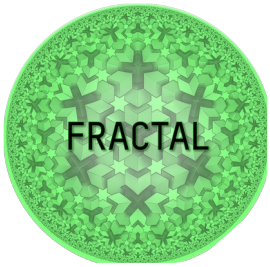


D6.2 FRACTAL edge controller design and implementation

Deliverable Id:	D6.2
Deliverable Name:	FRACTAL Edge controller design and implementation
Status:	Final
Dissemination Level:	Public
Due date of deliverable:	M31 March 2023
Actual submission date:	03/10/2023
Work Package:	WP6 “CPS Communication Framework”
Organization name of lead contractor for this deliverable:	Zylk.net
Author(s):	Alfonso González, ZYLK Vahid Mohsseni, UOULU Anabela Berenguer, UOULU Matti Vakkuri, HALTIAN Antti Takaluoma, OFFCODE Andrianoelisoa Nambinina Rakotojaona, U SIEGEN Pascal Muoka, U SIEGEN Daniel Onwuchekwa, U SIEGEN Mattia Modugno. ROTECH Damiano Vallocchia, ROTECH Nadia Caterina Zullo Lasala, ROTECH
Partner(s) contributing:	ZYLK UOULU HALTIAN OFFCODE U SIEGEN ROTECH

Abstract:

This deliverable is a part of FRACTAL WP6 T6.2, and details the Edge Controller design and implementation, a communication & System monitoring component to optimize the overall system resources of a group of Fractal nodes. The deliverable is divided in three parts, first, the design and architecture of the Edge Controller is shown, second, the implementation aspects are provided, together with installation details and usage guidelines. Third, the Hardware level Edge-Controller and the Runtime Manager components design and implementation are described.

ECSEL Joint Undertaking
Electronic Components and Systems for European Leadership

This project has received funding from the ECSEL Joint Undertaking (JU) under grant agreement No 877056

Co-funded by the Horizon 2020 Programme of the European Union under grant agreement No 877056.

Contents

1 History

Version	Date	Modification reason	Modified by
V0.1	06/08/2022	First version and content proposal	Alfonso González (ZYLK)
V0.2	12/13/2022	Content update	Alfonso González (ZYLK)
Final	01/09/2023	Reviewed document and innovation section included	Alfonso González (ZYLK)

2 Summary

This deliverable covers the main research outcomes from T6.2, which focus on the development of an open-source software component to provide the Edge platform with self orchestration and independence mechanisms at various levels, from the physical hardware (HW-Level Edge Controller), the node (Edge Controller & Custom Orchestrator) and the runtime operations (Runtime Manager). All these mechanisms are built to provide an open-source implementation of the edge controller infrastructure.

There are four main sections (4, 5, 6 and 7) in D6.2:

Sections 4 and 5 are dedicated to describing the Edge Controller and Custom Orchestrator components' design and implementation, respectively. In Section 4 all the design aspects are detailed, with explanations about the design choices made to fulfill with the Fractal characteristics of the platform. Section 5 describes the implementation steps, installation methods, and technical descriptions and aspects of the components. These two sections are divided into three parts, one for each reference platform architecture (High-End node, Mid-Range node and Low-End node).

Section 6 is dedicated to the Hardware-level Edge Controller, which controls the underlying hardware where the software stack is running, This work comes from previous developments in D4.4, and in D6.2 the communications and computations aspects of the HW gateway for multiple-node architectures are included.

Finally, Section 7 covers the Runtime Manager design and implementation, a component fully developed within the Fractal project which is in charge of coordinating and scheduling the operations between multiple running modules.

2.1 Achievements

Throughout this deliverable, the Design and Implementation details of every Fractal component developed during the course of T6.2 are presented.

These components have been developed with the Fractal objectives in mind (Section 3), in order to perform significant research activities and achieve a satisfactory degree of novelty. The resulting Fractal components have a technological relevance for the project and can be used as a primary tech stack by the Use Cases to build the solutions for each problematic.

Highlights

- After the development of the T6.2 components, all of the Fractal reference platforms (High-End, Mid-Range and Low-End nodes) have a custom orchestrator available for each of their tasks.

- The orchestration capabilities have been implemented at all levels: (1) Physical hardware (HW-Level Edge Controller) (2) Application level Edge Controller (Custom Orchestrator and Edge Controllers) and (3) Runtime and communication level (Runtime Manager).
- A complete set of instructions and software tools has been provided to orchestrate the platforms, and no additional tools to support missing functionalities are needed.
- Data ingestion tools are proposed for each of the three reference nodes. For each of the tools proposed, installation and usage steps are provided together with documentation, so that the Use Cases have a wide range of transformation and ingestion tools to extract and load data from and into different data sources.
- Storage tools (databases) are proposed for each of the three reference nodes. Several databases have been selected, installed, and tested into each of the platforms, providing a complete set of options for each of the data sources available. There are relational, non-relational, IoT oriented and time-series oriented data bases as options, so the Use Cases can choose which options suits their solution and data formats.

Lowlights

- The resulting software components may lack of testing and may present bugs or inconsistencies, specially if they are new developments and new code which has not been implemented into productive environments. These inconsistencies will be addressed during T6.3 which is dedicated to testing and validation of the results from T6.1 and T6.2, and will later be implemented into the Use Cases.
- The technological challenge of this task was higher than expected at the beginning of the task. This challenge made it necessary to extend the task such that the resulting components had the necessary quality in terms of code and functionalities, extending the development phases and providing also extra time to generate the adequate documentation in the form of training material, demos, and usage videos.

Results

WP6T62-01 - Data Ingestion

https://github.com/project-fractal/WP6T62-01-data-ingestion

WP6T62-02 - Federated Data Collection

https://github.com/project-fractal/WP6T62-02-Federated_Data_Collection

WP6T62-03 - Run time Manager

https://github.com/project-fractal/WP6T62-03-Runtime-Manager

WP6T62-04 - Hardware Edge Controller

https://github.com/project-fractal/WP6T62-HW-Edge-Controller

WP6T62-06 – Orchestration (Edge controller)

https://github.com/project-fractal/WP6T62-06-edge-controller-orchestrator

WP6T62-06 - Orchestration (Mid-range node orchestrator)

https://github.com/project-fractal/WP6T62-06-mid-range-orchestration

WP6T62-06 – Orchestration (Low-end node orchestrator)

https://github.com/project-fractal/WP6T62-06-low-end-node-orchestrator

3 Introduction

Following the working structure of D6.1, this deliverable has been divided into three main sections, the Edge Controller design, the Edge Controller implementation, and the Hardware level edge controller. each one referring to one of the main processing architectures of the proposed FRACTAL platforms. The Edge Controller is a FRACTAL component both at the software and hardware levels that provides the edge nodes of orchestration mechanisms to be able to schedule workloads dynamically, considering the workload parameters to optimize the available resources.

At the software level (Sections 4 and 5), the Edge Controller was designed and implemented for the three main reference platforms described below:

The first platform is the High-End node, which relies on an ARM64 architecture, and is the most powerful of the platforms in terms of processing capabilities and computation (for example, the Xilinx VERSAL board).

Secondly, the Mid-Range node is an intermediate platform based on RISCV-64 architectures, typically running on less powerful platforms and MPSoCs, but still being able to perform most AI inference operations, low-weight training, and with a low power consumption.

Lastly, the Low-End node is based on RISCV-32 architectures, running on resource restrained platforms (usually PULP platforms) which can perform AI operations with a very low power consumption.

A hardware-level edge controller is a H/W Edge orchestration used in the Gateway of the Network-on-Chip (Interface that connects the on-chip with the off-chip) to schedule the injection time of messages from Network-on-Chip (NoC) to the off-Chip and vice versa. The Hardware-level Edge Controller aims to reduce the message collision in Hierarchical Systems with multiple nodes by using a precomputed schedule. It also ensures the synchronization of Multiple nodes.

Objectives and Approaches

The design of the different Edge controller was done considering the overall objectives of the FRACTAL Project:

Objective 1	Design and Implement an Open-Safe-Reliable Platform to Build Cognitive Nodes of Variable Complexity
Objective 2	Guarantee FRACTAL nodes and systems extra-functional properties (dependability, security, timeliness and energy-efficiency)
Objective 3	Evaluate and validate the analytics approach by means of AI to help the identification of the largest set of working conditions still preserving safety and security operational behaviors.
Objective 4	To integrate fractal communication properties (scale free networks) to FRACTAL nodes.

Table 1: FRACTAL Project Objectives

Given that T6.2 belongs to WP6, which is focused on building a communication framework for the FRACTAL Platform, the Edge Controller focused on complying O4, while also approaching O2, searching that the systems composing the FRACTAL nodes are self and context aware, and can adapt to changes in their own resource capabilities and the external nodes' status.

The SW-level Edge Controller was designed following a two-fold approach, the multi-node communications and the One-node communications.

Multi-node communications refer to different nodes in the same network communicating between each other. In a typical IoT architecture, nodes communicate between them (bottom-up) or directly with a centralized node (top-down) which is in charge of the load and task balancing between the nodes. The Edge Controller design was done such that the components developed in T6.2 can serve both purposes, allowing for bottom-up and top-down orchestrations, with a single implementation.

The main goal of Multi-node communications is to provide the nodes with context awareness, in such a way that the nodes know what the system resources of all of the other collaborating nodes are, being able to fractally adapt to overloads and re-allocate the tasks over free nodes.

One-node communications are all communications happening inside one node between its different components. This means managing the information inside the node itself, without any of this information being exposed to other nodes, and the node itself is taking the appropriate decisions based on this data.

This concept of a node taking decisions on its own status without considering external inputs is called self-orchestration, and is a key capability in intelligent systems, because nodes must be able to operate by themselves even if the communications to all other nodes are lost. The most paradigmatic examples are

self-driving and autonomous cars, which must be able to keep on driving autonomously even if they are going through a tunnel, where there is no signal reception.

The objective of the Edge Controller components is to provide the FRACTAL platform of orchestration capabilities, without the need of a heavy resource-consuming external orchestrator like Kubernetes, and being able to monitor the node resources and take decisions based on the targeted parameters. Secondary objectives are described in Table 2:

Orchestration capabilities	Provide the FRACTAL platform of orchestration capabilities
System resources monitoring	The FRACTAL platform must be able to know the system resources of individual nodes, and also the surrounding nodes (self-awareness and context-awareness capabilities).
Simple extensibility	The design must be done such that the orchestration and resource monitoring capabilities are easily extendable and non-rigid, so they can be broadened in the future with new implementations by the open-source community.
Platform independent	The implementation must be done taking into account that the platforms have architectural and resource differences, but the code must be as inter-operable between platforms as possible to avoid divergency between the developments.
Containerized solution	Container virtualization is the preferred deployment method for the Edge Controller. Other installation and deployment methods on bare-metal and VMs will be studied.

Table 2: Edge Controller objectives

4 Edge Controller design

This section refers to the Software Edge Controller design. The Hardware Edge Controller design and implementation are described in Section 6.

The Software Edge Controller (simply referred to as the Edge Controller during Sections 4 and 5) is a software stack that can be deployed on an arbitrarily large number of nodes (from 1 to potentially any number of nodes) to monitor the status of each of the nodes separately. The information gathered from each of the nodes is then collected and analyzed, giving the system the capability of knowing the overall state of the system, and each node will know the status of each of its neighboring nodes, providing context-awareness and self-awareness capabilities to the Fractal Edge Platform.

During this Section, the design of the Edge Controller is described and its components and main functionalities are explained for each of the reference platforms (High-End node, Mid-Range node and Low-End node).

4.1 High-end node (ARM64)

The Edge Controller design was done following a modular design, in a way that all the components of the Edge Controller can work independently (similarly to a microservice approach) on different nodes. For the High-end and Mid-range nodes, the components were designed and developed trying to keep the compatibility as high as possible, so the developments done for the High-End node will also run properly on the Mid-range nodes. The main differences in design will be at the Ingestion & Storage capabilities, given that the processor architecture is different, some databases and storage tools may not be available for the RISC-V architecture, so alternatives will be studied to cover all the Fractal technological stack.

The High-End node refers to the reference platform for the Xilinx VERSAL board, which has an ARM64-bit architecture.

4.1.1 One-node Edge controller

The One-node Edge controller's goal is to monitor the system resources and translate all the data about the system processor and HW usage into readable data. It was designed based on the Resource Manager microservice, as it can be seen in the following Figure:

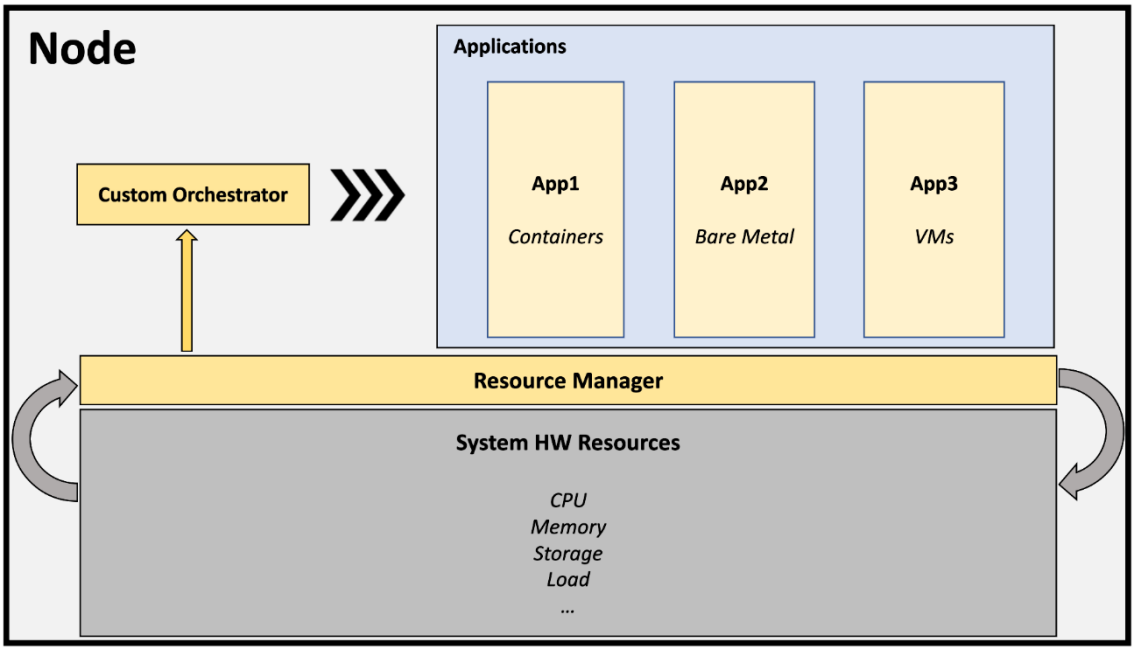


Figure 1: One-node Edge controller architectural design

The Resource Manager is in charge of getting all the information about the Hardware system, CPU usage, memory available, storage, and any relevant information that could be used to monitor the overall system load. To build this microservice, the glances Python package has been targeted, as it is a lightweight tool that can be run on most Linux OSs and is able to provide diverse information like the processor sensor's operating temperatures.

In addition to glances, a second process collects the information about the node and transforms it into a readable format, providing this information to the Custom Orchestrator, which will later make decisions based on the collected information to optimize the system performance by applying restrictions or rescheduling workloads inside the node.

