated with Slurm and, as such, it is possible for the users to request an accelerator when they launch a job (using the *srun* command or a *sbatch* script). It is also possible (of special interest for customers not using Slurm) to "manually" launch the accelerators. To achieve this, some scripts are provided that can be invoked by other schedulers (for instance, OpenStack) to manage the accelerators. These entry-points will be adapted for their use in the IO-SEA project.

In addition, HSM services will be used to preload any needed data and move it back to permanent storage before the ephemeral service is destroyed.

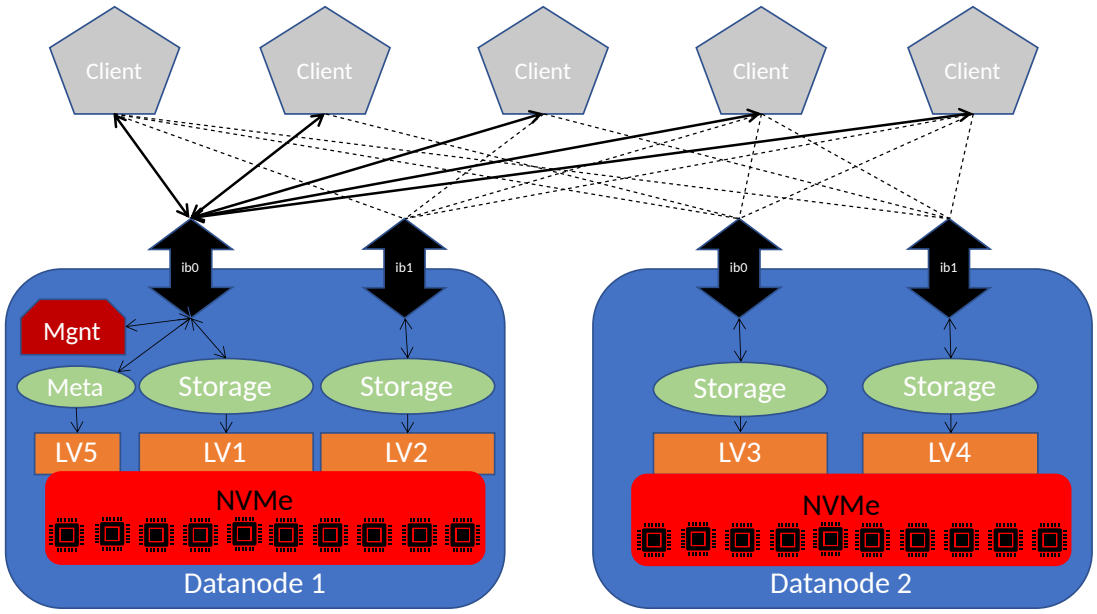


Figure 10: Global Bunch of Flash.

Use of NVRAM

Flash Accelerators currently support NVMe but not yet NVRAM devices. Support will be added during the timeframe of the IO-SEA project allowing Flash Accelerators to take advantage of that hardware, which will be available on the data nodes.

Adding Other Filesystems to GBF

Global Bunch of Flash will be enhanced to support other filesystems in addition to BeeGFS. It will be possible for administrators to deploy their own scripts instantiating different filesystems, which will be invoked by GBF's framework. These scripts will have to provide a predefined API for GBF to be able to invoke it. When launching a job with GBF, user's will be able to specify the filesystem they want to use, from the list of the supported filesystems (those supported natively by GBF, and those for which the administrators provided an instantiation script).

This feature is currently under heavy development and is expected to be delivered in the next release of Flash Accelerators.

5.2 Motr / CORTX

CORTX Motr (formerly “Mero” and initially funded by the SAGE and Sage2 projects [15]) is the foundational open-source object-store used in the IO-SEA project on top of which various storage and I/O services (S3, pNFS, etc.) are exposed [16]. CORTX Motr is essentially an object-store and a KV store that can be used to describe and reference the objects. Data can be organized and stored in a flat hierarchy of objects. Motr forms the core object-store. Motr and all the tools and utilities needed to instantiate, manage and access Motr is called CORTX. CORTX Motr runs on top of various types of storage device technologies (SSD, HDD, NVRAM). The storage devices are assumed to be distributed through networks such as 10/40/100 GbE, InfiniBand, etc.

Motr is accessed through the Motr API that has an access and an extension interface. The access interface is used to drive object and KV store I/O. Since Motr is a very low-level representation of data, higher level “gateways” such as POSIX, pNFS, S3, etc as shown in Fig. 11), can be easily built on top of the Motr API. These gateways are analogous to the services used in IO-SEA ephemeral services.

The extension interface is a publisher/subscriber interface on top of which third party applications can be added (such as HSM applications, Archival applications, Backup applications, fsck-like file system checking utilities, etc.).

