POLITECNICO DI TORINO

Master's Degree in Computer Engineering

Master's Degree Thesis

Scheduling Microservice-Based Applications Across Distributed Edge Clusters

Supervisors

Prof. Fulvio RISSO

Prof. Giacomo VERTICALE

Dott. Stefano GALANTINO

Candidates

Gabriele FILAFERRO

Luca NICOSIA

September 2021

Summary

Cloud Computing and Edge Computing are becoming the standard for the development and deployment of software applications. Scheduling frameworks for resource management of an infrastructure of nodes play a key role in supporting these approaches.

This thesis describes PHARE, a centralized gang scheduler that can handle microservice applications taking into account both the resource and cost requirements of the single components and those derived by the communication between multiple components.

Testing and validation demonstrated an average of 10 ms scheduling time, per application, on infrastructures composed of 100 nodes. Increasing the number of nodes to 1000 the scheduler kept the average scheduling time under 100 ms. To provide more value to the result a performance comparison with the state of the art Firmament scheduler was conducted. The measuring showed a performance improvement in scheduling time and cost management.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Cloud Computing is becoming the standard for the development of software applications and each day old monolithic software is being rewritten following the cloud-native architecture. The cloud computing approach is based on the paradigm of microservices, where the tasks of an application are splitted into small dedicated software components. This approach leads to better maintainability, ease of scale and higher fault-tolerance. From a low-level implementation perspective, each component of the application becomes a software container that will interact with other services using a internet-based protocol.

The growth of cloud computing is also related to the decrease in cost and the increase in accessibility to computing resources, thanks to the rise of cloud computing providers, such as Google Cloud and Amazon Web Services. These providers remove the need from software companies to build and maintain a dedicated infrastructure to run their set of services, offering a great range of solutions that suit everyone's needs, going from a bare-metal offer, such as hosting a simple virtual machine in their own cloud, to provision a fully managed software-as-a-service application.

Cloud providers also need to maintain a big, and always increasing, infrastructure to satisfy the needs of their costumers. A single application to deploy for a customer, becomes one of thousands from the provider's perspective. The providers needs to carefully manage their deployments to avoid wasting resources, leading to cost increase and excessive power consumption. Even software companies, who exploit resources offered by providers, could need to deploy big and complex applications, composed of an intricate set of microservices. This could be done to satisfy multiple users across the globe, or to provide high level of performances and low down times.

Given the described scenarios, scheduling a containerized applications over a computing infrastructure in the right or wrong way could highly impact the

performances and cost of running a service. The scheduling part of managing software needs to be performed in an automated way, this led to the growth of research towards fast and optimized scheduling paradigms.

1.1 The Goal of the Thesis

The Computer Networks Group at Politecnico di Torino is working hard on the state of the art of cloud computing technologies, focusing mostly on the Kubernetes orchestrator. This led to the birth of the Liqo project [1], an extension of Kubernetes [2] to allow the seamless interaction between multiple clusters of nodes.

This new type of approach to cloud computing highlighted an issue regarding the scheduling of resources in this new scenario. Since in a Kubernetes cluster, all the nodes of the cluster are usually on the same datacenter the scheduler doesn't need to take into account the communications between containers in an application, because they will remain internal to the datacenter and hence will probably have no usage limitations or constraints. When instead scheduling the containers across multiple clusters the communication factor needs to be taken into account, given that datacenters could be placed in various corners of the world and the link between them could become a physical limitation to the reliability and the performance.

The current state of the art doesn't provide feasible solutions for this problem, the Liqo team developed a scheduling algorithm to satisfy the communication constraints in the new described scenario. This thesis describes the work done to:

- **Validate** the algorithm using various test cases.
- **Extend** the algorithm to support more complex scenarios.
- **Create** a working implementation.

The structure of the thesis is the following:

- **Introduction**. This chapter provides a brief introduction on the need for scheduling technologies in cloud computing and the reasons behind the decision to develop this thesis.
- **State of the Art**. This chapter illustrates the current state of the art on scheduling technologies and their limitations, focusing on the Kubernetes orchestrator and the Firmament scheduler.
- **PHARE Scheduler Architecture**. This chapter describes the conceptual architecture of the scheduler, highlighting the algorithmic core and its fundamental design choices.

- **PHARE Scheduler Implementation**. This chapter describes the implementation of the scheduler, analysing the various parts composing the scheduler.
- **Validation and Testing**. This chapter evaluates the performance of the scheduler in various test cases, also comparing the performances to the Firmament scheduler.
- **Conclusion and Future Works**. This final chapter provides an analysis on the current state of the scheduler and its results. The chapter also focuses its advantages and disadvantages, exploring the possible improvements and future work.

Chapter 2

State of the art

2.1 Virtual Machines and Containers

At the base of Cloud Computing are the concepts of Virtual Machines and Containers which are the most used approaches when developing and deploying applications on a computing infrastructure.

Virtual machines (VMs) are a technology for building virtualized computing environments, they are considered the first generation of cloud computing. A virtual machine is an emulation of a physical computer. VMs can run what appears to be multiple machines, with multiple operating systems, on a single computer. VMs interact with physical computers by using software layers called hypervisors. Hypervisors can isolate VMs from one another and allocate processors, memory, and storage among them. In traditional virtualization, a hypervisor virtualizes physical hardware. The result is that each virtual machine contains a guest OS, a virtual copy of the hardware that the OS requires to run, an application and its associated libraries and dependencies. VMs with different operating systems can be run on the same physical server.

A container is a standard unit of software that packages up code and all its dependencies so that an application can run quickly and reliably from one computing environment to another. A container image is a lightweight, standalone, executable package of software that includes everything needed to run an application: code, runtime, system tools, system libraries and settings. Container images become containers at runtime and these containers will always run the same, regardless of the infrastructure underneath. To summarize, containers main features are:

- Standard: containers are portable anywhere even on different operating systems.

- Lightweight: containers share the machine's OS system kernel and therefore do not require an OS per application, driving higher server efficiencies and reducing server and licensing costs.

- Secure: applications are safer in containers and provides the strongest default isolation capabilities in the industry.

Containers enable microservice architectures, where application components can be deployed and scaled more granularly. This is an attractive alternative to having to scale up an entire monolithic application, hosted on a Virtual Machine, in case a single component is struggling with load.

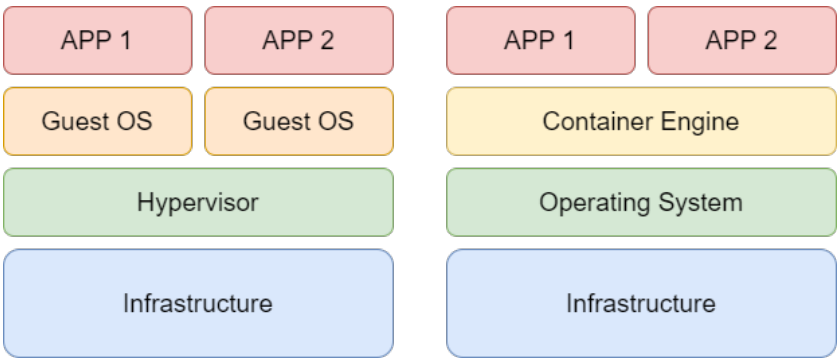

Figure 2.1: Container and Virtual Machine Architectures

2.2 Scheduling approaches

When looking at the scheduling of microservices-based application on an infrastructure, two main approaches have been studied and are currently used: sequential scheduling and gang scheduling. The two approaches differ in the perspective used over the application to be scheduled when making decision.

Sequential scheduling implies that when finding a placement solution for an application, the scheduler considers its components one at a time. For each component the scheduler takes into account the necessary requirements of the component and the current state of the infrastructure, looks for the best solution and assigns the microservice to a specific node, the process starts again for the next microservices until all the application is scheduled.

Gang scheduling instead has a wider perspective on the scheduling process. This type of scheduler considers the components of an application and how they interact with each other. The scheduler needs to fine a suitable solution that satisfies the

requirements of all the components of an application at the same time and then assigns all of them to the corresponding node of the infrastructure.

Both styles have trade-offs. Using a sequential approach, usually leads to a faster and easier implementation with respect to a gang one. Although sequential scheduling could find sub-optimal or incomplete solutions by not considering the interaction between components. Gang scheduling, on the other hand, requires more effort because the scheduling algorithms used are usually more articulated but the placement decision should be more accurate and optimized.

Looking at the current state of the art surrounding the scheduling of applications on resources in the cloud and on the edge, two interesting projects, one for each mentioned approach, rise above the rest: the Kubernetes Scheduler and the Firmament scheduler.

2.3 Kubernetes Scheduler

2.3.1 Kubernetes Architecture

Kubernetes is the standard de-facto system for automating deployment, scaling, and management of containerized applications in the cloud. Kubernetes abstracts all the concepts of the microservices world into a declarative API. A Kubernetes cluster consists of a set of worker machines, called nodes, that run containerized applications. Every cluster has at least one worker node. At the core of the Kubernetes workflows are controllers, a controller is a process that looks at the current state of the cluster and checks whether the desired state, defined through the API matches it; in case the two states are not equal, the controller will change the necessary components in the cluster to remove differences. This approach proved to be very reliable and fault-tolerant. The state of a cluster is represented using many *kubernetes resources*, a declarative data structure. The containers that compose an application are represented using the Pod resource (*fig. 2.2*). The Pod API provides all the necessary information needed to run an application, the most relevant are the following:

- the container images, e.g. Docker images, used by the containers inside the pod;
- the number of replicas of the same pod, to allow for load-balancing workloads;
- the requested resources needed to run the containers properly, e.g. CPU and RAM usage;

- the upper-bound limit for the containers' resource consumption, in case the application encounters errors, to avoid cluster resource starving;
- policies on the node placement to allow specific scheduling behaviours.

The nodes of the cluster are represented with the Node resources. This includes the information about:

- the state of the node, whether is reachable and/or is available for the placement of pods
- the current usage of the resources of the node
- policies to allow for specific scheduling decisions

To run an application inside the cluster the control plane of the framework needs to place each pod onto a node, i.e. the role of the scheduler, taking into account all the data present in the resources.

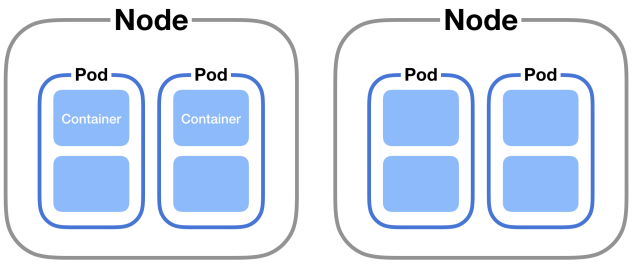

Figure 2.2: The internal representation of microservices in Kubernetes.

2.3.2 Scheduling Model

Among the controllers of Kubernetes, the one responsible for choosing where to place a pod on a node is the *kube-scheduler*. For every newly created pod or other unscheduled pods, the *kube-scheduler* selects an optimal node for them to run on. However, every container inside a pod has different requirements for resources, therefore, existing nodes need to be filtered according to the specific scheduling requirements.

The selection of a node for a pod is divided in 2 steps:

- Filtering: the filtering step finds the set of nodes where it's feasible to schedule the pod. For example, a filter can check whether a candidate node has enough

available resources to meet a pod's specific resource request, or can check whether a candidate node has a particular hardware component required by the pod. After this step, the node list contains any suitable Nodes. If the list is empty, that Pod isn't (yet) schedulable.

- Scoring: the scheduler ranks the remaining nodes to choose the most suitable pod placement. The scheduler assigns a score to each node that survived filtering, basing this score on the active scoring rules.

Finally, the *kube-scheduler* assigns the Pod to the Node with the highest ranking. If there is more than one node with equal scores, kube-scheduler selects one of these at random. The assignment phase is known as "Binding phase".

2.3.3 Limitations

The scheduling framework underneath the kube-scheduler is built following a pipeline approach *fig. 2.3*, where each step of the scheduling process can be customized, this provides flexibility to the user but doesn't allow the scheduler to fully overcome some limitations.

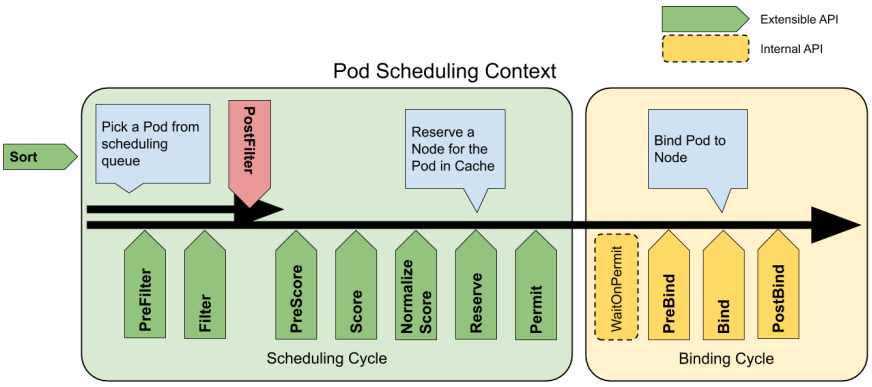

Figure 2.3: The pipeline architecture of the Kubernetes scheduler.

Given an application composed by a set of communicating pods, the limitations of this type of scheduler arise. Since it selects a node for a pod one at a time, it cannot have a full picture on how the complete application will be placed in the infrastructure, this presents two issues:

- a choice for a certain pod could lead to a state where a later pod cannot be placed. The situation could be avoided by considering the application as a whole;

- the effect of the communication between the pods, (e.g. the cost related to the bandwidth used between two pods) is not taken into account, which could lead to an imperfect solution.

Another limitation of the kubernetes scheduler is that it is designed with a datacenter scenario in mind. This makes it not suitable for placing jobs in a edge-network context.

2.4 Firmament Scheduler

2.4.1 Scheduling Model

Firmament [3] is a gang scheduling algorithm created at Google. It was created to perform quick scheduling at scale. It creates an abstract representation of the pair application-infrastructure as a flow network *fig. 2.4*. The flow network is a directed graph whose structure is defined by the scheduling policy, the arcs carry flow from source nodes (i.e. pod nodes) to a sink node. A cost and capacity associated with each arc constrain the flow, and specify preferential routes for it. The algorithm models the scheduling problem as a min-cost max-flow (MCMF) optimization over the flow network. In response to events and monitoring information, the flow network is modified according to the scheduling policy, and submitted to an MCMF solver to find an optimal (i.e., min-cost) flow. Once the solver completes, it returns the optimal flow, from which Firmament extracts the implied task placements. The scheduling framework provides three policies:

- Load-spreading: The effect is that the number of tasks on a machine only increases once all other machines have at least as many tasks. This policy neither requires nor uses the full sophistication of flow-based scheduling.

- Quincy: This policy is suitable for batch jobs, and optimizes for a trade-off between data locality, task wait time and preemption cost.

- Network-aware: it avoids overloading machines' network bandwidth, which can degrade task response time. This policy is used only to illustrate Firmament's potential to make high-quality decisions, but is not ready for a production scenario.

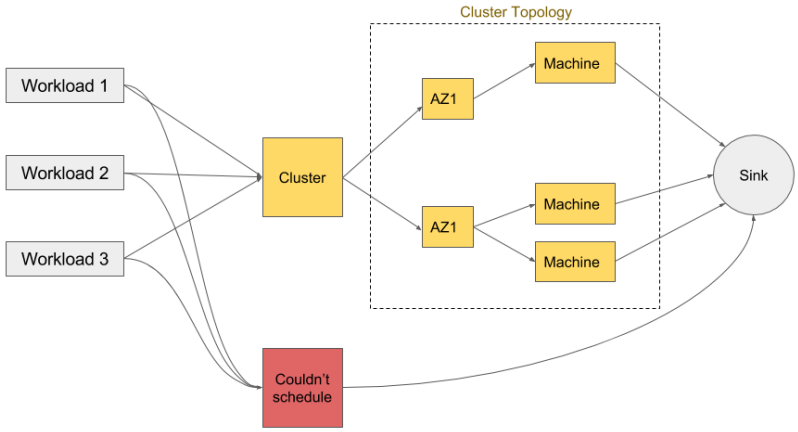

Figure 2.4: The flow network representation used by the Firmament scheduler.

