H2020-JTI-EuroHPC-2019-1

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

Grant Agreement Number: 956560

D3.2

REGALE prototype v2.0

Version: 2.0

Author(s): Federico Tesser, Daniele Cesarini (CINECA)

Contributor(s): Julita Corbalan, Shleer Qamar, Lluís Alonso Jané (BSC), Eishi Arima (TUM), Mathieu Stoffel (ATOS), Andrea Bartolini, Mohsen Seyedkazemi Ardebili, Francesco Beneventi, (UNIBO), Michael Ott (LRZ)

Date: 29.06.2023

Project and Deliverable Information Sheet

<table>
<tr><td rowspan="7">REGALE Project</td><td colspan="2">Project Ref. №: 956560</td></tr>
<tr><td colspan="2">Project Title: REGALE</td></tr>
<tr><td colspan="2">Project Web Site: https://regale-project.eu</td></tr>
<tr><td colspan="2">Deliverable ID: D3.2</td></tr>
<tr><td colspan="2">Deliverable Nature: Other</td></tr>
<tr><td rowspan="2">Dissemination Level: PU</td><td>Contractual Date of Delivery: 30 / 06 / 2023</td></tr>
<tr><td>Actual Date of Delivery: 30 / 06 / 2023</td></tr>
<tr><td colspan="2">EC Project Officer: Evangelos Floros</td></tr>
</table>

* - 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

<table>
<tr><td rowspan="6">Document</td><td colspan="2">Title: REGALE prototype v2.0</td></tr>
<tr><td colspan="2">ID: D3.2</td></tr>
<tr><td>Version: 2.0</td><td>Status: Final version</td></tr>
<tr><td colspan="2">Available at: https://regale-project.eu</td></tr>
<tr><td colspan="2">Software Tool: Google Docs</td></tr>
<tr><td colspan="2">File(s): REGALE_D3.2_ver2.0.docx</td></tr>
<tr><td rowspan="2">Authorship</td><td>Written by:</td><td>Federico Tesser, Daniele Cesarini (CINECA)</td></tr>
<tr><td>Contributors:</td><td>Julita Corbalan, Shleer Qamar, Lluís Alonso Jané (BSC), Eishi Arima (TUM),</td></tr>
</table>

		Mathieu Stoffel (ATOS), Andrea Bartolini, Mohsen Seyedkazemi Ardebili, Francesco Beneventi (UNIBO), Michael Ott (LRZ)
	Reviewed by:	Yiannis Georgiou, Georgios Goumas
	Approved by:	Yiannis Georgiou, Georgios Goumas

Document Status Sheet

Version	Date	Status	Comments
0.1	22.05.2023	Draft	Initial version
1.0	22.06.2023	For internal review	
2.0	29.06.2023	Final version	For submission

Document Keywords

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

Table of Contents

Table of Figures

1. Executive Summary

This document presents the current progress of WP3: REGALE prototyping. The primary objective of REGALE is to integrate tools and create a European software stack for power and workflow management in next-generation supercomputers. Within WP3, we will provide three deliverables that contain the outcomes of specific tasks. By utilising the tools developed by our partners (additional details in WP1-D1.1/WP1-D1.2), we have identified three integration scenarios that address the requirements already outlined in WP1. This deliverable builds upon the prototype implementation described in D3.1 by introducing an additional middle layer that facilitates interoperability among the various REGALE tools.

This deliverable focuses on three main work topics: i) component-to-component integration, ii) extensions of scenarios, and iii) 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 more 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. WP3 focuses on the PowerStack path which aims to prototype a software stack to enable full-scale production-grade solutions for holistic power management of a supercomputing system.

This deliverable extends the prototype implementation defined in D3.1 by proposing a middle layer to support the interoperability of the different REGALE tools. Indeed, when implementing the integration scenarios we faced the challenge of connecting the different REGALE tools. Two approaches have been taken:

1. Component-to-component integration
2. Middle layer incorporation

In D3.1 we proposed an integration of the different tools by mean of restructuring the codes to insert interfaces between the tools:

1. COUNTDOWN - EXAMON
2. EAR - EXAMON
3. COUNTDOWN - EAR
4. EAR - OAR
5. BEO - OAR
6. EAR - DCDB

The issue with this approach is that the number of interfaces to be supported increases exponentially with the number of tools, with a growing need of an interoperability layer which allows to abstract the communication from the policy integration. For these reasons in D3.2 we also evaluated the opportunity of having a unified middle layer approach. Different software approaches were considered such as callback [20], virtual table [21], and different communication frameworks with robust inter-agent messaging. In this regard, a promising communication framework is Data Distribution Service (DDS) [10], which is fully distributed, enables low-latency messaging, and provides configurable Quality of Service (QoS) profiles that customise messaging parameters. Moreover being brokerless, DDS, allows flexible communication topologies between the REGALE actors (node, job, system managers, etc) which executes in different parts of the ICT infrastructure.

In this document we introduce the REGALE library, which is a standard communication layer that will be used by all REGALE tools to agnostic communicate among them in order to integrate tools in a loosing couple fashion. This allows the REGALE Power Stack to be easily extensible and not depending on a specific tool, but it would be possible to expand in future with new tools. The REGALE library is built around the eProsima FastDDS communication framework.

The document is structured as follows. In Chapter 3, we describe the component to component integration. In Chapter 4 we discuss an extension of the integration scenarios which can be covered by the component to component integration. In Chapter 5 we present the initial version of the REGALE library and how it is implemented, furthermore, we will provide an initial extension of the tools with the REGALE library.

3. Component-to-component Integration

In this chapter, we describe all the progress done on the integration for each tool. The chapter is organised in sections where each section is focused on a specific component-to-component integration. We describe the effort and the mechanisms that we use in the integration to support a common SW stack. We also describe the extension developed for each tool in order to be integrated in the REGALE SW stack.

COUNTDOWN - EXAMON

EXAMON (Exascale Monitoring) is a lightweight monitoring framework for supporting accurate monitoring of power/energy/thermal and architectural parameters in distributed and large-scale high-performance computing installations. EXAMON is composed of different layers, each of them with multiple components. The integration of different data sources is handled by the modular nature of the infrastructure, where new components can be added seamlessly provided that they respect the correct data formats. The cornerstone of EXAMON is the middleware layer provided by MQTT brokers[1], which are the receivers of the data generated by the low-level plugins. On top of the MQTT brokers lies the data storage layer, where the data is uniformly formatted. From the storage layer the data can be fed to the high-level applications layer, by exploiting a client that exposes the underlying data collected and stored within EXAMON.

On the backend of EXAMON are located the sensor collectors, which are low level components responsible for reading the data from the several sensors scattered across the system. Once collected, the info is delivered by them, in a standardised format, to the upper layer of the stack (the frontend component of EXAMON).

These sensor collectors are then organised in two parts: the first one is the **MQTT API**, while the second one is the **Sensor API** object. As their names suggest, the former implements the MQTT protocol functions, while the latter deploys the custom sensor functions related to the data sampling; and this one, unlike the MQTT one (common to all the sensor layer), is specific for each kind of collector. For example, we can distinguish collectors that have direct access to hardware resources like IPMI, PMU units in a CPU, and collectors that sample data from other applications (i.e Ganglia and Nagios) and batch schedulers (i.e. Slurm).

[1] MQTT Approach: The MQTT (Message Queuing Telemetry Transport) protocol is a lightweight messaging protocol designed for efficient communication between devices in constrained environments. It follows a publish-subscribe model, where devices act either as publishers or subscribers. Publishers send messages, known as "publishing," to a central broker, while subscribers express interest in specific topics and receive relevant messages. The broker acts as a mediator, routing messages from publishers to interested subscribers. This decoupled approach enables scalable and efficient communication, as publishers and subscribers are unaware of each other, reducing dependencies and allowing flexible device connectivity [1].

COUNTDOWN is an open-source runtime library that is able to identify and automatically reduce the power consumption of the computing elements during communication and synchronisation of MPI-based applications. COUNTDOWN saves energy without imposing a significant performance penalty by lowering CPU power consumption only during waiting times for which performance state transition overheads are negligible. This is done transparently to the user. Since COUNTDOWN targets performance-neutral energy savings, its goal is to avoid performance penalties for a large set of MPI-based applications. Thus, COUNTDOWN focuses on saving energy only when this has no effects on performance.

Moreover, COUNTDOWN can extract traces from the underlying running applications, always with negligible overhead, letting also the user decide some raw events to be examined, adding them to the default analysed one.

All the information obtained from COUNTDOWN then, related to the computational, communication, energy and user defined aspects, can be used as a runtime instrument to understand how well the application is doing, and can increment the database gathered by EXAMON. But to let the users interpret all this info, a more human readable approach than the extracted traces is needed, preferably graphic and interactive. And it is in this path that the work explained in this subsection fits.

In order to achieve a successful component-to-component integration of the COUNTDOWN and EXAMON tools, we took several steps to ensure that important metrics were being accurately reported. One of the key steps we took was to update COUNTDOWN so as to send metrics to EXAMON using the MQTT publish/subscribe approach.

These metrics included average frequency, power, and energy per package and DRAM, as well as MPI metrics like MPI call counts for functions such as *mpi_barrier*, *mpi_cast*, *mpi_send*, *mpi_recv*, *mpi_reduce*, *mpi_wait*, *mpi_comm_size*, *mpi_isend* and more.

We defined topics for COUNTDOWN metrics on the EXAMON side. By leveraging the EXAMON data model, we were able to seamlessly add any new data stream to EXAMON with a predefined topic, using the MQTT publish-subscribe method.

In EXAMON, the

collector2TimeSeriesDB(<metric>,<timestamp>,<key0>/<value0>,<key1>/<value1>,..)
interface is used for storing the telemetry data which are collected from COUNTDOWN in NoSQL time series database. It provides a data insertion mechanism, a bridge between the collector (MQTT protocol) and the time series NoSQL DB (KairosDB).

Following the update of COUNTDOWN and EXAMON, we now have all of the monitoring data from COUNTDOWN in EXAMON. To access and visualise the data, EXAMON provides various interfaces, such as Examon's SQL-like querying language, RESTFul API, and Cassandra Query Language Shell (*CQLSH*). With the CQLSH, users can execute Cassandra Query Language (CQL). In addition, Examon has two web user interfaces: Grafana and KairosDB user interfaces.

COUNTDOWN-EXAMON Dashboard: The EXAMON Dashboard (which uses Grafana [11]) is a powerful tool that allows EXAMON users to create personalised dashboards tailored to their specific needs. In addition to the customizable options, we have developed a dashboard that showcases the key features and capabilities of the COUNTDOWN.

Figure 1 provides a set of snapshots of the EXAMON dashboards, highlighting its ability to visualise crucial metrics such as average frequency, power, and energy collected from the COUNTDOWN. By presenting this information in an intuitive graphical format, users can easily monitor and analyse the performance of the COUNTDOWN.

To enhance the analytical capabilities of the dashboard, we have incorporated additional metrics collected from the IPMI plugin. This integration provides valuable insights into the applications running on the system and enables users to assess the overall performance. By combining these metrics, users gain a comprehensive understanding of the system's behaviour and can make informed decisions regarding optimizations or resource allocation.

One of the standout features of the COUNTDOWN-EXAMON Dashboard is its inclusion of various maps that facilitate correlation analysis. These maps depict the relationship between frequency and MPI rank, as well as MPI call time and count with MPI metrics (like mpi_barrier, mpi_cast, mpi_send, mpi_recv, mpi_reduce, mpi_wait, mpi_comm_size, mpi_isend). By visually representing these correlations, users can identify patterns, bottlenecks, or anomalies in the job's behaviour over time. The sampling rate is set to one-second intervals.

Up until now, communication between the COUNTDOWN and EXAMON systems has been primarily one-way, with data flowing from COUNTDOWN to EXAMON. However, we have actively explored alternative approaches to enable bidirectional communication between the systems. Our investigations have led us to consider multiple options, including Virtual Tables, Callbacks, and the utilisation of libraries like Variorum[2] [2] for Node Manager Wrapper. We have also explored the use of ROS2, a widely adopted communication middleware primarily used in robotics. ROS2 leverages high-reliability DDS and RTPS (Real-Time Publish-Subscribe) protocols, both of which are field-proven and ensure robust and efficient communication [3].

In our pursuit of making the COUNTDOWN system MQTT compliant, we have explored the feasibility of developing an EXAMON and COUNTDOWN agent. This agent would enable seamless integration with MQTT-based communication protocols. As part of this exploration, we have investigated FastDDS, a high-performance implementation of the DDS standard. The adoption of FastDDS not only enables MQTT compliance for the COUNTDOWN system but also offers significant advantages for the REGALE middleware [4].

[2] Variorum is an extensible, vendor-neutral library for exposing power and performance capabilities of low-level hardware dials across diverse architectures in a user-friendly manner. It is part of the ECP Argo Project, and is a key component for node-level power management in the HPC PowerStack Initiative [2].

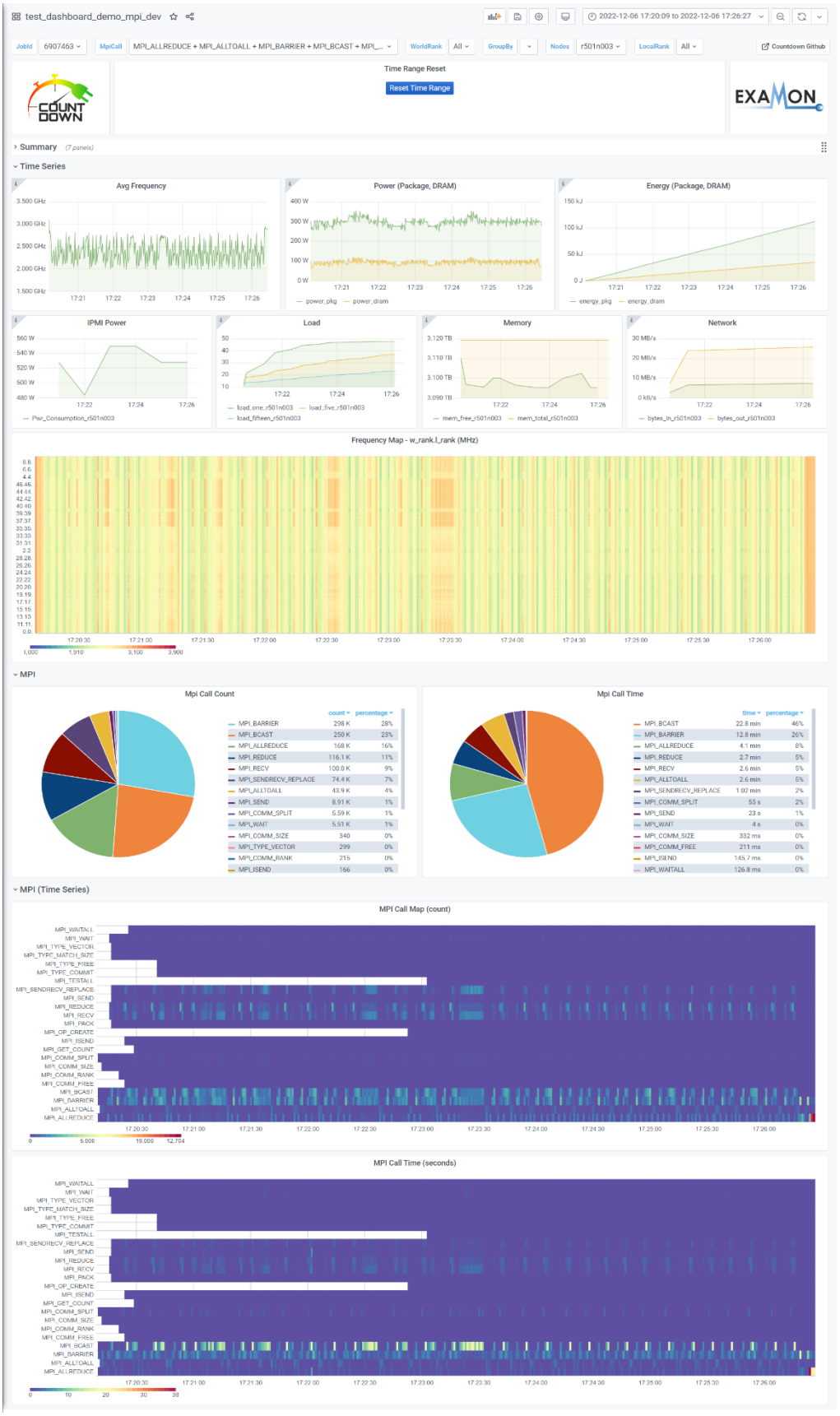

Figure 1a: Dashboard of the information obtained both by COUNTDOWN and EXAMON integration, containing information (composition of the MPI calls, power consumption, etc.) on the job collected at execution time.