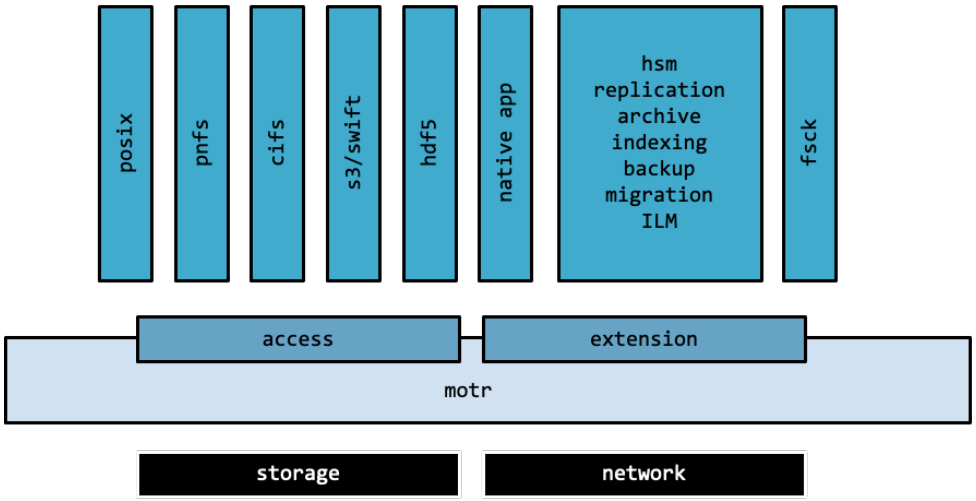


Figure 11: Motr architecture.

Motr is built to be horizontally scalable, where the system grows linearly as more nodes are added with no metadata hot spots and a share-nothing I/O path. Motr can also scale vertically to utilize more memory and processors on the nodes. Motr is also highly fault tolerant with data that can be striped and distributed across the storage network using very flexible erasure coding techniques.

Unlike other object-stores, observability is built into Motr where telemetry information from the Motr stack can be gathered and dialed up/down as needed (through so called structured ADDB, or, Analysis and Diagnostics Data Base records). It is to be noted that Motr runs in user space on any version of Linux.

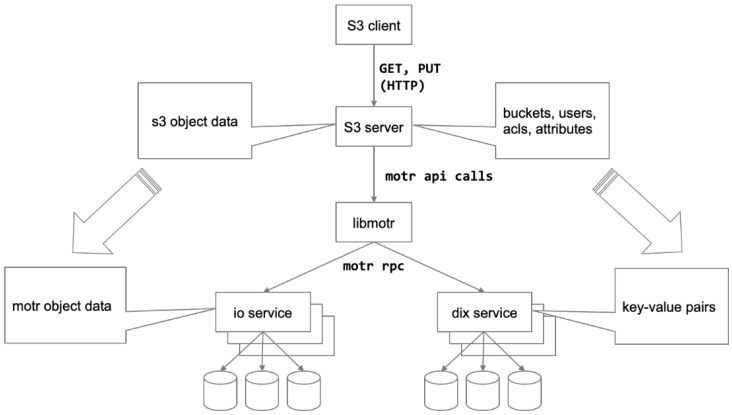


Figure 12: Example of Motr data flow with and S3 service on top.

Fig. 12 depicts the data flow in the Motr stack that uses a service on top of the Motr API, say, an S3 server (which could potentially be an ephemeral service). The S3 objects are mapped to Motr objects and their attributes are stored as key-value pairs. Library *libmotr* is the main piece of the Motr API. I/O service and DIX service form the key components of the Motr service.

Motr objects have FIDs (or OIDs). The objects are mapped across different devices and across different enclosures (the "c"s in Fig. 13) and potentially different racks using Parity Declustered RAID algorithms, with $N + K + S$ (N data units, K parity units and S spare units), where the N, K and S can be manipulated. Each object hence has a layout that describes the mapping to storage resources.

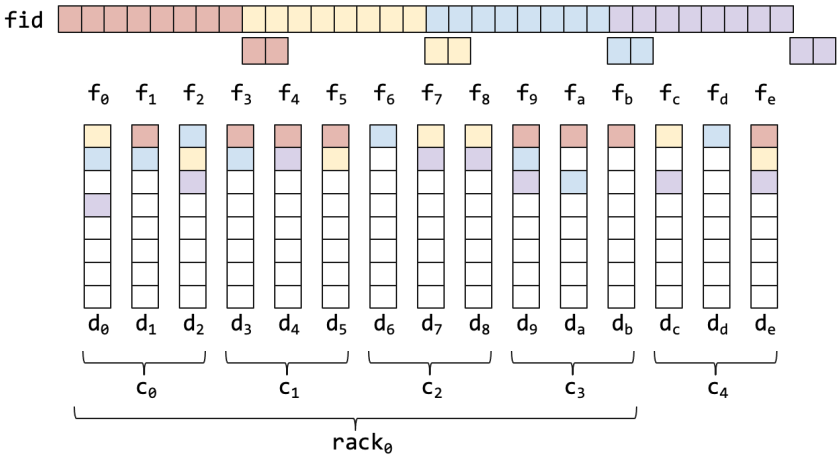


Figure 13: Motr objects mapped to different devices, enclosures and racks.

An object in Motr is an array of blocks. When creating a new object a block size is selected and it is stored according to a layout. There are a few different layouts in Motr, the default is parity de-clustered layout.

In Fig. 13), the object can be seen represented at the top of the figure as horizontal blocks which are divided into groups/stripes.

For each stripe, Motr will compute additional parity blocks. In Fig. 13), for every 8 data blocks, 2 parity blocks are built. Data and parity blocks are then scattered across several devices. If any 2 of the 10 blocks (8 data blocks and 2 parity blocks) are lost. They can be rebuilt from the remaining 8 blocks.