2.4.2 Limitations

Even though Firmament is a great algorithm for scheduling inside datacenters, where bandwidth requirements are not concerning, it doesn't fully satisfy gang-scheduling applications taking into account the communication between pods. The network-aware policy is not ready and would take incomplete decision in a production context, leading to an overloading of the network infrastructure or a sub-optimal decision derived by assigning the wrong priority to the pod resource requests.

2.4.3 Kubernetes Implementation

The Firmament scheduler can be used inside a Kubernetes cluster, thanks to the Poseidon/Firmament project [4], which is in early stages of development. At a very high level, the Poseidon/Firmament scheduler augments the current Kubernetes scheduling capabilities by incorporating new novel flow network graph based scheduling capabilities alongside the default Kubernetes scheduler. Due to the inherent rescheduling capabilities, the new scheduler enables a globally optimal scheduling for a given policy that keeps on refining the dynamic placements of the workload.

This integration can be achieved thanks to the multiple-scheduler architecture of Kubernetes. Normally each new pod is scheduled by the default scheduler, but Kubernetes can also be instructed to use another scheduler by specifying the name of another custom scheduler at the time of pod deployment. In this case, the

default scheduler will ignore that Pod and instead allow Poseidon scheduler to schedule the Pod on a relevant node.

Firmament was not built to work inside Kubernetes, this exposes some issues that must be addressed in order to be able to deploy it inside a cluster:

- Kubernetes provides a rich API consisting of jobs, replica sets, deployments, whereas Firmament's API consists of tasks that are grouped into jobs. Code that bridges between Kubernetes' and Firmament's API is required.
- Firmament is implemented in C++. Therefore, a Kubernetes C++ client is required or code that uses the Go client and that communicates to Firmament is required.
- Firmament requires utilization statistics of the Kubernetes cluster when placing pods.

The following design choices needed to be made to solve the issues. Poseidon is implemented in Go and acts as a bridge between the Kubernetes cluster and Firmament. fig. 2.5 represents an overview of Firmament's Kubernetes integration and where Poseidon fits. Poseidon watches for updates from the Kubernetes cluster, transforms pods information to Firmament compatible concepts (i.e., tasks & jobs), receives scheduling decisions and informs the Kubernetes server API of pod bindings. Moreover, Poseidon provides a gRPC service which receives utilization statistics from Heapster. These statistics are transformed from per pod to per task stats and forwarded to the Firmament scheduler which stores the last N samples.

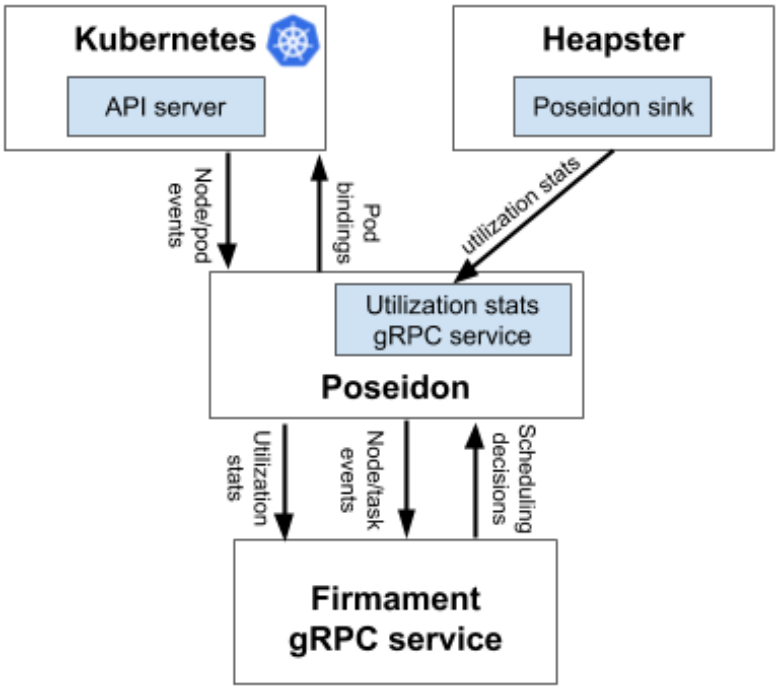

Figure 2.5: The use of Firmament in Kubernetes.

Chapter 3

PHARE Scheduler Architecture

3.1 Introduction

All schedulers presented in the section before have some limitations: none of them take into account the infrastructure communication requirements. Moreover Kubernetes scheduler is only driven by resource demands and nodes' available resources without taking into account the cost of the scheduling, Firmament instead accounts nodes' resources costs, but not bandwidth ones.

Ignoring communication requirements during the scheduling of an application[1] can produce a scenario where each component[2] is working flawlessly, but they are slower than expected because links used to communicate are overloaded. Looking at fig. 3.1, supposing that the *red* links require high bandwidth, the *green* link will not be able to handle all the packets at maximum speed, slowing down the communication and, as consequence, slowing down the application. The configuration at fig. 3.2 distributes application nodes on three infrastructure nodes rather than two, reducing the stress on the *green* link and also speeding up the *orange* application aggregating all its components inside a single node. The "Network-aware" policy of Firmament schedules accounting links requirements, but it cannot be used together with another policy that takes into account the resources of nodes. All tests performed in section 5.1 are performed using this policy.

[1] application refers to a set of pods that interact together

[2] an application component is one of the microservices that compose an application

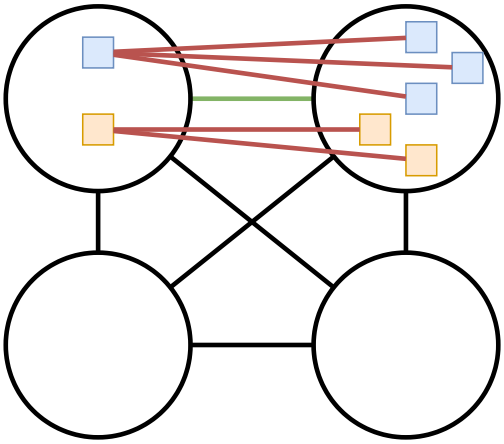

Figure 3.1: Extreme link pressure

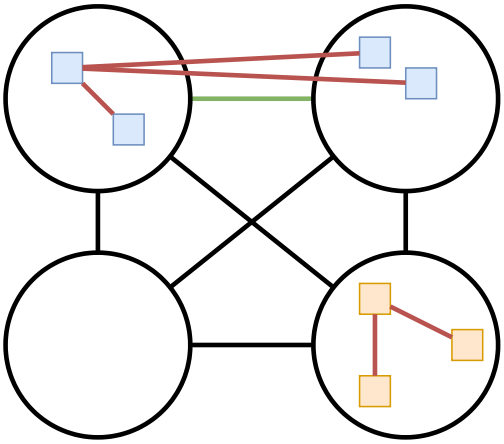

Figure 3.2: Reduced link pressure

Supposing to have a set of nodes that can host an application component, the Kubernetes scheduler decides following a scoring system which by default is not related to the cost of scheduling on each infrastructure. Both Firmament and PHARE scheduling decisions are also driven by the cost.

PHARE is a multi-cluster Kubernetes scheduler designed to reduce the cost of applications inside a cluster. The main advantage of scheduling an application rather than a pod is that the scheduler knows how each pod is connected to others and it will schedule a pod taking into account not only its requirements but also the connections with other pods. From now on, pods and clusters will be also referred respectively as "application components" and "infrastructure nodes".

3.2 Scheduler System Model

The high-level representation of the scheduler is shown in fig. 3.3. The PHARE scheduler takes as main inputs some configuration parameters needed to customize the scheduling decisions, and the application and infrastructure specifications. This information is used to compute the data structures used during the scheduling phase. Finally the scheduling algorithm is called recursively until all the components of an application are scheduled, producing the solution.

The next part describes the main steps in the overall scheduling process.

First of all, when an application is loaded into the scheduler, its components are sorted based on their "difficulty" to be scheduled. The "difficulty" of a component to be scheduled is related to its resource requirements, network requirements and, eventually, scheduling constraints. Sorting from the harder to the easier component

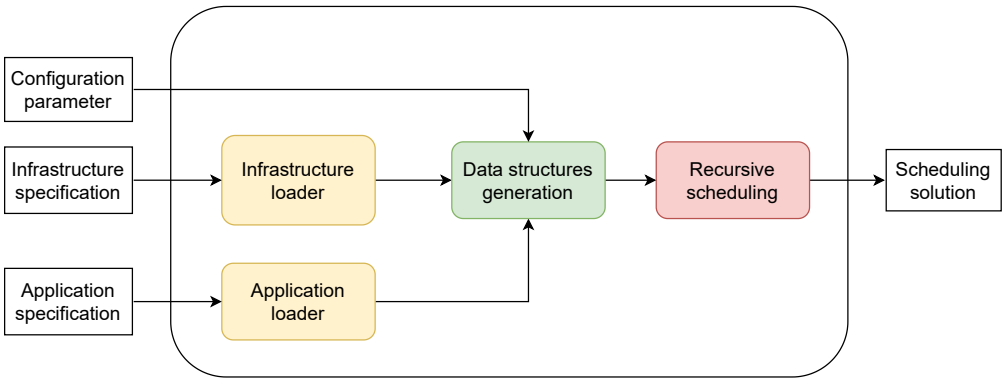

Figure 3.3: PHARE system model

was decided to increase the ratio of schedulable applications.

The next step is to compute the affinity matrix with an entry for each pair <component,node> and a related value (the affinity value) which is calculated over the amount of resources required by the component and its neighbors[3]. This matrix is used by the PHARE scheduler to retrieve which infrastructure nodes are more likely to be the scheduling target for each evaluated component.

The next phase is the calculation of the cost matrix: this matrix, as well as the affinity matrix, has an entry for each pair <component,node>. Each value represents the cost of scheduling an application component to an infrastructure node. This value is updated every time a component is scheduled.

At this point the scheduler takes the sorted set of components, explained in the first paragraph of this section, and starts scheduling one component at a time. For each component, the scheduler needs to calculate which infrastructure nodes can be selected to schedule the evaluated component and give each one of them a value calculated using both cost and affinity matrix. This value is used to sort the set of infrastructure nodes. It calculates a set of nodes instead of a single node because if later on, during the scheduling of other components, there is not enough space for that component, the scheduler will rollback to the previous application component and retry the scheduling phase selecting the next node from the sorted set.

Finally, when all the components of an application are scheduled, the next application is evaluated. During the rest of this chapter, all main steps described above are furthermore detailed.

[3]'neighbors of x' are components/nodes that are connected via a link to 'x'

3.2.1 Sorting Application Components

The PHARE scheduler schedules components of an application sorted by both importance and constraints. The importance is evaluated taking into account the resources required by the component (a.k.a. *weight*) and by the components which are connected to the evaluated component (a.k.a. *scaledWeight*): even if the first value is very simple to calculate, the second one is a bit more complex because there is the need to estimate with a single value if the connection between two components is requiring an high amount of resources or not. In order to compute this value, the scheduler divides each connection's requirements by the total amount of resources requested by the application's connections (the sum of resources requested by each connection).

Once these values are calculated for each application component, the next step is to sort them from the higher to lower component requirements. These requirements can be split into two different areas: the amount of resources and component constraints. The resources sorting can be performed comparing components *weight* or *scaledWeight* setting a value inside the configuration file. Component constraints are similar to `NodeSelector` in Kubernetes: an infrastructure node must match application component constraints in order to be selected as target node for the component. The sorting based on component constraints assumes that a node with a constraint is harder to schedule than once without because it can be scheduled only on a subset of the nodes of the infrastructure. In case the compared components both have or do not have a policy, the scheduler will compare them using their resource requests.

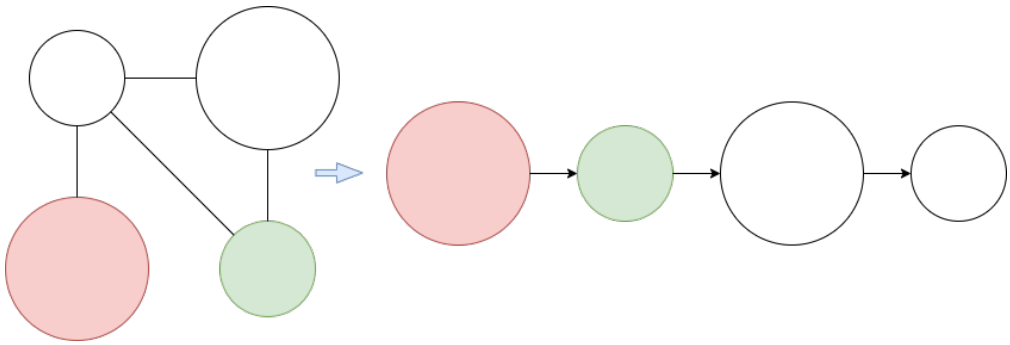

Figure 3.4: Jobs ordering of an application

To summarize, the final job order will consider jobs with constraints first and job without later; jobs are then ordered from the highest requirements one to the lowest one. In the example showed in fig. 3.4, colorized circles represent components with constraints. The sorted result displays that components with constraints have the priority over components without, even if the component with constraint has less

requirements than the other.

3.2.2 Affinity Matrix

Another important element of the scheduler is the Affinity Matrix, which contains a value that displays the affinity between each application component and infrastructure node. The affinity value can assume values between 0 and 1, where 1 means "very affine" and 0 means "low or not affine". Affinity refers to the amount of resources of a component and its neighbors that can be hosted by an infrastructure node and is calculated following section 3.3.6 and section 3.3.7. The affinity value is a product of two different values: one of them is computed over the amount of resources requested by the evaluated component and its neighbors (a.k.a. *computing-affinity*), and the other one is calculated taking into account connections' resources with its neighbors (a.k.a. *communication-affinity*).

If a job cannot be hosted, the affinity value is set to 0. Assuming that the job can be hosted, the *computing-affinity* and the sum of the resources that can be hosted on a node accordingly to its policies are calculated. This value related to policies is fundamental because it allows to place the evaluated component on an infrastructure node which has policies that can host the component itself and also the majority of its neighbors.

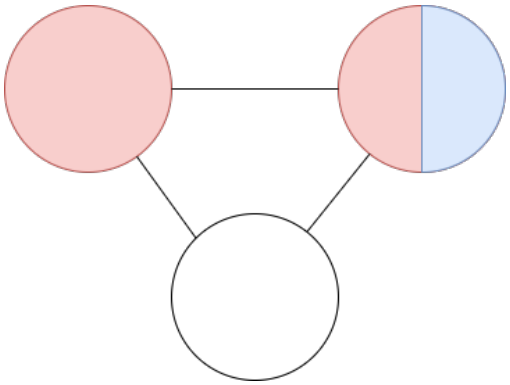

Figure 3.5: Infrastructure with policies

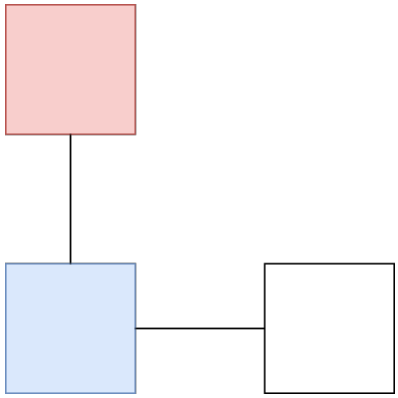

Figure 3.6: Application with policies

In the example above, the infrastructure at fig. 3.5 has 3 nodes, one of them that can host *red* policy and another one that can host both *red* and *blue* policies. The application (ref. fig. 3.6) has 3 components: one with a *red* label and another with a *blue* label. Lets assume that the first scheduled component is the *red* one: it can be placed in both *red* and *red-blue* infrastructure nodes, but it's more convenient to place it in the *red-blue* one because in this way its *blue* neighbor can be placed on the same node and, for this reason, reduce the cost.

In the next step the *communication-affinity* is multiplied with the *computing-affinity* value to produce the *affinity matrix* value.

3.2.3 Cost Matrix

The *cost matrix* is the data structure used by the scheduler to keep track of the evolution of scheduling costs. It contains, for each application component and infrastructure node, the cost of placing the component on the node. The value is computed multiplying the amount of resources requested by the component and the unitary cost of the infrastructure node. Every time a job J is scheduled on a node N, the scheduler forces neighbors of J to be scheduled on N or on its neighbors. For this reason, the value of the *cost matrix* relative to J's neighbors must be updated with the cost of links between N and its neighbors.