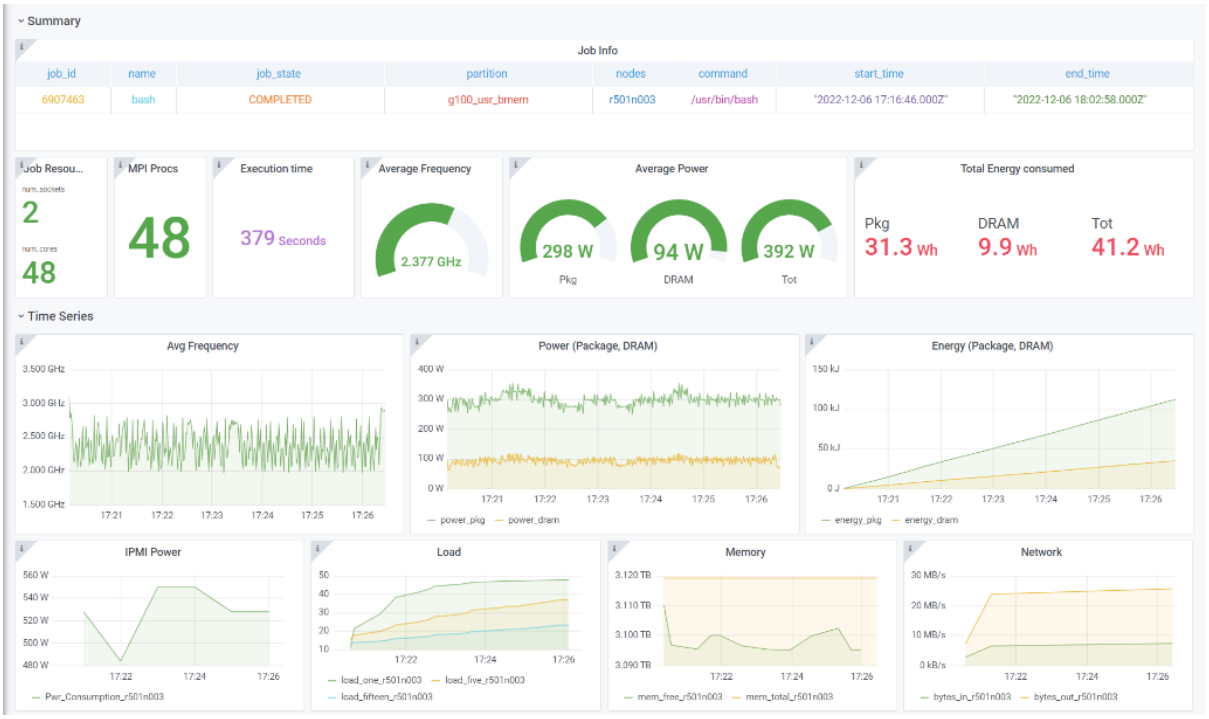

Figure 1b: We reported here COUNTDOWN information like "number of cores used", "number of nodes used", "number of MPI ranks", "total execution time", "average frequency, power and energy of the job" of the application. This information is given at execution time, so the user can see changing plots in real time. Moreover, additional information obtained by EXAMON sensor collectors are reported: "IPMI power", "load", "memory and network usage", etc.

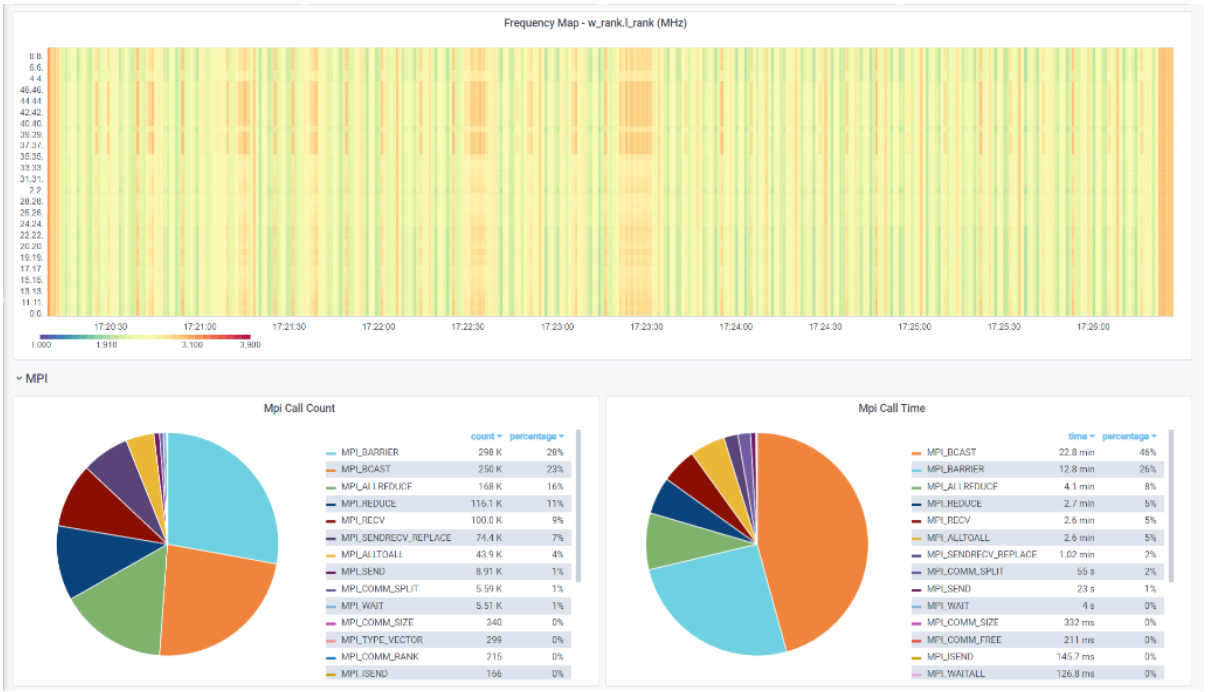

Figure 1c: COUNTDOWN-EXAMON dashboard where, at the top, we can see the heatmap of cores' frequency, the information reported is real time. At the bottom of the figure there is a summary of the MPI calls count and time of the job. Data is collected by COUNTDOWN at the execution time and updated by Examon every 10s.

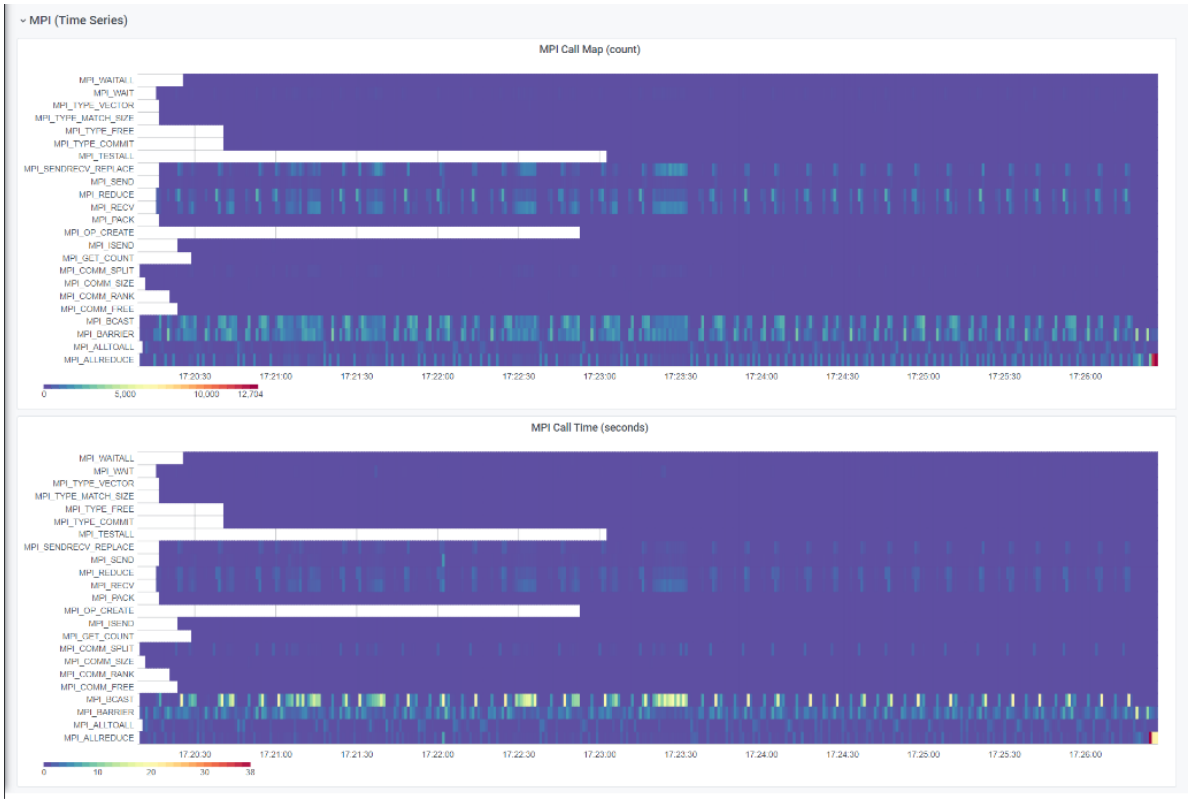

Figure 1d: Heatmap showing the count (at the top) and duration (at the bottom) of each MPI call (y-axis) during the execution of a job (x-axis).

EAR - EXAMON

EAR is a system software for energy management, monitoring and optimization. These services are provided by different EAR components. For example, the EAR Job manager (or EAR library) provides application monitoring and optimization. The EAR Node Manager (or EAR daemon) provides node monitoring, job accounting and node powercap. Given it also provides node monitoring, it partially executes actions associated with the Monitor in REGALE terminology. The EAR database manager complements the monitoring actions together with the node manager. Finally, the EAR System power monitor (EAR Global Manager) provides system monitoring and cluster powercap. During this period we have been working in the integration of individual EAR components with other tools in REGALE. In this section, for example, we have implemented the strategy to report the EAR node manager and EAR library data through EXAMON. We will also present how to report EAR data through DCDB. These integrations will allow, for example, to extend data already collected by EXAMON (or DCDB) with EAR collected data. This data will include data from running jobs both in average and at runtime. In the next section we will also show how COUNTDOWN can co-exist with the EAR Node Manager, taking advantage of the node power cap for example. Finally, we have also identified the requirements to interact with any scheduler and we have implemented the integration with OAR.

The information we provided regarding EXAMON at the COUNTDOWN - EXAMON component-to-component integration, such as the EXAMON MQTT approach and the

collector2TimeSeriesDB and dashboard, is also applicable here. Therefore, we will not repeat the relevant information.

EAR provides, among other energy/power services, job accounting and power monitoring focusing on power and application performance. EAR collects metrics in two granularities: loop and application. Loop corresponds to a piece of code executed in a repetitive way. One application can have one or multiple loops (iterative regions). EAR reports performance and power metrics periodically for each loop. For each loop, EAR reports the set of "iterations", *jobid*, *stepid* and *nodename*. For each of this set of iterations EAR measures the Loops Signature and reports its data. This is a set of metrics computed during the application execution in short intervals (default value 10 sec.). EAR also reports the Application Signature, the same set of metrics used at the loop level but applied to the granularity *jobid-stepid-nodeid*.

EAR includes a reporting mechanism based on plugins with a simple API, one for each EAR type. The following shows a subset of the report API.

```
 /*Report EAR Data Types*/
state_t report_applications(report_id_t *id, application_t *apps, uint
count);
state_t report_loops(report_id_t *id, loop_t *loops, uint count);
state_t report_events(report_id_t *id, ear_event_t *eves, uint count);
state_t report_periodic_metrics(report_id_t *id, periodic_metric_t *mets,
uint count);
state_t report_misc(report_id_t *id, uint type, const char *data, uint
count);
```

For the integration of the EAR tool with the EXAMON framework, we developed a new EAR report plugin, named *examon*. This report plugin was built in three stages. The first stage to collect EAR data by implementing the EAR report APIs of *report_loops*, *report_periodic_metrics* and *report_events*, and the second stage was to reformat the collected data into an acceptable format by the EXAMON data model format, and then in the final stage publish the data into the ExaMon broker.

Figure 2 shows the current reporting architecture. The EAR library (EARL) uses the EARD report plugin to send the collected data to the EAR Daemon (EARD). Then the EARD uses the Examon report plugin which implements the report_loops, report_periodic_metrics and report_events APIs to publish the EAR data into the ExaMon Broker using the MQTT messaging protocol.

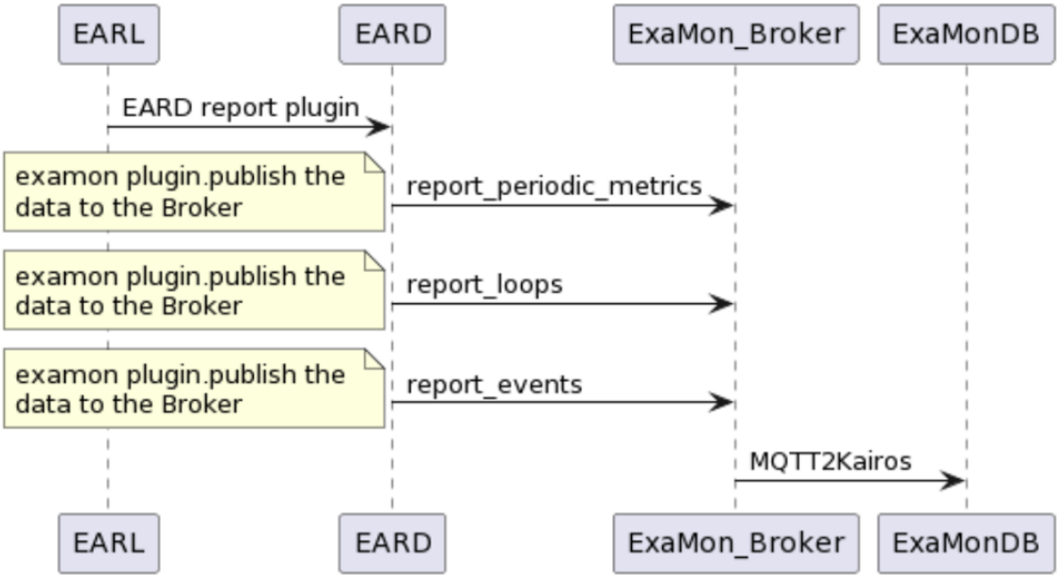

Figure 2: EAR-EXAMON Integration PlantUML Diagram

Table 1 lists the metrics that were added to the EXAMON monitoring system through EAR in the EXAMON-EAR integration.

