H2020-JTI-EuroHPC-2019-1

REGALE: An open architecture to equip next generation HPC applications with exascale capabilities

Grant Agreement Number: 956560

D3.3

REGALE prototype v3.0

Version: 2

Author(s): Yiannis Georgiou (RYAX)

Contributor(s): Bruno Raffin (UGA), Federico Tesser (CINECA), Olivier Richard (UGA), Lluis Alonso (BSC), Julita Corbalan (BSC)

Date: 30.04.2024

Project and Deliverable Information Sheet

REGALE Project	**Project Ref. №:** 956560	
	Project Title: REGALE	
	Project Web Site: https://regale-project.eu	
	Deliverable ID: **D3.3**	
	Deliverable Nature: Other	
	Dissemination Level: PU	**Contractual Date of Delivery**: 30/3/2024
		Actual Date of Delivery: 1/5/2024
	EC Project Officer: Matteo Mascagni	

* - The dissemination levels are indicated as follows: PU = Public, fully open, e.g. web; CO = Confidential, restricted under conditions set out in Model Grant Agreement; CI = Classified, information as referred to in Commission Decision 2001/844/EC.

Document Control Sheet

Document	**Title:** REGALE prototype v3.0	
	ID: D3.3	
	Version: 2	**Status:** Final version
	Available at: https://regale-project.eu	
	Software Tool: Google Docs	
	File(s): REGALE_D3.3_v2.docx	
Authorship	**Written by:**	Yiannis Georgiou (RYAX)
	Contributors:	Bruno Raffin (UGA) Federico Tesser (CINECA) Olivier Richard (UGA) Lluis Alonso (BSC) Julita Corbalan (BSC)
	Reviewed by:	Georgios Goumas (ICCS), Eishi Arima (TUM), Michael Ott (BADW-LRZ)
	Approved by:	Georgios Goumas (ICCS)

Document Status Sheet

Version	Date	Status	Comments
0.1	06.02.2024	Draft	Initial version
1	29.04.2024	Pre-final	Pre-final version
2	30.04.2024	Final	Final version

Document Keywords

Keywords:	REGALE, HPC, Exascale, HPC PowerStack, Power Management

Table of Contents

1. Executive Summary

This document presents the final version of WP3: REGALE prototype. The primary objective of REGALE is to integrate tools and create a European software stack for power and workflow management in next-generation supercomputers. WP3 is responsible for providing three deliverables that contain the outcomes of specific integration tasks. By utilising the tools developed by REGALE partners we have identified various integration scenarios that address the requirements already outlined in WP1. This deliverable supplements and completes the different integrations described in WP3-D3.1/D3.2 and builds upon the previously defined prototype implementation. This deliverable focuses on three main work topics: i) component-to-component integration covering aspects that have not been tackled in previous deliverables, ii) extensions of integration scenarios, iii) perspectives for workflow and powerstack paths integration and iv) the development of the REGALE library. In the component-to-component integration we describe the progress done since the last WP3 deliverable (D3.2) and in the chapter related to the REGALE library we describe the overall structure and organisation of this communication framework that we develop to facilitate the tool integration.

List of Abbreviations and Acronyms

Abbreviation / Acronym	Meaning
AAPC	Application-Aware Power Capping
API	Application programming interface
BEO	Bull Energy Optimizer
BMC	Baseboard Management Controller
CQL	Cassandra Query Language
CQLSH	Cassandra Query Language Shell
DB	Database
DCPS	Data-Centric Publish Subscribe
DDS	Data Distribution Service
EARD	EAR Daemon
EARL	EAR Library
EXAMON	Exascale Monitoring
FSPC	Fair Sharing Power Capping
SW	Software
HW	Hardware
IDL	Interactive Data Language
IoT	Internet of Things
IPMI	Intelligent Platform Management Interface
IS	Integration Scenario
M2M	Machine-to-Machine
MQTT	Message Queuing Telemetry Transport
MPI	Message Passing Interface
NM	Node Manager
NoSQL	No Structured Query Language
OMG	Object Management Group
PR	Pull Request
PPE	Power and Performance Estimator
QoS	Quality and Service
RAPL	Running Average Power Limit
REST	REpresentational State Transfer

ROS	Robot Operating System
RTPS	Real-Time Publish-Subscribe
RJMS	Resource and Job Management System
TTS	Time To Solution
XML	eXtensible Markup Language
JM	Job Manager
JSON	JavaScript Object Notation

2. Introduction

The REGALE project follows two main paths (i) the PowerStack path that focuses on the power-efficient operation of modern supercomputers, and (ii) the Workflow Engine path that focuses more on the execution of complex, workflow-based applications on modern supercomputers. This deliverable tracks the updates being done in both paths while presenting some specific works done. Furthermore, the deliverable discusses particular component-to-component integration, details further different integration scenarios and explains particular REGALE library extensions. The remainder of the deliverable tackles the Ryax-BeBiDa-OAR-SLURM integration in section 3 providing different techniques to facilitate and optimize the design and deployment of workflows executing on HPC resources; Section 4 proposes specific integration scenarios; Section 5 combines the workflow engine and powerstack paths integrations along with perspectives while section 6 brings the latest version of the REGALE library facilitating the usage of the different REGALE tools.

3. Component-to-component Integration Updates in Workflow Engine Path RYAX-BEBIDA-OAR-SLURM

This section provides updates upon the latest integrations among specific REGALE components that were finalised during the last period of the project. Besides the integration of the different REGALE components presented in D3.2, this section presents the latest integration added in the list among the REGALE components RYAX - BEBIDA - OAR - SLURM.

Overview

This integration brings together the “Ryax path”, driven by the workflow engine Ryax, adapted for HPC offloading using SLURM as described in deliverables D1.3 and D4.3; along with the “Elastic Resource Management”, driven by the system manager BeBiDa bringing elasticity for Big Data workloads on the traditionally rigid RJMS: OAR and SLURM as described in deliverables D2.3 and D1.4. In particular it enables the combination of various REGALE components in a way to address the programmability and elasticity of new types of workloads which on one side they have more dynamic behaviours using more Cloud-based frameworks and on the other side are composed by multiple stages needing different types of resources and are managed by non-HPC experts. The tight integration of Ryax with OAR and SLURM allows users to leverage the advantages of using an easy to use, intuitive web user interface to design and deploy complex computational workflows and execute them on different Cloud and HPC resources, through HPC offloading, while abstracting the complexity of the HPC infrastructures and HPC tools usage. In parallel, the tight integration of BeBiDA with OAR and SLURM brings elasticity in the resource management of typical HPC control through specific non-HPC intrusive techniques.

The integration of all the above components goes beyond the current state of the art since it combines programmability and elasticity, bringing the advantages of Cloud Computing on HPC without sacrificing the key HPC values: performance, robustness and scalability.

In more detail the integration of the aforementioned REGALE components RYAX, BEBIDA, OAR and SLURM also involves the **Kubernetes** orchestrator as the resource manager in control for the Big Data, Cloud part of the infrastructure upon which Ryax resides.

In addition, it involves the **Singularity** containerization service, which is used to package the environment to be deployed on the HPC resources, when offloading the computation to HPC resources through the BeBiDa tool taking advantage of the integration with OAR and Slurm.

Furthermore, it makes use of **Spark** as the Big Data programming framework and runtime with which the different dynamic use cases have been developed and needed for their execution.

Figure 1 shows the high-level view of the architecture of the particular integration featuring the usage of the Ryax workflow engine for the design and high-level control of the computational workflows (center of the figure). These may be composed of various parts which are deployed by default on the underlying Kubernetes resources (bottom of the figure). In case of compute-intensive Spark actions, these can be either executed on the Kubernetes resources (bottom of the figure) or may be offloaded to HPC using BeBiDa and techniques using combination of Kubernetes, SLURM, OAR and, of course, the runtime of Spark (right part of the figure).

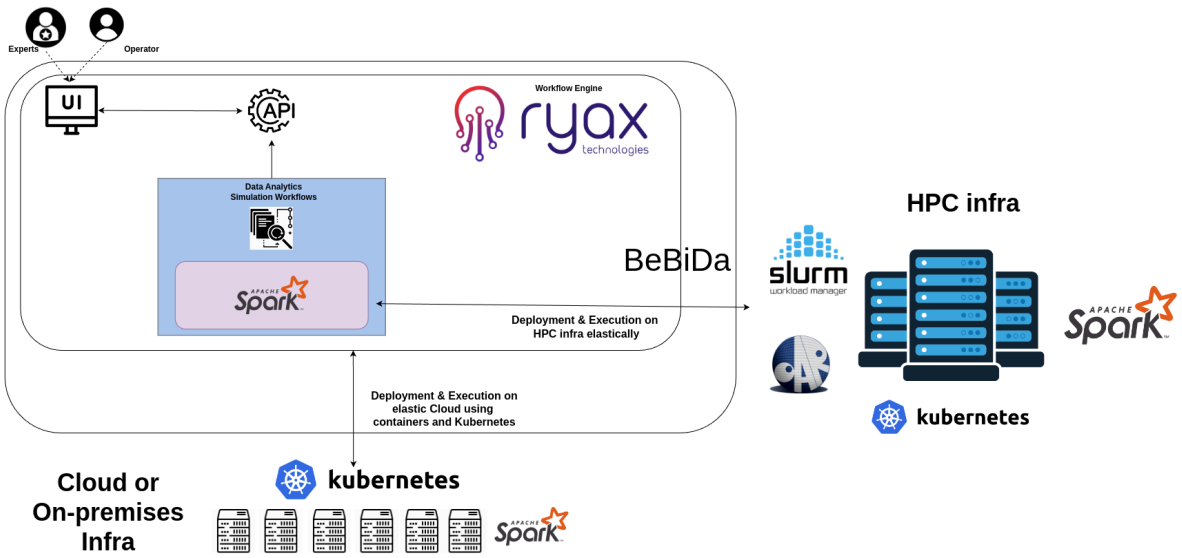


Figure 1: *Overview of the architecture of the RYAX-BEBIDA-OAR-SLURM components integration bringing programmability and elasticity in REGALE*

Integration internals: Ryax platform

The Ryax platform developed by Ryax Technologies is an open-source low-code, API-first, workflow management system. It provides the means to create, deploy, update, execute and

monitor the execution of data processing workflows on hybrid Cloud and HPC computing infrastructures. It enables users to create their data automations and expose them with APIs through fully customizable workflows. It makes use of a powerful, hybrid serverless/microserverices-based runtime, abstracting completely the complexity of building and deploying containers with their dependencies on Cloud and HPC infrastructures. The software platform has been designed with a focus on data analytics and offers a variety of built-in features and a repository of various integrations to facilitate users in the process of creating complex computational workflows.

The Ryax platform abstracts the complexity of: i) developing data analytics pipelines by using workflows requiring developers only to describe the structure of what will be deployed, and how it will be connected through a simple to use, intuitive web interface and visual programming tools connected to a YAML- based declarative design; ii) building the environments to be deployed with the necessary dependencies through built-in internal mechanisms based on Nix functional package manager, iii) deploying and monitoring containers through a fine-integration with the Kubernetes orchestrator and iv) providing efficient autoscaling capabilities based on built-in resource management techniques considering the request demands and combining both Horizontal Pod Autoscaling (Kubernetes) and Cluster Node Autoscaling (Cloud Infrastructure provider).

Ryax, through its design and abstractions offers a unified handling of batched and streaming data definitions to cover processing of both bounded and unbounded datasets, which makes it an attractive engine for hybrid Big Data/HPC applications.

In the context of the REGALE project the **Ryax workflow engine** has been enhanced with the **HPC offloading capabilities** which allows non-HPC experts to improve the programmability and flexibility of distributed data analytics (AI, Big Data) applications to be executed on HPC infrastructures and enabling the bridge between common Cloud usage and traditional HPC executions. This is done by abstracting the complexity of deploying an application on HPC infrastructures and simplifying the way the interactions of the user with the complex HPC systems is done.

As a reminder, the principal concepts of Ryax engine are:

- **An action,** which is an independent task representing a separate building block of a broader data processing application. It may have inputs, outputs and a specific code that manipulates the inputs with some logic to produce the outputs. There are **four types of actions** in Ryax: trigger, processor, stream operator and publisher. For more details the reader may consult Regale deliverable 1.1.
- **The workflow,** which is a complete data processing application composed of actions linked together in the form of a directed acyclic graph (DAG). The intermediate links are data streams. Each action uses some data from the input stream and adds its output data to the stream. In this way, the data that an action outputs, is accessible

to every downstream action. In other words, any action has access to the data of upstream actions.

The internal architecture of Ryax is composed of different components deployed as microservices and managed by the Kubernetes orchestrator to guarantee interoperability, flexibility, auto-healing, easy upgrade/downgrade and fault-tolerance of the deployed components' microservices. In particular these are composed by the **Ryax client services**: WebUI, ADM and CLI; the **Ryax internal services**: Authorization, Repository, Studio, Action Builder, Runner and Action Wrapper; and the Ryax external services: Datastore, Filestore, Broker, Registry.
The new functionalities which have been developed in the context of Ryax HPC offloading for the REGALE project to facilitate dynamic Big Data/Spark based use cases are the following: i) the preparation of code to be run on HPC (the execution in the Cloud is usually done using Docker whereas in HPC using Singularity a.k.a Apptainer), ii) the automated creation of HPC Slurm (or OAR) resource manager script based on a web interface parameter selection process, iii) the stage-in and stage-out of data to and from the HPC system, related to the execution, iv) the real-time logging of the execution on the HPC system directly on the Ryax web interface, v) the seamless connection of the results of an HPC execution to other actions of the workflow which may be executed on the Cloud or on the same or different HPC system, vi)the dynamic execution of Big Data streaming applications based on Spark framework through fine integration with BeBiDa system management tool and some new enhancements which allow optimal seamless execution of Spark applications on the rigid HPC infrastructures, while taking advantage of the high-level workflow view of Ryax engine.

The code of the Ryax workflow engine has been changed from proprietary to open-source since the beginning of REGALE. Furthermore, the REGALE related enhancements for HPC offloading have been initially introduced as a prototype and after passing the necessary testing and validation they have recently been released in the upstream production version of the software.

Ryax HPC offloading and integration to SLURM & OAR

As mentioned before, one of the new features brought in the Ryax platform, in the context of REGALE, is the enhancement of offloading computation to HPC clusters. For this to take place It requires an SSH access to the frontend node and a shared home directory across nodes to be able to manage the IO and capture logs.