3.2.4 Nodes Priority

Once components are sorted, the scheduler will process them one at a time. For each job, the scheduler must select the best node on which to schedule the job, schedule it and recur on the next job. If the recursion determines that the a job wasn't schedulable, the scheduler will remove the job from the infrastructure node, select another node and recur again (hence following a different path with respect to the previous one).

The set of nodes on which the job can be scheduled is sorted in ascending order by the priority value explained in the following. For each infrastructure node that can host the evaluated job, the priority value is computed evaluating both the affinity and the cost. Sometimes the priority value computed on two different nodes is equal. If this is the case, given that the goal of the PHARE scheduler is to keep applications as close as possible, the scheduler will check if infrastructure nodes have jobs already scheduled on them or not: if only one of them is not empty (i.e. has no jobs scheduled on), the non-empty node is selected; otherwise, if both have(n't) job scheduled, the selected node will be the one with the higher amount of resources because it is more likely that the node will enough resources to host next applications (and avoid the use of another node). If also the amount of resources of both nodes are equal, the alphabetic order of nodes' id is evaluated.

In the example at fig. 3.7, $N1$ and $N2$ have the same amount of resources, $N3$ instead has less resources. Both $N1$ and $N3$ have already some application components scheduled on them, $N2$ instead is empty. Supposing that a job has these three nodes that can be chosen as scheduling target, the node $N1$ is selected first because is the non-empty one with the bigger amount of resources, the second

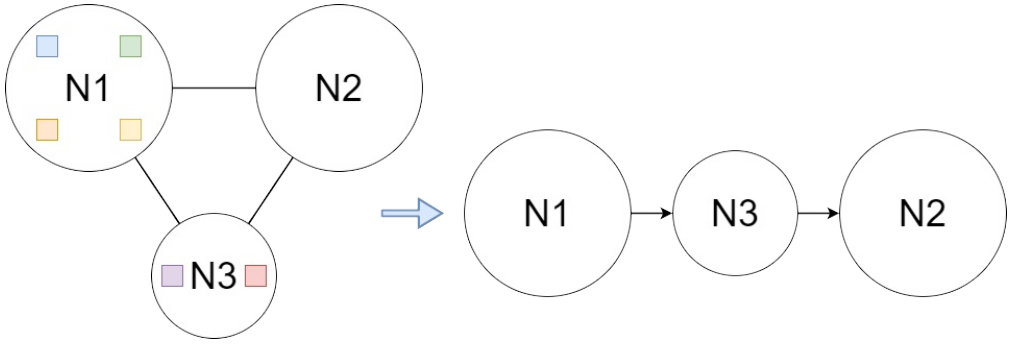

Figure 3.7: Nodes Priority ordering

is $N3$ because, even if it is smaller than $N2$, has enough resources to host the job and it is already been used as target of other jobs.

3.3 Algorithm core

Before describing the main concepts behind the algorithm logic, the document describes the main challenges of scheduling components in distributed constrained infrastructure. Then, the algorithm and detail its details are analyzed.

3.3.1 Main Concepts

When taking the decision of scheduling a particular component on a given cluster, a key role is played both by how big the component is (i.e., how much cluster resources it demands) and by how much it communicates with other components of the same application. A component that requires a lot of computational resources will be harder to schedule (it has less feasible matches) compared to small components, but this is also true for small components that feature intensive mutual communication (e.g., if the chosen host cluster has not enough bandwidth, the communication with any component placed outside will not occur properly).

Since edge infrastructures are highly scattered and constrained, it may be particularly hard to satisfy the communication or computation requirements of all the components to be deployed. Intuitively, the more components are scheduled the harder will be to schedule the next ones. Therefore, to quickly converge to a feasible placement, the algorithm should somehow prioritize the scheduling of "harder" components, i.e., the ones featuring more stringent constraints.

3.3.2 Algorithm Model

A distributed edge infrastructure was considered where resource are grouped in clusters. Potentially, each cluster $v \in \mathcal{N}$ is owned by a different edge provider. In the following, it is used index $v \in \mathcal{N}$ also for indicating the provider owner of cluster v. Clusters are heterogeneous and may provide different resource capabilities (e.g., centralized data centers, network access base stations, central offices, but also isolated user devices). In this document work capabilities are considered in terms of *computing resources* (i.e., total amount of CPU and RAM shares available in the cluster), and *communication resources* (i.e., amount of network bandwidth that the cluster uses to communicate with the rest of the world). From this point on for simplicity the RAM notation was omitted from the dissertation since the same principles apply as for the CPU. In particular, on cluster $v \in \mathcal{N}$, the budget of CPU resources is denoted with $b_v^{CPU} \in \mathbb{R}_+$ and with $b_v^{BW} \in \mathbb{R}_+$ the budget of bandwidth resources.

Requests for deploying applications are issued to edge providers. Each application $i \in \mathcal{I}$ is formed by a set of components $\mathcal{M}_i \in \mathcal{M}$, where $\mathcal{M}$ is the set of all possible components. Each component of an application can be individually scheduled, but it may feature inter-component dependencies: component $j \in \mathcal{M}_i$ has given resource demands both in terms of computing $\rho_j^{CPU} \in \mathbb{R}_+$ (amount of CPU required) and communication with other components $\rho_{j,k}^{BW} \in \mathbb{R}_+$ (bandwidth that j requires to communicate with component k from the same application).

Edge providers jointly deploy applications across their clusters, thus forming a federated edge infrastructure. Upon receiving the request for deploying an application, the concerned edge provider decides which of the application components should be executed locally (i.e., on its own cluster) and which of them will instead be offloaded to foreign clusters across the federation.

Each available resource on every cluster features a given price per unit. To preserve generality, lets assume that every provider may see different costs for the same resources (i.e., resources may be exposed with different prices to different partners of the federation). $c_{v,v'}^{CPU} \in \mathbb{R}_+$ and $c_{v,v'}^{BW} \in \mathbb{R}_+$ refer respectively the prices of allocating CPU and bandwidth units on cluster v as seen by provider v'. The following equation denotes

$$x_{j,v}^i \in \{0,1\}, \text{ for } i \in \mathcal{I}, j \in \mathcal{M}_i, v \in \mathcal{N}, \tag{3.1}$$

the decision variable that indicates if component j from application i is deployed on cluster v. When allocating a certain component $j \in \mathcal{M}_i$ on cluster v, edge provider v' experiences a cost given by multiplying the amount of each demanded

resource for the cluster resource price:

$$C_{v'}(j,v) = \rho_j^{CPU} c_{v,v'}^{CPU} + \sum_{k \in \mathcal{M}_i} \rho_{j,k}^{BW} c_{v,v'}^{BW} 1_{\{x_{k,v}^i \neq 1\}}. \tag{3.2}$$

Note that the cost $\rho_{j,k}^{BW} c_{v,v'}^{BW}$ due to the communication between components j and k is accounted only if j and k are not deployed on the same cluster.

When scheduling application components on the available clusters, the Edge Provider seeks cost minimization of the overall deployment, and its decision is subject to the resource constraints of the federated edge infrastructure. The formulation of such optimization problem for Edge Provider[4] v' as follows:

$$\text{minimize} \qquad \sum_{j \in \mathcal{M}_i} \sum_{v \in \mathcal{N}} x_{j,v}^i C(j,v), \forall i \in \mathcal{I} \tag{3.3}$$

subject to

$$\sum_{i \in \mathcal{I}} \sum_{j \in \mathcal{M}_i} x_{j,v}^i \rho_j^{CPU} \leq b_v^{CPU} \qquad \forall v \in \mathcal{N} \tag{3.4}$$

$$\sum_{i \in \mathcal{I}} \sum_{j \in \mathcal{M}_i} \sum_{k \in \mathcal{M}_i \setminus \{j\}} x_{j,v}^i \rho_{j,k}^{BW} 1_{\{x_{k,v}^i \neq 1\}} \leq b_v^{BW} \qquad \forall v \in \mathcal{N} \tag{3.5}$$

where constraint (3.4) ensures that components deployed on each cluster do not violate the computation constraint, while (3.5) enforces the bandwidth constraint by considering the communication demands between components that are deployed on different clusters.

3.3.3 Overview

When the request for the deployment of a new application $i \in \mathcal{I}$ is received, the algorithm first evaluate every component $j \in \mathcal{M}_i$ of application i and assigns an *importance* metric z_j to each of them (section 3.3.5). Then, an affinity score is computed for each pair (j, v) of components j and available clusters $v \in \mathcal{N}$. The affinity provides an indication of how good it is to assign component j to cluster v, with respect to a trade-off between convergence speed and optimality of the final

[4]Since the algorithm operates in a decentralized fashion at the level of an Edge Provider, it will be, from now, omitted the underscript $\times_{(v')}$ for simplicity.

scheduling decision. In particular, two separate affinities $\Phi_{j,v} \in [0,1]$ (*computing affinity*, see section 3.3.6) and $\Psi_{j,v} \in [0,1]$ (*communication affinity*, see section 3.3.7) are computed and combined. The algorithm then computes costs $C(j,v)$ for every component j and feasible cluster v, i.e., the marginal cost that would be required if j is scheduled on v. Such raw costs are adjusted using the affinity values computed at the previous steps, thus obtaining the so called *perceived cost* $C(j,v)/(\Phi_{j,v}\Psi_{j,v})$; the less is the affinity between component j and cluster v, the higher will be the perceived cost of scheduling j on v. In the last step, the algorithm iterates the components sorted by their importance z_j (descending), and assigns each component to the cluster v^* with the less perceived cost $v^* = \arg\min_{v* \in \mathcal{N}} (C(j,v)/\Phi_{j,v}\Psi_{j,v})$.

3.3.4 Sorting Components Following Constraints

Each application component can have some constraints which must be matched by a cluster in order to be selected as target for the component. An application component with constraints will probably have less clusters on which be scheduled on, for this reason a component with constraints compared to a component without will ignore the importance value explained in section 3.3.5 and be considered more important. These constraints are also referred as "policies" in the document.

Each constraint is identified by a key and has one or more values associated to the key. Moreover there is another parameter which can assume value "In" or "Out": let's call V_{KA} the values of the application component A referred to the key K and V_{KC} the values of the cluster C referred to the key K: if the operator is "In", at least one of the values related to V_{KA} must be equal to values of V_{KC}. On the other hand, if the operator is "Out", none of the values of V_{KA} must be equal to values of V_{KC}.

3.3.5 Sorting Components by Importance

The evaluation of the importance of a component j mainly based on its demand in terms of computing resource ρ_j^{CPU} and its constraints. The importance value is combined with the demands ρ_k^{CPU} of each "neighbor" component k, i.e., all those component that feature some communication constraint with j. By prioritizing components with "big" neighbors the probability that such neighbors are scheduled on the same clusters is increased.

Before describing how to compute the importance z_j of a component j, there is the need to provide the following definition of *communication factor*.

Definition 1. *(Communication Factor $\theta_{j,k}$). Given an application $i \in \mathcal{I}$ and two of its components $j,k \in \mathcal{M}_i$, $\theta_{j,k} = \rho_{j,k}^{BW} / \max_{j',k' \in \mathcal{M}_i}(\rho_{j',k'}^{BW})$ is the communication*

factor between components j and k of application i.

The communication factor $\theta_{j,k}$ is an indicator of how intense is the communication demand between components j and k. The communication factor is used to weight the contribution of each neighbor of j when computing the importance metric z_j, as follows:

Definition 2. *(Importance z_j). Given component $j \in \mathcal{M}_i$ of an application $i \in \mathcal{I}$, the importance of component j is defined as $z_j = \rho_j^{CPU} + \sum_{k \in \mathcal{M}_i \setminus \{j\}} \theta_{j,k} \rho_k^{CPU}$.*

Intuitively, components that feature *(i)* high computing demands, *(ii)* neighbors with high computing demands, and *(iii)* high communication demands will have high importance factors.

The scheduler uses both constraints and the importance z_j to sort components of an application so that those featuring more stringent deployment constraints are scheduled first. Additionally, the importance z_j is used to compute the affinity of j with the available clusters as described in the next section.

3.3.6 Components-Clusters Affinity: Computing

The first affinity factor evaluated only takes into account the computing resources of the target cluster, without considering its communication capability. To estimate the affinity between component j on cluster v, first it is evaluated the quantity $y_j - r_v^{CPU}$, where r_v^{CPU} is the residual computing resource on cluster v, while y_j is amount of overall computing resources required by j and all its neighbors, i.e., $y_j = \rho_j^{CPU} + \sum_{k \in \mathcal{M}_i \setminus \{j\}} \rho_k^{CPU} \mathbb{1}_{\{\rho_{j,k}^{BW} > 0\}}$.

The computing affinity is then evaluated based on the quantity $y_j - r_v^{CPU}$ as follows:

$$\Phi_{j,v} = \begin{cases} e^{-c \frac{y_j - r_v^{CPU}}{y_j - z_j}} & \text{if } y_j - r_v^{CPU} > 0, \\ 1 & \text{if } y_j - r_v^{CPU} \leq 0. \end{cases} \tag{3.6}$$

where c is a coefficient used to adjust how fast the affinity decreases with respect to the lack of resources on the cluster.

To understand the rationale between Equation (3.6) it is helpful to visualize the relationship between $\Phi_{j,v}$ and the quantity $y_j - r_v^{CPU}$ (Figure 3.8). An intuition is provided below.

When $y_j - r_v^{CPU} \leq 0$, i.e., cluster v has enough resources to host j and all its neighborhood, then the affinity is set to 1 (maximum affinity value).

If the resources on v are not enough for hosting j and all its neighbors, the affinity start to drop slowly, until the quantity $y_j - r_v^{CPU}$ reaches a critical value where

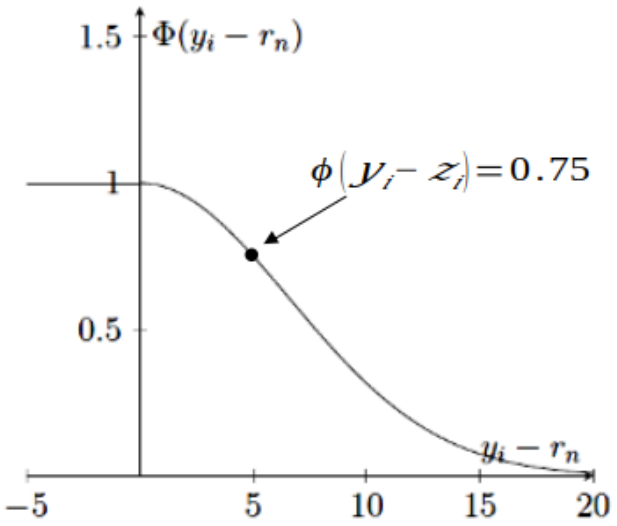

Figure 3.8: Cluster remaining resources and application resource requirements correlation.

$\Phi_{j,v} = 0.75$; the coefficient c is set so that $\Phi_{j,v} = 0.75$ when $y_j - r_v^{CPU} = y_j - z_j$,[5] i.e., when the residual resources on v are numerically equal to the importance z_j of component j. The importance value here is used to estimate the portion of neighborhood that is more significant for j: therefore, if the cluster has enough resources for hosting j and a significant portion of its neighborhood, then the affinity $\Phi_{j,v}$ will be higher than 0.75.

Finally, after the critical value where $y_j - r_v^{CPU} = y_j - z_j$ is reached, the affinity $\Phi_{j,v}$ starts to drop quickly, with values close to 0 when the available resources are few compared to the demand of component j.

3.3.7 Components-Clusters Affinity: Communication

The computing affinity $\Phi_{j,v}$ is used in combination with a communication affinity that also takes into account the networking capabilities of the target cluster. It is important to understand that a task j will consume the communication capabilities (e.g., bandwidth) of the host cluster v only if its neighbors have been placed on some external cluster other than v, since if j's neighbors are host in v, j will not need v's bandwidth to communicate with them). For this reason, when designing

[5]This is achieved when $c = 0.287682$.

the communication affinity, there was the need of a mechanism that reduces the affinity between component j and cluster v the more it is difficult to accommodate the communication demands of j, but that have a lower impact if cluster v is big enough for possibly hosting j neighbors.