The two main components of the Resource Manager are:

1. Glances https://pypi.org/project/Glances/
2. Edge-monitoring (developed during Fractal T6.2)

Glances is a Python package focused on system monitoring which scrapes and presents a large amount of information through a terminal or web interface. It can be easily installed on most Linux distributions. It is easy to use, utilizes low resources, and can be used in a containerized application.

A container has been created packaging the glances Python package to monitor and expose this information via a node port, where an API is exposed to be requested the information about the system. More details on the implementation can be found on Section 5.1.

The Edge-monitoring tool is a Python program packaged inside another container, which performs periodic requests on the already exposed glances API, and is responsible of gathering all the system information and translating it into a readable format for the custom orchestrator, based on a set of metrics.

Some important aspects about the Edge-monitoring tool:

1. It can be chosen what parameters to monitor, CPU, memory, processor load, or alerts.
2. It must be specified what container orchestrator the node is using, whether it is Kubernetes or Docker (orchestrator-less).
3. Hostname, port, IP, set of resources to monitor and orchestrator can be provided via a YAML file.

This architectural design is completely scalable, as it will be shown in the design of the Multi-node Edge controller, which uses this modular microservices to build a system that monitors the resources of potentially any number of nodes.

More details about the implementation can be found on Section 5.1.

There is another microservice that works together with the Resource Manager to complete the Fractal Orchestration capabilities, the Custom Orchestrator.

The Custom Orchestrator is a (containerized) system to monitor the containers, tasks, and processes being run on your system processor. It can dynamically schedule, start, stop, and up or down scale your processes. for the high-end and mid-range nodes. It has been designed in a modular way, this means that it can be operated from both architectures following the same design, and orchestrate containers from a docker host or K8S control plane, having very similar capabilities independently of the processor architecture and the reference platform being used, and being able to use both docker and K8S as container orchestrators, implementing orchestration capabilities for docker hosts and improving the already existing K8S orchestrating operations by adding custom orchestration capabilities and a full awareness of the status of each of the nodes in the cluster (from 1, to N number of nodes).

For deployments consisting of a single node cluster (One-node Edge controller deployments), the custom orchestrator is deployed together with the resource manager and the node exporter (glances container) in the same node. This is done by building and deploying three container images which work together as a typical microservices architecture. The metrics exporter container is in charge of gathering all the information about the CPU and hardware available resources, then, these resources are used by the resource manager container and an updated tainted nodes list is sent to the custom orchestrator container, who ultimately is in charge of applying the appropriate restrictions depending on the container orchestrator that is specified in the node's configuration file.

4.1.2 Multi-node Edge controllers

To keep the architecture as modular and re-usable as possible, the Multi-node Edge controller was built as an extended capability of the already designed One-node Edge controller, where the Resource Monitoring tool is deployed to all the nodes of the Fractal environment to be monitored. Then, these nodes are monitored by a dedicated Master node which will gather the information of every node.

The Multi-node Edge controller architecture is described in the Figure below:

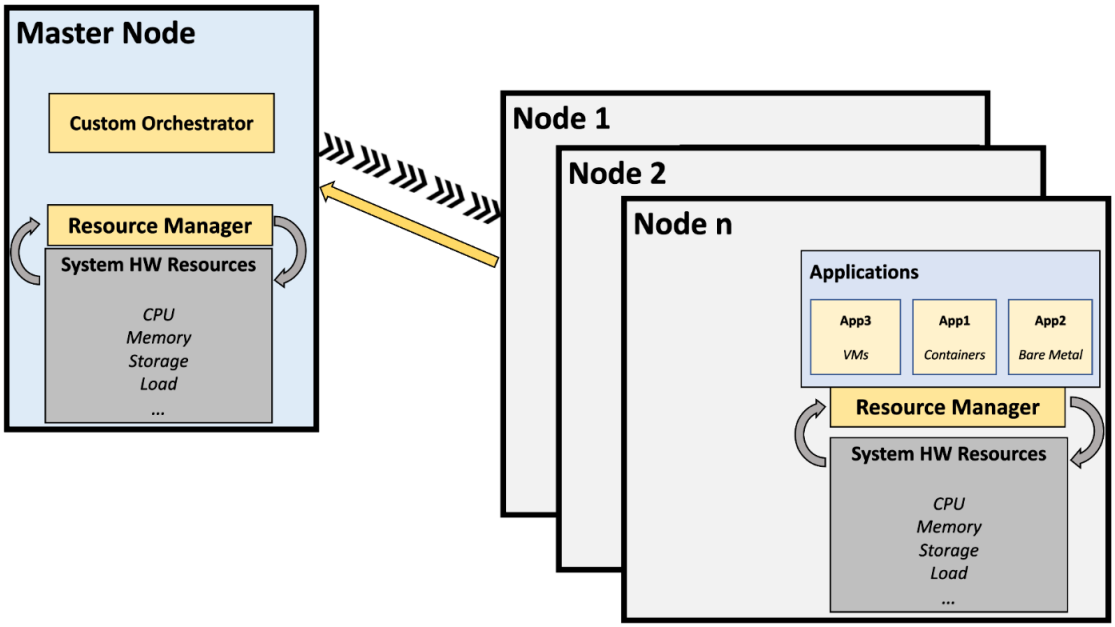


Figure 2: Multi-node Edge controller architectural design

As it can be seen in the design architecture, there is an arbitrarily big number of nodes, and each node is running a Resource Manager instance. The Resource Manager consists of the previously described glances container, which gathers and exposes the HW system information via a dedicated web server with a REST API. The Master Node will then be running the Node-monitoring Resource Manager container, with a provided YAML configuration file, where all the nodes on the Fractal environment are specified, and the Master Node will gather the resource information for each node, even if these nodes are disconnected temporarily from the network or new nodes are added. Note that, as the Resource Manager is able to run both the glances container and the node-monitoring tool, every single node on the cluster can still monitor itself, or other nodes, which means that each node is fully aware of the status of:

1.- Itself

2.- Surrounding nodes

3.- Overall resource status of the whole cluster

Thus, providing the self-awareness and context-awareness capabilities of the Fractal platform.

Finally, the Custom Orchestrator being run on the Master node is in charge of managing the K8S API so that enhanced orchestration methods can be applied into the cluster. This is achieved by deploying a pod or container which is in charge of getting an updated list of tainted notes from the resource manager, and applying certain actions on all the K8S resources which it is given permissions to modify. The usual set-up would be a custom orchestrator pod which is in charge of applying the restrictions to the control plane and master node, so when the tainted node list gets updated in the custom orchestrator container, these taints can directly be applied. The most usual taint is “No-Schedule”, which avoids new pods to be created into the restricted node.

4.1.3 Custom Orchestrator

The custom orchestrator is designed in a way that keeps the compatibility between the different architectures, i.e., ARM, RISC-V, and X86-64, as much as possible. To achieve this matter, the communication protocol is implemented from scratch by defining the message schema handled by sockets directly. This new implementation requires no external libraries and dependencies for the systems. The design also follows the principle of microservice programming and makes the scalability of the systems easy. There are three primary services. As illustrated in Figure 3, the first part is the API server, the second component is the service manager, and the last is the executor daemon. The main functionality of each service is explained as follows.

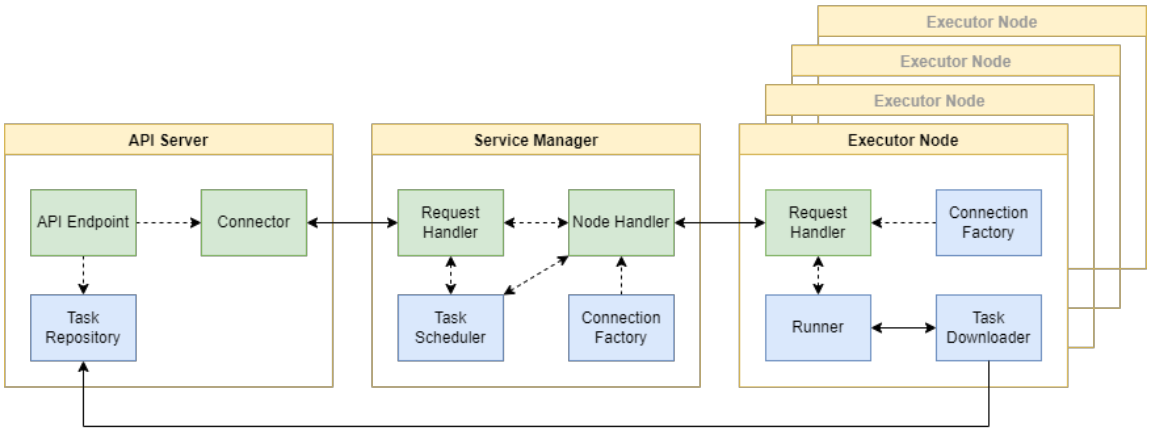


Figure 3: The general architectural view of the Orchestrator

- API server: As the name of this component hints, it is developed to make user interaction with the system more accessible. It sends the requests directly to the manager service and does not communicate directly with the nodes on which the daemon service is running. In this way of the

communication model, the service provides separate storage or a repository for future tasks.

- Service Manager: The Service Manager is the core component of the system here. From the architectural point of view, it has four essential functions. The connection factory is responsible for handling connections from the executor nodes. It accepts new connections and passes its instance to the node handler. The node handler maintains the connection, tries to keep it up, and makes the executor node ready for other procedural activity by submitting that information to the task scheduler. The task scheduler is closely related to the node handler and always waits for new tasks incoming from the API server, which is handled by the request handler. The request handler bridges the API server and the service manager. Any requests made by the user will be delivered and governed by this function in the service manager.

- Executor: This piece of software is the part of the node/container/machine on which you want that task to be run. It constantly communicates with the Service Manager, sends its status, and waits for a task to be assigned by keeping the socket connection open to the server. By decoupling this component from the Service Manager, it is easier to scale out or in the runners according to the requirements of the load in the system. The time to wait for a runner to be up and running is eliminated by provisioning them before scheduling any tasks. Practically, it is possible to add up as many runners as we need, and a single instance of the Service Manager will handle the load.

4.1.4 Ingestion & Storage capabilities

Ingestion & Storage capabilities refer to the set of tools that are available for the Fractal platform and that will be used to collect data from different data sources and store these data in the most optimal database for each use case.

Here is provided a set of tools for ingestion & storage together with a description of each tool and their main features. Instructions on how to implement and install these tools can be found in Section 5 (implementation).

Ingestion

Data ingestion becomes a crucial task in IoT and Edge computing scenarios where the data sources are very diverse, and relying on powerful and flexible ETL (Extract, Transform, Load) tools is a must in the Fractal Platform. Multiple choices are provided for data ingestion tools, so the vast majority of scenarios can be approached.

For the High-End Node, it must be noted that ARM64 is the most popular processor architecture for IoT devices and Edge computing frameworks. This is why it has the most complete set of tools to be installed. For this component, two open source ETL tools are proposed, one for Java and one for Python programming languages:

Java based tool:

Apache NiFi

Although Java is not the preferred programming language in Edge computing architectures, an exception is to be made with Apache NiFi as the overall best performing open-source ETL tool available. It is widely used for data stream processing and counts with a wide and active community. If an ETL and stream processing tool is required, Apache NiFi is highly recommended and reliable.

Python based tools:

In case that Python is preferred over Java for your specific application, or Apache NiFi lacks any specific functionality required for your use case, two other open source and Python-based alternatives are given:

Apache Spark (PySpark)

Apache Spark is a multi-language engine for executing data engineering, data science, and machine learning on single-node machines or clusters.

Faust

Faust is a stream processing library that processes from Apache Kafka streams, built in Python. It supports the following extensions:

Name	Version	Bundle
Rocksdb	5.0	pip install faust[roscksdb]
redis	aredis 1.1	pip install faust[redis]
datadog	0.20.0	pip install faust[datadog]
statsd	3.2.1	pip install faust[statsd]
uvloops	0.8.1	pip install faust[uvloop]
eventlet	1.16.0	pip install faust[eventlet]
yaml	5.1.0	pip install faust[yaml]

Table 3: Available Faust extensions

The main advantage of Faust is that it is a Python library and fully based on Python, so you can integrate it with any other Python library or system as long as you have a Kafka stream to process. Faust can publish and consume from Kafka streams and perform stream processing operations in a distributed manner.

Others:

A third alternative is given as a flow-based programming tool, *Node-RED*, written in JavaScript and commonly used as an IoT data flow tool because of the big number of plugins available from the community.

Node-RED consists of a Node.js based runtime that you point a web browser at to access the flow editor. Within the browser you create your application by dragging nodes from your palette into a workspace and start to wire them together. Then. with a single click the application is deployed back to the runtime where it is kept running.

Storage

Data storage refers to open-source databases available to install to the ARM64 processor architecture, providing a complete set of options to cover a wide range of applications into IoT and Edge deployments.

For ARM64 Linux-based devices, two open source data collections tools are proposed, one being relational and the other being a non-relational database.

Relational Database: CrateDB

A relational database, stores information in tables. Often, these tables have shared information between them, causing a relationship to form between tables. This is where a relational database gets its name from.

CrateDB is a distributed SQL database management system. It is designed for high scalability since it is open source and written in Java, and it includes components from Facebook Presto, Apache Lucene, Elasticsearch, and Netty. CrateDB was developed with the intention of putting IoT data to use and supports IoT data analytics: Time series, AI, geospatial, text search, joins, aggregations, etc.

Non-Relational Database: MongoDB

A non-relational database, sometimes called NoSQL (Not Only SQL), is any kind of database that doesn’t use the tables, fields, and columns structured data concept from relational databases. There are various types of NoSQL databases.

MongoDB is a document-oriented database software. It is classified as a NoSQL database application. It makes use of JSON-style documents with schemas.

Other alternatives

The two databases described above are the recommended databases for storing data in the Fractal High-End node, and the ones for which an installation guide is

provided. However, the list of available databases is very extensive and a list of good alternatives is provided:

InfluxDB: NoSQL database for Time-Series data.

Apache Cassandra: High-performance NoSQL database.

Apache IoTDB: High-performance database for Time-Series data.

SQLite: Highly-portable embedded relational database.

4.2 Mid-range node (RISC-V64)

The Mid-Range node refers to FPGA boards and MPSoCs with a RISC-V64-bit architecture. The Edge Controller has been designed to be compatible with both the High-End and Mid-Range nodes. This means that the previously described design is also applicable to Fractal nodes with a RISC-V64 architecture processor.

For validation purposes, the Python glances module was installed in a RISCV node with a Linux OS, obtaining similar results to the ones in the ARM64 devices, and containers running in the RISC-V machines showing similar performances.

The rest of the components are Python scripts and programs which will be able to be executed in any node with Python installed, no matter what the processor architecture is.

4.2.1 One-node Edge controller

Figure 1 shows the One-node Edge controller architecture design for the High-End node. This architecture is still valid for Mid-Range nodes, based on a Resource Manager that collects and processes the information about the processor's status, and a Custom Orchestrator which takes actions based on this information.

The main difference between the One-node and Multi-node edge controllers for both reference platforms the high-end node and the mid-range node is the container orchestrator to be used by the system. Although there are some lightweight K8S distributions for RISCV64 architectures available, they are not still mature enough to be used for production solutions. For this reason, Docker has been chosen to be the container manager platform. Based on some rules determined by the host's resources, containers can be dynamically scheduled on docker hosts which have their daemons exposed. These rules are based on CPU and memory limits which can be set for restrictions to be applied.