Ryax is leveraging Singularity (a.k.a Apptainer) to package and run containers on the HPC cluster. To enable HPC offloading for a Ryax Action, a particular parameter named "hpc addon" has been created which needs to be added on the ryax_metadata.yaml file as shown below:

```
apiVersion: "ryax.tech/v2.0"
kind: Processor
```

```
spec:
  # ...
  addons:
    hpc: {}
  # ...
```

This parameter added within the metadata of the action will notify the Ryax builder that the particular action has to be built as a Singularity image. Hence the NIX based builder is equipped with the necessary libraries to build the Singularity image to be used in the context of HPC offloading, instead of the typical one which is Docker based and is used for the Kubernetes based executions. Once the Singularity image is built then it can be used as a typical action within Ryax workflows. In the context of a Ryax workflow it's particularity is that in the workflow design and configuration phase there are specific HPC related parameters that are activated and need to be configured in order to allow the user to specify which HPC cluster they are using along with the job characteristics related to the HPC resource manager. Parameters such as the IP address of the frontend node of the HPC cluster and the SLURM Sbatch (or OAR batch) script along with the jobs' requirements have to be specified. An example of some of the parameters can be seen in the screenshot presented in figure 2 which shows the Ryax action at the left (as part of a workflow) along with the different parameters needed to be filled-up at the right. Among others it features parameters such as the sbatch script , the username , the ssh private key added in a hidden mode as password and the HPC frontal node of the cluster used for the HPC offloading.

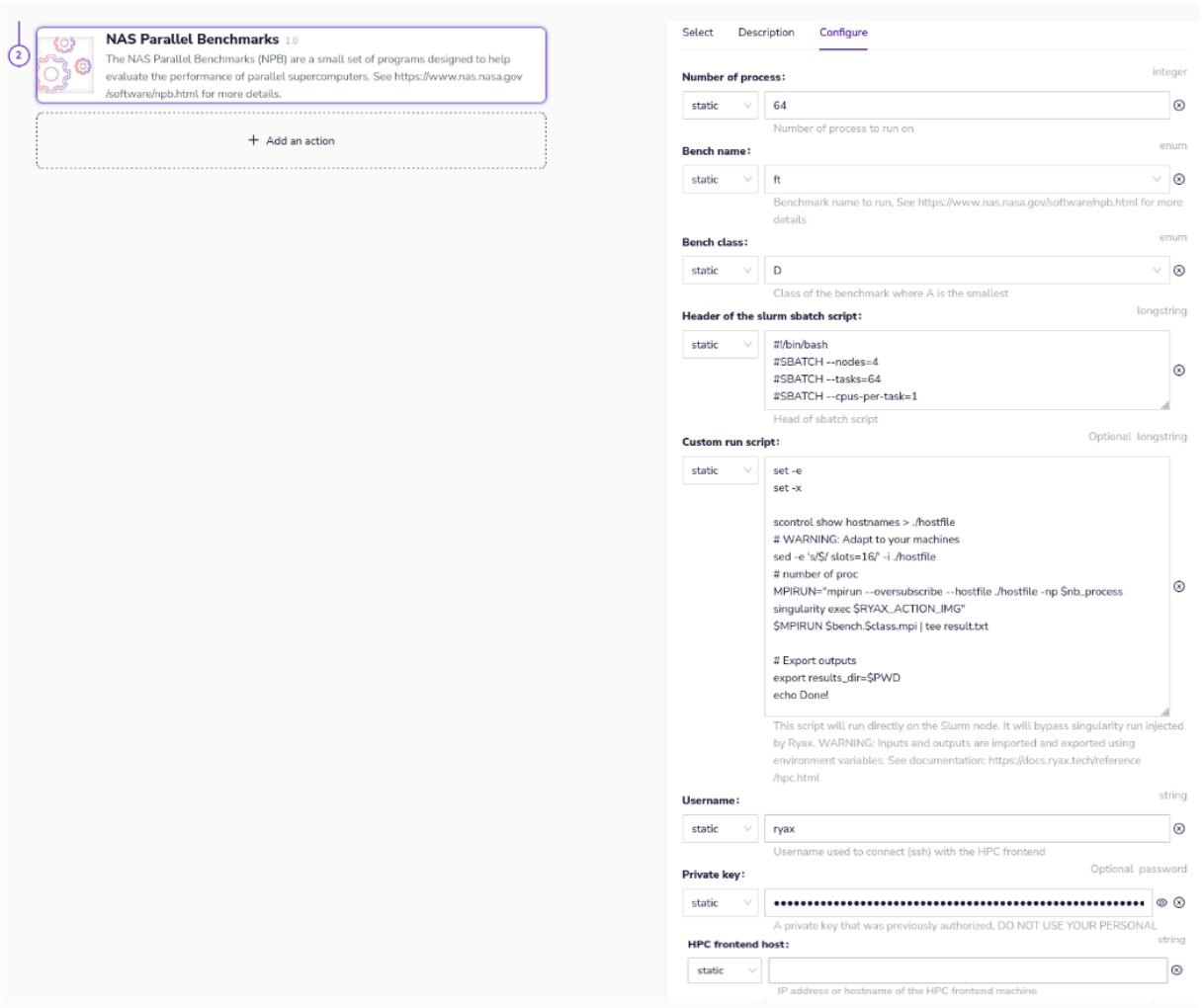


Figure 2: *Ryax WebUI parameters configuration view for an action enabling HPC offloading to an external Slurm based HPC cluster*

Based on these parameters, once the configuration of the action and workflow is done and the workflow is deployed then HPC offloading enables the particular action to open an ssh connection to the HPC frontend node, in the name of the user; then the singularity image of the action will be transferred transparently to the home directory of the user and once the transfer is done the particular sbatch will be launched in the name of the user.

A particular variation of this technique enabled through the “addons hpc” parameter is to make use of the headless service which will launch a custom HPC script with sbatch (or oarsub), directly on the cluster, without building a singularity image. This means that the user will have to prepare everything - the singularity image or the adapted environment along the executable - in advance on the HPC cluster.

Thus, Singularity needs to be installed on the nodes of the cluster and Python3 must be present on the frontend because it is required to wrap actions when using the custom_script which runs directly on the node.

Ryax will create a .ryax_cache folder in the home directory of the user for whom the execution is automated. This will contain the container images, the IO, and the working directory of each execution.

The actions will run just like regular Ryax actions but instead of being executed by Kubernetes on the underlying on-premise or Cloud resources they will be offloaded to HPC using Slurm (or OAR) and deployed as Singularity containers inside the HPC cluster. Slurm runs the containers in parallel following the parameters of the sbatch header (typically setting number of nodes and number of tasks). As usual, Ryax will provide inputs and retrieve outputs upon execution completion.

Furthermore, what is interesting for the HPC offloading is that the stage-in/out of data along with the control of inputs and outputs of the offloaded action and the connection to the previous or following actions of the workflow is done transparently to the user. By default the internal object storage of Ryax - based on minio - is used to pass the data from one action to the other. In the case of the offloaded action, Ryax will make use of the HPC file system (i.e NFS, Lustre, etc) and the .ryax_cache folder which is created to enable all the necessary IO for HPC offloading. Hence, besides the exchanges performed between Ryax actions run on Kubernetes and the HPC offloaded actions where data are transferred from minio to the HPC file system for stage-in and vice-versa for stage-out; when the following action of the workflow is offloaded to HPC, then the data are stored on the HPC file-system under a specific folder under .ryax_cache.

Finally, we have also integrated the logging/debugging mode of Ryax workflows and actions to function homogeneously for all Ryax actions including the ones executed through HPC offloading on external HPC clusters. This is done by getting the logs of the sbatch (or OAR) batch scripts periodically and displaying them under the executions' debugging view of workflow's actions on Ryax. An example of an HPC offloaded action logging/debugging view based on Regale pilot 3 is shown on figure 3.

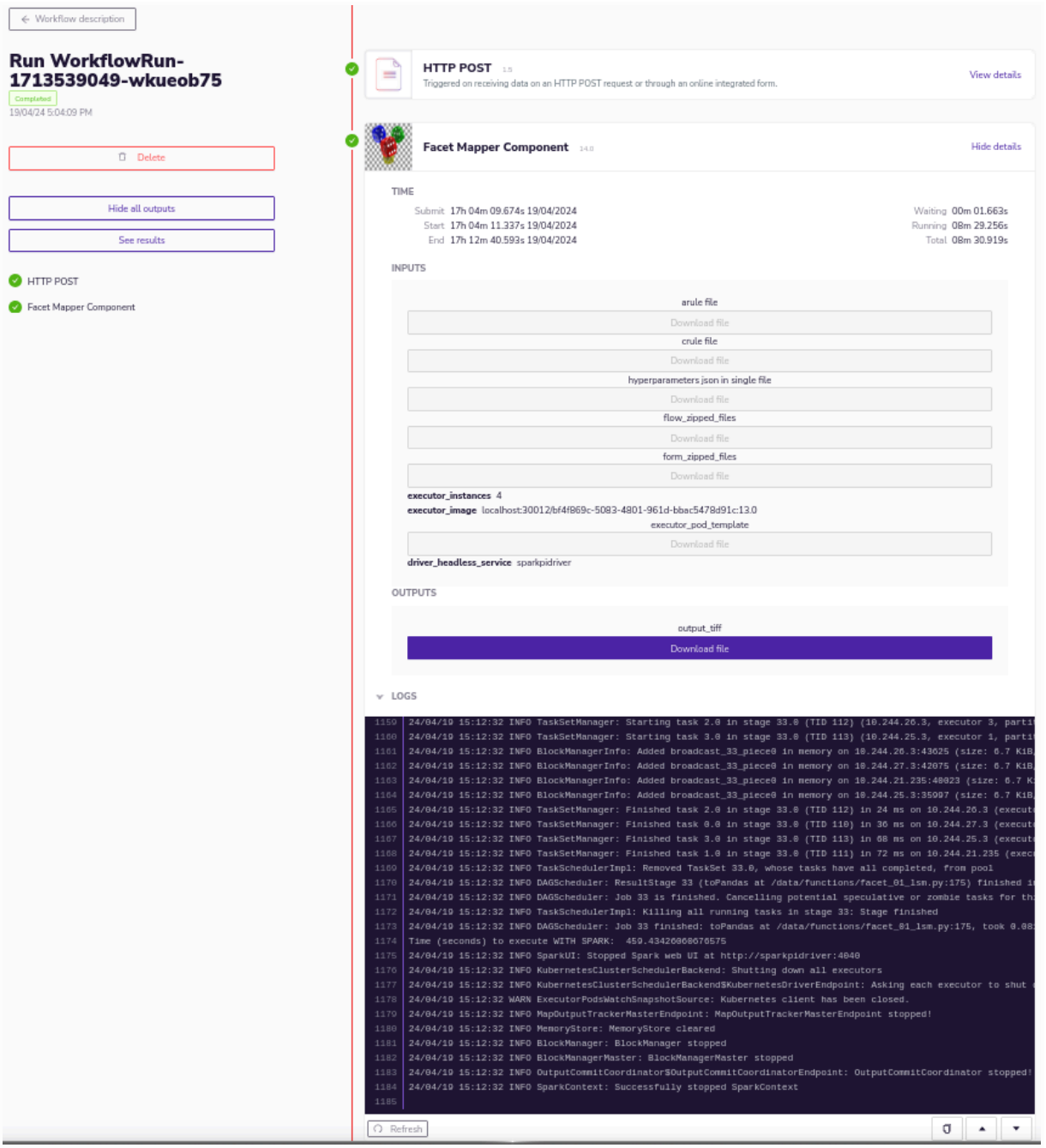


Figure 3: *Ryax WebUI workflow real-time logging/debugging view for an action being executed through HPC offloading to an external Slurm based HPC cluster*

BeBiDa integration to SLURM & OAR for elastic execution of Spark applications

BeBiDa is a system management component which allows the execution of Big Data frameworks such as Spark upon rigid HPC clusters. In particular, BeBiDa will leverage the advantages of the Cloud-based resource manager Kubernetes without sacrificing the performance capabilities of HPC resource managers.

BeBiDa has been adapted to function using Kubernetes as the main Big Data resource manager and supports both OAR and Slurm for the execution on HPC through their prolog and epilog scripts mechanisms. Spark applications are being executed through Kubernetes

executions which means that Spark executors are spawned dynamically as Kubernetes pods upon the unutilized HPC resources.

The motivation, research dimensions and details of the particular integration is provided in detail under the Regale deliverable D2.3 where we explain the interest and implementation aspects of elastic resource management on HPC clusters.

Ryax integration to BeBiDa for elastic execution of Spark applications on SLURM & OAR HPC clusters

This integration combines all the four Regale components (Ryax-BeBiDa-OAR/Slurm) and allows users to elastically execute Spark applications on HPC clusters.

On the one hand we have the Ryax platform deployed upon a Kubernetes cluster outside of the HPC cluster. This can be either on the Cloud or on some on-premise VMs in the same data center with the HPC cluster. On the other hand we have the typical HPC cluster managed by Slurm (or OAR). One of the requirements is that Ryax and in more detail the Kubernetes cluster, through which the workflows are launched and offloaded to HPC, will need to somehow have a network connection (ssh access) with a login node to the HPC cluster managed by the Slurm (or OAR) resource manager in order to enable the submission of batch jobs.

Furthermore, BeBiDa is configured to enable the dynamic control of Big Data jobs submitted on Kubernetes enabling the usage of unutilized HPC resources by putting in place the prolog/epilog mechanisms of SLURM (or OAR) as mentioned in the previous section and described in detail in D2.3.

The remaining piece of the puzzle is the integration of the Ryax platform with the BeBiDA tool which can enable the support of Big Data Spark jobs to be executed as typical Ryax actions on the underlying resources managed by Kubernetes. BeBiDa can dynamically bring more Kubernetes resources (non-utilized HPC resources) in the pool of available resources for Ryax to use directly for the different workflows. Hence, BeBiDa can enable the usage of HPC resources in the context of Big Data Spark applications without making use of the standardjob submission through the traditional HPC resource managers, but rather exploiting the capabilities of Ryax and its tight integration to Kubernetes orchestrator.

In more detail, the integration of Ryax with BeBiDa is done by enabling a tighter integration of Ryax to Big Data Spark applications. This is done by enabling the specific Ryax action which encapsulates the Spark application to launch a spark submit command, acting in cluster mode, which will start a spark driver deployed as a Kubernetes pod. Then based on the tight integration of Kubernetes and Spark[1] we enable the spawning of multiple spark executors across Kubernetes.

