Improving Container Deployment in Edge Computing Using the Infrastructure Aware Scheduling Algorithm

Luis Augusto Dias Knob*†, Carlos Henrique Kayser*, Tiago Ferreto*
*Pontifícia Universidade Católica do Rio Grande do Sul (PUCRS)
Porto Alegre, Brazil
†Federal Institute of Rio Grande do Sul - Campus Sertao
Rodovia RS 135, Km 25 – Sertão, Brazil
carlos.kayser,luis.knob@edu.pucrs.br, tiago.ferreto@pucrs.br

2021 IEEE Symposium on Computers and Communications (ISCC) | 978-1-6654-2744-9/21/$31.00 ©2021 IEEE | DOI: 10.1109/ISCC53001.2021.9631490

***Abstract*—The increase of applications with low latency requirements has promoted edge computing as an enabler technology by bringing computational resources closer to end-users. However, this new paradigm presents several challenges, such as the fast and continuous provision of applications on geographically distributed heterogeneous devices at the edge, often with constraint resources. Currently, there are several strategies for scheduling applications in edge environments to decrease application deployment time. However, they do not consider the network bandwidth available or the download queue on each node. In this paper, we present a new scheduling strategy called *Infrastructure Aware* considering these characteristics. We validated our proposal through simulation. Results show that the *Infrastructure Aware* scheduling algorithm can, on average, decrease the deployment latency by more than 52% and 40% compared to the Kube-Scheduler and Layer Locality strategies, respectively.**

***Index Terms*—Container Management, Edge Computing, Orchestration, Container Scheduling**

I. Introduction

With more than 67% of the world's population owning a mobile subscription, and with a forecast of over 25 billion highly connected equipment by 2025 [1], mobile operators play a key role in the growth of the Internet and the development of new services, such as augmented reality and natural language processing. Among the requirements for enabling these applications, low latency and processing power near end-users are the most important ones.

Cloud Computing has a great responsibility in implementing such applications until today, but it does not meet the requirements for their large-scale use. As a result, new paradigms, such as Multi-Access Edge Computing and Fog Computing, have emerged to bring the cloud closer to the end-user. These paradigms, generalized as Edge Computing, are creating a continuum, where servers or data centers can be deployed in a variety of locations, such as wireless access points or radio antennas, to deploy resources like storage and processing power [2].

However, the geographic distribution, the heterogeneity of the physical infrastructures, and the requirements of the applications in these scenarios present management requirements that are not fully met by any solution currently available. In several works [3] [4], the use of containers is considered an essential technology for the implementation of near-to-user solutions. Still, its orchestration is today firmly focused on traditional data center infrastructure, with a specific location and low latency between nodes.

Despite being the *de facto* standard orchestrator in almost every cloud, the *Kubernetes* behavior lies on the same problems. Its default scheduling algorithm, called Kube-scheduler, distributes the applications on the topology almost equally between the set of available nodes without considering the heterogeneity of the network edge links. Unfortunately, this approach can increase the total time needed to deploy a given application (deployment latency), mainly because nodes with constrained links may receive the same volume of applications as nodes with high-capacity links. Some methods to decrease the deployment latency were already proposed in the literature, with a focus on the cache usage [5] or changes on the container runtime [6]. In addition, new scheduling algorithms were also presented by [7] and [8]. These works extend the Kube-scheduler adding new priorities based on latency and cache usage, showing promise results in a constrained scenario.

In edge scenarios, the main reason for the deployment latency lies in the image download from an external registry, which can take several seconds on constrained-resource nodes. Therefore, we understand the scheduler must know the bandwidth available and the current non-finished requests on each node to speed up the deployment process on edge computing. However, existing scheduling algorithms do not consider these two constraints in their allocation process.

With that in mind, we propose in this paper a new scheduling algorithm, called *Infrastructure Aware*, that seeks to reduce the deployment latency through a better container placement by using the download queue and available network bandwidth as priorities to the scheduler. Furthermore, we also integrate the layer match as proposed by [8] in our solution. At last, we evaluate our scheduling algorithm against the image and layer match schedulers, as also the Kube-scheduler, in a simulated scenario using a large number of applications generated using the Top 24 downloaded images from DockerHub [9].