AVG_FREQ_KHZ	Node Average Frequency
AVG_DC_POWER_W	Node Average DC Power
AVG_DRAM_POWER_W	Node Average DRAM Power
AVG_PCK_POWER_W	Node Average CPU+DRAM Power
AVG_GPU_POWER_W	Node Average GPU Power
ITERATIONS	Total Iterations
DC_NODE_POWER_W	Node DC Power
DRAM_POWER_W	Node DRAM Power
PCK_POWER_W	Node CPU+DRAM Power
EDP	Energy Delay Product
MEM_GBS	Memory
IO_MBS	Input/Output
TPI	Memory Transactions per Instruction
CPI	Cycles per Instructions
GFLOPS	gigaFLOPS
time	Time
FLOPS	Floating Point Operations per Second
L1_MISSES	Cache misses type 1
L2_MISSES	Cache misses type 2
L3_MISSES	Cache misses type 3
INSTRUCTIONS	Total Instructions
CYCLES	Total Cycles
AVG_CPUFREQ_KHZ	Average CPU Frequency
AVG_IMCFREQ_KHZ	Average Memory Frequency
DEF_FREQ_KHZ	Default CPU Frequency
PERC_MPI	MPI Percentage

Table 1: Examon metrics used for the integration with EAR

Figure 3 displays a screenshot of the dashboard we designed for EAR in EXAMON, which shows the EAR-EXAMON component-to-component integration. This figure features two metrics (Frequency and Power) of two compute nodes collected by EAR in EXAMON.

Figure 3: EAR-EXAMON Dashboard

COUNTDOWN - EAR

For the fine grained level that COUNTDOWN can reach, analysing at runtime the monitored applications (already explained in subsection COUNTDOWN-EXAMON), another interesting integration was of course the one with EAR. That in fact would let EAR have a more refined control on the power and energy levels, for the communication phases of the applications running on the systems, monitored by this NM.

One area of agreement among COUNTDOWN [5][6][[7] and EAR [8][9], was to modify the first tool to use the *Userspace* governor. COUNTDOWN by default does not consider it, working on specific CPUFreq files like *scaling_max_freq* and *scaling_min_freq* (affected by the *Performance* and *Powersave* governors). This approach is explained by the fact that COUNTDOWN does not need any special privileges to run, if it works in tandem with some *SPANK Slurm* plugins which act in the prolog and epilogue phase of the job, respectively modifying the content and the permissions of the necessary files, and reverting them back to their original state.

EAR instead needs the Userspace governor, and so it necessitates to interact with the file *scaling_setspeed* to do its job, so we needed to let COUNTDOWN understand, in a configuration phase, what is the underlying current governor and to choose on this information, on what configuration files act to apply its frequency reduction strategy.

Once done, the next step was to be able to share information among COUNTDOWN and EAR, using the *scaling_setspeed* file mutually, trying to avoid all possible conflicts that arise from a shared file with write permissions.

That is in fact the only file, using the *Userspace* governor, which modifies the current frequency of the correlated CPU (scaling_setspeed file under cpufreq driver paths). This file is used to set fixed frequency per CPU and it is used for energy optimization by scaling down the frequency (known as DVFS) and to limit power consumption by reducing the frequency when other hardware approaches are not available. EAR and COUNTDOWN use the scaling_setspeed file and, even though its is expected to do it at different granularities (COUNTDOWN in a more fine grain granularity than EAR since it acts at the mpi call level) it is easy to see that there could be a possible write contention among the two tools.

So, we decided to apply the "*flock*" approach, before writing into the selected system file, being able with this choice to both lock it in case of modification of the running frequency, and at the same time to not be too limited, in an execution time fashion, by the overhead of a mandatory lock. *Flock* in fact places advisory locks, non enforcing a full locking scheme on the specified file (or files), which can result in a more responsive mechanism, being not surrounded by all the constructs added to a mandatory lock.

But of course, it is needed both the tools being aware of the necessity to use *flock*, for the *scaling_speed* file. This last consideration means that, being a process free to ignore an advisory lock, all the participating processes must cooperate to explicitly acquire the lock, otherwise they will fall in a race condition.

The base behaviour of COUNTDOWN and EAR working together and using the Userspace governor, is the following: COUNTDOWN read the current value in the *scaling_setspeed* file, previously set by EAR and representing the current maximum frequency, and store it in an internal variable. Starting an MPI phase, COUNTDOWN will modify the *scaling_setspeed* content, setting it equal to the minimum available frequency for the underlying system, to reduce power consumption during communication phases, Once the MPI event finishes, then COUNTDOWN read again the value contained in the *scaling_setspeed* file; if it is different from the minimum one previously stored by itself, than it means that the NM (Node Manager, in this case EAR) planned to modify current frequencies as a result of its power capping logic, and then COUNTDOWN will apply silently this new value, doing nothing (except for updating the stored value of the maximum available frequency). Otherwise, it will modify the content of the *scaling_setspeed* file, reapplying the maximum value stored before entering the MPI event, to speed up the computational parts.

Hereafter a graphical representation (see Figure 4) of what was previously described: 1) COUNTDOWN stores the current maximum value for the frequency, imposed by EAR; 2) entering an MPI event, COUNTDOWN writes the new content of the *scaling_setspeed* file, placing it equal to the minimum operating frequency the processor can run at (information foundable in *cpuinfo_min_freq* file); 3) entering an APPLICATION phase, after an MPI event,

the heuristic of running at the maximum available frequency is applied by the Job Manager (COUNTDOWN) and the Node Manager (EAR).

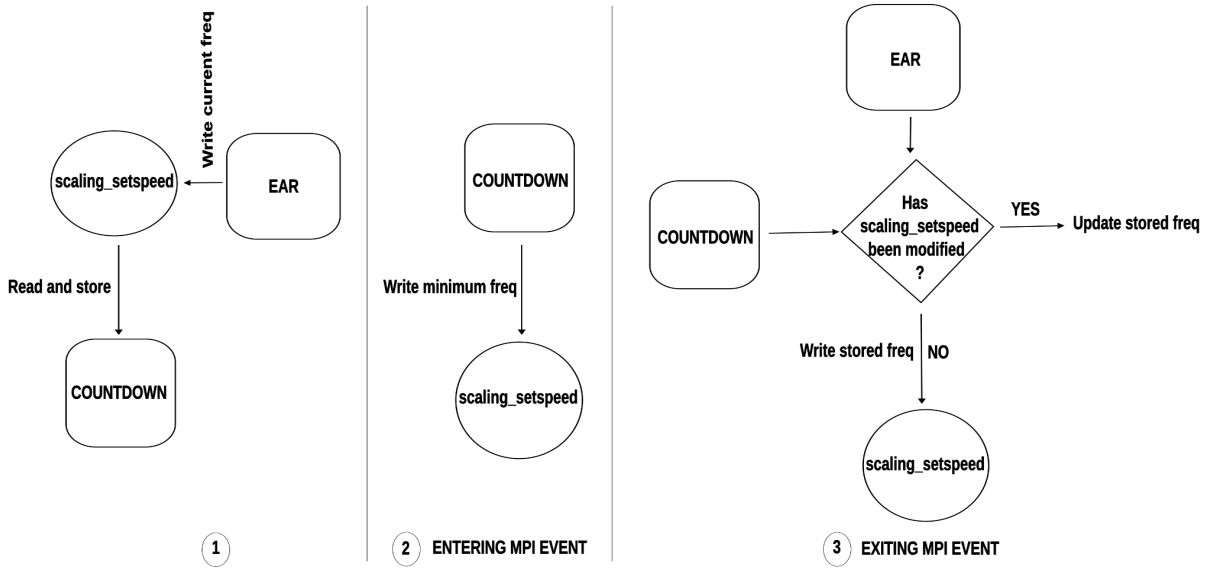

Figure 4: A scheme of the integration work done by COUNTDOWN an EAR

However in the previous approach, a problem still remain unresolved: suppose that EAR decide to set, during an MPI event managed by COUNTDOWN, the maximum current frequency equal to the minimum one; in this case, COUNTDOWN will not be able to understand that the value written in the *scaling_setspeed* file was the new one decided by EAR, and not the one set by itself, entering the MPI event. In this case the frequency will rise up, diverging from the power capping logic of the NM. A different approach like the one explained in section number 4.5, could help to prevent this unmanaged situation.

EAR - OAR

EARPlug is the EAR component related to the system scheduler. The main function of this component is to provide support to connect the scheduler with EAR, in order to get EAR working when users launch their jobs on the system. The following will describe the main workflow of EARPlug and the required background information to integrate EAR with a new scheduler.

In general terms, the scheduler must connect with the EARD and send all the information it requires to properly configure the EAR loader library. The information to send to EARD is job related, such as the job id, the user and the group who is submitting it, the number of tasks invoked, etc. On the other hand, the scheduler must inform the EARD when the job ends. Therefore, the hook/plugin must act as a job prologue and epilogue to inform the start and the end of that job, respectively. Also, the scheduler integration must properly set the environment variables required by the EARL to work with the configuration requested by the Administrator during the installation of EAR or by users when submitting their jobs.

Below is the list of the job's execution cycle events between any scheduler and the EARD, those are the general workflow to connect with EARD and send a job-related information: i) New Job/End Job, ii) New Task/End Task, and iii) New Step/End Step.

In general, the scheduler intercepts and deploys in two different contexts, those contexts referred to as local contexts and global contexts. For any scheduler integration we must consider which context the scheduler uses.

The local context is used to identify events triggered when the job is still in the node where it is submitted, e.g., job just submitted, job queued, job sent to compute nodes. The remote context refers to events triggered at compute nodes, e.g., the compute node just received the job, the job's top shell run just started, application tasks just invoked.

The task workflow the scheduler integration is expected to follow according to what is explained in the section above. For each of the events the *EARPlug* must report to EARD what to do at such phase, like which kind of information must be collected, at which context the operation must be performed, etc.

- *New job event*: This is the first phase of the hook/plugin, where it reports a new job starting in the compute node where EARD is running. This phase is performing the remote context. The list of steps required at this stage:
    1. Fill in the application/job information.
    2. Inform EARD about the start of a new job with its information.
    3. Update the configuration based on what EARD returns.
    4. Set the required environment variables. These are the minimum required:
        a. EAR_INSTALL_PATH
        b. EAR_ETC
        c. EAR_TMP
        d. EAR_VERBOSE
        e. LD_PRELOAD

- *End job event*: This last phase is called when all tasks related to a job (or last step) end. This phase is performing the remote context.
- *New step event*: If your scheduler is working with steps, then at this phase it needs to communicate with EARD the creation of a new step. The communication process is very similar to the new job event with the step related info.
- *End step event*: Analogous with New step event. These two events must be performed for each node where the step is executing (remote context).
- *New task event*: This event is triggered by the scheduler once per task created, on each compute node (remote context).

For the OAR-EAR integration, the first approach that was followed and tested was manually using the available EAR APIs for prolog/epilog messages: new job, end job APIs using *ejob*

tool, where the OAR tool was modified to prolog/epilog with those EAR command APIs using the mentioned EAR tool, also the OAR tool was manually setting all the needed environment variables for the EAR to load successfully. Figure 5 shows the integration approach that was used using the EAR *ejob* tool and manually setting the EAR environment variables.

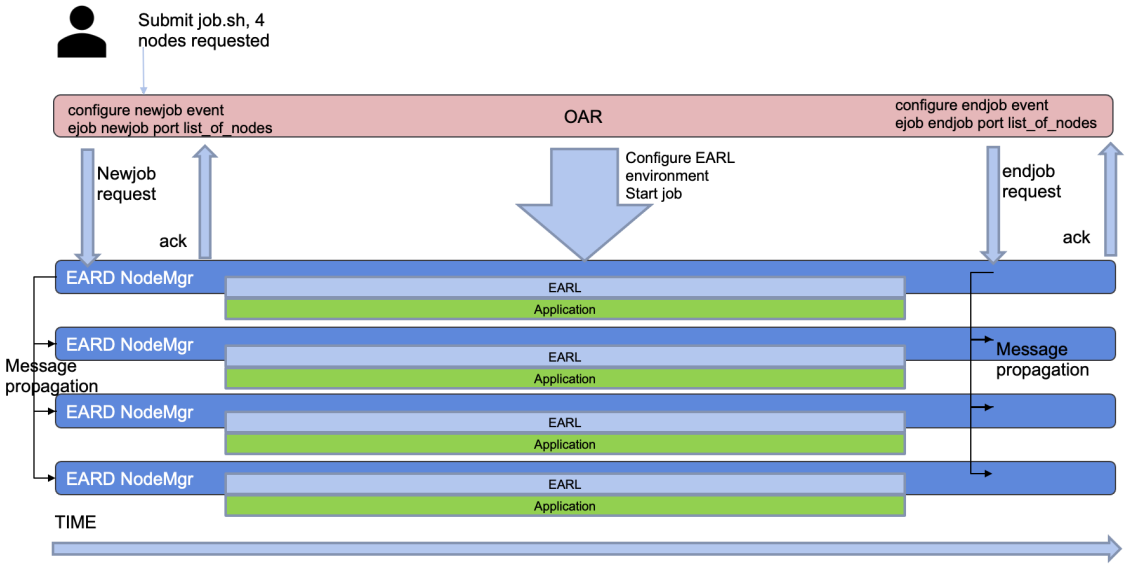

Figure 5: OAR-EAR First Approach Integration

The second approach that we followed for this integration is more dynamic and it has been tested with the new EAR release successfully, following the same concept that was explained in the first approach of OAR scheduler communicating with EAR the start and end of the jobs while setting the needed environment variables, however using a single EAR tool called *erun*, where it's not needed to use the prolog/epilog messages or manually set the environment variables anymore.

The first scheduler integration implemented in the EAR program was a SLURM spank plugin. In order to support other schedulers or other scenarios, we created the *erun* command [12]. This command replicates all the SLURM and EAR SLURM Plug-in communication pipeline. It is the official EAR tool to replace all the steps done by the SLURM pluginit does the *new_job*/*end_job*/*new_task* communication and sets the environment variables. It can be used to set up EAR at job submission.

erun will simulate on the remote node both the local and remote pipelines for all created processes. It has an internal system to avoid repeating functions that are executed just one time per job or node, like SLURM does with its plugins.

BEO - OAR

In the context of the first integration scenario, a new feature is to be implemented in Bull Energy Optimizer (BEO): a mechanism to automatically adapt the power capping enforced on

jobs, depending on the characteristics of the latter. To implement this mechanism, called Application-Aware Power Capping (AAPC), BEO needs information about the state of the managed partition of compute nodes, notably regarding the jobs. That is why BEO is intended to be integrated with OAR through a prologue and an epilogue, whose implementation depends on the design of the AAPC mechanism. The architecture of this integration is presented in the first sub-section.

However, the first experiments associated with the implementation of the AAPC mechanism in BEO exhibited a counter-intuitive and counter-productive phenomenon: when a compute node is under a heavy load (e.g. FireStarter [16]), a high latency tends to exist between the moment a power cap is set, and the moment it is effectively enforced (e.g. several minutes). If this observation is confirmed, the design of both the AAPC mechanism and the integration between BEO and OAR should be modified. That is why the implementation of the prologue and epilogue for the integration of BEO with OAR was put on hold. Additionally, an experimental evaluation of the latency associated with the enforcement of a power cap on an HPC compute node was initiated, and is presented in the second sub-section.