Using parity de-clustering, the striping can be done over 10, 20, 100 or more devices. Having the ability to stripe over multiple devices, repairs can become more efficient since the same set of blocks are scattered over many devices. In the event of device loss or failure. A smaller percentage of the storage capacity needs to be read to complete the repair.

Motr also considers device hierarchy and does not assume that the system will be a flat array of devices. In Fig. 13), each device will be part of an enclosure, which will be part of a rack. As such the striping algorithm can be setup to tolerate different levels of failure and by distributing data across hierarchically arranged devices. Accessing data when a device or an enclosure has failed will still be possible.

Fig. 14 shows a possible mapping of S3 objects to Motr objects that sit behind Motr services (I/O Service).

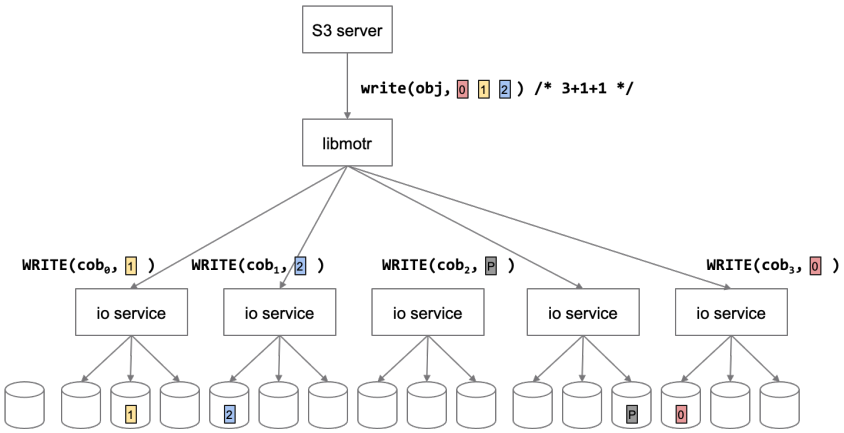


Figure 14: Mapping of an S3 object to different Motr objects (using Motr I/O services) using a specific $N + K + S$ PDRAID configuration.

Motr also has a Distributed Transaction Manager (DTM) such as Motr operations can touch multiple nodes and nodes can fail independently. DTM groups Motr operations into atomic transactions that are atomically dealt with in case of failures. This enables the system to “roll back” to previous stable states of the object storage system in case of any failures.

ADDB provides built in fine grained telemetry which is very useful as the system grows and complexity increases. ADDBs answer questions such as:

- How well the system is utilised?
- Is it failure or expected behaviour?

- Is it system or application behaviour?

ADDB is instrumented on the client and server and has a payload of 16 64-bit values, which can always be turned on. These records can be time-stamped and cross referenced. Listing 5.1 shows an ADDB record in Motr and Fig. 15 of the type of analysis possible:

```
* 2020-02-20-14:36:13.687531192 alloc size: 40, addr: @0x7fd27c53eb20
| node <f3b62b87d9e642b2:96a4e0520cc5477b>
| locality 1
| thread 7fd28f5fe700
| fom @0x7fd1f804f710, 'IO fom' transitions: 13 phase: Zero-copy finish
| stob-io-launch 2020-02-20-14:36:13.629431319, <200000000000003:10000>, count: 8,
    bvec-nr: 8, ivec-nr: 1, offset: 0
| stob-io-launch 2020-02-20-14:36:13.666152841, <100000000adf11e:3>, count: 8, bvec-nr:
    8, ivec-nr: 8, offset: 65536
```

Listing 5.1: Example ADDB record in Motr

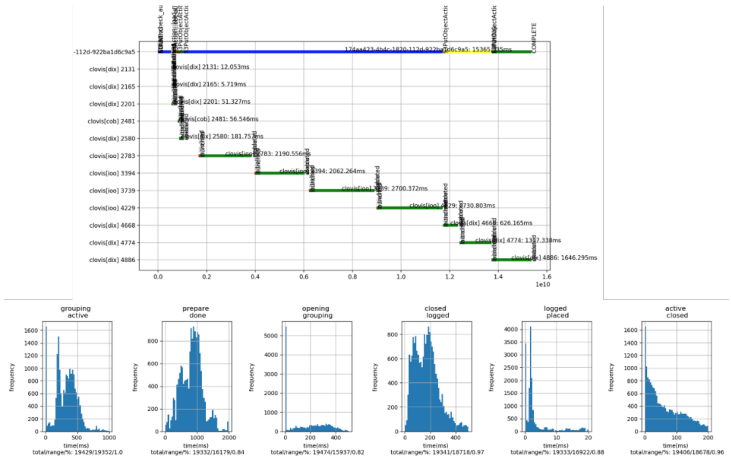


Figure 15: Example ADDB analysis.

ADDB records can also potentially be fed into simulators to understand "what if" scenarios.

5.2.1 Motr API

We here describe the key aspects of the Motr API on top of which various services can be launched. Motr operations are typically asynchronous. Objects are 128-bit persistent identifiers assigned by the user. Objects can be created and deleted, as well as the can be read, written synced. The same goes for the KV store. In addition its possible to GET/PUT/DEL/NEXT on the KV store pairs.

Motr objects are an array of data blocks in the range from 0 to 2^{64} ($[0, 2^{64})$). They have a flat name-space and are network striped as described earlier. There are no usual metadata (attributes,

etc.) exposed to users, such as object size. This is to maximize I/O performance by avoiding updating object attributes. Motr also has internal object attributes such as data layout. There are CREATE/DELETE/WRITE/ALLOC/FREE operations and SCATTER GATHER READ/WRITE operations defined for these objects.