The remainder of the paper is organized as follows. Section II presents the main container schedule strategies. After, in Section III, we describe the *Infrastructure Aware* scheduling algorithm. Section IV discusses the experiments and results. Finally, Section V concludes the paper.

II. Container Schedule Strategies

This section describes the scheduling process of the Kubernetes [10], one of the major frameworks for container orchestration available currently, and new strategies proposed in the literature to schedule edge applications.

A. Kubernetes Scheduling Strategy

In this framework, an application (i.e., microservices) is provisioned and managed by the Kubernetes as a pod. A pod consists of grouping one or more containers with shared resources, such as network and storage, and definitions of how to run each container.

Kube-scheduler [10] is the component responsible for finding the best node to host a pod in the Kubernetes cluster. It is the default component of the Kubernetes cluster for scheduling decisions. This scheduling mechanism uses policies based on predicates and priorities to delimit the eligible nodes and prioritize them to select the node to host container-based applications. It determines the most appropriate node in two phases: a) the filtering process; and b) the scoring process. The first one selects eligible nodes based on predicates policies, while the second one ranks nodes based on priorities.

Kube-scheduler supports predicate-based policies. Some examples include:

1) **PodFitsResources**: This policy checks if the free computational resources of the node, such as CPU and memory, are enough to support the pod requirements. Otherwise, it classifies the node as ineligible and removes it from the ranking step;
2) **MatchNodeSelector**: This policy allows to filter the set of nodes based on labels across pod's node selector and node's labels;
3) **NoDiskConflict**: This policy checks if the node has capacity enough in terms of volumes requested by the pod;

After the filtering process, Kube-scheduler uses policies based on priorities to rank the nodes. Some of the policies supported are:

1) **SelectorSpreadPriority**: This policy tends to spread the pods across nodes, taking into account the pods that belong to the same service;
2) **LeastRequestedPriority**: The score of the node is calculated taking into account the amount of free computational resources, such as CPU and memory, balancing the workload between nodes;
3) **MostRequestedPriority**: In this policy, the score of the node is calculated similarly to the LeastRequestedPriority policy; the main difference is that this policy prioritizes the nodes with the most requested resources;
4) **NodeAffinityPriority**: The node that has the labels specified by the pod's node selector are ranked with the highest scores; that is, this policy favors them;
5) **InterPodAffinityPriority**: Unlike the previous policy, this policy favors the nodes that already have some pod allocated based on pod affinity rules. If the node has a pod allocated defined in the pod affinity rules of the requested pod, it will receive a higher score;

At the end of the filtering and ranking process, Kube-scheduler selects the node with the highest score to provision the pod. If there is a tie, Kube-scheduler chooses one of these at random.

However, in an edge computing scenario composed of heterogeneous devices, these policies may not be enough to deploy container-based applications. For instance, they don't consider potentially scarce resources at the edge, such as network bandwidth, to meet the requirements of applications without compromising their quality of service.

B. Dependency Aware Strategy

Fu et al. [8] proposed new dependency scheduling policies to rank the nodes based on how much their cache overlaps with those of the requested pod. It aims to take advantage of the nodes' local cache and speed up the provisioning time of container-based applications. The authors proposed two approaches: a) image-match approach; and b) layer-match approach.

As the name suggests, the image-match approach favors the nodes that already have locally the image(s) of the pod requested. So, for example, considering deploying a pod composed by the image *mongodb:4.4.6*, this policy will give a node that already has this dependency locally a higher score.

The second one, the layer-match approach, has practically the same behavior. However, this policy favors the nodes with the most dependencies locally at the layer level rather than the image level, i.e., the nodes with more dependencies attended locally will receive the higher scores.

Considering that some dependencies are allocated already in the node, this strategy presents benefits concerning the application provisioning time and the overall cluster storage utilization. That relies on the fact that container images are created from a base image and may share equal layers.

Although these policies reduce the total provisioning time of applications, their efficiency may be impacted by network infrastructure heterogeneity. For instance, it may be faster to deploy an entire container with all layers in a new node with high bandwidth capacity instead of sending a single layer to a constrained node. In addition, these policies do not consider the download queue on the nodes since more applications can be waiting to be downloaded at the node, increasing the provisioning time.