As a final note, we mention that an integration based on a prolog and an epilog was already performed between Bull Dynamic Performance Optimizer (BDPO), another tool developed by AtoS, and OAR in the context of the WP2 of the REGALE project. Thus, working skeletons for both the prologue and the epilogue related to BEO and OAR were implemented, based on this previous work.

Integration of BEO with OAR regarding the Application-Aware Power Capping (AAPC) mechanism

To begin with, let us specify the use case to be addressed by the AAPC mechanism. In a few words, it is based on the following empirical observation: HPC applications exhibit a wide range of workloads, which all have different sensibilities to power capping regarding performance. Indeed, on the one hand, some applications/use-case pairs, for instance NEMO TOP/PISCES solver applied to GYRE [25], tend to be heavily memory-bound. As a consequence, they do not constantly require the computing cores to consume their whole nominal power budgets to execute the workload associated with the application. Thus, when considering the average behaviour of the application, the nodes which execute it can be constrained by a power cap with only moderate to no impact on the performance of the application. On the other hand, the execution of applications which are much more compute-bound, such as HPL [26], highly and almost permanently stresses the computing cores. As a result, decreasing the raw computational power exhibited by the nodes, for instance by enforcing a power cap on them, induces a significant performance degradation. That is why compute-bound applications tend to be greatly sensitive to power capping.

Based on those experimental observations, the rationale associated with the AAPC mechanism consists in leveraging knowledge about the applications executed on a partition of compute nodes at a given time to dynamically redistribute the power budgets allocated to

the jobs. The goal is to favour compute-bound jobs so as to increase the job throughput of the considered partition, when compared to the standard Fair Sharing Power Capping (FSPC) strategy. Indeed, by trying not to power-constraint the nodes executing compute-bound jobs, heavy performance degradations for the associated applications could be avoided, which should tend to increase the job throughput for the considered partition. On the contrary, the shift of power budget from nodes allocated to memory-bound jobs to nodes allocated to compute-bound jobs should tend to increase the performance degradations for the applications associated with the former jobs. This will translate to a decrease of the job throughput for the considered partition. However, as explained beforehand, the avoided performance degradation for compute-bound jobs should positively counterbalance the induced performance degradation for memory-bound jobs. Hence, on average, at the scale of a power-constrained partition of nodes, the job throughput should be increased by the AAPC mechanism when compared to the FSPC strategy.

Overview of the architecture of the solution

An overview of the software architecture associated with the implementation of the AAPC mechanism, and of how it integrates in the management stack of a supercomputer is presented by Figure 6. The AAPC mechanism consists of two main components, whose interactions are described by the sequence diagram presented by Figure 7:

- An AAPC extension plugin for the Resource and Job Management System (RJMS) - in the context of REGALE, the RJMS is OAR;
- An AAPC module for BEO.

The role of the former is to notify the latter for two kinds of events, namely the start and the termination of jobs. The notifications are accompanied with additional information about the concerned job: its name and ID, the list of nodes allocated to the job, and the tag of the application associated with the job (more details about the tags in the next subsection). The AAPC module for BEO can thus build and maintain an internal representation of the state of the partition of compute nodes it manages regarding power capping. Incidentally, the power budget for the aforementioned partition is specified by the administrator of the supercomputer and is used as input by the AAPC module for BEO. Thanks to those pieces of information, it is possible for the AAPC module for BEO to compute a sharing of the power budget between the nodes, and to update it at each job start or termination. Using the features implemented by the core engine of BEO, it is then possible to dynamically create, update and enforce power capping rules on the compute nodes. It should also be noted that in case of deactivation of the AAPC mechanism, the site-specific default power capping rules should be enforced again.

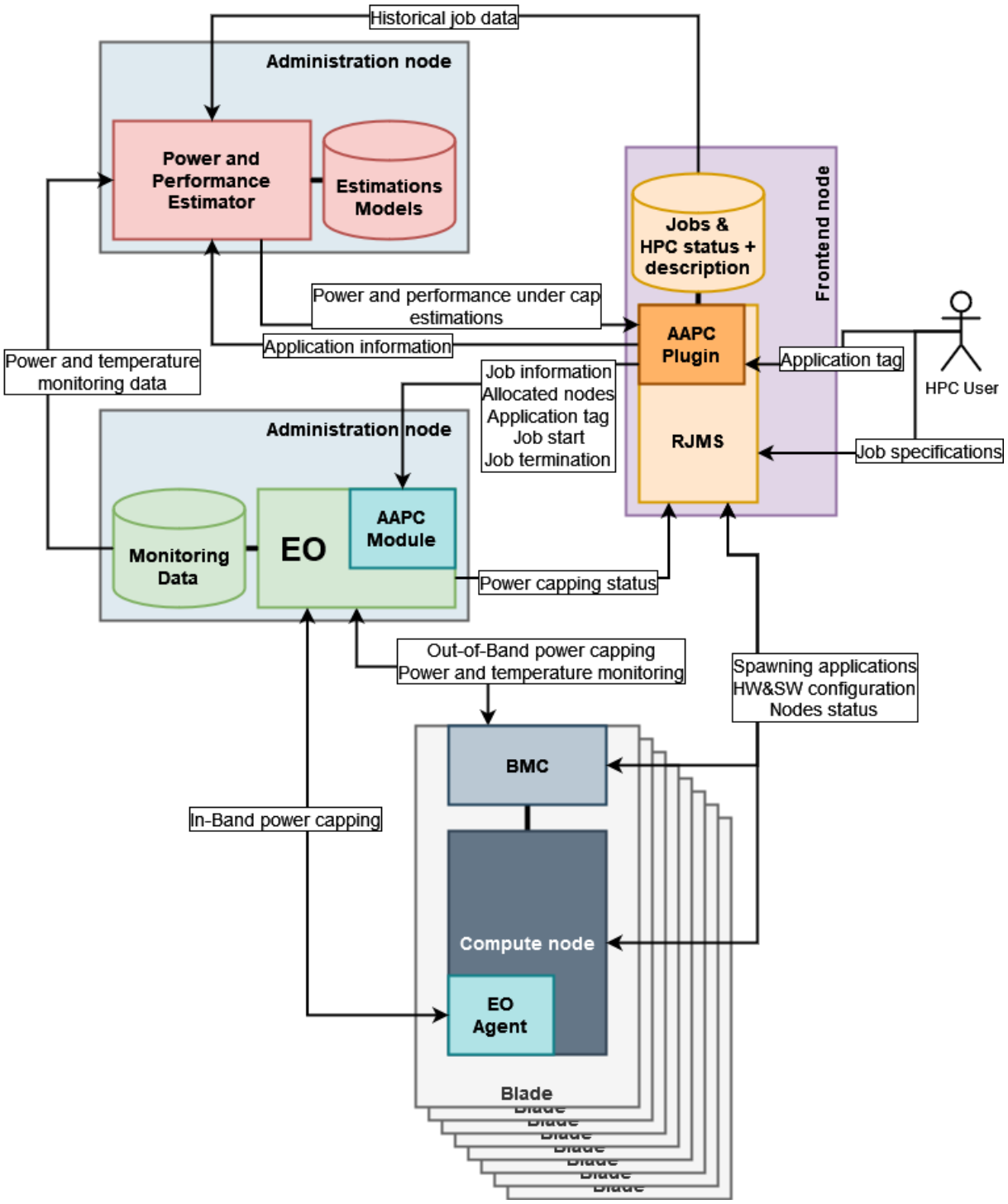

Figure 6: Overview of the architecture of the AAPC mechanism, and of its integration in the management software stack of a supercomputer.

Another remarkable component presented by Figure 7 is the Power and Performance Estimator (PPE), to which a database system storing performance models it built and estimations it performed is joined. Its role is to estimate the power consumption and performance of an application and to evaluate the impact of a power constraint on the latter, based on temperature and power consumption monitoring data of previous executions

and/or of executions of similar applications. Those estimations are then sent to the AAPC extension plugin of the RJMS so as to attribute a tag to the job.

Elements about the application tags

To start with, let us define what application tags are. In a few words, as explained beforehand, HPC applications exhibit a wide range of workloads which can be classified according to several different taxonomies. The one used in the context of this work about the AAPC mechanism defines three categories of applications:

- *COMPUTE tag*: The application is mainly compute-bound, and hence its performance tends to be heavily degraded under power cap;
- *MEMORY tag*: The application is mainly memory-bound, and hence its performance tends to be lowly degraded under power cap;
- *MIXED tag*: The application exhibits several interlaced behaviours, and hence the impact of a power cap on its performance tends to be moderate on average.

From the point of view of the AAPC mechanism, the tag associated with an HPC application defines a soft relative lower bound on the power constraint to be applied to a node executing the latter. Relative, since the associated power cap is defined as a percentage of the nominal power consumption of the node. Soft, because the AAPC mechanism tries to find a sharing of the power budget allocated to the partition which makes it possible for each individual power cap enforced on the compute nodes to be compliant with the tags associated with the jobs they execute, but it can set lower power constraints if such a sharing does not exist. On top of that, the application tags are used to build a priority order regarding nodes to be power-capped: nodes executing an application tagged as MEMORY should be power-constrained before nodes executing an application tagged as COMPUTE.

Furthermore, application tags are related to the PPE component. Indeed, the goal of the PPE is to estimate the power consumption and performance level of an HPC application and the impact of a hypothetical power cap, based on previous executions of the concerned application, and executions of similar applications. Those estimations are then to be used by the AAPC extension plugin of the RJMS to associate a tag with the considered execution of the application. Note that PPE will probably be implemented in a second step of development of the AAPC mechanism. As a result, in a first step, AAPC will most probably rely on user input regarding application tags.

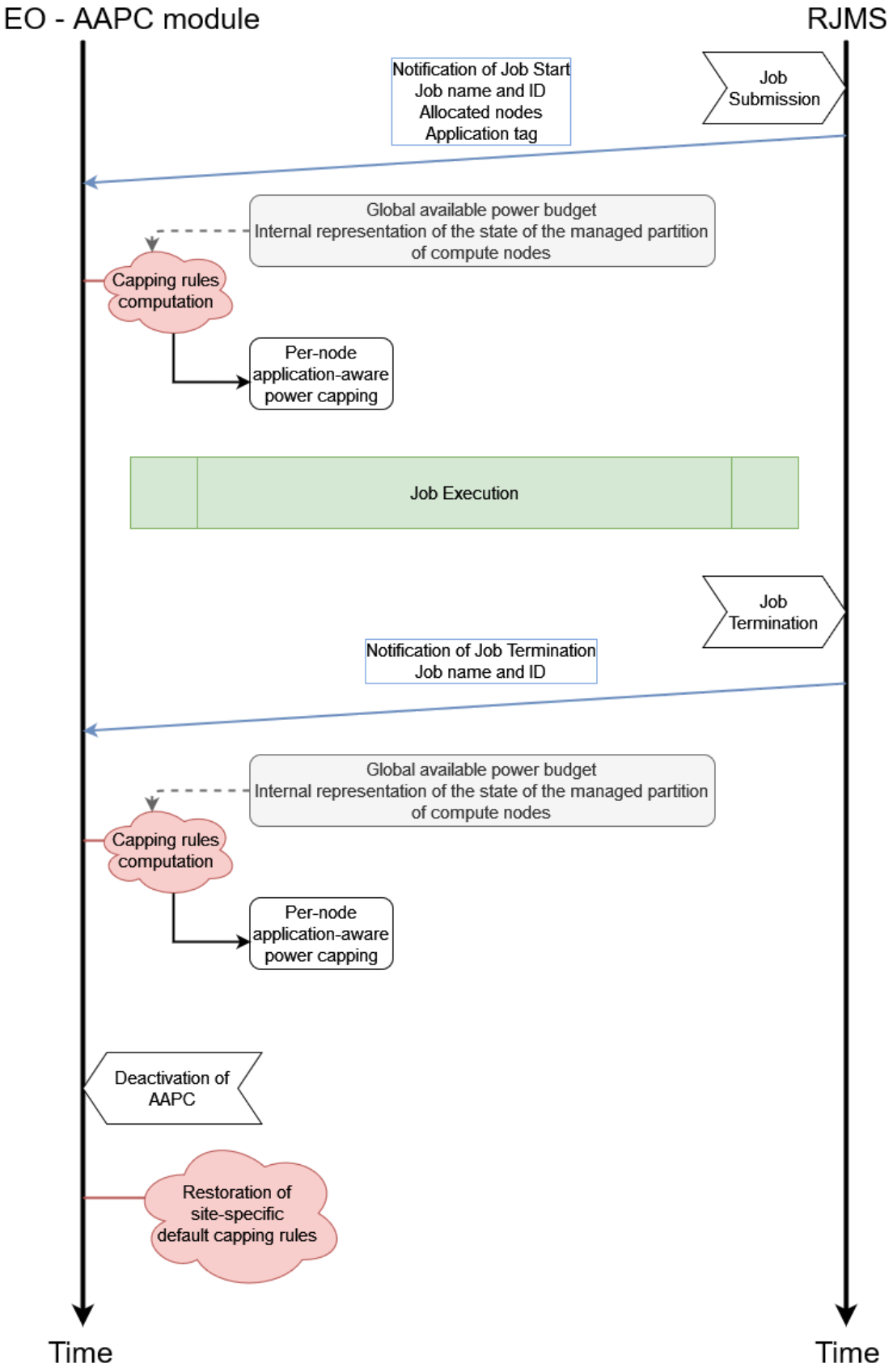

Figure 7: Sequence diagram describing the interaction between RJMS, and the AAPC module of BEO.

Experimental evaluation of the latency associated with the enforcement of a power cap

Anecdotal evidence has shown that in some circumstances administration commands to adjust processor power consumption are not always taken into account, i.e. there is no change in frequency even after a delay of several minutes. This behaviour has been observed on systems with x86-64 processors from both AMD and Intel. The purpose of this study is to better understand the operation of processor power management commands across a range of HPC compute nodes.

Experimental protocol

The following factors, and when applicable the associated ways of varying them, were identified as having the potential to influence the success or failure of a capping operation:

- *Management interface*: In-band (RAPL) or out-of-band (IPMI or Redfish);
- *Capping direction*: From high-power cap to lower power cap and vice versa;
- *Size of the capping step in Watts*: cap from 600W to 300W in a single step or through 10W decrements;
- *The order of capping operations*: (1) setting a cap threshold and then activating the capping system, or (2) activating the capping system and then changing the cap power level, or (3) activating the capping system, setting a new power level and sending a second, nominally redundant, activation command;
- *The level of CPU load*: starting from 100% and progressively reducing the load. This is done through two distinct mechanisms. First by progressively reducing the workload across all online processors equally, and then by running all processors at 100% and decreasing the number of running threads. Additionally, for the case with decreasing load on all cores, the period over which the average is calculated is increased. For example, with a load of 99%, calculated over a period of 10 ms, there will be an idle period of 100 µs, whereas the same load calculated over 1s will have an idle period of 10ms;
- *The processor characteristics* (e.g. model, family, step, ...);
- *The environment* (e.g. version of the Linux kernel, firmware versions, ...).

On a single system, where processor type, firmware and possibly management interface are fixed, the other factors are permuted resulting in a lower threshold approaching 1,000 tests to execute per system (i.e. (1) cap from high to low, with capping active at 100% load in a single step, and (2) cap low to high with other factors remaining constant). A single test typically runs for between thirty seconds and two minutes.

