ESCOLA POLITÉCNICA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO
MESTRADO EM CIÊNCIA DA COMPUTAÇÃO

CARLOS HENRIQUE KAYSER

MINIMIZING CONTAINER-BASED APPLICATIONS SLA VIOLATIONS ON EDGE COMPUTING ENVIRONMENTS

Porto Alegre
2022

PONTIFICAL CATHOLIC UNIVERSITY OF RIO GRANDE DO SUL
SCHOOL OF TECHNOLOGY
COMPUTER SCIENCE GRADUATE PROGRAM

MINIMIZING CONTAINER-BASED APPLICATIONS SLA VIOLATIONS ON EDGE COMPUTING ENVIRONMENTS

CARLOS HENRIQUE KAYSER

Master Thesis submitted to the Pontifical Catholic University of Rio Grande do Sul in partial fulfillment of the requirements for the degree of Master in Computer Science.

Advisor: Prof. Dr. Tiago Coelho Ferreto

Porto Alegre
2022

Ficha Catalográfica

K23m Kayser, Carlos Henrique

Minimizing container-based applications SLA violations on edge computing environments / Carlos Henrique Kayser. – 2022.
46 f.
Dissertação (Mestrado) – Programa de Pós-Graduação em Ciência da Computação, PUCRS.

Orientador: Prof. Dr. Tiago Coelho Ferreto.

1. Edge Computing. 2. Container-based Applications. 3. Container Orchestration. 4. Container Scheduling. I. Ferreto, Tiago Coelho. II. Título.

CARLOS HENRIQUE KAYSER

MINIMIZING CONTAINER-BASED APPLICATIONS SLA VIOLATIONS ON EDGE COMPUTING ENVIRONMENTS

This Master Thesis has been submitted in partial fulfillment of the requirements for the degree of Master in Computer Science, of the Computer Science Graduate Program, School of Technology of the Pontifical Catholic University of Rio Grande do Sul

Sanctioned on March 25th, 2022.

COMMITTEE MEMBERS:

Primeiramente, gostaria de agradecer toda a minha família, por todo apoio, carinho e incentivo. Obrigado também pela paciência e compreensão nos momentos de minha ausência.

Agradeço ao meu orientador, Tiago Ferreto, pelas oportunidades e principalmente por toda a confiança e ensinamentos. Agradeço também aos demais professores do Programa de Pós-Graduação em Ciência da Computação, pela qualidade das disciplinas mesmo em tempos atípicos.

Por fim, agradeço também a todos os meus novos amigos que conquistei durante esse período, por toda ajuda e companheirismo.

ACKNOWLEDGMENTS

This work was supported by the PDTI Program, funded by Dell Computadores do Brasil Ltda (Law 8.248 / 91). The author 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.

MINIMIZANDO VIOLAÇÕES DE SLA DE APLICAÇÕES BASEADAS EM CONTAINERS EM AMBIENTES DE COMPUTAÇÃO NA BORDA

RESUMO

O surgimento de aplicações com requisitos rígidos como baixa latência e privacidade motivou a aproximação de recursos computacionais e usuários na borda da rede devido às dificuldades do paradigma de computação em nuvem em suprir tais necessidades. Nesse novo paradigma de computação distribuída, assim como em computação em nuvem, as técnicas de virtualização baseadas em contêiner também são consideradas para provisionamento de aplicações devido ao baixo consumo de recursos, rápido provisionamento e baixo espaço de armazenamento em comparação com máquinas virtuais (VM). No entanto, a alta variabilidade da capacidade computacional e largura de banda dos nós de borda impactam diretamente no tempo de provisionamento das aplicações em um ambiente de computação de borda. Além disso, a localização dos usuários finais também é um fator importante a ser considerado ao escalonar as aplicações, pois a distância entre os usuários e os nós de borda afeta a latência da comunicação. Nesse contexto, este trabalho apresenta um novo algoritmo de escalonamento, chamado *Latency and Provisioning Time SLA Driven Scheduler* (LPSLA), que coordena o provisionamento de aplicações em infraestruturas de borda para minimizar as violações de *Service Legel Agreements* (SLA) em termos de latência e tempo de provisionamento. O algoritmo proposto considera a latência entre a localização dos usuários finais e os nós de borda e a capacidade dos nós de borda em baixar aplicações baseadas em contêiners. Como resultado, a solução proposta é capaz de minimizar as violações de SLA em todos os cenários avaliados.

Palavras-Chave: Computação na Borda, Aplicações Baseadas em Contêineres, Orquestração de Contêineres, Escalonamento de Contêineres.

MINIMIZING CONTAINER-BASED APPLICATIONS SLA VIOLATIONS ON EDGE COMPUTING ENVIRONMENTS

ABSTRACT

The emergence of applications with strict requirements such as low latency and privacy motivated the approximation of computing resources and users at the network's edge due to the difficulties of the cloud computing paradigm in fulfilling such needs. In this new distributed computing paradigm, like cloud computing, container-based virtualization techniques are also considered for application provisioning due to low resource consumption, fast provisioning, and low storage footprint compared to virtual machines (VM). However, the high variability of the edge nodes' computational capacity and bandwidth directly impact the applications' provisioning time in an edge computing environment. In addition, the end-users location is also an important factor to consider when scheduling applications, as the distance between end-users and edge nodes impacts communication latency. In this context, this work presents a novel scheduling algorithm, called *Latency and Provisioning Time SLA Driven Scheduler* (LPSLA), which coordinates application provisioning on edge infrastructures to minimize latency and provisioning time Service Legel Agreements (SLA) violations. It considers the latency between the end-users location and edge nodes and the capacity of edge nodes in downloading the container-based applications. As a result, the proposed solution is capable of minimizing the SLA violations in all evaluated scenarios.

Keywords: Edge Computing, Container-based Applications, Container Orchestration, Container Scheduling.

LIST OF FIGURES

LIST OF TABLES

LIST OF ALGORITHMS

CONTENTS

1. INTRODUCTION

The cloud computing paradigm allows on-demand remote access to computational resources such as servers, network, and storage hosted on data centers, sometimes geographically faraway over the Internet [26]. However, the emergence of applications such as augmented reality, autonomous vehicles, smart home and smart city, with strict requirements such as low latency, bandwidth usage, and privacy motivated the approximation of users and computational resources at the edge of the network, giving rise to a new paradigm of distributed computing called *edge computing* [41, 36, 40, 37].

Like cloud computing, application provisioning and management are needed in the edge computing scenario, where container-based virtualization techniques are also used [36, 21]. According to Ismail et al. [19], Docker is a good candidate for enabling virtualization in edge computing environments. In addition to being a much lighter and faster alternative compared to virtual machines (VM), it has a shorter provisioning time. However, during the deployment phase, as well as virtual machines, the Docker images must be transferred to one or more servers at the edge, which can result in a long provisioning time depending on the server and its bandwidth [1].

There are several challenges in edge computing, such as security, reliability, and privacy [21]. However, one of the main challenges is resource allocation, as application provisioning must take into account the heterogeneity of computational nodes and network links, application requirements (i.e., low-latency, privacy), latency and bandwidth of network links, and user location and mobility in order to provide a better quality-of-experience (QoE) and quality-of-service (QoS) [33, 21, 25].

Nevertheless, popular container-based application provisioning platforms (e.g., Kubernetes) overlook the heterogeneity of the nodes' capacity during the scheduling process, leading to a long provisioning time and stressing the nodes' bandwidth in edge computing scenarios. In addition, such platforms do not consider fundamental characteristics of an edge scenario, such as the latency of applications in relation to their users (e.g., response time) and application provisioning time.

In the context of this work, we understand that the application provisioning time, as well as the latency in relation to its users, could be modeled as Service Level Agreements (SLAs), as in cloud computing, where the cloud provider proposes to fulfill performance guarantees, such as high availability.

Recent investigations extend the functionality of provisioning platforms to meet the demands of edge computing environments by adopting location awareness to reduce end-to-end latency [34, 35], or minimizing provisioning time (i.e., start-up latency) [17, 12, 23]. Some of them focus on the placement of Docker layers to provide a cache for the edge servers [42], propose optimized registries [3], and improve the registry placement in the infrastructure topology [11].