For each loop in the execution of the tainted nodes list update, the customer orchestrator takes into account if the nodes were previously tainted in the past or they have been recently tainted. For each of these two cases, different actions are taken, in an increasing degree of restriction. For newly tainted nodes, the already running containers and tasks in the host have their resources limited by applying restrictions in the number of CPUs that they can use and also by restricting the

available memory for them. In the next iteration, it will be checked whether the node was already tainted in the past or not, and in that case, containers will be periodically stopped and rescheduled into the other available nodes if any, until the orchestrated node is free in resources again. When the nodes are finally untainted, restrictions on them are lifted, but the overall edge controller orchestrator is still monitoring their resources in case any further orchestration is needed.

4.2.2 Multi-node Edge controller

For edge controller deployments in multiple nodes architectures, where each of them is running a Docker instance which is in charge of managing their containers that are being run, the Custom Orchestrator is able to apply a set of restrictions on the nodes separately, rescheduling containers from exhausted nodes into nodes with a lower process pressure.

Containers can be dynamically stopped and started between the different nodes, maintaining the overall pressure of the cluster low enough for all the processes to be executed properly. Before containers are stopped on an exhausted node, some preventive measures are taken, for instance, when a single node appears to be exhausted on resources, all the containers running on that host see their available resources limited in terms of CPU and memory. If these measures are not enough and the host is still exhausted by the execution of the container processes, these containers are reallocated into the rest of the nodes.

4.2.3 Custom Orchestrator

As mentioned in section 4.1.3, the current design and implementation of the custom orchestrator let the component run smoothly on different architectures, including the RISC-V. The acquired libraries in this component are the generally available python libraries that come with its installation. Moreover, the system calls featured in this component are chosen to be as general as possible to maintain compatibility between the OS distributions.

In some cases, the mid-range nodes are located behind a NAT network. In this case, the means of direct control of the nodes are limited. Without a pingable IP address of the node, it is impossible to send commands or updates to the nodes. To address this issue, the connection is always established by the executor node. It should be mentioned that the Service Manager must have an IP address that can be recognizable by those nodes. When the connection is established from the side of the executor, the Service Manager will maintain the connection by sending and receiving primary status data; for example, it can be the information about the memory and CPU (that can be informative for the decision of the scheduler).

4.2.4 Ingestion & Storage capabilities

For the mid-range node, it must be taken into account that some of the packages required to install the different ETL and database tools and databases may not be available for the RISCV64 processor architecture.

Implementation and installation steps are provided for the available tools in Section 5, together with references to the official documentation which provides building steps in case the source code ought to be built from source.

For ETL tools, the same tools are proposed for both the High-end and Mid-range nodes, so the installation may result more complex in RISCV systems because of the non-availability of pre-built packages. In the case of database and storage tools, options which have available packages for the RISCV64 architecture are provided which simplify the installation.

Once installed, the usage and functionalities of the recommended tools should be the same in both the Mid-Range and the High-End node, only differing in the computing capabilities and limitations of one platform over the other.

5 Edge controller implementation

This section is devoted to the implementation of the Edge Controller components. The implementation will include both the code design details, giving thorough information about how the different components are built and work from the inside, how they connect and work together and how the interaction between the different components brings context-awareness and self-awareness capabilities to the Fractal platform where they are deployed.

Finally, it will include the installation steps and how-to-use starting guides, for the users of the Fractal platforms to have a reference document to visit when implementing the Edge Controller into their technological stacks.

5.1 High-end node (ARM64)

5.1.1 One-node Edge controller

As described in Section 4.1.1, the Edge controller is composed of two main services, the Resource Manager and the Custom Orchestrator.

The **Resource Manager** is the component in charge of collecting and processing all the information about the system status. This is done through the glances Python package, which is executed in the web server mode by the *glances –w* command.

This command exposes a REST API in port 61208 (which has been chosen as default) with all the collected information. This API is documented at https://github.com/nicolargo/glances/blob/fieldsdescription/docs/api.rst , where all the information about what do the monitored resources mean can be found.

This REST API has been containerized with the following Dockerfile:

```
FROM ubuntu:21.10

RUN apt-get update && DEBIAN_FRONTEND="noninteractive"
TZ="America/New_York" apt-get install -y tzdata

RUN apt-get install glances -y

RUN useradd -ms /bin/bash metricsexporter

USER metricsexporter

WORKDIR /home/metricsexporter

EXPOSE 61208
```

```
COPY glances.conf /etc/glances/glances.conf

ENTRYPOINT glances -w
```

Notice that the base image is ubuntu:21.10, but any other Linux distro will be fine if there is a preference. During the image build, a user metrics exporter is created to avoid the container to be running with the root user, glances is installed and then executed as a web server.

To deploy this container into a Docker host, run the following command:

```
docker run -d --network=host --pid=host --hostname
metricsexporter <image_name>
```

Where <image_name> is the name of the Docker image you built before (by default it will be metrics-exporter), and the rest of the flags mean:

-d : Run this container in detached mode, so it will not use the current terminal session.

--network=host : Use the container host's networking interface to expose the API. This parameter is required to be able to access the API from outside the container (especially in the Multi-node Edge Controller)

--pid=host : This allows the container to access the host's processes. This way the rest of the host's processes and containers can be monitored, otherwise only the glances container process will be shown which is of no interest for monitoring purposes.

--hostname metricsexporter : Set the container's hostname.

Once the glances container is deployed on the node to be monitored, another container has to be deployed, with all the software and scripts to make use of the information exported by the glances container.

This is done through the following Dockerfile:

```
ARG PYTHON_VERSION=3.9.13

FROM python:${PYTHON_VERSION}

RUN useradd -ms /bin/bash resource_manage
```

```
USER resource_manager

WORKDIR /home/resource_manager

COPY        --chown=resource_manager:resource_manager        utils
/home/resource_manager/utils

COPY                     --chown=resource_manager:resource_manager
requirements.txt /home/resource_manager/requirements.txt

COPY     --chown=resource_manager:resource_manager     nodes.yaml
/home/resource_manager/resource_manager/nodes.yaml

COPY                     --chown=resource_manager:resource_manager
resource_manager.py /home/resource_manager/resource_manager.py

RUN python3 -m pip install --upgrade pip

RUN pip install -r requirements.txt

ENTRYPOINT python3 resource_manager.py
```

As in the glances container, a dedicated user is created in the image build to avoid using the root user. Then, all the necessary files for the Resource Manager to monitor the exposed information by the metrics-exporter container are copied into the container, and finally the resource_manager.py script is executed.

The resource_manager.py script is where the main loop of the component is executed. First, the nodes.yaml file is read with all the information about the nodes to be monitored. The syntax and usage of this file will be detailed in the Multi-node Edge controller section. Then, the nodes availability is checked, and for the nodes not available, a list of down-nodes is created.

For the alive nodes, the information is requested to their respective glances REST APIs, and this information is parsed into manageable formats, and then processed by importing the infotreatment.py.

Once the information has been parsed into Python objects, it is sent to the actions.py script, which takes decisions based on pre-defined rules.

For example, if CPU and memory are being monitored, alerts will be sent by the program if any of these parameters go over 80%. These percentages and alerts are customizable and can be adjusted fit the user needs.

Finally, if any of the nodes is noted to have an active alert, it is listed into the Tainted nodes list, which will be handled by the Custom Orchestrator.

Once the Custom Orchestrator comes into play, it first reads the Tainted nodes list, which refers to the list of nodes that have some of their parameters too high or exhausted on resources so they need actions to be taken on them.

5.1.2 Multi-node Edge controller

It was mentioned in the previous section that a file called ***nodes.yaml*** is copied inside the resource manager container. This file is where all the nodes to be monitored are included, detailing what resources must be monitored from each of them, what orchestrator they are using and other useful information for the resource manager script to be able to find the nodes. This is an example of a 2-node configuration:

```
- node:
    hostname: "fractal-node1"
    IP: 192.168.0.83
    port: 61208
    orchestrator: "docker"
    resources: ["cpu","mem"]

- node:
    hostname: "fractal-node2"
    IP: 192.168.0.111
    port: 61208
    orchestrator: "kubernetes"
- custom-orchestrator:
    hostname: “fractal-orchestrator”
    IP: 192.168.0.23
```

With this configuration, both nodes will be monitored, but notice that the first node has a dedicated array of resources to be monitored, including “cpu” and “mem”. In this case, only these two resources will be monitored, and for the second node, all the resources will be monitored, including cpu, mem, processing load and alerts given by the glances service. For deployments where a custom orchestrator

container is running, another node must be added to the nodes.yaml configuration file. This custom orchestrator node will give the resource manager the information about what host is running the custom orchestrator container, so that it can send the updated tainted notes list. For One-node deployments, the customer orchestrator will of course be the host running all the orher services and being orchestrated by itself

The main difference between the deployments in One-node and Multi-node architectures is that this nodes.yaml file must be provided with a list of nodes to monitor in the Multi-node architecture, while in the One-node, the nodes.yaml file must only contain information about the node where the Edge Controller is being run (hostname can even be set as localhost if running outside a container).

For Multi-node deployments, the Resource Manager will be aware of the status of all surrounding nodes specified in the nodes.yaml file, and it will be in charge of sending an updated list of Tainted nodes and exhausted nodes to the Custom Orchestrator, which will then take the necessary actions.

The Edge Custom Orchestrator will receive this list of tainted nodes together with the resource which is the cause of the alert, so the appropriate actions can be taken.

If the node is being orchestrated by K8S, the custom orchestrator pod is able to apply paints on the control plane and master node, so that node is no longer able to schedule pods until the taint is lifted. For the custom orchestrator pod to be able to apply taints to a given K8S node, it must be granted permissions by the role-based access control policies. Other actions that this pod can take include rescheduling pods, creating deployments, services, and virtually any other resource that should be orchestrated externally to ensure that the system resources stay low enough for the processes to be completed normally on each of the cluster nodes.

5.1.3 Custom Orchestrator

In the previous sections, it was mentioned that this component has three sub-components with different functionalities and specific roles. In order to make this component run in the cluster of nodes, first, it is important to have a view of the nodes on which this component will run. The proposed single-node and multi-node views are both supported by this component. However, it is required that the master node has an IP address reachable by other nodes in the system. So, the first component to be initialized is the Service Manager. The following instructions provide a step-by-step guide to set it up. First, it is required to clone the repository from the GitHub.

```
$ git clone https://github.com/vahidmohsseni/k8s-manager

$ cd k8s-manager/backend-service-manager
```

```
$ pip install –r requirements.txt

$ python service.py
```

The above commands will run the main service in a machine. Note that this service will use port 5555 (to listen to the API Server) and 5556 (to listen to the Executors) of the machine, so it is necessary to check if it is not already in use. If the port is unavailable, it can be changed from the file `server.py`.

The next step is to make the API server run and connect to the above service. The API server can be run on any node which can have network access to the machine that the service manager is running on. For the following guide, it is assumed that the code base is already cloned.

```
$ cd k8s-manager/api

$ pip install –r requirements

$ python app.py
```

The written steps will run the API server on port 5001 of the machine. Again, if necessary, it is possible to change the port to any arbitrary number. The example assumes that the API server and service manager are running on the same machine with the default configuration. However, the mechanism for changing the IP address of the destination is available in the `controller/v1.py` directory. In case there is a change in the port of the service manager, it is a must to replace the default port in the directory mentioned above.

The final service to be run is the executor service. The executor service should be running on those nodes that will run the tasks and functions in the cluster of the devices. The number of nodes can be any number starting from 1. Although this custom orchestrator is designed to handle large-scale applications, if the number of nodes is more than 500, the service manager's node requires at least 1 GB of bandwidth and more than 16 processing cores.

```
$ cd k8s-manager/frontend-service-manager

$ pip install –r requirements.txt

$ python service.py
```

In case of a change port in the service manager, the default configuration of these executors should be changed. And once more, the executor nodes must have network access to the service manager.

All the functionalities provided by the service manager are accessible through the API server. The API server has a RESTful design so that a frontend UI-based application can be developed for it to take the ease-of-use advantage of the system. However, the following list provides the curl-based API calls for the system to run any tasks on the nodes. It should be mentioned that the requirements for the tasks (to be run on those executors) must be already satisfied. In other words, the executor nodes should be capable of running the tasks with the given commands.

Explanation	URL	`curl` parameters
Get list of tasks	`/api/v1/tasks/`	`-X GET`
Create a new task	`/api/v1/tasks/<task_name>`	`-X POST -F "file=@<filename>" -F "cmd=<args for python including file name>" -F "rt=<return type>"`
Delete a task	`/api/v1/tasks/<task_name>`	`-X DELETE`
Stop a task	`/api/v1/tasks/<task_name>/stop`	`-X POST`
Start a stopped task	`/api/v1/tasks/<task_name>/start`	`-X POST`
Check Status of a task	`/api/v1/tasks/<task_name>/status`	`-X GET`

Table 4: Custom orchestrator API reference

5.1.4 Ingestion & Storage capabilities

For the High-End node, all the chosen ETL and database tools have pre-built packages that can be directly installed on ARM64 devices without needing to build the code from source. Here we provide installation and usage steps for each of the tools described in Section 4.1.3:

Ingestion

Prerequisites

Make sure that your system has the following available dependencies before installing each of the ingestion tools

Apache NiFi

- Requires Java 8 or Java 11
- Supported OS:
  - Linux, Unix, Windows, macOS
- Local installation:
  - Download the binaries or Sources from the Official Download Page
- Docker installation:
  - Download the images from the Official NiFi DockerHub

Apache Spark (PySPark)

- Python3.8 or above
- pip3
- Java 8 or later with JAVA_HOME section
- For ARM users, PyArrow is required for PySpark SQL, if PyArrow installation fails, try installing PyArrow >= 4.0.0

Node-RED

- Local installation: A supported version of Node.js.
- Docker installation: Docker Engine
- From source:
  - A supported version of Node.js
  - A git client
  - The grunt-cli npm module installed locally.

Faust

- Python3.6 or above
- A running Kafka broker

Installation steps

Apache NiFi

- Download

First, and assuming you are working on a Linux OS, download the tarball file from https://nifi.apache.org/download.html . Then, decompress the file into the desired installation directory.

- Starting Apache NiFi

Decompress and untar into desired installation directory

Make any desired edits in files found under /conf to match your deployment requirements.

At a minimum, it is recommended editing the nifi.properties file and entering a password for the nifi.sensitive.props.key (see System Properties below). This nifi.properties file has many configuration aspects that should be reviewed before starting the application, like ports where to expose the HTTP/HTTPs user interfaces, user/password credentials, network interfaces to use, etc.

Once you have configured your NiFi environment, from the /bin directory, execute the following commands by typing ./nifi.sh <command>:

- start: starts NiFi in the background
- stop: stops NiFi that is running in the background
- status: provides the current status of NiFi
- run: runs NiFi in the foreground and waits for a Ctrl-C to initiate shutdown of NiFi
- install: installs NiFi as a service that can then be controlled via systemctl as:
    - service nifi start
    - service nifi stop
    - service nifi status

These commands are used to control your application process, stop and restart the service, or check the application status. Once your application is running, you can visit the UI at (by default) https://localhost:8000.