Architecture of the experimental framework

The system uses a client-server model, with the server being an agent running on the system under test. The agent performs two distinct roles:

1. Launches the workload with the requisite configuration;

2. Monitors the per-socket power consumption through the Running Average Power Limit (RAPL) interface and sends this to the client at the end of each run.

The client iterates over the different permutations of test configurations requesting the agent to launch the workload. Then, after a warm-up period, it will apply the capping operation via the management interface. During the run, the client polls the Baseboard Management Controller (BMC) for system-level power consumption. At the end of each test run, the client loads the test configuration along with the timestamped RAPL and BMC power data and stores these in a relational database.

To assist with the analysis, a web-based, data-driven, graphical interface, which plots the data for each test held in the database, was developed. This too uses a client-server model with a web server offering a custom REST API to the database which is consumed by the web interface.

The choice of the client-server model was imposed by the typical HPC system configuration, whereby there is a single network interface used by both the BMC and the host server. This usually disallows communication between the host and its own BMC.

Preliminary experimental observations

The experimental framework is still under active development, but, as part of this process, it has been run against a (single) x86-64 test system. Whilst this has not been a formal campaign and as such no hard conclusions can be drawn, it is possible to make the following observations:

- The evaluated compute node ignored capping requests, when either capping from high power to low power, or from low to high;
- One possible sequence of capping commands that appears to reliably have successful and rapid capping effects was identified. This has to be confirmed once the experimental framework is complete and additional blade-types are tested;
- For the system used for the development a capping operation appears to reduce the frequency on just one of the two sockets, the other continuing at full power. It is still to be determined if it is always the same socket.

Next steps

To conclude this section, we describe the next envisioned steps regarding the integration of BEO and OAR:

- Launch a formal test campaign across a variety of HPC compute nodes with different CPU models. Full analysis of the results. If possible identify the conditions for systematic, guaranteed successful capping operations;
- If required, adapt, accordingly to the result of the aforementioned test campaign, the design of the prologue and epilogue underlying the integration of BEO with OAR;

- Implement the AAPC mechanism in BEO, together with the aforementioned prologue and epilogue;
- Experimentally evaluate the AAPC mechanism, and compare it to a FSPC mechanism regarding job throughput.

EAR - DCDB

As outlined in the previous section on the EAR/EXAMON integration, EAR focuses on energy management, monitoring and optimization of HPC systems. DCDB on the other hand focuses on holistic monitoring of entire data centres, spanning from system hardware over system software and applications to supporting infrastructures like cooling or power distribution. For its energy management and optimization functionality, EAR collects metrics on application performance and power consumption that would typically also be collected by DCDB. Since EAR has certain requirements towards readout frequency and latency that are higher than those of general monitoring, it seemed reasonable for EAR to collect the metrics it relies on itself. However, in order to reduce monitoring overhead on the compute nodes and avoid potential conflicts accessing the same sensor data, we decided to integrate both tools and have EAR send the metrics it collects to DCDB such that DCDB still has a global view on all metrics without having to read the same data twice..

EAR collects metrics in two granularities: loop and application. Loop corresponds to a piece of code executed in a repetitive way. One application can have one or multiple loops (iterative regions). EAR reports performance and power metrics periodically for each loop. For each loop, EAR reports the set of "iterations", *jobid*, *stepid* and *nodename*. For each of this set of iterations EAR measures the Loops Signature and reports its data. This is a set of metrics computed during the application execution in short intervals (default value 10 sec.). EAR also reports the Application Signature, the same set of metrics used at the loop level but applied to the granularity jobid-stepid-nodeid.

EAR includes a reporting mechanism based on plugins with a simple API, one for each EAR type. The following shows a subset of the report API.

```
/*Report EAR Data Types*/
state_t report_applications(report_id_t *id, application_t *apps, uint
count);
state_t report_loops(report_id_t *id, loop_t *loops, uint count);
state_t report_events(report_id_t *id, ear_event_t *eves, uint count);
state_t report_periodic_metrics(report_id_t *id, periodic_metric_t *mets,
uint count);
state_t report_misc(report_id_t *id, uint type, const char *data, uint
count);
```

On the other hand, the sensor plugin framework is available within *dcdbpusher*, DCDB's data acquisition daemon that is responsible for gathering telemetry data. It is designed to acquire

telemetry data from arbitrary APIs, interfaces or protocols. Once acquired, the data is transferred to the internal sensor cache from where it is available for further downstream processing by so-called operator plugins (e.g. for aggregation, statistics, analysis). Optionally, it can be pushed towards the *CollectAgent* daemon via the MQTT protocol and then onwards to the persistent storage layer. By default, sensor plugins are loaded at the startup of *dcdbpusher*, but can also be loaded or unloaded during the lifetime of the daemon via its RESTful API.

The EAR report plugin, which we call *dcdb-report-plugin* [13], is used to send the EAR collected data to DCDB. It implements the report periodic metrics, report loops, and report events APIs of the EAR reporting APIs described above. The integration is using a shared memory region to exchange data between EAR and DCDB. When the *dcdb-report-plugin* is loaded, the collected EAR data per report type is written to the shared memory region which is accessed by DCDB to collect the data.

A new sensor plugin, called *ear-sensor*[14], was developed on the DCDB side. This ear-sensor collects the EAR metrics from the shared memory region and maps their names from the EAR namespace to a DCDB-compatible sensor naming scheme. The collected EAR metrics data is pushed into the storage layer using MQTT messages.

The Figure 8 below shows the flow of the data from the EAR components that generates

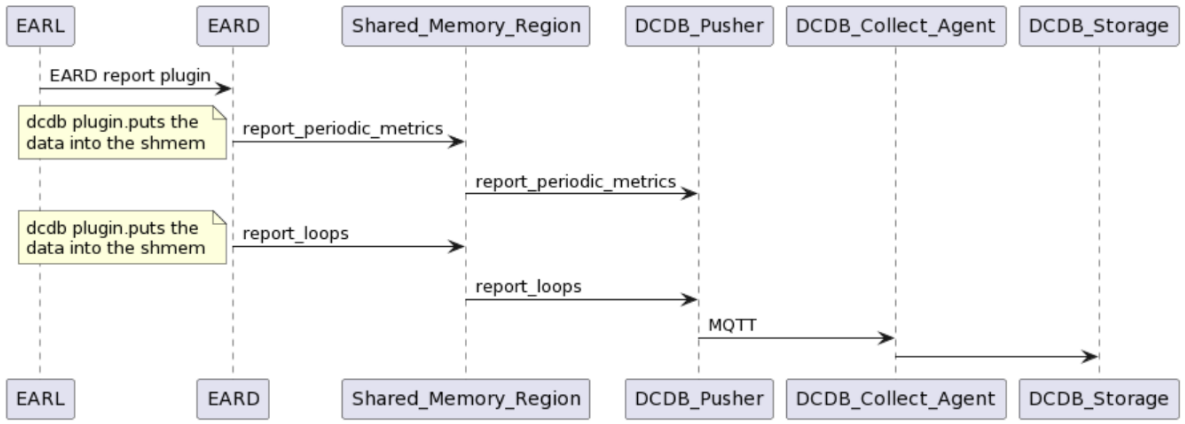

Figure 8: Data Flow between EAR and DCDB Componentes

them to the DCDB storage layer. The EAR library (EARL) and EAR Daemon (EARD) are the responsible components to generate the EAR data metrics, the *dcdb-report-plugin* on the EAR tool side is responsible to create/setup the shared memory segment that is being used for data transfers between EAR and DCDB. It is the only producer that writes data to this shared memory segment, and the DCDB plugin the only consumer that reads from it. It is organised as a ring buffer and data is written as a well-defined C-struct which allows for efficient data transfer between the two plugins. The EAR *dcdb-report-plugin* writes the collected data per report type API into the shared memory, then the *ear-sensor* on the DCDB

tool periodically checks whether new data is available in the shared memory, reads it, stores it in its sensor cache, from where it enters DCDB's standard telemetry processing pipeline.

The collected metric's names and the frequency of collecting the data between both EAR and DCDB tools are configurable from both tool´s sides and it must be synced on both report plugins sides to work properly. The currently developed version of EAR *dcdb-report-plugin* keeps records of the last five collected values per metric type into the shared memory.

The collected metrics include but are not limited to frequency, power consumption (CPU, DRAM, GPU), energy, memory bandwidth, and temperature.

This integration between EAR and DCDB is necessary since performance counter data can only be collected by a single entity on each compute node. Although the Linux perf interface DCDB and EAR use to read performance counters in theory allows for shared access, this would result in lower temporal resolution at best. More often, it causes conflicts between multiple consumers that lead to starvation of one of them. With this integration, EAR can collect the performance counter data it needs for its runtime optimizations and only forwards it to DCDB that can seamlessly insert it into its data processing pipeline. When EAR is not running, e.g. because a particular job requested to not run under EAR control, the corresponding perf plugin in DCDB can be switched on via its REST API and DCDB will collect the performance counter data on its own. At job termination, the plugin can be switched off again.

4. Extensions of Integration Scenarios

The above component-to-component integrations play a vital role to realise the integration scenarios we presented in D3.1. As presented in D1.2/D3.1, the integration scenarios cover most of the use cases defined in WP1, and we are extending these scenarios to enhance the coverage and functionalities. Table 2 lists the properties of use cases with different sophistication levels and demonstrates how these integration scenarios are mapped on it. As shown in the table, these scenarios cover most of the planned areas, and the extensions will be enhancing the quality within the currently covered area or extending the coverage by combining with other work packages/tasks.

Scenario 1

	Level 1	Level 2	Level 3	Level 4
Objective(s)	None	One (system or app focused)	Multiple objectives	…
Constraint(s)	One (e.g., power constraint)	Two (e.g., power + temperature)	More (e.g., power + temp + anomaly detectable)	…
Temporal Granularity	Statically set by site admin	Statically set when job launch	Dynamically adjusted at runtime	…
Spatial Granularity	Entire compute nodes (all nodes work uniformly)	Intra- xor inter-node optimization	Both intra- and inter-node optimization	Include other kinds of nodes or facilities
Knob(s)	CPU power mgmt knob	Power mgmt knobs of multiple components	Include job scheduling optimizations	Scenario 3 …
Optimizer(s)	None	One (system xor user level optimization)	Both system and user level optimization	Scenario 2 …

Table 2: Sophistication levels and mapping of integration scenarios

As mentioned above, this deliverable focuses on the PowerStack path, however we also consider the workflow engine path, which is currently dealing with the integrations with our 5 pilot applications in WP4. Converging those two different paths and evaluating the effectiveness of our power management schemes for real applications powered by the workflow engines, and exploring the synergies as well as the corporations between those two different paths (power budgeting per a set of ensemble runs) will be a significant extension of our approach and will be covered in D3.3.

Furthermore, as we also have the sophistication effort in WP2, all the tasks in WP2 can also be potential extensions for the three integration scenarios. For instance, T2.1 in WP2 is about profile-driven job characterization and energy estimation, which will help the core estimation part of the integration scenarios. T2.2 attempts to optimise application performance/power

in a phase-aware manner by a job manager, which is also promising to extend the integration scenarios. T2.3 handles temperature-aware power-performance optimization and anomaly detections, which complements the missing pieces in the integrations, i.e., two or more constraints. The other tasks (I/O-aware resource allocation, elastic resource management, co-scheduling) are also effective to enhance the use cases/scenarios, and we will carefully pick the sophistications and map them to the integration scenarios as extensions, and also consider multi-objective optimization.

5. REGALE Library

The ambition of REGALE is to materialise the PowerStack initiative fulfilling the integration scenarios with an action which overpasses the single power management tool integration providing a unifying layer for the interoperability of the power management tools in the context of the HPC. In its essence this means to expose all the properties related to power management knobs and sensors of compute nodes and running jobs phases, to power management policies running at different abstractions in a coordinated manner. This is not very different from a fleet/swarm of drones solving complex tasks. However, a swarm of drones can be programmed by a control engineer with a limited computer science background, while implementing a component in the HPC PowerStack requires a PhD in computer science. In the robotic community this is achieved thanks to a unifying middleware - namely ROS2 [23] - which abstracts the low-level platform-dependent interfaces with robust inter-agent messaging based on publish/subscribe mechanism [24]. To accomplish this the ROS2 library leverages the DDS [10], which is suitable for real-time distributed systems due to its various transport configurations (e.g., deadline and fault-tolerance) and scalability. ROS2 converts data for DDS and abstract DDS from its users.

We believe that this approach can match the requirements of the HPC PowerStack and we embraced it for creating the unifying middle layer for the power management in HPC - namely the REGALE library. With a similar spirit we selected DDS as a communication layer and hid its complexity within the REGALE library.

DDS Basics

DDS [10] is a protocol used to make distributed softwares able to communicate among each other. To achieve this, DDS implements a publish/subscribe protocol, which is a communication protocol that facilitates the exchange of messages or events between publishers and subscribers in a decoupled manner. It provides a flexible and scalable mechanism for information dissemination, where publishers do not need to have direct knowledge of the subscribers. Instead, publishers publish messages or events to specific topics or channels, and subscribers express their interest in receiving messages from certain topics or channels. The protocol ensures that published messages are delivered to interested subscribers in a timely and efficient manner.

DDS implements its Data-Centric Publish Subscribe (**DCPS**) model defining three key entities in its implementation.

- **Publication entities**: they are responsible to define the information-generating objects and their properties.
- **Subscription entities**: they define the information-consuming objects and their properties;

- **Configuration entities**: they define both the types and the QoS (Quality and Service) properties of the information transmitted as topics between publication and subscription entities.

In the DCPS model (see Figure 9), four basic elements are needed in the system of communicating applications.

- **Publisher**. It is in charge of the creation and configuration of the **DataWriter** it implements, which in turn is the one which deals with the actual publication of the messages. Of course a topic will be needed, under which the messages are published.
- **Subscriber**. It is in charge of receiving the data published under the topics to which it subscribes. The **DataReader** object which it serves, is instead responsible for communicating the availability of new data to the application.
- **Topic**. It binds publications and subscriptions. It is unique within a DDS domain (see the following picture).
- **Domain**. This concept is needed to link all publishers and subscribers belonging to one or more applications, which exchange data under different topics.

  The individual applications that participate in a domain are called **DomainParticipant**, and they define the DDS domain to which they belong to, specifying a domain ID, which is a unique trait of each DDS domain. Two DomainParticipants with different IDs are not aware of each other in a network, then different communication channels can be created.

  Imagine for example a scenario where different applications are involved, but some of them must not interfere with each other. In this case then, their respective DomainParticipants act as containers for publishing, subscribing, and topic entities, but the domain IDs behave as a logical barrier able to isolate some applications from others.

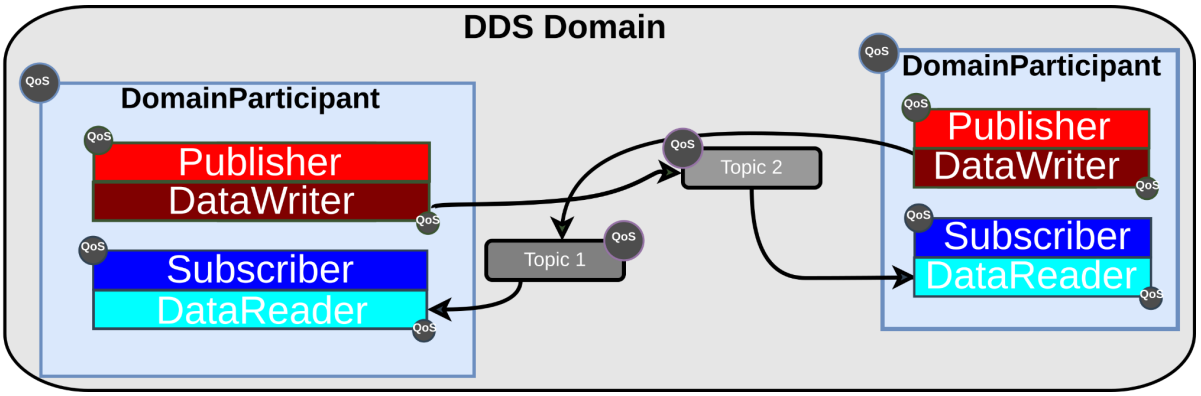

Figure 9: Elements in the DCPS model

Under DDS applications we can find the RTPS (Real-Time Publish Subscribe) protocol, actually designed to support that specific data distribution system.