Despite previous research attempts, resource scheduling in edge computing still presents challenges. Many edge computing workloads are short-lived applications (short tasks) and/or low-latency sensitive applications – which can be impacted by long provisioning times and high latency [17]. In this way, scheduling policies should consider the application provisioning time and communication latency between applications and their users (e.g., response time) so that the benefits of edge computing deployments are not undermined.

To tackle these problems, in this work, we propose a heuristic called *Latency and Provisioning Time SLA Driven Scheduler* (LPSLA), which focuses on enforcing container-based applications SLAs compliance in terms of provisioning time and latency during the application placement phase. A variation of LPSLA is also proposed, called LPSLA-DS, that focuses on meeting applications SLAs as well as preserving the most qualified edge nodes for tighter SLA applications. Both variations are also validated through simulations and compared with three strategies: 1) Kube-Scheduler [24], 2) Infrastructure-Aware [12], and 3) *Deployment Latency SLA Enforcement Scheduler* (DLSLA) [23]. Results show that the proposed solution overcomes the evaluated strategies on latency and provisioning time SLAs in all scenarios evaluated.

The remaining of this work is organized as follows: In Chapter 2, we present the main concepts related to the scope of this document. Based on such a discussion, Chapter 3 presents the works related to the scope of this work, Chapter 4 presents our proposed algorithm called LPSLA, while Chapter 5 presents the evaluation methodology and results based on simulations. Finally, Chapter 6 presents the final considerations of this study and gives directions for future research.

2. BACKGROUND AND MOTIVATION

This section presents a theoretical background on edge computing, container-based virtualization, and provisioning strategies, describing the main features of this distributed paradigm and its application provisioning challenges.

2.1 Edge Computing

Cloud computing is a model that allows on-demand access to computing resources hosted in remote data centers over the Internet. In this context, the provider can quickly provision computing resources in the pay-as-you-go model. That is, the users pay only for the resources and time they use, enabling users to increase their need for computing resources as needed. In this model, computing resources are centralized in large data centers positioned in strategic places worldwide, with a good supply of electricity, cooling, labor, and security.

According to the National Institute of Standards and Technology (NIST) [26], cloud computing is a model that enables on-demand access to configurable computing resources over the Internet, such as servers, services, and storage. It is defined by a set of essential characteristics and different service models. Some of these features are: 1) it provides users with the ability to provision computing resources as needed automatically and without human intervention on the part of the provider; 2) access can be made by different devices through the Internet; 3) computing resources need to meet the needs of their users on demand; 4) computing resources are scalable, to quickly meet user demands; and 5) there must be transparent ways to assess the use of computing resources.

However, the long distance between end-users and data centers is insufficient to provide low latency access to emerging applications, such as augmented reality. In addition, with the popularity of smart devices, such as sensors and security cameras, bandwidth utilization can end up being an issue when transferring data to be processed in the cloud.

Given these needs, a new distributed computing model has emerged called edge computing, where the computing resources such as servers and storage services (edge node) are placed closer to end-users and end-devices, such as Internet of Things (IoT) (e.g., sensors, actuators, and cameras), as illustrated in Figure 2.1, where the north-south communication interconnects the different layers (i.e., cloud computing, edge computing and end-devices), while the east-west communication interconnects the nodes on similar layers. The main objectives of this new paradigm are to provide lower latency for response time-sensitive applications and reduce the required bandwidth used by processing data closer to the data sources instead of sending and receiving information from remote data centers [8, 41, 40].

Unlike cloud computing, at the network's edge the processing power is limited due to the heterogeneity of the edge servers which can range from powerful computers to single board computers (e.g., Raspberry Pi). In addition, the network infrastructure may not be as stable, as servers can use from high-bandwidth links to wireless networks.

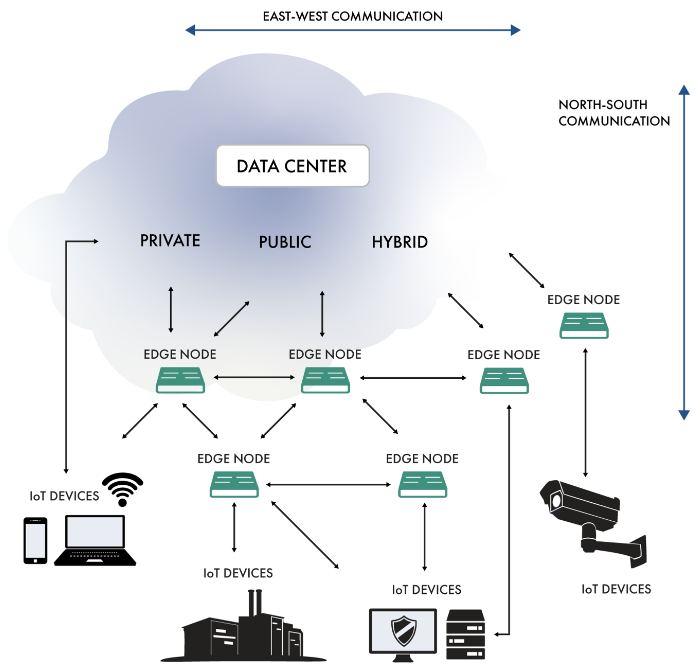


Figure 2.1: Edge computing architecture [8].

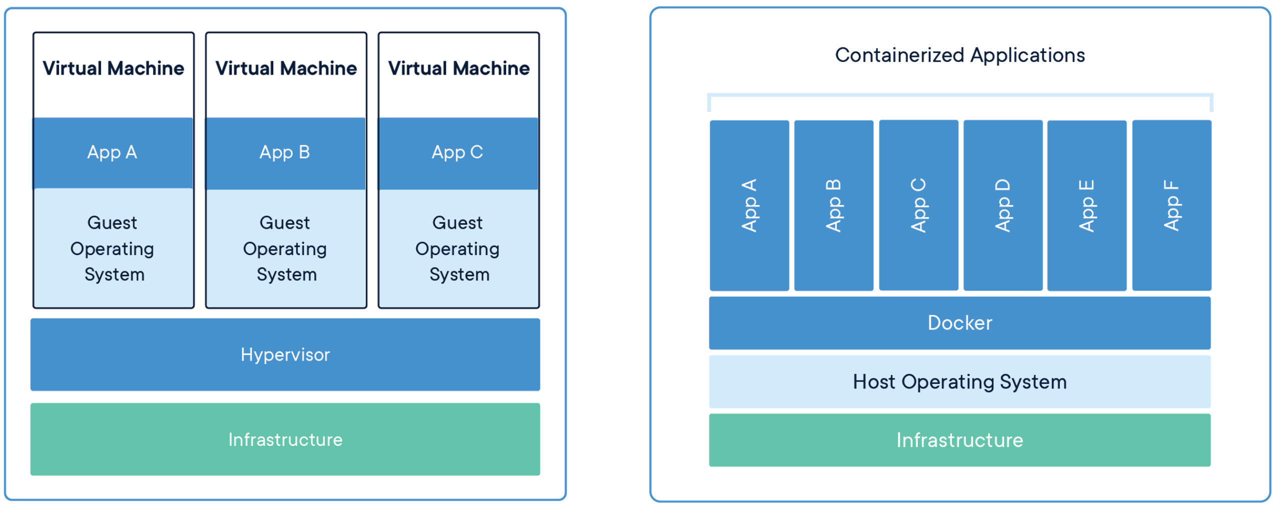
In the literature, there are different terms for edge computing, such as: cloudlets [38], micro data-centers [2] or fog computing [7]. The existence of different terminologies can lead to a misunderstanding of this paradigm. Despite the existence of small technological differences behind these terminologies, some works [6, 8] propose the use of the term edge computing to embrace all these perspectives.

The edge computing paradigm allows the use of computing resources (edge nodes) for processing as well as storage, just as it does in cloud computing through virtualization techniques such as hypervisors or containers [36], allowing encapsulation of applications and their dependencies.

2.2 Container-Based Virtualization

Container-based virtualization is a lighter alternative to resource virtualization compared to hypervisor-based virtualization. In container-based virtualization, all virtual resources (i.e., applications) share the same kernel and host operating system resources, as illustrated in Figure 2.2b. As a result, containers can be started very quickly, and it is possible to run hundreds of them on a physical host [4, 39, 27].