To enable this Spark driver pod to have the ability to spawn new pods on kubernetes dynamically, we have to make use of particular Kubernetes resources such as

[1] https://spark.apache.org/docs/latest/running-on-kubernetes.html

service-accounts, labels and annotations. For this reason we have created a new addon in Ryax, named “kubernetes addon” which enables ryax actions to specify: service_account_name, labels, and annotations to be used in the context of a Ryax action execution. This new addon allows Ryax to enable a tight integration for Spark applications but is generic enough for other types of executions that need similar Kubernetes resources specifications.

The kubernetes addon requires to be associated with a serviceAccountName that can list/delete/create on services, pods, and configmaps kubernetes' resources. This will grant kubernetes access required for spark to spawn the workers across desired kubernetes nodes. The serviceAccount must be created beforehand in namespace ryaxns-execs with a serviceAccountName that is later used with the kubernetes addon.

The required service account to be used by Spark and BeBiDa can be created by applying the following yaml:

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: spark
  namespace: ryaxns-execs
---
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  namespace: ryaxns-execs
  name: spark-role
rules:
- apiGroups: [""]
  resources: ["configmaps"]
  verbs: ["*"]
- apiGroups: [""]
  resources: ["pods"]
  verbs: ["*"]
- apiGroups: [""]
  resources: ["services"]
  verbs: ["*"]
---
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: spark-role-binding
  namespace: ryaxns-execs
subjects:
```

```
- kind: ServiceAccount
  name: spark
  namespace: ryaxns-execs
roleRef:
  kind: Role
  name: spark-role
  apiGroup: rbac.authorization.k8s.io
```

For the adapted Kubernetes nodes to be used by Spark, a headless service is required so that executor pods can connect to a valid service on the network. This is performed by two steps, first we need to create the headless service and then associate it with specific nodes by setting a selector. This service can be created through the following yaml.

```
apiVersion: v1
kind: Service
metadata:
  name: sparkpidriver
  namespace: ryaxns-execs
spec:
  clusterIP: None
  selector:
      run: sparkpi
  ports:
    - protocol: TCP
      port: 4041
      targetPort: 4041
```

Then on the action, featuring the kubernetes addon parameters, we need to specify the label run=sparkpi to match the predefined selector.

Finally, in this context and once the Kubernetes cluster is ready to support the transparent execution of Spark applications, BeBiDa will allow the usage of unutilized HPC resources by dynamically managing the addition of new Kubernetes nodes (the ones not being utilized by HPC jobs). In case we want to have Ryax enabled Spark jobs to only use HPC resources brought by BeBiDa then we can make BeBiDa aware by selecting the nodes where the spark executors will run. To select the specific nodes where to schedule, we need to provide specific toleration parameters such as the one used in the following template pod:

```
apiVersion: v1
kind: Pod
metadata:
```

```
  labels:
    ryax: spark-executor
spec:
# The pod template starts here
  containers:
    - command:
      - bash
      - /data/spark-entrypoint.sh
  # Only schedule on bebida hpc nodes
  tolerations:
    - effect: NoSchedule
      key: bebida
      operator: Equal
      value: spark
```

On the side of the nodes each BeBiDa node dynamically added in the pool of unutilized HPC resources becoming available for allocation for the Spark jobs, the following taint needs to be created so that the previously mentioned toleration can use the particular type of nodes.

```
kubectl taint nodes bebida-hpcnode bebida=spark:NoSchedule
```

Once the previous preparations have been effectuated, the Ryax action encapsulating the Spark job can be defined using a ryax_metadata.yaml file featuring the “Kubernetes addon” in order to make use of the specific service-account and label that have been specifically defined, as mentioned previously. An example of a simple spark application calculating pi with multiple executors is expressed as a Ryax action with the following metadata yaml file. It is interesting to see how the new “addon: Kubernetes” is defined and how the specific service-account and labels are expressed in this file to empower the particular Ryax action with the previously mentioned characteristics which give them the needed capabilities to launch Spark on the underlying Kubernetes cluster. Furthermore, we define the necessary dependencies, inputs and outputs, as a typical Ryax action.

```
apiVersion: "ryax.tech/v2.0"
kind: Processor
spec:
  id: pispark.bebida
  human_name: Pi spark bebida example that works every time
  type: python3
  resources:
```

```
    memory: 4G
    cpu: 1
    time: 72h
  description: Pi spark for bebida with multiple executors
  addons:
    kubernetes:
      service_account_name: spark
      labels: run=sparkpi
  dependencies:
  - spark
  - hadoop
  - scala
  - unixtools
  - openssh
  - jdk
  - gnugrep
  - gnused
  - psutils
  inputs:
  - help: Number of iterations
    human_name: n
    name: n
    type: integer
  - help: executor image
    human_name: executor_image
    name: executor_image
    type: string
  - help: executor pod template
    human_name: executor_pod_template
    name: executor_pod_template
    type: file
    optional: true
  outputs:
  - help: String with estimated pi value
    human_name: pi
    name: pi
    type: string
```

Besides the ryax_metadata.yaml file we need a specific ryax_handler.py file which will define the executable to be launched for the particular action. As a typical Ryax action, this file needs to contain a function named handle which will have to manipulate the defined inputs and outputs. A simple -non complete- example of ryax_handler.py for the previously defined pi calculation with Spark metadata file is shown here.

```
...
def handle(ryax_input):
    n = int(ryax_input["n"])
    executor_image = ryax_input.get("executor_image")
    executor_pod_template =
ryax_input.get("executor_pod_template"
...
  return { "pi": result_pi }
```

Based on these files, the environment (Docker image) containing the executable, is built transparently to the user through the Ryax builder respecting the dependencies defined in the metadata file.

Concerning the Spark related code of the action there are different ways to define it in the application. This can be done directly with python code since Spark offers an internal Pyspark library or by enabling the usage of a ".jar" file since the default Spark language is Java/Scala.

The following code section shows an example of Regale pilot 3 implementing its Spark related expressions using Python pyspark as part of the ryax_handler.py

```
  #----------------------SPARK----------------------#
  spark_config_list = [
    ("spark.kubernetes.driver.pod.name", pod_name),
    ("spark.kubernetes.namespace", "ryaxns-execs"),
    ("spark.kubernetes.executor.podTemplateFile",
executor_pod_template),
    ("spark.kubernetes.container.image", executor_image),
    ("spark.kubernetes.executor.request.cores", "1"),
    ("spark.executor.memory", "8g"),
    ("spark.driver.host", driver_headless_service),
    ("spark.driver.port", "4041"),
    ("spark.driver.bindAddress", "0.0.0.0"),
    ("spark.executor.instances", f"{executor_instances}"),
  ]
    spark_conf = SparkConf()

spark_conf.setMaster("k8s://https://kubernetes.default:443")
    spark_conf.setAll(spark_config_list)

    # start a spark session
  spark =
SparkSession.builder.appName("facet").config(conf=spark_conf).get
OrCreate()
```

The following code section shows an example of Regale pilot 4 implementing its Spark related expressions using a certain command to be spawned through “spark-submit” passing different configuration parameters including a jar containing the code of the Spark application, as part of the ryax_handler.py

```
  # submit_bash_cmd = [
   "spark-submit",
   "--master",
   "k8s://https://kubernetes.default:443",
   "--deploy-mode",
   "client",

  # Enable Garbage collection of pods
    "--conf",
    f"spark.kubernetes.driver.pod.name={pod_name}",

    # Use Ryax actions namespace ryaxns-execs
    "--conf",
    "spark.kubernetes.namespace=ryaxns-execs",

  # Use the input specified template or the one hardcoded is case
it is absent
    "--conf",

f"spark.kubernetes.executor.podTemplateFile={executor_pod_templat
e}",
  # Image to run for the executors
    "--conf",

#"spark.kubernetes.container.image=ryaxtech/spark-on-k8s:v0.1.0",
    f"spark.kubernetes.container.image={executor_image}",

  # Amount of cores requested per executor pod
    "--conf",

f"spark.kubernetes.executor.request.cores={spark_executor_cpu}",
   # Amount of memory required per executor pod
    "--conf",
    f"spark.executor.memory={spark_executor_memory}",

 # Use a headless service called sparkdriver to reach driver pod
(current pod)
    "--conf",
    f"spark.driver.host=sparkpidriver",
# Default port and bind parameters
```

```
    "--conf",
    "spark.driver.port=4041",
    "--conf",
    "spark.driver.bindAddress=0.0.0.0",