The KV store is distributed and contains bit strings with values accessed as a whole. Iterations are done in the order of keys. Operations allowable are CREATE, DELETE, LOOKUP and LIST for the indices, and, GET, PUT, DEL and NEXT for index queries. Scatter-gather-scatter operations are also permitted. Users use indices to build metadata structures – names-paces and file attributes, etc. Table 1 summarize the various operations.

Entity Operations (for both object and index)	M0_EO_CREATE	**Create an entity**
	M0_EO_DELETE	**Delete an entity**
	M0_EO_OPEN	**Open an entity**
	M0_EO_SYNC	**Flush entity data to storage devices**
	M0_EO_GETATTR	**For Motr internal uses only**
	M0_EO_SETATTR	**For Motr internal uses only**
	M0_EO_LAYOUT _GET	**For composite layout only**
	M0_EO_LAYOUT _SET	**For composite layout only**
Object Operations	M0_OC_READ	**IOs in scatter-gather-scatter fashion**
	M0_OC_WRITE	
Index Operations	M0_IC_GET	**Index queries, including GET, PUT, DEL and NEXT**
	M0_IC_PUT	
	M0_IC_DEL	
	M0_IC_NEXT	
	M0_IC_LOOKUP	**Check an index for an existence**
	M0_IC_LIST	**Given index id, get the list of next indices**

Table 1: Basic operations (objects and indices) through the Motr API.

Object Access	m0_obj_init/fini()	**Initialise/finalise in-memory object data structure**
	m0_entity_create()	**Create and initialise operation for object creation and deletion**
	m0_entity_delete()	
	m0_entity_open()	**Once opened, the object data structure can be used in later READ/WRITE ops**
	m0_obj_op()	**Create and initialise an object operation specified by the opcode**
	m0_sync_op_init()	**Create and initialise a SYNC operation**
	m0_sync_entity_add()	**Add an entity to SYNC operation**
	m0_sync_op_add()	**Add an 'op' to SYNC operation**
Object Lock	m0_obj_lock_init()	**Currently Motr only supports exclusive "whole object" lock in a group**
	m0_obj_lock_fini()	
	m0_obj_lock_get_sync()	
	m0_obj_lock_get()	
	m0_obj_lock_put()	

Table 2: Object access operations through the Motr API.

5.2.2 Improvements for the IO-SEA Project

Fig. 16 shows the overview of Motr usage in the IO-SEA project. Motr API (Client) and services are deployed on the data nodes exposing the NVRAM/NVMe storage resources there as described earlier, exposing the ephemeral services. HSM needs to be involved in moving the data to long-term data storage after the ephemeral services and the Motr services are shut down.

The improvements needed within Motr to work in the ephemeral environment are:

- Ability to address and work with distributed NVRAM resources across the data nodes
- Ability to provide Object ID (OID) management for various services
- Ability to interface with DASI
- Ability to interface with all access protocols within the ephemeral environment – as defined as part of the co-design process
- Support metric gathering and serialization of ADDB records to a common defined IO-SEA format.

Index (Key-value) Access	m0_idx_init/fini()	**Initialise/finalise in-memory object data structure**
	m0_entity_create()	**Create and initialise index creation/deletion operations**
	m0_entity_delete()	
	m0_idx_op()	**Create and initialise index query operation. Motr supports GET, PUT, DEL and NEXT queries**
	m0_sync_op_init	**Flush index key and value to persistent storage devices**
	m0_sync_entity_add m0_sync_op_add	

Table 3: Index operations through the Motr API.