In comparison, virtualization based on a hypervisor, as shown in Figure 2.2a, enables the virtualization of multiple isolated complete virtual machines (VM), each with its own operating system, kernel, applications, and dependencies, resulting in excessive use of computing resources by virtualization, therefore, in addition to having a much slower boot process, hosts can provision just a limited number of VMs on a host [4, 39].



(a) Hypervisor-based virtualization.

(b) Container-based virtualization.

Figure 2.2: Container-based virtualization and hypervisor [14].

Container isolation is usually accomplished through host operating system resources. Considering the Linux operating system, resources such as network and file systems are isolated through the use of namespaces, while resource management is performed through *cgroups* [39]. The isolation of containers is considered inferior to hypervisor-based virtualization [44, 9]; however, currently, there are already initiatives to tackle this problem[1].

Several container-based virtualization technologies for Linux were created, such as Linux VServer (2001), OpenVZ (2005), and Linux Containers (LXC) in 2008 [39]. Despite that, the Docker platform was released in 2013, bringing an extended solution with a more friendly interface and portable images, facilitating the creation and distribution of containers [27].

[1]https://katacontainers.io/

2.2.1 Docker

Docker is a platform composed of tools that allow developing, distributing, and running applications and their dependencies encapsulated in light and isolated environments called containers. A container is an executable instance of a Docker image [14].

In addition to a Docker container running faster and consuming less resource compared with a virtual machine (VM) running on a hypervisor, the container images are portable and has a low storage footprint since different images can share the same layers. For example, images compiled from the same base image (e.g., alpine) can reuse the same base data to provision multiple containers. This virtualization also has a fast deployment and teardown, making it possible to scale quickly [19].

Creating an image usually originates from customizing a base image (e.g., Ubuntu, Alpine, Debian) through a configuration file called *Dockerfile*. Each command or instruction in the *Dockerfile* creates a layer above the chosen base image or the previous layer, and each layer has a *sha256 hash* as a unique identifier, allowing several different images to share identical layers [14].

As illustrated in Figure 2.3, the image's layers are read-only (R/O). When a container is provisioned from an image, a new layer is added on top of the others, called the container layer, which allows writing. The versioning of images occurs through the use of *tags*, allowing the creation of customized versions of the applications. If an image is created that does not contain the image version, a *tag latest* is created. This *tag* represents the last build performed without a specified version.

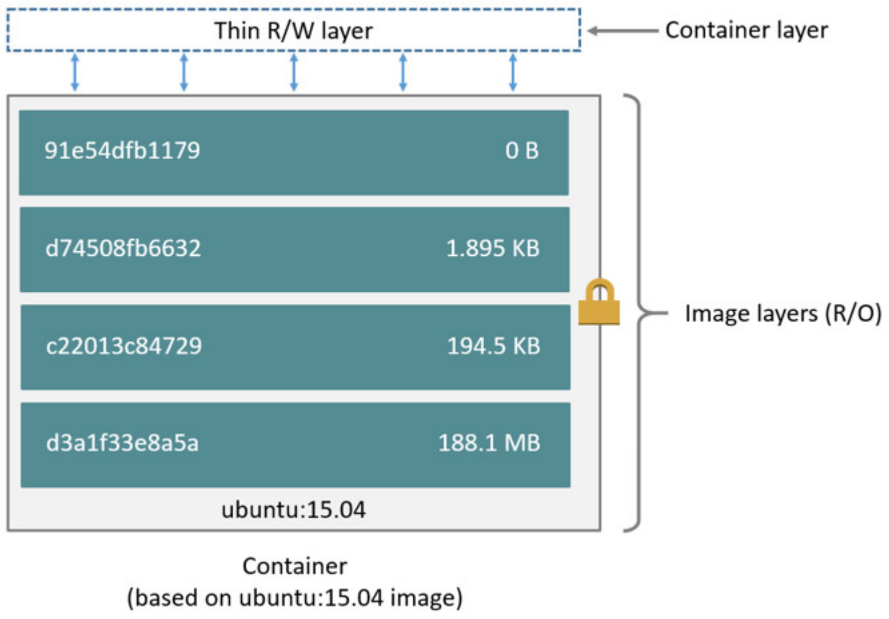


Figure 2.3: Container based on Ubuntu 15.04 image [14].

The Docker architecture, as illustrated in Figure 2.4, is composed of three main components: Docker Client (Command Line Interface - CLI), Docker Daemon, and Docker Registry. Container provisioning is performed via the Docker daemon, while the distribution of images occurs via a public or private Docker registry, such as the Docker Hub[2]. The container provisioning process by the Docker daemon consists of downloading the image from a registry if it is not present locally. If there is already any layer of the desired image locally, the Docker daemon downloads only the missing layers. Finally, it starts a new container based on the image.

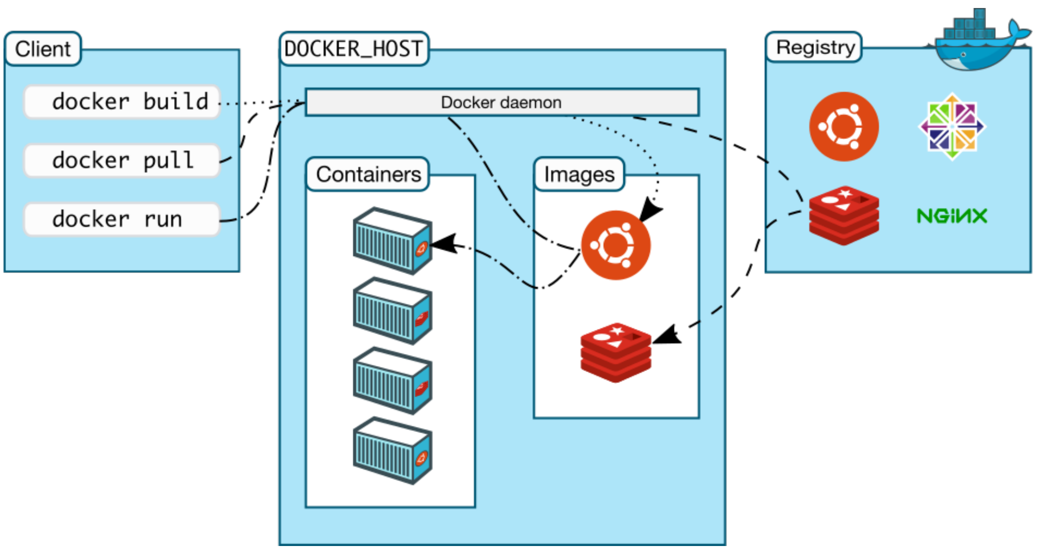


Figure 2.4: Docker architecture [14].

2.3 Provisioning

Several platforms have been developed to manage and provision containers, also known as orchestrators. Among them, some possibilities are *Docker Swarm* [16] and Kubernetes [24]. In this work, we will address only Kubernetes.

2.3.1 Kubernetes

Kubernetes is an open-source container orchestrator proposed by Google. Its first version was released in 2015. This platform is based on a master-slave architecture model. As illustrated in Figure 2.5, the master node is composed of a scheduler (Kube-scheduler) that is responsible for provisioning strategies, in addition to API services (API Server), con-

[2]https://hub.docker.com

trollers (Controller Manager) and an Etcd key-value database[3], while the primary responsibility of the slave nodes is the execution of the containers [24].