# Amount of executor pods to run
    "--conf",
    f"spark.executor.instances={executor_instances}",

# Require a service account with read,write,list on ryaxns-execs
# see README.md for details
    "--conf",

"spark.kubernetes.authenticate.driver.serviceAccountName=spark",

## === START TO CUSTOMIZE YOUR APPLICATION HERE========== ##
    "--jars",
    "file:///data/postgresql-42.6.0.jar",

# Call SparkPi application
     "--class",
    "evaluation.Main",

    # Finally the applicaiton with parameters
    f"file://{application_jar}",
    f"{asset}",
    f"{asset_asset_relationship}",
    f"{asset_relationship}",
    f"{vulnerability}",
    f"{vulnerability_v31}",
    f"{indices}",
    f"{output_dir}",
    f"{limit}",
  ]
```

Once the image is built, the action can be used in the context of a Ryax workflow where even a “non-HPC-expert” can simply configure the workflow to be run. Figure 4 shows an example of the Ryax WebUI for pi calculation with Spark featuring the number of iterations needed along with other parameters to be set by the operator. Furthermore, the figure shows how transparent the usage of BeBiDa is, at this stage for the final operator, since the action is run as a typical Ryax action and the only parameter needed to declare the usage of BeBiDa is the definition of the pod template which includes the tolerations to be bound on the relevant

BeBiDa taints as mentioned previously. The pod template is passed as a parameter in the configuration of the workflow as shown in figure 4.

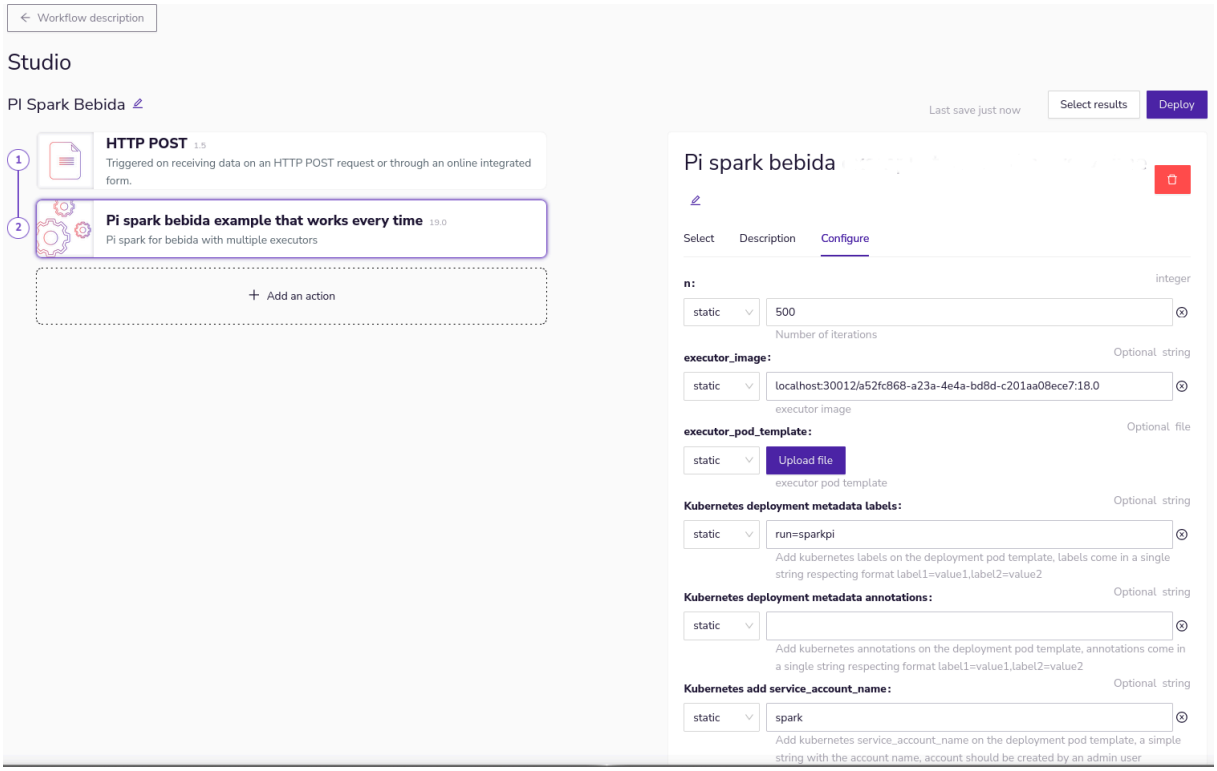


Figure 4: *Ryax WebUI parameters configuration view for an action enabling the execution of a Spark (pi calculation) application to be executed through BeBiDa technique on an external Slurm based HPC cluster*

Once the user triggers the deployment of the workflow, the execution is triggered and Ryax delegates the orchestration of the actions to Kubernetes for execution. The particular action will spawn a Spark driver deployed as a master Kubernetes pod which will then spawn Spark executors as Kubernetes pods deployed on the unutilized HPC resources provided by BeBiDa, respecting the tolerations described previously. In this context, BeBiDa handles the lifecycle of HPC nodes and dynamically adds unutilized HPC nodes to Kubernetes pool of nodes while removing the ones which are needed for typical HPC jobs. The tight integration of Kubernetes and Spark along with the fault-tolerant capabilities of Spark will make sure that during the execution of the Spark job, whatever happens to the Spark executors (for example in BeBiDa context a Spark executor may be killed due to a Kubernetes node being evicted by a higher priority HPC job) , the Spark application and as a result the Ryax workflow will be finalized successfully.

Figure 5 shows a successful termination of a workflow featuring a Ryax action performing the pi calculation through Spark by executing the particular action through the Ryax-BeBiDa integration on an external HPC cluster.

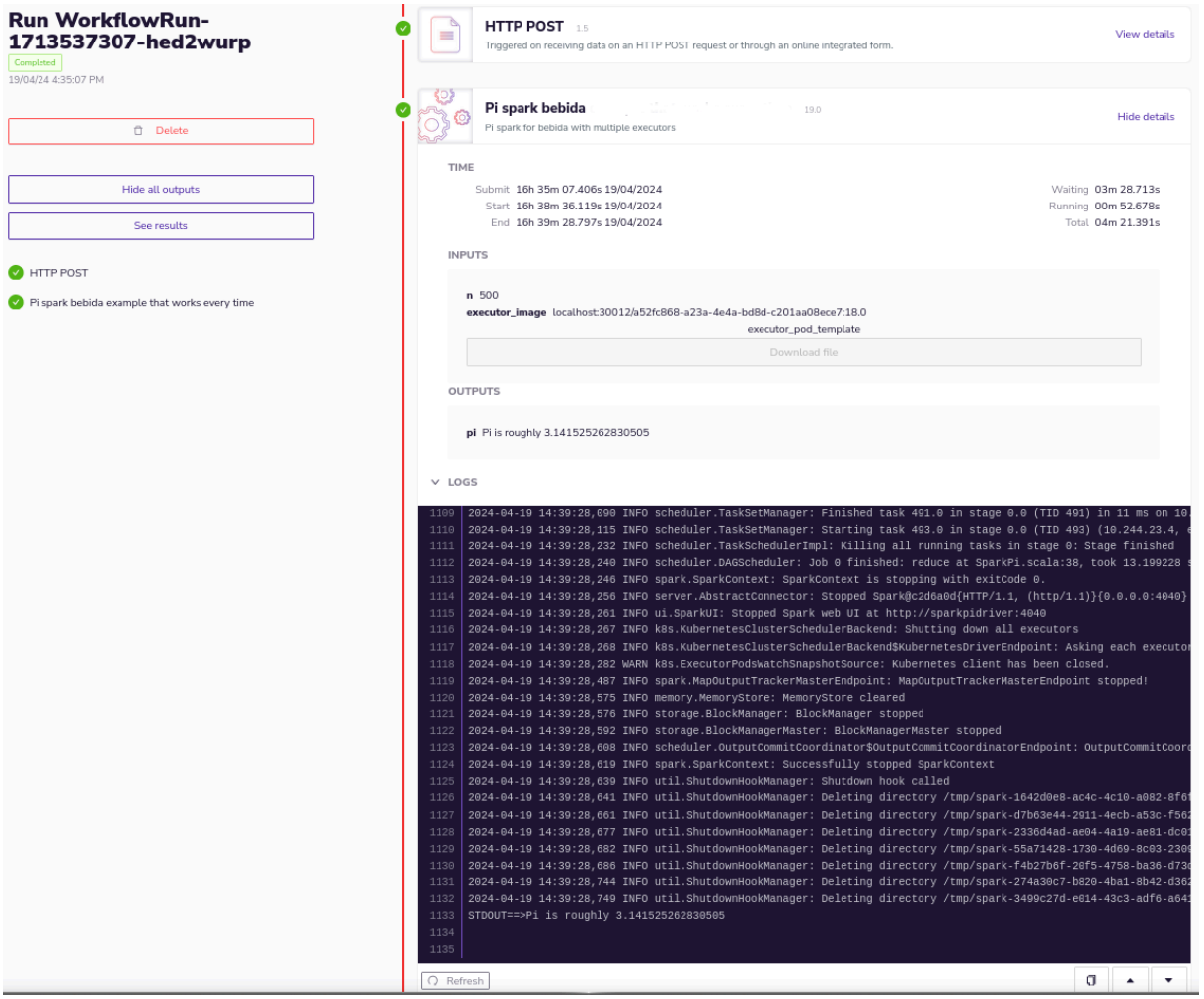


Figure 5: *Ryax WebUI workflow real-time logging/debugging view for an action Spark (pi calculation) application executed through BeBiDa technique on an external Slurm based HPC cluster*

The main difference of Ryax-BeBiDa mechanism from the previously described HPC offloading technique are the following:

- instead of building a Singularity container and establishing the communication with the HPC resource manager, for tasks such as copying the image, performing the stage-in/out of necessary data, submitting jobs, collecting results (through ssh), Ryax builds a typical Ryax action (Docker-based) and since the unutilized HPC resources are made available to Ryax as new Kubernetes nodes through BeBiDA then the execution is done natively on Kubernetes without needing the usage of ssh protocol to access the HPC nodes.
- The HPC offloading goes through typical execution of Slurm (or OAR) batch submission through ssh while Ryax-BeBida integration enables the dynamically added/removed BeBiDa related Kubernetes nodes to be specified through taints and the jobs to be executed there through tolerations.
- The HPC offloading technique can be used for all types of HPC jobs while Ryax-BeBiDa technique can be used mainly for Big Data Spark jobs

In the context of workflows execution on HPC, what is interesting with Ryax-BeBiDa integration is that **BeBiDa brings a more transparent usage of HPC resources for Spark applications to Ryax** allowing the elastic resource management of BeBiDa to be enabled on the workflow level, without needing to pass through HPC offloading and packaging of Singularity containers.

Summary - Conclusions - Links

This section provided a description of different integrations around REGALE's workflow engine Ryax, the system manager BeBiDa and resource managers OAR & Slurm, bringing different ways to combine the design and deployment of workflows based on user-friendly and Cloud-native approaches with the execution on HPC clusters for performance and scalability. It provided an overview of the Ryax platform and then focused on two different ways to executing workflows on HPC resources: either through HPC offloading using the ssh protocol and the typical commands of Slurm and OAR for batch jobs execution supporting all types of HPC jobs; or through unutilized HPC resources controlled by Slurm or OAR and switching the control to Kubernetes for elastic execution of Big Data Spark applications using the BeBiDa system manager and its prolog/epilog job techniques. The repositories related to Ryax, BeBiDa, Slurm, OAR integrations and studies are in the following github links[2][3][4].

[2] https://github.com/RyaxTech/ryax-engine
[3] https://github.com/RyaxTech/bigdata-hpc-collocation
[4] https://github.com/oar-team/regale-nixos-compose/tree/main/bebida

4. Integration Scenarios Progress

Integration Scenario #1 “Tag-based Application-Aware Power Capping”

In a few words, the main idea underlying the Tag-based Application-Aware Power Capping (Tag-based AAPC) mechanism is to leverage simple black-box information about the jobs being executed on a supercomputer partition operating under a power cap to share power budgets between the aforementioned jobs more efficiently, when compared with a Fair-Sharing Power Capping (FSPC) policy (every node of the capped partition contributes the same ratio of its reference power consumption to meet the global power envelope).

For the proof-of-concept associated with Tag-based AAPC, the aforementioned black-box information consists in a set of three labels, associated with three classes of HPC applications:

- COMPUTE: associated with compute-bound applications, for instance HPL;
- MEMORY: associated with memory-bound applications, for instance HPCG;
- MIXED: associated with applications exhibiting interleaved memory intensive and compute intensive phases, without any clear predominance of the latter two, for instance NAMD.

For this first development iteration, the labels are specified through user input at the time of job submission. However, for further development iterations, a mixed approach to infer those labels, based on multiple Machine Learning (ML) techniques and user input is under investigation.

Objective of the scenario

The underlying approach of AAPC relies on empirical observations: High Performance Computing (HPC) applications display diverse workloads, each reacting differently to power capping in terms of performance. For example, applications like NEMO TOP/PISCES solver applied to GYRE tend to be heavily memory-bound, meaning they do not consistently require the full computing power exhibited by the node to reach their peak performance. Hence, applying a power cap to nodes executing such applications tends to have only a moderate impact on the performance of the latter. Conversely, compute-bound applications like HPL continuously stress computing cores, making them highly sensitive to power capping. Therefore, enforcing a power cap on nodes running compute-bound applications significantly degrades performance.

Drawing from these experimental findings, the rationale behind the AAPC mechanism involves dynamically redistributing power budgets among compute nodes based on the applications running at a given time. The aim is to prioritise compute-bound jobs to enhance job throughput in the partition compared to the standard "Fair Sharing Power Capping" (FSPC) strategy. By reducing power constraints on nodes running compute-bound jobs, significant performance degradations can be prevented, potentially increasing job

throughput. Conversely, reallocating power budgets from nodes allocated to memory-bound jobs to those allocated to compute-bound jobs may increase performance degradations for memory-bound applications, reducing job throughput. Nevertheless, the prevented performance degradation for compute-bound jobs should offset the induced degradation for memory-bound jobs. This should thus result in an overall increase in job throughput within the power-constrained partition when compared to the FSPC strategy, which is the objective of the Tag-based AAPC mechanism.

By way of conclusion, we note that deliverable D2.3 - “Final integration of sophisticated policies in the REGALE prototype” - provides further details on the rationale of AAPC, as well as a description of the associated algorithm.

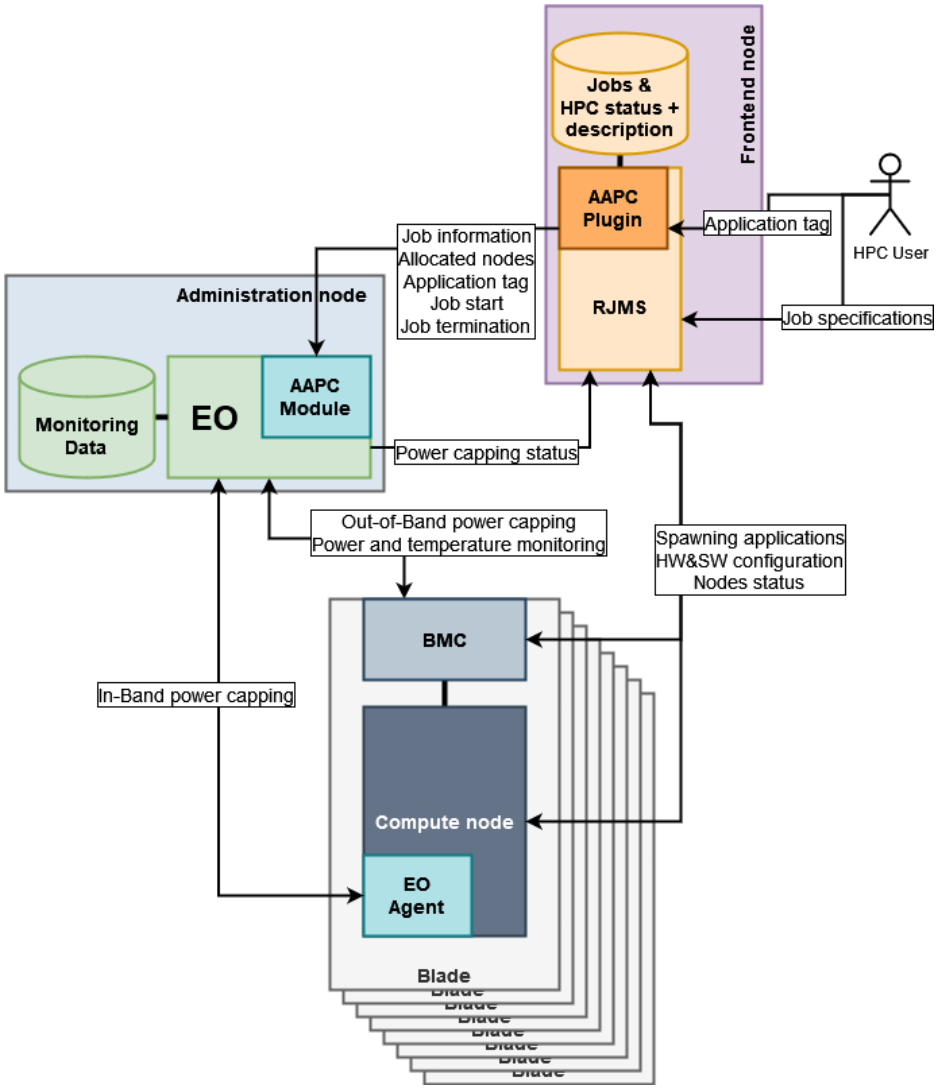


Figure 6: *Overview of the architecture of the Application-Aware Power Capping (AAPC) mechanism, and of its integration in the management software stack of a supercomputer.*

Implementation

An overview of the software architecture associated with the implementation of the AAPC mechanism is presented on figure 6.

The integration in the management stack of a supercomputer of the Tag-based AAPC mechanism is presented by Figure 6, and also detailed in deliverable D2.3 - “Final integration of sophisticated policies in the REGALE prototype”. Its software architecture comprises two main components, which implementations are presented below:

- An AAPC extension plugin for the Resource and Job Management System (RJMS);
- An AAPC module for Bull Energy Optimizer (BEO).

Extension of the RJMS

In the context of this proof-of-concept, the prototype was developed and evaluated on an Atos on-premise HPC system. Consequently, the prototype integrates with the software environment deployed on this on-premise cluster, not with the REGALE one. The main difference is the fact that Slurm is used as the RJMS, and not OAR.

That is why a SPANK plugin[5] was implemented to manage the communication from Slurm (i.e. the RJMS), to the AAPC module for BEO. Indeed, as presented by Figure 6, the AAPC module needs to be notified of jobs’ start and termination, and to be supplied some basic information about the jobs in question (notably the allocated nodes and the tag of the application).

Those requirements translate to three key elements regarding the implementation of the SPANK plugin. First, an option was registered into the Slurm CLI tools to make it possible for end-users to specify the tags of the application associated with the jobs they submit. Figure 7 is a verbatim illustrating the usage of this option.