By default, the installation script generated a random username and password that can be edited in the nifi.properties file. If using the default configuration (which is highly deprecated), the credentials can be found in the application logs at logs/nifi-app.log, under the Generated Username and Generated Password lines.

To change the Username and Password you can execute the command:

```
$ ./bin/nifi.sh set-single-user-credentials <username>
<password>
```

And then access the dashboard where you can create your dataflows from the User Interface at https://localhost:8443/nifi (again, it is recommended to change this default port in the NiFi properties file).

- Docker container deployment

Once you are familiar with how NiFi works and are able to install it and deploy the application on bare metal servers, you are ready to deploy NiFi as a Docker container. While you can directly run a Docker container, it is highly encouraged to install the application first to get a deeper insight on how it works and how to configure it.

The Official Docker Image can be found on NiFi's Official DockerHub, where you can also find all the required information to configure and run the container.

PySpark

Apache Spark is a multi-language engine for executing data engineering, data science, and machine learning on single-node machines or clusters.

PySpark is available as a Python3 package, it can be installed by executing:

```
pip install pyspark
```

Take into account that Spark requires the following Python package dependencies:

Dependencies

Package	Minimum supported version	Note
pandas	1.0.5	Optional for Spark SQL
NumPy	1.7	Required for MLlib DataFrame-based API
pyarrow	1.0.0	Optional for Spark SQL
Py4J	0.10.9.5	Required
pandas	1.0.5	Required for pandas API on Spark
pyarrow	1.0.0	Required for pandas API on Spark
Numpy	1.14	Required for pandas API on Spark

Figure 4: Apache Pyspark dependencies

NodeRed

To install NodeRed there are two possibilities:

- Installing with npm

```
sudo npm install -f --unsafe-perm node-red
```

```
```

And confirm the installation was successful in the end of the command output:

```
+ node-red@1.1.0

added 332 packages from 341 contributors in 18.494s

found 0 vulnerabilities
```

- Installing with Docker (recommended)

```
docker run -it -p 1880:1880 -v node_red_data:/data --name
mynodered nodered/node-red
```

This command will create a Docker container and a data volume at /data for your node-red container. Going to http://localhost:1880 will bring you to the User Interface

Faust

To install Faust, just run:

```
pip install faust
```

This command will install Faust in your system as a Python package.

Storage

Installation

CrateDB

- Package-based method (Linux)

This method is suitable for Debian, Ubuntu, RedHat and CentOS based systems.

- Debian/Ubuntu

First of all you will need to add CrateDB package repository to your system.

```
# Install prerequisites.

apt-get install sudo

sudo apt-get install curl gnupg software-properties-common
apt-transport-https apt-utils

# Import the public GPG key for verifying the package
signatures.

curl -sS https://cdn.crate.io/downloads/deb/DEB-GPG-KEY-crate
| sudo apt-key add -

# Register with the CrateDB package repository.

[[ $(lsb_release --id --short) = "Debian" ]] &&
repository="apt"

[[ $(lsb_release --id --short) = "Ubuntu" ]] &&
repository="deb"

distribution=$(lsb_release --codename --short)

sudo add-apt-repository "deb [arch=amd64]
https://cdn.crate.io/downloads/${repository}/stable/ $
{distribution} main"
```

Now update the package sources:

```
sudo apt update
```

You should see a success message. This indicates that the CrateDB package is successfuly registered. Now you can install CrateDB:

```
sudo apt install crate
```

Once installed you can control the crate service with systemctl utility program:

```
sudo systemctl <command> crate
```

Replace COMMAND with start, stop, restart, status and so on.

- Red Hat/CentOS

First of all, you will need to add CrateDB package repository to your system.

Install prerequisites.

```
yum install sudo
```

Import the public GPG key for verifying the package signatures.

```
sudo rpm --import https://cdn.crate.io/downloads/yum/RPM-GPG-
KEY-crate
```

Register with the CrateDB package repository.

```
sudo rpm -Uvh
https://cdn.crate.io/downloads/yum/7/x86_64/crate-release-7.0-
1.x86_64.rpm
```

With everything set up, you can install CrateDB:

```
sudo yum install crate
```

Once installed you can control the crate service with systemctl utility program:

```
sudo systemctl <command> crate
```

Replace COMMAND with start, stop, restart, status and so on.

- Docker

CrateDB and Docker are great matches thanks to CrateDB's shared-nothing, horizontally scalable architecture that lends itself well to containerization.

In order to spin up a container using the most recent stable version of the official CrateDB Docker image, use:

```
docker run --publish=4200:4200 --publish=5432:5432 crate
```

MongoDB

- Package-based method (Linux)

Assuming you are using an Ubuntu system, import public GPG Key for MongoDB using:

```
wget -qO - https://www.mongodb.org/static/pgp/server-6.0.asc |
sudo apt-key add -
```

The operation should respond with an OK.

Create the list file /etc/apt/sources.list.d/mongodb-org-6.0.list for your version of Ubuntu. The following example is for Ubuntu 20.04 (Focal).

```
echo "deb [ arch=amd64,arm64 ]
https://repo.mongodb.org/apt/ubuntu focal/mongodb-org/6.0
multiverse" | sudo tee /etc/apt/sources.list.d/mongodb-org-
6.0.list
```

Reload the local package database.

```
sudo apt update
```

Install MongoDB packages:

```
sudo apt-get install -y mongodb-org
```

Once installed you can control the MongoDB service with systemctl utility program:

```
sudo systemctl COMMAND mongod
```

Replace COMMAND with start, stop, restart, status and so on.

- Docker

MongoDB can run in a container. The official image available on DockerHub contains the community edition of MongoDB and is maintained by the Docker team.

In order to spin up a container using the most recent stable version of the official MongoDB Docker image, use:

```
docker run --name mongodb -d -p 27017:27017 mongo
```

5.2 Mid-range node (RISC-V64)

5.2.1 One-node Edge controller

For the RISCV architecture, there are steps to follow to build the edge orchestrator are essentially the same that must be followed in the high-end node to build the necessary containers. The resource manager and the metrics exporter should be the same container images but built dedicatedly for different processor architectures. However, the custom orchestrator container plays a major role in RISCV platforms, because there is no stable K8S distribution available for these nodes to take care of dynamic container deployment and management. Docker Engine must be used as the container manager, and the custom orchestrator will be the system in charge of providing Fractal orchestration capabilities to the host.

The custom orchestrator container image is built with the following Dockerfile:

```
ARG PYTHON_VERSION=3.9.13

FROM python:${PYTHON_VERSION}

RUN useradd -ms /bin/bash custom-orchestrator

COPY --chown=custom-orchestrator:custom-orchestrator utils /home/custom-orchestrator/utils

COPY --chown=custom-orchestrator:custom-orchestrator requirements.txt /home/custom-orchestrator/requirements.txt

COPY --chown=custom-orchestrator:custom-orchestrator custom_orchestrator.py /home/custom-orchestrator/custom_orchestrator.py

RUN pip install --upgrade pip

RUN pip install --no-cache-dir -r /home/custom-orchestrator/requirements.txt
```

```
USER custom-orchestrator

WORKDIR /home/custom-orchestrator

ENV FLASK_APP=custom_orchestrator.py

ENTRYPOINT flask run --host=0.0.0.0 --port=9999
```

During the execution of the container, a Flask API is run, which contains all the required methods for the resource manager to send the updated list of tainted notes and the custom orchestrator to apply the adequate actions over the Docker daemon.

When a node is included in the list to be tainted for the first time, the first action to be taken by the orchestrator is to limit all the resources that containers being run on the node have access to, by limiting the CPUs and free memory available for these containers. if the node is not untainted for a given period of time, then the orchestrator will take more severe actions, stopping the containers one by one until the node gets out of the tainted list. Once the processor pressure gets low enough and the node is finally untainted, that restrictions over the node are lifted, except for their resource limiting on the containers being run, to avoid the processor from overloading again.

5.2.2 Multi-node Edge controller

For Multi-node edge controller deployments, the nodes.yaml configuration file must be provided to the resource manager container, the same way it is done for the high-end node platform deployment. The resource manager will use the configuration file to get a complete list of the nodes to be monitored and being part of the cluster, and also to know which node will be in charge of running the custom orchestrator container.

Once the deployment has been done, with the metrics exporter being run on each node of the cluster, the resource manager scraping the information about the system processor for each node, and the custom orchestrator running on the master node of the cluster, the setup is ready to start orchestrating on Docker Engine hosts by managing the containers being run, stopped, and started on each of the nodes, while also limiting the resources available for exhausting containers.

The Edge Controller ultimately allows the Fractal node to have dedicated orchestration capabilities by running a piece of software which also makes the platform and the overall system context aware. Each of the nodes in the cluster has access to real-time information about the state of itself and the state of all the other nodes in the fractal environment. It also overcomes the difficulties that arise from running containerized solutions in platforms which have no support or stable K8S distributions.

5.2.3 Custom Orchestrator

In the design section, it was noted that the design of this component uses the native and general libraries of python to maintain compatibility among different hardware architectures. So, as a general rule, the component can run with the exact instructions provided for the high-end nodes. However, the decoupled implementation of this component allows for running the components on different machines. With this feature, executing the service manager on a node with enough resources and fault tolerance is recommended.

5.2.4 Ingestion & Storage capabilities

For the Mid-End node, all the chosen ETL and database tools the software packages must be downloaded from the official repositories and built from source following the instructions on the documentation.

Ingestion

Prerequisites

Make sure that your system has the following available dependencies before installing each of the ingestion tools

Apache NiFi

- Requires Java 8 or Java 11
- Supported OS:
  - Linux, Unix, Windows, macOS
- Local installation:
  - Download the binaries or Sources from the Official Download Page
- Docker installation:
  - Download the images from the Official NiFi DockerHub

Apache Spark (PySPark)

- Python3.8 or above
- pip3
- Java 8 or later with JAVA_HOME section
- For ARM users, PyArrow is required for PySpark SQL, if PyArrow installation fails, try installing PyArrow >= 4.0.0

Node-RED

- Local installation: A supported version of Node.js.
- Docker installation: Docker Engine
- From source:
  - A supported version of Node.js
  - A git client

- The grunt-cli npm module installed locally.

Faust

- Python3.6 or above
- A running Kafka broker

Source code download links:

Apache NiFi:

https://www.apache.org/dyn/closer.lua?path=/nifi/1.18.0/nifi-1.18.0-source-release.zip

Apache Spark (PySpark):

https://spark.apache.org/docs/latest/api/python/getting_started/install.html#installing-from-source

Node-RED (build from GitHub using npm):

```
git clone https://github.com/node-red/node-red.git

cd node-red

npm install

npm run build

npm start
```

Faust:

Download the latest version from https://pypi.org/project/faust/

```
$ tar xvfz faust-0.0.0.tar.gz

$ cd faust-0.0.0

$ python setup.py build

# python setup.py install
```

Storage

These instructions have been executed on a RISC-V64 virtual machine with Debian OS running on Qemu with the following specifications:

```
root@debian-riscv64:~# uname -a

Linux debian-riscv64 5.16.0-5-riscv64 #1 SMP Debian 5.16.14-1 (2022-03-15) riscv

64 GNU/Linux

root@debian-riscv64:~# lscpu

Architecture:          riscv64

  Byte Order:          Little Endian

CPU(s):                4

  On-line CPU(s) list: 0-3

NUMA:

  NUMA node(s):        1

  NUMA node0 CPU(s):   0-3
```

Given the nature of RISC-V64, there is a notable lack of package release for the architecture. Thus, the need for compiling source-code is usually a must for complex applications. Compiling source-code is never a trivial task and providing a guide for it goes beyond the scope of this deliverable.

Instead, we will focus on databases that can be easily installed and should work as an out-of-the-box solution.

MySQL

MySQL is maybe the most popular open-source database and it comes with a release package for Debian on RISC-V architecture:

```
apt-get install mysql-server
```

Once installed you can control the server with systemctl.

Apache IoTDB

The IoTDB database runs on Java which does have release packages for RISC-V. First, we need to install some preliminary packages.

```
apt install default-jdk curl unzip
```

Download and unzip the binaries:

```
wget https://dlcdn.apache.org/iotdb/0.13.2/apache-iotdb-0.13.2-all-bin.zip

unzip apache-iotdb-0.13.2-all-bin.zip -d
```

We can start the database server with:

```
nohup sbin/start-server.sh >/dev/null 2>&1 &
```

To start a client, we can then run:

```
# Start the client

sbin/start-cli.sh -h 127.0.0.1 -p 6667 -u root -pw root
```

5.3 Low-End node (PULP RISC-V32)

As a result of WP3, Nuttx RTOS was ported to the PULP low-end systems to offer a Posix completable application environment. See D3.6.

While the system is limited, it still offers a nice abstract development environment.

5.3.1 Cloud communications

As discussed on D3.6 the Nuttx based PULP (RISC-V) node is not able support high end tools such as Python and Java. However, the system took an approach to utilize c/c++ tools. As an example, an Azure IotHub was integrated to the Nuttx. Low-end connectivity framework is presented on figure below.

Nodes connect to cloud, where they are orchestrated based on their identity (e.g. keyhash). After acceptance they become visible to the application.

The framework Low-end nodes were verified/demonstrated, by using ESP32-C3 embedded RISC-V devkit using the build in WIFI. The connectivity was based on Azure IotHub, however, the actual mechanism implemented is abstract. Other IoT-implementations are possible to utilize (Google, Amazon, etc.)

The main idea is that if the connectivity implementation below changes, the Fractal low-end Nuttx application does not need to change.

The code clip below is from the existing demo application.

Keys:
Here the security is based on keys, built in binary.

I “real” system, each individual device should have individual keys flashed at the production. Based on those individual production keys each of the units can be individually accepted/revoked from the cloud.

```
https://docs.microsoft.com/en-us/azure/iot-dps/concepts-symmetric-key-attestation?
tabs=linux#group-enrollments
// This is group key for fractal-demo (normally _NOT_ included on device binary)
static const uint8_t shared_group_key[] = {
       0xc6, 0xd3, 0x82, 0x3c, 0xf2, 0x3e, 0x99, 0x33, 0xfd, 0x27, 0xa6, 0xab, 0xc7,
        0x28, 0xe4, 0xd9, 0xd5, 0x2c,0xc7, 0x6c, 0xb3, 0xd2, 0xa9, 0x25, 0x81, 0xcc,
        0x3a, 0x1c, 0x32, 0x34, 0x7a, 0x24, 0xd7, 0xb8, 0x7d, 0xa2, 0xe5, 0x8f,0x0f,
        0xc8, 0x20, 0xb0, 0x6f, 0x6e, 0x9d, 0x9b, 0xb4, 0x96, 0x4e, 0x0c,0xec, 0xc2,
        0xe9, 0x30, 0x07, 0x29, 0xb6, 0xbf, 0xa2, 0xf6, 0x2c, 0x25, 0x03, 0x97
};
```

The main application below is the task that performs the main functionality of demo-case. Function ***do_provisioning()*** connects to cloud and if succeed the system sends/receives data by ***do_send_datatest()*** function. Note actual manipulation of LED and reading sensors is not presented here, but on RTOS systems that is quite trivial.

```
int main(int argc, char **argv)
{
    do_exit = false;
    (void) IoTHub_Init();

    led_init();
    led_set_value(1);

    THREAD_HANDLE *handle;
    ThreadAPI_Create(&handle, run_ui, NULL);

    ui_set_registration_state("none");
    ui_set_registration_reason("");
    ui_set_connection_state("none");
    ui_set_connection_reason("");
    ui_set_temperature("??.?");
    ui_set_humidity("rh: ??.?");
    ui_set_led_state(led_get_state());

    signal(SIGINT, intHandler);

    while (do_exit == false) {
```

```
        do_provisioning(device_id);


        if (strlen(g_iothub_connection_string) != 0) {
            do_send_datatest(g_iothub_connection_string, device_id);
        } else {
            printf("Can't send data: no connection string available.\n");
        }


        sleep(5);
    }


    printf("Exitting...\n");


    // Free all the sdk subsystem
    IoTHub_Deinit();


    return 0;
}
```

Behind ***do_provisioning()*** and ***do_send_datatest()*** are the IotHub implementations. For details see the code above.

