Haute école d'ingénierie et d'architecture Fribourg
Hochschule für Technik und Architektur Freiburg

Haute école d'ingénierie et d'architecture Fribourg
Bd de Pérolles 80
CH-1700 Fribourg, Switzerland

Technische Universität Berlin
Straße des 17. Juni 135
10623 Berlin, Germany

Bachelor of Science HES-SO in Engineering

Department: Computer Science and Communication Systems

Carbon-aware spatiotemporal workload shifting using FLUIDOS

Author:
Martin Roch-Neirey

Under the supervision of:
Sébastien Rumley
Vlad Coroamă
Nasir Asadov
Matthias Finkbeiner

Expert:
Pascal Felber

Written as part of a bachelor thesis

Fribourg, HES-SO//Bachelor, August 2, 2024
Version: 1.0

Version history

Version	Date	Modification
0.0.0	2024.05.27	Document creation
0.0.1	2024.06.10	Analysis
0.0.2	2024.06.23	Introduction
0.0.3	2024.07.16	Carbon-aware scheduling algorithm
0.0.4	2024.07.20	Implementation
0.0.5	2024.07.31	Conclusion, implementation
0.0.6	2024.08.01	Implementation
1.0	2024.08.02	Final review before submission

Table 1: Version history

Acknowledgements

First and foremost, I would like to express my sincere gratitude to Sébastien Rumley, my thesis advisor, for having given me the chance to obtain this topic for my Bachelor's thesis, for his continuous support, his confidence, and his many pieces of advice.

I would also like to thank my other main supervisor, Vlad Coroamă, for his support throughout the duration of my thesis. The numerous discussions during our meetings provided me critical reflections and guidance that greatly enhanced the quality of my work. I am particularly grateful for his meticulous review of my final report, which significantly contributed to its clarity and comprehensiveness.

I am also grateful to Stefano Braghin, research engineer and technical lead at IBM Research, for his invaluable help and willingness to iron out bugs and add new features to the framework within which the algorithm was developed.

My deep appreciation also goes to Matthias Finkbeiner, who provided me with an opportunity to join his team as a collaborator.

I am also grateful to the Chair of Sustainability (SEE - TU Berlin) for providing me with the necessary resources and pleasant environment for my work.

Finally, I would like to extend my heartfelt thanks to Nasir Asadov, research associate at SEE and one of my supervisors, who worked closely with me every day during my nine-week-long bachelor thesis. His constant support, guidance, and collaborative spirit were instrumental in the completion of this work. The time and effort he dedicated to assisting me and answering my questions were really helpful.

Contents

List of Figures

List of Tables

List of Equations

1 Introduction

This manuscript documents the Bachelor's thesis of Martin Roch-Neirey from the Haute Ecole d'Ingénierie et d'Architecture of Fribourg, entitled "Carbon-aware spatiotemporal workload shifting using FLUIDOS".

1.1 Context

As part of the Horizon Europe funding program [1] created in 2021, the FLUIDOS[1] project [2] aims to create a flexible, decentralised and secured computing continuum. The project envisages the use of "liquid computing" as a continuum of resources and services enabling the transparent and efficient deployment of applications, independently of the underlying infrastructure. To do so, the technical implementation is based on Kubernetes [3] and LIQO [4], an open-source project developed at POLITO (Politecnico di Turino). Kubernetes is a well-known workload orchestrator used worldwide, while LIQO is used to share resources across multiple Kubernetes clusters, allowing workload shifting between them.

The aim of the project is to allow multiple Kubernetes clusters to be seen as one FLUIDOS cluster, where each Kubernetes cluster represents a FLUIDOS node. This FLUIDOS cluster, composed of multiple Kubernetes clusters, can then execute all types of workloads with the ability to schedule and shift them across multiple locations. The computing continuum allows anyone to join and quit, independently of the computing capacity (most of the time expressed in floating points operations per second - FLOPS) available on each system. A FLUIDOS cluster can then be made of multiple Internet-of-Things (IoT) devices, managed Kubernetes clusters using public cloud providers, and on-premise Kubernetes clusters. It also allows the use of different Kubernetes implementations and overlays like K3s, MiniKube, or KubeEdge.

[1] Flexible, scaLable and secUre decentrallzeD Operating System

The figure 1.1 shows a basic representation of a FLUIDOS cluster with different Kubernetes distributions and hardware.

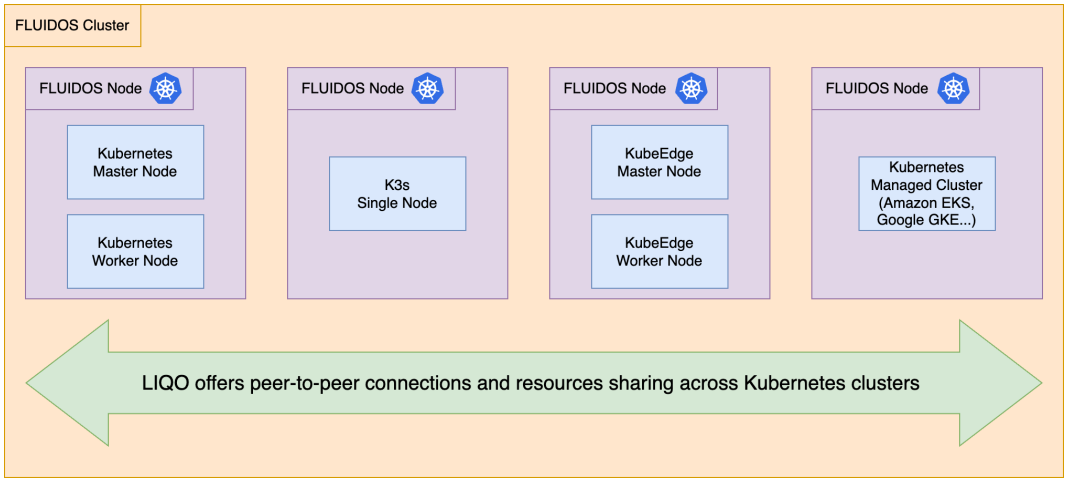

Figure 1.1: Basic representation of a FLUIDOS cluster

Within FLUIDOS, as part of the "Cost-effective and energy-aware infrastructure" work package, the Chair of Sustainable Engineering (SEE) institute of the Technische Universität Berlin (a.k.a. TUB or TU Berlin) is developing a computing model that focuses on optimising energy consumption, carbon emissions, and costs. After optimising the energy consumption per unit load, further carbon optimisations can be achieved through strategic workload scheduling, leveraging the spatial and temporal variations in the carbon intensity of electricity. By scheduling tasks during periods or in locations where the grid is powered by low-carbon sources such as renewables, the carbon footprint of pod execution can be significantly lowered. This strategy requires scheduling algorithms that can predict the availability of low-carbon electricity and resources and dynamically adjust workloads scheduling to align with these optimal periods [5].

Such algorithms have been designed during the last months at the TUB. This project aimed to implement, benchmark, give feedback and improve those algorithms within the FLUIDOS architecture. In addition to this aspect, this project also proposes to evaluate and test the FLUIDOS documentation and architecture by installing it locally and using it as a user might.

1.2 Objectives

The main objectives of this project are:

1. Demonstrate that FLUIDOS can be remotely installed on computers located at HEIA-FR or locally on a laptop and:
   (a) Test and provide feedback on the FLUIDOS documentation.
   (b) Compare this installation with other existing installations (Politecnico di Torino - POLITO, TUB...).
   (c) Demonstrate the capability of FLUIDOS by realising a demonstration of a workload shifting from one location to another.

2. Clarify and document the FLUIDOS API thru which a scheduling algorithm can:
   (a) Interrogate the status of various FLUIDOS locations.
   (b) Schedule jobs across these locations,.

3. Implement a spatiotemporal shifting algorithm that considers the carbon embodied during production in FLUIDOS using the above API and benchmark this algorithm.

In addition to these key objectives, three secondary objectives have been further defined on a "nice-to-have" basis, and remaining capacity after achieving the main objectives permitting:

- Compare the results of the spatiotemporal algorithm with other existing FLUIDOS scheduling algorithms (cost- and latency-effective for example).
- Analyse the carbon- and energy-impact of moving the data needed by the job, and compare with carbon- and energy-savings realised on the compute side.
- Compare the benefits of moving a workload toward green energy with the benefits of moving green energy toward a workload.

1.3 Structure and information about this report

This report is organised in 5 chapters, from the introduction to the conclusion:

The first chapter introduces the thesis, with the context, objectives, and the structure of the report. The second chapter analyses the FLUIDOS architecture and provides a simplified explanation of the project components. Certain elements are deliberately simplified throughout this report as they are not the main elements of this thesis. The third chapter explains the motivation and the design of the carbon-aware scheduling algorithm developed within the FLUIDOS architecture. The chapter 4 explains how the algorithm has been developed and also explains how a simple simulator has been developed to test the algorithms easily. Finally, the chapter 5 concludes this thesis and gives the possible next steps.

At the beginning of each chapter, its structure is explained. Some of the technical vocabulary is explained in the glossary at the end of this document. Each word inserted in the glossary is underlined the first time it appears in the document.

This report was written by hand with partial translation assistance from DeepL and local formulation assistance from ChatGPT (GPT-4 and GPT-4o models). A declaration of honour made by the author is available at the end of the conclusion chapter.

2 people will be mentioned several times in this report:

- Nasir Asadov (`nasir.asadov@tu-berlin.de`), a doctoral student at Technische Universität Berlin and main designer of the algorithm.
- Stefano Braghin (`STEFANOB@ie.ibm.com`), research engineer and technical lead at IBM Research, who mainly developed the model-based meta-orchestrator framework in which the algorithm has been developed and executed.

2 Analysis

This chapter introduces in details the FLUIDOS project, starting with the definition of the computing continuum and the vision, terminology, and context of this thesis within the project.

The second part explains the most important components and workflows of the FLUIDOS architecture related to this bachelor thesis.

Contents

2.1 The FLUIDOS project

This section focuses on the analysis of the whole FLUIDOS project, from its computing continuum to its components, API, and terminology. The main purpose of this analysis is not to provide a complete description of the FLUIDOS project, but more to explain general concepts and key components that are used in this project.

The main goal of the FLUIDOS project is to provide a way to connect multiple Kubernetes clusters together, which are maintained by different entities, thereby allowing them to share their resources and use them more efficiently. Any type of Kubernetes distribution may join the continuum, provide resources, and use resources from others. Within the continuum, each Kubernetes cluster has to establish a peering session with other clusters to discover the resources they have.

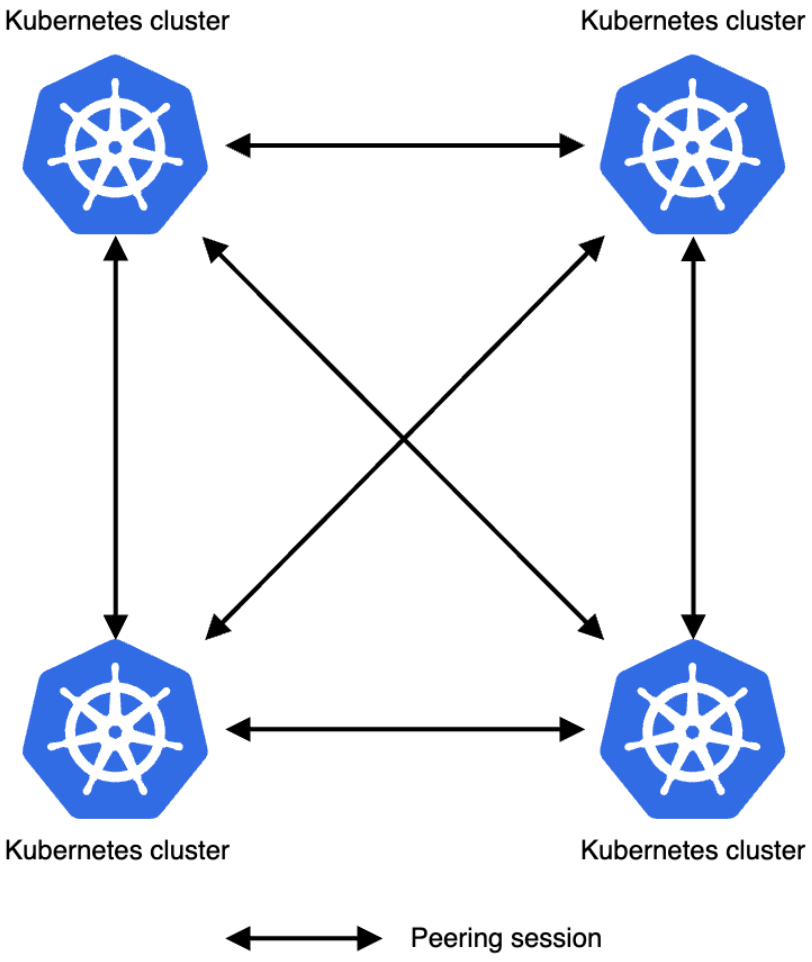

Figure 2.1: Multiple Kubernetes clusters with established peering sessions

In the Figure 2.1, the computing continuum is represented as a fully meshed network using 4 clusters. Each cluster has a peering session with all others.