C. Others Edge Schedulers

Other schedulers were proposed based on distinct objectives that can also affect an edge infrastructure. For example, in [7] the authors introduced a policy that makes use of round trip time (RTT) labels attached to the nodes to decide the most suitable place to deploy an application based on its configuration (i.e., target location). Additionally, the policy checks if the most appropriate node has enough bandwidth capacity to support the application's requirements. Results show that the proposed approach compared to the Kube-scheduler achieved a reduction of 80% in terms of network latency. However, it only considers the application's latency requirement during execution in a given destination region, but it does not consider its deployment phase.

In [11], the authors propose a greedy scheduling algorithm called FPA (Fog Placement Algorithm) to improve the total throughput between all applications in a Fog Computing scenario. It allocates the *Fog modules* on the same region where the *control microservice* resides, which is usually placed in a bigger server, in the fog or the cloud. Although the authors do not use information about the bandwidth on the scheduler, the paper presents results showing that one of the main problems to a more significant throughput was the intra-region connections that can create bottlenecks on communication. Furthermore, we understand that the same problem may happen in the instantiation phase, where the bottlenecks will increase the total amount of time needed to deploy the applications.

III. Infrastructure Aware Scheduling

In Edge Computing, placing container-based microservices in edge nodes that can guarantee minimal latency is essential. This characteristic is the main reason for its adoption, instead of only relying on the cloud. However, choosing the right edge nodes that minimize the time required to deploy the containers is also necessary, especially when dealing with microservices that may present a short-term existence.

Kube-scheduler is the default component in Kubernetes responsible for choosing the edge nodes to deploy a container. As presented in Section II, it provides different scheduling policies to handle several cases. However, no policy considers metrics, such as the network bandwidth, which may significantly impact the deployment time.

We propose the *Infrastructure Aware* scheduling algorithm for reducing deployment time while considering different metrics such as download queue on each node and available network bandwidth.

The main goal of the *Infrastructure Aware* scheduling algorithm is to speed up the application deployment time while avoiding congesting the network interface of the edge nodes. Furthermore, it can be easily implemented in container orchestration frameworks, such as Kubernetes since it does not require any modification in the infrastructure.

Algorithm 1 presents the *Infrastructure Aware* scheduling algorithm. Initially, it creates a dictionary (`aL`) composed of the layer digest (key) and the layer size (value) (lines 1-3). After that, it computes, for every eligible node, the time to instantiate the application (`ttInst`). It considers the layers already cached locally, the queue of layers waiting to be downloaded by the node, and the bandwidth between the container register (e.g., Docker Hub, GitHub Container Registry) and the node (lines 4-15). Finally, it rescales the time to instantiate the application between zero and `w`, where `w` is the weight of this policy on the other predicates (lines 16-24).

Algorithm 1: Least Congested Node Priority

```
Input: $application$: application to be scheduled;
       $chosts$: list of the container nodes; $w$: default
       weight
Output: List of the container hosts with updated score
1  aL ← {};
2  for $layer \in application_{image}$ do
3      $\texttt{aL}_{digest} \leftarrow size(layer)$;
4  ttInst ← {};
5  for $c \in chosts$ do
6      mL ← {};
7      for $app \in c_{scheduleApps}$ do
8          for $layer \in app_{layers}$ do
9              $\texttt{mL}_{app} \leftarrow size(layer)$;
10     cL ← {};
11     for $image \in c_{cache}$ do
12         for $layer \in image_{layers}$ do
13             $\texttt{cL}_{image} \leftarrow size(layer)$;
14     $\texttt{s} \leftarrow \sum_{i \in (aL \setminus mL \setminus cL)} i$;
15     $\texttt{ttInst}_c \leftarrow s \div bandwidth(c)$;
16 $\texttt{minTime} \leftarrow \min_{\forall t \in ttInst} t_{time}$;
17 $\texttt{maxTime} \leftarrow \max_{\forall t \in ttInst} t_{time}$;
18 if $minTime = maxtime$ then
19     for $c \in chosts$ do
20         $c_{score} += w$;
21 else
22     for $c \in chosts$ do
23         $\texttt{score} \leftarrow (w - ((w - 0) \div (maxTime - minTime) \times (ttInst_c - minTime))$;
24         $c_{score} += score$;
25 return chosts
```