The RTPS protocol (see Figure 10) is designed to support both unicast and multicast communications, and at its top the Domain can be found (inherited from DDS), which as

already said defines separate planes of communication. And inside a Domain there are the **RTPSParticipants**, which are elements capable of sending and receiving data using as their Endpoints:

- **RTPSWriter**: able to send data.
- **RTPSReader**: able to receive data.

And as for DDS, communication among participants is resolved around **Topics**, which define and label the data being exchanged and do not belong to a specific participant. The participant, through the RTPSWriters, makes changes in the data published under a topic, and through the RTPSReaders receives the data associated with the topics to which it subscribes. The communication unit is called **Change**, which represents an update in the data that is written under a Topic. RTPSReaders/RTPSWriters register these changes on their **History**, a data structure that serves as a cache for recent changes.

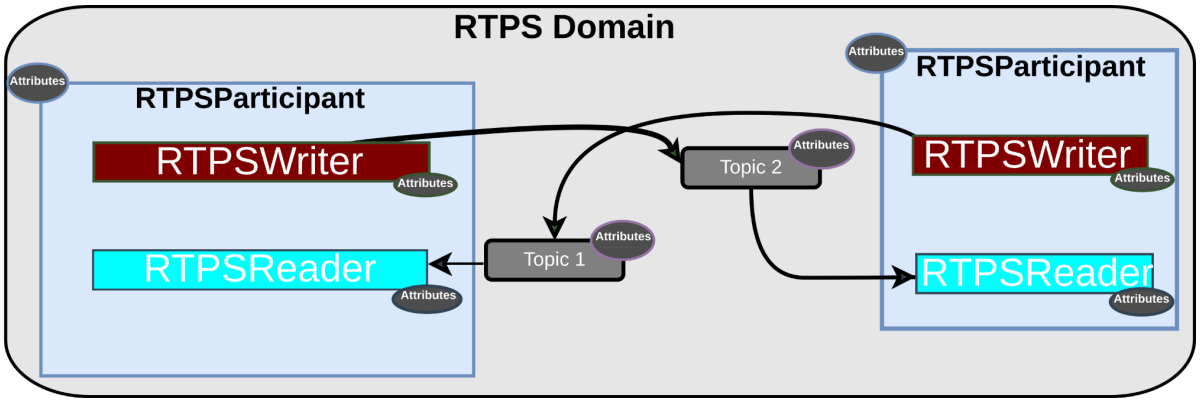

Figure 10: Elements in the RTPS model

To sum up, when you publish a change through a RTPSWriter endpoint, the following steps happen behind the scenes of a default configuration:

- The change is added to the RTPSWriter's history cache.
- The RTPSWriter sends the change to any RTPSReaders it knows about.
- The RTPSReaders receive the data, and update their history cache with the new change.

Structure of the REGALE middle layer based on FastDDS

Our goal is to implement a library in a complete form in the first stage, acting as a communication middle layer for all the software modules composing this ecosystem. And under the hood of this library, one can find the eProsima FastDDS [1] middleware, which is a C++ implementation providing both the OMG DDS and RTPS wire-protocol standards, and which ensures the necessary transmission speed in the order of microseconds, required to tweak the knobs of the power and energy efficiency field. What is presented in this deliverable intends to act as a starting point for a future software stack, in which all the components that will be part of it might be easily interchanged, still communicating and exchanging data among one common middle layer software: the REGALE library.

The basic idea behind this library is, following also the mechanics behind FastDDS, to provide the users with some simple methods useful to create publishers, subscribers, and after having picked some topics, being able to both publish and subscribe to those, exchanging specific information in a less invasive way than what was presented in the previous chapters of this document. With that in mind, the picture starts to become clear: substitute all the hacks and modifications done (or to be done), created to allow two software modules to communicate, with one communication layer easy to link with, and that exposes clear and easy-to-use methods. In this way, the code of all software modules and libraries currently in play, will demand very few changes to be able to communicate with all the other participants, and not just with one specific.

At the time this text is written, the REGALE library has a main core file, the "*regale.cpp*" one, which contains the definitions of the methods that can be called by the software which will link the correlated library ("*libregale.so*"), and include the concerning header file ("regale.h"). The methods contained in these files, are just C wrappers for the members defined (and declared) in all the other *.cpp* files which are composing the project, and that are classes' methods. We decided to go with straight C, because it is more clear for setting the standard for the methods and the APIs, and it is usable both in C and C++ projects.

As can be seen in Figure 11, the actual structure of the REGALE library contains additional files briefly described next.

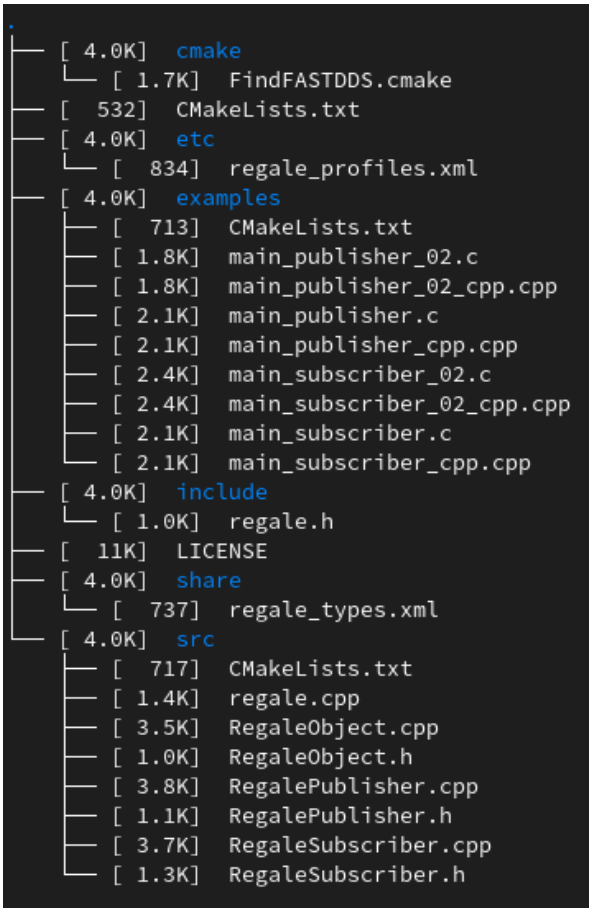

Figure 11: Structure of the REGALE library

Directory: *src*

- ***RegaleObject***: this file contains the parent object of the RegalePublisher and the RegaleSubscriber, containing all the members and code common to the two specialisations, like the loading of the two *.xml* configuration files within the folders *share* an *etc* (we will cover them later), the creation of the *topics* (in FastDDS, they are objects), and the dynamic creation of the data type, chosen by the user, at runtime.
- **RegalePublisher**: this file contains, in addition to what is derived from the parent RegalObject, the creation of the DataWriter object which is, as the previous paragraph reports, the dedicated object dealing with the actual publication of the messages. Moreover, here are specified the Quality of Service (QoS) for the publishing object, like the *kind* (which defines the reliability kind of the endpoint) and the *max_blocking_time* (which defines the maximum period of time certain methods will be blocked), and it is specified also the "*publish*" method, which is the actual method to publish something. Lastly, quite important is to cite here the subclass *PubListener*, contained in the *RegalePublisher* object and derived from FastDDS *DataWriterListener*. This member is important because its method *on_publication_matched*, overridden, is the one which takes charge of checking if there are some matches for the parent publisher and, in the positive case, triggers the specified action.
- **RegaleSubscriber**: as for the previous one, here we are specialising the REGALE objects, so worth of notes can be the specification of the QoS (as for the RegalePublisher) and the methods "*on_subscription_matched*" and "*on_data_available*"of the subclass *SubListener*, derived from the FastDDS class *DataReaderListener*. If the first one is the counterpart of the publisher "on_publication_matched", the last one is a non blocking method which checks if new data has arrived, and if yes stores it. The interesting thing about this subscriber approach is that with this last method, once a subscriber is created and a specific topic is passed to it, then the discovery of new data will happen automatically, without any call of some sort of "*subscribe*" method, and will continue until the parent program which has instantiated the Subscriber object will terminate its operations.
- **regale**: this file contains the following wrappers:
  - ➔ "*Regale_create_publisher*": which is a method returning a pointer to a *RegalePublisher* struct, and that calls the constructor of the *RegalePublisher* object. This method must be used in the software which is linking the REGALE library, where it is needed to create and initialise a publisher (examples follow in the next section).
  - ➔ "*Regale_create_subscriber*": which is a method returning a pointer to a *RegaleSubscriber* struct, and that calls the constructor of the *RegaleSubscriber*

object. This method must be used in the software where an initialised Regale subscriber is needed (for some hints on how to use it, see the next section).

➔ "*Regale_publish*": which is the method wrapping up the *publish* member of the class *RegalePublisher*, and is the method which should be called to publish something to the subscribers waiting on the related topic (for related examples, see the next section).

➔ "*Regale_delete*": which is the method responsible to call the virtualized destructor of the specialised RegaleObject.

Directory: *etc*

- **regale_profiles**: this *.xml* configuration file (see Figure 12) contains (see picture below) information related to the transport types available for the user, which can be chosen at run time. Once a RegaleObject is being created, during its initialization it loads this XML file, and it gets the transport type using the FastRTPS function getTransportById, which retrieves a transport instance by its id (e.g., in this case: "udpv4_transport", "tcpv4_transport", "shm_transport").

```
<dds xmlns="http://www.eprosima.com/XMLSchemas/fastRTPS_Profil
    <profiles>
        <transport_descriptors>
            <transport_descriptor>
                <transport_id>udpv4_transport</transport_id>
                <type>UDPv4</type>
            </transport_descriptor>
            <transport_descriptor>
                <transport_id>tcpv4_transport</transport_id>
                <type>TCPv4</type>
            </transport_descriptor>
            <transport_descriptor>
                <transport_id>shm_transport</transport_id>
                <type>SHM</type>
            </transport_descriptor>
        </transport_descriptors>
    </profiles>
</dds>
```

Figure 12: Configuration file REGALE library

Directory: *share*

- **regale_types**: this *.xml* configuration file (see Figure 13) contains, instead, information related to the data types that the user wants to use (see picture below) in a specific run. Using the FastRTPS function *getDynamicTypeByName*, once the configuration file has been loaded during the initialization of the desired *RegaleObject*, the requested data type is dynamically built and created, and ready to be used to store values, or to be sent.

In this specific case, two structs have been added to this configuration file: the first one is "*struct_freq*", which contains two *uint32* members: "*max_freq*" and "*min_freq*". The second structure to be in play, is the "*struct_power*" one. This one instead contains three unsigned integer members, of 32 bits each: "max_power", "min_power", "avg_power".

```
<dds xmlns="http://www.eprosima.com/XMLSchemas/fastRTPS_Profiles">
    <types>
        <type>
            <struct name="struct_freq">
                <member name="max_freq" type="uint32"/>
                <member name="min_freq" type="uint32"/>
            </struct>
        </type>

        <type>
            <struct name="struct_power">
                <member name="max_power" type="uint32"/>
                <member name="min_power" type="uint32"/>
                <member name="avg_power" type="uint32"/>
            </struct>
        </type>
    </types>
</dds>
```

Figure 13: Configuration file REGALE memory types

Examples and DDS basics

In the following, we present some examples to provide a better understanding of the overall picture of the REGALE library, of its implementations and of its usage.

The library itself is equipped with some examples, present in the directory *examples*. We will see a couple of them, to show its ease-of-use. The selected files will be "*main_publisher.c*" and "*main_subscriber.c*". First of all, a couple of hints about their code (see following figures):

- **main_publisher**: This file (see Figure 14) calls the "*Regale_create_publisher*" function, which demands as input four values: the first two are the absolute paths to the configuration files (*regale_types.xml* and r*egale_profiles.xml* we presented previously), while the last two are strings, representing the transport id chosen and the topic to which publish (all the 4 input parameters are passed by the user at runtime). This method returns a pointer to the *RegalePublisher* struct, which will be used by the wrapper of the publishing method (*Regale_publish*) to route the corresponding member of the relative C++ class. Depending by the topic chosen (in this case, it can be equal to "*struct_power*" or "*struct_freq*", which are actually the two options at this moment we have for the data type choice, for a sake of simplicity), the values to be published will be different: a vector of three or two *uint32_t* values.

Regale_publish is called three times, sending different arrays of data with one second breaks. At last, "*Regale_delete*" is called, to invoke the specific wrapped destructor.

- **main_subscriber**: This file calls (see Figure 15) the method "*Regale_create_subscriber*", which acts as a counter part of the *Regale_create_publisher*, asking however five input parameters. The first four are the same as the publisher; the last one instead, is a pointer to a function which we would like to trigger, once received some data. As soon as the *RegaleSubscriber* pointer has been created (*ptr_func* will vary depending of the content of the topic), the *main_subscriber.c* starts to execute a little loop kernel, just to show what we said previously about the non blocking approach of the subscriber: if someone is publishing, than the message will be subscribed, and it will applied to it the function passed as input parameter. Otherwise, nothing happens, and the "computational" part, independent of the messages exchange, is completed.

  As the last method call, the "*Regale_delete*" is present, to delete the subscriber.

```
int main(int argc, char** argv) {
    char* types_file = argv[1];
    char* profiles_file = argv[2];
    char* topic = argv[3];
    char* transport = argv[4];
    uint32_t freqs[] = {3100000, 1000000};
    uint32_t freqs2[] = {2800000, 1500000};
    uint32_t freqs3[] = {2500000, 2000000};
    uint32_t pows[] = {1000, 2000, 3000};
    uint32_t pows2[] = {4000, 5000, 6000};
    uint32_t pows3[] = {7000, 8000, 9000};
    uint32_t* values = NULL;
    uint32_t* values2 = NULL;
    uint32_t* values3 = NULL;

    if (!strcmp(topic,
                "struct_freq")) {
        values = freqs;
        values2 = freqs2;
        values3 = freqs3;
    }
    else if (!strcmp(topic,
                     "struct_power")) {
        values = pows;
        values2 = pows2;
        values3 = pows3;
    }

    if (argc != 5) {
        help_function(argv[0]);

        return 1;
    }

    printf("Starting publisher.\n");

    RegalePublisher* pub = Regale_create_publisher(types_file   ,
                                                   profiles_file,
                                                   transport    ,
                                                   topic);


    Regale_publish(pub,
                   values);
    sleep(1);

    Regale_publish(pub,
                   values2);
    sleep(1);

    Regale_publish(pub,
                   values3);
    sleep(1);

    Regale_delete((RegaleObject*)pub);

    printf("Ended publisher.\n");

    return 0;
}
```

Figure 14: Source code of an example of REGALE publisher

```c
void print_subscribed_freq_values(void* values) {
    printf("Received message with values %ld %ld\n",
           ((uint32_t*)values)[0]                  ,
           ((uint32_t*)values)[1]);
}

void print_subscribed_power_values(void* values) {
    printf("Received message with values %ld %ld %ld\n",
           ((uint32_t*)values)[0]                       ,
           ((uint32_t*)values)[1]                       ,
           ((uint32_t*)values)[2]);
}

int main(int argc, char** argv) {
    char* types_file = argv[1];
    char* profiles_file = argv[2];
    char* topic = argv[3];
    char* transport = argv[4];
    uint64_t i;
    uint64_t j = 0;
    void (*ptr_func)(void*);

    if (!strcmp(topic,
                "struct_freq"))
        ptr_func = print_subscribed_freq_values;
    if (!strcmp(topic,
                "struct_power"))
        ptr_func = print_subscribed_power_values;

    if (argc != 5) {
        help_function(argv[0]);

        return 1;
    }

    printf("Starting subscriber.\n");

    RegaleSubscriber* sub = Regale_create_subscriber(types_file   ,
                                                     profiles_file,
                                                     transport    ,
                                                     topic        ,
                                                     ptr_func);

    for (i = 0; i < 2000000000; i++) {
        j += i;
    }
    printf("j = %ld\n", j);

    Regale_delete((RegaleObject*)sub);

    printf("Ended subscriber.\n");

    return 0;
}
```

Figure 15: Source code of an example of REGALE subscriber

We show how to execute the two programs, and the results obtained.

