Author copy of paper published at *2021 IEEE 14th International Conference on Cloud Computing (CLOUD)*
©2021 IEEE. Official publication: https://doi.org/10.1109/CLOUD53861.2021.00085

Pogonip: Scheduling Asynchronous Applications on the Edge

Thomas Pusztai*
Distributed Systems Group
TU Wien
Vienna, Austria
t.pusztai@dsg.tuwien.ac.at

Fabiana Rossi*
DICII
University of Rome Tor Vergata
Rome, Italy
f.rossi@ing.uniroma2.it

Schahram Dustdar
Distributed Systems Group
TU Wien
Vienna, Austria
dustdar@dsg.tuwien.ac.at

***Abstract*—The microservice architectural style is changing the design of modern applications. Orchestration tools, such as Kubernetes, deploy them on computing nodes assuming that resources are interconnected through fast communication links. However, running microservices in the emerging edge computing environments requires considering the heterogeneity and non-negligible network delays among edge resources. In this context, although the problem of scheduling synchronous microservice-based applications has been widely explored, scheduling asynchronous applications, where microservices interact using a queue system, has only recently started to be investigated. In this paper, we present Pogonip, an edge-aware scheduler for Kubernetes, designed for asynchronous microservices. We formulate an optimization problem and a heuristic for determining the placement of microservices, which is tailored for edge environments. We integrate them in Kubernetes by building custom scheduler plugins. Using a benchmark application, we show the advantages of the proposed network-aware solutions over other state-of-the-art solutions.**

***Index Terms*—Edge, scheduler, placement problem, microservices, asynchronous, Kubernetes**

I. Introduction

Today's trend for designing efficient and scalable applications relies on the microservice architectural style [1]. It decomposes an application into autonomous and decoupled services, each with specific and independent functionalities. They are typically deployed separately to one another, especially resorting on software containers, which enable grouping a microservice with all its dependencies thus simplifying its deployment. Two main communication styles between microservices exist: synchronous and asynchronous. With synchronous communication, a microservice m_1 contacts a microservice m_2 directly and maintains the connection until it receives a response. This increases coupling and limits the application's flexibility to change. So, this is often considered an anti-pattern [2]. With asynchronous communication, m_1 and m_2 communicate through an indirection layer, e.g., a message queue [3]. This allows decoupling the application's microservices, improving scalability and flexibility.

In the last years, we are witnessing the diffusion of computing resources located at the edge of the network. Edge computing extends cloud computing by using computational capabilities of devices at the edge of the Internet [4]. This concept suits applications whose data are generated and consumed at the network periphery [5], but it brings new challenges, mainly due to the heterogeneity and decentralized distribution of computing and networking resources. In this context, the *placement (or scheduling) problem* is of utmost importance; it defines the mapping between the application microservices and the computing nodes. An improper node selection can negatively impact the application performance and, in a pay-per-use scenario, can lead to higher execution costs.

To simplify the deployment of microservice-based applications, we resort to orchestration tools like Kubernetes[1]. When a new application should be executed, Kubernetes uses a component, i.e., the *scheduler*, to solve the placement problem. Although Kubernetes is one of the most popular production-grade orchestrators [6], it has been originally designed for cluster environments, so it is not well suited to run applications on the edge. It does not consider that edge nodes may be connected with different link types (e.g., WiFi, 4G, 5G) that exhibit varying quality of service characteristics, such as latency [7]. Edge computing requires new placement strategies that explicitly take into account the presence of heterogeneous resources and non-negligible network delays. As surveyed in [8], [9], different solutions exist in literature. However, to the best of our knowledge, all of them consider only the placement of synchronous applications. Asynchronous applications exhibit peculiar features, like increased throughput, that cannot be neglected [10]. For example, in a smart mobility scenario, where cars and road-side devices report traffic and safety data to the analyzer microservices, a message queue significantly reduces coupling between the participating microservices allowing a non-blocking communication. In such applications, the queue represents a key component that should be allocated as close as possible to all microservices, as it is the logically centralized communication component.

In this paper, we present *Pogonip*, an edge-aware scheduler for Kubernetes, designed for asynchronous microservice-based applications. The main paper contributions are as follows:

1) We formulate the placement problem as an Integer Linear Programming (ILP) optimization problem. It places

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 871403.

[1] https://kubernetes.io

microservices by considering constraints on network latency towards the queue system on edge nodes. Moreover, if not enough edge resources are available, it can offload microservices to third-party cloud nodes, while keeping the additional costs low.

2) We define the Pogonip heuristic to quickly find an approximate solution for real-world execution scenarios, because the optimization problem is NP-hard.
3) We implement the heuristic as a scheduler prototype for Kubernetes and release it as open-source.

Using an asynchronous edge application and a Kubernetes cluster, we evaluate our solutions against two state of the art placement policies and the default Kubernetes scheduler.

The remainder of this paper is structured as follows. We first discuss related works (Section II), our system model, and the problem to solve (Section III). Then, we formulate the optimization placement problem (Section IV) and the Pogonip heuristic (Section V). In Section VI, we describe the heuristic integration in Kubernetes and, in Section VII, we present the experimental evaluation. Section VIII concludes the paper.

II. Related Work

In this section, we analyze existing approaches dealing with the placement of microservices on edge and fog computing resources. Many are applicable to both environments. Several frameworks for developing microservices have been originally designed to run on centralized cloud data centers (e.g., [11], [12]). While these represent interesting solutions, the differences between cloud and edge resources prevent their direct adoption in an edge environment. Different Internet connectivity and bandwidth, as well as resource distribution, call for strategies that explicitly take into account the presence of heterogeneous resources and non-negligible network delays. To organize and summarize the most relevant approaches, we present the key methodologies and orchestration tools used to allocate microservices in an edge/fog environment.

Existing placement policies rely on a wide set of methodologies, ranging from mathematical programming approaches to heuristics. The mathematical programming approaches exploit methods from operational research in order to solve the application placement problem (e.g., [13], [14]). For example, to save energy consumption, Huang et al. [13] model the mapping of IoT services on edge/fog devices as a quadratic programming problem. Although simplified into a linear formulation, it may require prohibitive resolution time when the problem size grows. The main drawback of mathematical programming solutions is scalability; the placement problem is well-known to be NP-hard, therefore, efficient heuristics are needed. Different heuristics have been proposed to solve the placement problem of applications, as surveyed in [8], [9]. However, in recent years, the most popular approaches resort to greedy heuristics and genetic algorithms (GAs). For example, Faticanti et al. [15] propose a throughput-aware approach that first partitions each application into two chunks, and then uses a greedy heuristic to allocate them on the available computing resources. While the first chunk is always executed in the cloud, the heuristic can allocate the second one either in the fog or in the cloud. Pallewatta et al. [16] consider fog nodes organized as a tree. When a fog node receives an application execution request, it uses a greedy heuristic to allocate microservice starting from the leaf nodes; if no resources are available, it can forward the request towards the parent node. This decentralized approach promises to reduce latency and network usage. To solve the edge/fog placement problem, different works (e.g., [17]–[20]) rely on GAs. For example, [20] introduces the concept of edge sites to decentralize the resolution of the microservices placement problem optimizing the application response time. Each site uses a GA to decide which microservices and how many instances of them to place in the current site as well as those to propagate to the neighbor sites. Even though GAs considerably reduce the average time to find a good solution, especially for a large solution space, they may react slowly to changes of an edge/fog computing environment. All previously mentioned policies consider the problem of scheduling synchronous applications. However, the recent tendency is to design asynchronous applications as well [21], where a (logically centralized) message queue supports the microservice communication. Although asynchronous applications are starting to be investigated in the context of load balancing [22], to the best of our knowledge, they are so far poorly explored in the context of service placement. Thus, we propose Pogonip, an approach for solving the placement problem of asynchronous microservice-based applications in edge and fog environments. Moreover, Pogonip can extend edge resources resorting on cloud nodes (e.g., as [15], [17] do). However, differently from the existing works, it allocates application components on cloud nodes aiming to minimize the incurred cost of using cloud resources.