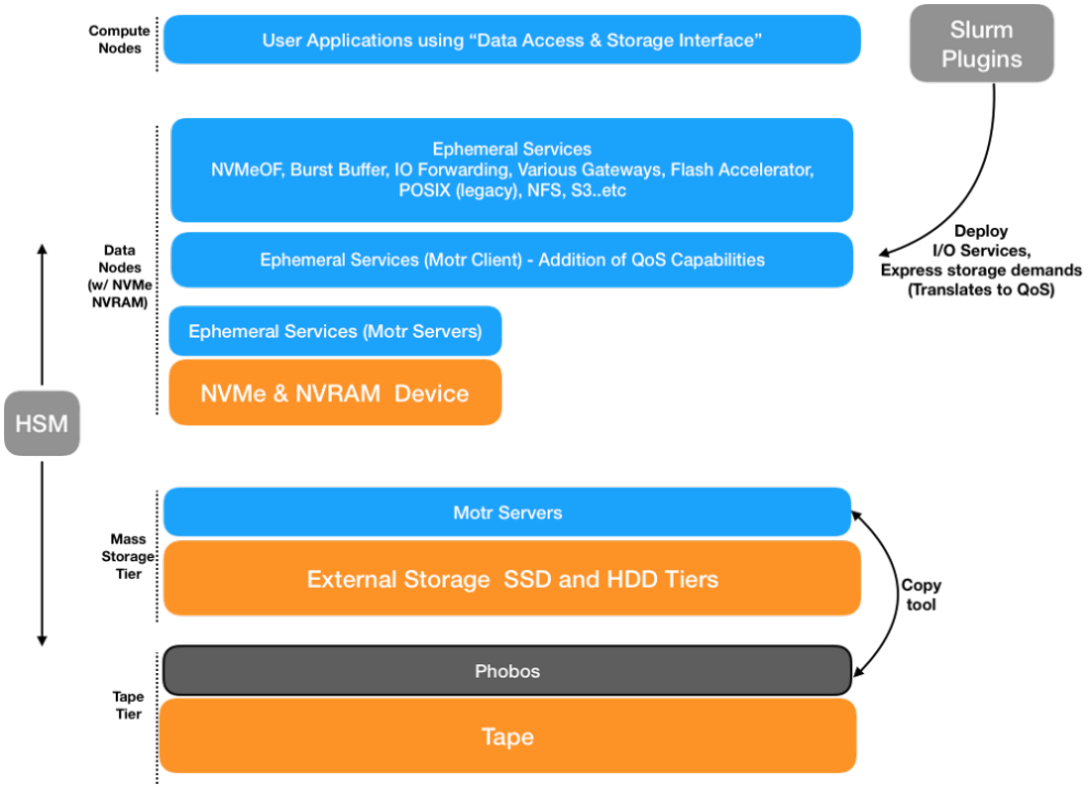


Figure 16: Motr in IO-SEA.

5.3 ParaStation Management

ParaStation Management, the process management facility of ParaStation Modulo, offers a complete process management system that can in turn be combined with an outer and more generic resource manager together with a batch queuing system plus job scheduler like Slurm. The process management of ParaStation Modulo includes the creation of processes on remote nodes, control of the I/O channels of the remotely started processes, and the management of signals across node boundaries.

Since ParaStation Management knows about the dependencies between the processes and threads building a parallel session on a number of nodes of the cluster, it is able to take them respectively into account. That way, processes are no longer independent but form an entity in the same sense as the nodes are no longer independent computers but form a cluster of nodes as a self-contained system. This feature of ParaStation Management for handling distributed processes as a single unit plays an important role especially in the context of job control and allocation-internal scheduling.

One important key to ParaStation's scalability is its efficient communication subsystem for inter-daemon messages. This subsystem, which uses the implementation of a highly-scalable Reliable Datagram Protocol (RDP), is used for resource monitoring as well as for launching and controlling parallel processes by means of a network of ParaStation Daemons (psid). So, for example, this subsystem also performs process pinning, I/O forwarding and signal handling, and it ensures a proper resource cleanup after job termination.

psslurm When used together with Slurm, a ParaStation Management plugin called *psslurm* comes into play. This plugin (shown in Fig. 17) replaces all components of Slurm that are running on each compute node, namely *slurmd* and *slurmstepd*.

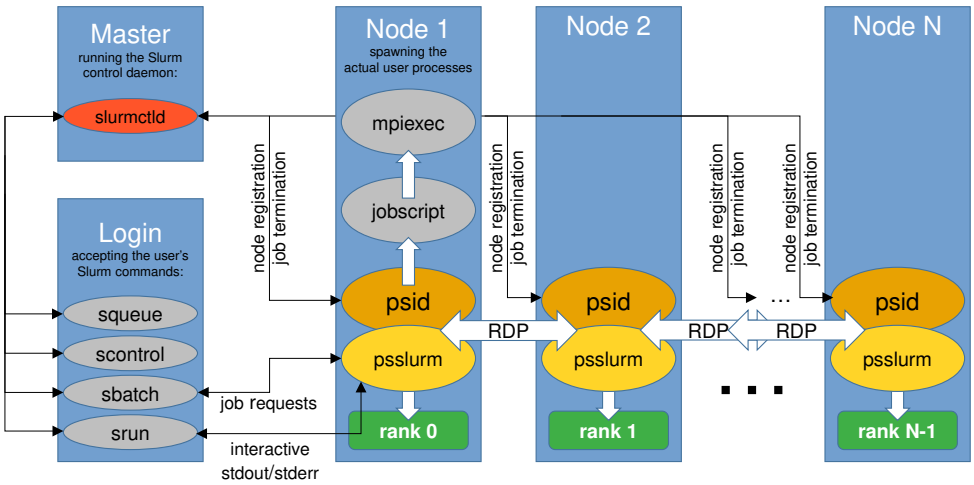


Figure 17: Orchestration between Slurm and ParaStation Management via its psslurm plugin.

For the integration of Flash Accelerators at compute node level the "*Slurm Plug-in Architecture for Node and job Kontrol*" (SPANK) will be used. SPANK provides a generic interface for stack-able plugins which may be used to dynamically modify the job environment. ParaStation Management's psslurm plugin has basic support for SPANK, but not all API calls are fully support yet. The reason for this are different design decisions on how to start the prologue and compute processes.

In earlier versions, Slurm was unable to execute the prologue in parallel on all compute nodes when a job was started. The prologue could only be spawned on the mother superior node. Additionally an error in the prologue execution could re-queue the job, but since the job was already in state "*running*" accounting data was affected. To overcome this shortcomings psslurm is using the slurmctld prologue to execute the prologue on all nodes of the allocation in job state "*configuring*". This enabled the best integration of the ParaStation Healthchecker to verify the node health status before the actual job was even started. But this results in the prologue running in a different process context than in vanilla Slurm.