```
# Memory-bound job started using srun:
user@host $> srun -J jobname \
|   |   |   |   --aapc-job-tag MEM \
|   |   |   |   -N 5 -n 120 --ntasks-per-node 24 \
|   |   |   |   /path/to/application

# Compute-bound job started using sbatch, with tag specification on the CLI:
user@host $> sbatch --aapc-job-tag CPU /path/to/script.sbatch

# Mixed job started using sbatch, with tag specification in the sbatch script:
user@host $> head -n 7 /path/to/script.sbatch
#!/bin/bash

#SBATCH --job-name=$NAME
#SBATCH --nodes=$NODES
#SBATCH --ntasks=$NTASKS
#SBATCH --ntasks-per-node=$NTASKS_PER_NODE
#SBATCH --aapc-job-tag=MIX

user@host $> sbatch /path/to/script.sbatch
```

Figure 7: *Verbatim illustrating the use of the --aapc-job-tag option registered by the SPANK plugin associated with AAPC into the CLI tools of Slurm for job submission.*

Second, a callback function was associated with the `slurm_spank_user_init` hook to notify the AAPC module that a job has started, and to send the list of allocated compute nodes.

[5] SPANK is a C interface exhibited by Slurm to make it possible to register callback functions to be executed at specific key stages of the lifecycle of an HPC job.
More information here: https://slurm.schedmd.com/spank.html

This hook was selected because it is, chronologically, the first hook executed after resources are allocated to the considered job, and the root privileges are dropped. Since it is executed on each node allocated to the job, it is enforced that only the head node effectively performs the aforementioned actions.

Third and last, a callback function was associated with the `slurm_spank_exit` hook to notify the AAPC module that a job is terminating. This hook had to be selected since the job ID and the list of allocated nodes are not available anymore in the `LOCAL`[6] context of Slurm at this stage of the job lifecycle. Thus, the last hook of the `REMOTE`[7] context of Slurm had to be used to access that information. Once again, it was enforced that only the head node should send the termination notification to the AAPC module.

AAPC module for BEO

The Tag-based AAPC module is a standalone program, written in C. It can be executed anywhere as long as: (1) it can reach the instance of BEO monitoring the target HPC partition, (2) it has the required privileges to issue power capping directives to BEO, and (3) it can be reached by the compute nodes of the target supercomputer. That being said, it was designed to be executed on one of the management nodes of the considered HPC system. Additionally, the AAPC module was designed to be spawned when an administrator enables this power capping mechanism for a given partition of compute nodes. It can be terminated at will, and, in this case, will restore the power constraints which were being enforced when it was spawned.

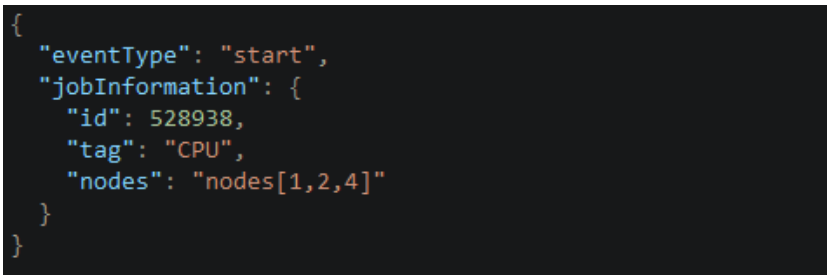


Figure 8: *Example of JSON file encapsulating the information about the creation of a new job, sent to the AAPC module by the associated SPANK plugin.*

We now focus on the interfacing of the AAPC module with BEO and the aforementioned SPANK plugin. First, the AAPC module uses the REST API of BEO to create, enforce, and remove capping rules to be applied to the compute nodes of the managed HPC partition. Those rules are computed based on the internal representation of the managed HPC partition[8] kept by the AAPC module. Once again, more details can be found in deliverable

[6] Roughly, the `LOCAL` context refers to the execution context of `slurmctld` associated with the considered job.

[7] Roughly, the `REMOTE` context refers to the execution context of `slurmstepd` on the compute nodes associated with the considered job.

[8] This representation notably contains which compute nodes are idle, the list of running jobs, their tags, and the nodes allocated to them.

D2.3 - “Final integration of sophisticated policies in the REGALE prototype”. To build the aforementioned representation, the AAPC module requires information sent by the dedicated SPANK plugin, at job allocation and termination. The AAPC module opens a TCP socket, binds to a port and listens for upcoming HTTP POST requests sent by the SPANK plugin. The latter requests contain JSON files encapsulating all the aforementioned information requested by the AAPC module. An example of a JSON file is featured by Figure 8. As a final remark, Deliverable D1.4 - “REGALE Evaluation” - presents the protocol designed and implemented to experimentally evaluate the AAPC mechanism, and the associated results.

Integration Scenario #2 “Application-aware energy optimization under a system power cap”

This scenario covers a case where both the Job Manager and the Node Manager enforce the powercap, optimising the power management states of the computational resources, in function of the needs of the running application.

The Node Manager in fact, can select thanks to its heuristics, the appropriate frequencies for the cores on which the application is running, and the Job Manager will enforce this decision on a fine grained level (which specifically is the one of the entrances and exits of the MPI events).

The Monitoring system will provide details to the system administrators and to the users about the platform's metrics and energy efficiency, reported both by the Node Manager and by the Job Manager, in addition to those obtained independently.

The goal of this scenario is to demonstrate a fruitful collaboration among implementations of these different entities in the REGALE ecosystem, leaving to each one their specific characteristics and instead reinforcing the overall action.

To accomplish this, the following components will be used:

- **Job Manager (JM)**: specifically we employ COUNTDOWN, which can reduce and restore frequencies, at the beginning and at the end of the MPI phases of the application. The current maximal frequency is chosen, depending on the decisions of the Node Manager during the execution of the program. COUNTDOWN obtains the current Node Manager frequency at the end of each MPI phase using it as the restoring operating frequency.
- **Node Manager (NM)**: we employ EAR to provide an in-band power capping service.
- **Monitor**: we employ EXAMON, a monitoring system that is composed of a set of frameworks used for the collection of monitoring signals and a NoSQL database to store the large-scale time-series monitoring data. It supports different visual and command line interfaces for communication and data extraction.

All the reported components have been extended to be compliant with the REGALE library; this means that they could also be easily plugged out, leaving the possibility to other implementations of the listed actors of replacing them (for example, changing EXAMON with DCDB) without any further modifications, except the ones needed to comply with the REGALE requirements.

Moreover, the unresolved problem reported in the Deliverable 3.2, regarding the frequencies written in the *scaling_setspeed* file (specifically, if the Node Manager decides to apply, exiting an MPI phase, a frequency equal to the minimum one, the Job Manager will not be able to understand this intention, and it will be overwritten incorrectly), can be resolved by moving to this different approach, which is using the REGALE library as communication layer among the involved tools.

The original collaboration among EAR and COUNTDOWN in fact, was based on the idea of using the *userspace* governor, which lets the content of the *scaling_setspeed* file to be modified and written, to change the current frequency for a group of CPUs. But the usage of a common file can, as we have already seen, lead to some problems. The usage of the REGALE layer should smooth these issues.

Moreover, if previously the sending of the information to the Monitoring was based directly on the MQTT protocol, with the REGALE library this behaviour happens on a more abstract layer, leaving each component sending information through specific APIs, which wraps the underlying functionalities of the DDS protocol used by the new communication layer.

EXAMON has been extended with a separate component called BRIDGE (usable by other Monitoring systems), which takes care of receiving DDS messages and translating them following the specific and desired data model; moreover, it resends the info via the MQTT protocol. We provide more details in Section 6 “REGALE Library Extensions”.

Integration Scenario #3 “Application-aware power capping with job scheduler support”

Objective of Scenario

This integration scenario covers a use case where the job scheduler plays an active role in the power management. In this scenario, the System Power Manager applies the cluster powercap algorithm to dynamically setting the node powercap based on application characteristics. The System Power Manager will collect data from the running jobs and will reallocate power (if needed). The Job Scheduler receives information from the System Power Manager about the system status in terms of power consumption. Based on this information, the Job Scheduler can take several actions/decisions: It can (1) influence the scheduling policy, the order of jobs, and job priorities in order to adapt the scheduling to the system status, and (2) it can force the System Power Manager to reduce the allocated power of running nodes in order to guarantee some power for new jobs.

The System Power Manager uses application characteristics information. This information is provided at runtime and, depending on the tool, could be reported by the Job Manager, Node Manager, and Monitor. Also, depending on the tool, it could be available through a direct call to the component or indirectly through the DB.

The Job Scheduler interacts with the System Power Manager, the Workflow Manager or the Power predictor. As an instance, it can estimate the power required of waiting jobs, running jobs, and from the workflow engine. In this scenario, the Job Scheduler will offer some mechanism to (1) notify of basic events such as job start and end in order to (2) define the settings to automatically load the Job Manager.

The Node Manager will apply the powercap selected by the System Power Manager.

The **goal of this scenario** is to implement a cooperation between the Job Scheduler and the System Power Manager while maintaining the distribution of the responsibilities to improve the overall job scheduling and power capping management. This scenario eliminates the complexities of the power capping management from the scheduler. The metric of success will be the throughput of the system.

Implementation

To implement *IS#3*, the following components have been combined and extended:

- **Job Scheduler (OAR)**: The Job Scheduler has been extended to execute the OAR-EAR plugin. EAR uses the Job Scheduler to get the notification of some events and to automatically load the Job manager. In order to make it as independent, flexible, and portable we have defined it based on a prolog/epilog approach. Furthermore, python bindings for the SPM (EARGM) REGALE API have been created which abstract the direct EAR-facing API and provide a generic interface for other SPM tools to serve, should they choose.
- **System Power Manager (EARGM)**: The EARGM is the EAR System Power Manager. It has been extended to implement an outward facing API that can be used to retrieve the status of the cluster and to dynamically set the current power budget. The EARGM now implements an API to export the power consumption, the “status' of the cluster regarding the power capping and the overall power consumption. The Job Scheduler will use this information as hints. Through the REGALE API layer, the SPM also provides the option to send events (such as job execution start/end) and new actions to execute on a global level when certain conditions, which are also sent through the API, are met.
- **Node Manager (EARD)**: The EARD is the EAR Node Manager. It is in charge of applying the power capping limit of the node and, in the case of EAR, it reports runtime information used by the System Power. Through the REGALE API, it now provides a generic interface to set the power limit of a node independent of the EAR-specific API, allowing for further extensions with other SPM tools in the future.

- **Monitor (EAR/Examon/DCDB)**: The Monitor is used in this scenario to report information in the database that will be later used by the Power predictor. It can be any one of the Monitors in the project. EAR data can be reported to DCDB and Examon via specific plugins or through the REGALE library Monitor API. There are no specific extensions associated with the scenario at this stage of the integration.
- **Job Manager (EARL, CNTD)**: The Job Manager is part of this scenario in a very collateral way. There are no specific extensions associated with the scenario at this stage of the integration. In the case of EARL, the role of this component is to provide hints and runtime metrics to be used by the System Power Manager, the Node Manager, and the Job Scheduler. However, the JM is not doing the predictions by itself, but it uses the ones done by the Power estimator, indirectly. In the case of EARL and CNTD, the two components will guarantee that they will not take actions that could generate a power consumption exceeding the powercap limit.

While the implementation of the REGALE API has allowed the ease of communication between tools, as well as interchanging different tools on the same position (i.e. provider A's NM for provider B's NM), the main effort of this Scenario has been on the integration and testing of a mechanism to turn on/off job executions depending on system workload (SPM-JS integration). This has been done through the creation of two scripts (one for each functionality) that directly call the corresponding JS (OAR) commands to stop or start job executions, and that are used whenever the total power consumption reaches certain thresholds set by the administrator. Further work will focus on observing and testing how the alteration in job scheduling interacting with the SPM's power distribution method affects the total system throughput.

Testing implementation

Further development has been dedicated to the implementation and integration of a set of simulators that allows us to test the scalability of the solution on large clusters without having to allocate real resources for this purpose. To do so, we have used the following tools:

Batsim: an open-source Resource and Jobs Management Systems (RJMS) simulator. It allows to simulate the behaviour of a computational platform on which workloads are executed according to the rules of a scheduling algorithm. Built on top of the SimGrid simulation framework. Batsim aims at improving practice in the implementation of resource management algorithms.

Batsim's simulation involves two components. The Batsim component orchestrates the simulation and manages the computational resources, while the decision-making component makes decisions. The two components interact through an event-based protocol.

While it offers many capabilities, we mainly used it to simulate the slowdown that jobs suffer when a powercap is being enforced and the compute nodes cannot run at their target settings.

Batsched: a set of Batsim-compatible scheduling algorithms implemented in C++ to make decisions and communicate them to Batsim using an event-based protocol. This is the other main component involved in a simulation with Batsim. We used this to simulate the Scheduler component in the stack. Several portions of Batsched have been modified to incorporate the capability to interact with the tools created specifically for this test.

Cluster_sim: a cluster simulator with EARDs on every compute node that was built to simulate a big size cluster which connects to an EARGM to test the power capping scenarios. This was built with the specific purpose of testing EARGM's power distribution capabilities as well as testing the basic integration of the SPM-Scheduler.

The first stage design of cluster_sim was running using a prepared workload trace file as input for the job triggers data, in addition to the cluster size, max number of jobs, and the number of simulated jobs. The second (and final) stage of implementation of cluster_sim is using the input job data event triggers from Batsim/Batsched. It generates two output CSV formatted traces: one with the jobs' summary data including start time, end time, wait time, number of nodes, job id, average power, average powercap, average stress and requested power; the other is a timestamp-based trace where, for every event (new job, end job, reallocation) an entry is generated for every active job plus another entry for all the idle nodes. The first trace is used to obtain the measurements of the performance of the system, while the second one can be used to generate a timeline of what is happening in the simulated system.

Cluster_sim - Batsim/Batsched Integration Description

As previously mentioned, several extensions were made to both cluster_sim and Batsched/Batsim (specifically on Batsched's side) to have the tools cooperate with each other to evaluate EARGM's powercap distribution. This is achieved through connecting cluster_sim with Batsim+Batsched, where Batsched is used as a scheduler simulator and cluster_sim acts as the cluster's EARDs which are connected with EARGM to apply the power capping. Batsim's role is to send the submission of jobs to Batsched, as well as simulating the time it takes for said jobs to execute once Batsched decides to start them.

To start off, Batsim and Batsched already worked with each other as follows: Batsim processes an input file that contains the workload to simulate. It takes the jobs submitted in the workload and sends the corresponding event (JOB_SUBMISSION) to Batsched. Batsched then decides whether to start the job immediately or to keep in the queue depending on the availability of resources and the scheduling algorithm currently in use. When a decision to start a job is taken, Batsched will communicate that to Batsim and Batsim will simulate the job, communicating to Batsched when it is finished. All this communication is done through their own interface, using Batsim's event-based protocol.

Similarly, EARGM and cluster_sim already worked with each other as of the first cluster_sim implementation. The way it works is: whenever a new job or end job event is received, the EARGM will request the power status of the cluster to cluster_sim, then process the information received and, when necessary, send to cluster_sim the new power distribution for all the nodes in the cluster. This implied a small change to EARGM just for testing purposes, since the power check and its following reallocation is now done on demand instead of periodically (as EARGM does in a live environment with a cluster of "real" EARDs/NMs). Since EARGM is a stateless component, meaning that it carries no information from one period to the next one, the only change was that it now "sleeps" until cluster_sim sends a message to "wake up" and initiate a new period. All the communication between components is currently done through EAR's internal communication API, and the change to use the standardised REGALE API has been added to the roadmap so that other SPMs can be used for this test.

To proceed with this integration we needed to connect cluster_sim with it to receive all the job events and do its work based on that information. To do so, modifications have been made on both simulators.

Since Batsched is responsible for generating most of the simulation events, and it also receives the ones it does not generate, it was initially decided to use a shared memory region to send the job information as well as the simulation start and end events to cluster_sim. This shared memory region is created by either Batsched or cluster_sim, while the other component will detect that it exists and simply open it. The order of execution does not matter since either component will block until it detects that the other is ready to start the entire simulation process.