When multiple application components should be executed, we usually resort on orchestration tools that automatize container provisioning, management, communication, and fault-tolerance. Although several orchestration tools exist [23], Kubernetes, Docker Swarm, and Apache Hadoop YARN are among the most popular solutions. They allow to deploy containers across computing nodes assuming that resources are locally distributed (i.e., network latencies are negligible). To overcome these limitations, some works extend the existing orchestration tools (e.g., [12], [24]–[27]). Fahs et al. [26] and Rossi et al. [27] present orchestration frameworks based on Kubernetes to determine the number and location of replicas that are necessary to meet the application QoS requirements. Although these works take into account the peculiarities of an edge/fog environment in the placement problem, only monolithic applications are considered. To integrate our placement policies in Kubernetes, we extend it to explicitly work with asynchronous applications and heterogeneous resources. To this end, we define novel plugins that participate in the Kubernetes scheduling cycle as well as a new custom scheduler.

III. System Model and Problem Definition

In the following, we focus on identifying edge-aware placement solutions for asynchronous microservice-based applications. We consider an edge-cloud environment shared by multiple independent applications. For each application, we assume that its microservices are highly decoupled and that they communicate through a message queue. In Table I, we summarize the used notations.

We consider a geographically distributed edge environment, where multiple edge clusters provide computing resources on-demand. The edge resources are organized in different edge clusters. An edge cluster can be modeled as a graph $G = (N, E)$, where the set of nodes N represents the distributed computing resources and the set of links E represents the logical connectivity between nodes. We characterize each edge node $n \in N$ with the following attributes: C_n, the available computing resources in n; M_n, the available memory in n; P_n, the cost (on a time basis) of using n for hosting application components. We characterize each logical link $(n, m) \in E$ with the network latency $d_{n,m}$ between the nodes n and m. Such a logical connectivity between computing resources results from the underlying physical network paths and routing strategies. These attributes can be known a-priori or can be monitored and estimated at run-time. Each edge cluster has a control node (CN), the entry point of the cluster. When a client submits an application to the CN, the edge cluster scheduler solves the placement problem. We denote as $\mathbb{A}$ the set of all managed asynchronous applications. An application $A \in \mathbb{A}$ consists of multiple microservices and a queue system q. We define $i \in A$ as an application component, i.e., a microservice instance or the queue system. We assume that the user correctly sizes the queue system q, so that it can sustain the application workload without affecting application integrity and performance. To simplify the problem formulation, we use $A' = A \backslash \{q\}$ when the queue system should not be considered. Each application component i is characterized by the required CPU C_i and memory M_i. Differently from synchronous applications, asynchronous applications usually do not aim to minimize response time, because microservices indirectly interact with one another. In a distributed environment, we are interested in allocating microservices close to the message queue, so that they can quickly receive messages from the queue. Therefore, each application $A \in \mathbb{A}$ exposes its requirements in terms of $ND_{A,\max}$, i.e., the maximum network delay between the queue and each microservice allocated on edge resources. For allocating the application components, the edge cluster scheduler can select nodes from the edge or from the cloud. The key idea is to first grant edge resources and then the cloud ones, if there are not enough computing resources on the edge. We denote as A^* the microservices forwarded to the cloud. In general, propagating any application component to the cloud introduces costs and communication delays, which can be detrimental for the application performance. For our investigation, we can reasonably assume that the cloud offers almost infinite computing capacity. We denote as S the set of cloud nodes. We characterize each cloud node $s \in S$ with its available CPU capacity, C_s, memory capacity, M_s, and cost, P_s. We consider cloud resources that are managed by a third party, so we should favor the utilization of edge resources. This is the case of the queue system that, being the application's key communication component, should be up and running for all the application life time. Conversely, the application microservices can be managed more easily, as they can be restarted on a different location without compromising the application availability (so we can temporarily place them on cloud resources). For this reason, we assume that queue systems can be placed only on edge nodes.

TABLE I: Placement Problem Notations.

Entity	*Notation*	*Definition*
Edge Cluster	CN	Control Node of the edge cluster
	$\mathbb{A}$	Set of applications to place
	N	Set of nodes within the edge cluster
	C_n	CPU capacity of node $n \in N$
	M_n	Memory capacity of node $n \in N$
	P_n	Cost of node $n \in N$
	$d_{n,m}$	Network latency between nodes $n \in N$ and $m \in N$
Cloud	A^*	Set of microservices forwarded to the cloud
	S	Set of cloud nodes
	C_s	CPU capacity of node $s \in S$
	M_s	Memory capacity of node $s \in S$
	P_s	Cost of node $s \in S$
Application	$A = \{q\} \cup A'$	Set of application components, with $A \in \mathbb{A}$
	q	Message queue system of application A
	A'	Set of Application Microservices
	C_i	CPU demand of the application component $i \in A$
	M_i	Memory demand of the application component $i \in A$
	$ND_{A,max}$	Maximum network latency required by A

Following a *divide et impera* approach, we divide the placement problem formulation in two sub-problems, i.e., edge and cloud placement problem. This simplifies the placement problem formulation, speeding up the resolution phase and allowing to more easily integrate other objective goals. The *edge placement problem* takes into account the placement on the current edge cluster. If the edge nodes do not have enough resources, some of the application microservices are forwarded to the cloud for processing. In this case, we should solve a second problem, i.e., the *cloud placement problem*. The goal of this problem is to minimize the cost of the used cloud resources, which are rented from a third party.

IV. Optimization Problem Formulation

In this section, we formulate optimization problems to solve the edge and cloud placement problems of asynchronous microservice-based applications.

A. Edge Placement

We model the application placement in the edge cluster with binary variables $x^A_{i,n}$, $A \in \mathbb{A}$, $i \in A$, $n \in N$, where $x^A_{i,n} = 1$ if the component i of the application A is placed on the edge node n, and $x^A_{i,n} = 0$, otherwise. For each $A \in \mathbb{A}$, we use the binary variables z^A_i, with $i \in A$, to indicate the application components to execute in the cloud: $z^A_i = 1$ if the application component i is forwarded to the cloud and $z^A_i = 0$ otherwise. We denote the application placement on edge resources with the vector $\boldsymbol{x} = \langle x^A_{i,n} \rangle$, with $A \in \mathbb{A}$, $i \in A$, and $n \in N$ and the application components forwarded to the cloud with the vector $\boldsymbol{z} = \langle z^A_i \rangle$, with $A \in \mathbb{A}$, $i \in A$.

Edge Resources Cost. For any application component placed on edge nodes, we incur a resource cost, $F(\boldsymbol{x})$:

$$F(\boldsymbol{x}) = \sum_{n \in N} P_n \cdot f_n \tag{1}$$

where the binary variables f_n denote whether $n \in N$ hosts at least one component (i.e., a microservice instance or a message queue system). Therefore we define $f_n, \forall n \in N$, as follows:

$$\frac{\sum_{A \in \mathbb{A}} \sum_{i \in A} x_{i,n}^A + \zeta_n}{\Gamma} \leq f_n \leq \sum_{A \in \mathbb{A}} \sum_{i \in A} x_{i,n}^A + \zeta_n \tag{2}$$

where Γ is a large number and ζ_n is a constant. $\zeta_n = 1$ if n already hosts at least one application component, 0 otherwise. Note that, if $P_n = 1$, the edge resources cost counts only the number of edge nodes used for running the applications.