5.3.2 Task scheduling

Nuttx is a posix compatible OS, and it supports posix tasks. Example code above is one task. There can be multiple parallel tasks in the system and each task may have multiple threads. Communication between tasks typically/threads happen by sockets or by variables (protected by mutexes).

Limited memory and processing requirements must be considered when defining the system.

Tasks can be started automatically I.e. by **/etc/int.d/** or manually from terminal command line.

5.3.3 Ingestion & Storage capabilities

Nuttx supports normal posix sockets, where data flow control can be handled e.g. by signals. With adequate driver various communication mediums may be added. Thus, the socket is a typical way connect communication interfaces. Such as:

- Ethernet
- Wifi
- 1.5G to 6G
- Serial, can, RS485
- USB

Nuttx supports normal posix file system. With adequate driver various storage mediums may be added. Such as:

- Read only file system allocated on processor Flash
- Read/write flash file system on system board
- MMC card interface with removable media
- SATA interface for hard disks
- USB
- Network storage utilizing e.g., NFS

Nuttx systems typically offer a console interface. Normally that is used for testing and development purposes. Typically, it is via serial port and it offers (linux style) tools to control and monitor applications and daemons on the system.

6 Hardware-level Edge Controller

The hardware edge controller controls the underlying hardware, such as communications and computations. The underlying hardware is based on a network-on-chip (NoC) multicore architecture that supports heterogeneous cores connected via NoC. The description of the underlying NoC-based multicore architecture is reported in contribution D4.4 (WP4). However, in this deliverable D6.2, we report on the HW gateway architecture that controls communication between multiple nodes. The HW gateway controller follows a pre-calculated schedule computed during design time and configures the network HW gateway when systems begin to manage network gateway communications, as described in Section 6.1.1 below.

6.1 Overview of NoC based Multicore Architecture

The NoC-based multicore architecture is the hardware used for computation and communication in the Fractal Node. As shown in Figure 5, the multicore architecture consists of a core responsible for computations and NI for accessing the NoC. The on-chip interconnects a network of routers that connect the different cores within the chip. We can also see the NGW, the network gateway for accessing the off-chip domain. The NoC-based multicore follows a time-triggered schedule to control the communication and computation of tasks in the multiple cores within the node. Thus, each message communicated within the cores is injected at a predefined time to avoid message collision.

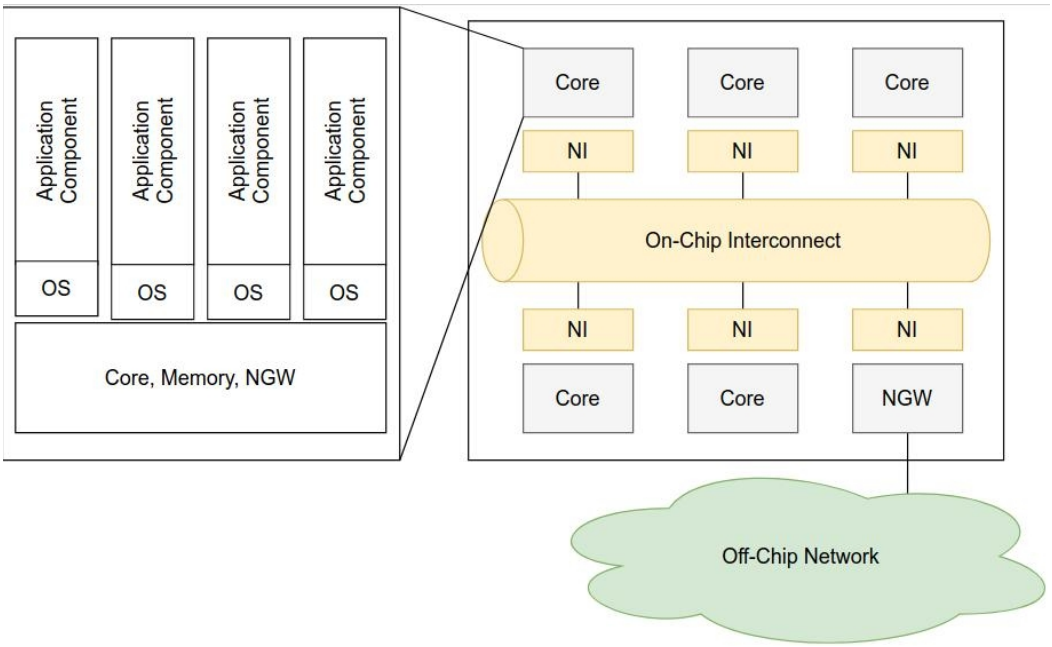


Figure 5: Multiple core architecture

6.1.1 Network Hardware Gateway Architecture

On-chip/off-chip gateways establish end-to-end communication in heterogeneous and mixed-critical networks, as shown in Figure 6. An off-chip/on-chip gatewayc ontrols the redirection of messages between the NoC and the off-chip communication network.

The following sections explained the different services of the gateway architecture.

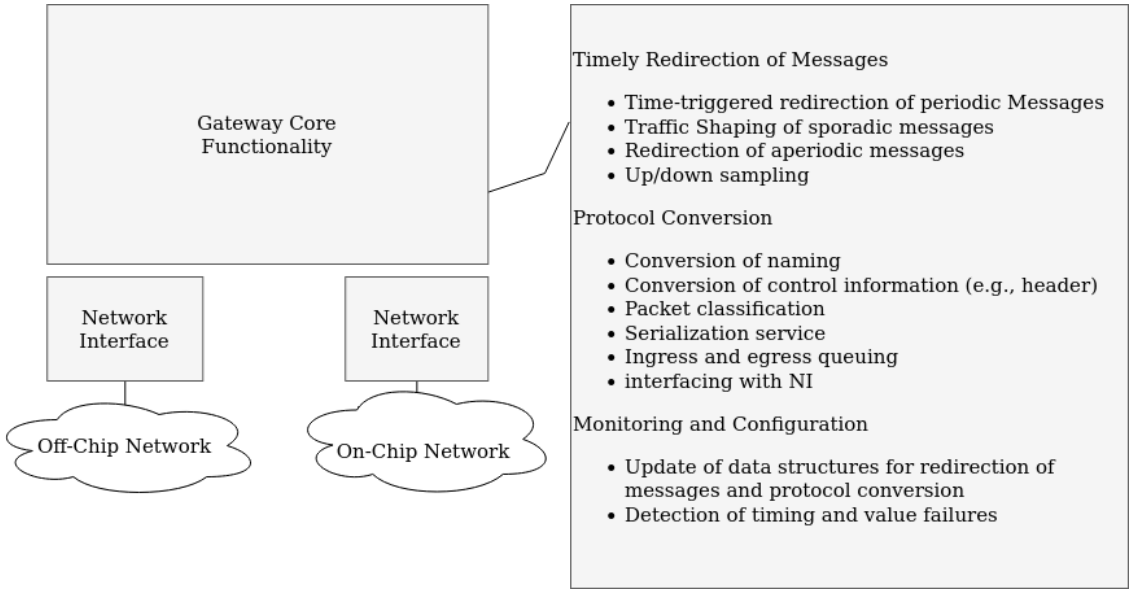


Figure 6: Off-chip/On-chip Network Gateway Services

Message-Classification Service

Message classification is based on the concept of Virtual Link (VL). The concept of VL is used to realize bandwidth partitioning, which is an end-to-end multicast channel between a sender component and multiple receiver components. The message classification service is responsible for classifying incoming messages from NI to decide on the appropriate buffer (i.e., VL queues and egress queues) based on the message type and configuration parameters. In addition, the message classification service checks the format of the message and its control information, such as the VL identifier (VLID). If the message has an invalid format, it is discarded. In addition, the message classification services use the configuration parameters to check the integrity and validity of the periodic and sporadic messages; this includes checking the message size and whether the messages arrive with the correct VLID. In addition, the gateway verifies that the periodic messages arrive within the specified receive windows of the VL.

Message-Scheduling Service

This service guarantees the determinism of the redirection of periodic messages within the on-chip/off-chip gateway. Each periodic message has predefined timing parameters, such as a period and a phase. According to the predefined configuration for message scheduling, this service determines the times the periodic messages are forwarded.

Traffic-Shaping Service

This service guarantees the minimum interarrival time between two consecutive sporadic messages on the respective VL. The minimum interarrival time is part of the configuration parameters for each VL.

Relaying of Aperiodic Message

This service is responsible for forwarding aperiodic messages between ingress and egress queues based on their respective data directions and destination addresses.

Down Sampling

This service allows message exchange between networks with different periods or rate constraints for sporadic messages. Down sampling is also required to compensate for differences in bandwidths between off-chip and on-chip networks. As a result, the gateway must redirect a subset of incoming messages to meet the timing requirements of the destination network. In addition, the redirection needs to be synchronized between networks to ensure consistent data is forwarded. In the down sampling service, the gateway sends the most recent periodic message that arrives before the next send time. The traffic shaper discards all messages that arrive within the minimum arrival time for sporadic messages.

Protocol Conversion

The protocol conversion service encapsulates and decapsulates incoming and outgoing messages. The gateway adapts the message format and controls information according to the communication protocol (e.g., headers with addresses, flow control information, CRC). In addition, gateways must determine a new address for the destination network for each incoming message. This computation is performed based on the address information of the incoming message and differs depending on the traffic and network type. For periodic and sporadic traffic, the new addressing information is either a VLID, a routing path to the final destination, or another gateway. The routing path is required for source-based routing, which is common in many NoCs. The VLID or routing path can be determined by looking up the incoming address information in the gateway configuration. For aperiodic traffic, the new addressing information is either destination addresses or a dynamically computed routing path. The gateway can dynamically use the spanning tree protocol to determine the destination address [2].

Egress-queuing Service

The egress queues consist of a periodic egress queue, several sporadic queues, and an aperiodic egress queue. Each sporadic queue has its priority level. The deterministic behavior of periodic messages is guaranteed by the message scheduling service (see Message scheduling service in the previous subsection) in combination with a higher priority than sporadic messages. The deterministic behavior guarantees that no conflict occurs in the egress queue. Therefore, a queue that must provide buffer capacity for a single periodic message of maximum size is sufficient. To control the resolution of conflicts between sporadic messages, we distinguish multiple queues according to their priorities. These queues are used to

multiplex the frame flow coming from the internal message queues. The queues provide guaranteed buffer capacities, which can also be realized by dynamic memory allocation. The guaranteed buffer capacities prevent the loss of messages due to the bounded accumulation of sporadic messages, which is determined by the rate constraints.

Ingress Queuing Service

The ingress queue consists of a FIFO queue for each network. The incoming messages from the network are queued in the ingress queue; then, the ingress queuing service notifies the message classification service.

Virtual-Link Queuing Service

VL queues belong to two groups : one for the periodic messages and the other one for the sporadic messages.

- Periodic VL buffers: Each periodic VL has one periodic VL buffer, which provides buffer space for exactly one message. In case this buffer is full and another message arrives with the same VLID, the newer message replaces the old one.
- Sporadic VL queues: Each sporadic VL has one queue. It is possible to store several messages of the respective VL in this queue.

Serialization Service

The serialization service forwards the messages from the egress queues to the network (off-chip or on-chip) according to the priority. The highest priority is assigned to periodic messages, whereas aperiodic messages have the lowest priority. Also, the serialization service uses either shuffling or timely blocking to resolve contention between different traffic types. The timely block mechanism disables the sending of other messages in the egress queues during a guarding window prior to the transmission of a periodic message. For the shuffling mechanism, no guarding window is needed. In the worst-case, the gateway delays a periodic message for the duration of a sporadic or aperiodic message of maximum size.

Configuration Parameters

The configuration parameters of the gateway are as follows:

- Guaranteed buffer capacity: Each ingress queue, egress queue and VL queue is associated with a corresponding guaranteed minimum buffer capacity. The buffer capacity is determined by the maximum message size and the message timing. This buffer capacity can avoid message omissions of sporadic and periodic messages based on rate-constraints and message periods

- Address information of ports: The VL associated with a port and the data direction (from the off-chip network or to the off-chip network) are defined.
- Message type: The message type is defined such as periodic, sporadic or aperiodic.
- Timing parameters: In case of periodic messages, the parameters include the period and phase. For sporadic messages, the interarrival time, the jitter and the priority are specified. In case of aperiodic messages, no timing parameters are required.

6.1.2 Processing of Different Traffic Types

This section describes the processing of messages in the off-chip/on-chip gateway. As depicted in Figure 7, the network Gateway architecture consists of different components such as bridge, serialization, ingress, egress, and VL queue layers.

The message bridge handles incoming messages through timely redirection, protocol conversion, monitoring, and configuration services. The network interface provides the interface between the network and message bridging. It also performs message classification and serialization. Each network interface connects the gateway either to an off-chip network (TSN) or an on-chip network (ATTNoC).

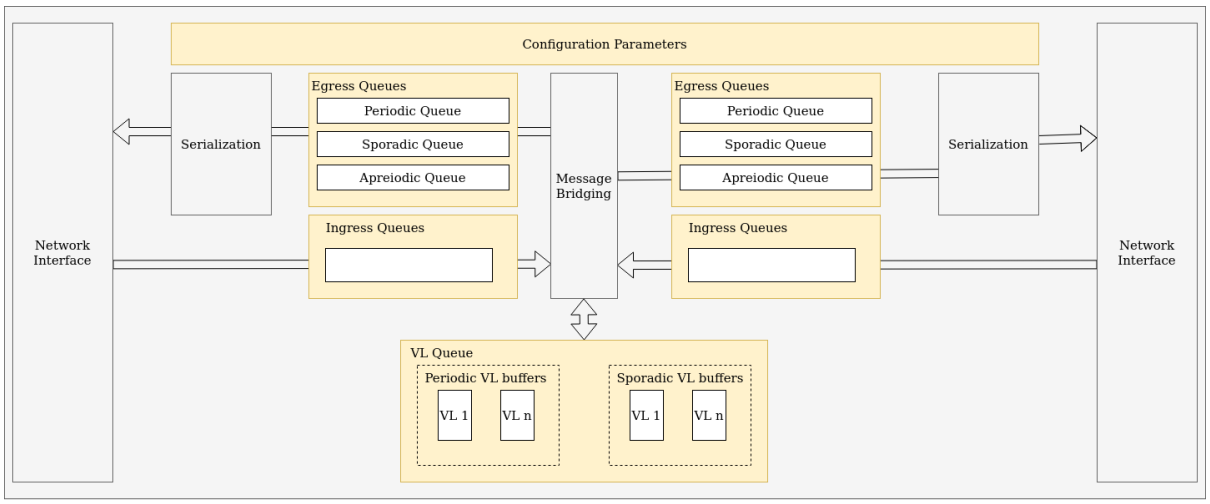


Figure 7: Off-chip/On-chip Network Gateway Architecture [1]

The ingress layer is invoked by an incoming message from the on-chip or off-chip network. The incoming messages are relayed to the bridge layer in the ingress layer. The bridge layer classifies the incoming messages based on the type (i.e., periodic, sporadic, and aperiodic). Below we explain the processing for each message type.