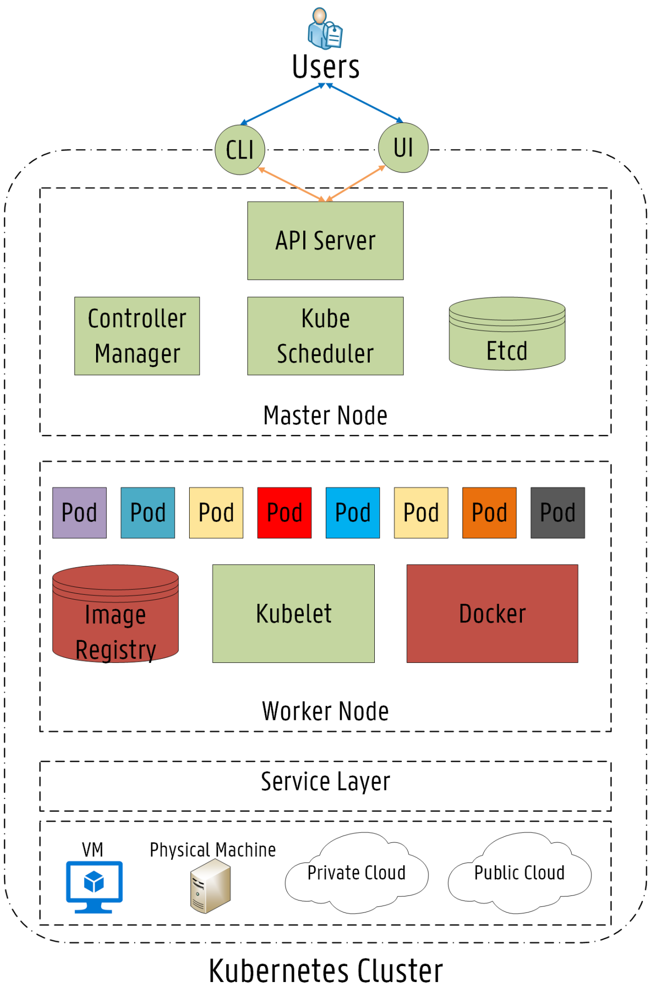


Figure 2.5: High-level view of the Kubernetes architecture [34].

The smallest compute unit that can be provisioned in Kubernetes is called a pod. A pod consists of a grouping of one or more containers with shared resources such as network and storage, and definitions of how to run each container. Kubernetes's default pods provisioning mechanism, kube-scheduler, is responsible for selecting the nodes that will run the pods through two main processes: filtering and ranking.

In the first process (filtering), it selects the eligible nodes based on predicate-based policies. Some examples include:

1. **PodFitsResources**: This policy classifies the node as eligible if the node has enough computational resources to allocate the pod requirements;

2. **MatchNodeSelector**: This policy classifies the nodes as eligible based on labels across pod's node selector and node's labels;

[3]https://etcd.io/

3. **NoDiskConflict**: This policy classifies the node as eligible if the node can host the pod based on the volumes that it request.

After the filtering process, the scheduler ranks the eligible nodes selected previously based on priorities-based policies. Some possibilities are:

1. **SelectorSpreadPriority**: Spreads pods among the nodes, considering pods that belong to the same service;
2. **BalancedResourceAllocation**: Favor nodes with balanced resource usage;
3. **ImageLocalityPriority**: Favors nodes that already have container images belonging to the pod to be provisioned locally cached;
4. **EqualPriority**: Sets a weight equal to one for all nodes;
5. **LeastRequestedPriority**: Prioritize nodes based on amount of free computing resource;
6. **MostRequestedPriority**: Prioritize the nodes with the most requested resources.

In addition to enabling the operator to customize the provisioning process by choosing policies, the Kube-scheduler also allows the cluster operator to customize the provisioning process by configuring priority policies with different weights, thus making it possible to increase or decrease the relevance of a given policy[4].

After the scheduling decision, the selected worker node starts downloading the application Docker image. If any of the layers already exist locally in the node cache, they are not downloaded again. By default, the worker downloads just three layers at a time, the others, including other images from other applications, stay in a queue waiting to be downloaded (download queue).

2.4 Final Remarks

Cloud computing has the ability to provide great computing power, being able to meet the demand of complex and heavy applications. However, the distance between data centers and their end-users and devices implies high latency due to long distances [40, 41]. As a result, the user experience of latency-sensitive applications ends up being degraded. Furthermore, large amounts of data produced by users and end-devices can be processed at the edge of the network, avoiding overloading the network infrastructure if sent to the cloud [40, 36].

[4]https://kubernetes.io/docs/reference/scheduling/config/

There are several challenges in the process of provisioning containers in an edge computing platform, such as minimizing application provisioning time due to distributed architecture, edge servers limitations, and variations in terms of bandwidth and computational capacity. In addition, the main advantage of edge computing over cloud computing is low latency, so the latency between applications provisioned at the edge and their end-users must also be taken into account. In this way, it is essential to choose which is the most suitable edge server to host an application to minimize these problems [40].

In the cloud computing model, there are concerns regarding the reliability of the services provided to its users, such as the availability of computing resources. For this reason, agreements between cloud providers and their users can be made through a Service Level Agreement (SLA) [29]. For example, an SLA can guarantee a user that access latency to a given service provided by the cloud provider always happens within a range. So in this work, we understand that SLAs can be applicable in the edge computing paradigm, providing guarantees to users.

3. STATE OF THE ART

Some works address the application scheduling problem acting directly on the Kubernetes orchestrator. These works extend modules of the Kube-Scheduler to provide new strategies regarding the provisioning of applications to reduce the provisioning time [17, 12, 23], and network latency [34, 35]. In addition, we also found in the literature, works that focus on reducing application provisioning time, from focusing on the placement of Docker layers to provide a cache for the edge servers [42], improve the registry placement in the infrastructure topology [11, 10], and even creating better registries for edge computing environments [3]. We divided these works into two sections: 1) Application Scheduling, and 2) Application Image Distribution.

3.1 Application Scheduling

Fu et al. [17] advocate that the Kube-scheduler should take into account the application's dependencies when provisioning it since the container provisioning time is related to the download of the images. Thus, the authors propose two priorities-based policies to prioritize the nodes that already have some cache locally: 1) image-level cache and 2) layer-level cache. The results obtained during the evaluation of the strategies showed improvements in the provisioning time of up to 2.3 times in some scenarios and a reduction in the CPU and storage utilization of the nodes.

Santos et al. [34] propose an extension to the Kube-Scheduler, called Network-Aware Scheduler (NAS), allowing the provisioning of applications to be carried out considering the location of edge nodes, round trip time (RTT), and its bandwidth limitations to select the most suitable edge server to host latency-sensitive applications based on its target location. The results obtained with the proposed strategy reached an 80% reduction in terms of network latency, respecting the bandwidth limitations of the network infrastructure. However, the authors did not consider the need to download the application images by the node that will perform the provisioning, which can end up causing service interruptions due to high bandwidth usage.

Santos et al. [35] propose a Service Function Chain (SFC) controller to enhance the placement of service chain in edge computing environments since Kube-Scheduler does not consider bandwidth and latency limitations. The proposed solution is based on two scheduling strategies: latency and location-aware, extending the Kubernetes scheduling mechanism. For the former, the best candidate node is selected based on Dijkstra's shortest path algorithm concerning other applications in the service chain. The latter selects the best candidate node based on minimizing end-to-end latency concerning the application target location. Both policies take into account the available bandwidth capacity to choose the most suitable node. The experiments performed showed an 18% reduction in terms of network latency while the nodes' bandwidth was conserved compared to the default behavior of Kube-Scheduler.

Kaur et al. [20] argue that Kubernetes do not efficiently deal with interference between applications (i.e., multiple applications or workloads vying for shared resources on the node where they are allocated), and challenges regarding energy minimization in edge computing scenarios. To address this problem, the authors formulate the problem of task scheduling using Integer Linear Programming (ILP) based on multiobjective optimization problem. They also propose a scheduler strategy called *Kubernetes-based energy and interference driven scheduler* (KEIDS) for container management on edge-cloud nodes, taking into consideration the emission of carbon footprints, interference, and energy consumption. Simulation-based results show improvements comparing to three baseline algorithms.

Knob et al. [12] proposed a priority-based policy to the Kube-Scheduler in order to let the scheduling process be aware of the server node bandwidth, node image download queue (i.e., applications allocated to the server waiting to be downloaded), and an estimate of the application provisioning time, called Infrastructure Aware (IA). This strategy was designed to compose the Kube-Scheduler priority-based policies, that is, it is used by the scheduler to prioritize eligible nodes based on a score. To calculate the nodes score, this strategy estimates the provisioning time of a given application based on the node cache, download queue and the node bandwidth rate. Results based on simulations present an average of 52% reduction compared to the Kube-Scheduler.