Cost of Forwarding to Cloud. An edge cluster aims to run microservices locally, optimizing resource utilization of edge nodes. However, to correctly deploy the application, the CN has to enforce the allocation of all of the application microservices. To extend edge resources, the CN can use the cloud. This results in a cost of forwarding microservices to the cloud $Z(\boldsymbol{z})$, which we assume to be proportional to the number of forwarded microservices' instances:

$$Z(\boldsymbol{z}) = \sum_{A \in \mathbb{A}} \sum_{i \in A'} z_i^A \tag{3}$$

Application Constraints. Considering application-level requirements, the placement policy explicitly models the network delay between nodes, and allocates the application microservices only on nodes n and $v \in N$ whose network delay $d_{n,v}$ is below an application-defined critical value $ND_{A,\max}$. Note that microservices communicate asynchronously through the message queue. Therefore, for each application $A \in \mathbb{A}$, it is important that each microservice $i \in A'$ is as close as possible to the queue system q. We define $\gamma_{i,q}^A$ as the network distance between the microservice i and the queue system of the application A. Formally, $\forall A \in \mathbb{A}$ and $\forall i \in A'$ we have:

$$\gamma_{i,q}^A = \sum_{(n,v) \in N \times N} y_{(i,q)(n,v)}^A \cdot d_{n,v} \tag{4}$$

The $y_{(i,q)(n,v)}^A$ variables model the logical AND between placement variables $x_{i,n}^A$ and $x_{q,n}^A$, $\forall A \in \mathbb{A}, i \in A \backslash \{q\}$ and $n, v \in N$: $y_{(i,q)(n,v)}^A = x_{i,n}^A \cdot x_{q,v}^A$.

Similarly, for each $A \in \mathbb{A}$, also the queue system q should be as close as possible to the CN, being the CN the access point to the edge cluster. Therefore, we formalize the following constraint: $d_{CN,n} \cdot x_{q,n}^A \leq ND_{A,\max}$.

Edge Placement Problem Formulation. We formulate the placement problem as an ILP model that determines the optimal mapping between the applications' components and the edge nodes. Our problem formulation considers an objective function that minimizes the edge and cloud resources cost. We define the objective function $G(\boldsymbol{x}, \boldsymbol{z})$ as the sum of the QoS metrics to be minimized:

$$G(\boldsymbol{x}, \boldsymbol{z}) = F(\boldsymbol{x}) + Z(\boldsymbol{z}) \tag{5}$$

The Edge Placement problem is formulated as follows:

$$\min_{\boldsymbol{x}, \boldsymbol{z}} G(\boldsymbol{x}, \boldsymbol{z})$$

subject to:

$$\sum_{n \in N} x_{i,n}^A + z_i^A = 1, \quad \forall A \in \mathbb{A}, \forall i \in A \tag{6}$$

$$z_q^A = 0, \quad \forall A \in \mathbb{A} \tag{7}$$

$$\sum_{A \in \mathbb{A}} \sum_{i \in A} C_i \cdot x_{i,n}^A \leq C_n, \quad \forall n \in N \tag{8}$$

$$\sum_{A \in \mathbb{A}} \sum_{i \in A} M_i \cdot x_{i,n}^A \leq M_n, \quad \forall n \in N \tag{9}$$

$$d_{CN,n} \cdot x_{q,n}^A \leq ND_{A,\max}, \quad \forall A \in \mathbb{A}, n \in N \tag{10}$$

$$\gamma_{i,q}^A \leq ND_{A,\max}, \quad \forall A \in \mathbb{A}, i \in A' \tag{11}$$

$$\sum_{v \in N} y_{(i,q)(u,v)}^A = x_{i,u}^A, \;\; \forall A \in \mathbb{A}, i \in A', u \in N \tag{12}$$

$$\sum_{u \in N} y_{(i,q)(u,v)}^A \leq x_{q,v}^A, \; \forall A \in \mathbb{A}, i \in A', v \in N \tag{13}$$

$$\frac{1}{\Gamma}\left(\sum_{A \in \mathbb{A}} \sum_{i \in A} x_{i,n}^A + \zeta_n \right) \leq f_n \quad \forall n \in N \tag{14}$$

$$\sum_{A \in \mathbb{A}} \sum_{i \in A} x_{i,n}^A + \zeta_n \geq f_n \quad \forall n \in N \tag{15}$$

$$x_{i,n}^A \in \{0,1\} \quad \forall n \in N, A \in \mathbb{A}, i \in A \tag{16}$$

$$z_i^A \in \{0,1\} \quad \forall A \in \mathbb{A}, i \in A \tag{17}$$

$$f_n \in \{0,1\} \quad \forall n \in N \tag{18}$$

where (6) ensures that all the application components are either assigned to edge nodes or forwarded to the cloud, and guarantees that each of them is placed on one and only one node. The constraint (7) forces the placement of the queue systems on edge nodes only. Constraints (8) and (9) limit the placement of the application components on an edge node $n \in N$ according to its available resources, while (10) and (11) limit the network delays among edge nodes used to run the application. Constraints (12) and (13) model the logical AND between the placement variables. Finally, (14) and (15) define the f_n variables, $\forall n \in N$, indicating whether n is used to run any of the application components.

B. Cloud Placement

If any microservice instance should be forwarded to the cloud, we have to solve the *cloud placement problem*. For each microservice $i \in A^*$, we model the microservice i placement on a cloud node $s \in S$ with new binary variables $t_{i,s}$: where $t_{i,s} = 1$ if microservice i is placed on the cloud node s and $t_{i,s} = 0$ otherwise. We denote the cloud placement vector as $\boldsymbol{t} = \langle t_{i,s} \rangle$, with $i \in A^*$ and $s \in S$.

Cloud Resources Cost. The cloud resources cost $P(\boldsymbol{t})$ accounts for the active cloud nodes for running the applications' microservices:

$$P(\boldsymbol{t}) = \sum_{s \in S} \delta_s \cdot P_s \tag{19}$$

where the binary variables δ_s denote whether $s \in S$ is active and hosts at least one microservice. We formally define $\delta_s, \forall s \in S$ as follows:

$$\frac{\sum_{i \in A^*} t_{i,s} + \psi_s}{\Gamma} \leq \delta_s \leq \sum_{i \in A^*} t_{i,s} + \psi_s \tag{20}$$

where Γ is a large number and ψ_s is a constant such that $\psi_s = 1$ if s hosts at least a microservice (as result of previous optimization rounds), 0 otherwise.

Cloud Placement Problem Formulation. We formulate the cloud placement problem as an ILP problem which defines a mapping of the applications' microservices on the cloud nodes with the aim of minimizing the cost of used cloud resources. The Cloud Placement problem is formulated as follows:

$$\min_{\boldsymbol{t}} P(\boldsymbol{t}) \tag{21}$$

$$\sum_{s \in S} t_{i,s} = 1 \qquad \forall i \in A^* \tag{22}$$

$$\sum_{i \in A^*} C_i \cdot t_{i,s} \leq C_s \qquad \forall s \in S \tag{23}$$

$$\sum_{i \in A^*} M_i \cdot t_{i,s} \leq M_s \qquad \forall s \in S \tag{24}$$

$$\frac{\sum_{i \in A^*} t_{i,s} + \psi_s}{\Gamma} \leq \delta_s \qquad \forall s \in S \tag{25}$$

$$\sum_{i \in A^*} t_{i,s} + \psi_s \geq \delta_s \qquad \forall s \in S \tag{26}$$

$$t_{i,s} \in \{0, 1\} \qquad \forall s \in S, \forall i \in A^* \tag{27}$$

where (22) ensures that all the forwarded microservices are placed on cloud nodes. The constraints (23) and (24) limit the placement of microservices on a cloud node $s \in S$ according to its available resources. Finally, (25)–(27) are used to define the δ_s variables, $\forall s \in S$.

V. The Pogonip Heuristic

Allocating asynchronous applications on (edge or cloud) computing resources is an NP-hard problem; so, the ILP formulations might not scale well as the problem instance increases in size. To overcome this issue, we propose the Pogonip greedy heuristics. First, we present the *greedy edge placement heuristic*, a network-aware policy to determine the placement of asynchronous applications in the edge cluster. Then, we describe the *greedy cloud placement heuristic* that allows to reduce the number of cloud nodes used to host the forwarded application microservices.