To calculate the communication affinity, first evaluate the quantity $y_j^{BW}/r_v^{BW} - 1$, where r_v^{BW} is the residual communication capacity on cluster v, while y_j^{BW} is the overall communication demands for component j towards all its neighbors, i.e., $y_j^{BW} = \sum_{k \in \mathcal{M}_i \setminus \{j\}} \rho_{j,k}^{BW}$. Note that the quantity $y_j^{BW}/r_v^{BW} - 1$ is equal to zero when $y_j^{BW} = r_v^{BW}$.

The communication affinity is then evaluated as follows:

$$\Psi_{j,v} = \begin{cases} e^{-a y_j^{BW}/r_v^{BW} - 1} & \text{if } y_j^{BW}/r_v^{BW} - 1 > 0, \\ 1 & \text{if } y_j^{BW}/r_v^{BW} - 1 \leq 0. \end{cases} \tag{3.7}$$

$$\Psi_{j,v} = \begin{cases} e^{-a(y_j^{BW}/r_v^{BW} - 1)} & \text{if } y_j^{BW} > r_v^{BW}, \\ 1 & \text{if } y_j^{BW} \leq r_v^{BW}. \end{cases} \tag{3.8}$$

where coefficient a is used to adjust the weight of the communication affinity so that it has a lower impact if the target cluster v is big enough to host all j neighborhood: the higher coefficient a is, the more the communication affinity will affect the final solution (see below for a description of how a is computed differently for each cluster).

The relationship between $\Psi_{j,v}$ and the quantity $y_j^{BW}/r_v^{BW} - 1$ is visualized in Figure 3.9. When cluster v has enough networking resources for accommodating all the communication demands of component j, the communication affinity is set to the maximum value 1. If the resources are not enough (e.g., the residual bandwidth is less compared to the bandwidth needed to j to communicate with the other components of the application), $\Psi_{j,v}$ starts to drop exponentially with a decreasing factor that is based on the coefficient a: for higher a, the value of $\Psi_{j,v}$ drops more quickly.

Compute the coefficient a. First compute the coefficient a so that $\Psi_{j,v}$ decreases more slowly, the more cluster v is likely to host some neighbors of j. The rationale is that if cluster v has enough computing resources to host a subset of j's neighborhood, then it is unfair to decrease the affinity between v and j based on v's communication capabilities (since j probably will not need them). a is computed using a sigmoid

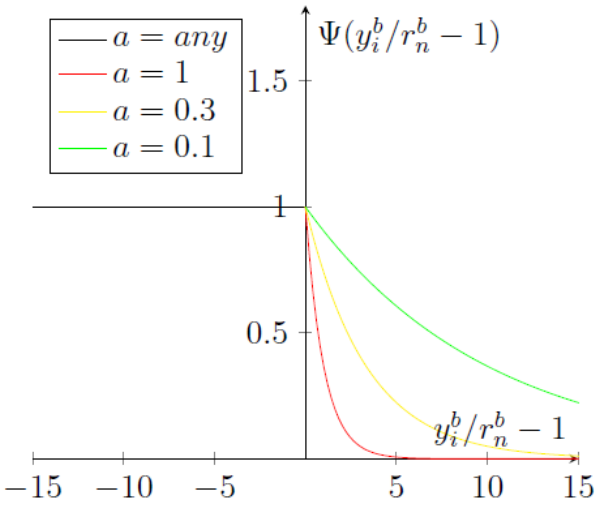

Figure 3.9: Bandwidth affinity correlation

function

$$a = S\Big(8\frac{y_j - r_v^{CPU}}{y_j - z_j} - 4\Big), \tag{3.9}$$

where S is the sigmoid function $S(x) = 1/(e^{-x} + 1)$. If cluster v has enough resource compared to the demand of j and its neighborhood, then $a \simeq 0$, i.e., the communication affinity will have a negligible impact. If residual resources are not enough for hosting j and a significant portion of its neighborhood (estimated with z_j, then $a \simeq 1$ and the communication affinity will have its maximum impact.

3.3.8 Multiple Resource Constraints

Until now the dissertation focused on the CPU as the only discriminating computational resource for any scheduling decision. The same concepts described in the previous sections can also be applied on Memory as well since they experience similarities both in term of provisioning and cost modelling. Nevertheless realistic application deployments feature always two or even more conjoined computational constraints; now it is described how multiple resource constraints can then be combined with the heuristics.

The values of computing and communication affinity defined respectively at eq. (3.6) and eq. (3.7) provide an indication of the confidence of scheduling a

component on a given cluster; having multiple computational constraints result in having multiple $\Phi^t_{j,v}$ and $\Psi^t_{j,v}$ for every component j, one for each type of resource t. The computation of the final values of affinity can then be generalized:

$$\Phi'_{j,v} = min(\Phi^t_{j,v}) \text{ and } \Psi'_{j,v} = min(\Psi^t_{j,v}) \tag{3.10}$$

extracting the minimum values of affinity among the resources, hence considering then the most conservative scenario.

Chapter 4

PHARE Scheduler Implementation

4.1 Introduction

The prototype version of PHARE is implemented using Golang [5], a recently developed programming language by Google, mainly focused on high-performance both in networking and in multiprocessing. The framework has been designed to be easily extensible, so that additional scheduling algorithms can be integrated and tested in the same conditions. The implementation is based on an initial work conducted in the Computer Networks groups of Politecnico di Torino [6].

In this section there will be a description about the implementation details of the algorithm presented in the previous sections, providing also insights of the main technical challenges addressed in the development process to achieve the goal of optimal job placement with low latency.

4.2 Overview

The scheduling framework operates on simulated environments in which both the infrastructures and the component-based applications are represented as data structures stored in memory; they can be either imported manually, to replicate specific scenarios, or they can be automatically generated for testing purposes. Particularly the automatic generation is performed through configuration files defining the tuple `(minValue-maxValue)` for each type of resource both for application demands and for infrastructures available resources; the framework then generates randomized

infrastructures and applications within the specified ranges for the scheduler to operate.

The scheduling framework has been designed with the key concept of modularity: multiple scheduling algorithms can be integrated and can operate on the previously described environments (either imported or generated). The framework strongly relies on interfaces to ensure such property: each scheduling algorithm must implement an interface named `Scheduler` which has a couple of methods, one of them is the function `Schedule` that receive the information related to the infrastructure and the components of the application and actually perform the scheduling operation. The runtime environment takes care of calling the Schedule function every time a new application scheduling request is issued.

Execution metrics are finally exported to evaluate the performance of each scheduling algorithm; they describe many aspects throughout the execution of the tests and they mainly focus on the scheduling success rate, the cost of the identified solutions, the scheduling time and the distribution of the applications within the infrastructure.

4.3 PHARE Scheduler Structure

The scheduler structure is loaded via a configuration file that describes which algorithm will be used (alongside with other customizable parameters), the infrastructure and the list of applications that expects to be scheduled.

The implementation of the Scheduler interface for PHARE relies on recursion to replicate the algorithm described in the previous chapter (see Code 4.1). Within the Schedule function first the components are sorted by importance according to definition 2, identifying the ones with strongest requirements (i.e. the ones that will be scheduled first); an affinity matrix is then computed between each tuple <component,cluster> as the product between the computing and the communication affinity.

The maximum depth of the recursion is equal to the number of components of the application and within each recursive call the list of available nodes is computed, sorted from the most to the least promising; finally to compute the final placement sequentially adding the current cluster to the solution, recur on the following component and eventually remove the cluster if the recursive branches did not lead to a feasible solution.

```
func (s CASTOScheduler)InitScheduler(c config){
  s.infrastructure = loadInfrastructure(c)
  jobs = getApplications(c)
  for j := range jobs {
    s.Schedule(j,s.infrastructure)
  }
}
func (s CASTOScheduler)Schedule(j []Job, infra []
    Infrastructure) {
  sJobs = sortJobsByImportance(j)
  AM = computeAffinityMatrix(sJobs, infra)
  CM = computeCostMatrix(sJobs, infra)
  if recursiveScheduler() == true {
    solution.found(true)
  } else {
    solution.found(false)
  }
}
func recursiveScheduler() bool {
  if depth >= len(jobs) {return true}
  NP = createNodesPriority()
  for node in NP {
    solution.Add(jobs[depth], node)
    if recursiveSchedule() == true {return true}
    solution.Remove(jobs[depth], node)
  }
  return false
}
```

Code 4.1: PHARE implementation pseudo-code

4.4 Application Sorting

The application sorting is the first phase of the algorithm during the scheduling of each application. To summarize what written in section 3.2.1, application's components are sorted taking into account their policies and their resource requirements.

The most interesting thing in Code 4.2 is the node sorting, which is performed using the `sort`[1] library which sorts slices using the QuickSort algorithm.

[1]https://pkg.go.dev/sort

```
1  func sortJobsByImportance(jobs []Job, totConnRequest float,
       schedulingOrder string) []Jobs {
2    sortedJobs = []
3    for j := range jobs {
4      weight = j.GetWeight()
5      scaledWeight = j.GetWeight()
6      for connJob := range j.ConnectedJobs() {
7        connJobWeight = connJob.GetWeight()
8        weight += connJobWeight
9        theta = LinkBetween(j,connJob).GetWeight() /
       totConnRequest
10       scaledWeight += theta * connJobWeight
11     }
12     j.SetConnWeight(weight)
13     j.SetConnScaledWeight(scaledWeight)
14     sortedJobs.append(j)
15   }
16   sort.Slice(sortedJobs, func(i, j int) bool {
17     return componentComparator(sortedJobs[i], sortedJobs[j
       ], schedulingOrder)
18   })
19   return sortedJobs
20 }
21
22 func componentComparator(jA Job, jB Job, schedulingOrder
       string) bool {
23   if !(jA.HasSchedPol() != jB.HasSchedPol()) {
24     // both have(n't) the schedPolicy -> use the '
       schedulingOrder'
25     if schedulingOrder == "weight" {
26       return jA.GetConnWeight() > jB.GetConnWeight()
27     } else if schedulingOrder == "scaledWeight" {
28       return jA.GetConnScaledWeight() > jB.
       GetConnScaledWeight()
29     } else {
30       // error
31     }
32   } else {
33     // this returns 'true' if jA has a schedPolicy and jB
       not ,'false' otherwise
34     return jA.HasSchedPol()
35   }
36 }
```

Code 4.2: PHARE sortedJobsByImportance pseudo-code

4.5 Affinity Matrix

The affinity matrix, described in section 3.2.2, has an entry for each <node, component> and shows the "affinity" between an infrastructure node and an application component. Higher is the correlation, more likely the component will be scheduled on the node.

The pseudo-code at Code 4.3, the `phi` function (*line 17*) is implemented following eq. (3.6), and it is multiplied with the `nodeSelFactor` to compute the *computing-affinity* value. The *communication-affinity*, computed using the `psi` function (implemented reflecting eq. (3.7)), is multiplied with the *computing-affinity* value to produce the *affinity matrix* value (*line 19*).

```
1  func computeAffinity(jobs []Job, infraNodes []
       Infrastructure, ..) .. {
2    AM = make(map...)
3    for j := range jobs {
4      neighWeight = j.GetConnWeight()
5      for infra := range infraNodes {
6        if !infra.CanHost(j){
7          AM[j][infra] = 0
8        } else {
9          // computing affinity
10         totWeight = j.GetWeight()
11         for connJob := range j.ConnectedJobs() {
12           if infra.CanHostNodeSelector(connJob.
       GetNodeSelector()) {
13             totWeight += connJob.GetWeight()
14           }
15         }
16         nodeSelFactor = totWeight / neighWeight
17         AM[j][infra] = phi(...) * nodeSelFactor
18         // communication affinity
19         AM[j][infra] = AM[j][infra] * psi(...)
20       }
21     }
22   }
23   return AM
24 }
```

Code 4.3: PHARE computeAffinity pseudo-code

4.6 Cost Matrix

The cost matrix keeps a value for each pair <node, component> which refers to the cost of scheduling the component of the infrastructure node.

The function `updateCM` is called after a job has been placed on a node. Assuming a job J has just been scheduled on a node N: this function doesn't update all the entries of the cost matrix, it updates only entries of the J's neighbors and N's neighbors. Each cost is updated taking the amount of resources requested by the link connecting J's neighbor and J itself, and multiplying it with the cost of the link between N and its neighbor.

```
1  func computeCostMatrix(jobs []Job, infraNodes []
       Infrastructure) .. {
2    CM = make(map..)
3    for j := range jobs {
4      for infra := range infraNodes {
5        weight = j.GetWeight()
6        nodeUnitCost = infra.GetCost()
7        CM[j][infra] = weight * nodeUnitCost
8      }
9    }
10 }
11
12 func updateCM(CM CMStructure, placedJob Job,
       selectedInfraNode Infrastructure, ..) ... {
13   for connJob := range placedJob.ConnectedJobs() {
14     for connNode := selectedInfraNode.ConnectedNodes() {
15       infraLinkCost = LinkBetween(selectedInfraNode,
       connNode).GetCost()
16       jobLinkWeight = LinkBetween(placedJob,connJob).
       GetWeight()
17       CM[connJob][connNode] += infraLinkCost *
       jobLinkWeight
18     }
19   }
20   return CM
21 }
```

Code 4.4: PHARE costMatrix pseudo-code

4.7 Nodes Priority

The Nodes Priority (ref. section 3.2.4) is computed for each component of an application. It is an array which contains nodes that can be selected as target for

job, ordered from the most to the least convenient node.

```
func createNodesPriority(job Job, infraNodes []
    Infrastructure, NPSize int, ..) .. {
  NP = make(..)
  for infra := range infraNodes {
    if infra.CanHost(job) {
      cost = CM[job][infra]
      nodeAff = AM[job][infra]
      if !linksCanHost(infra, job.ConnectedScheduledJobs())
    {
        continue
      }
      if nodeAff == 0 {
        NP.append(priority: MaxFloat)
      } else {
        NP.append(priority: cost / nodeAff)
      }
    }
  }
  if len(NP) < NPSize {
      return NP.top(compareNP,NPSize)
  } else {
      return NP.sort(compareNP).getFirstN(NPSize)
  }
}
func compareNP(a_NP, b_NP) bool {
  if a_NP.priority != b_NP.priority {
    return a_NP.priority < b_NP.priority
  } else {
      a_NPIsUsed = a_NP.IsUsed()
      b_NPIsUsed = b_NP.IsUsed()
    if a_NPIsUsed != b_NPIsUsed { // only one is used
      return b_NP.IsUsed() // return true if 'b' is used,
   false otherwise
    }
    comp = compareResources(a_NP,b_NP)
    if comp == 0 {
      return string.compare(a_NP.ID(),b_NP.ID())
    }
    return comp < 0
  }
}
```

Code 4.5: PHARE nodes priority pseudo-code

An important speed-up that was added in this code (ref. Code 4.5) is the sorting phase: a parameter that can be passed to `createNodesPriority` is `NPSize`: this value defines the size of the Nodes Priority array. If `NPSize` is smaller than the

length of the Nodes Priority array, only the first `NPSize`-elements of the sorted `NP` are returned. This means that also a top-algorithm can compute the `NP` sorting. Thie "top" algorithm is faster than a QuickSort sorting one when the `NPSize` is small. Due to lack of time, the algorithm selector is not the smartest one and can be improved: right now, the algorithm selection is driven by a simple `if-else` statement (*line 17*) and sometimes the top-algorithm is used even if it's not the fastest solution.

4.8 Internal representation

Each infrastructure or application is saved into a directory which contains two subdirectories for nodes and links called respectively `nodes` and `edges`, a file called `graph.dot` which contains the representation of interconnections between nodes in `dot` format graph, and finally, only for infrastructures, a file called `components.json` which contains the definition of each physical link that compose virtual links stored inside the `edges` directory.

Each node stores its specifications inside a JSON file. After a closer look to Code 4.6, it is shown that applications have a list of resource-types (labels) and, for each type, `request` and `limit` are the same adopted by Kubernetes: `request` and `limit` define respectively the minimum and the maximum amount of resource-type requested by the application node.

The next element inside the application node specification is the `nodeSelector`: this object is very similar to its homonym in Kubernetes, and its role is to define a filter for infrastructure nodes on which the application node can be scheduled. The `nodeSelector` is composed by:

- `key`: unique identifier
- `values`: set of values to compare with values stored in infrastructure nodes referring to the same key
- `operator`: this field specifies how to compare infrastructure values and application values (ref. section 3.3.4).