```
typedef struct
{
   int flag;
   int simulation;
   double start_time;
   double end_time;
   char type[64];
   char id[16];
   int alloc;
} shmem_data_t;
```

To ensure that the simulation will only start once both components are connected, a flag in the shared memory region is used which shows whether the shared memory region has been opened or not, and once both components have opened the shared memory region, cluster_sim will update this flag, which will trigger Batsched to send the necessary events to Batsim for the simulation to start. This first extension was done on both cluster_sim and Batsched's code.

On further extensions, pipes have been created and used between both cluster_sim and Batsched. These pipes are used to block both programs until the simulation can start, as well as to share data regarding each job that needs to be updated. This data includes: sending the job's nodelist from Batsched to cluster_sim so that cluster_sim knows exactly what compute node (and by extension what EARD/NM) is running each job; sending the list of pstates changes per node from cluster_sim to Batsched that are a consequence from the varying powercap of each node; and sending the order to stop/restart the execution of new jobs from cluster_sim to Batsched when the EARGM deems this appropriate.

The diagram of figure 9 provides a rough view of how a job is processed through the different tools to obtain the results. In the event of a new job submission, Batsched will call the algorithm with the "make_decision" method to queue the job and, if available, assign it the requested resources and then execute it. For the integration cluster_sim required all this information about a new job submission and thus we created a new method called "make_decision_with_data" which will be used by Batsched instead of the original one. This function is the same as the original one except it will add all the job information in a temporary list and share it with cluster_sim through the previously initialized shared memory and pipes.

At this stage, on cluster_sim's side, when a new job submission event is received, it will process the list of the nodes. Then, it will communicate with EARGM so that it processes the current power needs of all the cluster's nodes. EARGM will return the changes in power allocation for the nodes, if applicable. Finally, if necessary, cluster_sim will update the nodes' pstates (according to the new power settings) and it will send the list of changes to Batsched through the corresponding pipe. Batsched has been modified to check if there are any pstate change requests from cluster_sim and, if there was a request, it will get the list of the nodes and their updated pstates and proceed with updating the node's pstate on Batsim's side accordingly, using the function "set_machine_state".

On the event of an end job, which will have the event type "JOB_COMPLETED" and is sent by Batsim to Batsched, Batsched will update the shared memory region with the job data including event_type, job_id, and end_time. Once cluster_sim has received an end job event, it will set the end time of the job in its records, then set the power of the nodes used for the job to their base idle power and will send the job record data into the final trace output file.

Once the simulation is ended cluster_sim will be notified through both the shared memory region and the pipes using a "simulation end" flag. Then cluster_sim will generate the final output trace of the simulation which includes finishing any jobs for which an end job event has not been received (which should be none).

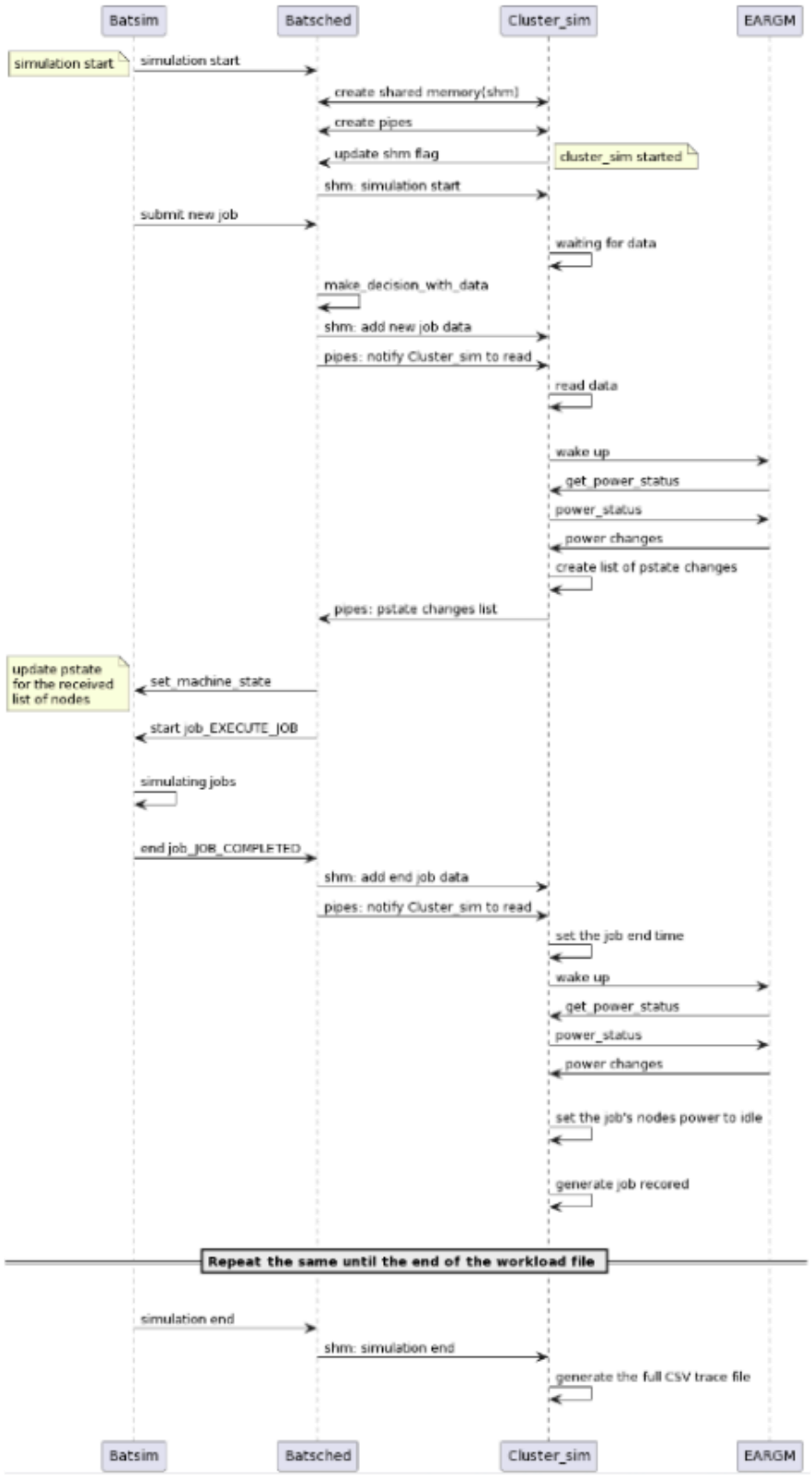


Figure 9: Diagram of a workload simulation, showing all the steps of the first job's simulation.

Integration Scenario #4 "Powercap with Scheduler and Job Manager support"

Objective of Scenario

Similarly to IS#3, this scenario covers a use case where both the job scheduler and the job manager play an active role in the power management.

There are two sub-scenarios depending on the use case:

Scenario 4.1 Job Manager as a powercap balancer

In this scenario, the System Power Manager applies the cluster powercap algorithm to dynamically set the node powercap based on node power requests. The System Power Manager will collect data from the nodes and will reallocate power (if needed). The Job Scheduler receives information from the System Power Manager about the system status in terms of power consumption. Based on this information, the Job Scheduler can take several actions/decisions: It can (1) influence the scheduling policy, the order of jobs, and job priorities in order to adapt the scheduling to the system status, and (2) it can force the System Power Manager to reduce the allocated power of running nodes in order to guarantee some power for new jobs.

The Job Manager will act as a second layer of power distribution, in this case among nodes of the job it controls. The Job Manager uses application characteristics information. This information is provided at runtime and, depending on the tool, could be reported by the Job Manager itself, Node Manager, and Monitor. Also, depending on the tool, it could be available through a direct call to the component or indirectly through the DB.

The Job Scheduler interacts with the System Power Manager, the Workflow Manager or the Power predictor. For instance, it can estimate the power required of waiting jobs, running jobs, and from the workflow engine. In this scenario, the Job Scheduler will offer some mechanism to (1) notify of basic events such as job start and end in order to (2) define the settings to automatically load the Job Manager.

The Node Manager will apply the powercap selected by the System Power Manager.

The Job Manager will (if necessary) reallocate the power given to the Node Managers that correspond to the nodes running its job depending on their necessities. The total power budget of a job will be the power granted to the Node Managers by the System Power Manager.

Scenario 4.2 Job Manager as a powercap allocator

In this scenario, the System Power Manager acts as a global power monitoring tool first and foremost, with additional power limiting capabilities available as a fail-safe mechanism. The

System Power Manager will periodically collect power consumption data from the nodes (Node Manager) which the Job Scheduler can request. Based on this information, the Job Scheduler can (1) influence the scheduling policy, the order of jobs, and job priorities in order to adapt the scheduling to the system status, and (2) grant power allocations on a per-job basis.

The Job Manager will act as a power allocator among the nodes of its corresponding job. It will interact with the Node Managers, which will act as powercap enforcers. The Job Manager uses application characteristics information. This information is provided at runtime and, depending on the tool, could be reported by the Job Manager itself, Node Manager, and Monitor. Also, depending on the tool, it could be available through a direct call to the component or indirectly through the DB. The power budget given by the Job Scheduler will be distributed by the Job Manager amongst Node Managers depending on this information.

Implementation

Scenario 4.1

This scenario is the main extension of scenario 3 since the roles of the System Power Manager and the Job Scheduler remain unchanged. As such, the implementation of this scenario only affects the Node Manager and the Job Manager. For the first version of the implementation, EAR's Node Manager and Job Manager are being modified to provide this functionality.

On the Node Manager's side, an API to attend the new commands from the Job Manager must be implemented. Due to the way the SPM-NM communication works, this has been implemented through three different calls which the JM will send to the NM:

- First, a function to retrieve current powercap information for all the nodes involved in a job. This includes current powercap allocation, desired power increase (if necessary) and amount of stress each node is under. The sum of the current powercap allocated to all the nodes under a job would represent that job's *current* power budget, which would be dynamic (the System Power Manager may give or take away power from the nodes, making the budget change).
- The second function will send the new powercap settings to the Node Manager. The Node Manager will then reply with an OK if the new settings can be applied, or with a REJECT if they cannot. The decision on whether the new settings can be applied or not is taken based on whether there is a power redistribution process between the NM and the SPM, since this process (cluster-level) supersedes the JM-NM interaction (at job-level) even if the latter has started the process first. This step will not apply those new settings, but the NM will store them.
- Finally, if all the nodes reply with an OK, a confirmation call (ACK) will be sent to apply those settings. Otherwise, a rejection command (REJECT) will be sent to the NMs so they can clear the stored settings, and they will remain as they are.

Figure 10 shows the interaction between the JM (EARL) and the NM's (EARD) API when distributing power between two NMs and no SPM power reallocation happens.

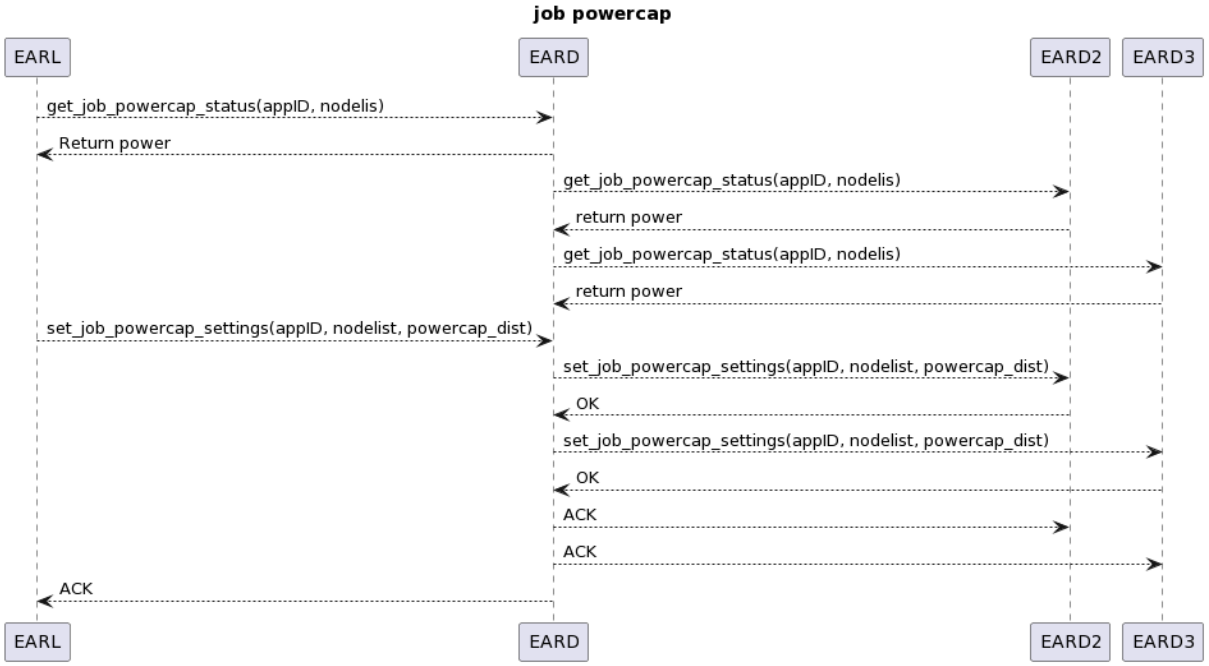


Figure 10: Diagram of a standard power redistribution between the JM (EARL) and several NMs (EARD, EARD2 and EARD3, coordinated by the first one).

This functionality has been created and tested on the NM's side, with positive results as to the latency of the communications and the robustness of the system when other power management (namely, the SPM global power distribution) methods are in place. To do so, a small synthetic tool, which uses the exact same calls that the Job Manager would, was created and, in conjunction with an administrator tool (EAR's *econtrol*) that already implemented the SPM-related calls, all the possible scenarios in regards to functionality were tested.

On the Job Manager's side, a first draft on how to approach power distribution was written, but the implementation has not started yet. The general idea was to first use the stress each node is under (which is retrieved by the first call, *get_powercap_status*) as an indicator, and favour removal of power in nodes with low or no stress to then move that power to the nodes which have the highest stress value. In doing so, a better power distribution than a purely homogeneous one would be achieved, since nodes with less computing needs would receive less power which would then be moved to the ones that need it most. This would also help in fighting imbalances in barrier times, since all nodes would finish their computational load at a similar rate, regardless of their actual power needs. Further power distribution methods, which would use the application-specific data that the Job Manager has access to, would be the next options to explore once the first one has been tested and measured.

Scenario 4.2

For this scenario, the Job Manager takes a more active role in power distribution, with increased output from the Job Scheduler, while the System Power Manager acts only as a fallback. Implementation-wise, this scenario could be considered an extension of scenario 4.1, since most of its functions translate directly to this one, where most of the work would be put in the decision-making algorithm for the Job Manager and the power prediction/allocation done by the Job Scheduler.

The Job Manager would now receive a static power budget per job, instead of a dynamic one in the case of IS#4.1, and this would be given by the Job Scheduler. A way to provide this information to the JM would need to be implemented. Since there is already a pre-existing integration between the JM-JS (in the case of EAR-OAR), an extension of that should be the way to proceed. With this small change, a first version would be viable for testing using the power distribution algorithms of IS#4.1. Since the previous scenario is a more complex case (with the power budget being dynamic and thus having the need to reassess the situation with a higher amount of variables), any improvements made there would, in theory, translate to improvements to this scenario. While it is not guaranteed, a first assessment would be needed and, should the previous work not translate to this integration scenario, a new power distribution method could always be added afterwards.

On the Job Scheduler side, the current cluster power usage and the available power to give out to jobs would have to be taken into account when allocating them, with a power budget directly associated with each and every job. This information would have to be passed on to the Job Manager, but this implementation would be done in tandem with the previous point which would allow the development to move rapidly onto the power predictor and the new needs that this scenario would put on it.

Finally, the System Power Manager would need to change its role from power distributor to a power monitor, and act only as a failsafe mechanism. In the case of EAR, this functionality is already provided with several ways to configure. A typical one, where the compute nodes only apply powercap when the entire system is under stress (ie, the cluster is consuming >90% of the total power allocated by the administrator) would translate well to this scenario, with none or very minor modifications needed to start work on it.

Final notes - Conclusion

While this integration scenario is out of scope for the REGALE project, it is in our current interest to continue this line of research and develop the aforementioned solutions. This also shows that the work on the REGALE project has continuity further than the project itself, and the integration between the tools forming it should only increase moving forward.

Integration Scenario #5: Control energy budget by coupling Job Scheduler, Power Manager and Workflow Application

The goal of this scenario is to provide integration of the Job Scheduler and part of the System Power Manager with the Workflow manager. The concrete approach consists of coupling Melissa, OAR and EAR to monitor and control the energy budget. Additionally we use the NixOS-Compose framework to build and provide a portable and reproducible full software stack for demonstration and test purposes.

The Melissa workflow manager interacts with the Job (or batch) scheduler to request the execution of the various simulation instances required for sampling the simulation behaviour in the parameter space and compute statistics or train a deep surrogate depending on the set-up server enabled to process on-line the produced data.

Power Control

Three different online power control strategies have been considered. They all leverage the elastic/fault-tolerance capabilities of Workflow application (Melissa). The execution order of the simulation instances is fully left to the batch scheduler (elasticity). Melissa data processing algorithms are capable of working with any data arrival order. A simulation instance can be killed (and restarted) at any moment (fault-tolerance). Melissa takes care of making sure that no data is missing or processed several times.

1. Best Effort Queue: OAR can be configured to enable a best effort queue where jobs can be killed without notice. First strategy consisted in submitting part of Melissa simulations on a OAR best effort queue and let OAR decide to kill these jobs if it detects with the help of EAR that some power limit has been reached.

2. Power Tokens: OAR can be set to have a fixed number of power tokens where each token represents a given amount of power. When allocating jobs to compute resources, OAR also assigns the necessary number of tokens representing the job power consumptions. No new job is started if no token is available. This is a simple strategy, not very precise, but easy to set up.

3. Power Capping: When EAR detects that the power cap on a node has been reached it informs OAR that in turn identifies the associated job (a Melissa simulation run) and kills it to respect the target power cap. Melissa fault-tolerant operates to detect the killed job and resubmit it to OAR that will execute later.

Portable and reproducible full software stack

The goal of NixOS Compose is to reduce the burden of setting up ephemeral distributed systems through the Nix functional package manager and the NixOS distribution. It allows us to build and deliver a complete portable and reproducible software stack, deployable at scale on test platforms like Grid'5000. The general principle is to declaratively describe the software stack as a composition of components with their main configuration settings. By processing this description, the framework creates a deployable image for the specific target (e.g. virtual machines or Grid'5000). The build is guaranteed to be reproducible thanks to the Nix package manager, it consists of cryptographically fixing and verifying the source used for all software including the build environment and the build recipe.

The code block below shows an excerpt of a software stack declaration in Nix language. All software stack declarations are available at:

https://github.com/oar-team/regale-nixos-compose.