Knob et al. [23] proposed a novel container scheduler called *Deployment Latency SLA Enforcement Scheduler* (DLSLA) using the Non-dominated Sorting Genetic Algorithm II (NSGA-II) to optimize the container allocation and reorder the node download queue in order to deploy applications within a given time without violating the application provision time SLA. Considering that different applications may have different provisioning time SLAs, this strategy focuses in ordering the download queue of nodes, prioritizing applications with a tight SLA value (i.e. changing the positions of applications in the download queue). Results based on simulations point out that the proposed solution is almost 200% better in ensuring the application SLA than the Kube-Scheduler since it does not consider the network infrastructure during the scheduling.

In [32], Rossi et al. argue that Kubernetes is not suitable for orchestrating containers in a geographically distributed cloud computing environment. Therefore, the authors propose a container scheduling strategy considering the network infrastructure, latency between application replicas and application provisioning time called *Geo-distributed and Elastic deployment of containers in Kubernetes* (ge-kube). In addition, ge-kube also contains a mechanism that scales the applications horizontally and/or vertically. Experiments performed showed that the combination of scheduling and scalability strategies showed benefits.

3.2 Application Image Distribution

Knob et al. [11] argues that well-positioned container registries on an edge computing network topology can significantly improve the deployment process, mainly reducing the container-based applications deployment latency (i.e., start-up time). They proposed a registries placement solution based on a fluid communities algorithm to find the best registry positioning in the infrastructure topology.

In order to optimize container provisioning time, Darrous et al. [10] propose two algorithms for positioning container images on edge servers, namely k-Center-Based Placement (KCBP) and KCBP-Without-Conflict (KCBP-WC). The strategy aims to reduce the maximum time of download images from any node at the edge, by allocating a set of layers in a set of nodes. The KCBP-WC algorithm has the same principle but avoids performing simultaneous downloads from the same node. The results obtained through simulations showed that the algorithms reduced the maximum recovery time of container images by 1.1x to 4x compared to traditional provisioning methods such as: Best-fit and Random.

Smet et al. [42] argues that there is an increase in services with unknown demand patterns at the edge of the network, and pre-deploying such services is not feasible. In order to tackle this problem, the authors proposed a Docker layer placement method to make it possible to provision the services within the desired response time.

Also, intending to speed up the image pulling from edge servers, Becker et al. [3] proposed a peer-to-peer (P2P) based container image registry to reduce the start-up of container-based applications (image pulling time) by edge servers rather than retrieve container images from centralized registries, following the subsequent requirements: non-intrusive deployment, reliable and fully decentralized, accelerated image distribution.

3.3 Final Remarks

Several works propose improvements in the deployment of applications based on containers at the edge of the network, focusing on provisioning strategies and even improvements in image repositories (i.e., registry). Table 3.1 presents the works related to the scope of this work. We can observe that some focus on reducing application provisioning time or avoiding provisioning time SLA violations while others the latency between the applications and end-users. However, to the best of our knowledge, there are no works considering both aspects, which make up the scope of this work.

Table 3.1: Related work.

Paper	**End-user Latency**	**Provisioning Time**	**Primary Goal**	**Architecture**
[32]	✗	✓	Reduce latency between application replicas	Cloud Computing
[20]	✗	✗	Reduce application interference and energy consumption	Edge Computing
[17]	✗	✓	Reduce application provisioning time	Edge Computing
[34]	✗	✗	Reduce application latency by location-awareness	Edge Computing
[35]	✗	✗	Optimize SFC scheduling considering latency	Edge Computing
[12]	✗	✓	Reduce application provisioning time	Edge Computing
[23]	✗	✓	Enforce application provisioning time SLA compliance	Edge Computing
This work	✓	✓	**Minimize latency and provisioning time SLA violations**	**Edge Computing**

The next chapter presents the proposed algorithm to tackle the container-based application scheduling problem in heterogeneous infrastructures as well as the system model considered in our modeling.

4. HEURISTIC FOR CONTAINER-BASED APPLICATIONS SCHEDULING

This chapter presents the proposed heuristic called *Latency and Provisioning Time SLA Driven Scheduler* (LPSLA) focused on container-based application scheduling at the edge computing infrastructures. Unlike most works found in the literature, our proposed solution focuses on minimizing the application SLA violations in terms of latency in relation to its end-users and provisioning time. This chapter initially presents our system model formulation, followed by the proposed solution.

4.1 System Model

This section details the edge computing infrastructure and container-based application problem addressed in this work. The notations used to describe our modeling are in Table 4.1.

Table 4.1: Summary of notations used in this work.

Symbol	**Notation**
$\mathcal{N}$	Set of edge nodes
$\mathcal{L}$	Set of network links
$\mathcal{A}$	Set of applications
β	Application image registry node
c_i	CPU capacity of edge node $\mathcal{N}_i$
m_i	Memory capacity of edge node $\mathcal{N}_i$
z_i	CPU demand of edge node $\mathcal{N}_i$
p_i	Memory demand of edge node $\mathcal{N}_i$
μ_i	Cache of edge node $\mathcal{N}_i$
η_i	Download queue of edge node $\mathcal{N}_i$
l_u	Latency of network link $\mathcal{L}_u$
b_u	Bandwidth of network link $\mathcal{L}_u$
ξ_j	CPU demand of application $\mathcal{A}_j$
ρ_j	Memory demand of application $\mathcal{A}_j$
λ_j	Set of layers that compose application $\mathcal{A}_j$
ω_j	Position of the user of application $\mathcal{A}_j$
ν_j	Provisioning Time SLA of application $\mathcal{A}_j$
τ_j	Latency SLA of application $\mathcal{A}_j$
freeResources($\mathcal{N}_i$)	Function that returns the amount of resources available of edge node $\mathcal{N}_i$
demand($\mathcal{A}_j$)	Function that returns the demand of an application $\mathcal{A}_j$

We consider an edge computing network infrastructure modeled as an undirected graph $\mathcal{G} = (\mathcal{N}, \mathcal{L})$, where $\mathcal{N}$ is the set of heterogeneous edge nodes and $\mathcal{L}$ is the set of network links connecting the nodes. We model an edge node $\mathcal{N}_i = \{c_i, m_i, z_i, p_i, \mu_i, \eta_i\}$, where c_i and m_i represents the edge node CPU and memory capacity, and z_i and p_i the edge node CPU and memory demand (i.e., resource usage). The set of application layers that the edge node $\mathcal{N}_i$ has in its local cache is denoted by μ_i, and η_i represents the order and set of applications layers waiting to be downloaded by the edge node $\mathcal{N}_i$, called download queue since many applications requests can be made to an edge node.

In our formulation, each network link $\mathcal{L}_u = \{l_u, b_u\}$ are bidirectional and contains two attributes. The first one, l_u, represents the link latency, while the latter, b_u, represents the link bandwidth capacity. The set of container-based applications to be provisioned in the infrastructure $\mathcal{G}$ is represented by $\mathcal{A} = \{\mathcal{A}_1, \mathcal{A}_2, ..., \mathcal{A}_{|\mathcal{A}|}\}$. Each application $\mathcal{A}_j = \{\xi_j, \rho_j, \lambda_j, \omega, \nu_j, \tau_j\}$ contains a CPU and memory demand represented by ξ_j and ρ_j, respectively. Furthermore, each application is considered as a single container, and they are independent (i.e., monolithic application). The set of layers that compose the application $\mathcal{A}_j$ is denoted by $\lambda_j = \{\lambda_{j,1}, \lambda_{j,2}, ..., \lambda_{j,|\lambda_j|}\}$, and the node from which the user will access the application by ω. Lastly, since each application could have a custom SLA, the ν_j represents the application provisioning time SLA, and τ_j the application latency SLA. We can check if the node $\mathcal{N}_i$ has a layer $\lambda_{j,k}$ in its cache through the matrix δ.

$$\delta_{i,j,k} = \begin{cases} 1 & \text{if layer} \lambda_{j,k} \in \mu_i \\ 0 & \text{otherwise.} \end{cases}$$

$$\aleph_{i,j,k} = \begin{cases} 1 & \text{if layer} \lambda_{j,k} \in \eta_i \\ 0 & \text{otherwise.} \end{cases}$$