Focusing on Code 4.7, it's shown that the `spec` on infrastructure nodes provides the following information:

- `quota` shows the total amount of resources available on the cluster, which is the sum of the resources of each device inside the cluster.

- `cost` refers to the unitary cost of the resource. Taking Code 4.7 as example, using 1 `cpu` will cost 1.
- `threshold` and `maxUnit` are explained at section 4.10

For the application nodes, the `spec` contains instead (following the Kubernetes approach):

- `request` shows the total amount of resources requested by a node.
- `limit` reports the maximum amount of resources that can be used by a node. Exceeding this value could lead to an unstable state of the infrastructure node.

```
{
  "labels": [
    {
      "meta": {
        "kind": "resource
    ",
        "type": "cpuDemand
    "
      },
      "spec": {
        "request": "1",
        "limit": "1"
      }
    }
  ],
  "nodeSelector": [
    {
      "operator": "In",
      "key": "user",
      "values": [
        "n0"
      ]
    }
  ]
}
```

Code 4.6: Application node

```
{
  "labels": [
    {
      "meta": {
        "kind": "resource
    ",
        "type": "cpu"
      },
      "spec": {
        "quota": "8",
        "cost": 1,
        "threshold": 0.5,
        "maxUnit": "8"
      }
    }
  ],
  "nodeSelectorLabels": [
    {
      "key": "user",
      "values": [
        "n0"
      ]
    }
  ]
}
```

Code 4.7: Infrastructure node

The scheduler keeps also a representation for connections pod-to-pod and cluster-to-cluster: these connections are edges. Both edges types (application and infrastructure) have a list of resource types inside `labels`. The application edge in Code 4.8 contains, for each resource type, the specification of `request` and `limit` as already explained above for the nodes. Infrastructure edges, ref. Code 4.9,

contain, for each resource type, inside the field called `components`, the reference to the representation of each physical link that connects the two pods (ref. Code 4.10). Each element inside Code 4.10 contains `cost`, which refers to the unitary cost of the resource, and `amount` which represents the amount of resources of a link.

```
{
  "labels": [
    {
      "meta": {
        "kind": "resource
    ",
        "type": "
    bandwidthDemand"
      },
      "spec": {
        "request": "1G",
        "limit": "1G"
      }
    }
  ]
}
```

Code 4.8: Application edge

```
{
  "labels": [
    {
      "meta": {
        "kind": "resource
    ",
        "type": "bandwidth
    "
      },
      "spec": {
        "components": [
          "n1",
          "n0"
        ]
      }
    }
  ]
}
```

Code 4.9: Infrastructure edge

```
{
  "bandwidth": {
    "n0": {
      "cost": "1n",
      "amount": "1G"
    },
    "n1": {
      "cost": "1n",
      "amount": "1G"
    },
    ..
  },
  "latency": {
    ..
  }
}
```

Code 4.10: Components

4.9 PHARE Scheduler Input

The scheduler needs a configuration file as input to retrieve information about the scheduler itself (f.e. the scheduling algorithm), the infrastructure definition and the applications to schedule. This configuration file (ref. Code 4.11) is composed by the `metrics` part (*line 2*), which contains informations about how to collect scheduling metrics like scheduling time or cost, and the `offline` (*line 4*) part that contains the declaration of the algorithm, and also the definition of infrastructure and applications. Focusing on the `offline` section, the `algorithm` field (*line 6*) contains the declaration of the scheduling algorithm. In the example is showed what parameters are needed by the PHARE scheduler, but different algorithms requires different parameters. The `fedConfig` (*line 9*) and `depsConfig` (*line 16*) contain paths to file locations respectively for infrastructures and applications ("infrastructure" is also named as "federation", as well as "deployment" for "application"). Each field inside `fedConfig` and `depsConfig` has been already explained in section 4.8.

```
1  name: liqo
2  metrics:
3    ...
4  offline:
5    ...
6    algorithm:
7      phare:
8        ...
9    fedConfig:
10     name: fed
11     graph: path/to/graph.dot
12     nodes: path/to/nodes
13     edges: path/to/edges
14     componentsFile: path/to/components.json
15     source: file
16   depsConfig:
17     - name: app0
18       graph: path/to/app0/graph.dot
19       nodes: path/to/app0/nodes
20       edges: path/to/app0/edges
21       source: file
22     - name: app1
23       config: path/to/appConfig.yml
24       directed: true
25       source: generator
```

Code 4.11: PHARE configuration file

An important testing feature is visible inside applications declarations: the

application `app0` (*line 17*) has `source` set to `file`, that contains the path from which the scheduler must load the application. The application `app1` (*line 22*) instead has the field `source` equal to `generator`: this means that the application will be runtime generated following specifications written into the `config` file.

4.10 PHARE Scheduler Evaluation

4.10.1 Features

Additional features have been implemented, allowing PHARE to operate also in more realistic scenarios and specify desired behaviors:

- defined threshold both for computational and network resource usage to prevent the saturation of the infrastructure;
- defined shared network links between clusters;
- undefined number of resource types to evaluate (CPU, RAM, GPU, memory, ..);
- defined maximum dimension of an application component to be scheduled;

The threshold is a value between 0 and 1, and it aims to add a limit to the amount of resources that can be used on the referred cluster. Application components with a placement constraint on a specific cluster can exceed the cluster threshold and, if there is enough room for the component, be scheduled. The scenario of threshold exceed was designed for the opportunistic scheduling: a user that connects its device to the infrastructure can add a threshold to avoid its saturation due to other users jobs. The user can still add his jobs on his device until the device is full. Right now there is no policy that checks if a user can insert a constraint of an application or not, but it will be a future improvement.

The scheduler has a representation of each connection between clusters via a virtual link; in the physical world, each cluster can be connected to another cluster via a switch or a couple of routers, or lots of other solutions. Looking at fig. 4.1, $C1$, $C2$ and $C3$ are connected via a Switch; links between each cluster are treated by the scheduler as a single virtual link composed by a set of physical links (ref. fig. 4.2).

PHARE scheduler is capable to make its scheduling decision with a the number of labels which is virtually without limit. The actual version of PHARE scheduler has a limited set of labels (CPU, RAM, bandwidth), but with some small modifications,

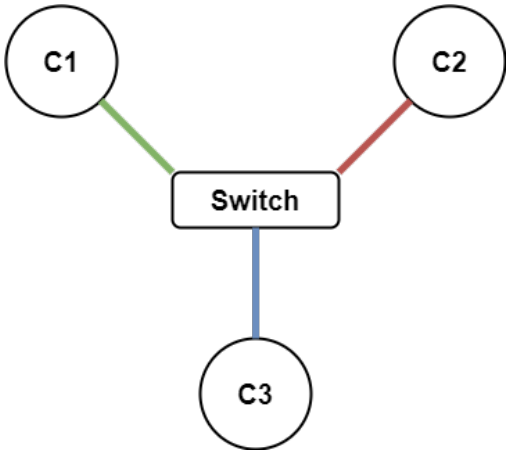

Figure 4.1: Physical links

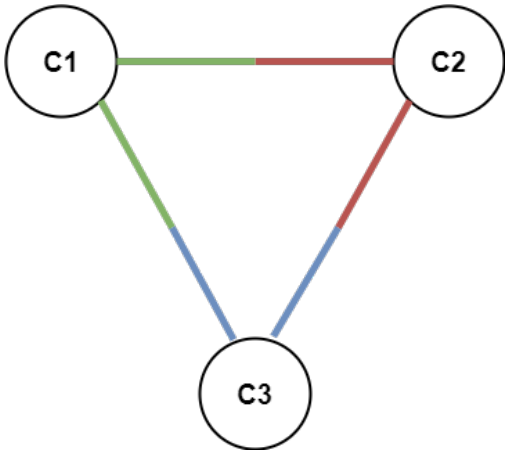

Figure 4.2: Virtual links

the scheduler user can add its own custom labels. Due to some implementations restrictions, only cluster labels can be multiple: only one label can be used on virtual links.

Each cluster is an opaque representation of one or more devices: the scheduler knows only the overall resource amount of a cluster and doesn't have any information about its devices specifications. A cluster administrator can add an information on his cluster that specifies the maximum amount of resources of an application node. This element is called `maxUnit`. This field can be set as the smallest amount of resources held by a device.

4.10.2 Limitations

As was already mentioned in above sections, the PHARE scheduler relies on recursion. The main limitation of the scheduler is that each level of recursion increments the memory used by the program. Since the depth of recursion is driven by the number of nodes inside an application, if this number is big enough it can produce some unexpected behaviours.

Another limitation of the scheduler is, again, related to the recursion. A possible way that was considered to speed up the scheduling phase was to parallelize the execution of the program; this was not possible because, when the scheduler recurs, there is no way to know if the path that it took will lead to a feasible solution or not. A possible way to speed up the execution of the scheduler is to launch a different instance of the program for each incoming application. This is only a theoretical solution and, at the time of writing, has not been tested.

The PHARE scheduler is currently using multiple resource types only on nodes (both applications and infrastructure ones) and not on links. The only other resource type different from "bandwidth" that could be useful on a link is the "latency"[2], but this resource is difficult to manage: first of all, differently from other resource types, the latency is a constant value which never changes during the scheduling life-cycle (the bandwidth will be decreased every time the link is used). Moreover each resource type must be comparable with another resource type: for example the scheduler considers equal 1 $core_{CPU}$ and 10 GB_{RAM} but it's not clear how to compare $seconds_{latency}$ with $Mb_{bandwidth}$.

4.10.3 Performance Improvements

Many optimizations have been performed in order to improve the performance of PHARE. Code profiling has definitely helped to identify critical code sections throughout the execution and, as many values slowly change between subsequent recursive calls, the implementation strongly relies on caching to prevent unnecessary redundant computations.

The heuristic-based PHARE scheduler has been designed to identify a-priori the most promising steps in order to speed up the scheduling process, but still the worst case complexity can be estimated as $\mathcal{O}(N * J)$, where N is the number of clusters of the infrastructure and J represent the number of components of the application. Such worst case scenario requires the simultaneous occurrence of numerous factors including: huge application size and huge, almost-saturated infrastructures. Although such scenario is theoretically possible, Cloud Providers usually prevent the saturation of their infrastructure for resilience reasons; still the worst case complexity can be reduced to $\mathcal{O}(M * J)$, selecting only a subset M of the N feasible clusters, with $M = 10$, since empirical evaluations have shown that the scheduling solution is always found within the first M clusters. Consequently the sorting algorithm used to rank the most promising clusters can be improved as well: the complexity of the Heap Sort $\mathcal{O}(N \log N)$ can be reduced using a modified version of the Selection Sort only for the first M clusters with complexity $\mathcal{O}(N)$ as $M << N$.

In order to find which are the critical parts of PHARE scheduler code, the Golang profiler `pprof`[3] was used: it produces a set of different graphs from which the developer can retrieve information about the performance of each function. The most analysed graph is the *Flame Graph*: this graph shows the percentage

[2]latency is the period of time used by a packet to traverse the link

[3]https://pkg.go.dev/net/http/pprof

and the absolute time usage of each function in the code. An extract of the PHARE scheduler Flame Graph is reported in fig. 4.3. Each line is referred to a function, lines below a line show functions called inside the above one. The width of a line is directly proportional with the function execution Time. From this graph is clear that the `reflection`[4] is the bottleneck of the `NewLabel` function (`reflect.<something>` functions). Most of the modularity of the code is achieved using this concept, and clearly it doesn't comes for free. In order to make the scheduler ready to go in production, there must be a refactoring process to speed-up this phase or reduce the modularity in favor of performance.

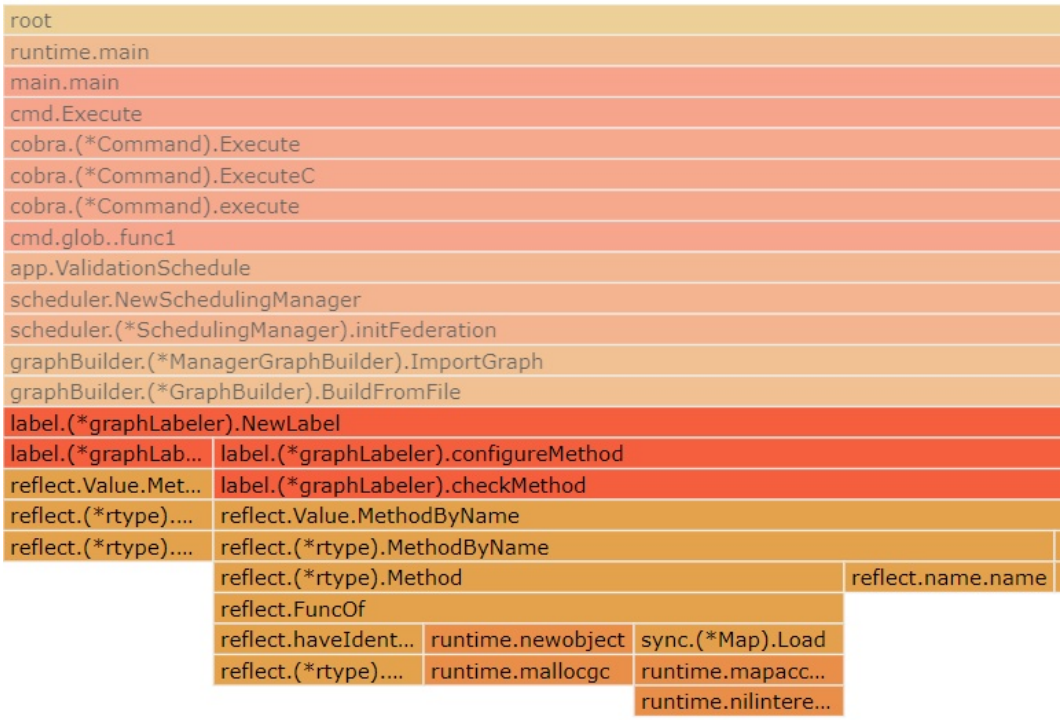

Figure 4.3: Code Flame Graph

Furthermore, pprof expose also a table called *Top Table* (ref. fig. 4.4) which has the list of functions sorted from the one with the longest cpu time usage to one with the shortest. From this table is visible that also maps manipulations are operations which slow down the scheduler, to avoid this issue, non-map types, such as slices, have been used where suitable.

[4]https://pkg.go.dev/reflect

Flat	Flat%	Sum%	Cum	Cum%	Name
0.09s	5.36%	5.36%	0.15s	8.93%	runtime.mapaccess1_faststr
0.09s	5.36%	10.71%	0.09s	5.36%	runtime.futex
0.07s	4.17%	14.88%	0.15s	8.93%	runtime.scanobject
0.06s	3.57%	18.45%	0.30s	17.86%	runtime.mallocgc
0.05s	2.98%	21.43%	0.11s	6.55%	runtime.mapassign_faststr
0.05s	2.98%	24.40%	0.08s	4.76%	runtime.mapaccess2_faststr
0.05s	2.98%	27.38%	0.08s	4.76%	runtime.heapBitsSetType
0.05s	2.98%	30.36%	0.05s	2.98%	memeqbody
0.04s	2.38%	32.74%	0.04s	2.38%	runtime.nextFreeFast
0.04s	2.38%	35.12%	0.04s	2.38%	runtime.memclrNoHeapPointers
0.04s	2.38%	37.50%	0.04s	2.38%	runtime.markBits.isMarked
0.04s	2.38%	39.88%	0.05s	2.98%	runtime.mapiternext

Figure 4.4: Code Top Table

Chapter 5

Validation and Testing

5.1 Experimental Results

This section shows the experimental validation of PHARE, under different parameters such as the scheduling success rate (sections 5.1.1 and 5.1.2), scheduling time (either in presence of abundant or scarce resources) (sections 5.1.1 and 5.1.2), the cost of the identified solution (sections 5.1.1 and 5.1.2), and scalability properties in terms of infrastructure size (section 5.1.1) and number of applications (section 5.1.2). Results are compared against Firmament, a state-of-the-art scheduler and one of the few ones available in Kubernetes.