A. Greedy Edge Placement Heuristic

The proposed greedy edge placement heuristic solves a variant of the bin-packing problem, while taking into account the available computing resources and the network delays between edge nodes (see Algorithm 1). First, it sorts the unplaced applications by their $ND_{A,\max}$ requirement in ascending order, i.e., the first applications of the list have more stringent $ND_{A,\max}$ values (line 4). Then, it places one application at a time (line 6–8). For each application A, the heuristic identifies N^q, the set of edge nodes that can host the queue system q and that have a network delay to the CN below $ND_{A,\max}$ (lines 11–12). If N^q is empty, the application is discarded. Otherwise, the heuristic computes β_n for each $n \in N^q$. The β_n factor estimates the node n capacity of hosting q, approximating the number of q instances that can be executed on n, considering the most critical resource (line 17). The edge node having maximum value of β_n is selected for the allocation of q. This allows to spread queue systems across nodes, avoiding node congestion.

Algorithm 1 Placement Heuristic on Edge Nodes

```
 1: Input: $\mathbb{A}$: Applications to deploy; $N$: Set of edge nodes;
 2: Output: $A^*$: Microservices forwarded to cloud;
 3: Output: $X$: Application placement;
 4: $\mathbb{A}$ = Sort $A \in \mathbb{A}$ by $ND_{A,\max}$ (in ascending order)
 5: $X = \{\}$
 6: for all $A \in \mathbb{A}$ do
 7:     applicationPlacement($A$, $N$, $X$, $A^*$)
 8: end for

 9: function APPLICATIONPLACEMENT($A$, $N$, $X$, $A^*$)
10:     $q \leftarrow$ Queue system of application $A$
11:     $N^q \leftarrow$ Filter $n \in N$ on $q$ resource requirements
12:     $N^q \leftarrow$ Filter $n \in N^q$ on $d_{CN,n} \leq ND_{A,\max}$
13:     if $N^q$ is empty then
14:         discard application $A$
15:         return
16:     end if
17:     Compute $\beta_n = \min(\lfloor \frac{C_n - C_q}{C_q} \rfloor, \lfloor \frac{M_n - M_q}{M_q} \rfloor)$, $\forall n \in N^q$
18:     $x^A_{q,n_q} \leftarrow$ Allocate $q$ on $n_q$ having maximum value of $\beta_n$
                                      ▷ spread applications across nodes
19:     $X \leftarrow X \cup x^A_{q,n_q}$
20:     for all microservice $i \in A \backslash \{q\}$ do
21:         $N^i \leftarrow$ Filter $n \in N$ on $i$ resource requirements
22:         $N^i \leftarrow$ Filter $n \in N^i$ on $d_{n_q,n} \leq ND_{A,\max}$
23:         if $N^i$ is empty then
24:             $A^* \leftarrow A^* \cup i$
25:             continue;
26:         end if
27:         Compute $\beta_n = \min(\lfloor \frac{C_n - C_i}{C_i} \rfloor, \lfloor \frac{M_n - M_i}{M_i} \rfloor)$, $\forall n \in N^i$
28:         $x^A_{i,n_i} \leftarrow$ Allocate $i$ on $n_i$ having minimum value of $\beta_n$
                                      ▷ maximize resource utilization
29:         $X \leftarrow X \cup x^A_{i,n_i}$
30:     end for
31: end function
```

Similarly, for each microservice i, the heuristic identifies the edge nodes N^i (lines 21–22). If N^i is empty, i is forwarded to the cloud. Otherwise, the heuristic greedily chooses the first candidate node that minimizes β_n (line 27). Note that this avoids spreading microservices across the computing nodes, while preferring to minimize the number of active nodes.

B. Greedy Cloud Placement Heuristic

The cloud placement heuristic uses a greedy approach to place the microservices received from edge control nodes (see Algorithm 2). For each microservice $i \in A^*$ and cloud node $s \in S$, the heuristic filters cloud resources according to the resource requirements of i, expressed in terms of CPU C_i and memory M_i demand. First, the heuristic computes β_s, which estimates the node s capacity of hosting i (line 5). The cloud node with minimum β_s value is selected to host i. This allows to reduce the number of used cloud nodes and, as a consequence, cloud usage cost. At the end, the application placement T is accordingly updated.

VI. Prototype

In this section, we present the prototype implementation of our heuristic, realized as a Kubernetes scheduler.

Algorithm 2 Placement Heuristic on Cloud Resources

```
Input: $A^*$: Microservices to deploy; $S$: Set of cloud nodes;
Output: $T$: Application placement;
for all microservice $i \in A^*$ do
    $S^i \leftarrow$ Filter $s \in S$ on $i$ resource requirements
    Compute $\beta_s = \min(\lfloor \frac{C_s - C_i}{C_i} \rfloor, \lfloor \frac{M_s - M_i}{M_i} \rfloor), \forall s \in S^i$
    $t_{i,s_i} \leftarrow$ Allocate $i$ on $s_i$ having minimum value of $\beta_s$
    $T \leftarrow T \cup t_{i,s_i}$
end for
```

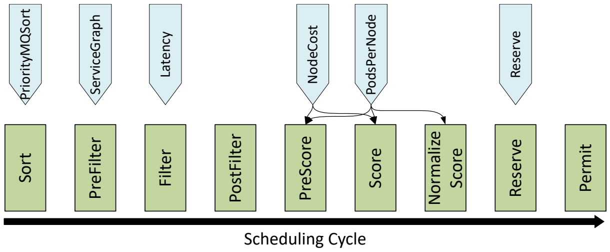

Fig. 1: Scheduling cycle (adapted from [28]) and Pogonip plugins.

A. *Kubernetes Scheduler*

A pod is the smallest deployment unit in Kubernetes. It consists of one or more tightly coupled containers that are co-located and scaled as an atomic entity. Each application component (i.e., a microservice or the queue system) is deployed using a pod. Kubernetes ensures that a given number of pods are up and running using a *Replica Set*. To manage the deployment of applications, the *Deployment* object is built upon the Replica Set concept, exposing a higher level abstraction, simplifying the pods' update and providing additional functionality (e.g., rolling updates). To manage stateful applications, whose pods must be deployed in a particular order, have persistent IDs, and/or always be connected to the same storage volumes (e.g., RabbitMQ, which is used in Section VII), Kubernetes introduces the *Stateful Set* concept. Differently from Deployments, a Stateful Set maintains a sticky identity for each managed pod, allowing its state recovery. When a new pod is created, Kubernetes triggers the scheduler to identify a suitable hosting node. The default Kubernetes scheduler is *kube-scheduler*, which is implemented using the *scheduling framework* [28], an extensible architecture for Kubernetes schedulers. It decomposes the scheduling process into two cycles: scheduling and binding. From a high-level perspective, the *scheduling cycle* is a sequence of filtering and scoring stages. First, it identifies the nodes that can run the pod by applying a set of filters. Then, it assigns a score to all eligible nodes according to different criteria. Finally, it selects the node with the highest score to host the pod. If multiple nodes achieve the same score, one of them is randomly selected. The mapping between the pod and the chosen node is committed to the cluster by the *binding cycle* [28]. The kube-scheduler includes a placement policy that spreads pods on computing resources located in the cluster. As such, it is not well-suited for placing pods in an edge computing environment and dealing with its heterogeneity. However, the modularity of Kubernetes allows us to easily integrate custom placement policies. There are two main ways to customize the placement process: (1) by changing the configuration of the default scheduler; or (2) by implementing a custom scheduler that runs instead of the default one [27]. Changing the configuration of the default scheduler is limited by the capabilities of kube-scheduler, requiring a custom scheduler for more advanced customizations. A custom scheduler is any application that observes the list of pods and assigns pods to nodes. However, relying on the modular architecture of the scheduling framework simplifies the development of new placement policies, splitting their logic into multiple and decoupled stages. Additionally, the scheduling framework allows reusing some or all the plugins from kube-scheduler [29], such as, e.g., `NodeResourcesFit`, which filters out nodes that do not satisfy a pod's resource requirements. Kubernetes schedules an application's pods independently: a failure to schedule/place one pod has no effect on the scheduling status of the other pods. The scheduler will retry to place a failed pod later. When the scheduling framework's sequence of stages cannot be applied (e.g., as in the optimal placement formulation), an independent custom scheduler is required.