When provisioning an application $\mathcal{A}_j$ to an edge node $\mathcal{N}_i$, the set of layers λ_j are transferred from the registry node β (i.e., image repository) to the edge node $\mathcal{N}_i$. If $\mathcal{N}_i$ has already a layer $\lambda_{j,k}$ locally in its cache (i.e., $\delta_{i,j,k} = 1$), then layer $\lambda_{j,k}$ is not downloaded from the registry. Otherwise, we transfer $\lambda_{j,k}$'s data, represented by $s_{j,k}$, from the registry. However, if a layer $\lambda_{j,k}$ is already in the edge node's download queue (i.e., $\aleph_{i,j,k} = 1$), we disconsider the download of $\lambda_{j,k}$. In this way, the application provisioning time of application $\mathcal{A}_j$ in edge node $\mathcal{N}_i$ using images stored in registry β can be described by ζ according to Equation 4.1, where w_i represents the total time needed to empty the download queue η_i (i.e. provision all applications).

$$\zeta(i,j) = w_i + \sum_{k \in \lambda_j}^{|\lambda_{j,k}|} \frac{s_{j,k}}{\gamma(\beta, \mathcal{N}_i)} \cdot (1 - sgn(\delta_{i,j,k} + \aleph_{i,j,k})) \tag{4.1}$$

In short, we disregard a layer $\lambda_{j,k}$ if it is already in cache ($\delta_{i,j,k} = 1$) or in the download queue ($\aleph_{i,j,k} = 1$), through the signum function (*sgn*) which results in zero only if the sum of $\delta_{i,j,k}$ and $\aleph_{i,j,k}$ are equal to zero. We denote the bandwidth available for provisioning an application $\mathcal{A}_j$ to an edge node $\mathcal{N}_i$ as $\gamma(\beta, \mathcal{N}_i)$. If no transfer between the container registry β and $\mathcal{N}_i$ has been done yet, $\gamma(\beta, \mathcal{N}_i)$ gets the minimum bandwidth between the links connecting $\mathcal{N}_i$ and β. Otherwise, we assume $\gamma(\beta, \mathcal{N}_i)$ as the last calculated bandwidth between β and $\mathcal{N}_i$.

The latency of an application $\mathcal{A}_j$ depends on which edge node $\mathcal{N}_i$ it will be provisioned in relation to the user's position ω_j as in Equation 4.2, which considers the sum of all network links latency l_u between an edge node $\mathcal{N}_i$ and the edge node where the user of the application is located ω_j, represented by $\theta(i, j)$, as we consider users to access applications through an edge node, such as a gateway. The set of links $\theta(i, j)$ is defined through the Dijkstra shortest path algorithm [13], using as weight the link latencies l_u.

$$\Omega(i, j) = \sum_{u \,\in\, \theta(i,j)} l_u \tag{4.2}$$

4.2 Proposed Heuristic

The main goal of the proposed heuristic is to reduce provisioning time and latency SLA violations of container-based applications by estimating application provisioning time considering the image layers present in the server's cache and the queue of applications waiting to be provisioned (i.e., edge node download queue). In the context of this work, each application is scheduled individually (i.e., we run the heuristic once for each application) according to the need for provisioning and in the order they are requested (i.e., the scheduler is not aware of the next applications that will be requested in the future).

Algorithm 4.1 presents the proposed heuristic. Initially, we create a set S of eligible edge nodes if $\mathcal{N}_i \in \mathcal{N}$ has enough computing resources to provision an application $\mathcal{A}_j$ (line 4), following the same logic as Kube-Scheduler's PodFitsResources [24]. Then, for each edge node, we estimate provisioning time $\zeta(i, j)$ of application $\mathcal{A}_j$ to edge server $\mathcal{S}_i$ (line 6), as described in Equation 4.1, and we compute the network latency $\Omega(i, j)$ between $\mathcal{S}_i$ and the application user edge node location ω_j (line 7), as described in Equation 4.2. Next, we sort the set of edge nodes $\mathcal{S}$ ascending based on the sum of the number of SLA violations $\psi(S_i, \mathcal{A}_j)$, according to Equation 4.3 (line 9).

$$\psi(S_i, \mathcal{A}_j) = [L_i > \tau_j] + [P_i > \nu_j] \tag{4.3}$$

In addition, in order to select the most suitable edge node to deploy a given application, for each edge node S_i we calculate the edge node score according to a cost function Δ presented in Equation 4.4, which considers the application provisioning time P_i and edge node latency L_i in relation to a particular application $\mathcal{A}_j$. As variables P_i and L_i have values on different scales, we normalize them using Min-Max Normalization [18] (line 9). Finally, we sort the set of edge nodes $\mathcal{S}$ ascending based on edge node score normalized and return the first edge node S_1 (line 10).

Algorithm 4.1 LPSLA algorithm.

```
1: function LPSLA($\mathcal{N}$, $\mathcal{A}_j$)
2:     $P \leftarrow \{\}$
3:     $L \leftarrow \{\}$
4:     $S \leftarrow \{\mathcal{N}_i \mid \mathcal{N}_i \in \mathcal{N} \wedge freeResources(\mathcal{N}_i) \geq demand(\mathcal{A}_j)\}$
5:     for each edge node $S_i \in S$ do
6:         $P_i \leftarrow \zeta(i,j)$
7:         $L_i \leftarrow \Omega(i,j)$
8:     end for
9:     $S \leftarrow$ Edge nodes $\in S$ sorted by Eq. 4.3 (asc.) and Eq. 4.4 (asc.)
10:    return $S_1$
11: end function
```

$$\Delta(P_i, L_i) = norm(P_i) + norm(L_i) \tag{4.4}$$

We understand that applications may have different SLA values. For example, a highly latency-sensitive application (e.g., augmented reality) needs lower latency than other applications (e.g., batch-based surveillance). In this way, it is possible to observe that Algorithm 4.1 will choose the edge node S_i with the lowest latency and provisioning time values to deploy a given application $\mathcal{A}_j$. However, depending on an application's SLA values (ν_j and τ_j), choosing the best edge node to provision it may not be necessary. In addition, by choosing the best edge node without the need (that is, preferring the best edge node without the need considering the SLA values of the application), it can jeopardize the SLA compliance of other applications with a tight SLA.

To understand the impact of the sorting order of edge nodes, we propose a variation of the Algorithm 4.1 called LPSLA-DS. As shown in the Algorithm 4.2, we just changed the sorting function (line 9) to prioritize edge nodes that do not violate any SLAs (Eq. 4.3), but have the highest score values (Eq. 4.4).

Algorithm 4.2 LPSLA Decrease Sorting (LPSLA-DS) algorithm.

```
1: function LPSLADecreaseSorting($\mathcal{N}$, $\mathcal{A}_j$)
2:     $P \leftarrow \{\}$
3:     $L \leftarrow \{\}$
4:     $S \leftarrow \{\mathcal{N}_i \mid \mathcal{N}_i \in \mathcal{N} \wedge freeResources(\mathcal{N}_i) \geq demand(\mathcal{A}_j)\}$
5:     for each edge node $S_i \in S$ do
6:         $P_i \leftarrow \zeta(i,j)$
7:         $L_i \leftarrow \Omega(i,j)$
8:     end for
9:     $S \leftarrow$ Edge nodes $\in S$ sorted by Eq. 4.3 (asc.) and Eq. 4.4 (desc.)
10:    return $S_1$
11: end function
```

In conclusion, the container-based application scheduling in highly distributed and heterogeneous environments needs to consider several factors so that the quality of service and user experience are not compromised. In this chapter, we present our system model and two scheduling algorithms: LPSLA and LPSLA-DS. In the next chapter, the evaluation methodology used to evaluate the proposed solution is presented.

5. EVALUATION

This chapter presents the process and methodology used to evaluate the proposed heuristic on container-based applications scheduling on an edge computing infrastructure. The chapter initially presents the edge computing infrastructure considered, edge nodes, applications, and SLA values. It is also presented the simulator used to evaluate the strategies. The remainder of the chapter presents and discusses the achieved results.