Some considerations can be made on the algorithm:

1) **Empty cache**: If at any time there are no images stored locally in the nodes' cache and also no images to be download in the queue, the algorithm will favor the nodes with the higher network bandwidth to decrease the application's deployment time;
2) **Queue with applications**: The algorithm gives the highest scores for nodes that, even having applications

in their download queue, can download all dependencies in the shortest time;

3) **Node bandwidth**: The algorithm considers that the network bandwidth between the edge node and the registry is periodically calculated since it is hard to determine with precision the bandwidth between nodes due to the variability of links' utilization.

Even if a node has the most bandwidth, it doesn't mean it can provision the given application in the shortest time as other nodes may already have locally or in the download queue the dependencies of the requested application. However, since the proposed policy ends up centralizing the applications on a given set of nodes, the high availability of services is affected, which also entails the load imbalance between nodes. In addition, it does not check whether the node has sufficient bandwidth capacity to support the requirements of the requested application.

IV. Evaluation

This section presents an evaluation of the Infrastructure Aware scheduling algorithm. The algorithm is compared to the Kube-scheduler, and the algorithms presented in [8] (Image and layer locality) in a simulated scenario based on the Brazilian research network topology using Docker Hub images. The metrics used for comparison include deployment latency, node storage utilization, among others.

A. Simulator

In order to perform the evaluation, an edge simulator was implemented in Python 3. The simulator uses the NetworkX library and models the orchestration of containers in an edge computing infrastructure. The main focus of the simulator is on the deployment process and image distribution from a registry to the edge nodes considering network topology behavior. The modeling of the network capacity variability is based on a fair sharing schedule policy based on the max-min fairness algorithm. Furthermore, it is implemented using the algorithm proposed by [12], which focuses on sharing an infrastructure based on an equal distribution between flows and maximum bandwidth usage to each link. This approach results in a more realistic simulation considering the network bottlenecks when provisioning containers in edge nodes.

The simulator allows the implementation of different scheduling strategies (e.g., kube-scheduler and random scheduler). In the case of the Kubernetes scheduler, we simplify the implementation presents on the official source code [13], where our simulator can adding and removing predicates, priorities, and setting different weights to each one. This process allows fine-grained control of the simulation scenario and a better understanding of how priorities affect container distribution in the infrastructure.

B. Topology

In order to evaluate the policies in a more realistic scenario, we configured the edge simulator to simulate the Ipê (Brazilian Research Network) network topology; that is, we set up the simulator with all the points of presence, network connections, link speed, and other network topology features. This topology interconnects all Brazilian universities and research institutes through 28 Points of Presence (PoPs) distributed over the country (Figure 1). The topology also connects to several international research networks, such as Clara (Latin America), Internet2 (United States), and Géant (Europe). In the experiments, the actual bandwidth and latency for each link were used, as described in [14].