The FLUIDOS project is based on the "consumer/provider" and "contracts" principles. Each cluster can consume resources from another one by buying its resources using a contract, and provide its own resources to all others. This is useful when all clusters do not have the same specifications and resources (e.g., some may have GPUs, some others may have a better network connectivity). The contracts are created and negociated using a new protocol developed within the FLUIDOS project: the REAR (REsource Advertisement and Reservation) protocol [6].

2.1.1 The computing continuum and vision

This part explains the difference between FLUIDOS' computing continuum and other similar concepts. With today's technology, various forms of computing continuums already exist with multiple applications that rely on different components executed at different locations (e.g. with the data gathered at the edge, aggregated at the telco cloud, and processed in the cloud). Anyway, these IT continuums have a number of shortcomings that FLUIDOS can help to resolve:

- **Deployment transparency:** When deploying a microservices application, it is required to specify for each component the location it must be executed (at the edge, in the cloud...). This makes it more difficult to shift workloads across different locations, as it requires the presence of an overarching orchestrator that re-deploys all the components to their new locations. The FLUIDOS intent-based interface aims to schedule workloads to the best location and also provides dynamic optimizations if needed.

- **Communication transparency:** The communication between multiple micro-services may be configured differently whether they are in the same Kubernetes cluster or not. For example, 2 pods can communicate within the same cluster using their ClusterIP object reference, but if they are on separated clusters they have to use other objects to communicate (for example using a NodePort or a LoadBalancer object). Therefore, services must be explicitly configured to talk to each other based on their respective location. This complicates a lot the possibility to redeploy workloads to different clusters, as the inter-workload communication configuration would not work anymore. With FLUIDOS, all communications between microservices are going through the FLUIDOS virtual network fabric, which guarantees seamless communications independently from the location of each microservice. Therefore, the configuration would be the same regardless of the location of the pods.

- **Resource availability transparency:** With the current technology, a workload can only use the resources available in its own Kubernetes cluster, even if other resources are available at another location of the continuum. This can lead to a service disruption because of a lack of resources on a specific Kubernetes cluster, while other clusters still have available resources. The FLUIDOS project exposes the available resources of each cluster to the others, so that a workload can be schedules or shifted from one cluster to an other, depending on the resources utilisation and requests.

The FLUIDOS project aims to provide a multi-ownership decentralised computing continuum where workloads can be scheduled using intents. The "fluid" term is used in reference of the idea that members can join and leave the continuum at any time, independently of their computing resources. The computing continuum can be made of IoT devices, data centres, personal devices, and evolve days after days.

2.1.2 Terminology

The FLUIDOS project uses numerous terms that are similar with Kubernetes-related environments, and this section aims to define each term so that the rest of the document is clearly understandable. Note that this part only focuses on FLUIDOS-related terms. A more general glossary is available at the end of the document.

The following terms are defined as follows:

- A **Kubernetes node** is a worker machine inside a Kubernetes cluster. It may be a virtual or a physical machine. A Kubernetes node can execute multiple pods.
- A **Kubernetes cluster** is a set of Kubernetes nodes that run containerized workloads, seen as pods. A Kubernetes cluster can be set up on different environments using multiple distributions (Kubernetes, K3s, MiniKube, KubeEdge...). Each of these distribution has specificities and rely to specific use cases.
- A **FLUIDOS node** is a unique computing environment composed of a single of multiple machines. A FLUIDOS node is orchestrated by a single Kubernetes control plane, so it can be seen as one Kubernetes cluster. As this is a Kubernetes cluster, a FLUIDOS node includes a set of resources (CPU, RAM, storage, networking, GPU cards...) that can be shared with other FLUIDOS nodes.
- A **FLUIDOS domain** consists of FLUIDOS nodes within the same administrative domain that have established peering relationships. Within this domain, one of the FLUIDOS nodes is designated as the master, known as the FLUIDOS supernode. The FLUIDOS domain utilises the same interfaces as the FLUIDOS nodes to present their combined information, facilitating hierarchical interactions between different domains.
- A **FLUIDOS supernode** is a FLUIDOS node that serves as the "master" for an entire FLUIDOS domain. It establishes peering relationships on behalf of all nodes within the domain with third-party nodes and acts as the "aggregator" of resources and services for the entire FLUIDOS domain.
- Finally, a **FLUIDOS cluster** is a virtual environment that spans a set of FLUIDOS nodes, even if they belong to different administrative domains. Within this cluster, all Kubernetes objects (such as pods, services, secrets, configmaps, etc.) are available just as they would be in a traditional physical cluster. This setup allows running objects, like microservices, to directly access the computing, storage, and network resources and services allocated to the virtual cluster by each participating FLUIDOS node, without any technical or administrative boundaries.

This terminology is visually shown in the Figure 2.2:

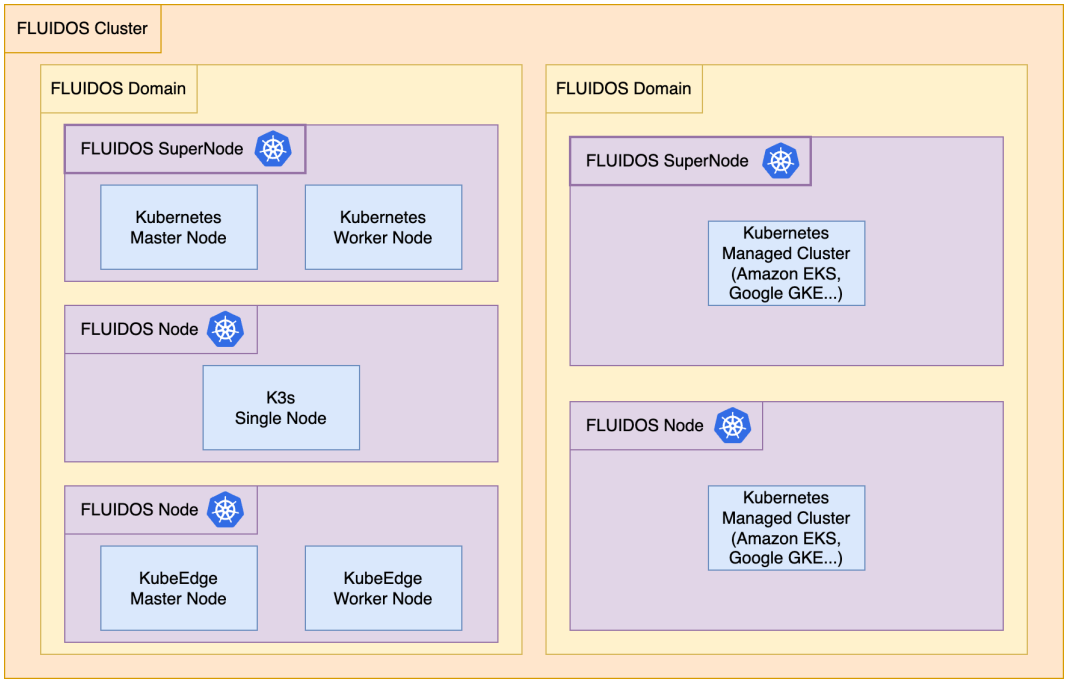

Figure 2.2: Visual representation of a FLUIDOS cluster components

Note that a FLUIDOS node can be made of any type of Kubernetes cluster, including MiniKube, K3s, or managed clusters in the public cloud. All these clusters are seen as FLUIDOS nodes and build together a computing continuum.

There are also other terms that are not shown in this image but still used in the FLUIDOS project:

- The **REAR (REsource Advertisement and Reservation) protocol** is designed to address the complexities of managing resources within a computing continuum. It provides standardized interfaces for interoperability, optimizes resource allocation, and ensures security for workload execution.

- In the context of the REAR protocol, a **Flavor** refers to a set of information describing various kinds of computing resources that can be advertised and reserved. In the FLUIDOS project, a flavor can be seen as a Kubernetes worker node with its technical specifications (CPU, RAM...). In the future, multiple types of flavors will exist (Kubernetes slice, sensors...).

2.1.3 Context of this thesis within the project

The FLUIDOS project is being developed by 16 different entities spread over 10 work packages. The different entities and work packages are listed in the Appendix A. Some actors from this list are mentioned in this report, such as the Fondazione Bruno Kessler (FBK) that is working on multiple work packages, including the WP6.

This project was carried out as part of work package 6 of FLUIDOS, called "Cost-effective and energy-aware infrastructure", in close conjunction with work package 4 'Intent-based decentralised FLUIDOS continuum'. The algorithm was developed using the framework [7] provided by WP4 (later called "Meta-Orchestrator").

2.2 Reference architecture

This section aims to explain how the FLUIDOS architecture is designed and what components are inside. Some components may be excluded from a detailed description, as they are not directly concerned by this thesis.

2.2.1 FLUIDOS node

Each FLUIDOS node (which technically represents a Kubernetes cluster) has different components, as shown in the Figure 2.3:

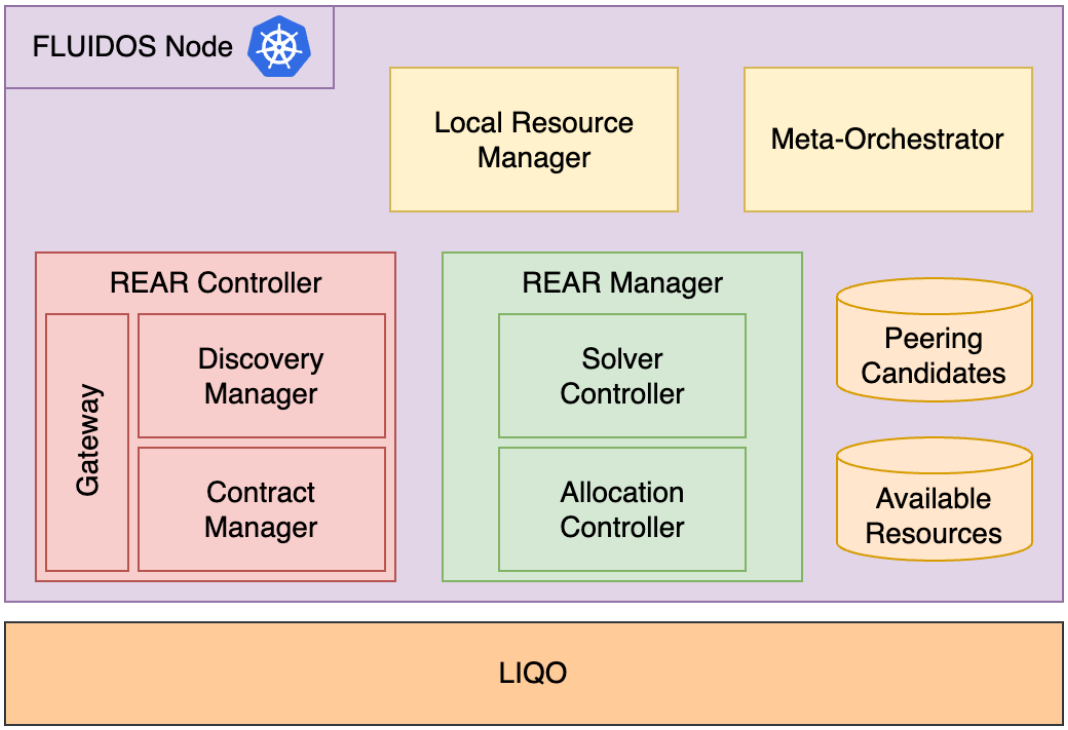

Figure 2.3: Components of a FLUIDOS node

There are 4 internal components, and 2 types of data that are stored on each node:

- The **Local Resource Manager** is in charge of managing the local resources of the current node. It maps resources (such as Kubernetes worker nodes) into flavors. These flavors are then stored as **Available resources**. Flavors are stored in the Kubernetes cluster as CRs.[1]
- The **REAR Manager** is in charge of managing the requests for resources. It is considered as the central and core component of the FLUIDOS node. The Solver Controller is a sub-component used to follow the resources requests active at any time, while the Allocation Controller is the sub-component that interacts with LIQO to ensure peering.
- The **REAR Controller** is in charge of communicating with other FLUIDOS nodes through the REAR protocol. The Discovery Manager is in charge pf searching other suitable FLUIDOS nodes. These nodes are stored as **Peering Candidates** on the local node. The Contract Manager manages the flavors that are reserved and bought through the REAR protocol by other nodes. Finally, the Gateway implements an HTTP client and an HTTP server, as well as the REAR protocol.
- The **Meta-Orchestrator**, also known as model-based meta-orchestrator, is explained in detail in Section 2.2.2 below.