B. *Prototype Architecture and Implementation*

The Pogonip scheduler implements the greedy heuristics presented in Section V leveraging the scheduling framework of Kubernetes v1.20.1 and most of its default plugins. The prototype is published as open-source under the name *rainbow-scheduler* in the RAINBOW[2] project's orchestration package[3].

Pogonip needs to be able to identify the pods that contain a queue system and be aware of the application's maximum tolerable network delay. For this prototype, we use Kubernetes labels to attach this information to each pod. As part of future work within RAINBOW, a service graph abstraction will be developed that will contain this and other relevant information about an application. Pogonip augments the default kube-scheduler functionality by adding the plugins to the scheduling cycle shown in Fig. 1. The green boxes show the scheduling cycle stages that are executed for every pod. Each stage provides an extension point, for which plugins can be registered. The `Sort` stage determines the order in which the incoming pods will be handled (only one plugin can be active in this stage). `PreFilter` and `Filter` are responsible for filtering out nodes that cannot host the newly added pod, e.g., because they have too few resources. The `PreFilter` stage is executed per pod to prepare information needed in the `Filter` stage, which, conversely, is executed for every node. If the set of remaining nodes is empty, the `PostFilter` plugins are executed. `PreScore` and `Score` plugins assign a score to each eligible node. Similarly to the filter-related stages, the `PreScore` stage is executed once per pod, while the `Score` stage is executed once per node. A `NormalizeScore` extension may be registered for each `Score` plugin to normalize its scores as an integer between 0 and 100, as is required by the scheduling framework. After this stage, the node with the highest score is selected to host the pod. `Reserve` plugins are

[2] https://rainbow-h2020.eu

[3] https://gitlab.com/rainbow-project1/rainbow-orchestration

notified with the outcome, allowing to update third-party data structures. Permit plugins are executed in the last stage of the scheduling cycle, to approve, deny, or delay a pod from being admitted to the binding cycle. In the default configuration, kube-scheduler registers multiple plugins in the scheduling cycle, such as NodeResourcesBalancedAllocation and NodeResourcesLeastAllocated. While the first plugin favors nodes that would obtain a more balanced resource usage, the latter prefers nodes that have few allocated resources. Consequently, the default scheduling strategy spreads pods: it prioritizes nodes with the least number of pods, without considering their heterogeneity or geographic distribution [29].

Pogonip extends kube-scheduler with custom plugins, as shown in Fig. 1. Building on top of the default kube-scheduler PrioritySort plugin, PriorityMqSort prioritizes the pods belonging to applications with more stringent $ND_{A,\max}$ requirements and ensures that the queue system pods are placed before the others. ServiceGraph is a PreFilter plugin that retrieves the graph for the application that the pod is part of. The Latency plugin is a Filter plugin that removes all nodes that do not meet the application's $ND_{A,\max}$ requirement. If the pod hosts a queue system, the network latency between the current node and the edge CN is considered. Otherwise, the plugin filters nodes limiting the network latency to the node hosting the application queue system. For cloud nodes, the Latency plugin is disabled. The PodsPerNode plugin ties into the PreScore and Score extension points. First, in the PreScore stage, the plugin retrieves the pod's required resources. Then, in the Score stage, for each edge/cloud node n, it computes the β_n factor (see Section V). For all application pods (but the queue system), we want to select the node n with the minimum β_n value. Once all β_n have been computed, they are normalized in the $[0, 100]$ range. Furthermore, to implement the preference for edge nodes, the PodsPerNode returns a score of zero for all cloud nodes, if at least one eligible edge node has been found. NodeCost is a Score plugin that assigns higher scores to cheaper nodes. To avoid compromising the optimizations by the Pogonip Score plugins, we disable the scheduling framework plugins NodeResourcesBalancedAllocation and NodeResourcesLeastAllocated. Finally, Reserve runs in the Reserve stage and updates the application placement.

To solve the optimal ILP placement formulation within Kubernetes, we develop a *custom scheduler* and a *Placement Resolver*. The custom scheduler is deployed as a pod and invoked by Kubernetes as soon as pods need to be allocated on the nodes. To solve the placement problem, the custom scheduler interacts with the Placement Resolver. It is an external service that exposes the ILP placement problem resolution as a service, through RESTful APIs. As soon as the custom scheduler obtains the pods placement, it defines the pod-to-node mapping using the Kubernetes abstractions.

C. Benchmark Placement Policies

In this section, we present the existing placement policies against which we evaluate our edge-aware solutions. Together with the default *kube-scheduler* policy, we include two well-known placement policies, that are often adopted in computing frameworks, namely Greedy First-fit and Round-robin.

We implement these placement policies using the Kubernetes scheduling framework. They both leverage the default PreFilter and Filter plugins to determine which nodes are capable of hosting a pod. However, we replace all default PreScore and Score plugins with a single Score plugin, which implements the corresponding placement policies.

The Greedy First-fit Heuristic. The Greedy First-fit heuristic is one of the most popular solutions used to solve the bin packing problem. It considers the application's pods as elements to be (greedily) allocated in bins, representing computing nodes. Specifically, for each pod, the Greedy First-fit policy defines the placement on the first node that fulfills the pod's resource requirements. Our GreedyFirstFit plugin greedily selects the first fitting node from the iteration order provided by the scheduling framework.

Round-robin Heuristic. The Round-robin heuristic organizes the nodes in a circular list, registering the latest node used for placement. A new pod to be allocated is assigned to the first node with enough resources, starting from the current position on the circular list. Akin to our Greedy First-fit implementation, we implement the Round-robin selection with a single RoundRobin plugin for the Score stage.

VII. Experimental Results

We define two sets of experiments aimed to show the benefits of our placement policies when the managed application is deployed in an edge computing environment using Kubernetes. First, in Section VII-B, we analyze the advantages of using edge-aware policies in a heterogeneous environment. Then, in Section VII-C, we generalize the achieved results and show the benefits of combining edge and cloud computing resources when multiple applications should be executed. We compare the edge-aware policies, presented in Section IV and V, against the benchmark placement policies, presented in Section VI-C.

A. Experiment Setup

As reference application, we use a modified version of an Internet of Things (IoT) taxi application[4] written for the Fogify fog emulator [30], [31]. It uses real-world taxi and limousine data[5] to generate its workload. We have modified this application's microservices to communicate asynchronously through a RabbitMQ[6] queue system. A single application deployment consists of one RabbitMQ instance and four taxi app microservices: an *IoT load generator* that sends location data from the dataset once per second to two *edge aggregator* instances; the latter buffer the received data for one minute and then send them to a *data storage* microservice, which permanently stores the data. The resource requirements are the defaults used by the RabbitMQ Kubernetes Operator [32] and reasonable values for the microservices, given their purposes:

[4] https://github.com/UCY-LINC-LAB/fogify-demo

[5] https://www1.nyc.gov/site/tlc/about/tlc-trip-record-data.page

[6] https://www.rabbitmq.com

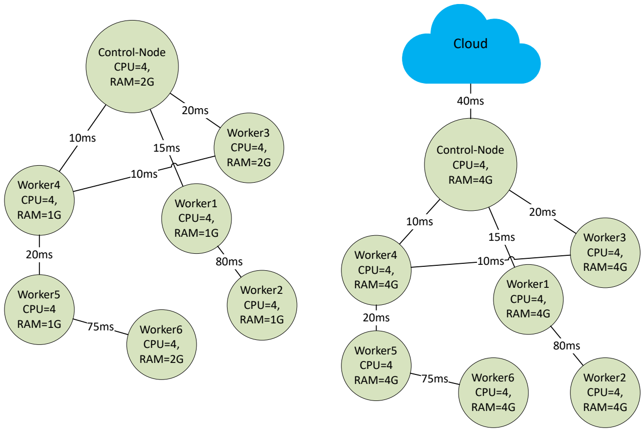

Fig. 2: Cluster topologies in experiments: Scenario 1 (left), Scenario 2 (right).