5.1 Experiments Description

We created the network topology used to evaluate the proposed heuristic based on the Ipê Brazilian Research Network [31] topology. This topology contains 28 Points-of-Presence (PoP) distributed by the country. In each PoP was created five edge nodes of different capacities, according to Table 5.1, chosen based on a normal distribution, as it presents a heterogeneous distribution, which is expected in edge infrastructure environments.

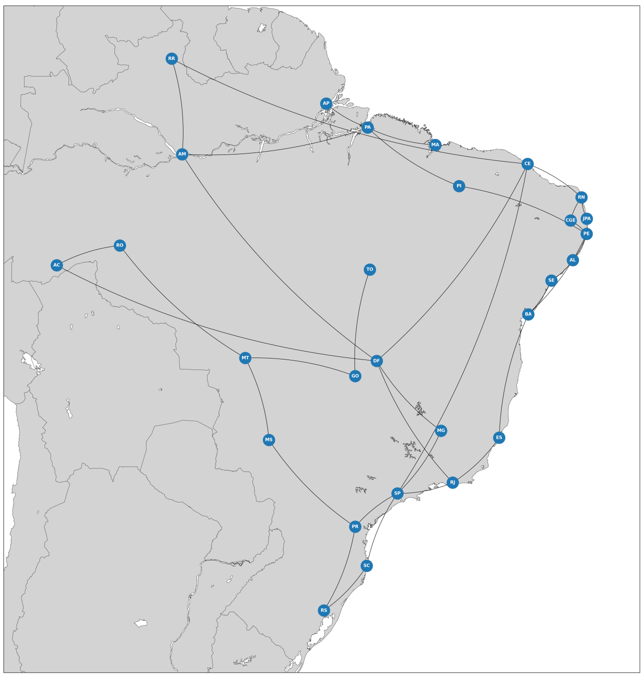


Figure 5.1: Ipê topology.

The edge nodes connection links bandwidth and latency have been configured following a normal distribution, with values between 10Mb/s and 200Mb/s with an interval of 10Mb/s (i.e., *10Mb/s, 20Mb/s, 30Mb/s, ..., 200Mb/s*) and 10ms, 12ms, 14ms, 16ms of latency, respectively.

Table 5.1: Edge node configurations.

Type	CPU (# of Cores)	Memory (GB)	Disk (GB)
Small	2	2GB	32GB
Medium	4	4GB	64GB
Large	8	8GB	128GB

The fifty most popular Docker Hub [15] images and its layers were used to create a set of applications that are deployed in the topology during the simulations to evaluate the heuristics. Each application has been configured with one of the groups of requirements presented in Table 5.2, following a normal distribution.

Table 5.2: Application requirements.

Millicores	Memory
250	256MB
250	512MB
500	512MB
1000	1024MB

The sum of the images used has a total size of 8705.82 MB (an average of 174.11 MB per image). However, since several images share equal layers, a node would need to download 6037.13 MB to have all applications (30% of similarity between images). Each application is set up to be provisioned on the infrastructure at a specific time during the simulation, called the scheduling time. In this way, we configured the scheduling time of each application following a normal distribution with values between 0 and 1000 seconds.

In our simulations, we put the docker registry (i.e., docker images remote repository) in the PoP of São Paulo due to the large amount of bandwidth in the region to prevent the registry from being a bottleneck in the simulation and because it is the primary connection to cloud providers [30]. Figure 5.2 presents an illustration of the final topology used, where the nodes in blue are network switches, the gray nodes are server nodes (i.e., worker nodes), and the red node is the application image registry.

To simulate users requesting to access the applications, we configured in each application a node that would refer to the abstraction of the user's location; in this way, we can calculate the latency between users and applications. In the context of this work, the application needs to be provisioned close to the user's position to avoid a huge network latency and, consequently, SLA violations.

As different applications may have different priorities and requirements, for example, health-oriented applications have higher priority over safety-oriented applications, two types of SLA were considered, one with a loose value (i.e., a value easily reached) and

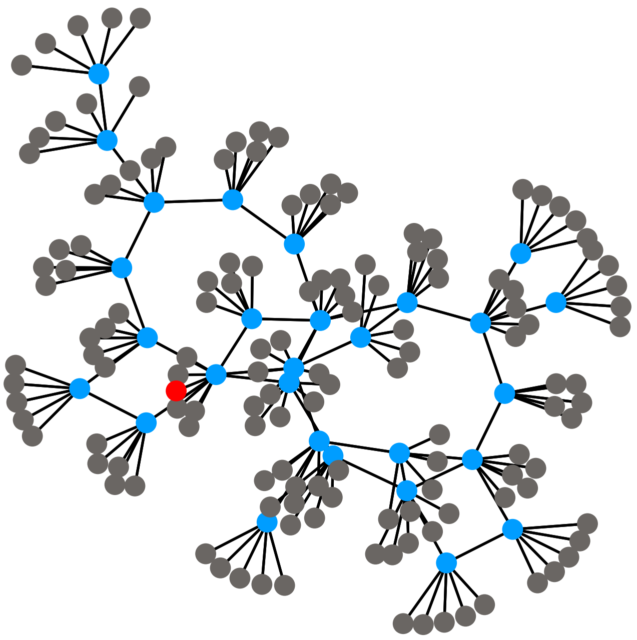


Figure 5.2: Final topology.

the other a tight value. In order to define the application SLA values, we compute the infrastructure topology capacity in terms of network latency and application provisioning time. To understand the impact of the proportion of SLA values, three scenarios were evaluated according to Table 5.3. In each scenario, the proportion of applications with loose and tight SLAs values are varied (e.g., the first scenario consists of 50% of applications with the loose and 50% with the tight SLA values).

Table 5.3: SLA scenarios.

SLA Values	**Scenario #1**	**Scenario #2**	**Scenario #3**
Loose	50%	80%	20%
Tight	50%	20%	80%

In the context of this work, we are comparing our proposed solutions, LPSLA and LPSLA-DS, with three heuristics presented next.

- **Kube-Scheduler (KS) [24]**: This is the default scheduling mechanism of Kubernetes. It selects the most appropriate nodes based on priority-based and predicated-based policies;

- **Infrastructure-Aware (IA) [12]**: This strategy extends the Kube-scheduler with a priority-based policy that prioritizes nodes based on the layers of the images it has in cache and the estimation of application provisioning time (considering the download queue);

- **Deployment Latency SLA Enforcement Scheduler (DLSLA) [23]**: It uses a NSGA-II algorithm to optimize the container allocation and reorder the node download queue in order to avoid provisioning time SLA violations;

5.2 Simulator

All experiments were executed in the *Edge Computing Orchestration Simulator* (ECOS) presented in [22]. ECOS is written in Python and uses the Networkx library [28]. The simulator allows us to evaluate different scheduling strategies (e.g., Kube-Scheduler, First-Fit, Best-Fit) in a customizable edge computing environment. Besides simulating the container-based applications scheduling process, the simulator also simulates the application image distribution from a registry to the edge nodes considering the network topology (i.e., connection links and connection speed).

The simulator network implementation uses a fair sharing schedule policy based on the max-min fairness algorithm proposed by Bertsekas et al. [5], which is based on the equal sharing of network connection links between active sessions (i.e., network packet). This approach enables a more realistic simulation due to the acceptable degree of approximation with real protocols (i.e., TCP) and simulating network bottlenecks when provisioning containers in edge nodes.

Some of the main features of max-min fairness are: 1) network packets have equal priority over the available network links bandwidth; 2) the network link bandwidth capacity is split evenly among the network packets, and 3) network packets are always using the maximum bandwidth possible based on the active network links [23].

About the Kube-Scheduler, we simplified the implementation based on the official documentation; that is, policies more specific to the operation of Kubernetes in real scenarios such as VolumeBinding, VolumeZone, VolumeRestrictions, and among others were not considered. Moreover, our implementation allows us to configure the scheduler in a customized way and change the weight of each priority-based policy.

5.3 Results

In the evaluation, we consider an edge computing infrastructure composed of 140 heterogeneous server nodes and a workload composed of 889 applications, representing 80% of the nodes' capacity. The simulations start with the infrastructure completely empty (i.e., with no applications running) and end after all applications are provisioned. The SLAs values were defined arbitrarily, for application provisioning time is 70s for the loose SLA and

15s for the tight SLA, while the SLA values for application network latency is 80ms for the loose SLA and 20ms for the tight SLA, according to Table 5.4.