Processing of Periodic Messages

Figure 8 depicts the flowchart for periodic message transmissions. In the classification state, a message analysis function extracts from the periodic message of the VLID. In case the incoming message does not have a defined VLID in the configuration parameters, the message is considered invalid. Invalid messages are dropped in the classification state, while valid messages result in a transition to the VL buffer state. Based on VLID, the check VL buffer status function retrieves the buffer identifier from the configuration parameters. Then, it puts the message into the VL buffer, which provides buffer space for exactly one message. In case this buffer is full, and another message arrives with the same VLID, the newer message replaces the old one. The “VL buffer status” for the corresponding VLID is updated when the message is buffered. Suppose the “VL buffer status” denotes that the buffer is not empty. In that case, the next transmission time function in the time triggered scheduling state determines the point when the periodic message is relayed according to the communication schedule, thereby ensuring deterministic communication behavior. At the next transmission time, the pass information to the redirection function sends the information (i.e., VLID, buffer identifier, and direction) to the redirection state. In the redirection state, the check VL buffer status function checks whether the corresponding VL buffer contains a message. This message is then sent to one of the egress objects according to the direction parameter, where the message is enqueued in a periodic egress queue. When the message is removed, the “VL buffer status” for the corresponding VLID is updated. These procedures are performed according to the communication schedule until the “VL buffer status” indicates that the buffer is empty. The serialization is responsible for forwarding the message from the egress queues to the on-chip or off-chip network interface according to the priority. The highest priority is periodic messages, whereas aperiodic messages have the lowest priority. Using these priorities, the serialization supports two mechanisms to resolve collisions between the different types of messages:

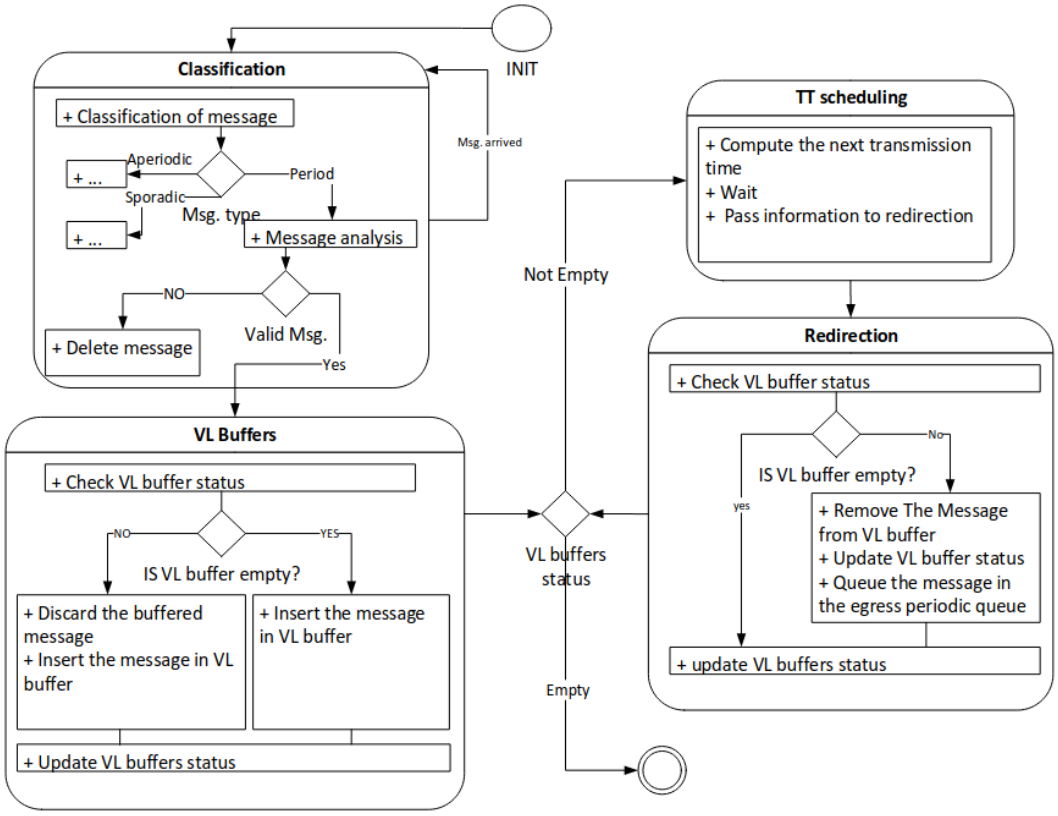


Figure 8: Flowchart for Periodic Messages.

- Timely block: According to the time-triggered schedule, the serialization knows in advance the transmission times of the periodic messages. Timely block means that the gateway reserves so-called guarding windows before every transmission time of a periodic message. The behavior of the timely block mechanism is illustrated in Figure 8. The egress queues have four egress queues with decreasing priorities: one queue for periodic messages, two queues for sporadic messages (each one for a different priority class) and one queue for aperiodic messages. The egress-queue status is updated when a message is enqueued in one of the egress queues. In case the status of the egress queue is “not empty”, the timely block checker function in the timely block state verifies that no guarding window is active. In case of guarding windows, the wait function imposes a delay until the next trans- mission time of the periodic message. If there is any periodic message, this message is sent. Otherwise, the process of the flowchart returns to the egress queue state. In case there are no guarding windows, the select message function in the send state selects one message out of the sporadic and aperiodic queues based on the priority and this message is sent. If the status of the egress queues is still “not empty”, the procedure is
repeated until the egress queues are empty.

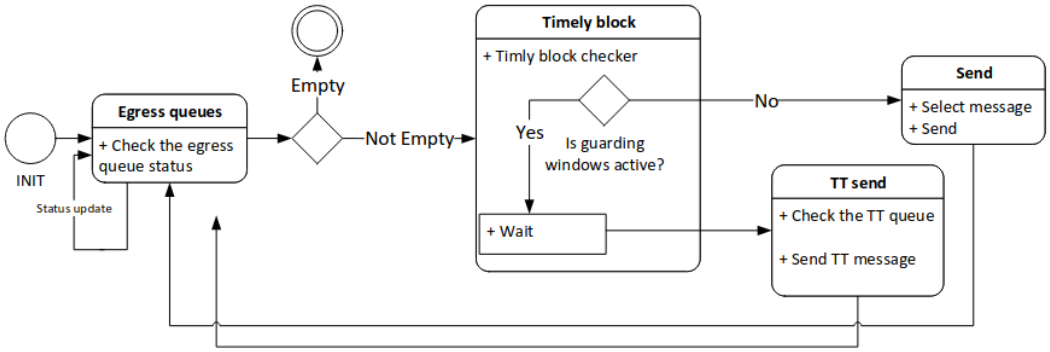


Figure 9: Flowchart for Timely Block Mechanism

- Shuffling: If a low priority message is being transmitted while a high priority message arrives, the high priority message will wait until the low priority message is finished. Figure 10 shows the flowchart for the shuffling mechanism within the serialization object. The egress queues status is updated when a message is enqueued in one of the egress queues. In case the status of the egress queue is “not empty”, the select message function removes one message from the egress queues based on the priority. The send function forwards the message to the network interface of the on-chip or off-chip network interface. If the status of the egress queue is still “not empty”, the procedure is repeated until the egress queues are empty.

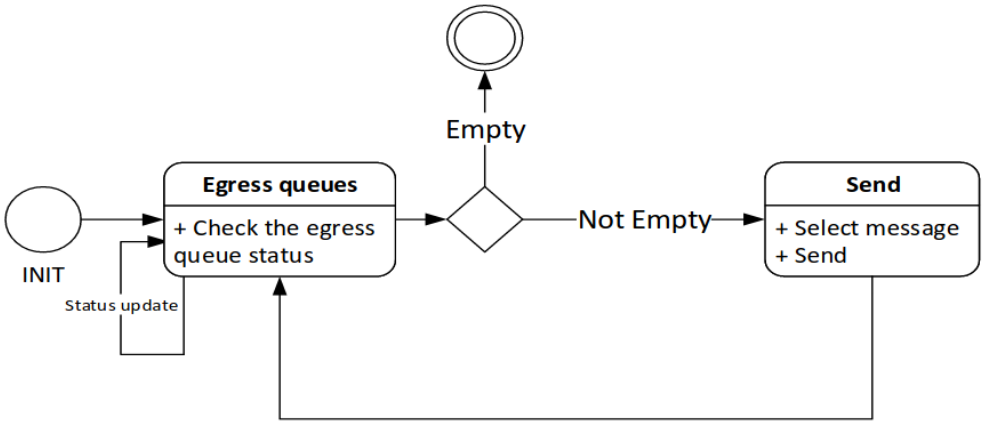


Figure 10: Flowchart for Shuffling Mechanism.

Processing of Sporadic Messages

Figure 11 depicts the flowchart for sporadic message transmissions. The message analysis function in the classification state checks incoming messages. The size of the message must be below the maximum message size according

to the configuration parameters of the VL. A valid message is enqueued in the corresponding VL queue. When the message is enqueued, the “VL queue status” for the corresponding VLID is updated. In case the VL queue was empty, the update time function updates the reception time of the last incoming VL message. This timestamp is essential for traffic shaping and temporal partitioning. In the sporadic shaper, the sporadic traffic regulator and controller function guarantees the minimum interarrival time between two consecutive instances of a sporadic message on the respective VL. The sporadic traffic regulator and controller function compute the necessary waiting time for each message based on the time of the latest incoming VL message. When the waiting time has expired, the redirection function passes the information (i.e., VLID, buffer identifier and direction) to the redirection state. In the redirection state, the remove message from VL queue function forwards the message from the VL queue to one of the sporadic egress queues according to the direction and priority parameters. In case the VL queue has another message, the time of the last incoming VL message is updated. This step allows the sporadic traffic regulator and controller function to send the next message after the minimum interarrival time. This procedure is repeated until the “VL queue status” is “empty”. Thereafter, the serialization is responsible for forwarding the message to the network interface of the off-chip or on-chip network as explained in the previous subsection.

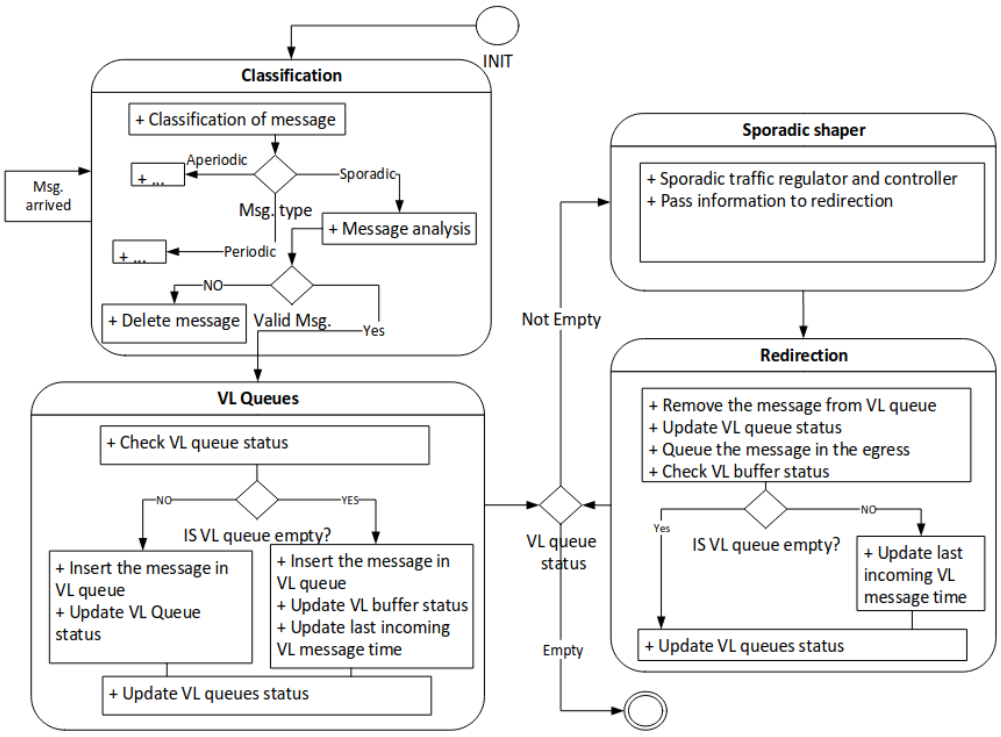


Figure 11: Flowchart for Sporadic Message.

Processing of Aperiodic Messages

Aperiodic messages have no timing constraints on successive message instances and no real-time guarantees. Therefore, the incoming messages are inserted into the corresponding aperiodic egress queue. The “egress queue status” is updated when the message is enqueued. After that, the serialization is responsible for forwarding the message to the network interface of the off-chip or on-chip network.

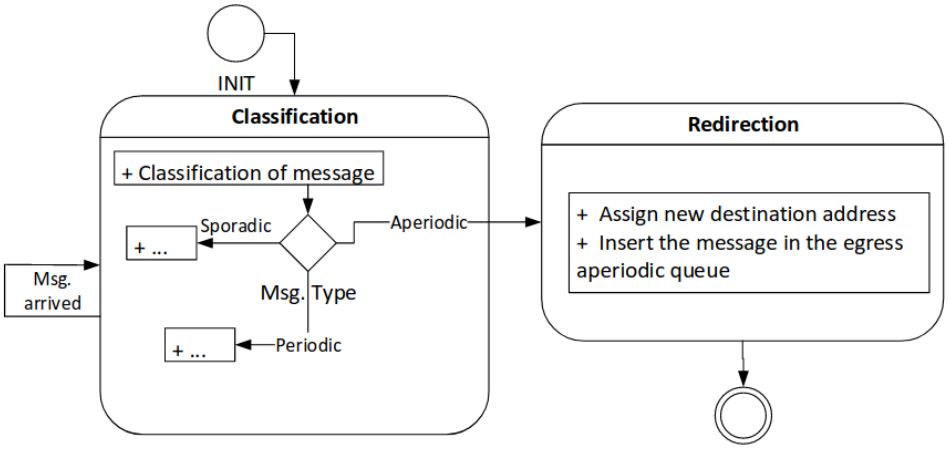


Figure 12: Flowchart for Aperiodic Message.