- RabbitMQ: 2 CPU cores, 2 GiB memory
- IoT load generator: 0.25 CPU cores, 0.25 GiB memory
- Edge aggregator (2x): 0.5 CPU cores, 0.5 GiB memory
- Data storage microservice: 1 CPU core, 1 GiB memory

The application requires $ND_{A,\max}$ to be equal to 50 ms.

We set up a cluster using the `kind`[7] tool, which allows running a Kubernetes cluster (v1.20.1) inside Docker, with each node being a container. Since nodes would report the CPU and memory capacity of the host VM as their available resources, we created two extended resources, `fake-cpu` and `fake-memory`, which we can explicitly configure for each node. The cluster runs on a VM with 22 virtual CPU cores and 62.9 GiB of RAM. The hosting server has an Intel Xeon CPU (Cascade Lake) with a base clock of 2.1 GHz. The cluster topology differs between the two experimental scenarios, as shown in Fig. 2. The vertices represent edge nodes, while links denote the network connections between nodes; on each link, we report the network latency expressed in milliseconds (ms) [33]. Both scenarios consider a single edge cluster.

As placement policies, we consider the optimal ILP formulation (referred to as OPT), the Pogonip heuristic, the Greedy First-fit heuristic, the Round-robin heuristic, and the default kube-scheduler. To solve the optimal ILP formulation, we use CPLEX 12.8. To minimize the number of used edge nodes, in the OPT we set $P_n = 1$ for each node of the edge cluster (see Eq. 1). For each scenario we execute 5 runs with every placement policy. All source code, including the experimental scripts, is available in our public repository.

B. *Application Deployment and Network Latencies*

In this experiment, we consider a single taxi application instance and only edge nodes. To model the edge environment, we have configured the `fake-cpu` and `fake-memory` resources to match those of the Raspberry Pi 3 Model B+ (4 CPU cores and 1 GiB of RAM) and the Raspberry Pi 4 Model B (4 CPU cores and 2 GiB of RAM)[8].

The goal of this experiment is to evaluate the latency between the microservices and the queue system that can be achieved by the various schedulers. Fig. 3 summarizes the results. The different schedulers obtain very different placements for the application, including solutions where microservices are allocated far away from the message queue. In such a case, the application performance can be significantly reduced.

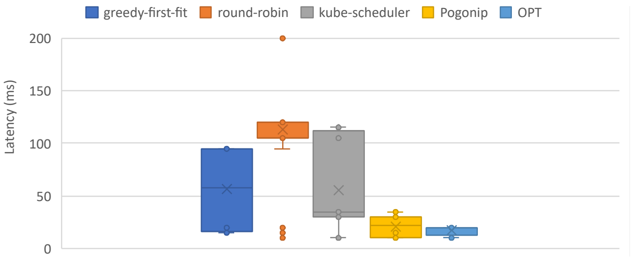

Fig. 3: Latency between microservices and the queue system, when a single application is deployed using different placement policies.

The Greedy First-fit policy does not meet the $ND_{A,\max}$ requirement for more than half of the microservices, with a median latency of 57.5 ms and a mean average of 56.25 ms. For each pod, it chooses the first node that fulfills the resource requirements in the list `(control-node, worker1, ..., worker6)`. Thus, the placement is the same on every run. Four nodes are used: the queue system is placed on the `control-node` and the four application microservices as follows: one on `worker1`, two on `worker2`, and one on `worker3`. Both, `worker1` and `worker3` have low network latency towards the `control-node`, which explains Greedy First-fit's minimum values of 15 ms. However, since `worker2` has a latency of 95 ms to the `control-node`, the $ND_{A,\max}$ requirement was clearly violated. The Round-robin policy and kube-scheduler policy spread the application pods on 5 edge nodes, which is the highest number of nodes with respect to the other configurations. The Round-robin policy organizes the edge nodes into a circular list. The queue system is placed on the `control-node` and the application microservices in the nodes `worker1` through `worker4` in the first run. This violates the $ND_{A,\max}$ requirement, because of the use of `worker2`. Subsequent runs performed even worse, because the next node in the circular list at the start of these runs was either `worker5` or `worker4`. Both are too small to host the queue system, resulting in it being placed on `worker6`, which has the second highest latency to all other nodes, thus, explaining the poor performance of the Round-robin policy. Kube-scheduler registers a mean average latency of 56 ms and a median latency of 35 ms. Conversely to Greedy First-fit and Round-robin, kube-scheduler uses all `worker` nodes across the runs (five in every run), but avoids the `control-node`. This may be related to the fact that this node hosts the Kubernetes master. Unlike the benchmark heuristics, OPT and Pogonip consider network delays while computing the application placement. OPT always computes the best placement, obtaining a mean average network latency of 17.5 ms and a median of 20 ms by using the `control-node`, `worker3`, and `worker4`. With Pogonip, we register a maximum latency of 35 ms, an average of 21.5 ms, and a median of 22.5 ms. Across all runs, Pogonip uses all nodes, except for `worker2` and `worker6`, registering

[7] https://kind.sigs.k8s.io

[8] https://www.raspberrypi.org

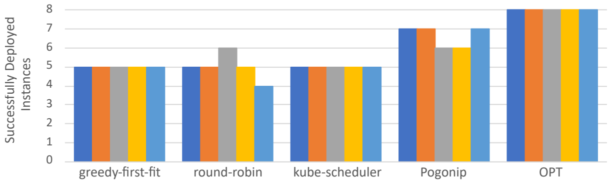

Fig. 4: Successfully deployed application instances (out of 8 submitted) when different placement policies are used. Each experiment is run 5 times.

a slight increase in the network latencies compared to the OPT solution. Anyway, Pogonip always meets $ND_{A,\max}$, outperforming all previous benchmark policies.

C. Allocating Applications on Edge and Cloud Nodes

In this experiment, we consider an edge cluster that receives application deployment requests and uses its control node to allocate them for execution. We submit 8 instances of the taxi application to the edge cluster. The benchmark policies (i.e., Greedy First-fit, Round-robin, and kube-scheduler) are not designed to distinguish between edge and cloud nodes. Therefore, they cannot complement the edge with resources rented from the cloud. Conversely, our policies can benefit from the cloud. We consider the computing infrastructure depicted in Fig. 2. All edge nodes have the resources of a Raspberry Pi 4 Model B with a hardware configuration of 4 CPU cores and 4 GiB of RAM. We use three types of cloud nodes, with 10 instances each, characterized as follows:

- *small*: 4 CPU cores, 4 GiB of RAM, cost of \$2/hour;
- *medium*: 8 CPU cores, 8 GiB of RAM, cost of \$4/hour;
- *large*: 16 CPU cores, 16 GiB of RAM, cost of \$8/hour.

Although these cloud node costs are fictional, their ratios match those of real-world cloud providers (e.g., [34]).

As soon as the pod placement is computed, Kubernetes enacts it. If not enough resources are available, some pods cannot be successfully placed, meaning that they remain in a pending state until resources are freed. Fig. 4 shows the number of successfully deployed applications and Fig. 5 the distribution of network latencies between the microservices and the queue system for the successfully deployed applications.