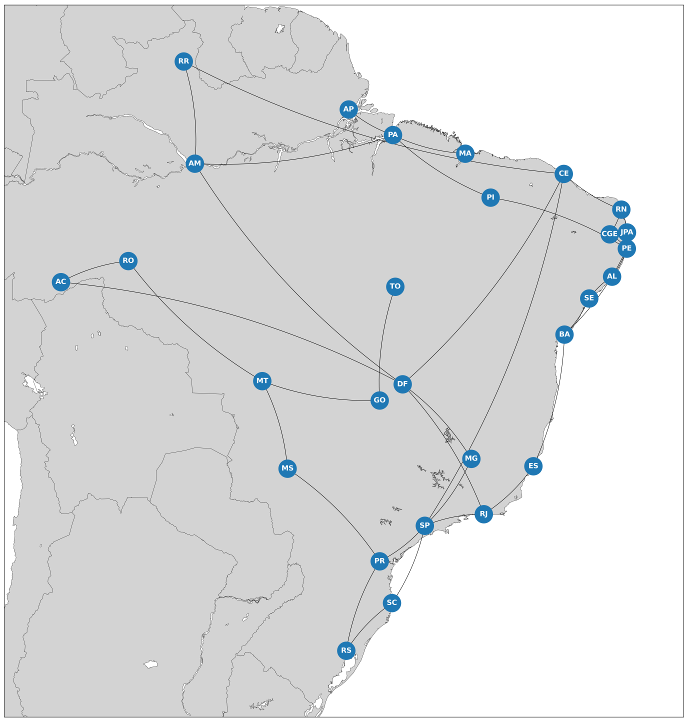

Figure 1: Ipê Brazilian Research Network Topology

To enable the comparison with the other scheduling algorithms (Kube-scheduler, Image and Layer locality [8]), we extended the Ipê topology. Each PoP included a large node named Server Node and five small nodes named Edge Nodes. The nodes differ in the bandwidth available on each one. For instance, the server node has 100 Mbps, and the smaller nodes have between 10 and 60 Mbps of bandwidth (distributed uniformly). For simplicity, the simulation only considers the network utilization for container provisioning, from the registry to the server or edge nodes. Any other communication that would be active in a real network is ignored.

The registry, where all container images are located, was placed on PoP-São Paulo (SP). This PoP is a central one in the topology and is the primary connection to cloud providers [15]. It was given a bandwidth of 10Gbps to avoid having the Registry Node as a bottleneck on the simulation.

C. Workload

The workload used in the simulation is based on real images on the Docker Hub [9]. Therefore, we selected the twenty-four most downloaded images, excluding base images, and allocated them in the Registry Node. The images have a total size of 3436.45 MB (an average of 143.19 MB per image). However, since several images share layers, the maximum amount that a given node needs to download to have all applications is 2152.78 MB (37% of similarity between images). A random number of applications between 5 and 25 (distributed uniformly) is created for each image. And for each application, a random number of replicas between 2 and 5 (distributed uniformly). Furthermore, each application has a random scheduling time between 0 and 1000 seconds (distributed uniformly), which defines when the application will be considered for scheduling in the topology. Table I presents the parameters used in the experiments.

Parameter	***Value***
Server Nodes	*28*
Server Links	*100 Mbps*
Edge Links	*10 - 60 Mbps*
Number of Images	*24*
Number of Applications	*350*
Number of Replicas	*1250*

Table I: Simulation parameters

D. Results

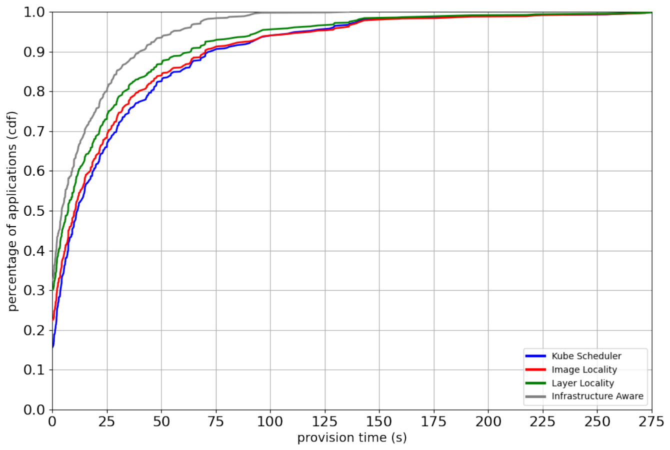

Figure 2: Provision time by scheduling algorithm

1) Deployment Latency: Figure 2 presents the CDF of the deployment latency for each application replica on the topology. The results show the gains of using information about the infrastructure on the container scheduling in edge computing. Even with a small number of containers allocated in each host node, all algorithms present a better result than the default behavior on the Kube-scheduler. As expected, when we increase the information's granularity used by the scheduler, the total amount of time needed to instantiate the applications decrease. On average, the Image Locality deploys the replicas 6% faster than the Kube-scheduler, while the Layer Locality is 25% better. Furthermore, with the download queue and the expected download duration predicates, the Infrastructure Aware is 37% and 52% smaller than the Layer Locality and the Kube-scheduler, respectively. Almost the same results can also be seen on the 99% percentile. The containers are deployed at most in 90.69 seconds in the Infrastructure Aware, while spent 192.11 and 207.27 seconds to deploy the same application with Layer Locality and Kube Scheduler.