6.1.3 Scheduling Problem in HW Network Gateway

An Optimised Reliable Task and Message Scheduling algorithm (OR-TMS) uses DPSO to find scheduling solutions given real-time objectives such as minimal makespan, total energy consumption, and failure rates of scheduling all tasks [3]. A task scheduling instance refers to mapping tasks to potential hosts, creating a distribution of tasks for each instance. The energy consumption of routing a message from source to destination is the sum of energy dissipated by routers in the message route while routing the message. On the other hand, message arrival time denotes the instant when all dependent messages arrive at a child task from a parent task; This also defines a precedence constraint between dependent tasks. For each task scheduling instance, the cost of executing all tasks is directly proportional to the schedule makespan, consumed energy, and failure rate. Therefore, a lower scheduling cost indicates better schedule instances. The objective of OR-TMS is to minimize the schedule execution cost, which depends on the DPSO mapping.

DPSO Task and Message Scheduling

DPSO initially establishes a population of TMS instances where each instance is evaluated through its cost value. This results in a multi-objective optimization problem employing a weighted sum of the system’s completion time (schedule makespan), total energy consumption, and failure rates. Next, DPSO minimizes the cost value of TMS instances, achieving optimization through updating the position of

particles while avoiding local convergence. Finally, OR-TMS integrates a Time Slot Message Scheduler (TSMS) into DPSO. TSMS is used to schedule the message traffic using conflict-free time slots to avoid signal interference. The pseudo-code for the DPSO-based task and message scheduling algorithm is shown in Algorithm 1. The particle's position Xi represents a solution for scheduling tasks to random hosts. A better particle position results in a lower cost value. Algorithm 1 is terminated when changes in particles' positions do not further result in better cost values.

Algorithm 1 DPSO-based Task and Message Scheduling Procedure

Input: Tasks, Particles
Output: G_{best}

1: **for** each particle p_i **do**
2: Initialize position X_i randomly
3: Initialize velocity V_i randomly
4: Initialize P_{best_i} with a copy of X_i
5: **end for**
6: **for** each particle p_i **do**
7: Evaluate $Cost(P_{best_i})$
8: Initialize G_{best} with a copy of P_{best_i} with the least cost
9: **end for**
10: **while** the convergence condition is not met **do**
11: **for** each particle p_i **do**
12: **Call Algorithm 2 OR-TMS(T)**
13: Evaluate $Cost(X_i)$ using Eq. 8.17
14: **if** $Cost(X_i) < Cost(P_{best_i})$ **then**
15: $P_{best_i} \leftarrow X_i$
16: **end if**
17: **if** $Cost(X_i) < Cost(G_{best})$ **then**
18: $G_{best} \leftarrow X_i$
19: **end if**
20: **end for**
21: **if** the convergence condition is met **then**
22: Output G_{best}; Break
23: **else**
24: **for** each particle p_i **do**
25: Update V_i according to Eq. 8.18
26: Update X_i according to Eq. 8.19
27: **end for**
28: **end if**
29: **end while**

DPSO-based task and message scheduling procedure.

Figure 13: DPSO-based Task and Message Scheduling Procedure

OR-TMS initially computes each unscheduled task t its top-level cost tlt. Then, all unscheduled tasks are sorted by the weight of their tlt. Algorithm 2 applies Algorithm 3 to all sorted tasks until scheduling all tasks of a particle's position. Algorithm 4 calculates routes and schedules messages on conflict-free time slots. Finally, algorithm 5 inherits the physical interference model of the system to find feasible time slots used to transmit every message.

Algorithm 2 OR-TMS(T)

Input: G_{App}, G_{Arc}
Output: F_T, en_T and $makespan$

```
Sort(tasks T)
for t ∈ T_sorted do
    Call Algorithm 3: Task t Scheduling
    F_T += F_t
    en_T += en_t
    makespan = max(arrival)
end for
return F_T, en_T and makespan
```

Pseudo-code of Algorithm 2 used to schedule all tasks to their corresponding hosts, outputs are used to evaluate the current DPSO instance.

Figure 14: OR-TMS(T)

Algorithm 3 Task t Scheduling

Input: unscheduled task t
Output: $arrival$, en_t and F_t

```
for m ∈ M_t do
    Find the redundant routes (RR(m)_h)
    for rt(m) ∈ RR(m)_h do
        Call Algorithm 4 Message Scheduler
        m_arrival ← m_IT + m_e2eD
        Update arrival = max(m_arrival)
        if (arrival + t_et) > t_dl then
            Invalid solution, break.
        end if
        Update en_m
        Update F_m
    end for
end for
Find(t_rt)
if (t_rt + t_et) > t_dl then
    Invalid solution, break.
end if
Compute en_t, F_t using Eqs. (8.15, 8.16).
return arrival, en_t and F_t
```

Pseudo-code of Algorithm 3 used to schedule a task to its corresponding host.

Figure 15: Task Scheduling

Algorithm 5 Message Interference Analysis

Input: path l_i of fr_m, fr_m, n (current slot number)
Output: n^* (next slot number)
Initialize : find_slot = false, $r = l_i.receiver$, {*r is the router to which the fragment* fr_m *is sent.*}

1: **while** find_slot == false **do**
2: **if** $Slt(n)$ == null **then**
3: Insert l_i on $Slt(n)$
4: find_slot = true
5: en_r += $en_{r.m}$, et_r += $et_{r.m}$
6: $n^* = n$
7: **return** n^*
8: **else**
9: **for** each path $l_j \in Slt(n)$ **do**
10: Compute $I_{\text{slt}}(Slt\{l_j\}, l_i)$ {*the interference on* l_i *by all paths* l_j*'s in time slot number n.*}
11: I += I_{slt}
12: Update SINR value
13: **if** $l_j.receiver == r$ **then**
14: Sum_en_r += $en_{r.m^*}$, Sum_et_r += $et_{r.m^*}$, where $m \neq m^*$
15: **end if**
16: **end for**
17: **if** SINR $\geqslant \beta$ **then**
18: Insert l_i on $Slt(n)$
19: find_slot = true
20: en_r += $Sum\_en_r + en_{r.m}$, et_r += $Sum\_et_r + et_{r.m}$
21: $n^* = n$
22: **return** n^*
23: **else**
24: Increment n
25: **Continue**
26: **end if**
27: **end if**
28: **end while**

Pseudo-code of Algorithm 5 used to check the feasibility and determine the number of time slots of each fragment fr_m.

Figure 16: Message Interference Analysis

Simulation and Evaluation of OR-TMS

Various system application models are generated as random forest fire-directed graphs. The dependency and constraints between tasks in the application models vary across the models. Figures 17 and 18 illustrate the platform topologies for which OR-TMS generates time-triggered schedules for exchanging messages within the system. Every access point creates a local network with four static wireless TSN-enabled hosts with a 50 Mbps data rate. Any node out of energy is assumed a failed node where time and energy are measured in milliseconds and joules, respectively.

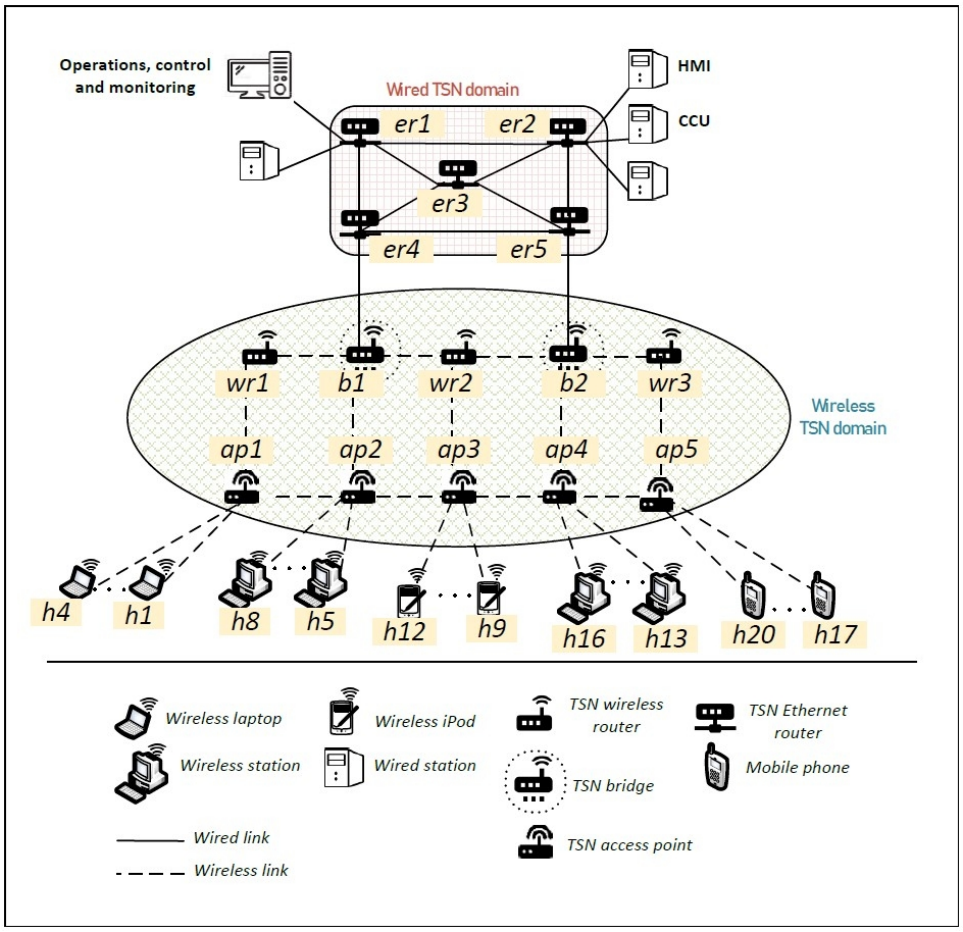


Figure 17: System Model (Mesh Topology)

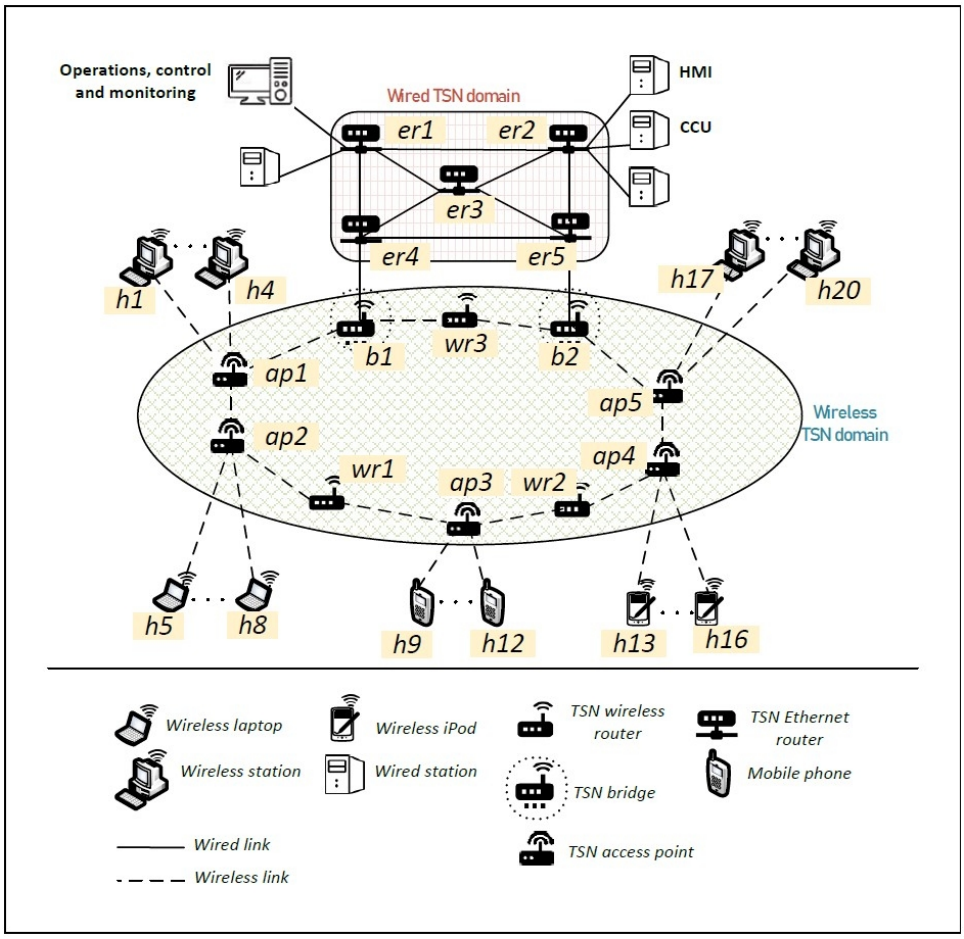
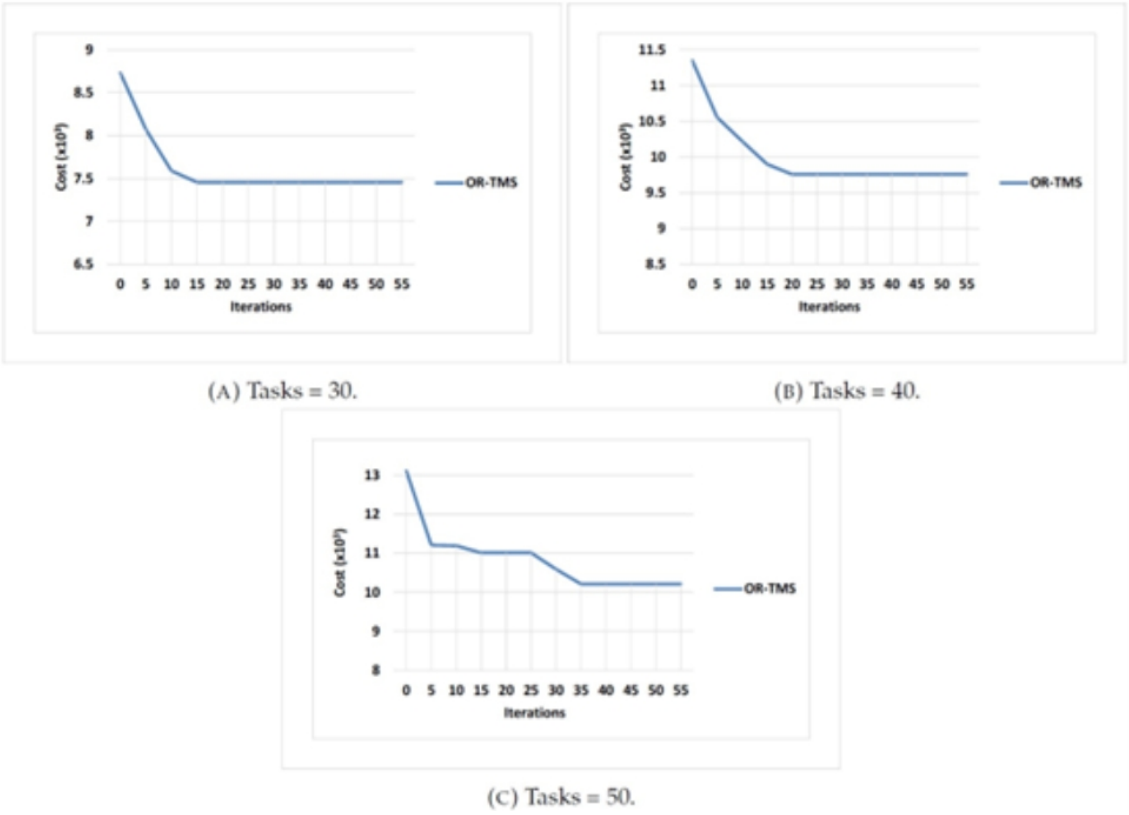


Figure 18: System Model (Ring Topology)

The convergence rate of OR-TMS refers to the number of iterations for which OR-TMS converges, and an optimized scheduling solution is found. In evaluating the OR-TMS convergence rate, application models of 30, 40, and 50 tasks and messages with task deadlines of 2000 mS are used where the experimental step size is set to 5. Figure 19 shows a population of 10 particles converges after 15 iterations for an application model of 30 tasks and messages. For the application models of 40 and 50 tasks and messages, OR-TMS converges after 20 and 35 iterations, respectively. This represents a direct proportionality between the application's number of tasks and messages and the convergence rate.