The LIQO underlay is used to establish the peering and communication between FLUIDOS nodes for the scheduling and pod execution part. As LIQO is well integrated in the FLUIDOS architecture, there is a guarantee that it shows and shares the exact amount of resources previously traded between the FLUIDOS nodes using the REAR protocol.9

These components are replicated over all FLUIDOS nodes. Each node directly interacts with LIQO to establish peering and communication with other nodes, as shown on the Figure 2.4 representing a 2 nodes FLUIDOS cluster:

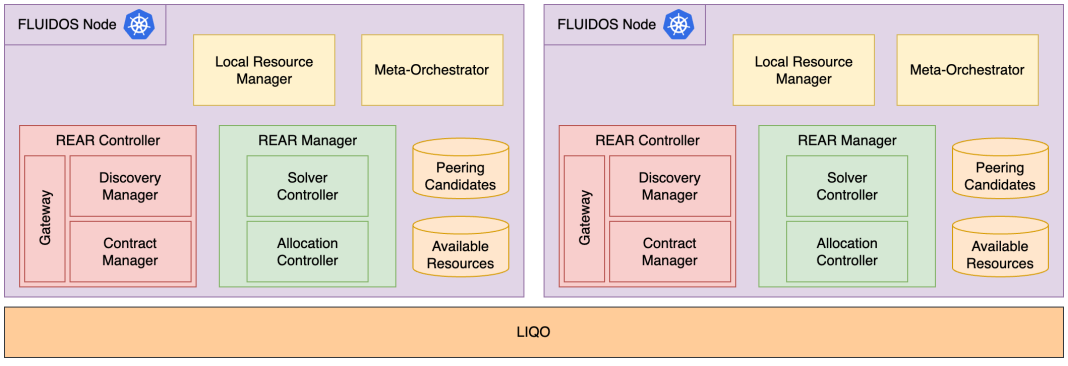

Figure 2.4: Example of a 2 nodes FLUIDOS cluster

[1] A CR (custom resource) is an extension of the Kubernetes API that is not necessarily available in a default Kubernetes installation.

2.2.2 Model-based meta-orchestrator

The model-based meta-orchestrator is the component in charge of scheduling workloads on the FLUIDOS continuum based on requests it receives. Those requests are called intents, and provide information on how the workload should be executed and which scheduling algorithm should be used (e.g., latency-aware or carbon-aware). It is inside this component that the carbon-aware scheduling algorithm is developed.

When a job is submitted to FLUIDOS, a component of the architecture is called to choose the best FLUIDOS node and flavor to execute it. This component is called the Meta-Orchestrator and is executed as an operator on each FLUIDOS node (meaning this is a Kubernetes Operator executed on each cluster).

The Meta-Orchestrator decides which specific flavor and related `PeeringCandidate` to buy using the REAR protocol. This choice is made using the logic of the model chosen when submitting the job to FLUIDOS. Multiple algorithms exists for specific purposes. For example, an algorithm tries to avoid latency between 2 locations by choosing flavors and peering candidates with the best bandwidth available, and spatially near from each other.

The main purpose of this bachelor thesis is to develop a carbon-aware scheduling algorithm that will be implemented as part of the FLUIDOS meta-orchestrator. Each algorithm can rely on metrics gathered by the flavors and PeeringCandidates to choose wisely the final location of the job.

2.2.3 Simplified workflow example

The Figure 2.5 presents a simplified workflow of a node executing FLUIDOS in a cluster, and having a job submitted on it. Note that this version represents a simplified view of the overall workflow.

1. When a FLUIDOS node is turned on, the Local Resource Manager detects the Kubernetes nodes labeled as FLUIDOS workers and maps each node into a flavor. These flavors become the "Available Resources" of the FLUIDOS node.

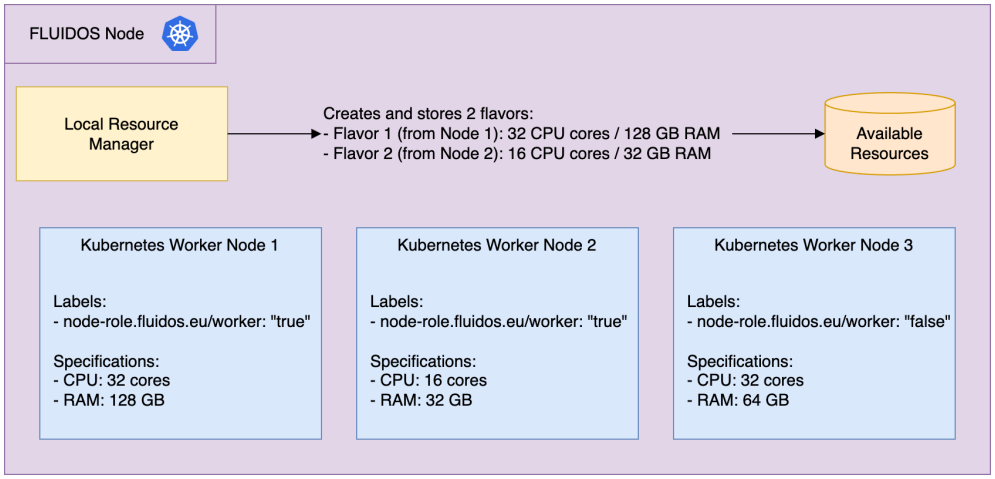

Figure 2.5: Workflow example - Step 1: Resource detection and creation

As the Figure 2.5 shows, this process is executed on each FLUIDOS node at the startup. Each node is responsible for its own local flavors, and can reserve and buy remote ones using the REAR protocol.

2. The user of FLUIDOS can then use the `kubectl` FLUIDOS plugin to push new jobs on the FLUIDOS cluster. The user writes an intent, which represents a written declaration of what it wants to have in the cluster. An intent example is shown below (truncated):

```
apiVersion: apps/v1
kind: Deployment
metadata:
  annotations:
    fluidos-intent-deadline: "12"
  labels:
    app.kubernetes.io/name: "intent-example"
spec:
  replicas: 1
  selector:
    matchLabels:
      name: intent-operator
  template:
    metadata:
      labels:
```

```
        name: intent-operator
        app.kubernetes.io/name: "intent-example"
    spec:
      containers:
        - name: intent-example-container
          image: "foo/bar"
          command:
            - /hello-world
```

Note the FLUIDOS-specific annotations, specifying what the user wants. In this case, the pod must be started within the next 12 hours. This intent can then be sent to the cluster with the following command:

```
kubectl fluidos -f intent-example.yaml
```

3. Once the intent is processed by the `kubectl` FLUIDOS plugin, the Meta-Orchestrator is called to select the appropriate scheduler model, depending on the intents of the user.[2]

4. When the model has been chosen, the algorithm developed within this model executes its logic and outputs as a result the required resources and specifications to schedule the pod. For example, the algorithm may output a specific flavor that is low-latency. In the context of this project, the algorithm outputs a flavor and a delay, defining the location and time where and when the pod will be executed.

5. The FLUIDOS resources are then reserved, purchased and ready to execute the pod using the REAR protocol. The pod is then scheduled on the selected resource using LIQO.

2.2.4 Existing installation

During this project, 2 different installations have been setup locally. These FLUIDOS installations are explained in details in the 2 demonstration of this report: the 4.1 "FLUIDOS remote workload scheduling demonstration" section and the 4.3 "Demonstration of the MBMO and the carbon-aware scheduler" section.

An objective of this project (1.b) was to compare these installations with other existing ones (at POLITO, TUB...). After some discussions with POLITO engineers and others involved in the project, it became clear that there was no definitive FLUIDOS cluster yet. All development is being carried out using KinD to simulate clusters locally. In the final days of the project, discussions were held to start preparing for the deployment of FLUIDOS on an initial physical infrastructure using RaspberryPi.

[2]This functionality is not fully implemented yet and the model must be chosen by hand in the code. The WP4 responsibles are still working on this feature.

3 Carbon-aware scheduling algorithm

This chapter explains in detail how the carbon-aware scheduling is designed and what problem it tries to solve. The first part explains the motivation of designing such an algorithm, and the second part explains the problem and dives into detailed explanations.

The algorithm is explained from a theoretical point of view with the definition of the optimisation problem and definitions, and then presented in pseudo-code.

Contents

3.1 Motivation

The "Motivation" section has been written in close connection with the work carried out before the start of the thesis. This previous work was mainly carried out by Nasir Asadov (*nasir.asadov@tu-berlin.de*).

The motivation behind developing a carbon-aware scheduling algorithm is driven by the rising energy demand and corresponding carbon footprint of computing [8]. As trends like AI, IoT, streaming, and the shift towards cloud and edge computing increase [9], the energy consumption and environmental impact of data centres have grown significantly. Traditional hardware efficiency gains [10] are no longer sufficient to offset this increase, necessitating new paradigms for energy- and carbon-efficient computing. The primary goal is to devise strategies that can reduce the carbon footprint of computation without compromising performance, through methods such as deploying energy-efficient computing resources and utilising low-carbon electricity sources.

One effective approach to reducing the carbon footprint in distributed systems is through carbon-aware computing [11], which involves shifting computing loads in space and time to leverage periods of low-carbon electricity availability. This strategy, known as carbon-aware workload scheduling, focuses on minimising the carbon emissions associated with the operation of computational tasks by aligning their execution with times or locations where renewable energy is more prevalent. Unlike traditional scheduling methods that prioritise performance metrics like speed [12] and resource utilisation [13], carbon-aware scheduling integrates environmental considerations by prioritising tasks based on the varying carbon intensity of the electricity grid.

3.2 Design of the algorithm

This section explains what problem the algorithm tries to solve, and also presents it in pseudo-code.

3.2.1 Problem definition

The problem can be seen as a NP-complete mixed-integer linear problem. The goal is to minimise the total carbon footprint of workloads by optimally scheduling them across geographically distributed nodes and timeslots. The carbon footprint consists of operational emissions and embodied emissions.

The algorithm must ensure that for each case (combination of timeslot and node), the node will be able to execute the pod with its own resources that are still available, and that the timeslot is acceptable for the pod (meaning the timeslot starts before, or at least at the same time as the deadline defined in the intent request file of the pod).

Below is shown a visualisation of the problem:

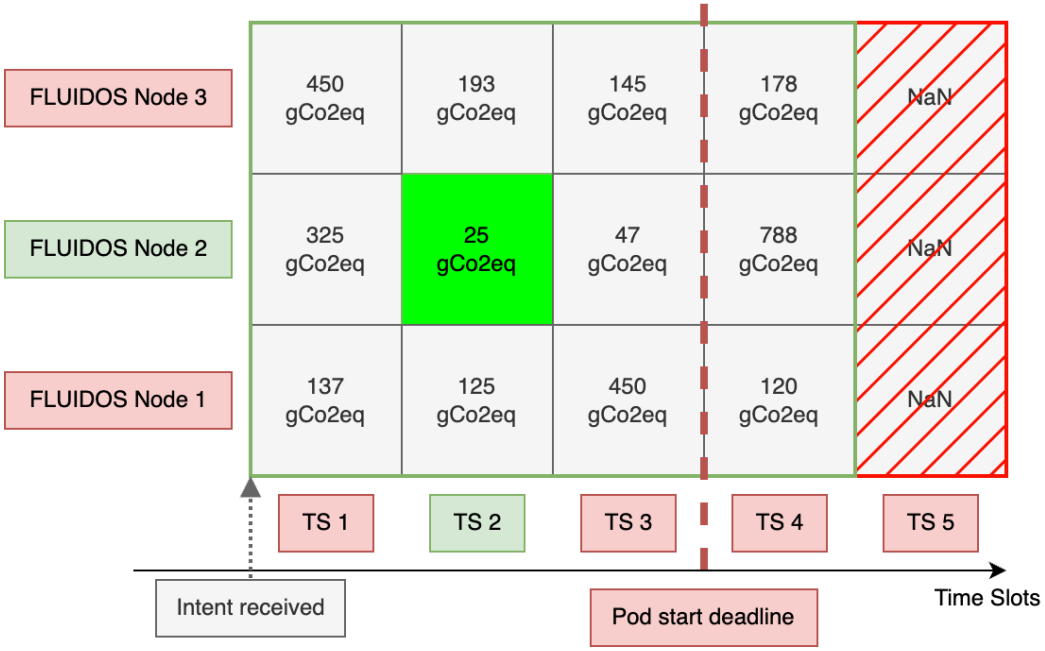

Figure 3.1: The algorithm selects the best combination of a timeslot and a node, so that the emissions are as low as possible

In the example presented in the Figure 3.1, the timeslot no. 5 is not valid because it starts after the deadline of the pod. The combination `[TS2; FLUIDOS node 2]` is considered as the best solution regarding the carbon emission problem. The algorithm still has to ensure that the node can handle the pod on this timeslot with its own resources.

Assuming that each timeslot is 2 hours long and the datasource for the carbon intensity forecast outputs 1 hour long timeslots, the average of 2 consecutive forecasted data can be considered for each timeslot of the algorithm. The goal is then to select the lowest possible value for each location, and compare them while ensuring that the node will be able to handle the pod.

The way the algorithm translates the forecast that comes from the Electricity Maps API to usable data is shown below.