It is important to notice that the Image and Layer locality will only present consistent results after a long period on the topology, i.e., after several containers become fully downloaded on the edge nodes and the layers be available on the cache. Furthermore, cache substitution policies can also decrease the performance of these schedulers, while pre-cached images can positively influence that. However, these problems do not affect Infrastructure Aware since it can reduce the deployment latency even when the container is not available in any node of the region by allocating them to a less congested node.

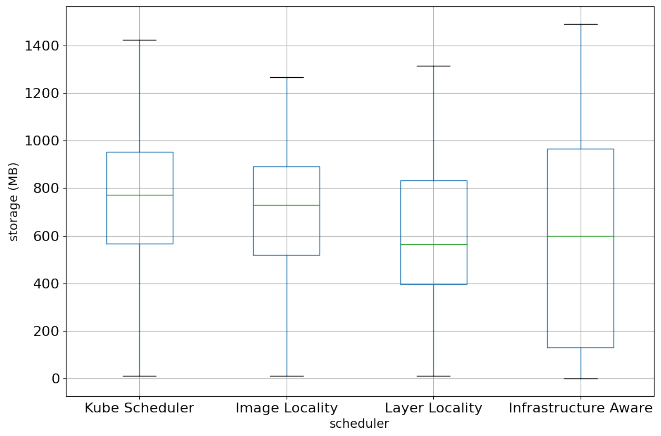

Figure 3: Node storage utilization by each scheduling algorithm

2) Node Storage Utilization: Usually, Kube-scheduler has as default behavior the equal distribution of containers between nodes that surpass the filtering phase on the scheduler. In Figure 3, this can be seen by the well-distributed percentiles presented by this algorithm on the node storage usage. However, the Image and Layer Locality schedulers tend to centralize on few nodes that already contain the image or layers from that given application. It can be identified by the decrease of the average storage used on each node, with the Layer Locality having the best results in comparison to the Kube-scheduler, using only 595.535 MB instead of 749.31 MB. This result is also present in the network utilization, with the total amount of data transferred on the infrastructure been 25% bigger on the Kube-scheduler than the Layer Locality. However, the Infrastructure Aware scheduling has a more dynamic behavior, adapting between both trends, sometimes spreading the applications to ensure the best network utilization and sometimes using the cache nodes to decrease the provision time. This

Algorithm	*Provision time (s)*			*Cache (un)*		*Storage Usage (MB)*				*Distribution (un)*		
	avg	*99%*	*max*	*hit*	*miss*	*min*	*avg*	*max*	*stdDev*	*min*	*max*	*used nodes*
Kube-scheduler	28.03	207.27	279.31	2042	8169	9.94	749.31	1423.77	2.57	2	12	168
Image Locality	26.56	221.84	279.31	2618	7593	9.94	701.40	1266.60	2.60	2	12	168
Layer Locality	22.20	192.11	279.31	3808	6403	9.94	595.35	1314.16	3.09	2	16	168
Infrastructure Aware	13.47	90.69	136.04	3913	6248	0.00	584.31	1490.30	5.46	0	16	150

Table II: Additional statistics from the simulations

behavior is represented in the figure by the most significant differences between percentiles, where the storage utilization was largely distinct on each node. However, even with this, the total amount of data transferred on the network was only 0.8% bigger, and the average storage used was more than 11 MB smaller than the Layer Locality.

3) Additional Metrics: We also verified the distribution in applications deployed per node. The Layer Locality and the Infrastructure Aware present the worst distribution with a standard deviation of 3.09 and 5.46 from the average, respectively. Finally, we also collected the cache hits and misses from the simulation, i.e., when a layer can be found in the nodes' cache during provisioning. As expected, the best results also occur on the Layer Locality and the Infrastructure Aware, with 37.29% and 38.31% of the cache hit, respectively. Table II summarizes the results obtained in the simulation.

V. Conclusion

This paper presents the *Infrastructure Aware* scheduling algorithm, a novel approach to decrease the deployment latency of containers on an edge topology. It excels current algorithms by using the network bandwidth and the downloading queues on each node as priorities for the scheduling process. Together with the Layer locality algorithm [8], these new priorities can, on average, decrease the deployment latency by more than 52% compared to the Kubernetes default behavior. Notwithstanding, Infrastructure Aware is 40% better than using just the Layer locality priority, mainly because it optimizes the deployment process even in regions where a given image has no cached layers.