(A) Tasks = 30.

(B) Tasks = 40.

(C) Tasks = 50.

Figure 19: OR-TMS Convergence Rate

7 Run-time manager: Edge Controller communications

7.1 Description

The Runtime Manager is a component developed to coordinate and manage task scheduling and load balancing operations between modules in one or more fractal nodes at runtime. RM performs the scheduling of various operations and it is possible to configure every aspect of the tasks that need to be performed, for example what data needs to be exchanged and when, how and to which module it has to be sent. In addition, it provides load balancing capabilities using the interface with the Load Balancer component, sending the task execution to a different node.

RM was initially needed in VAL-UC6 to manage task scheduling and interaction between the several application modules of the intelligent totem nodes of the use case. Further details about the implementation in the UC context are reported in *D8.1 - Specification of Industrial validation Use Cases* and *D8.2 - System Requirements*.

Considering the aforementioned RM features, we decided to develop the component to be as much general purpose as possible in order to be used in broader contexts, so the outcome of the job carried out is the development of a component capable of being deployed/used in different scenarios.

7.2 Design and Implementation of component

7.2.1 Design

The purpose of the Runtime Manager is to enable communication and data dispatch among the various components installed on the node, and to manage the load balancing operations, when needed, by assigning the execution of the activities to a different instance of the Runtime Manager module installed on another node.

The module is completely configurable — in terms of data flow to manage and in terms of components to communicate with — by means of a set of configuration files which will be detailed in section 7.2.2.

This process is represented in the following flow-chart diagram:

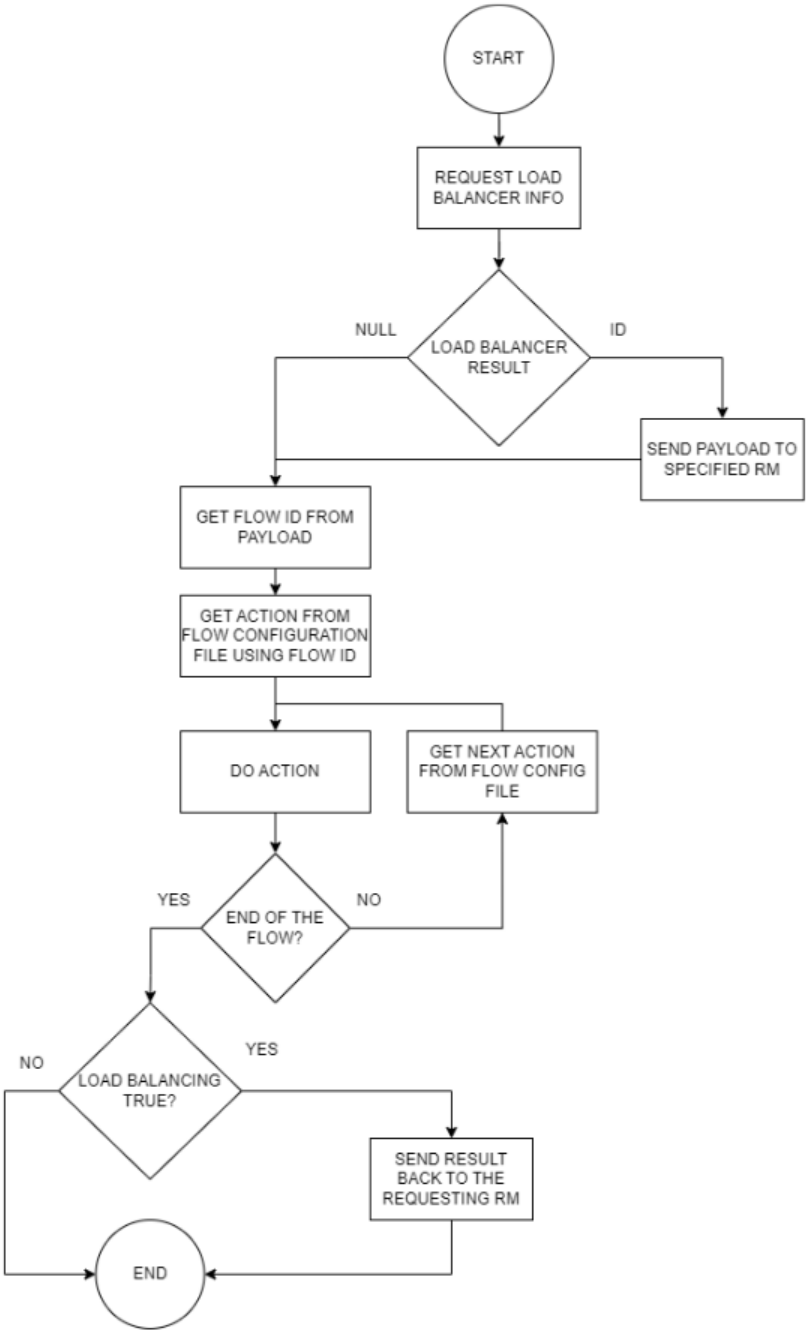


Figure 20: Runtime Manager Flow Chart Diagram

During the design phase, we also analyzed a typical interaction between the Runtime Manger and several applications as an example flow from the dedicated configuration file. The following sequence diagram depicts a general data flow the Runtime Manager might have to handle:

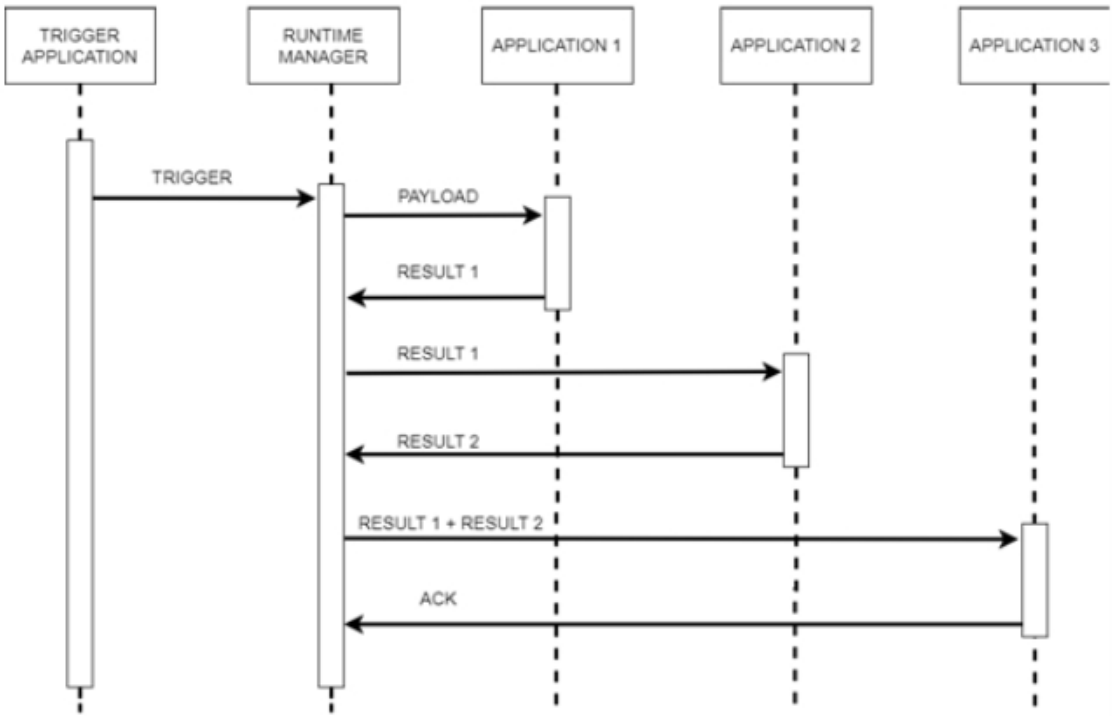


Figure 21: Runtime Manager Sequence Diagram

The diagram shows a component triggering the Runtime Manager by publishing a message on the configured MQTT topic. The trigger message contains information about the flow that needs to be executed and the payload to be exchanged.

In the particular case depicted in Figure 21, RM sends the payload to *Application 1*, whose result is subsequently sent to *Application 2*. This, in turn, will return a result which, together with that from *Application 1*, will form the payload for *Application 3*, the last part of the data flow, which acknowledges the end of its computations.

7.2.2 Implementation

The Runtime Manager software module has been implemented in Python following the OOP paradigm. It is a configurable module based on four configuration files “load_balancer”, “nodes”, “components'' and “flows”; each of them serves a specific purpose, as suggested by the names: “load_balancer” collects the values of protocol, ip, port and endpoint needed to contact the Load Balancer; “nodes” and

“components”, respectively, contain the information regarding the nodes and the components in the system which the RM may need to contact; finally, “flows” is the configuration file containing the instructions for the RM to execute the multiple processes scheduled by the system. Each configuration file is in JSON format, except for “flows.conf” which follows the rules explained inside the README in the project repository, in order to unambiguously define the data flows.

There are two ways to trigger the functionalities of the Runtime Manager: an MQTT interface and a REST interface.

The messages received either via MQTT or REST will be forwarded to the Action Dispatcher module, which instantiates the appropriate object according to whether the action is to be performed on the current Runtime Manager (*Home Execution*), or sent to a Runtime Manager on a different node (*LB Execution*).

MQTT subscriber service

On launching the MQTT interface script, the RM subscribes to the topic configured and will be ready to receive any message published on it.

A JSON string published on the configured topic is the standard entry point to the Runtime Manager, and the method to be used in order to trigger the RM. The JSON string format is as follows:

```
{
        "id_flow": "1",
        "payload": "0x03abcdefghil"
}
```

where *id_flow* is the ID of the flow to be read in the configuration file and executed, and *payload* is the data which might be required to be passed to some of the components involved in the execution.

When triggered via MQTT, the Runtime Manager will always call the Load Balancer service to know whether it must hand over the execution to a different node (in which case the Load Balancer will return the ID of the node to contact) or whether it can perform a “home execution” (in which case the Load Balancer will return *null*). In the former case, the file “nodes.config” will provide all the necessary information regarding the node to call via REST request.

REST API service

On launching the REST interface script, the RM exposes a POST API identified by the configured endpoint.

The most straightforward process flow happens when the RM is triggered by a POST request to the REST service exposed, which is dedicated to load-balancing functionalities. The JSON object in the body of the POST request will have the following format:


```
{
   "id_flow": "1",
   "is_load_balancing": true,
   "payload": "0x03abcdefghil"
}
```


where *id_flow* and *payload* are as in the previous paragraph, and *is_load_balancing* indicates whether we are in load balancing.

In this case, the RM shall avoid the call to the Load Balancer service, and directly instantiate the *Home_Executioner* which will read the requested flow on the “flows.config” file and then execute it.

8 Conclusions

In this deliverable we gather all the information about the work that has been done during the course of T6.2, dedicated to Edge Orchestration and its components.

This deliverable provides details about all the Fractal components developed related to Edge Orchestration services.

Sections 4 and 5 are dedicated to Edge Orchestration services at the application level, and are dedicated to the design of these components, being Section 4 focused on the design of their architecture and Section 5 on their implementation.

Section 6 is focused on Edge Orchestration services at the Hardware level, and Section 7 is based on the Network Orchestration Fractal component (the Run-time manager).

With the addition of these components into the Fractal ecosystem, the Fractal Edge platform is provided with ingestion and storage capabilities for all the Fractal Edge platforms (ARM64, RISCV64 and RISCV32 architectures). This means that any Use Case being developed with the Fractal components will have at its disposal a wide collection of tools and strategies that allows them to process data streams with potentially any data format or source.

These ingestion tools have been chosen considering the most popular languages (Java-based, Python-based and JavaScript-based), so that the platform administrators will be able to choose based on their preferences.

With respect to the Data storage capabilities, both relational and non-relational databases have been included for the High-End and Mid-End nodes, providing in each case different alternatives to install and manage the data sinks, from Docker containers (when available), package manager installation steps, and build-from-source installations.

The Container Orchestrator (K8S, Docker Swarm...) has been a hot topic of research during the course of the project (WP5 T5.1 and T5.4, and WP6 T6.1). In this deliverable the design and implementation of independent orchestration components are detailed, providing novel orchestration strategies that support the already existing Container Orchestrators, and giving the platform the possibility to implement orchestration strategies even in orchestrator-less deployments. These components have been designed on a micro-services and containerized approach, which ensure high-availability and resilience of the deployments.

This deliverable also show the results of T6.2, providing Edge Orchestration capabilities to all the three reference platforms (ARM64, RISCV64 and RISCV32), at all layers from the node layer (HW-Level Edge Controller), the application layer

(Edge Orchestrator and Custom Orchestrator) and process and runtime operations (Runtime Manager).

9 Bibliography

Fractal Project related work

D4.4 FRACTAL SAFETY SERVICES

D5.5 Specification of AI methods for FRACTAL system control

D6.1 FRACTAL processing node design and implementation

Related FRACTAL Components

WP6T62-01 - Data Ingestion

WP6T62-02 - Federated Data Collection

WP6T62-03 - Run time Manager

WP6T62-04 - Hardware Edge Controller

WP6T62-06 – Orchestration (Edge controller)

WP6T62-06 - Orchestration (Mid-range node orchestrator)

WP6T62-06 – Orchestration (Low-end node orchestrator)

References

[1] 2015 IEEE 18th International Conference on Computational Science and Engineering. Mohammed Abuteir, Roman Obermaisser.

[2] IEEE standard for local and metropolitan area networks: Media Access Control (MAC) bridges. IEEE Std 802.1D-2004 (Revision of IEEE Std 802.1D-1998), pages 1 – 277, 9 2004.

[3] J. Kennedy and R. Eberhart, “Particle swarm optimization,” in Proceedings of ICNN’95-International Conference on Neural Networks, IEEE, vol. 4, 1995, pp. 1942–1948.

10 List of figures

11 List of tables

12 List of Abbreviations

MPSoC – Multi-Processor System on Chip
AI – Artificial Intelligence
NoC – Network-on-Chip
SW – Software
IoT – Internet of Things
VM – Virtual Machine
API - Application Programmable Interface
YAML – YAML Ain’t Markup Language
HW – Hardware
K8S – Kubernetes
ETL – Extract, Transform, Load
SQL – Structured Query Language
JSON – javaScript Object Notation
NoSQL – Not Only SQL
FPGA – Field Programmable Gate Array
REST – Representational State Transfer
VL – Virtual Link
VLID – Virtual Link Identifier
FIFO – First in First out
OR-TMS - Optimised Reliable Task and Message Scheduling algorithm
TSMS - Time Slot Message Scheduler
OOP – Object Oriented Programming