Once compiled the codes in a classical way (remember to include the header of REGALE, "*regale.h*", and to link its library), the two executables generated can be launched on different nodes (as it is in this case) or on the same node, as follows:

- ./main_publisher ~/test_regale_installation/install/share/regale_types.xml ~/test_regale_installation/install/etc/regale_profiles.xml struct_power udpv4_transport
- ./main_subscriber ~/test_regale_installation/install/share/regale_types.xml ~/test_regale_installation/install/etc/regale_profiles.xml struct_power udpv4_transport

to obtain the following output from the subscriber (the publisher does not execute anything than sending data), where the printed values are the ones stored into the arrays *pows*, *pows2*, *pows3*, in *main_publisher.c*:

```
Starting subscriber.
Received message with values 1000 2000 3000
Received message with values 4000 5000 6000
Received message with values 7000 8000 9000
j = 1999999999000000000
Ended subscriber.
```

Figure 16: The subscriber has received some data from a publisher registered on the same topic, and using the same transport layer

or the following one, if the publisher has not been activated.

```
Starting subscriber.
j = 1999999999000000000
Ended subscriber.
```

Figure 17: The subscriber has not received anything, but it has continued and finished its computational part, without any blocking interference

Countdown extensions for supporting FastDDS

COUNTDOWN is a tool to identify and automatically reduce the power consumption of the CPU cores, during communication phases of an MPI-based application. It is a pure reactive mechanism (based on a timer), and not based on a learning mechanism. By default, the timer is set to 500 microseconds, because that has been evaluated as the time needed by the hardware logic to effectively take into consideration, and apply, the change of frequencies.

You can see in Figure 18 an example: the first MPI call lasts less than the default timer, and so the callback to lower down is not triggered. Viceversa, the second MPI event is very long, and so COUNTDOWN can reduce the frequency, and save power during communication.

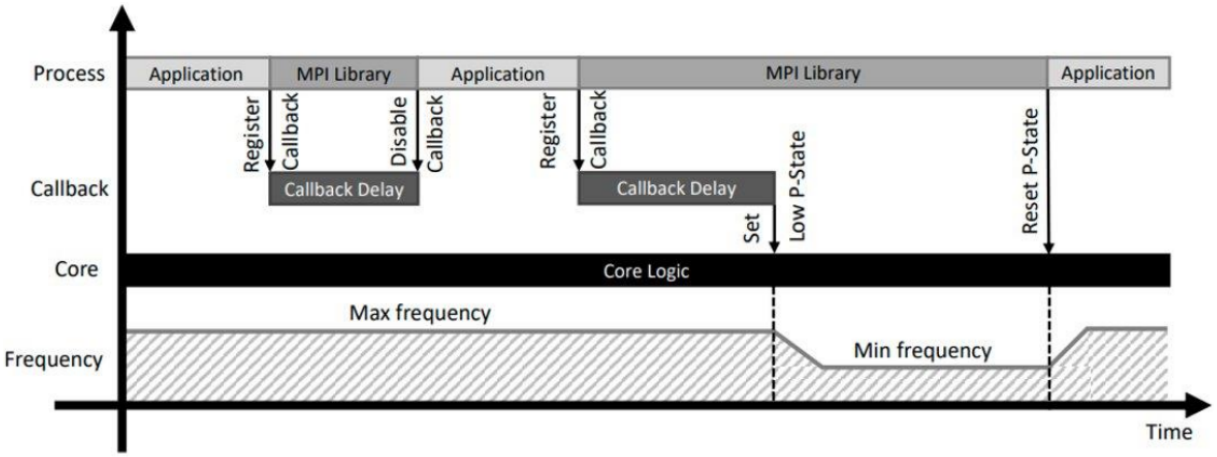

Figure 18: COUNTDOWN reactive mechanism

This saving of energy takes place without imposing significant time-to-completion increase, by lowering CPUs power consumption only during idle times for which power state transition overheads are negligible. This is done transparently to the user, without requiring labour-intensive and error-prone application code modifications, nor requiring recompilation of the application.

Hereafter a schematic approach (Figure 19) of how COUNTDOWN works transparently to the user, standing among the application and the MPI library, intercepting the latter's calls. The only thing asked to the user, is to set the *LD_PRELOAD* variable, with the path of the COUNTDOWN library.

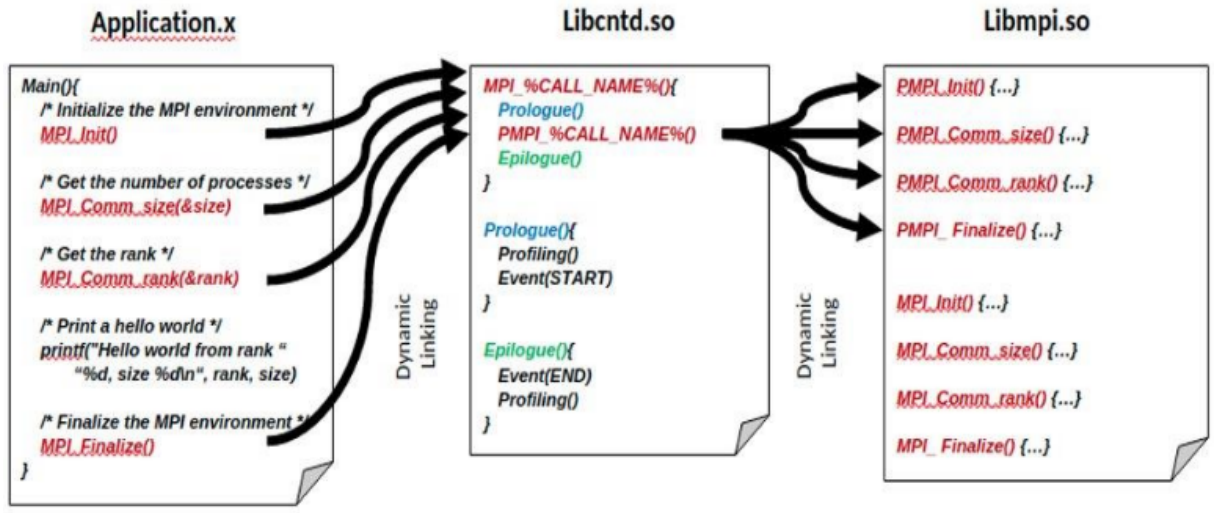

Figure 19: Dynamic link mechanism of COUNTDOWN library

As it can be seen, the COUNTDOWN library is able to automatically track at fine granularity MPI and application phases, injecting power management calls whenever it is convenient to do so.

In the vision of the REGALE project, COUNTDOWN can change its normal behaviour for single applications, finding its place in the project specific software stack as a Job Manager (JM).

This entity, in the REGALE ecosystem, directly interacts with the Node Manager (NM), to let it have a more refined vision of the underlying monitored system.

Under this assumption, the REGALE library will be used by COUNTDOWN (see Figure 20) in a straightforward manner, to publish its frequency hints to the NM, which will be subscribed to the corresponding topic. Let's have an example, to better explain the idea behind this sentence.

Entering MPI phase for which COUNTDOWN believes that frequency can be lowered, a DDS message is sent to the NM with a specific data types containing the minimum frequency at which COUNTDOWN thinks that the application could run, without getting any performances and Time To Solution (TTS) losses, correlated to a preselected topic. Once the NM receives the message, it is free to apply or not the hint sent by COUNTDOWN, following its more wider logic for the power capping of the entire monitored system. If it judges that the frequency proposed by COUNTDOWN is compatible with its choices, then it will apply that frequency, modifying those system files (like *scaling_max_freq*, *scaling_min_freq*, *scaling_setspeed*, depending on the governor in use) associated to the frequency modification.

The same approach can be utilised for when COUNTDOWN exits the MPI phases for which it speculates if it is feasible to increase the current frequency, reaching the maximum one available on the system, to speed up the TTS of the underlying application. In this case, a similar DDS data type to the previous one will be sent to the NM, containing a hint for the maximum frequency at which the job runs. If the NM estimates that that tip is still aligned with its power capping methodology, then it could apply it.

In all cases however, COUNTDOWN will continue its work, using the frequency for the MPI and APPLICATION events that it will find available on the system.

In Figure 20, a schematic illustration is reported on how easily the REGALE library can be used by the different tools (they need just the header and the library). In the specific case of COUNTDOWN, after linking this library we can use the method *Regale_publish* to send the desired frequency to all the subscribers listening for its specific topic. Another method, *Regale_create_publisher*, must be called during the initialization phase of Countdown, to store a publisher entity and easily later use it, to publish the frequency values hinted by COUNTDOWN.

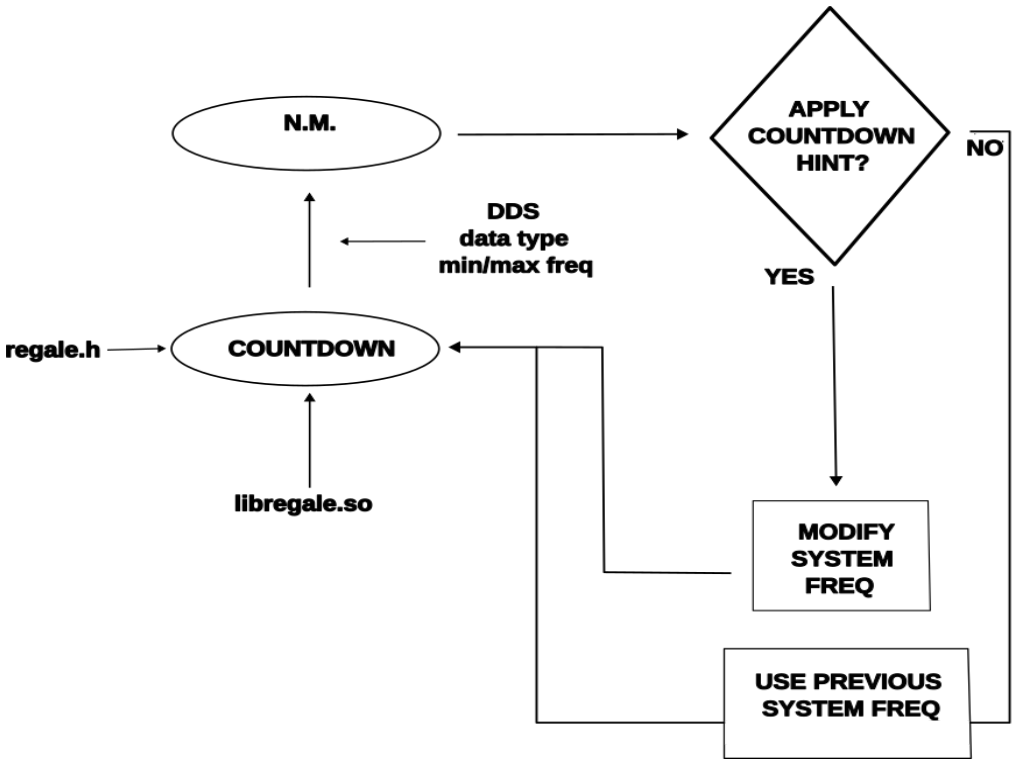

Figure 20: Representation of how REGALE library can be used in the interaction among the JM (COUNTDOWN) and the NM

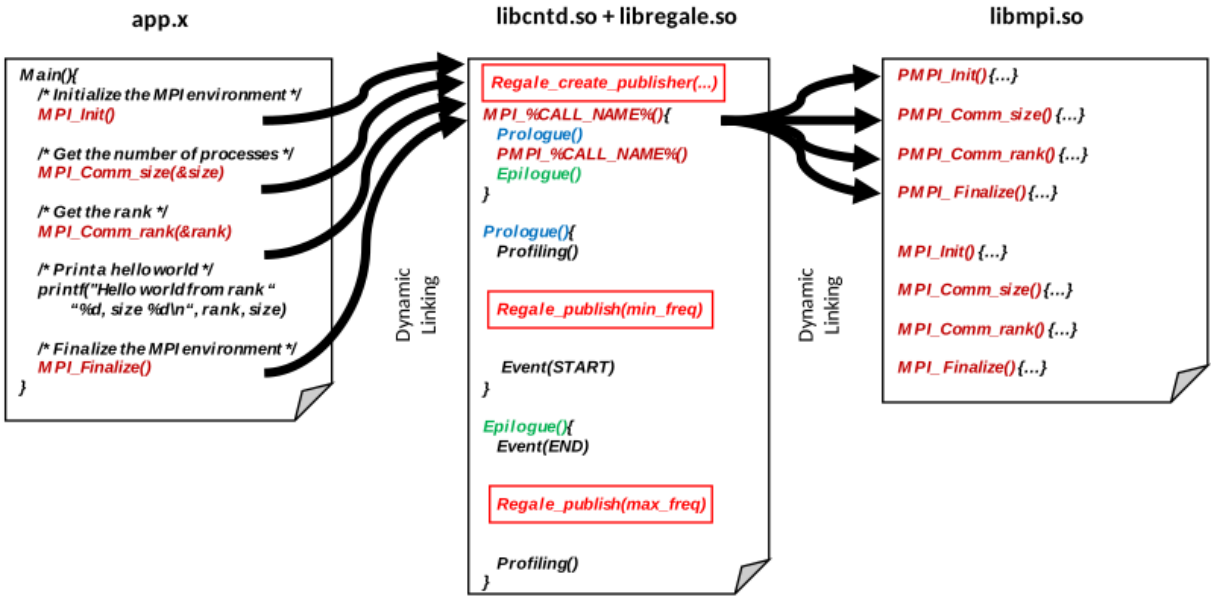

Figure 20: Dynamic linking of COUNTDOWN + REGALE library

ExaMon extensions for supporting FastDDS

As a data provider, ExaMon offers several ways of accessing data. The main interfaces are the MQTT bus for streaming and real-time applications and a RESTful interface for accessing data

in batch mode. In this section, a possible implementation of the interface between MQTT and DDS will be described.