There is a similar design difference between psslurm (Fig. 18a) and slurmd (Fig. 18b) at the startup phase of the compute processes. In vanilla Slurm the slurmd is executing one slurmstepd process which is responsible for all processes of the local compute node. In contrast the psid spawns one psidforwarder for every compute process in addition to a psslurmstep forwarder. Since the psidforwarder is a child of the psid changes done in SPANK calls processed in the psslurmstep forwarder will not be reflected in the psidforwarder contexts.

Depending on the complexity of the SPANK plugins it can be challenging to adopt it to the ParaStation Management runtime environment. To ensure a seamless support of the SPANK plugins developed in IO-SEA the psid design will be refined with the goal to run SPANK plugins without any modification.

5.3.1 Improvements for the IO-SEA Project

ParaStation Management will be enhanced to support the extended resource management described in this document. The details of the work to be done depends on the implementation details of the components involved and have to be determined at a later stage.

In summary we need to support

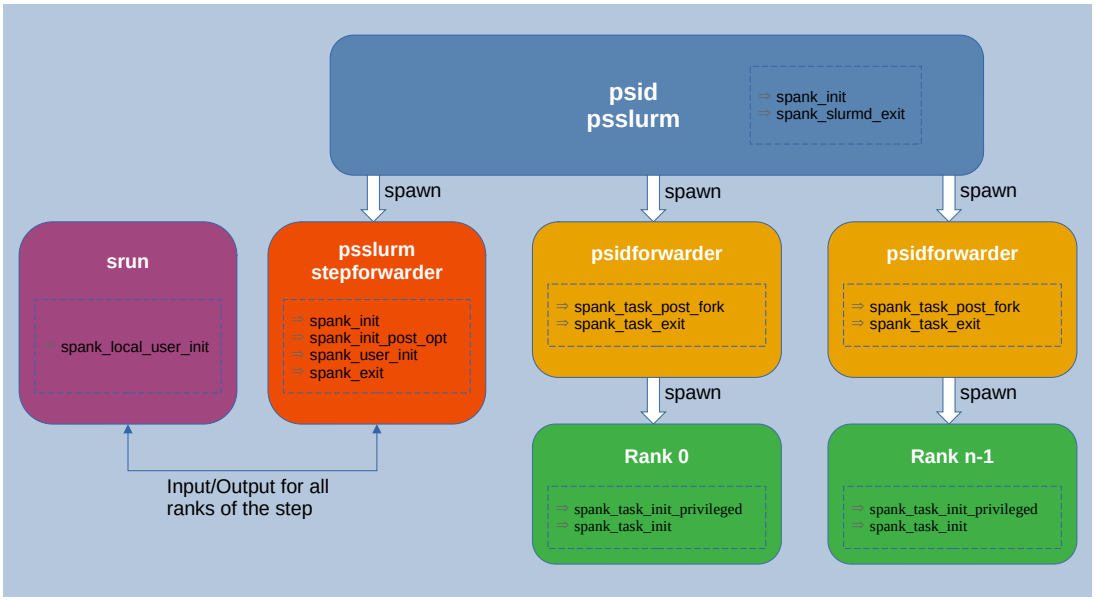
- General information exchange between components of the process management

  This might be forwarding of environment variables or configuration files.