In the simulations are replicated random infrastructure topologies of different sizes with the number of clusters ranging between 50 and 1000. Each single cluster of the infrastructure features a given random amount of CPU cores and is connected with all the other clusters in the infrastructure (i.e. full mesh topology) with virtual connections characterized by the value of available network bandwidth. Moreover each resource features a given cost expressed in $\$/unit$, correctly sized to match major Cloud Provider resource costs (table 5.1 summarizes the main configuration values of the infrastructures). In order to produce repeatable tests, the configuration file of applications and infrastructures contains the seed for the random generator and *max-min-boundaries* of each random value.

It was selected a sample 10-tier microservices application called Online Boutique [7] for the simulated workload, monitoring the resource demands of the application both in term of CPU and bandwidth usage, allowing us to later use those information to correctly define the workload. A set of 500 applications was randomly generated, each consisting of 10 microservices dimensioned according to table 5.2.

Ultimately it was evaluated how the communication-aware placement of PHARE

impacts the network congestion within the infrastructure by performing some further set of tests, focusing the evaluation on the bandwidth consumption of the scheduling solutions (section 5.1.3).

To cope with the randomness of both the infrastructures and the simulated workload all the simulations were replicated 20 times (`simulation samples`) with different configurations.

Infrastructure size:	50 — 1000 clusters
Number of vCPU per cluster:	300 — 500 cores
Cost of vCPU per cluster:	0.15 — 0.4 $\$/core$
Amount of RAM per cluster:	128 — 512 GB
Cost of RAM per cluster:	0.01 — 0.06 $\$/GB$
Upstream bandwidth per cluster:	5 — 40 Gb/s
Cost upstream bandwidth per cluster:	0.01 — 0.4 $\$/Gb$

Table 5.1: Infrastructure setup.

Number of deployed applications:	500
Application size:	10 microservices
Microservice CPU requests:	4 — 20 vCPU
Microservice RAM requests:	1 — 2 GB
Microservice connectivity ratio:	30%
Microservice bandwidth requirements:	0.6 — 20 Mb/s

Table 5.2: Workload setup.

5.1.1 Scalability on Infrastructure Size

Now, the focus is on the evaluation of horizontal scaling on the infrastructure size. Specifically the objective of the tests is to deploy PHARE with infrastructures of variable sizes, in order to understand the performance: 1) with few (possibly saturated) clusters with very limited scheduling solutions 2) with multiple clusters (up to 1000 for the study case) and with many possible scheduling solutions to be evaluated.

Deployment success rate

fig. 5.1 represents the percentage of successfully scheduled applications out of the complete set; one application is accounted as scheduled only if all its microservices have been successfully scheduled. Small infrastructures do not have enough resources to accommodate all the applications, hence infrastructures with less than 200 clusters experience a success rate $< 100\%$. On the other hand infrastructures with more than 200 clusters can host all the applications, so the success rate is always 100%.

There are no visible differences between the success rate with PHARE and Firmament, meaning that both the solutions are able to correctly consolidate the application deployments even with small infrastructures.

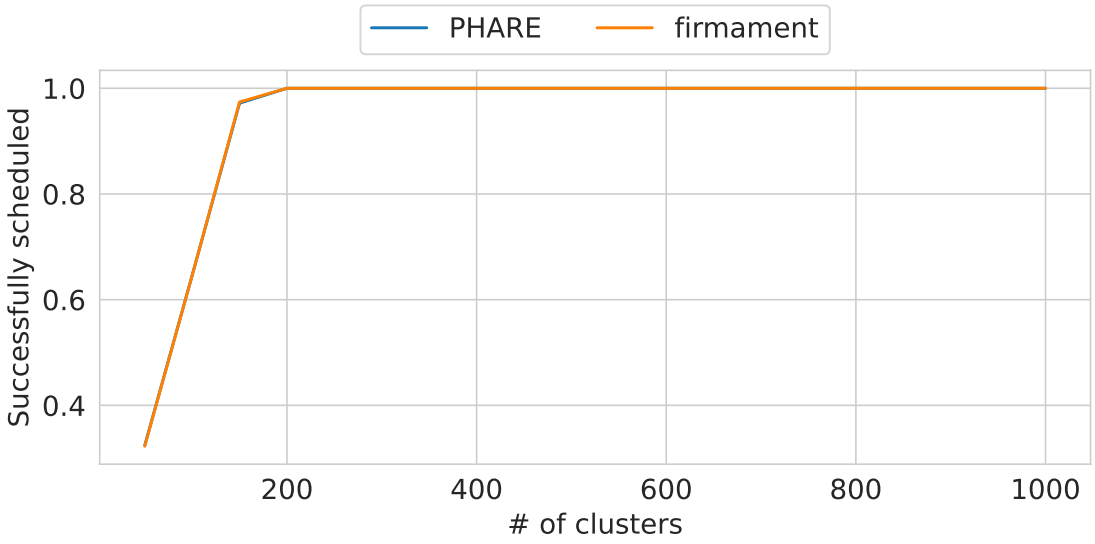

Figure 5.1: Scheduling success rate

Scheduling time evaluation

fig. 5.2 represents the average scheduling time needed to place each of the 500 applications (the time needed to identify unfeasible scheduling placement is accounted as well); PHARE outperforms the scheduling time required by Firmament, being able to identify suitable placement always within $80ms$ even for large infrastructures.

Additionally PHARE scheduling time not only experiences a linear trend with the size of the infrastructure, but also the variance of the distribution across different simulation samples is very limited (see fig. 5.3); this means that the

heuristic is always able to identify the most promising steps in order to speed up the scheduling process. Conversely, Firmament experiences very high response time especially on small infrastructures mainly because it has been designed to consider an application scheduled even if only a subset of the microservices has been successfully scheduled, resulting in multiple calls to the solving algorithm.

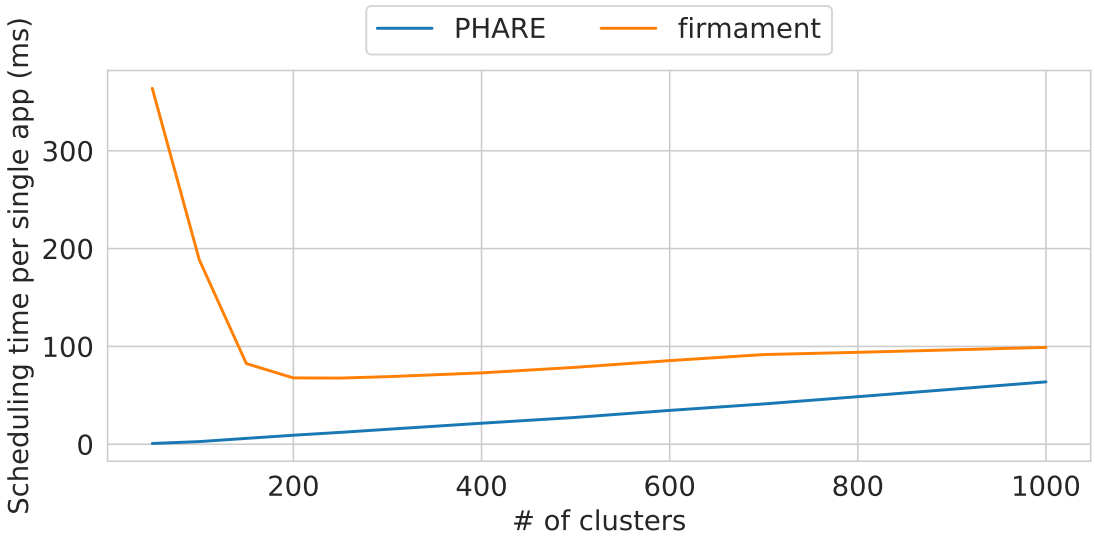

Figure 5.2: Application scheduling time

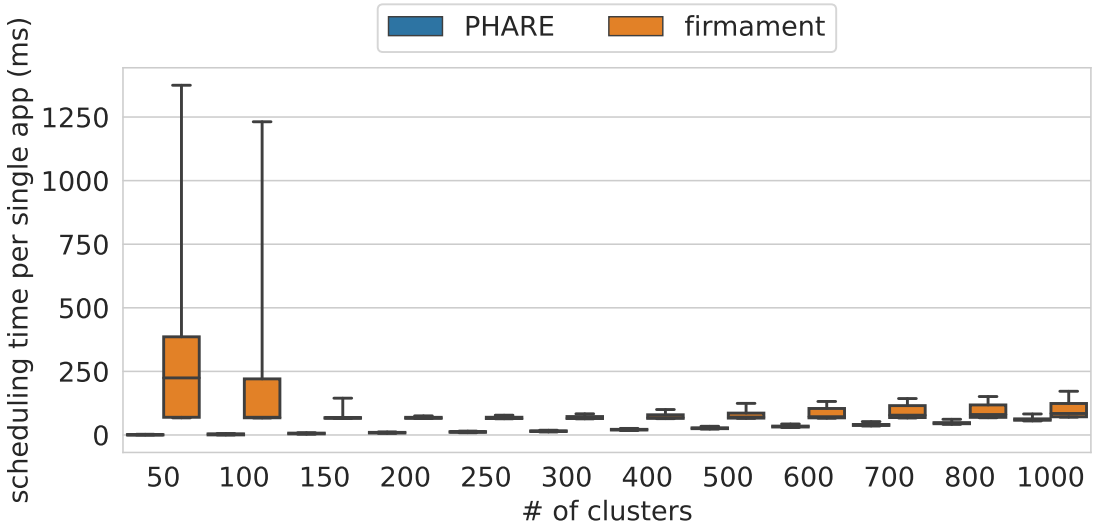

Figure 5.3: Application scheduling time box

Deployment cost evaluation

Previous results could raise the question on whether the smallest scheduling time requested by PHARE translates to worst solutions in term of experienced cost. Figure 5.4 represents the cumulative deployment cost of all the applications; not-scheduled applications contribute with $cost = 0$ to the final deployment cost. Three phases can be identified in the perceived cost throughout the test: 1) infrastructures with few clusters (< 150) experience a rapid growth in the applications deployment cost given that the scheduler has more and more available resources to accommodate the applications 2) infrastructures with more than 150 clusters have enough resources to schedule all the application, hence the total deployment cost starts to drop as the number of clusters grows 3) in case of infrastructures with more than 600 clusters the scheduler is not able to reduce even more the deployment cost because even if there are more possible solutions, they are all very similar in term of cost, given the bounded randomness of the infrastructure parameters.

The target placements both for PHARE and Firmament experience the same perceived cost, meaning that even if fewer steps are considered by PHARE, the heuristic succeed to identify the most promising one first.

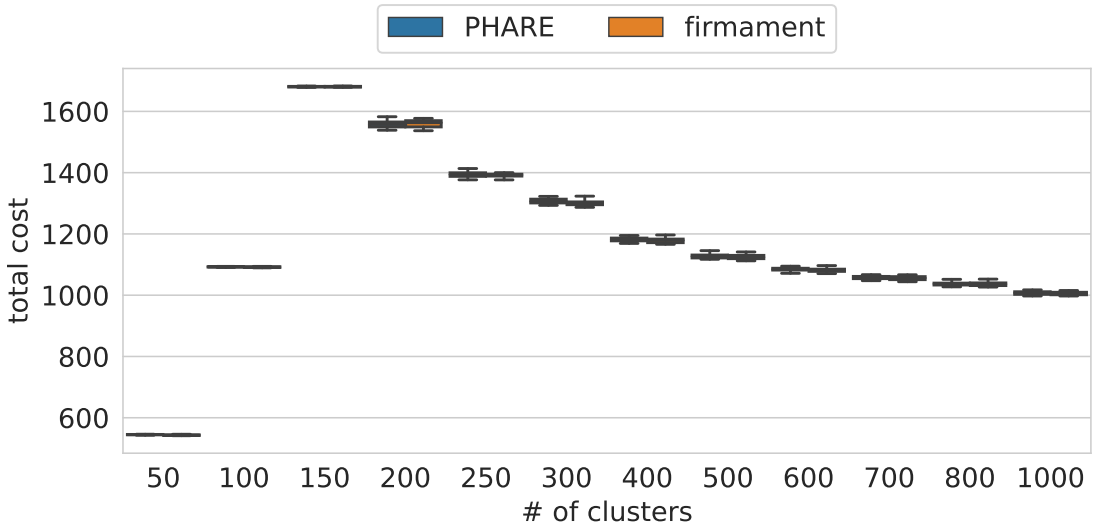

Figure 5.4: Federation deployment cost box

5.1.2 Scalability on Number of Applications

In the previous set of tests were identified the 100-cluster infrastructure as one of the most challenging one, given the limited amount of available resources to

fulfill the demands of all the applications; hence analyzing closely such scenario it is possible to understand the performance of PHARE both when the infrastructure is almost saturated and the possible solutions are very limited, but also when there are no more available resources in the infrastructure and the algorithm must quickly identify such situations.

Deployment success rate

fig. 5.5 depicts the scheduling success rate for the complete set of applications; one application is accounted as scheduled only if all its microservices have been successfully scheduled. The infrastructure does not have enough resources to accommodate all the applications, in fact only the first 325 applications can always be scheduled across all the simulation samples. PHARE is able to achieve some slight better performance, scheduling some few more applications with respect to Firmament thanks to a more accurate management of the available resources.

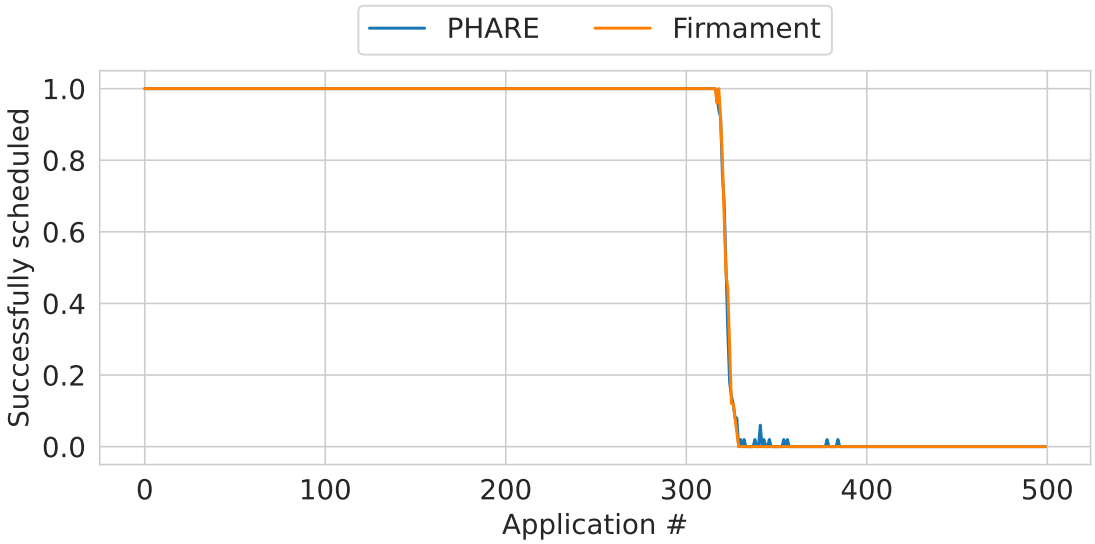

Figure 5.5: Scheduling success rate

Scheduling time evaluation

figs. 5.6 and 5.7 represent the scheduling time for the i_{th} application. As in the previous section PHARE outperforms Firmament scheduling time, being able to converge to a suitable scheduling solution not only in complex almost-saturated infrastructures, but also in detecting unfeasible solutions when the amount of available resources are not enough to host the remaining applications;