The Greedy First-fit policy successfully executes 5 application instances out of the 8 submitted, in each run of the experiment. Greedy First-fit results in network latencies to the queue system comparable to those of the previous experiment. However, in this case, the minimum latency was 0 ms; due to the more powerful nodes, some microservices are colocated with their queue system. In spite of that, 58% of the application microservices exceed the $ND_{A,\max}$ constraint. The Round-robin policy successfully executes between 4 and 6 application instances, with 5 being the median number. With a median latency of 80 ms, the $ND_{A,\max}$ constraint was violated by 56% of the microservices. As expected, the Round-robin placement is pseudo random, as it depends on the number and order of edge nodes as well as the number of pods already allocated. Since Kubernetes does not treat the application as a whole, we observe a high variation in the number of successfully executed applications across the runs. For example, in the third run, Round-robin successfully executes 6 applications (even though 3 queue system pods are placed on nodes interconnected with high latency). Instead, in the last run, only 4 applications are successfully executed. This is due to Kubernetes, which executes one message queue pod more than in the other runs and, because of the limited available resources, this prevents other microservices from running. This reveals a non-deterministic behavior of Kubernetes in prioritizing pods for execution. First, when priorities are not explicitly assigned, pods are sorted by their creation time (see `PrioritySort` [29]). Second, the different Kubernetes controllers, e.g., Deployment and StatefulSet, work in parallel, so they concurrently manage the creation and execution of different application pods. In our case, each queue system is a StatefulSet, whereas the other microservices are controlled by a Deployment. Kube-scheduler successfully executes 5 application instances in all runs. The latency distribution between the queue system and each microservice is slightly better than the one obtained using Round-robin. With a median latency of 35 ms, 60% of the pods fulfilled the $ND_{A,\max}$ constraint.

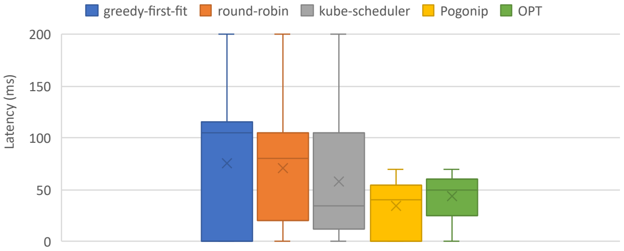

Fig. 5: Latency between microservices and the queue system, when multiple applications are deployed using different placement policies.

Pogonip executes between 6 and 7 instances in this experiment, because, unlike the previous policies, it can place the application microservices in the cloud as well (only the queue system is required to be on an edge node). With 18 to 20 pods on the edge, Pogonip places fewer pods there than the benchmark policies. This is due to the prioritization of the queue system pods, which have the highest resource requirements of all pods. Since bigger pods are placed on the edge, fewer resources are left there for the other microservices, which are instead placed in the cloud. All application microservices are placed on three small cloud nodes, resulting in a total cloud cost of \$6/hour. This is the lowest possible value; e.g., in the first run, all pods placed in the cloud require a total of 12 GiB of memory, which could also be met by two medium nodes or one large node, both would result in the higher price of \$8/hour. Despite placing about half of the pods on cloud nodes, Pogonip achieves much better latencies than the benchmark policies by avoiding the two high-latency edge nodes. With a median latency of 40 ms, Pogonip fulfilled the $ND_{A,\max}$ constraint for about 62% of the pods. The other pods violated the constraint by an average of 10.3 ms, i.e., less than a sixth of the next best benchmark policy, Round-robin.

Why does Pogonip not place all instances, despite prioritizing the queue system and using the cloud? Their creation timestamps showed that some queue system pods are created

TABLE II: Scheduler Resolution Times

Scheduler	Time per Instance (8 Total)	Time per Instance (10 Total)	Increase
Greedy First-fit	21.6 ms	30.2 ms	40%
Round-robin	26.0 ms	32.8 ms	26%
kube-scheduler	38.4 ms	43.9 ms	14%
Pogonip	131.0 ms	152.7 ms	17%
OPT	334.8 ms	576.0 ms	72%

after some other microservice pods. We also noticed that Kubernetes starts scheduling before all pods have been created by the StatefulSet and Deployment controllers. Since pods are the schedulable units, Kubernetes considers a pod ready for scheduling as soon as it has been created. Pogonip's prioritization algorithm also has to adhere to this limitation. OPT executes all 8 application instances, placing 20 pods on the edge and 20 pods in the cloud. It also results in the lowest latency distribution of all other policies, with a median of 50 ms. Differently from the other scheduling policies, OPT waits for all 40 pods to be created before computing the optimal placement solution for all the applications. This results in all applications being successfully executed on both edge and cloud nodes. OPT uses 4 small cloud nodes for running 20 application microservices, resulting in a cost of $8/hour. It requires one more cloud node than Pogonip, because OPT executes one more queue system on the edge where there are fewer resources available for other microservices. The 20 pods selected for scheduling in the cloud require a total of 13.5 GiB of memory, which could be met by four small nodes, two medium nodes, or one large node – all for the same price of $8/hour, making the selected solution the cheapest.

These two experiments showed the importance of considering application and computing features while determining the placement. By considering network latency, Pogonip and OPT reduce the communication delay between the application queue and its microservices (resulting also in limited latency variance). This can be critical to the proper functioning of an edge application. The ability of combining edge and cloud computing allows allocating a greater number of applications with respect to the other benchmark placement policies. We conclude the section with some consideration on the resolution time of each placement policy. We measure the resolution time as the time needed to compute the application placement. Technically, we measure it as the time a pod needs to move from the `PreFilter` stage until the `Reserve` stage and then sum the times for all pods in the run. For OPT, we measure it as the time needed to compute the placement for the edge and for the cloud. We conducted an additional experiment, with 10 application instances deployed at once, which constitutes a 25% increase in the number of pods that need to be placed. Table II shows execution times for a single instance and the increases in execution time between 8 and 10 instances.

Greedy First-fit and Round-robin have similar resolution times; their implementation is rather simple and they differ only in one `Score` plugin. Kube-scheduler results in a slightly longer resolution time, because it contains more `Score` plugins. Pogonip takes 131 ms on average for 8 instances and 152.7 ms for 10 instances. This is about 3.5 times as long as kube-scheduler, which is likely caused by the `Latency` plugin, which has to evaluate paths through the cluster graph. The execution time increases by about 17% between 8 and 10 instances, which is below the increment in number of pods. OPT aims to find the optimal solution of the ILP formulation; it registers the longest resolution time, with 335 ms to place 8 instances and 576 ms to place 10 instances. In this case, the resolution time increases by about 72%, which is almost three times the increment in the number of pods. We observe that even though the number of applications is rather limited, OPT requires more than half a second to find a placement solution. Of course, we expect this time to exponentially increase as the number of applications increases as well, thus resulting in an impractical approach when it comes to working in a dynamic edge environment. This limitation of OPT justifies the adoption of edge-aware placement heuristics that can compute the placement of asynchronous components more quickly.

VIII. Conclusion

Microservices are an architectural style for developing an application as a suite of autonomous and decoupled services, that communicate using synchronous or asynchronous techniques. Although the placement problem is widely explored in the context of synchronous applications, so far, to the best of our knowledge, the problem of allocating asynchronous applications has not been investigated. Therefore, in this paper, we presented an approach for solving the placement problem for asynchronous microservice-based applications in an edge environment. First, we formulate the problem as an ILP model. Since the problem is NP-hard, it may suffer from scalability issues when the number of managed microservices increases. Thus, we propose Pogonip, a novel edge-aware heuristic. It can quickly allocate asynchronous microservices by explicitly taking into account the peculiarities of edge nodes. Moreover, if microservices require more capacity than available in the edge, it can complement the computing environment by exploiting cloud computing. Integrating these policies in Kubernetes, we conducted an extensive evaluation using an edge application that processes taxi location data. The experimental results showed the benefits of combining edge and cloud resources as well as the importance of explicitly considering the edge environment's peculiarities while allocating applications, resulting in better adherence to their requirements.