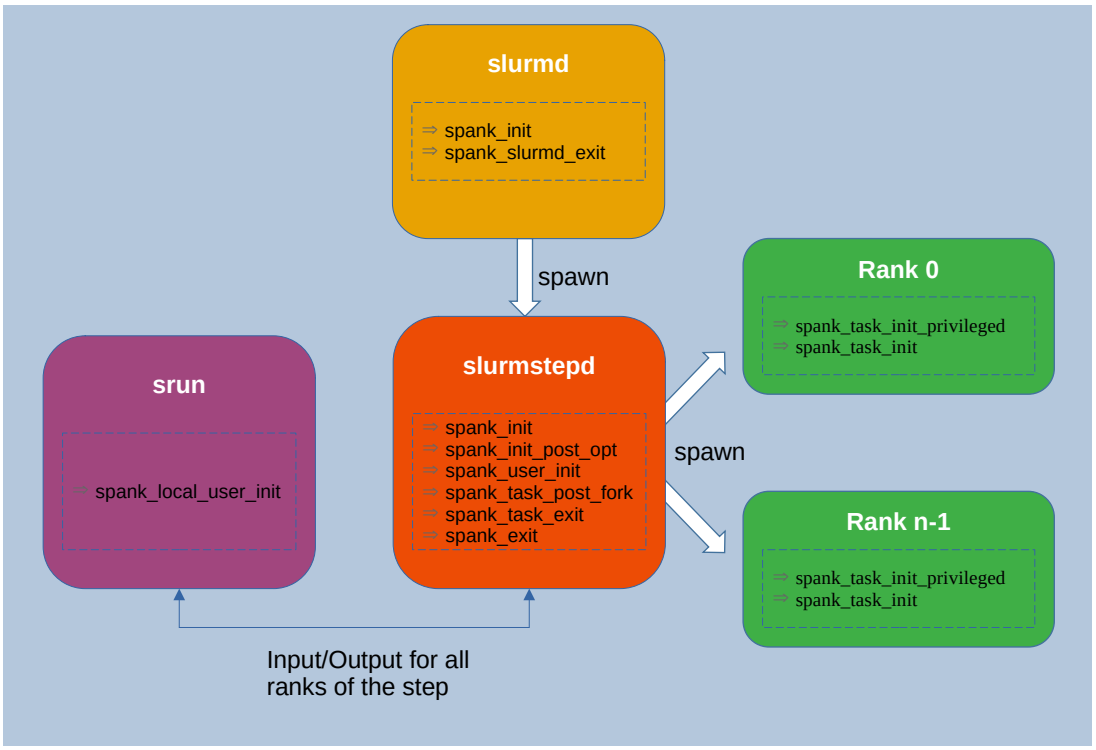
- Workflow support

- SBB Slurm plugin developed by Atos

  Since *psslurm* replaces parts of Slurm that are responsible to run parts of Slurm plugins, it sometimes needs adjustments (to *psslurm*) to smoothly run more complex Slurm plugins as we expect to be used in the IO-SEA architecture.



(a) SPANK API implementation in ParaStation Management



(b) SPANK API implementation in Slurm

Figure 18: SPANK API implementation differences

6 Summary

During the ten months since Work Package 2 started, the main achievement was to reach a common understanding of this work package's goals, completed with a global picture of the IO-SEA project. We held several discussions within our work package and with the other work packages in order to understand what we could offer them, what they expected from Work Package 2 and what we could expect from them.

This effort, which includes a small "Proof of Concept", translated into the architecture described in this document. This architecture is not final because the other technical work packages are also working in their architectures, things are still moving, and our current understanding and decisions will probably have to adapt to those changes. At the same time, as our own understanding improves and evolves, some decisions may be reviewed. Nevertheless, only small changes and corrections are expected. The foundations of our architecture will remain unchanged.

The work done during these months helped us identify several ideas. We determined which ephemeral services to provide, but also that the integration of other ephemeral services should be simple. We also decided that the Yorc orchestrator will be used to manage the data nodes and ephemeral services, but that it should be easily replaceable in case we need to use another orchestrator. The integration with Slurm and OpenStack should be generic enough to allow other tools to manage the workflows. In other words, the solution should be flexible enough to be independent of and particular technology.

The next step is to start developing all these concepts. We will first use just a few ephemeral services with no interaction with the HSM (Work Package 4) to get everything just working and validate the proposed ideas. In a second step, we will use all the ephemeral services, with data moving across the different storage tiers, to reach to the final solution.

Glossary

Symbols

ParaStation Management Process management facility of ParaStation Modulo.

A

ADDB Motr's Analysis and Diagnostics Data Base.

API An Application Programming Interface is a connection between computer programs.

Atos Atos is Europe's largest digital services deliverer.

B

BeeGFS BeeGFS is a hardware-independent POSIX parallel file system.

burst buffer A fast intermediate storage layer positioned between the front-end computing processes and the back-end storage systems. It bridges the performance gap between the processing speed of the compute nodes and the Input/output (I/O) bandwidth of the storage systems.

C

CEA The French Alternative Energies and Atomic Energy Commission.

CORTX A Motr extension handling the access rights and providing and S3 interface.

CPU Central Processing Unit.

D

DASI Data Access and Storage Interface developed in Work Package 5.

data node Data nodes are cluster nodes dedicated to providing I/O services. They are equipped with multiple NVMe and/or NVRAM devices used for fast, local storage.

dataset A set of data logically related that is usually produced, stored and used together. For instances, data produced during a given campaign.

DIMM DIMM or dual in-line memory module, is commonly called a RAM stick.

DIX Motr's Distributed IndeX and distributed index module.

Docker	Docker is a set of platform as a service products that use OS-level virtualization to deliver software in packages called containers.
Docker Compose	Compose is a tool for defining and running multi-container Docker applications.
F	
FID	File IDentifier.
fsck	A UNIX command to check a file system.
G	
GbE	Gigabit Ethernet: an ethernet network interface capable of transfer rates in the gigabit range.
GBF	The Global Bunch of Flash is a temporary parallel filesystem management product included in the ATOS Flash Accelerator suite.
GNSS	A Global Navigation Satellite System such as Europe's Galileo or United States of America's Global Positioning System.
GPFS	The General Parallel File System is a high-performance clustered file system software developed by IBM.
GPU	Graphics Processing Unit.
H	
HDD	Hard Disk Drive.
HSM	Hierarchical Storage Management.
HTTP	HyperText Transfer Protocol: common Internet protocol to access web pages or download files.
I	
I/O	Input/Output.
InfiniBand	InfiniBand (IB) is a computer networking communications standard used in high-performance computing that features very high throughput and very low latency.
K	
KV store	Key-Value store. A store that keeps values addressing them with a key instead of an address.
L	

layout	In Motr a layout is a map determining where file data and meta-data are located. The layout is by itself a piece of meta-data.
LD_PRELOAD	A special environment variable that, when set to point a shared lybrary, will instruct the dynamic linker to bind symbols from that library before any other library.
Linux	Linux is a free and open-source, monolithic, modular, multitasking, Unix-like operating system kernel widely used in High Performance Computing clusters.
LQCD	Lattice quantum-chromodynamics is a numerical framework for calculating physical properties of hadrons, composite particles composed of quarks.
Lustre	The Lustre®file system is an open-source, parallel file system that supports many requirements of leadership class HPC simulation environments.
LV	A Logical Volume is a part of a VG that can be exposed as a block device, on top of which a filesystem can be created.
LVM	The Logical Volume Manager is a device mapper framework that provides logical volume management for the Linux kernel.
M	
Motr	Seagate's object store.
MSA	Modular Supercomputing Architecture.
N	
namespace	How the stored information will be presented to an application. For a POSIX namespace, it will be the directory tree and files; for an S3 namespace, the alternative object identifiers.
NFS	Network File System is a distributed file system protocol allowing a user on a client computer to access files over a computer network much like local storage is accessed.
NVDIMM	A DIMM of NVRAM.
NVMe	Non-Volatile Memory Express.
NVMe-oF	NVM Express over Fabrics.
NVRAM	Non-Volatile Random Access Memory.
O	