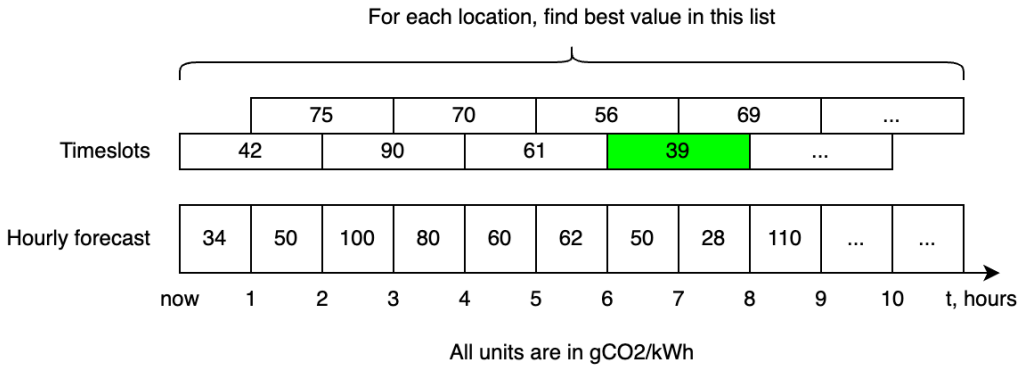

Figure 3.2: The forecasted data are reorganised in 2 hours timeslots

On the Figure 3.2, the best timeslot for the given location is the one that has 39 gCO2/kWh, which is the average of the 2 consecutive 1 hour long forecasted data (50 and 28 gCO2/kWh).

The concepts presented in the following subsections (i.e., definitions, carbon emissions calculation, objective function, and the pseudo-code algorithm) are heavily inspired by the work of Nasir Asadov. Over the course of this thesis, various discussions and brainstorming sessions led to several clarifications and punctual theoretical improvements, but most of the work had been done before.

3.2.2 Definitions

This part sets out all the required definitions to understand the algorithm.

P : Set of pods, indexed by i
N : Set of nodes, indexed by j
T : Set of time slots, indexed by k
$\overline{\text{CPU}}_i, \overline{\text{RAM}}_i$: Resource requirements of pod i
U_i : Duration of pod i
Deadline_i : Deadline of pod i
$\widehat{\text{ACI}}_{jk}$: Predicted average carbon intensity for node j at time slot k
$\widehat{\text{Util}}_{jk}$: Predicted utilisation for node j at time slot k
c_j^{emb} : Embodied emissions rate for node j
L_j : Lifetime of node j
C_j^{emb} : Total embodied emissions of node j
P_i : Power consumption of pod i
TimeSlotLength : Length of each time slot
M : A large number for constraints
$x_{ijk} \in \{0, 1\}$: 1 if pod i is scheduled on node j at time slot k, 0 otherwise

3.2.3 Carbon Emissions Calculation

This part explains how the carbon emission is calculated for each pod i, on node j, at timeslot k.

Operational Emissions
The operational emissions for scheduling pod i on node j at timeslot k are given by:

$$C_{ijk}^{op} = \widehat{\text{ACI}}_{jk} \times \text{U}_i \times \text{P}_i \times x_{ijk}, \quad \forall i \in P, j \in N, k \in T \tag{3.1}$$

Embodied Emissions
The embodied emissions for scheduling pod i on node j are given by:

$$C_{ijk}^{emb} = c_j^{emb} \times \text{U}_i \times \frac{\text{P}_i}{P_{tot}} \times x_{ijk}, \quad \forall i \in P, j \in N, k \in T \tag{3.2}$$

The embodied emission is not dependent on the chosen timeslot, as the production emissions have already been emitted and cannot be affected by the CO_2 emission forecast.

3.2.4 Objective Function

Minimise the total carbon emissions, consisting of operational and embodied emissions:

$$\min \sum_{i \in P} \sum_{j \in N} \sum_{k \in T} \left(C_{ijk}^{op} + C_{ijk}^{emb} \right) \tag{3.3}$$

3.2.5 Constraints

Scheduling Constraint
Each pod must be scheduled exactly once:

$$\sum_{j \in N} \sum_{k \in T} x_{ijk} = 1 \quad \forall i \in P \tag{3.4}$$

Resource Constraints
The resource usage of all pods scheduled on a node at a given time slot must not exceed the node's capacity:

$$\sum_{i \in P} \overline{\text{CPU}}_i \times x_{ijk} \leq \text{CPU}_j \times \left(1 - \widehat{\text{Util}}_{jk}\right), \quad \forall j \in N, \forall k \in T \tag{3.5}$$

$$\sum_{i \in P} \overline{\text{RAM}}_i \times x_{ijk} \leq \text{RAM}_j \times \left(1 - \widehat{\text{Util}}_{jk}\right), \quad \forall j \in N, \forall k \in T \tag{3.6}$$

Utilisation Constraint
The total duration of pods scheduled on a node in a given time slot must not exceed the available capacity:

$$\sum_{i \in P} \text{U}_i \times x_{ijk} \leq \left(1 - \widehat{\text{Util}}_{jk}\right) \times \text{TimeSlotLength}, \quad \forall j \in N, k \in T \tag{3.7}$$

3.2.6 Deadline Constraint

Pods must be scheduled within their deadlines:

$$\sum_{k \in T} \text{End}_k \times x_{ijk} \leq \text{Deadline}_i, \quad \forall i \in P \tag{3.8}$$

3.2.7 Algorithm

The algorithm complexity is quadratic, following the $O(n^2)$ form. It schedules pods in a carbon-aware manner across geographically distributed nodes and time slots. It considers resource availability, utilisation, and deadline constraints to minimise carbon emissions. The scheduling algorithm is triggered upon the arrival of a new pod request (intent file).

Algorithm 1 Carbon-Aware Spatiotemporal Shifting Algorithm with On-Demand Scheduling

Data: Set of nodes N, set of time slots T, resource requirements and deadlines of pods, predicted average carbon intensities and utilisation of nodes.

Result: Feasible schedule of pods on nodes and time slots that minimises total carbon emissions.

Initialise an empty schedule S Initialise resource availability R_{jk} for all nodes $j \in N$ and time slots $k \in T$ Calculate embodied emissions rate c_j^{emb} for each node j as $c_j^{emb} = \frac{C_j^{emb}}{L_j}$

Initialise $C^* \leftarrow \infty$
Initialise $j^* \leftarrow -1$
Initialise $k^* \leftarrow -1$

foreach *time slot* $k \in T$ **do**
  if $End_k \leq Deadline_i$ **then**
    foreach *node* $j \in N$ **do**
      if $\overline{CPU}_i \leq CPU_j - \sum_{m \in S_{jk}} \overline{CPU}_m$
       $\wedge \overline{RAM}_i \leq RAM_j - \sum_{m \in S_{jk}} \overline{RAM}_m$
       then
        if $\sum_{m \in S_{jk}} U_m + U_i \leq \left(1 - \widehat{Util}_{jk}\right) \times TimeSlotLength$ **then**
          $C_{ijk}^{op} = \widehat{\mathrm{ACI}}_{jk} \times \mathrm{U}_i \times \mathrm{P}_i$
          $P_{tot} = \sum_{m \in S_{jk}} \mathrm{P}_m + \mathrm{P}_i$
          $C_{tot} = C_{ijk}^{op} + \left(c_j^{emb} \times \mathrm{U}_i \times \frac{\mathrm{P}_i}{P_{tot}}\right)$
          if $C_{tot} < C^*$ **then**
            $C^* \leftarrow C_{tot} \quad j^* \leftarrow j \quad k^* \leftarrow k$
          end
        end
      end
    end
  end
end
if $j^* \neq -1 \wedge k^* \neq -1$ **then**
  Schedule pod i on node j^* at time slot k^*
end

4 Implementation

This chapter describes how the project has been technically implemented. The first part shows a demonstration of FLUIDOS' capability to schedule workload on a remote location. The second part explains in detail how the algorithm has been developed within the model-based Meta-Orchestrator framework. The third section shows how to run the algorithm. The penultimate section describes how and why an algorithm simulator has been developed during the project. Finally, a small section that explains other parts of the work done ends the chapter.

Contents

4.1 FLUIDOS remote workload scheduling demonstration

This demonstration shows the ability of FLUIDOS to schedule a pod from one node to another one (in other words, to schedule a pod from one Kubernetes cluster to another one). It is based on the available demonstration on the node GitHub repository [14]. The architecture that has been set up using KinD [15] (Kubernetes in Docker) is shown in the Figure 4.1:

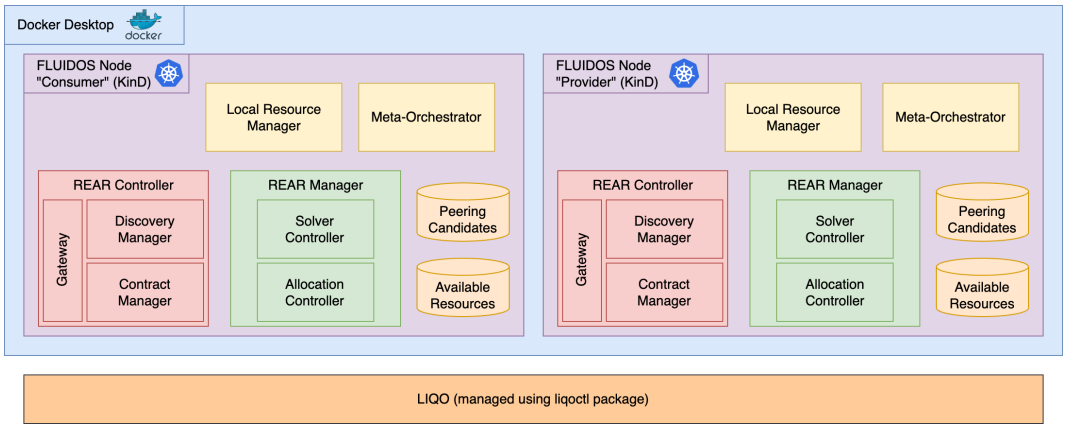

Figure 4.1: Architecture set up for the basic FLUIDOS demonstration

The goal of this demonstration is to show that the Consumer cluster (on the left) can establish a peering session with the Provider cluster (on the right) using LIQO, and start scheduling workloads remotely. This demonstration corresponds to the objectives 1.a and 1.c of the 1.2 "Objectives" section.

The demonstration procedure is described in the README file of the repository. Note that both `KinD` and `liqoctl` are required. The important stages are also listed and explained in details below for a better understanding:

1. Clone the repository:

```
git clone https://github.com/fluidos-project/node.git
```

2. Move into the KinD testbed folder:

```
cd testbed/kind
```

3. Execute the `setup.sh` script to install the 2 FLUIDOS nodes using KinD:

```
chmod +x setup.sh
./setup.sh
```

This script deploys the 2 Kubernetes clusters using KinD, installs the CRDs of FLUIDOS and also installs the LIQO stack on both clusters.

4. After the installation process, both FLUIDOS nodes are being executed. It is possible to check the pods on each of them using the following commands:

```
export KUBECONFIG=consumer/config
kubectl get pods -n fluidos

export KUBECONFIG=provider/config
kubectl get pods -n fluidos
```

As an example, here is the output for the consumer cluster:

```
NAME                                              READY   STATUS   [...]
node-local-resource-manager-77657dbdf4-8c6z9      1/1     Running  [...]
node-rear-controller-67bdd49dbb-vrptw             1/1     Running  [...]
node-rear-manager-79c96d6b75-5l8nc                1/1     Running  [...]
```

As each Kubernetes cluster represents a FLUIDOS node, each of them executes the 3 pods representing the Local Resource Manager, the REAR Controller and the REAR Manager. Note that the Meta-Orchestrator is not visible here for 2 reasons:

(a) It is a Kubernetes operator, not a pod.

(b) It is not implemented yet in this demonstration provided by the FLUIDOS node repository. A demonstration of the Meta-Orchestrator component is available in the 4.1.2 “Meta-Orchestrator demonstration” section.

It is also possible to check the state of the LIQO peering session between the 2 clusters using the following command:

```
liqoctl status peer
```

The Figure 4.2 shows the status of the LIQO peering session. The status is set to `Connected`, and the resources offered by the provider cluster to the consumer cluster are shown.