```
  nodes = let
    earConfig = import ../lib/ear_config.nix {inherit pkgs modulesPath nur setup;};
    oarConfig = import ../lib/oar_config.nix {inherit pkgs modulesPath nur flavour;};
    commonConfig = import ../lib/common.nix {inherit pkgs modulesPath nur flavour;};
    melissa = import ../lib/melissa.nix {inherit pkgs modulesPath nur flavour;};
  in {
    frontend = {...}: {
      imports = [commonConfig oarConfig earConfig melissa];
      nxc.sharedDirs."/users".server = "server";

      services.oar.client.enable = true;
      services.oar.web.enable = true;
      services.oar.web.drawgantt.enable = true;
      services.oar.web.monika.enable = true;
    };
    server = {...}: {
      imports = [commonConfig oarConfig earConfig melissa];
      nxc.sharedDirs."/users".export = true;

      services.oar.server.enable = true;
      services.oar.dbserver.enable = true;
      services.ear.database.enable = true;
    };
    eargm = {...}: {
      imports = [commonConfig oarConfig earConfig melissa];
      nxc.sharedDirs."/users".server = "server";

      services.ear.global_manager.enable = true;
    };

    node = {...}: {
      imports = [commonConfig oarConfig earConfig melissa];
      nxc.sharedDirs."/users".server = "server";
      systemd.enableUnifiedCgroupHierarchy = false;
      services.oar.node.enable = true;
      services.ear.daemon.enable = true;
      services.ear.db_manager.enable = true;
    };
  };
```

5. Towards Workflow and Powerstack Paths Integration

In this section we discuss the efforts towards the integration of our two distinctive paths in REGALE: the Workflow path and the Powerstack path.

In particular we describe the following combinations: **Ryax-OAR-EAR** and **Melissa-OAR-EAR**

The **Ryax-OAR-EAR** integration study allowed us to bring energy-awareness in the Ryax workflows. This energy-awareness is brought to the Ryax workflow engine by giving users the capabilities to express a certain energy cap at the workflow level and in particular this is directly related to the definition of the Ryax actions to be offloaded on HPC. The energy-cap is expressed as a parameter on the ryax_metadata.yaml file. At the time of deployment the execution goes through ssh using OAR command while at the same time EAR command is submitted through ssh to effectively enable a certain energy cap for the Ryax action to be executed. The integration of Ryax-OAR-EAR is currently developed in low-TRL levels and more work is planned to validate and enable the efficient integration between Ryax and EAR.

The **Melissa-OAR-EAR** integration is presented and discussed in Section 4, subsection "Integration Scenario #5: Control energy budget by coupling Job Scheduler, Power Manager and Workflow Application" . For power monitoring, the solution developed is effective and integrated into the Melissa code, so any user, given that OAR and EAR are enabled on the supercomputer, can track in real time the power consumption of the running Melissa workflow. We will also support Slurm in the near future. For power control, i.e. enabling a dialogue between melissa, OAR and EAR to constrain the execution to meet some power budget, we have experimented various scenarios as discussed in the Scenario#5 section. But these strategies are still at low TRL (research prototype). More experiments and validations at scale, impaired by the difficulty to have a large machine with EAR and OAR enabled, are required. Extra work steps are required before we can make such features available to users and envision integration with Powerstack.

6. REGALE Library Extensions

The REGALE library has been quite extended from what has been presented in the Deliverable 3.2.

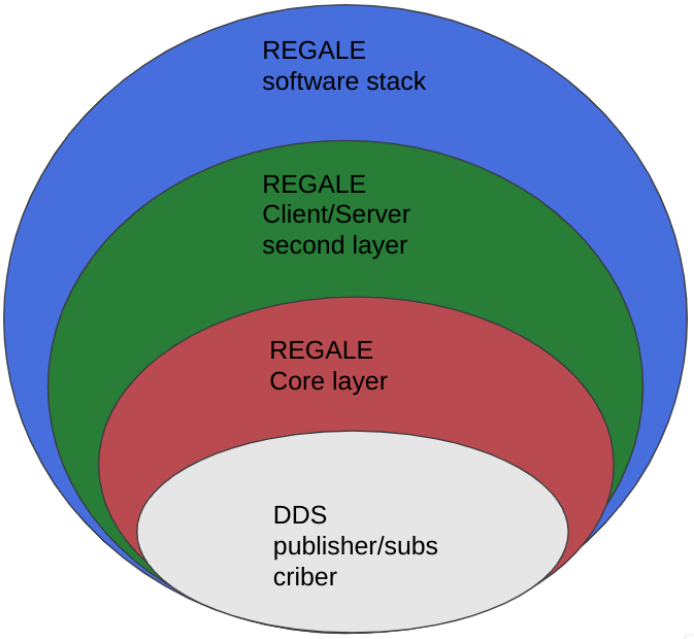


Figure 11: Architecture overview of the REGALE library

In the figure 11 we can see that the REGALE library is based on the DDS functionalities, which are wrapped and made easy to use by the *Core layer*, the layer responsible for the management of the publishers and subscribers, plus everything correlated to the DDS aspects, options and optimizations. Higher up we find the *Client/Server layer*, which provides the API for the REGALE agents involved in the REGALE ecosystem, and which need to interface with its standard. To conclude, in the *Software stack* final level at the moment there exist implementations of the Job Manager, the Node Manager, the Monitoring and a number of synthetic agents, whose aim is to emulate the behaviour of the previously listed entities, deploying different (and more concise) implementations of their natural behaviour. We will provide an explanation later on.

The target behind the REGALE library is the creation of a standard as a communication layer among all the tools involved in the power stack. As such every component interested in using it, should just be coherent with its standard, respecting the declarations of its APIS and the definitions of the callbacks needed for the server side of the Client/Server APIs layer.

Core Layer

This layer is composed by 4 main components:

i)RegaleObject: a base class responsible of Q.O.S. (Quality Of Service), topics, transports and to get and set, dynamically, the data types. This is an improvement since the original implementation of the REGALE library, because now the library can manage different types

(integers, floats, chars, strings and arrays) automatically, without any internal ad hoc implementation for some specific messages. Moreover, this is also from how FastDDS (the actual DDS implementation used under the hood of the REGALE library) defines data types, which is by IDL (Interface Definition Language). Infact, an IDL structure will be converted into a C++ class in which the members of the structure, defined via IDL, are mapped to private data members of the class, whose getter and setter functions are also created to be able to access data members.

Instead of creating an IDL file and then call the FastDDS code generation tool, the REGALE library dynamically creates its own data types, which are still stored in the xml file from which they are parsed; the same behaviour is maintained also for other configurations, like the choice of the transport type (shm, udp, tcp).

```
<!--See: https://fast-dds.docs.eprosima.com/en/latest/fastdds/xml_configuration/making_xml_profiles.html
     section \"Rooted vs Standalone profiles definition\" -->
<dds xmlns="http://www.eprosima.com/XMLSchemas/fastRTPS_Profiles">
    <types>
        <type>
            <struct name="regale_struct">
                <member name="power" type="int32"/>
                <member name="frequency" type="float32"/>
                <member name="name" type="string"/>
                <member name="values" type="float64" arrayDimensions="10,1"/>
            </struct>
        </type>
        <type>
            <struct name="mqtt_string">
                <member name="topic" type="string"/>
                <member name="payload" type="string"/>
            </struct>
        </type>
    </types>
</dds>
```

ii)RegalePublisher: it is a derived class, responsible for its specific Q.O.S. and of the creation of the Data Writer, to actually publish data on specific Topic/Partition pair.
iii)RegaleSubscriber: it is the second derived class, responsible for the Subscriber Q.O.S. and of the Data Reader to receive actual data, if present. Same Topic/Partition combination for the RegalePublisher must be provided, to let the two objects communicate.
iv)regale: the corresponding source file is the actual bridge among the underlying C++ implementation and the wrappers in C, to be called from a C/C++ code.

This bridge implements the following methods:

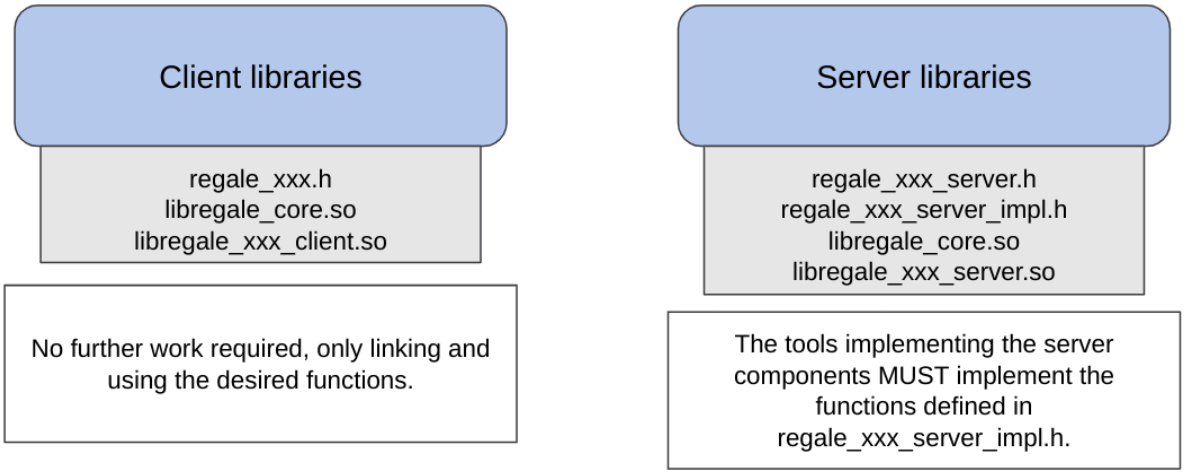
- *Regale_init*, which is the method used to initialise a **RegaleStruct**, which is the base element for the dynamic data getting/setting.
- *Regale_malloc/dealloc* and *Regale_finalize*, which are the functions to allocate/deallocate memory, for the total amount of **RegaleStructs** needed for the data type of a specific message.
- *Regale_create_publisher* and *Regale_create_subscriber*, which are the methods to actually create derived objects **RegalePublisher** and **RegaleSubscriber** (we recall that they need a topic, a partition and a paths to the profiles and data types xml files to be used, for the messages' exchange).

- *Regale_publish*, which is the method to actually publish the values for a specific data type. No symmetrical method for the subscriber, because it asynchronously "listens" for incoming messages.
- *Regale_delete*, to destroy **RegaleObjects** previously created.