object-store	A data storage that manages data as objects, as opposed to other storage architectures like file systems which manages data as a file hierarchy.
OID	Object IDentifier.
OpenStack	A free, open standard cloud computing platform mostly deployed as infrastructure-as-a-service (IaaS) in both public and private clouds.
P	
ParaStation Healthchecker	Tool running a configurable set of tests on a cluster node to check the node's health status.
ParaStation Modulo	HPC middleware and management stack for Modular Supercomputing.
PCM	Phase-Change Memory, Non-volatile memory technology.
pNFS	Parallel NFS.
POC	Proof Of Concept.
POSIX	Portable Operating System Interface (POSIX) is a family of standards specified by the IEEE Computer Society for maintaining compatibility between operating systems.
psid	The daemons forming the distributed management network of ParaStation Management.
psidforwarder	The compute node process managing a single client process in ParaStation Management.
psslurm	ParaStation Management plugin for Slurm support.
psslurmstep forwarder	The compute node process managing a step in ParaStation Management.
R	
RAID	Redundant Array of Independent Disks.
RDP	Highly-scalable Reliable Datagram Protocol.
REST API	An API providing remote services over HTTP.
S	
S3	Amazon's Simple Storage Service: HTTP-based protocol to access data. Initially developed by Amazon, its generalisation made it a de facto standard for data access in cloud services.
SBB	The Smart Burst Buffer is a burst buffer product included in the ATOS Flash Accelerator suite.

SBF	The Smart Bunch of Flash is a NVMe-oF management product included in the ATOS Flash Accelerator suite.
Seagate	Seagate is Europe's largest storage provider.
Slurm	Slurm is an open-source cluster management and job scheduling system.
slurmctld	The central management daemon of Slurm.
slurmd	The compute node daemon of Slurm.
slurmstepd	The compute node process managing a step in Slurm.
SPANK	Slurm Plug-in Architecture for Node and job (K)control.
SSD	Solid State Drive.
step	A step is an application running in a computing module. Concatenated steps form a workflow.
STT-MRAM	Spin-Transfer Torque Magnetic Random Access Memory (STT-MRAM) has been chosen by the industry as the non-volatile memory technology of choice to replace Embedded Flash at advanced technology nodes.
U	
URL	A Uniform Resource Locator, colloquially termed a web address, is a reference to a web resource that specifies its location on a computer network and a mechanism for retrieving it.
V	
VG	A Volume Group is a set of physical devices that will be managed by the LVM.
W	
workflow	A concatenation of steps processing data together in order to make a particular computation.
X	
XFS	XFS is an open-source, high-performance 64-bit journaling file system created by Silicon Graphics, Inc.
Y	
YAML	YAML stands for "YAML Ain't Markup Language" and is a human-friendly data serialization language for all programming languages.
Yorc	Yorc stands for "Ystia Orchestrator" and is an hybrid cloud/HPC TOSCA orchestrator.

Bibliography

[1] Seagate. Motr/CORTX. https://github.com/Seagate/cortx-motr, 2021.

[2] NVMe. https://nvmexpress.org.

[3] NV-RAM. https://en.wikipedia.org/wiki/Non-volatile_random-access_memory.

[4] Flash Accelerators. https://atos.net/en/2019/product-news_2019_02_07/atos-boosts-hpc-application-efficiency-new-flash-accelerator-solution.

[5] Yorc. https://github.com/ystia/yorc.

[6] Yaml. https://yaml.org.

[7] STT-MRAM. https://www.mram-info.com/stt-mram.

[8] Memristor. https://www.sciencedirect.com/science/article/pii/B9780857098030500114.

[9] Pcm. https://iopscience.iop.org/article/10.1088/1361-6463/ab7794.

[10] 3dxpoint. https://pcper.com/2017/06/how-3d-xpoint-phase-change-memory-works.

[11] Pmem. https://pmem.io.

[12] Percipient storage for exascale data centric computing 2. https://cordis.europa.eu/project/id/800999.

[13] Alberto Scionti, Jan Martinovic, Olivier Terzo, Etienne Walter, Marc Levrier, Stephan Hachinger, Donato Magarielli, Thierry Goubier, Stephane Louise, Antonio Parodi, et al. Hpc, cloud and big-data convergent architectures: The lexis approach. In *Conference on Complex, Intelligent, and Software Intensive Systems*, pages 200–212. Springer, 2019.

[14] E. B. Gregory, P. Couvée, and M. Golasowski. IO-SEA D1.1 Application and co-design. Technical report, IO-Software for Exascale Architectures, 2021.

[15] Sage2 project's home page. https://sagestorage.eu.

[16] Seagate. CORTX. https://github.com/Seagate/cortx, 2021.