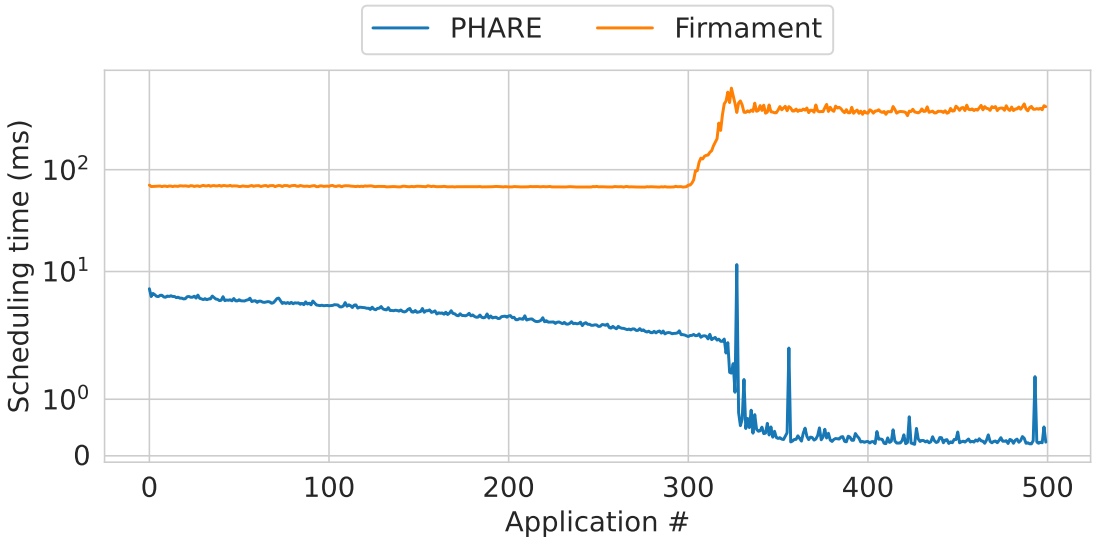

Figure 5.6: Scheduling time

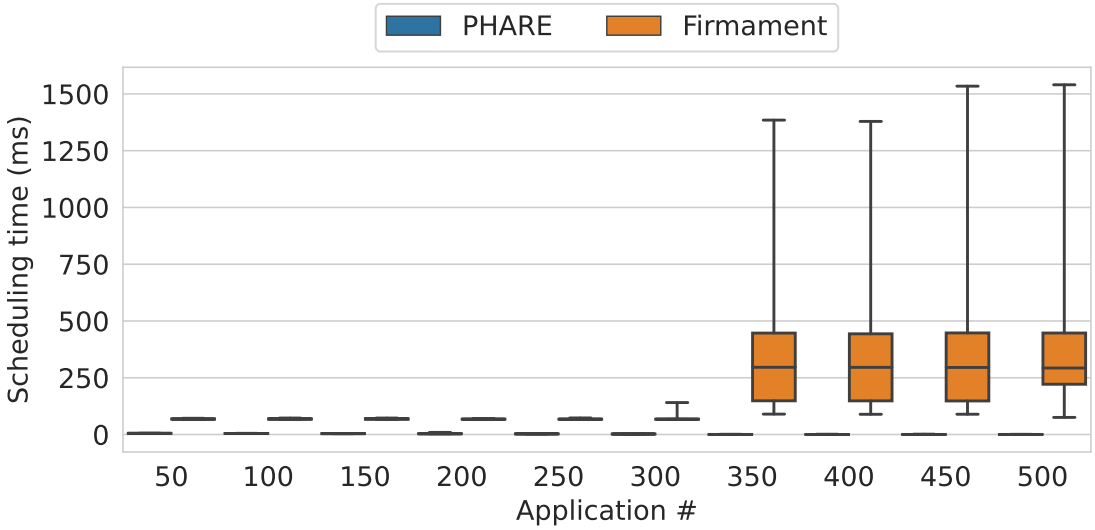

Figure 5.7: Scheduling time box

Deployment cost evaluation

fig. 5.8 represent the average deployment cost of the i_{th} application. Throughout the test it was experienced a linear increase of the deployment cost, motivated by the reduction of the number of possible scheduling solutions, until the saturation of the infrastructure was reached.

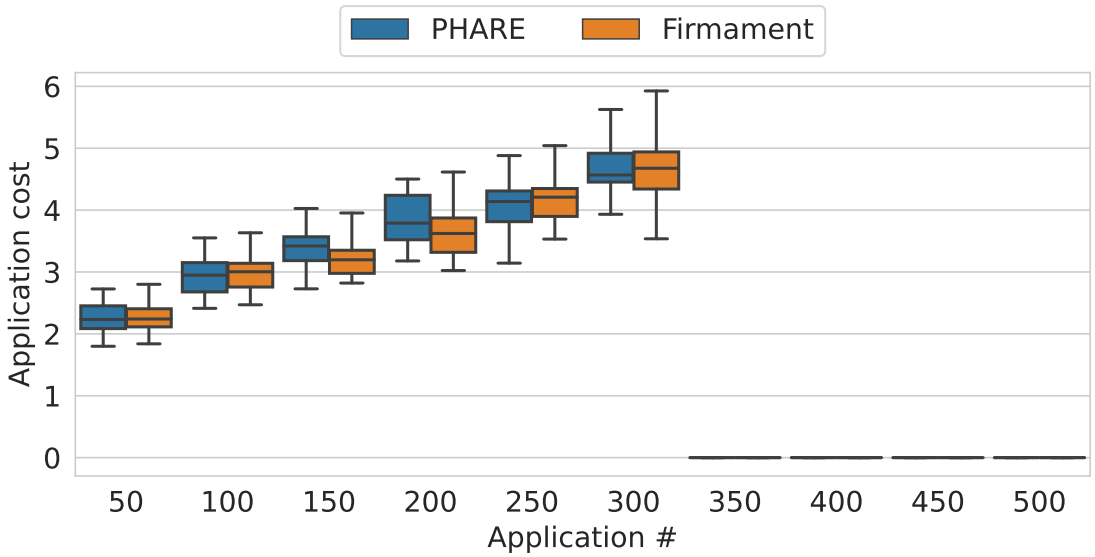

Figure 5.8: Deployment cost box

5.1.3 Bandwidth Consumption

Now it is evaluated how much the infrastructure can benefit from the communication-aware placement of PHARE, thus focusing especially on the overall bandwidth consumption of the links between clusters.

For this specific set of tests it was used a 60-cluster infrastructure (dimensioned according to table 5.1) and the workload that describes applications was slightly modified, increasing by a factor of 100 only the amount of bandwidth requirements for each microservice-to-microservice communication, hence increasing conceptually the amount of data transferred throughout the execution of the application itself. Furthermore some additional resource usage thresholds of 75% were introduced both for computational resources (CPU and RAM) and for network resources (bandwidth) to limit the amount of available resources, replicating a realistic scenario for Cloud Providers.

figs. 5.9 and 5.10 represent the percentage of allocated resources in the clusters, sorted in descending order. As expected PHARE tends to group as much as possible the microservices belonging to the same application onto the same clusters, avoiding the intra-microservice communication to span across different clusters, whereas Firmament prefers to spread more evenly the workload across the federation.

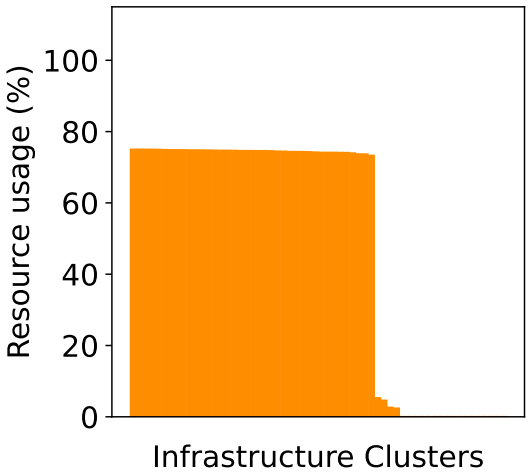

Figure 5.9: PHARE resource usage

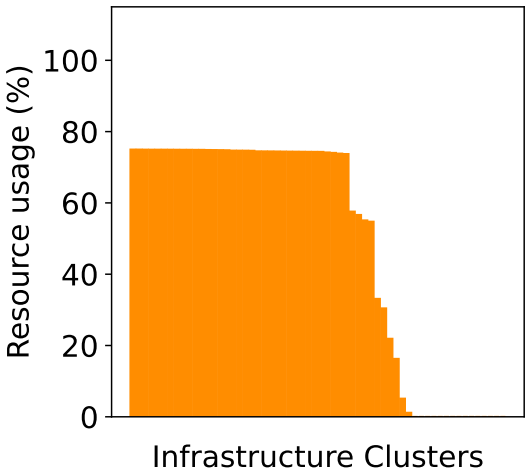

Figure 5.10: Firmament resource usage

As a result these scheduling decisions reflect also on the network resource usage within the federation. figs. 5.11 and 5.12 depict the resulting network pressure on the links between clusters; more specifically they represent the congestion matrix $N * N$, where N is the number of cluster of the federation and each cell (i, j) represents the percentage of network bandwidth usage between cluster i and j. Moreover each cell value is represented on a grey-scale color code, except the ones in red that exceed 100% network usage (i.e. the target placement require more bandwidth than the available amount). It's shown that the network-aware placement of PHARE is able to distribute accurately the applications across the federation, aggregating as much as possible the microservices of the same application within the same clusters; on the other hand Firmament is not aware of the communication between microservices thus the outcome of the scheduling process is nearly randomic for what concerns network consumption.

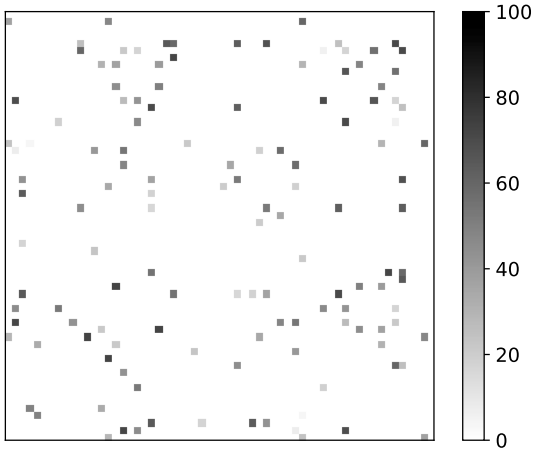

Figure 5.11: PHARE bandwidth usage

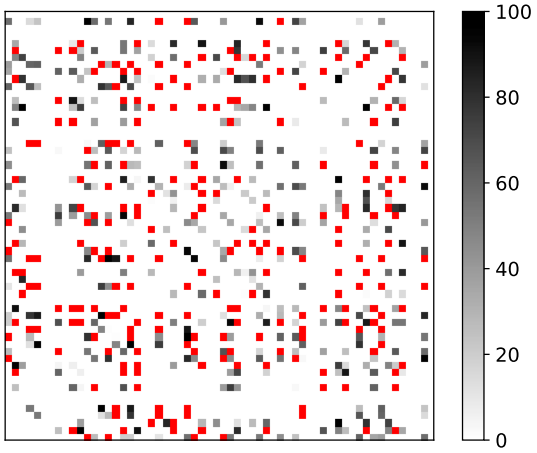

Figure 5.12: Firmament bandwidth usage

5.1.4 Resource Usage Distribution

General Scenario

After testing the performance of the scheduler, we decided to further investigate how it consumes its target infrastructure of available nodes, focusing on the topological resource distribution. The goal is to affirm that the scheduler, trying to keep the cost of the communications between pods to a minimum, aggregates the microservices of an application as much as possible, avoiding to spread them across the infrastructure.

To test this aspect we decided to schedule a number of toy-applications, based on the microservices demo Kubernetes deployments proposed by Google, on a simple infrastructure composed by equal nodes. The infrastructure has 50 nodes, each can host up to 700 cores of cpu-usage; the nodes are in a full mesh network, each link has 20 Gbps bandwidth availability. This infrastructure is designed to provide a non-challenging scenario to the scheduler.

To have a clear view on the evolution of the scheduling we took 4 snapshots of the state of the infrastructure, each after scheduling 20 applications. The following charts display the percentage resource usage of the nodes in the infrastructure.

The results confirm the desired behavior of the scheduler. Considering the snapshots one by one it is possible to note that, in each chart, only a single node is not completely full, this means that the scheduler tries to fill a node as much as possible before starting to fill the next one. This is achieved thanks to the sorting of nodes based on the nodesPriority mentioned in section 3.3.5. The trends keeps

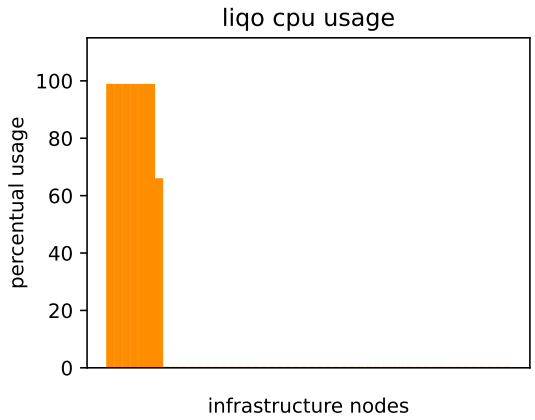

Figure 5.13: 20 apps scheduled

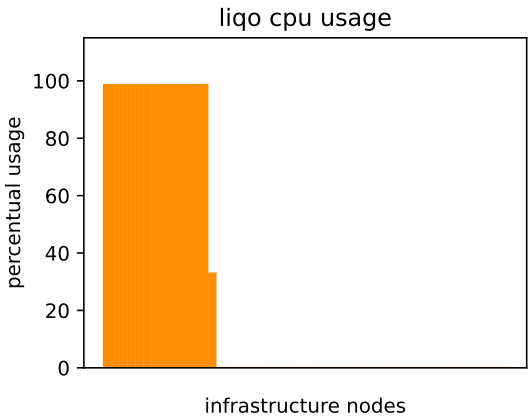

Figure 5.14: 40 apps scheduled

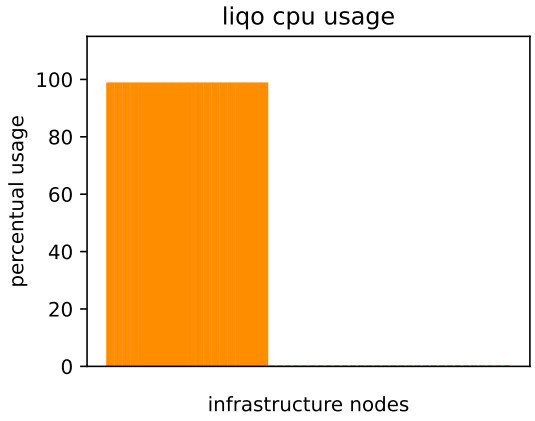

Figure 5.15: 60 apps scheduled

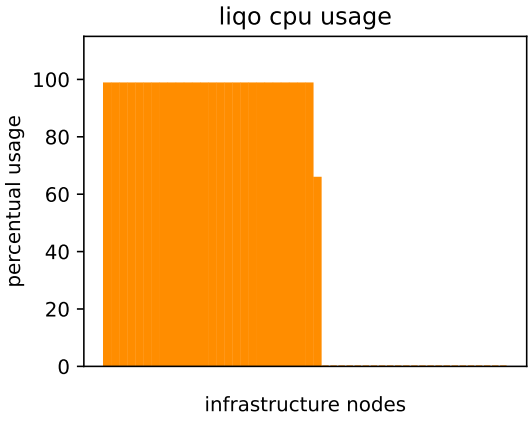

Figure 5.16: 80 apps scheduled

the same when considering subsequent snapshots in the scheduling process.

Opportunistic scenario

The results obtained in the above described test, led us to investigate the same metrics on a more challenging scenario. While the last infrastructure could represent a set of datacenters connected together in a full mesh network, the following is going to test the scheduler in an "opportunistic" scenario [8]. With the term "opportunistic" we intend an infrastructure composed by a large set of "small nodes", e.g. small workstations in a university laboratory. This is a scenario where a single application can access a lot of resources distributed across multiple "small" nodes. Each of the nodes has 8 cores of available CPU. We also added 2 other "big"

nodes, representing two clusters of servers, each with 100 CPU cores; this scenario could be achieved using the Liqo framework.

We scheduled 80 toy-applications. Each app is composed by 3 microservices in a chain topology (A-B-C). In table 5.3 are described the requirements of the toy-application. The microservice A of the application has a `nodeSelector` that forces the pod to be placed on a specific "small" node of the infrastructure. Among the 80 applications scheduled, the `nodeSelector` on pod A is the same every 20 applications.

Microservice ID:	CPU request	nodeSelector
A	1	Yes
B	1	None
C	4	None

Table 5.3: Application requirements.

For the test we took 4 snapshots of the resource usage of the infrastructure every 20 app scheduled. In fig. 5.21 is possible to see that not all 80 apps are scheduled, the scheduler can not place the last set of applications. Looking at the resource distribution is possible to see that the small nodes get filled up by the first 60 applications, thus when needing to scheduled the last 20, the scheduler cannot satisfy the `nodeSelector` constraint of pod A of each application. This behaviour is a consequence of the placement of C pod of previous applications, those pods take up a lot of the available space of "small" nodes, including those targeted by some `nodeSelector` thus making it impossible to fit other A nodes, which require little CPU cores but are bound to specific nodes.