We understand that the *Infrastructure Aware* scheduling algorithm may lead to an over-utilization of nodes with a large number of network resources, limiting the benefits shown by adding more constrained nodes in a given region. We also evaluated that, as we score the network priority by a snapshot in a given moment on the bandwidth usage, this may lead to undesired results. In the future, these problems can be addressed by tweaking the ratio used by each priority on the scheduler and the substitution from instantaneous bandwidth snapshot by the network usage average on the policy. Besides that, we also want to improve the simulation scenarios, adding more constraints and node limitations, like limited cache size, and will implement the predicate on Kubernetes to evaluate our scheduler in a real cluster.

VI. Acknowledgements

This study was supported by the Federal Institute of Education, Science and Technology of Rio Grande do Sul (IFRS) and the PDTI Program, funded by Dell Computadores do Brasil Ltda (Law 8.248/91). The authors acknowledge the High-Performance Computing Laboratory of the Pontifical Catholic University of Rio Grande do Sul (LAD-IDEIA/PUCRS, Brazil) for providing support and technological resources, which have contributed to the development of this project and to the results reported within this research. This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior – Brasil (CAPES) – Finance Code 001.

References

[1] GSMA, "The mobile economy 2019," *GSMA Association*, 2019.
[2] T. Taleb, K. Samdanis, B. Mada, H. Flinck, S. Dutta, and D. Sabella, "On multi-access edge computing: A survey of the emerging 5g network edge cloud architecture and orchestration," *IEEE Communications Surveys Tutorials*, vol. 19, no. 3, pp. 1657–1681, thirdquarter 2017.
[3] G. A. Carella and T. Magedanz, "Open baton: A framework for virtual network function management and orchestration for emerging software-based 5g networks," *Newsletter*, vol. 2016, p. 190, 2015.
[4] E. Schiller, N. Nikaein, E. Kalogeiton, M. Gasparyan, and T. Braun, "Cds-mec: Nfv/sdn-based application management for mec in 5g systems," *Computer Networks*, vol. 135, p. 96–107, 2018.
[5] J. Darrous, T. Lambert, and S. Ibrahim, "On the importance of container image placement for service provisioning in the edge," in *2019 28th International Conference on Computer Communication and Networks (ICCCN)*, 2019, pp. 1–9.
[6] A. Ahmed and G. Pierre, "Docker container deployment in fog computing infrastructures," in *2018 IEEE International Conference on Edge Computing (EDGE)*, 2018, pp. 1–8.
[7] J. Santos, T. Wauters, B. Volckaert, and F. De Turck, "Towards network-aware resource provisioning in kubernetes for fog computing applications," in *2019 IEEE Conference on Network Softwarization (NetSoft)*, 2019, pp. 351–359.
[8] S. Fu, R. Mittal, L. Zhang, and S. Ratnasamy, "Fast and efficient container startup at the edge via dependency scheduling," in *3rd USENIX Workshop on Hot Topics in Edge Computing (HotEdge 20)*, 2020.
[9] D. Inc, "Docker Hub." [Online]. Available: https://hub.docker.com/
[10] "Kubernetes," 2021. [Online]. Available: https://kubernetes.io/
[11] F. Faticanti, F. D. Pellegrini, D. Siracusa, D. Santoro, and S. Cretti, "Cutting throughput on the edge: App-aware placement in fog computing," *CoRR*, vol. abs/1810.04442, 2018. [Online]. Available: http://arxiv.org/abs/1810.04442
[12] D. P. Bertsekas, R. G. Gallager, and P. Humblet, *Data networks*. Prentice-Hall International New Jersey, 1992, vol. 2.
[13] Kubernetes, "Kube-scheduler component configs," 2021. [Online]. Available: https://github.com/kubernetes/kube-scheduler
[14] RNP, "Rede ipê." [Online]. Available: https://www.rnp.br/sistema-rnp/rede-ipe
[15] NIC.BR, "ix.br." [Online]. Available: https://ix.br/