As future work, we plan to extend the proposed policies to efficiently model other performance metrics that can be of interest for asynchronous applications in an edge environment (e.g., network usage, energy consumption, availability). Moreover, we want to investigate the impact of mobile nodes (e.g., smart cars) and service migrations in terms of performance penalty, to develop efficient placement adaptation heuristics. As a long term plan, we want to extend the proposed approach so as to control the elasticity of asynchronous applications.

Acknowledgment

*The authors Fabiana Rossi and Thomas Pusztai have contributed equally to this work.

References

[1] I. Nadareishvili, R. Mitra, M. McLarty, and M. Amundsen, *Microservice Architecture: Aligning Principles, Practices, and Culture*. O'Reilly, 2016.

[2] C. de la Torre, B. Wagner, and M. Rousos, *.NET Microservices: Architecture for Containerized .NET Applications*, v5.0 ed. Microsoft Corporation, 2020.

[3] E. Ntentos, U. Zdun, K. Plakidas, S. Meixner, and S. Geiger, "Assessing architecture conformance to coupling-related patterns and practices in microservices," in *Software Architecture*, ser. LNCS. Springer, 2020, vol. 12292, pp. 3–20.

[4] M. Satyanarayanan, "The emergence of edge computing," *Computer*, vol. 50, no. 1, pp. 30–39, 2017.

[5] Y. Ai, M. Peng, and K. Zhang, "Edge computing technologies for internet of things: a primer," *Digital Communications and Networks*, vol. 4, pp. 77–86, 2018.

[6] I. M. A. Jawarneh, P. Bellavista, F. Bosi, L. Foschini, G. Martuscelli, R. Montanari, and A. Palopoli, "Container orchestration engines: A thorough functional and performance comparison," in *Proc. of IEEE ICC'19*, 2019, pp. 1–6.

[7] A. Brogi, S. Forti, and A. Ibrahim, "Predictive analysis to support fog application deployment," *Fog and edge computing: principles and paradigms*, pp. 191–222, 2019.

[8] A. Brogi, S. Forti, C. Guerrero, and I. Lera, "How to place your apps in the fog - state of the art and open challenges," *Softw. Pract. Exp.*, vol. 50, pp. 719–740, 2020.

[9] V. Cardellini, F. Lo Presti, M. Nardelli, and F. Rossi, "Self-adaptive container deployment in the fog: A survey," in *Algorithmic Aspects of Cloud Computing*, ser. LNCS, vol. 12041. Springer, 2020, pp. 77–102.

[10] C. J. L. de Santana, B. de Mello Alencar, and C. V. S. Prazeres, "Reactive microservices for the internet of things," in *Proc. of ACM/SIGAPP SAC'19*, 2019, pp. 1243–1251.

[11] S. Nastic, M. Vögler, C. Inzinger, H. Truong, and S. Dustdar, "rt-GovOps: A runtime framework for governance in large-scale software-defined IoT cloud systems," in *Proc. of IEEE MobileCloud'15*, 2015, pp. 24–33.

[12] P. Garefalakis, K. Karanasos, P. Pietzuch, A. Suresh, and S. Rao, "Medea: Scheduling of long running applications in shared production clusters," in *Proc. of ACM EuroSys'18*, 2018.

[13] Z. Huang, K.-J. Lin, S.-Y. Yu, and J. Y. jen Hsu, "Co-locating services in IoT systems to minimize the communication energy cost," *Journal of Innovation in Digital Ecosystems*, vol. 1, no. 1, pp. 47–57, 2014.

[14] M. I. Naas, P. R. Parvedy, J. Boukhobza, and L. Lemarchand, "iFogStor: An IoT data placement strategy for fog infrastructure," in *Proc. of IEEE ICFEC'17*, 2017, pp. 97–104.

[15] F. Faticanti, F. De Pellegrini, D. Siracusa, D. Santoro, and S. Cretti, "Throughput-aware partitioning and placement of applications in fog computing," *IEEE Trans. on Netw. and Service Manag.*, vol. 17, no. 4, pp. 2436–2450, 2020.

[16] S. Pallewatta, V. Kostakos, and R. Buyya, "Microservices-based IoT application placement within heterogeneous and resource constrained fog computing environments," in *Proc. of IEEE/ACM UCC'19*, 2019, p. 71–81.

[17] O. Skarlat, M. Nardelli, S. Schulte, M. Borkowski, and P. Leitner, "Optimized IoT service placement in the fog," *Service Oriented Computing and Applications*, vol. 11, pp. 427 –443, 2017.

[18] Z. Wen, R. Yang, P. Garraghan, T. Lin, J. Xu, and M. Rovatsos, "Fog orchestration for internet of things services," *IEEE Internet Computing*, vol. 21, no. 2, pp. 16–24, 2017.

[19] C. Guerrero, I. Lera, and C. Juiz, "Genetic algorithm for multi-objective optimization of container allocation in cloud architecture," *Journal of Grid Computing*, vol. 16, pp. 113–135, 2018.

[20] H. Zhao, S. Deng, Z. Liu, J. Yin, and S. Dustdar, "Distributed redundancy scheduling for microservice-based applications at the edge," *IEEE Trans. Serv. Comput.*, pp. 1–1, 2020.

[21] H. Song, F. Chauvel, and P. H. Nguyen, *Using Microservices to Customize Multi-tenant Software-as-a-Service*. Springer, 2020, pp. 299–331.

[22] Y. Niu, F. Liu, and Z. Li, "Load balancing across microservices," in *Proc. of IEEE INFOCOM'18*, 2018, pp. 198–206.

[23] M. A. Rodriguez and R. Buyya, "Container-based cluster orchestration systems: A taxonomy and future directions," *Softw. Pract. Exp.*, vol. 49, no. 5, pp. 698–719, 2019.

[24] H. V. Netto, A. F. Luiz, M. Correia, L. de Oliveira Rech, and C. P. Oliveira, "Koordinator: A service approach for replicating Docker containers in Kubernetes," in *Proc. of IEEE ISCC'18*, 2018, pp. 58–63.

[25] J. Santos, T. Wauters, B. Volckaert, and F. De Turck, "Towards network-aware resource provisioning in Kubernetes for fog computing applications," in *Proc. of IEEE NetSoft'19*, 2019, pp. 351–359.

[26] A. J. Fahs, G. Pierre, and E. Elmroth, "Voilà: Tail-latency-aware fog application replicas autoscaler," in *Proc. of MASCOTS'20*, 2020, pp. 1–8.

[27] F. Rossi, V. Cardellini, F. Lo Presti, and M. Nardelli, "Geo-distributed efficient deployment of containers with kubernetes," *Computer Communications*, vol. 159, pp. 161–174, 2020.

[28] The Kubernetes Authors, "Scheduling framework — kubernetes." [Online]. Available: https://kubernetes.io/docs/concepts/scheduling-eviction/scheduling-framework/

[29] ——, "Scheduler configuration — kubernetes." [Online]. Available: https://kubernetes.io/docs/reference/scheduling/config/

[30] M. Symeonides, Z. Georgiou, D. Trihinas, G. Pallis, and M. D. Dikaiakos, "Fogify: A fog computing emulation framework," in *Proc. of ACM/IEEE SEC'20*, 2020.

[31] ——, "Demo: Emulating geo-distributed fog services," in *Proc. of the ACM/IEEE SEC'20*, 2020.

[32] "Using rabbitmq cluster kubernetes operator." [Online]. Available: https://www.rabbitmq.com/kubernetes/operator/using-operator.html

[33] T. Rausch, C. Lachner, P. A. Frangoudis, P. Raith, and S. Dustdar, "Synthesizing plausible infrastructure configurations for evaluating edge computing systems," in *Proc. of USENIX HotEdge'20*, 2020.

[34] Microsoft, "Linux virtual machines pricing — microsoft azure." [Online]. Available: https://azure.microsoft.com/en-us/pricing/details/virtual-machines/linux/