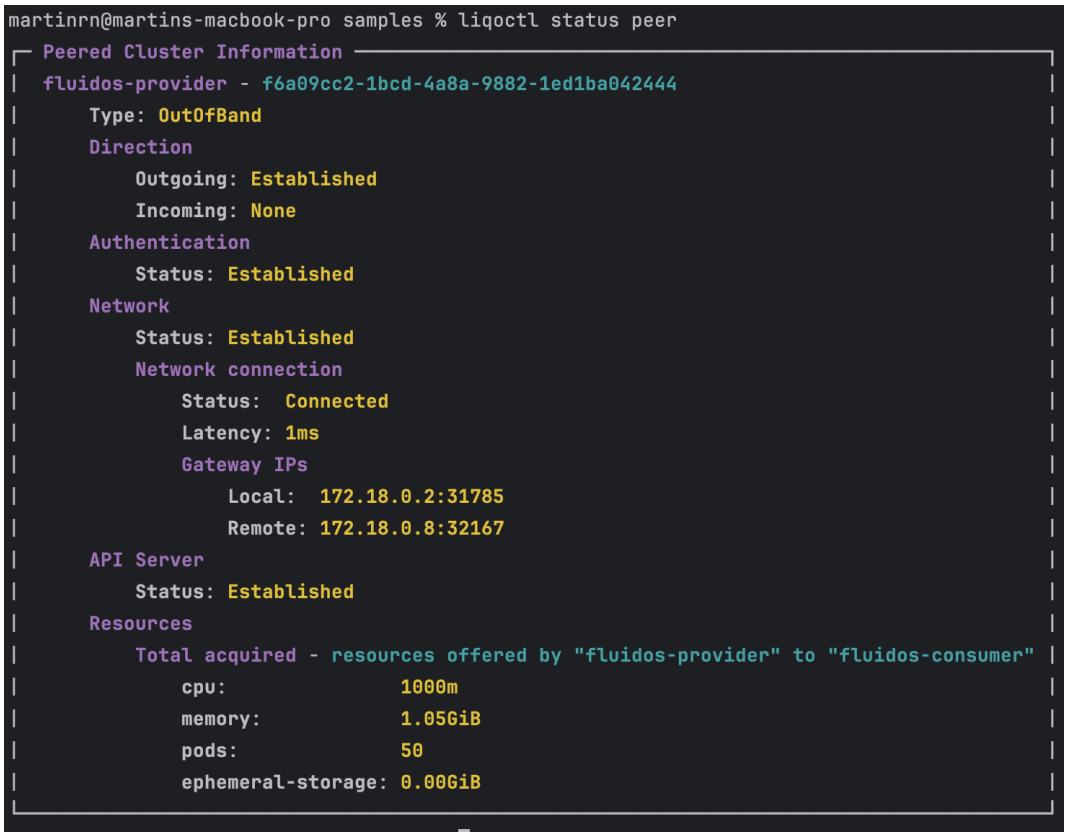

Figure 4.2: Result of the command "liqoctl status peer", showing the peering session between the consumer and provider clusters

5. Now that each FLUIDOS node is up and running, the next step is to deploy a *Solver* CR to allow the deployment of workloads. The Solver resource specifies which resource is requested and any specific requirement. A solver object is ready to use in the *deployments/node/samples* for the consumer cluster:

```
cd ../../deployments/node/samples
export KUBECONFIG=../../../testbed/kind/consumer/config
kubectl apply -f solver.yaml
```

As the solver is a Kubernetes resource, its deployment can be checked using the following command:

```
kubectl get solver -n fluidos
```

The output should look like this:

```
NAMESPACE  NAME  INTENT ID  FIND CANDIDATE  RESERVE AND BUY  PEERING [...]
fluidos    solv  intent     true            true             false   [...]
```

Other custom resources have been created, and the following commands shows their status:

```
kubectl get flavors.nodecore.fluidos.eu -n fluidos
kubectl get discoveries.advertisement.fluidos.eu -n fluidos
kubectl get reservations.reservation.fluidos.eu -n fluidos
kubectl get contracts.reservation.fluidos.eu -n fluidos
kubectl get peeringcandidates.advertisement.fluidos.eu -n fluidos
kubectl get transactions.reservation.fluidos.eu -n fluidos
```

6. At this point, the infrastructure for the resource sharing has been created. It is then possible to schedule workloads from the consumer cluster to the provider one. To do so, create a namespace, offload it using LIQO, and deploy a pod within this namespace. Perform these actions from the consumer cluster.

```
kubectl create namespace demo
liqoctl offload namespace demo --pod-offloading-strategy Remote
kubectl apply -f nginx-deployment.yaml -n demo
```

7. It is then possible to check the deployment state from the consumer side with the following commands:

```
export KUBECONFIG=../../../testbed/kind/consumer/config
kubectl get pods -n demo
```

The output is the following:

```
NAME                           READY   STATUS    RESTARTS   AGE
nginx-deployment-7d4c8fc69d   1/1     Running   0          2m41s
```

8. From the provider side, the pod can also be seen (this is where the pod is really excuted):

```
export KUBECONFIG=../../../testbed/kind/provider/config
kubectl get pods -A
```

The output is the following:

```
NAMESPACE                    NAME                           STATUS  [...]
demo-fluidos-consumer-9b1a29 nginx-deployment-7d4c8fc69d  Running [...]
```

The output is slightly different, as the namespace is not exactly the same. The consumer cluster has created the `demo` namespace, and offloaded it to the provider cluster. From the provider point of view, the namespace is called `demo-fluidos-consumer-9b1a29`, showing that it comes from the consumer cluster. The pod executed has exactly the same name on both clusters.

This demonstration showed that using the LIQO stack, FLUIDOS is capable of offloading workloads from one node to another.

Note that during this demonstration, the pod is forced to be scheduler on the remote cluster. This demonstration is made in a testing environment designed to test the features of the FLUIDOS node. In a production environment, the pod request (a.k.a. intent file) must be processed by the model-based Meta-Orchestrator. It will select a specific scheduler (latency-aware, cost-effective, carbon-aware...) that will schedule the pod according to its own logic.

The next section describes how the carbon-aware scheduling algorithm has been developed within the model-based Meta-Orchestrator. A demonstration of this MBMO (model-based Meta-Orchestrator) is also available in the 4.3 Demonstration of the Meta-Orchestration and the carbon-aware scheduler section.

4.2 Development of the algorithm within the FLUIDOS architecture

The biggest part of the source code of the algorithm is available on the WP4 GitHub repository [7] in the `fluidos_model_orchestrator/model/carbon_aware` folder. It is also possible to check all the changes made by the project by reading the Pull Request #41 [16].

The carbon-aware spatiotemporal scheduling algorithm has been developed within the FLUIDOS Meta-Orchestrator component, which is a Kubernetes operator running on each FLUIDOS node (in other words on each Kubernetes cluster). The algorithm is seen as a specific scheduler that can be called by the operator, when the intent file specifies the carbon-aware aspect. Other schedulers are being developed, such as latency-aware and cost-effective (the FLUIDOS architecture introduces the notion of cost when reserving remote resources using the REAR protocol). The development of this algorithm corresponds to the objective 3 of the 1.2 “Objectives” section.

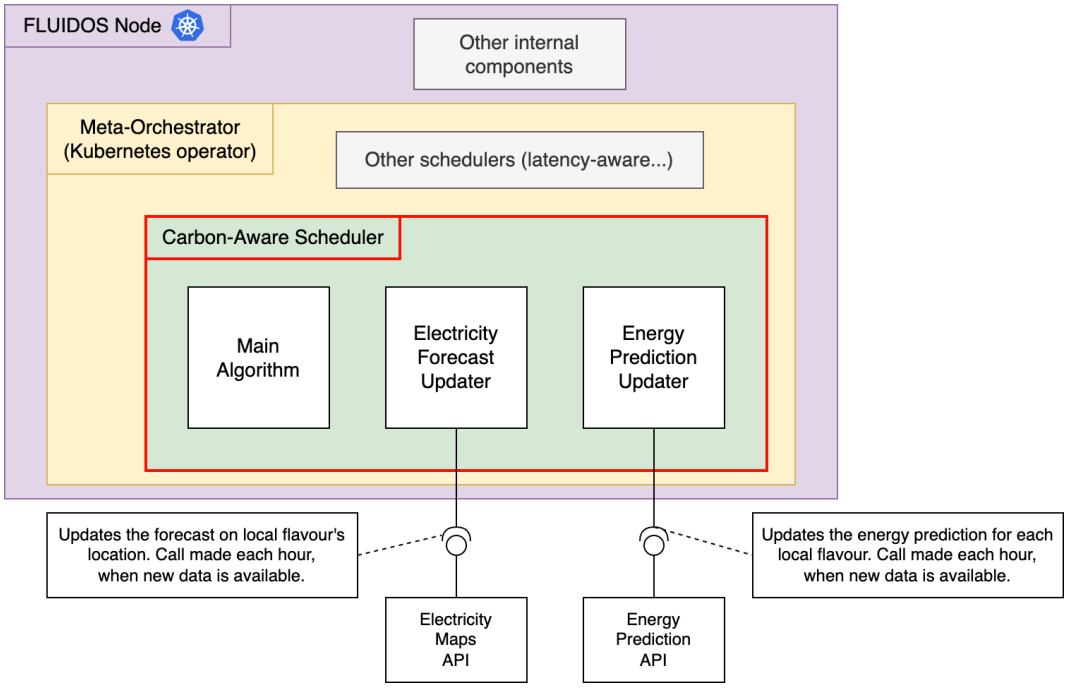

Figure 4.3: Environment of execution and components diagram of the algorithm within the FLUIDOS architecture

The Figure 4.3 shows the environment in which the algorithm is developed and executed. It also shows the external components that it interacts with.

The algorithm is made of three blocks:

1. The main algorithm, that is called by the Meta-Orchestrator and computes the problem explained in the 3.2 section.

2. The electricity forecast updater that uses an external API provided by the Electricity Maps company [17]. This updater is called each hour as a standalone component by the operator to update the forecast of each local flavor. This way, each FLUIDOS node is responsible of updating its own local flavors. When the algorithm needs to gather the forecasted data from remote flavors, it can simply take the information using the REAR protocol. The data model used is specified in the REAR-data-models GitHub repository [18].

The different useful information regarding the carbon-aware scheduler are shown in the Figure 4.4:

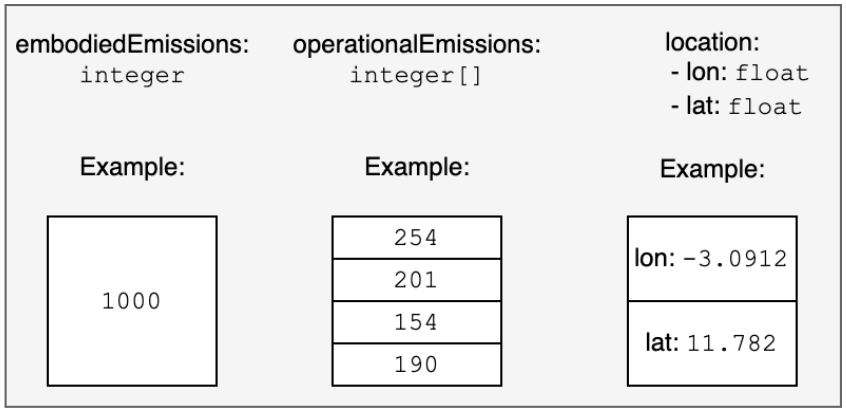

Figure 4.4: Data model used by the carbon-aware scheduler

3. The energy prediction updater, that is a component used to predict the resources utilization of nodes on specific timeslots. As the machine learning model developed by FBK did not correspond exactly to the needs of the algorithm, and needed to be reworked in depth, this component could not be developed. As a workaround, simple functions that randomly generate predictions were developed to obtain usable data.

4.2.1 Project structure

The algorithm has been developed within the WP4 GitHub repository (model-based Meta-Orchestrator). The carbon-aware algorithm uses different files and datasources to compute the optimization problem. The following files structure has been created within the model-based Meta-Orchestrator GitHub repository:

```
fluidos_model_orchestrator/
├── daemons_and_times/
│   └── flavor.py
└── model/
    └── carbon_aware/
        ├── classes/
        │   ├── carbon_aware_flavor.py
        │   ├── carbon_aware_pod.py
        │   └── carbon_aware_timeslot.py
        ├── fakers/
        │   └── workload_prediction_generator.py
        ├── forecast_updater.py
        └── orchestrator.py
```

The `carbon_aware` folder contains all the classes and functions used by the algorithm.

The implementation has been done on the `wp6-algorithm` branch of the MBMO GitHub repository (also called WP4 repository) [7]. A Pull Request has been created to easily review the code added during the thesis [16]. It has been needed multiple time to rebase the `wp6-algorithm` on the `main` branch to take advantage of the new features being developed by Stefano Braghin.

4.2.2 Classes

As the data used by the algorithm does not only rely on FLUIDOS' information, the three classes have been created to gather information from FLUIDOS flavors and other datasources (such as external APIs). These are very simple classes files that defines the objects required by the algorithm.

For example, the `CarbonAwareFlavor` class defines a FLUIDOS flavor with more information, such as embodied emissions and the electricity forecast for the location of the flavor:

```
class CarbonAwareFlavor:
    def __init__(self, id, embodiedCarbon, lifetime, totalCpu, totalRam,
    ↪  totalStorage, forecast):
        self.id = id
        self.embodiedCarbon = embodiedCarbon
        self.lifetime = lifetime
        self.totalCpu = totalCpu
        self.totalRam = totalRam
        self.totalStorage = totalStorage
        self.forecast = forecast
```

The same principle is used for both `CarbonAwarePod` and `CarbonAwareTimeslot` classes. The `CarbonAwarePod` class is the following:

```
from datetime import timedelta, datetime

class CarbonAwarePod:
    def __init__(self, id, deadline_hours, duration, powerConsumption,
    ↪  cpuRequest, ramRequest, storageRequest):
        self.id = id
        self.deadline = processDeadline(self, deadline_hours)
        self.duration = duration
        self.powerConsumption = powerConsumption
        self.cpuRequest = cpuRequest
        self.ramRequest = ramRequest
        self.storageRequest = storageRequest

def processDeadline(self, deadline_hours):
    now = datetime.now()
    delta = timedelta(hours=deadline_hours)
    return now + delta
```

The unusual aspects of this pod are the deadline and duration attributes, which refers to the ability to defer the execution of the pod in the future and the time expected for the pod to perform its job. Note that the deadline of the pod refers to the last time the pod can be started. The duration is used to compute both operational and embodied emissions during the execution of the algorithm.