```
RegaleStruct* regale_struct = regale_malloc(4);

regale_init(regale_struct,
            1            ,
            REGALE_INT   ,
            regale_struct + 1);
(*((int*)(regale_struct->elements))) = 14;

regale_init(regale_struct + 1,
            1               ,
            REGALE_FLOAT    ,
            regale_struct + 2);
(*((float*)((regale_struct + 1)->elements))) = 30000.00;

regale_init(regale_struct + 2,
            6               ,
            REGALE_CHAR     ,
            regale_struct + 3);
strcpy((char*)((regale_struct + 2)->elements), "test");

regale_init(regale_struct + 3,
            10              ,
            REGALE_DOUBLE   ,
            NULL);
for (int i = 0; i < 10; i++) {
    ((double*)((regale_struct + 3)->elements))[i] = i;
}
```

Client/Server layer

Each REGALE agent (Job Manager, Node Manager, Monitoring, etc.) has both a server and a client library to use, which internally call the core layer. If a tool intends to be REGALE compliant, it must just call the API proposed (client part) or implement the "interfaces" still defined for the server part (which actually must manage the behaviour of the callback used once received the data).



The agent compliant with the client part of this layer, has to (in addition to link the corresponding client library):

- Call the *regale_xxx_init* (where *xxx* stands for n.m., j.m., monitoring, etc.) method, which creates the publisher/subscriber pairs necessary for the communication and returns a regale_handler. One may specify a partition here (for example, the hostname of a compute node) to communicate only with the servers belonging to that partition; however, wildcards (*) are also available for the partition option.
- Call the *specific functions/API* of the particular agent. For example, retrieving the current power consumption of Node Manager servers, or sending telemetry to Monitor servers. These functions use the handlers created by the init function to filter who receives the messages.
- Call the *regale_xxx_finalize* method, which given a handler created by the init function, destroys the publisher/subscriber pairs and cleans up.

Regarding an agent which intends to act as a server side of the REGALE library, the steps to follow are the following:

- call the *regale_xxx_service_init* method, which creates the publisher/subscriber pairs necessary for the communication. The partition can be specified (for example, the hostname of a compute node) so that clients may use it as a filter.
- Call the *specific server implementation functions/API*, which are a set of functions that will be called when requests are received. They are defined by the spec.

  Some functions may require that certain structures are filled (like GET_INFO requests) to be returned to the clients, while others are sending information to be processed (like telemetry data being sent to the Monitor).

  The list of functions varies by the component, and can be found in *regale_xxx_server_impl.h*.

- Call the *regale_xxx_service_finalize*, which deletes the publisher/subscriber pairs and stops processing messages from the clients.

REGALE software stack layer

Here we report a schematic example of a proof of concept, to show how the interactions among the implementations of a Node Manager, a Job Manager and a Monitoring system (in the order EAR, COUNTDOWN, EXAMON, see Figure 12) take place, using the REGALE layer.

Moreover, this proof of concept will be compared to Integration Scenario #2", to underline the improved handling of the REGALE library, with respect to specific modifications done for each of the interactions needed, for the implementations which are at stake.

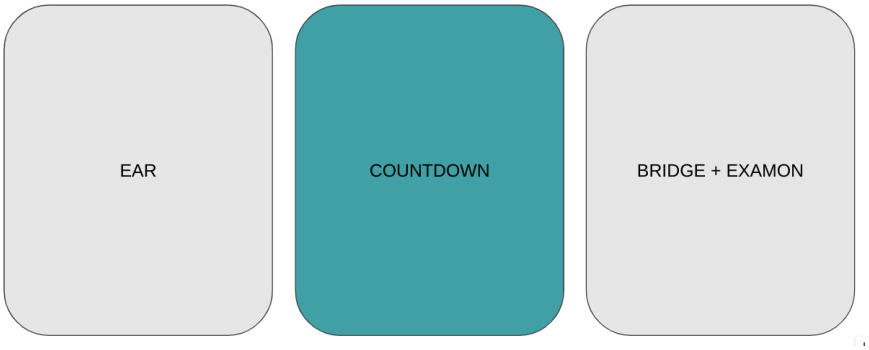


Figure 12: Schematic representation of the implementations of the entities involved in this section.

The **BRIDGE**, reported for EXAMON, is a specific extension to exchange data among the two different protocols used in the REGALE core and in the EXAMON internals. The first one is DDS, while the second one is MQTT. The BRIDGE enables the data received from DDS publishers, to be sent to an MQTT broker respecting the data model of EXAMON. This extension is specific to EXAMON, but it can be used by other Monitoring systems based on the MQTT protocol, of course adjusting it regarding the data models and the topics specific for each Monitoring MQTT-based system.

Moreover, it does not imply any changes to the monitoring client, residing within it 1) the server side of the REGALE Monitoring APIs, which cares of the specific callback functions called once the DDS message are received, and 2) the logic to convert and to make the information previously received available to the actual MQTT Monitoring system.

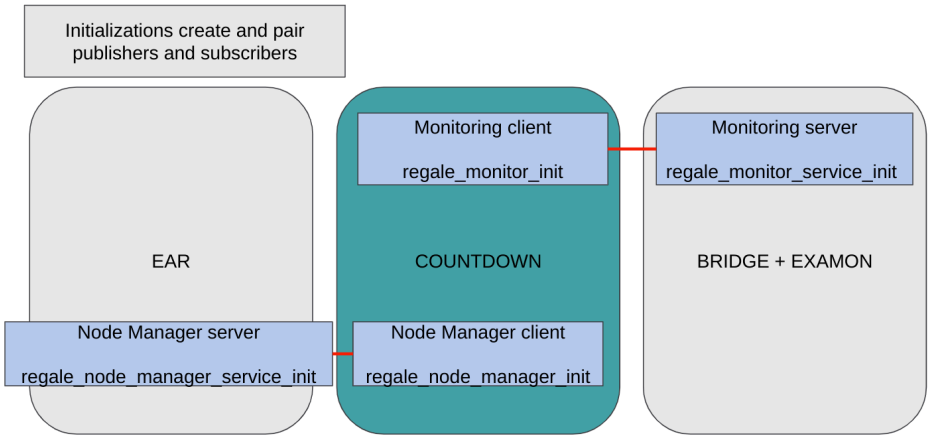


Figure 13: Initializations for the client and the server sides happen with the calling of the *regale_xxx_init* and *regale_xxx_service_init*, respectively.

The initialization phase includes the initialization for both the client (in this case COUNTDOWN) and the server (EAR and BRIDGE + EXAMON) sides (see Figure 13); publishers and subscribers are initialised and paired (the *xxx* present at the beginning of the subsection **Client/server layer** are now substituted by the corresponding entities names: node manager and monitor.

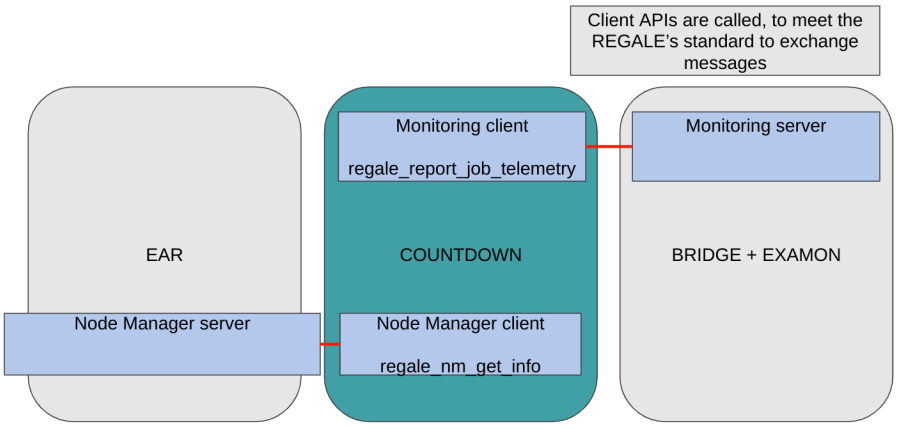


Figure 14: The client side of the Job Manager calls some specific APIs to get info from the Node Manager and to send info to the Monitoring System.

Later, the specific APIs/functions are called from the client side, to report the job telemetry and to get node related information (*regale_report_job_telemetry* and *regale_nm_get_info*, Figure 14).

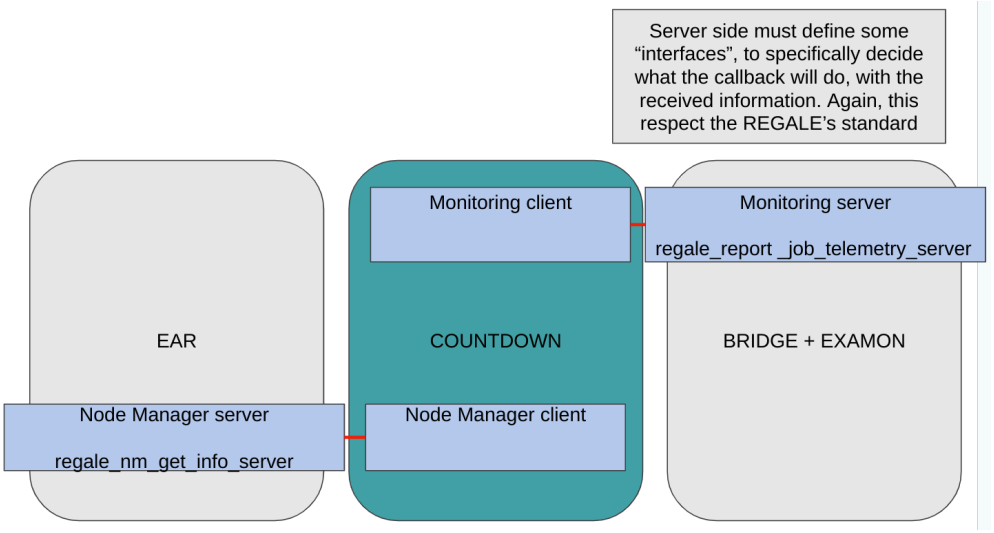


Figure 15: The callback triggered on the server side, once the client demands or sends something (see previous figure) with specific APIs, must be defined by the entities giving or using the data required or received.

For the server side, the interfaces left empty in the files *regale_xxx_server_impl.h* must be defined *(regale_nm_get_info_server* and *regale_report_job_telemetry_server*), to be able to reply to the requests of the client counterparts (see Figure 15).

What is shown here should clarify the main purpose of the REGALE library, which is the interoperability among different implementations of the entities in play. If in fact for example, the agent BRIDGE + EXAMON were replaced by the synthetic component (which actually just print the info received on the terminal), as we actually did, nothing would change in the Job Manager implementation or calls' list, nor in the Monitoring server side, other than the implementation of the *regale_report_job_telemetry_server*. But again, this must be done, to conform to the REGALE standard.

The same applies for the EAR Node Manager implementation: if plugged out and substituted by a synthetic component (also this already done), nothing would be actually required to be modified in the Job Manager operability or calls' stack.

7. Conclusions and Future Work

This deliverable provides a comprehensive review and update on the REGALE project's progress across its two primary paths: the PowerStack path aimed at power-efficient operations in supercomputing and the Workflow Engine path designed to optimize the execution of complex workflows. Significant strides have been made in both areas, evidenced by specific technical enhancements and the successful integration of crucial components.

The deliverable successfully highlighted the efforts made in updating the PowerStack path to ensure supercomputers operate with increased power efficiency. These enhancements are crucial in an era where energy conservation and sustainable computation are paramount. Furthermore, the Workflow Engine path developments have significantly improved the execution efficiency of complex, workflow-based applications. The latter has been particularly enhanced by the integration of diverse technologies such as Ryax, BeBiDa, OAR, and SLURM, enabling more seamless and optimized workflow operations on HPC resources as detailed in section 3. Complementarily, the integration scenarios presented in Section 4 offer a promising view into the potential for power-aware deployment of complex applications across supercomputing environments. In addition, the updates to the REGALE library as discussed in Section 6 ensure that stakeholders involved in the REGALE project can leverage these tools and enhancements effectively and continue to push the boundaries of what's achievable in supercomputing contexts.

Looking ahead, the project will continue to refine and enhance the integration solutions between the Workflow Engine and PowerStack paths. A particular focus will be laid on tightening the component-to-component integrations to ensure smoother orchestration and better resource management across the board. The goal will be to address any existing gaps in integration and to introduce new features that may bridge these gaps effectively.

In light of the rapid technological advancements in supercomputing and software engineering, future plans consist of exploring more advanced algorithms and machine learning models that could further optimize power utilization and workflow execution efficiency. Such innovations could potentially redefine standard operations within supercomputing environments. The power and energy capping definition in the context of workflows is also planned to be continued by better capturing the consumption and performing finer integration of the relevant REGALE tools.

Lastly, future versions of the REGALE library will be enhanced with richer features and better user documentation to accommodate the evolving needs of the supercomputing community. The ongoing updates will aim to simplify the complexity of managing supercomputer resources while maximizing performance.

8. Software Links

In Table 1, we report the link of the Git repositories of different WP3 REGALE tools and integrations.

Tools	**Integration Scenario**	**Git link**
Workflow Engine, System Manager	Ryax-BeBiDa-OAR-Slurm	https://github.com/RyaxTech/ryax-engine https://github.com/RyaxTech/bigdata-hpc-collocation

		https://github.com/oar-team/regale-nixos-compose/tree/main/bebida
System Power Manager, Node Manager	IS#1, and IS#2	https://gricad-gitlab.univ-grenoble-alpes.fr/regale/tools/beo
Resource and Job Management System	IS#1, IS#2, and IS#3	https://gricad-gitlab.univ-grenoble-alpes.fr/regale/tools/oar
Node Manager	IS#2, and IS#3	https://gricad-gitlab.univ-grenoble-alpes.fr/regale/tools/ear
Job Manager	IS#2, and IS#3	https://gricad-gitlab.univ-grenoble-alpes.fr/regale/tools/countdown
Monitor	IS#1, IS#2, and IS#3	https://gricad-gitlab.univ-grenoble-alpes.fr/regale/tools/examon_server
Monitor	IS#1, IS#2, and IS#3	https://gricad-gitlab.univ-grenoble-alpes.fr/regale/tools/dcdb
REGALE library	ALL	https://gricad-gitlab.univ-grenoble-alpes.fr/regale/tools/regale

Table 1: GIT repositories of REGALE tools