This scenario shows a potential drawback of the aggregating tendency of the PHARE scheduler, i.e. if the scheduler would have placed some C pods on other unused nodes, following a more distributed approach, all 80 applications would have been scheduled. Although this solution would have certainly increased the total cost of the deployment, given the higher amount of bandwidth consumed.

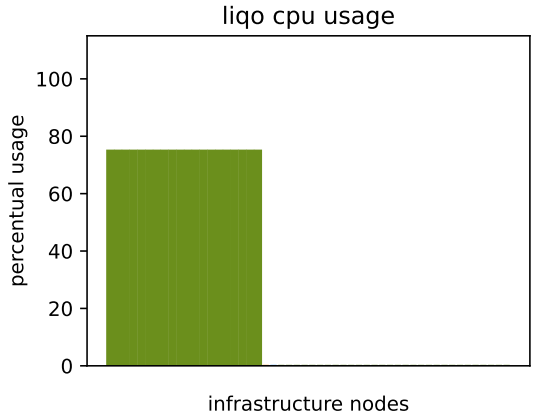

Figure 5.17: Cpu usage after 20 scheduled apps

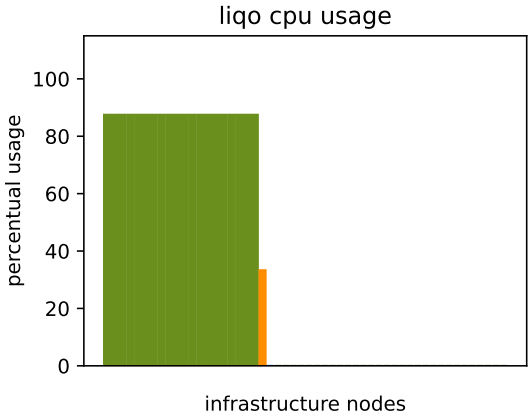

Figure 5.18: Cpu usage after 40 scheduled apps

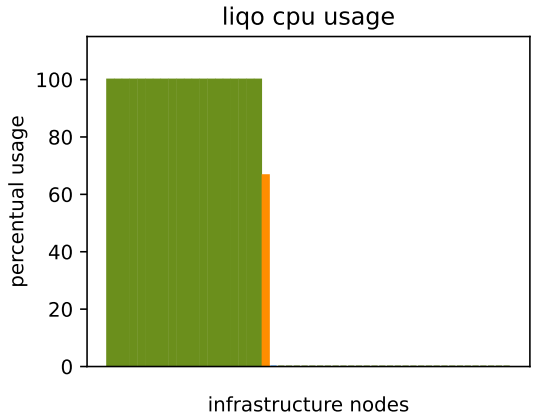

Figure 5.19: Cpu usage after 60 scheduled apps

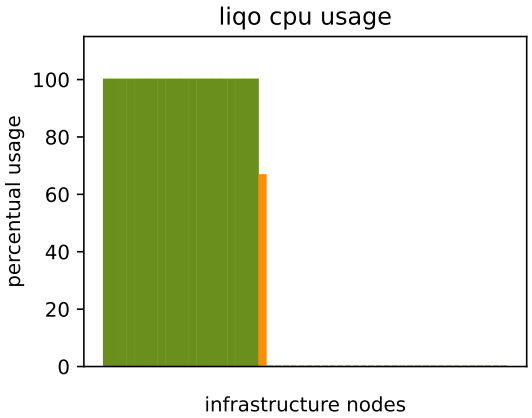

Figure 5.20: Cpu usage after 80 scheduled apps

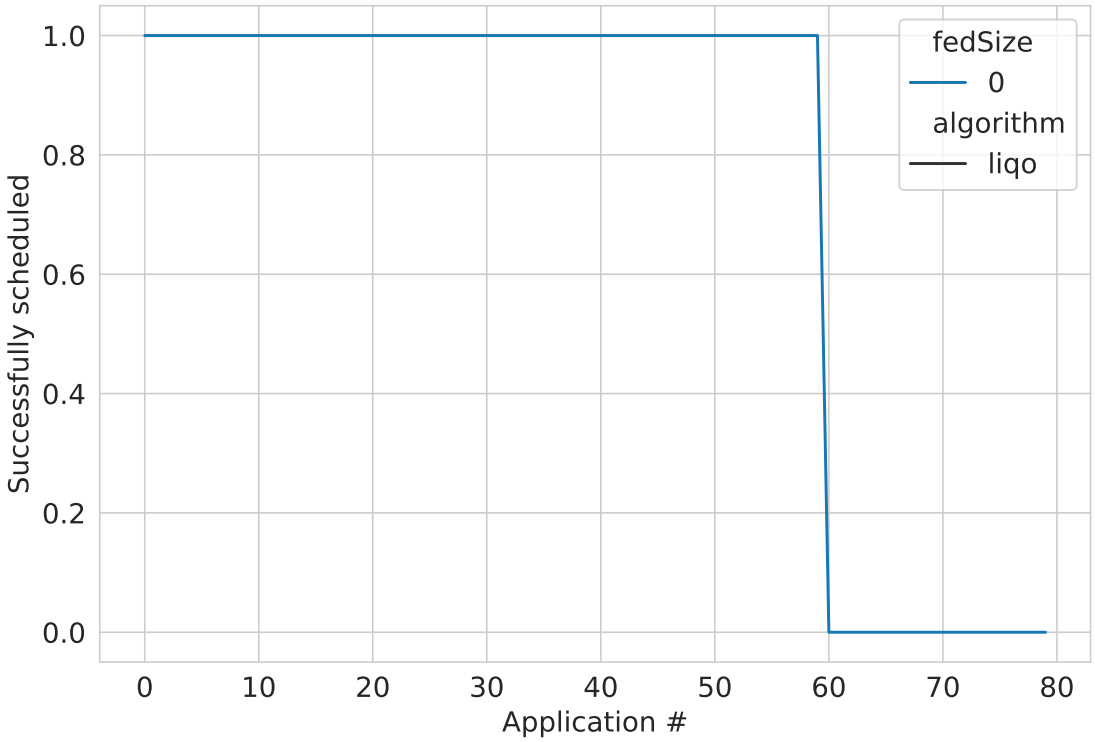

Figure 5.21: Successful scheduling

Opportunistic scenario optimized

This section analyses the test results of a possible implementation to fix the scheduling behavior of PHARE in the last described test scenario.

To accomodate the potential incompatibility between the aggregating tendency of the scheduler and the use of node-selecting policies on pods, we thought of combining the use of the `nodeSelector` and the `threshold` properties of application and infrastructure nodes. The new scheduling behaviour considers the resource threshold of a node depending of the `nodeSelector` relationship between a pod and a node. When examining if a node can host a pod, if the pod has a `nodeSelector` that targets a certain node, the scheduler does not consider resource threshold. This means that a certain pod could be perceived more or less resourceful depending on the pod perspective.

After implementing this behaviour we run again the previously described test and the result were positive. In fig. 5.26 is possible to note that the scheduler managed to place all 80 applications. The resource usage charts show that the placement kept the aggregating tendency without using unnecessary resources.

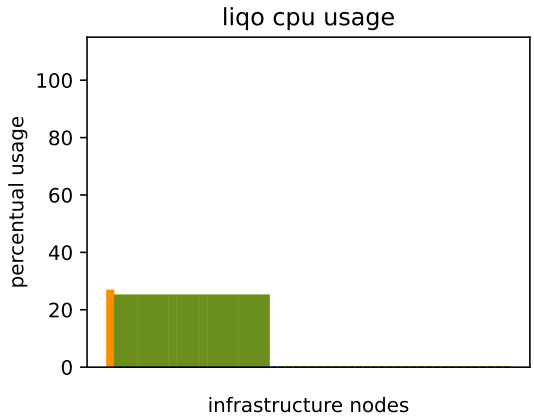

Figure 5.22: Cpu usage after 20 scheduled apps

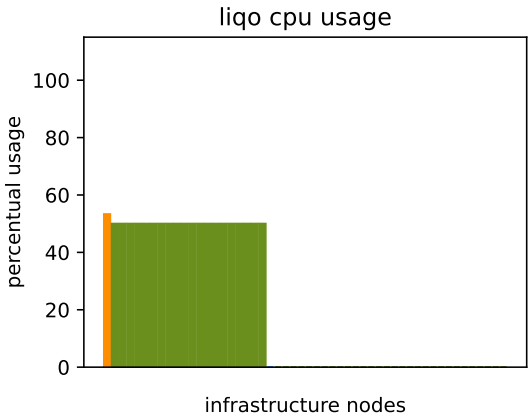

Figure 5.23: Cpu usage after 40 scheduled apps

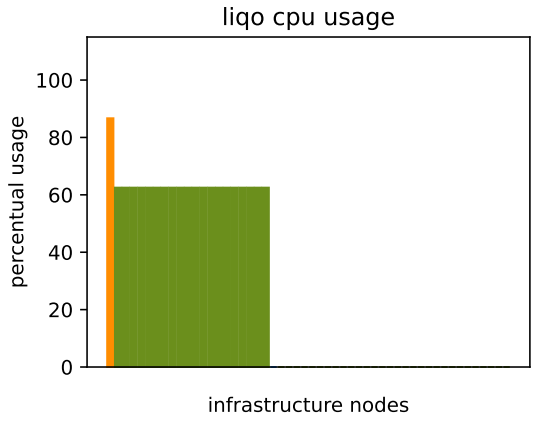

Figure 5.24: Cpu usage after 60 scheduled apps

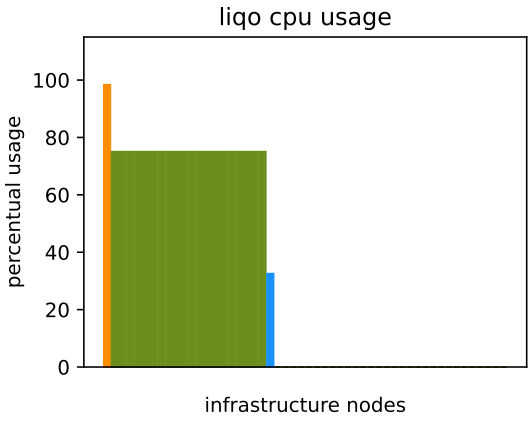

Figure 5.25: Cpu usage after 80 scheduled apps

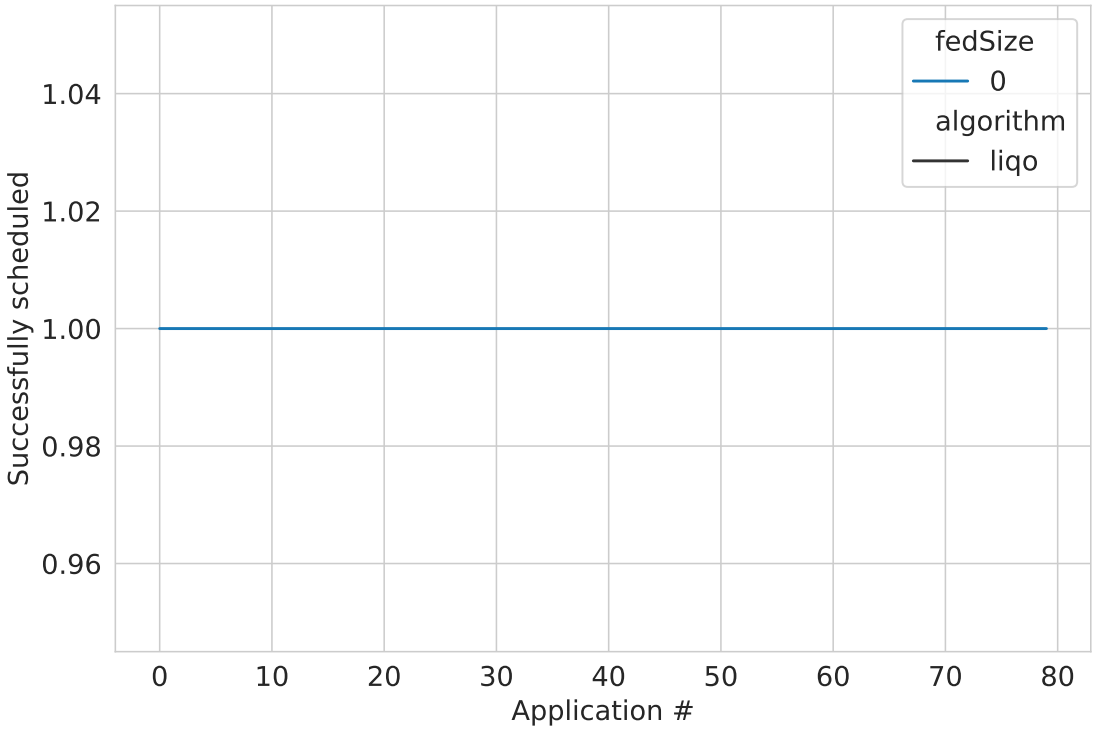

Figure 5.26: Successful scheduling

Chapter 6

Conclusions and Future Works

The thesis aims at producing a scheduler that can be used alongside with the Liqo framework on a multi-cluster/multi-provider infrastructure. The main features of the current version are:

- scheduling decisions take into account application resources, bandwidth constraints and cluster costs. The final decision is a trade off between these parameters resulting in a performing solution at the lowest price;
- the ability to adopt specific pod-to-node selection policies to allow for better placement in more articulated scenarios;
- the liberty left to the user to precisely customize the type and the handling of the resource assigned to each component or node, allowing to represent different scenario with various requirements and constraints.

The PHARE scheduler, at the time when the thesis was written, is efficient and can find a solution fast. To further improve the solution, the following modification ca be added:

- the high level of customization offered by the chosen implementation has a non-negligible impact on performances. In future versions the scheduler customization could be limited creating a more compact framework that can reach higher performances.
- every time the scheduler starts a recursive path, is impossible to know apriori if a solution will be found or not. A possible improvement can be the parallelization of the recursive section by exploring different paths of the recursion

tree at the same time, hoping that at least one of them returns a feasible solution for the application.

- the current way of computing costs for pod placement doesn't reflect real world scenarios, where the cost is usually not computed in this way: the cost is commonly relative to the number of active nodes that hosts the application. The Liqo team is currently working to produce a cost-oracle which will be interrogated by the scheduler to retrieve the cost of each component of the application.

- the current scheduling "policy" is to use the least amount of infrastrucute nodes. A possible future work could be the implementation of other policies which, for example, to split the application across the infrastructure.

Finally, the scheduler next step is to integrate the PHARE scheduler into Kubernetes in order to test it in a real multi-cluster environment and collect information to improve scheduler performances.

Bibliography

[1] *Liqo Official Website*. URL: https://liqo.io/ (cit. on p. 2).

[2] *Kubernetes Official Website*. URL: https://kubernetes.io/ (cit. on p. 2).

[3] Ionel Gog, Malte Schwarzkopf, Adam Gleave, Robert N. M. Watson, and Steven Hand. «Firmament: Fast, Centralized Cluster Scheduling at Scale». In: *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. Savannah, GA: USENIX Association, Nov. 2016, pp. 99–115. ISBN: 978-1-931971-33-1. URL: https://www.usenix.org/conference/osdi16/technical-sessions/presentation/gog (cit. on p. 9).

[4] Deepak Vij and Shivram Shrivastava. *Poseidon-Firmament Scheduler – Flow Network Graph Based Scheduler*. https://github.com/kubernetes-sigs/poseidon. Accessed: 2021-08-23 (cit. on p. 10).

[5] *Golang Official Website*. URL: https://golang.org/ (cit. on p. 28).

[6] Mattia Lavacca. «Scheduling Jobs on Federation of Kubernetes Clusters». In: (2020). URL: https://webthesis.biblio.polito.it/14522/ (cit. on p. 28).

[7] *Online Boutique Demo*. URL: https://github.com/GoogleCloudPlatform/microservices-demo (cit. on p. 44).

[8] Malcolm Atkinson Gary A. McGilvary Adam Barker. «Ad Hoc Cloud Computing». In: *2015 IEEE 8th International Conference on Cloud Computing* (2015). URL: https://ieeexplore.ieee.org/document/7214163 (cit. on p. 54).