The `CarbonAwareTimeslot` object refers to a time slot, within which a pod can be executed:

```
from datetime import timedelta, datetime

class CarbonAwareTimeslot:
    def __init__(self, id, startYear, startMonth, startDay, startHour, length):
        self.id = id
        self.start = datetime(startYear, startMonth, startDay, startHour)
        self.length = timedelta(hours=length)

    def getEnd(self):
        return self.start + self.length

    def getStart(self):
        return self.start
```

A timeslot must be valid (starting before or in the same time of the deadline of a pod) to handle a pod. To check if a timeslot is valid, the following function has been developed:

```
def _is_timeslot_valid(timeslot: CarbonAwareTimeslot, pod: CarbonAwarePod) ->
↪  bool:
    return (pod.deadline > timeslot.getStart()) & (datetime.now() <=
    ↪  timeslot.getEnd())
```

It it returns true, then the timeslot is valid and the algorithm starts iterating over all nodes to check if they can handle the pod, and find the lowest carbon value using the forecasted data for each location.

4.2.3 Fakers

During the implementation phase, some parts of the required APIs were not available, such as the energy and workload prediction model. To overcome this problem, a very simple file has been created to randomly generate a number representing the use of a resource. This function is available in the `workload_prediction_generator.py` file. Other similar functions were developed during the course of the project (i.e., regarding the electricity forecast data), but the progress made by WP4 in developing the framework meant that they were no longer necessary.

```
import random

def generate_resource_prediction(totalResource):
    return random.randint(0, totalResource)
```

This function is only used to generate a fake resource utilisation prediction. Once the machine learning will be updated and ready to use, the developers simply need to replace this function with a call to the model.

4.2.4 Forecast updater

The forecast updater is the component in charge of updating the local flavors each hour with new data coming from the Electricity Maps API. The FLUIDOS project has a private API key to obtain forecast data on different regions and countries around the world.

Technically, the component has been implemented in 2 different parts:

1. The `forecast_updater.py` file, which concentrates all the functions used to gather new forecast data and update the local flavors. This file has been developed only by Martin Roch-Neirey. It contains 3 functions:

   (a) The `_get_live_carbon_intensity(lat, lon)` function, that takes into parameter a latitude and longitude coordinates, calls the Electricity Maps API and returns the value of the live electricity carbon intensity at the given coordinates.

   ```
   def _get_live_carbon_intensity(lat, lon):
       BASE_URL = 'https://api.electricitymap.org/v3'
       API_KEY = CONFIGURATION.api_keys['ELECTRICITY_MAP_API_KEY']
       HEADERS = {'auth-token': str(API_KEY)}
       url = f"{BASE_URL}/carbon-intensity/latest"
       params = {'lat': lat, 'lon': lon}

       response = requests.get(url, headers=HEADERS, params=params)

       if response.status_code == 200:
           return response.json()["carbonIntensity"]
       else:
           logging.exception(f"Error fetching live data:
           ↪ {response.status_code} - {response.text}")
           return None
   ```

   (b) The `_get_forecasted_carbon_intensity(lat, lon)`, which is very similar to the `_get_live_carbon_intensity(lat, lon)` but outputs a list of values instead of one value. Each item of the list represents the forecast for 1 hour. The Electricity Maps API allows to gather forecast up to 48 hours. These values are updated each hour.

(c) The `update_local_flavor_forecasted_data(flavor: Flavor, namespace: str)` that takes into parameter a FLUIDOS flavor and a namespace, calls the 2 previous functions, and updates stores the new forecasted data in the flavor's data.

```
def update_local_flavor_forecasted_data(flavor: Flavor, namespace:
↪  str) -> None:
    lat = flavor.location.get("latitude")
    lon = flavor.location.get("longitude")
    new_forecast = _get_forecasted_carbon_intensity(lat, lon)
    new_forecast.insert(0,
                        _get_live_carbon_intensity(lat, lon))  #
                        ↪  index 0 = current intensity. Forecast
                        ↪  starts at index 1
    new_forecast_timeslots = []
    for i in range(len(new_forecast) - 1):
        average = (new_forecast[i] + new_forecast[i + 1]) / 2
        new_forecast_timeslots.append(average)
    optionalField = {"operational": new_forecast_timeslots}
    get_resource_finder(None, None).update_local_flavor(flavor,
    ↪  optionalField, namespace)
```

The function retrieves the location of the flavor, calls the 2 previous functions to retrieve the carbon intensity of the electricity in real time and the forecasts, then collates this data in a single table. The table is manipulated so that each item represents a timeslot of 2 hours, which corresponds to the duration with which the algorithm works. One area for improvement would be to have a system that adapts to the duration of CarbonAwareTimeslots, but it was decided at the start of the project that these timeslots would be fixed at 2 hours.

Once the data array is ready, the function calls the framework API (developed by Stefano Braghin) to update the field in the flavor data.

2. The `hourly execution process`, which is part of a new feature of the framework developed by Stefano Braghin: the observation feature that allows to perform task on each flavor at regular interval. This part has been developed by Stefano Braghin and Martin Roch-Neirey. The file used is `fluidos_model_orchestrator/daemons_and_times/flavor.py`. The full content is not shown here, but the most important part is the following:

```python
# meta represents the metadata of the flavor
# spec represents the specifications of the flavor
while True:
    if stopped:
        return
    flavor = build_flavor({
        "metadata": meta,
        "spec": spec
    })

    if namespace is None:
        namespace = "default"

    update_local_flavor_forecasted_data(flavor, namespace)

    await asyncio.sleep(CONFIGURATION.DAEMON_SLEEP_TIME) # 1 hour
```

This part is executed as a separate thread for each local flavor. One time per hour, the flavor metadata and specifications are retrieved, and passed as parameter of the forecast updater function. This done, each local flavor gets its forecast data updated one time per hour. The values can then be retrieved by local or remote components using the REAR protocol.

4.2.5 Main algorithm

The main algorithm is available in the `orchestrator.py` file. The entire file is relatively large, and only thep most important parts will be described below. The full source code is available on the GitHub repository [16].

Function and output

The `CarbonAwareOrchestrator` extends the `ModelInterface`, which means it inherits 2 functions:

- The `predict(self, data: ModelPredictRequest, architecture: str = "arm64") -> ModelPredictResponse | None` function, that is used to specify precise resources that must be used to schedule the pod.
- The `rank_resource(self, providers: list[ResourceProvider], prediction: ModelPredictResponse, request: ModelPredictRequest) -> list[ResourceProvider]` function, that is used to rank different flavors (contained in each `ResourceProvider` instance).

The `predict()` function has not been used in this project, as Stefano Braghin suggested to use the `rank_resources()` function and output a list of 1 `ResourceProvider` instance. This has been done because the `predict()` function cannot specify one precise flavor, which is the goal of the carbon-aware algorithm.

As a result, the `rank_resources()` function contains all the code of the algorithm, and the output is the following:

- A list of one `ResourceProvider` instance, that represents the flavor selected by the algorithm. This output represents the spatial aspect of the algorithm.
- The delay of execution of the selected flavor, which is specified using the `ModelPredictResponse` object given as parameter. This object is passed to the function by reference, which allows the algorithm to directly modify the value of its "delay" attribute, which corresponds to the waiting time for scheduling the pod. This output represents the temporal aspect of the algorithm.

Structure

The main algorithm is structured in 6 main parts:

1. The first part ensures that the intent file received is correct (i.e., the pod deadline is consistent, CPU and RAM demand are greater than 0). In case all conditions are not validated, the algorithm returns an empty list of `ResourceProvider`. Here is a simplified view, as an example for the CPU part:

```
deadline = np.nan
for intent in request.intents:
    match intent.name.name:
        case "cpu":
            cpuRequest = cpu_to_int(intent.value)
        [...] # other checks omitted in this example
if cpuRequest == np.nan or cpuRequest <= 0:
    logging.exception("CPU request must be provided greater than 0")
    return []
```

2. The second part creates the `CarbonAwareTimeslot` instances according to the deadline of the pod. Only required timeslots are generated.

```
timeslots = []
now = datetime.now()
start_time = now.replace(minute=0, second=0, microsecond=0)
for i in range(int(deadline)):
    slot_time = start_time + timedelta(hours=i)
    timeslot = CarbonAwareTimeslot(i, slot_time.year, slot_time.month,
    ↪  slot_time.day, slot_time.hour, 2)
    timeslots.append(timeslot)
```

   The `timeslots` list then contains all timeslots generated, and the algorithm will iterate over them.

3. The algorithm then creates all the `CarbonAwareFlavor` from the list of `ResourceProvider` given as parameter. Each `ResourceProvider` object contains one flavor, and each flavor contains its own metadata and specifications.

```
# providers is the list of ResourceProviders given as parameter to the
↪  algorithm
for provider in providers:
    flavor = provider.flavor
    flavors.append(
        CarbonAwareFlavor(
            flavor.id,
            flavor.optional_fields.get("embodied"),
            4, # 4 years of lifetime estimated
            cpu_to_int(flavor.characteristics.cpu),
            memory_to_int(flavor.characteristics.memory),
            flavor.characteristics.persistent_storage,
            flavor.optional_fields.get("operational")
        ))
```

   The `flavors` list then contains all the `CarbonAwareFlavor` generated, and the algorithm will iterate over them.

4. The fourth part is the creation of the `CarbonAwarePod` from the information retrieved in the intent file.

```
# request.id, deadline, cpuRequest, ramRequest are retrieved from the
↪  intent file
podToSchedule = CarbonAwarePod(request.id, deadline, 2, 0.03, cpuRequest,
↪  ramRequest, 0)
```

   For now, the `0.03` value represents 0.03kW of power consumption for this pod. This value is arbitrary and should be dynamic in the future, when more tools will be available to predict the power consumption of a pod based on the workload and the flavor's CPU. It has also been decided to not take into account any storage request for now.

5. The fifth part is the core of the algorithm. It iterates over each timeslot, and for each valid one, it iterates over all the flavors.

```
for ts in timeslots:
    #-------------------- New timeslot iteration --------------------
    if _is_timeslot_valid(ts, podToSchedule):
        for flavor in flavors:
            #-------------------- New node iteration --------------------
            if _check_node_resource(flavor, ts, podToSchedule):

                operationalEmissions = (flavor.forecast[
                    ts.id]) * podToSchedule.duration *
                    ↪  podToSchedule.powerConsumption  # grams, hours, kW

                embodiedEmissions = ((flavor.embodiedCarbon / (365 *
                ↪  flavor.lifetime * 24)) / (
                        1 + 1)) * podToSchedule.duration

                totalEmissions = operationalEmissions + embodiedEmissions
```

```
                if totalEmissions < minimal_emissions:
                    if best_node is not None and best_timeslot is not
                    ↪  None:
                    minimal_emissions = totalEmissions
                    best_node = flavor
                    best_timeslot = ts
```

The function `_is_timeslot_valid(ts, podToSchedule)` has already been explained in the 4.2.2 Classes subsection. When a timeslot is considered as valid, the algorithm checks for each flavor if it has enough resources to handle the pod. To do so, the function `_check_node_resource(flavor, timeslot)` generates fake resource prediction and compares them to the pod requests:

```
def _check_node_resource(flavor: CarbonAwareFlavor, timeslot:
↪  CarbonAwareTimeslot, pod: CarbonAwarePod):

    cpu_used_prediction = generate_resource_prediction(flavor.totalCpu)
    ram_used_prediction = generate_resource_prediction(flavor.totalRam)

    if (flavor.totalCpu - cpu_used_prediction) < pod.cpuRequest or (
            flavor.totalRam - ram_used_prediction) < pod.ramRequest:
        return False
    return True
```

Note that this is a temporary implementation that will be replaced by the machine learning model developed by FBK within the work package 6 of the FLUIDOS project.

If the current node (flavor) has enough resources to accomodate the pod, the algorithm computes both operational and embodied emissions as defined in the 3.2.3 Carbon Emissions Calculation subsection. The total emission is then compared to the current minimal value found, and replaces it if the value is lower. In such a case, the best node and best timeslot are also replaced.

6. The sixth and final step updates the prediction delay (a.k.a. the number of hours in which the pod will be scheduled) and returns the best flavor.

```
# prediction is an instance of ModelPredictResponse, given as a parameter
↪  to the algorithm.
prediction.delay = best_timeslot.id

for provider in providers:
    if provider.id == best_node.id:
        return [provider] # list of 1 element
return []
```

After having explained how the algorithm have been developed, the next section focuses on the demonstration of the model-based Meta-Orchestrator and the execution of the algorithm.

4.3 Demonstration of the MBMO and the carbon-aware scheduler

This section explains how to use the the model-based Meta-Orchestrator (MBMO) and how to execute the algorithm. The MBMO is explained in details in the 2.2.2 "Model-based Meta-Orchestrator" subsection and in the 2.2.3 "Simplified workflow example" subsection.