Table 3 shows a concise comparison of the two protocols:

	DDS	**MQTT**
Communication Model	Decentralised publish-subscribe	Broker-based publish-subscribe (request-response available in MQTT v5)
Data-Centric vs. Message-Based	Data-centric	Message-based
QoS Levels	Extensive and configurable	Three levels: 0, 1, and 2
Scalability and Extensibility	Supports scalability and extensibility	Lightweight and suitable for resource-constrained devices
Real-Time Capabilities	Comprehensive and configurable	Supports real-time communication
Implementation Complexity	More complex and feature-rich	Simpler and more lightweight
Widely Used in	Industrial, mission-critical systems, Internet of Things (IoT)	IoT and machine-to-machine (M2M) communication

Table 3: FastDDS vs MQTT protocol

The notable differences are that DDS does not rely on a centralised broker to manage the data flow. This is because DDS is focused on data exchange unlike MQTT, which instead uses the message as a discrete unit of communication. This allows DDS to have more control over properties such as QoS and real-time communication capabilities at the expense of more complex implementation. For example, for reliable communication and thus for the management of the handshaking protocol, DDS relies on all agents participating in the data exchange. In MQTT, this complexity is left to the broker. For example, any client that has to send a piece of data to thousands of receivers only has to be concerned with the single communication with the broker. The cost of this simplicity is transferred to the fact that the broker, if not properly managed, can represent a single point of failure.

MQTT-DDS Bridge: Generally, communication between applications adhering to different protocols can take place via a component called a “bridge”. The bridge takes care of the

management of both protocols and how messages are to be translated during the transition from one to the other domain.

The implementation of the MQTT-DDS bridge, as represented in Figure 21, is placed on the application layer. It is a software component that acts simultaneously as a publisher and subscriber on both protocols. In more detail, we can summarise the elementary operations performed by this component as follows:

1. *Connection*:
    a. MQTT: The bridge component establishes a connection to the MQTT broker and subscribes to the topics for incoming and outgoing data flow.
    b. DDS: It also connects to an existing DDS domain allowing it to interact with DDS publishers and subscribers
2. *Message Translation and delivery*:
    a. MQTT: When a MQTT message is received, the bridge translates it into a format compatible with the DDS data model and delivers it to the DDS domain. This operation is a mapping between the MQTT and DDS topics and data payloads.
    b. DDS: When a DDS message is received it is mapped to a MQTT topic and payload and delivered to the MQTT domain.
    c. Other than the message payload, the bridge can optionally manage the translation of other parameters like the QoS.

ExaMon to DDS data mapping: Message translation consists of mapping the topics and payloads of the two protocols. In this section, we first see an in-depth description of the individual elements, then we see a possible mapping using the ExaMon data model.

MQTT Topic: MQTT topics are simple strings through which complex hierarchies can be represented using the special character "/". For example, a valid topic can be: "facility/sensors/temperature".

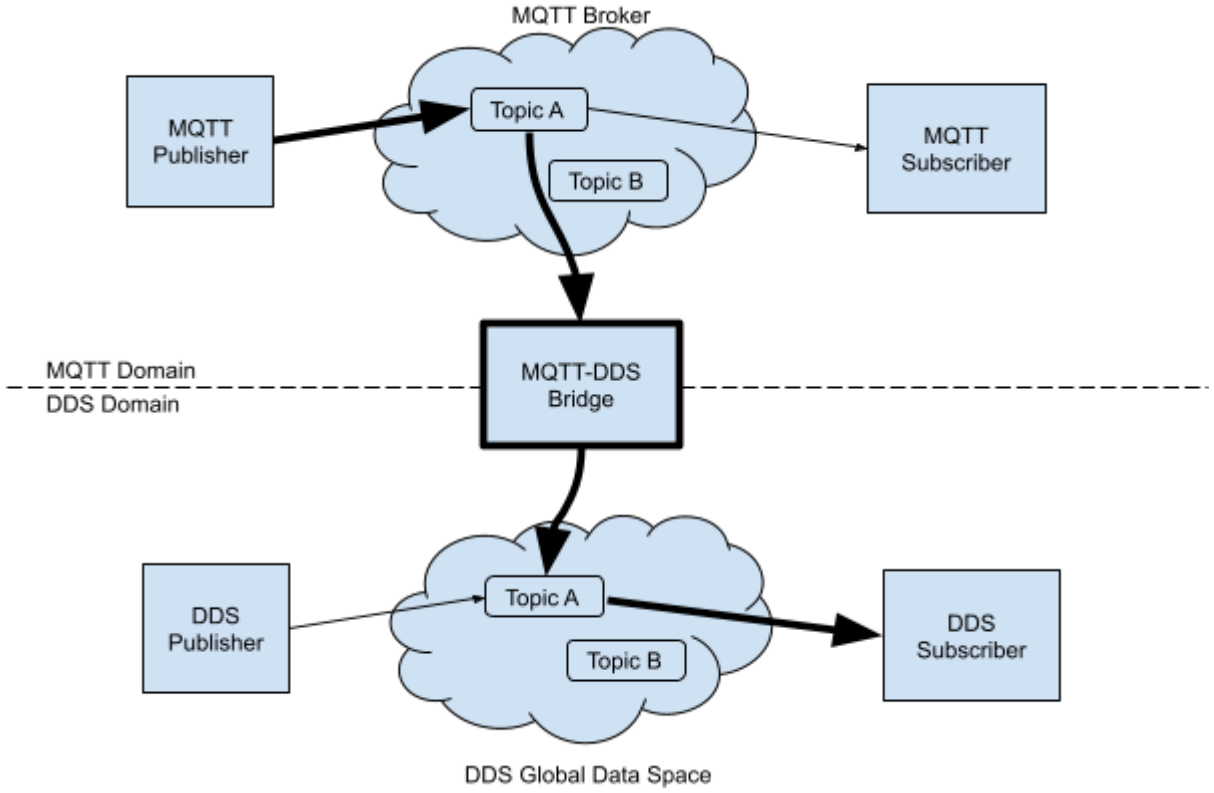

Figure 21: Representation of DDS and MQTT interactions

MQTT Payload: The MQTT messages represent the payload itself. The protocol specification gives no indication of its structure or data type. It may be a simple string, a complex serialised data structure (JSON/XML), or a binary data structure. Moreover, it can change radically within the same application without any implication on the communication layer and the handling (marshalling/unmarshalling) is totally up to the individual application.

DDS Topic: In DDS, topics are objects that generally contain a name and a user-defined data type that represents the structure of the data managed by the topic instance. Generally, the topic name is a simple or structured string describing the associated data.

DDS Payload: DDS protocol messages are strictly defined by the data structure associated with the topic on which they are intended to communicate. The data type can be a structured record, a defined IDL (Interface Definition Language) type, or a custom-defined data structure. This ensures that, given a topic, there can be no uncertainties about the interpretation of the data.

ExaMon Data Model and Mapping

ExaMon focuses on the management of a wide variety of data using the MQTT protocol. This is controlled by the definition of the data model, which is mainly reflected in the definition of the MQTT topic and payload [15].

ExaMon's topic format follows the MQTT protocol guidelines and expresses a hierarchy of elements (key/value) to describe the associated data. An example of a topic defined according to the ExaMon specification might be the following:

```
org/testorg/cluster/testcluster/node/testnode00/plugin/ipmi_pub/chnl/data/units/W/power
```

It represents the topic on which is possible to find the power measurements (in Watts) of the "*testnode00*" node installed in the "*testcluster*" cluster of the "*testorg*" organisation.

The corresponding MQTT payload, as defined in ExaMon is as follows:

```
723.0;1658832078.001
```

It is a string in CSV format where "723.0" is the value, ";" is the separator, and "1658832078.001" is the timestamp.

The mapping of this data model onto the DDS protocol consists mainly of defining the data structure associated with the topic that will receive the data from the MQTT domain. To do this, we follow the DDS standard and define the topic via the Interactive Data Language (IDL) language [16].

```
Struct MqttMsg {
    string value;
    float timestamp;
};
```

Figure 19: Design of a Struct of DDS used to map the Examon Data Model

In this implementation we follow the most generic approach possible, trying to support in DDS a large heterogeneity of data as it is handled by MQTT in ExaMon.

In particular, we define a generic data structure called *MqttMsg* which contains two fields within it:

- **value**: the string type member to which the value field of the MQTT payload will be mapped
- **timestamp**: a float type field to which the timestamp field of the MQTT payload will be mapped.

In this context, the “value” field of the MQTT payload is a string representing a number or a text depending on the data source type. For this reason, it is up to the DDS application to translate the “value” member of the data structure into the appropriate data type.

Communication Sequence

In this section, we describe a sequence of steps that makes up the data communication between MQTT and DDS.

1. During the initialization of the bridge, the connection is made to the MQTT broker and the DDS domain of interest.
2. After initialization, the bridge connects to the MQTT broker and subscribes to the topics of interest. For example, to subscribe to all topics of a given organisation, the MQTT protocol wildcard "#" can be used. For example "org/testorg/#".
3. When an MQTT publisher posts a message on the broker, this is automatically passed on to the bridge.
4. The bridge obtains the payload and the complete topic associated with it. The payload is decoded according to the pattern defined in the IDL language and published in the DDS domain using the same unmodified topic.
5. Any DDS subscriber interested in the data of a certain MQTT topic will subscribe to the equivalent DDS topic with the same name.
6. The data will be obtained and decoded according to the data structure defined in the IDL file.

Spack Deployment

To compile and run the REGALE library, few dependencies are required. So, the REGALE library needs:

- **eProsima FastDDS**: https://github.com/eProsima/Fast-DDS

eProsima FastDDS also depends from the following libraries:

- **ASIO**: https://think-async.com/Asio
- **TinyXML2**: http://grinninglizard.com/tinyxml2
- **OpenSSL**: https://www.openssl.org
- **foonathan-memory**: https://memory.foonathan.net
- **eProsima FastCDR**: https://github.com/eProsima/Fast-CDR

To simplify the deployment of the REGALE library and all its dependencies, we leverage on a well-known package manager called Spack [17]. Spack is a versatile package manager that enables the construction and installation of various software versions and setups across multiple platforms. It is compatible with Linux, macOS, and numerous supercomputers (it’s also the official package manager of Leonardo system at CINECA [18] and of SuperMUC-NG at LRZ [19]).

For this reason, we developed Spack packages for all the REGALE dependencies and we reintegrated in the official release of Spack. In the following, we report all the Pull Request (PR) that we made for the dependencies of the REGALE library to the official Spack project:

- **eProsima FastDDS, eProsima FastCDR, foonathan-memory**: https://github.com/spack/spack/pull/38079
- **ASIO**: already supported on Spack
- **TinyXML2**: already supported on Spack
- **OpenSSL**: already supported on Spack

After that, we also developed a Spack package for the REGALE library, at the time of this deliverable we are waiting the final approvation of the PR:

https://github.com/spack/spack/pull/38444

When the PR of REGALE library will be accepted, to install the REGALE library and all its dependencies will needs only few shell commands with Spack:

```
$ git clone https://github.com/spack/spack.git
$ cd spack/bin
$ ./spack install regale
```

6. Conclusions and Future Works

In this document we have presented how the integration paths of the different tools have proceeded, in comparison with the previous deliverable. A more technical approach has been used, to underline this behaviour and emphasise the software engineering part, that characterises all the tools involved.

Moreover, a new engineering and software effort has been suggested: the REGALE library, which aims to be a standardised middle layer for the power management and energy efficiency software stacks. Its intention is to propose a standardised approach following predefined API interfaces; in this way, no matter what are the actual implementations (COUNTDOWN, EAR, EXAMON, OAR, BEO, DCDB,...) of all the REGALE entities (Job Manager, Node Manager, System Manager, Monitoring System,...): these last one will be always able to interact among them, as long as their current (or future) implementations respect, and implement, the REGALE library interfaces.

Future works will focus on extending the integration of the REGALE library with the tools previously presented, following a standardised approach which aims to be the first of its kind.

6. GitLab Links

In Table 4, we report the link of the GitLab repositories of REGALE Power Stack tools.

Tools	Integration Scenario	GitLab link
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 4: GIT repositories of REGALE tools

7. References

[1] https://mqtt.org/mqtt-specification/

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

[3]https://www.esol.com/embedded/services/ros_engineering_services.html?gclid=CjwKCAjwyqWkBhBMEiwAp2yUFgYsicTubBZHNvy0XFhW39oyAne2eapVLd86NgZCvShNhhAQomEe4BoCvq0QAvD_BwE

[4] https://fast-dds.docs.eprosima.com/en/latest/index.html

[5] https://github.com/EEESlab/countdown

[6] D. Cesarini, A. Bartolini, P. Bonfà, C. Cavazzoni and L. Benini, "COUNTDOWN: A Run-Time Library for Performance-Neutral Energy Saving in MPI Applications," in IEEE Transactions on Computers, vol. 70, no. 5, pp. 682-695, 1 May 2021, doi: 10.1109/TC.2020.2995269.

[7] D. Cesarini, A. Bartolini, A. Borghesi, C. Cavazzoni, M. Luisier and L. Benini, "Countdown Slack: A Run-Time Library to Reduce Energy Footprint in Large-Scale MPI Applications," in IEEE Transactions on Parallel and Distributed Systems, vol. 31, no. 11, pp. 2696-2709, 1 Nov. 2020, doi: 10.1109/TPDS.2020.3000418.

[8]https://www.bsc.es/research-and-development/software-and-apps/software-list/ear-energy-management-framework-hpc

[9] Corbalan, Julita, and Luigi Brochard. "EAR: Energy management framework for supercomputers." Barcelona Supercomputing Center (BSC) Working paper (2019).

https://www.bsc.es/sites/default/files/public/bscw2/content/software-app/technical-documentation/ear.pdf

[10] https://fast-dds.docs.eprosima.com/en/latest/fastdds/getting_started/definitions.html

[11] Chakraborty, Mainak, and Ajit Pratap Kundan. "Grafana." Monitoring Cloud-Native Applications: Lead Agile Operations Confidently Using Open Source Software. Berkeley, CA: Apress, 2021. 187-240.

[12]https://gricad-gitlab.univ-grenoble-alpes.fr/regale/tools/ear/-/blob/main/src/slurm_plugin

[13] https://gricad-gitlab.univ-grenoble-alpes.fr/regale/tools/ear/-/tree/main/src/report

[14]https://gricad-gitlab.univ-grenoble-alpes.fr/regale/tools/dcdb/-/tree/master/dcdbpusher/sensors

[15] https://bit.ly/3qT17Yj

[16]https://fast-dds.docs.eprosima.com/en/latest/fastdds/dds_layer/topic/instances.html#dds-layer-topic-instances

[17] Gamblin, T., LeGendre, M., Collette, M. R., Lee, G. L., Moody, A., De Supinski, B. R., & Futral, S. (2015, November). The Spack package manager: bringing order to HPC software

chaos. In Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (pp. 1-12).

]18] https://www.hpc.cineca.it/software/spack

[19] https://spack.io/lrz-using-spack/

[20] Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1995). Design patterns: elements of reusable object-oriented software. Pearson Deutschland GmbH.

[21] ISO/IEC. (2020). ISO International Standard ISO/IEC 14882:2020(E) – Programming Language C++. Geneva, Switzerland: International Organization for Standardization (ISO).

[23] Macenski, S., Foote, T., Gerkey, B., Lalancette, C., & Woodall, W. (2022). Robot Operating System 2: Design, architecture, and uses in the wild. Science Robotics, 7(66), eabm6074.

[24] Tarkoma, Sasu. Publish/subscribe systems: design and principles. John Wiley & Sons, 2012.

[25] Madec, G., Bourdallé-Badie, R., Bouttier, P. A., Bricaud, C., Bruciaferri, D., Calvert, D., ... & Vancoppenolle, M. (2017). NEMO ocean engine.

[26] Dongarra, J. J., Luszczek, P., & Petitet, A. (2003). The LINPACK benchmark: past, present and future. Concurrency and Computation: practice and experience, 15(9), 803-820.