The goal of this demonstration is to explain how to interact with the model-based meta-orchestrator using an intent file. The Figure 4.5 shows the FLUIDOS cluster created using KinD:

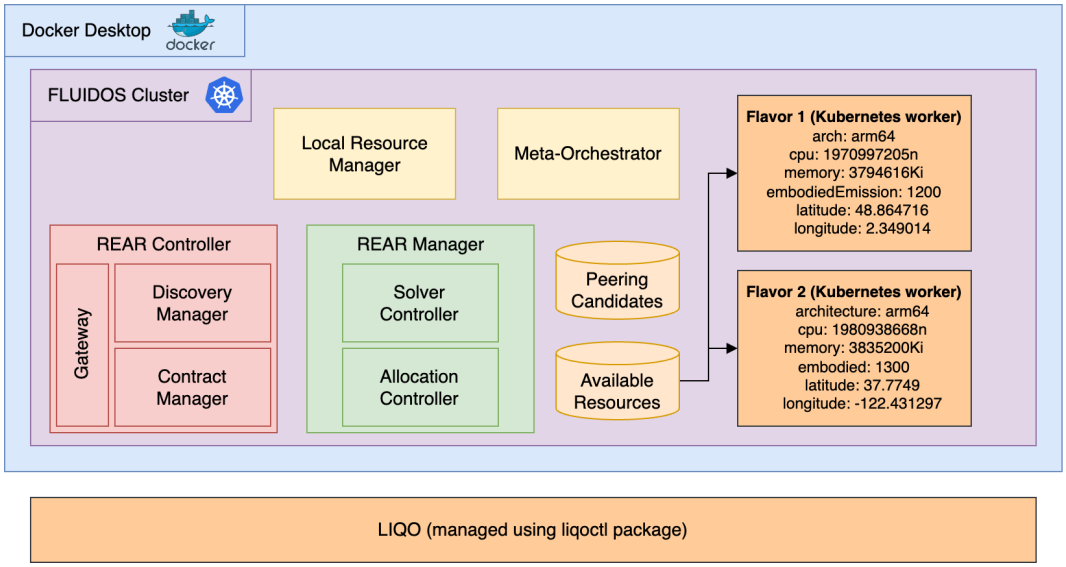

Figure 4.5: FLUIDOS cluster used for the MBMO demonstration

This cluster is made of a single FLUIDOS node with 2 flavors inside. It is totally possible to have multiple FLUIDOS nodes and remote flavors, but the testbed provided by the WP4 repository does not have such a configuration. Note that this cluster is different from the one used in the FLUIDOS remote scheduling demonstration because some components are still actively under development. The node repository and the MBMO repository does not have exactly the same components available. Note that this demonstration is based on the one available on the WP4 MBMO repository [7] and requires `liqoctl`, KinD, and a valid Electricity Maps API key with access to the countries where the flavors are located.

To execute the MBMO and the carbon-aware scheduler, the following steps are required (for an ARM-based architecture):

1. Clone the WP4 MBMO GitHub repository:

```
git clone https://github.com/fluidos-project/fluidos-modelbased-metaorche
↪  strator && cd
↪  fluidos-modelbased-metaorchestrator
```

2. Install the FLUIDOS cluster using KinD:

```
kind create cluster --name foo --config utils/cluster-multi-worker.yaml
↪  --kubeconfig utils/examples/dublin-kubeconfig.yaml
```

3. Install the required CRDs and packages:

```
kubectl apply -f utils/fluidos-deployment-crd.yaml
kubectl apply -f tests/node/crds
kubectl apply -f tests/node/crds/nodecore.fluidos.eu_flavours.yaml
kubectl apply -f utils/examples/arm-flavours-list.yaml
pip install -e .
```

4. As the carbon-aware scheduler connects to the Electricity Maps API, an API key is needed. The API key can be written in the `tests/data/example-mbmo-config-map.yaml` file, by replacing the `TEST_KEY_123!` placeholder. Once replaced, apply the ConfigMap:

```
kubectl apply -f tests/data/example-mbmo-config-map.yaml
```

   Note that this is not a production-ready secret management system, and this is only usable in a development environment. The WP4 responsibles have planned developing the system using Kubernetes Secrets in the future.

5. At this step, the FLUIDOS cluster is ready. The next step is to modify the intent file depending on the needs. The `utils/examples/carbon-intent.yaml` file contains a sample intent:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: test-deployment
  annotations:
    fluidos-intent-deadline: "12"
  labels:
    app.kubernetes.io/name: "test-deployment"
spec:
  replicas: 1
  selector:
    matchLabels:
      name: test-deployment
  template:
    metadata:
```

```
    labels:
      name: test-deployment
      app.kubernetes.io/name: "test-deployment"
  spec:
    containers:
      - name: nginx
        image: nginx:latest
        ports:
          - containerPort: 80
        resources:
          limits:
            cpu: "500m"
            memory: "256Mi"
          requests:
            cpu: "500m"
            memory: "256Mi"
```

The most important part of this file is the annotation `fluidos-intent-deadline: "12"` that informs that the pod can be delayed of a maximum of 12 hours.

6. Next step is to start the model-based Meta-Orchestrator using the following command:

```
kopf run --verbose -m fluidos_model_orchestrator
```

The command `kopf` stands for Kubernetes OPerators Framework, which is the framework used by the WP4 to develop the MBMO.

7. If no error are shown in the console, another terminal can be opened in the root folder of the project, and used to send the carbon-aware intent to the MBMO:

```
kubectl fluidos -f utils/examples/carbon-intent.yaml
```

The output should look like this:

```
kubectl_fluidos - INFO - Starting FLUIDOS kubectl extension
kubectl_fluidos - INFO - Invoking K8S with Intent Service Handler
kubectl_fluidos.modelbased - INFO - Wrapping request
kubectl_fluidos.modelbased - INFO - Converting to dictionary
kubectl_fluidos.modelbased - INFO - Sending request to k8s
```

8. On the first terminal (the one that started the MBMO), some logs should appear saying that the scheduler succeded:

```
[INFO default/test-deployment] Creation processed: 1 succeeded; 0 failed.
```

No pod will be executed, because the 2 local flavors are fake. They are visible to the MBMO but does not truly exist. This testbed is only made to test the scheduling and logic parts, but not the execution part that has already been shown in the FLUIDOS / LIQO demonstration.

4.4 Development of an algorithms simulator

One of the initial outputs of the project was a specifications document listing the possible features of an algorithm simulator to facilitate their development and improvement, before porting them to the FLUIDOS architecture.

After several days, and seeing that certain aspects of the implementation could be delayed due to other components, a small simulator was finally developed from scratch. It has then been possible to quickly test the algorithm, visualise the results, improve it and get an initial idea of its effectiveness before developing it on the FLUIDOS architecture. Note that this part of the thesis was not requested in the initial specifications. It has finally proven to be very helpful. The only thing initially requested was to specify the possible functionalities of such a simulator.

The project is only a proof of concept of what could be a more complete simulator, but it already offers the following features:

- Reproduction of the FLUIDOS architectural model, with no direct link to the project (nodes, pods, timeslots).
- Development of algorithms that use instances of these classes to solve a given problem.
- Automatic generation of algorithm input values.
- Results displayed in the form of an automatically generated plot.

The source code is not shown in this report as this is a basic Oriented-Object Programming (OOP) project without any complexity, but the source code of the simulator is available on the `SchedulingAlgorithmsSimulator` GitHub repository [19] as part of the FLUIDOS organisation. The structure of the project is the following:

```
schedulingAlgorithmsSimulator/
├── algorithms/
│   └── spatiotemporal01/
│       ├── main01.py
│       ├── main02.py
│       ├── main03.py
│       └── spatiotemporal01.py
├── models/
│   ├── node.py
│   ├── pod.py
│   └── timeslot.py
└── views/
    └── plot.py
```

The `algorithms/spatiotemporal01/` folder contains 3 executable files and one implementation of the spatiotemporal scheduling algorithm, identical to the one developed within the FLUIDOS architecture. Each executable file contains different

functions and initial data to start the algorithm. For instance, the `main01.py` file always generates the same input, as it would be done for a unit test. The `main02.py` generates random nodes, pods and forecast, while the `main03.py` is similar to the `main01.py` file.

The `models/` folder contains the classes used by the algorithms. They are similar to the one defined in the MBMO repository (CarbonAwareFlavor, CarbonAwarePod, CarbonAwareTimeslot).

The `views/plot.py` file contains numerous functions to create a visual representation of the results of the algorithm. The Figure 4.6 shows an example of the execution of the `main02.py` file that generates random inputs and calls the `spatiotemporal01` algorithm.

	TS 0	TS 1	TS 2	TS 3	TS 4	TS 5	TS 6	TS 7	TS 8	TS 9
Node 0 embodied: 1691334 gCo2	122.537	922.537	430.537	634.537	352.537	808.537	1572.537	1710.537		
Node 1 embodied: 866166 gCo2	1025.439	1329.439	107.439	131.439	337.439	335.439	135.439	1199.439		
Node 2 embodied: 1986838 gCo2	1833.404	515.404	1771.404	1747.404	2077.404			1421.404		
Node 3 embodied: 2206724 gCo2	1989.955	1123.955	965.955	1893.955	1563.955	1343.955	1243.955	1551.955		
Node 4 embodied: 1619621 gCo2	1416.444		1292.444	1898.444	1998.444	1492.444	190.444	1970.444		
Node 5 embodied: 2434653 gCo2	400.964	1242.964	502.964	1030.964		1306.964	1306.964	408.964		
Node 6 embodied: 1005455 gCo2	1811.389	331.389	1937.389	1793.389	339.389	1621.389	671.389	307.389		
Node 7 embodied: 2037692 gCo2	492.307	1480.307	534.307	1844.307	1452.307	842.307	432.307	336.307		
Node 8 embodied: 648671 gCo2	1883.025	1243.025	1957.025		561.025	1977.025	1257.025	1891.025		
Node 9 embodied: 906261 gCo2	1331.727	1119.727		1293.727	315.727		905.727			

Figure 4.6: Output of the algorithm simulator. Values in cells are in gCO_2

The nodes are listed on the left with their embodied emissions, and the timeslots are on the top. Here are the useful information to read the plot:

- The cell circled in blue represents the result. It corresponds to the tuple `[node;timeslot]` which is the output of the algorithm. This is where and when the algorithm decided to schedule the pod. It should be the minimum value between all cells (minimal CO_2 emissions).
- The value inside each cell represents the amount of CO2 emitted if the pod is scheduled here.
- The color of each cell is a simple way to view the best and worst cells. The coloration scheme is linear.

- Some cells are totally white, meaning the timeslot is not valid for the given pod or the node does not have enough resources left to accommodate the pod on the given timeslot.

For example, the white cells alone indicates that the node does not have enough resources left to accommodate the pod. The 2 last columns are also white because the timeslots are not valid (they start too late compared to the pod's deadline).

To obtain this result, the `main02.py` file generated fake pods, nodes, timeslots and carbon intensity forecast for each node, and then called the `spatiotemporal01` algorithm. The results retrieved are then given as parameters to the `showPlot()` function of the `plot.py` file that generates the image, table, fills the cells, colors them, highlights the chosen cell, and shows the final image produced. Note that the input data are randomly generated between real life data. The simulator does not use the Electricity Maps API to enable more people to use it without the need of an API key, and to leave this API key for production-grade execution.

The plot obtained by the simulator can give first elements of comparison between the carbon-aware scheduler and other schedulers (vanilla Kubernetes, latency-aware, cost-effective...). The algorithm ensures to take the lowest value in the available cells, ensuring that this is the most carbon-aware scheduler of all schedulers. With more time, more precise comparison could have been given.

The algorithm simulator may be improved easily by adding the following features:

- Ability to run a particular algorithm a certain times and get a summary of the results.
- Comparison of several algorithms on the basis of different criteria (CO2 savings, execution time, complexity, etc.).

This first version of the algorithm simulator was very useful throughout the thesis, for experimenting and validating modifications to the algorithm design. Discussions took place in the final days of the project with the FBK team to consider the development of a more complete simulator. Discussions will continue in September.

4.5 Other work done

In addition to the implementation of the algorithm within the FLUIDOS architecture and the simulator, some other tasks were carried out throughout the project. They do not constitute the main tasks of the project but are nevertheless documented below:

- Discussion and review on expanding the FLUIDOS MBMO API (ability to modify local flavors from a scheduler, add hourly caller function...).
- Creation of unit tests dealing with the carbon-aware scheduler.
- Follow-up of bi-weekly FLUIDOS meetings related to work packages 6 (the one concerned by this thesis), 3 (concerning the development of FLUIDOS nodes) and 4 (concerning the MBMO).
- Follow-up of other FLUIDOS meeting related to important topics on the project (major release, year review...).
- Active communication with the required work packages (2, 3, 4, 5) concerning the overall progress of the project and synchronisation of the various elements and work groups.

5 Conclusion

This chapter concludes the overall project, providing final results, conclusion, perspectives and a personal conclusion.

5.1 Global overview

The main objectives of the project were achieved. The FLUIDOS architecture has been explained and 2 main sides of the project have been demonstrated: the capability of FLUIDOS to schedule pods on remote nodes using the LIQO stack, and the way the model-based Meta-Orchestrator is executed and schedules pods across the cluster. Besides, the FLUIDOS API has been documented and explained in this thesis, using the WP4's framework which represents the execution environment of the algorithm developed. Finally, a spatiotemporal shifting algorithm has been implemented within the model-based Meta-Orchestrator, which is the scheduler component of the FLUIDOS architecture. This spatiotemporal algorithm schedules pods across geographically distributed flavors and time slots while minimising the carbon emissions.

It is important to note that this scheduler does not take into account all the CO2 emissions of the scheduling process. For example, this algorithm ignores the energy required to transfer persistent volumes from one node to another, or to transfer the image of the container to be executed. In addition, certain assumptions were made about different aspects, such as the fact that a pod's power consumption is constant, and that the pod lasts exactly 2 hours.

The secondary objectives were partially achieved, in particular the comparison of different scheduling algorithms. Several meetings were held with various work packages and players (IBM, FBK) to discuss latency-aware and cost-effective algorithms, but the analysis was not taken any further for lack of time.

In addition, the proof of concept of an algorithm simulator has been developed to enable different algorithms to be tested upstream without having to develop them on the FLUIDOS architecture. This point was not requested at the outset of the project and represents an added value that enables scheduling algorithms to be better prepared for deployment in complex environments such as the FLUIDOS one.

Certain difficulties were encountered during the project, particularly during the implementation phase. The FLUIDOS project brings together 16 different actors divided into 10 work packages, and WP6 is technically very dependent on WP4 (Intent-based decentralised FLUIDOS continuum). Numerous bugs have been found and reported to the framework developers, but this has led to delays in implementing the algorithm within the framework. This accumulated delay has not been in vain and has permitted to improve the framework, extend the APIs available to offer new functionalities and, more generally, move the FLUIDOS project forward.

5.2 Next steps

The FLUIDOS project was officially launched in September 2022, and is due to run until August 2025. With a duration of 3 years, 2 of which have already almost elapsed, here are the possible next steps to be taken within work package 6, in connection with this bachelor's thesis:

- When the new version of the FLUIDOS node and REAR-data-models are released, the framework's code and API will need to be adapted to interact directly with the corresponding data.
- Redefine the interfaces and objective of the machine learning model developed to predict node usage at a given time. In particular, the model needs to predict CPU and RAM usage.
- Extend the principle of 2-hour timeslots to a more variable duration, and also take into account the possibility that a pod may have a longer execution time than the duration of a timeslot. In such a case, we need to consider finding the best sliding average over the sets of consecutive timeslots that can execute the pod.
- Carry out complete benchmarks of the algorithm on various criteria (execution time, variation in input parameters, comparison with other algorithms, etc.). Two meetings have been held with the FBK team (one of the actor of the project) to start thinking about a tool for in-depth benchmarking of the algorithm, and a third is planned for September.
- Study the CO_2 savings achieved by the carbon-aware algorithm compared with other existing scheduling algorithms (vanilla Kubernetes scheduling, other MBMO schedulers, etc.).
- Extend the logic of the algorithm to more complex cases where the duration of the pod and its power consumption over time are variable.
- Extend the logic of the algorithm to include "re-scheduling", which consists of moving pods already running at location A to location B in order to emit less CO2. This logic must take into account any potential downtime by seeking to minimise it and also the energy cost required to transfer the pod.

5.3 Personal feedback

From a personal point of view, I'm very happy to have had a European research project as the topic of my Bachelor's thesis. I'm even happier about the fact that this project took place outside Switzerland, in a country I didn't know. I discovered a city, a university, an institute, and researchers who are passionate about their topics and dedicated to their work.

More generally, the FLUIDOS project enabled me to meet many people from different backgrounds, and made me realize the complexity and importance of communication and organisation in projects of this scale. This thesis has put me right at the heart of a project involving over 90 people scattered across Europe, and I've really enjoyed thinking, discussing and confronting ideas with my colleagues from WP6 and other work packages.

On the technical side, I also really enjoyed discovering the FLUIDOS architecture, testing it, pointing out bugs, reiterating, and finally making my own contribution by developing the algorithm within it. I also really enjoyed developing my own little simulator, which enabled us to save time thinking about and studying the algorithm's behavior, quickly, accurately and visually.

I'd like to thank everyone who contributed in any way to my Bachelor's project, and I'll treasure my memories of the experience.

5.4 HES-SO legal information

Declaration of honour I, the undersigned, Martin Roch-Neirey, declare on my honour that the work submitted is my own work. I certify that I have not resorted to plagiarism or any other form of fraud. All sources of information used and author citations have been clearly stated.

Martin Roch-Neirey

A | Actors and work packages of the FLUIDOS project

Number	Short name	Legal name	Country
1	MAR	MARTEL INNOVATE BV	NL
2	UMU	UNIVERSIDAD DE MURCIA	ES
3	FBK	FONDAZIONE BRUNO KESSLER	IT
4	POLITO	POLITECNICO DI TORINO	IT
5	RSE	RICERCA SUL SISTEMA ENERGETICO - RSE SPA	IT
6	ROB	ROBOTNIK AUTOMATION SLL	ES
7	DSME	EUROPEAN DIGITAL SME ALLIANCE	BE
8	IBM	IBM IRELAND LIMITED	IE
9	TID	TELEFONICA INVESTIGACION Y DESARROLLO SA	ES
10	TOPIX	CONSORZIO TOP-IX - TORINO E PIEMONTE ECHANGE POINT	IT
11	BOR	BORDERSTEP INSTITUT FUR INNOVATION UND NACHHALTIGKEIT GEMEINNUETZIGE GMBH	DE
12	HMU	ELLINIKO MESOGEIAKO PANEPISTIMIO	EL
13	STM	STMICROELECTRONICS GRENOBLE 2 SAS	FR
14	TUB	TECHNISCHE UNIVERSITAT BERLIN	DE
15	CYSEC	CYSEC SA	CH
16	TER	Terraview GmbH	CH

Table A.1: List of actors

Number	Name	Lead beneficiary
WP1	FLUIDOS Governance	1 - MAR
WP2	FLUIDOS reference architecture	4 - POLITO
WP3	Modular and extensible FLUIDOS node	10 - TOPIX
WP4	Intent-based decentralised FLUIDOS continuum	8 - IBM
WP5	Seamess, zero-trust security and privacy	3 - FBK
WP6	Cost-effective and energy-aware infrastructure	14 - TUB
WP7	Business use cases and market validation	16 - TER
WP8	Community building and Open Calls	7 - DSME
WP9	Agile integration & testing	2 - UMU
WP10	Project Coordination	1 - MAR

Table A.2: List of work packages

B | Draft of a research paper

A draft of a paper regarding the implementation of the algorithm and the whole project is accessible on the next page.

This draft only contains the structure of the paper. Due to a lack of time, Vlad Coroamă, Nasir Asadov and Martin Roch-Neirey decided to write this paper in September.

Implementation and benchmarking of a carbon-aware scheduling algorithm in a distributed computing continuum

NASIR ASADOV, Chair of Sustainable Engineering (SEE), Germany
MARTIN ROCH-NEIREY, Institute of artificial intelligence and complex systems (iCoSys), Switzerland
VLAD COROAMA, Chair of Sustainable Engineering (SEE), Germany

abstract

Additional Key Words and Phrases: a, b, c

ACM Reference Format:
Nasir Asadov, Martin Roch-Neirey, and Vlad Coroama. 2024. Implementation and benchmarking of a carbon-aware scheduling algorithm in a distributed computing continuum. X, X, Article X (X 2024), 1 page. https://doi.org/XXXXXXX.XXXXXXX

Received X June 2024; revised X June 2024; accepted X June 2024

Authors' Contact Information: Nasir Asadov, nasir.asadov@tu-berlin.de, Chair of Sustainable Engineering (SEE), Berlin, Germany; Martin Roch-Neirey, Institute of artificial intelligence and complex systems (iCoSys), Fribourg, Switzerland, martin.roch-neirey@hefr.ch; Vlad Coroama, coroama@tu-berlin.de, Chair of Sustainable Engineering (SEE), Berlin, Germany.

ACM XXXX-XXXX/2024/X-ARTX
https://doi.org/XXXXXXX.XXXXXXX

References

[1] *Flexible, scaLable and secUre decentralIzeD Operating System | FLUIDOS Project | Fact Sheet | HORIZON*. en. URL: https://cordis.europa.eu/project/id/101070473 (visited on 06/11/2024).

[2] *FLUIDOS project*. en-US. Sept. 2021. URL: https://www.fluidos.eu/ (visited on 06/11/2024).

[3] *Kubernetes*. en. URL: https://kubernetes.io/ (visited on 06/11/2024).

[4] *liqotech/liqo: Enable dynamic and seamless Kubernetes multi-cluster topologies*. URL: https://github.com/liqotech/liqo?tab=readme-ov-file (visited on 06/06/2024).

[5] Sébastien Rumley Nasir Asadov Vlad Coroamă. *Context of the bachelor thesis*. 2024.

[6] FLUIDOS contributors. *REAR protocol*. 2024. URL: https://github.com/fluidos-project/REAR (visited on 07/25/2024).

[7] FLUIDOS contributors. *FLUIDOS model-based meta-orchestrator GitHub repository*. 2024. URL: https://github.com/fluidos-project/fluidos-modelbased-metaorchestrator (visited on 07/25/2024).

[8] IEA. *Electricity 2024: Analysis and forecast to 2026.* Tech. rep. IEA, 2024.

[9] Ralph Hintemann and Simon Hinterholzer. "Energy consumption of data centers worldwide". en. In: 2019.

[10] Moore, G. E. "Cramming more components onto integrated circuits." In: Electronics, 38(8). (1965).

[11] Ana Radovanovic et al. "Carbon-Aware Computing for Datacenters". en. In: (June 2021). arXiv:2106.11750 [cs, eess]. URL: http://arxiv.org/abs/2106.11750 (visited on 07/27/2024).

[12] Poonam Singh, Maitreyee Dutta, and Naveen Aggarwal. "Bi-objective HWDO Algorithm for Optimizing Makespan and Reliability of Workflow Scheduling in Cloud Systems". en. In: *2017 14th IEEE India Council International Conference (INDICON)*. Roorkee: IEEE, Dec. 2017, pp. 1–9. ISBN: 978-1-5386-4318-1. DOI: 10.1109/INDICON.2017.8487600. URL: https://ieeexplore.ieee.org/document/8487600/ (visited on 07/17/2024).

[13] Xiangqiang Gao, Rongke Liu, and Aryan Kaushik. "Hierarchical Multi-Agent Optimization for Resource Allocation in Cloud Computing". en. In: *IEEE Transactions on Parallel and Distributed Systems* 32.3 (Mar. 2021), pp. 692–707. ISSN: 1045-9219, 1558-2183, 2161-9883. DOI: 10.1109/TPDS.2020.3030920. URL: https://ieeexplore.ieee.org/document/9224163/ (visited on 07/16/2024).

[14] WP3 authors. *FLUIDOS Node - Testbed (KIND)*. 2024. URL: https://github.com/fluidos-project/node/tree/main/testbed/kind (visited on 07/11/2024).

[15] The Kubernetes Authors. *kind*. 2024. URL: https://kind.sigs.k8s.io/ (visited on 07/11/2024).

[16] Martin Roch-Neirey and Stefano Braghin. *Carbon-Aware scheduler 41 Pull Request - FLUIDOS Model-Based Meta-Orchestrator GitHub repository*. 2024. URL: https://github.com/fluidos-project/fluidos-modelbased-metaorchestrator/pull/41 (visited on 08/01/2024).

[17] Electricity Maps ApS. *Electricity Maps*. 2024. URL: https://www.electricitymaps.com/ (visited on 07/29/2024).

[18] FLUIDOS contributors. *FLUIDOS REAR data models*. 2024. URL: https://github.com/fluidos-project/REAR-data-models (visited on 07/29/2024).

[19] Martin Roch-Neirey. *FLUIDOS Scheduling Algorithms Simulator GitHub Repositoy*. 2024. URL: https://github.com/fluidos-project/scheduling-algorithms-simulator (visited on 08/02/2024).

Glossary

API An Application Programming Interface is a set of protocols, tools, and definitions that allow different software applications to communicate with each other, enabling them to request and exchange data and services. 3

Internet-of-Things Network of interconnected physical devices embedded with sensors, software, and other technologies to collect and exchange data, enabling them to communicate and interact with each other and external systems over the internet. 1

KinD Kubernetes in Docker is a tool for running local Kubernetes clusters using Docker container nodes, primarily designed for testing Kubernetes environments. 14

Kubernetes Open-source orchestrator used to deploy scalable workloads as containers. 1

LIQO Open-source platform enabling dynamic and seamless multi-cluster Kubernetes orchestration and resource sharing across different environments. 1

Oriented-Object Programming Programming paradigm that organises software design around data, or objects, rather than functions and logic, emphasising the concepts of classes and objects, inheritance, encapsulation, and polymorphism. 41