Table 5.4: Simulation parameters.

Parameter	**Value**
Server Nodes	*140*
Number of Images	*50*
Number of Applications	*889*
Loose Latency SLA	*80ms*
Tight Latency SLA	*20ms*
Loose Provisioning Time SLA	*70s*
Tight Provisioning Time SLA	*15s*

The results obtained from the simulation of the three SLA scenarios for the analyzed heuristics are depicted in Table 5.5. Initially, we observe that the proposed strategies significantly overcomes the other strategies on all evaluated metrics. The analysis of the results is divided into three sections: *1) Latency Analysis; 2) Provisioning Time Analysis; and 3) Resource Utilization Analysis*.

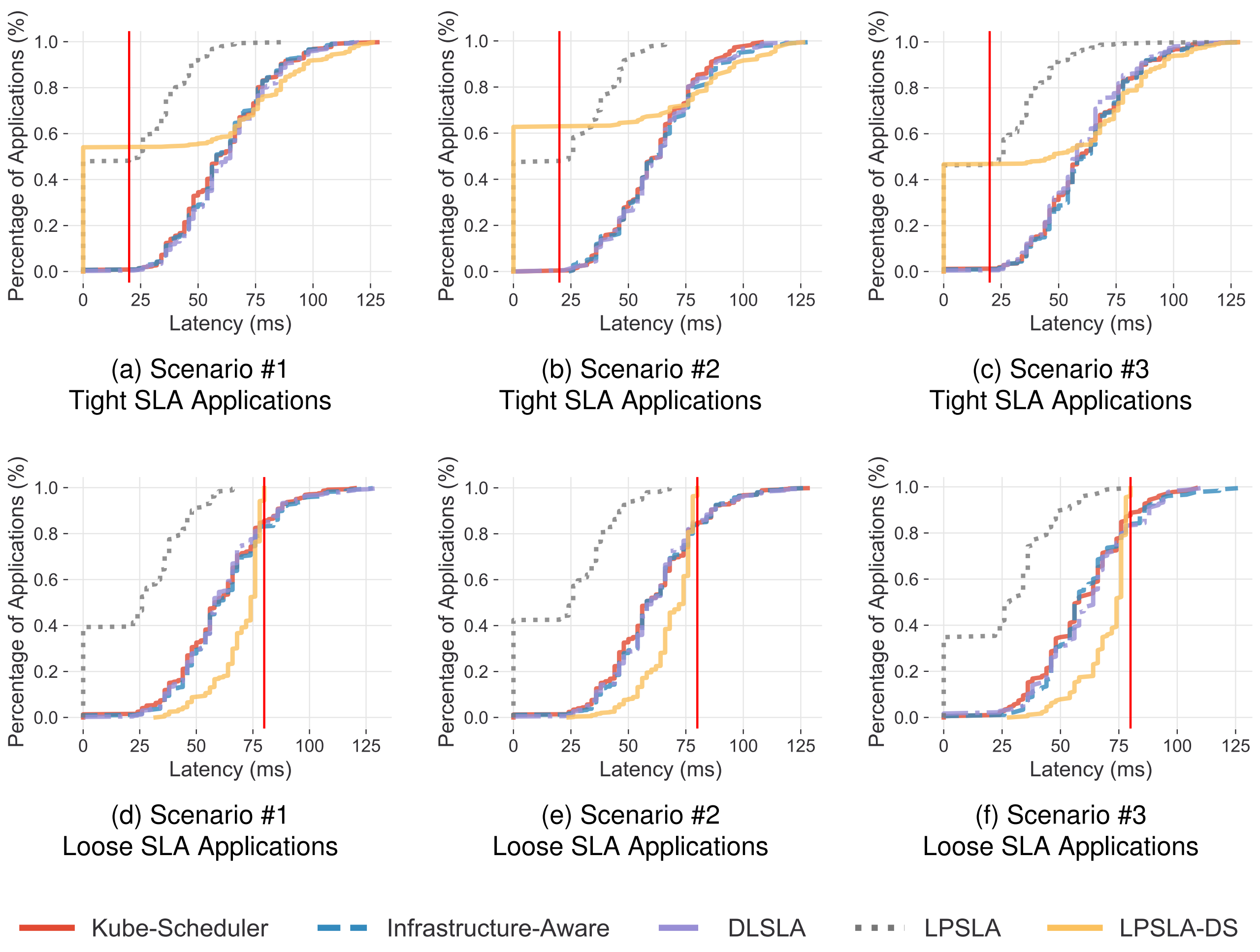
Table 5.5: Experiment results in all scenarios.

Scenario	**Strategy**	**Number of SLA Violations**			
		Latency		**Provisioning Time**	
		Total	**Percentage**	**Total**	**Percentage**
Scenario #1	Kube-Scheduler	503	57%	88	10%
	Infrastructure-Aware	514	58%	84	9%
	DLSLA	508	57%	68	8%
	LPSLA	**228**	**26%**	**47**	**5%**
	LPSLA-DS	**203**	**23%**	**20**	**2%**
Scenario #2	Kube-Scheduler	283	32%	45	5%
	Infrastructure-Aware	285	32%	32	4%
	DLSLA	279	31%	29	3%
	LPSLA	**92**	**10%**	**13**	**1%**
	LPSLA-DS	**65**	**7%**	**14**	**2%**
Scenario #3	Kube-Scheduler	721	81%	139	16%
	Infrastructure-Aware	732	82%	131	15%
	DLSLA	736	83%	98	11%
	LPSLA	**377**	**42%**	**92**	**10%**
	LPSLA-DS	**377**	**42%**	**32**	**4%**

5.3.1 Latency Analysis

The latency of an application comprises the aggregated delay of all links used to communicate the node that hosts and the node on which user is connected in. As shown in

Table 5.5, our proposed solutions managed to decrease the number of latency SLA violations in all evaluated scenarios significantly.



(a) Scenario #1
Tight SLA Applications

(b) Scenario #2
Tight SLA Applications

(c) Scenario #3
Tight SLA Applications

(d) Scenario #1
Loose SLA Applications

(e) Scenario #2
Loose SLA Applications

(f) Scenario #3
Loose SLA Applications

Figure 5.3: Latency CDFs for tight and loose SLA applications in all scenarios.

The Figure 5.3 presents the cumulative distribution function (CDF) of application latencies of all evaluated scenarios. The vertical red lines mark SLA values. In general, high latency SLA violation rates are an expected result for KS, IA, and DLSLA, since they do not consider users' latency when scheduling applications. In this way, all of them have similar behavior. In addition, the number of violations increases in the third scenario because it is the most difficult, as 80% of applications have a tight SLA of 20ms.

In the first scenario, the LPSLA-DS violates 23% of latency SLA while LPSLA 26%. According to the CDF presented in the Figure 5.3a, the LPSLA-DS was able to provision 54% of tight SLA applications with up 20ms of latency, while LPSLA 49% of applications justifying the number of violations because in this scenario, 50% of applications have a latency SLA of 20ms. Regarding loose SLA applications, as shown in Figure 5.3d, both of them provisioned all loose SLA applications with up to 80ms.

In the second scenario, 20% of applications have a tight latency SLA of 20ms, the LPSLA violates 10% of the applications, because according to Figure 5.3e it provisioned

43% of applications with a loose SLA with up to 20ms, that is, possibly ended up using the best nodes to provision applications with loose SLAs instead of tight SLAs, as shown in the Figure 5.3b. On the other hand, LPSLA-DS provisioned 0% of applications with a loose SLA with up to 20ms, keeping the best nodes for applications with tight SLA and violating only 7% of general applications.

In the third scenario, the most difficult one, 80% of applications have a tight latency SLA of 20ms. According to the CDF presented in Figure 5.3f, the LPSLA provisioned 35% loose SLA applications with up to 20ms, possibly wasting nodes suitable for tight SLA applications. Unlike LPSLA-DS, which provisioned 0% of loose SLA applications with up to 20ms, keeping the best nodes for tight SLA applications. However, both of them end up violating 42% of applications, possibly due to overutilization of nodes or due to interference from other objectives, such as provisioning time SLA.

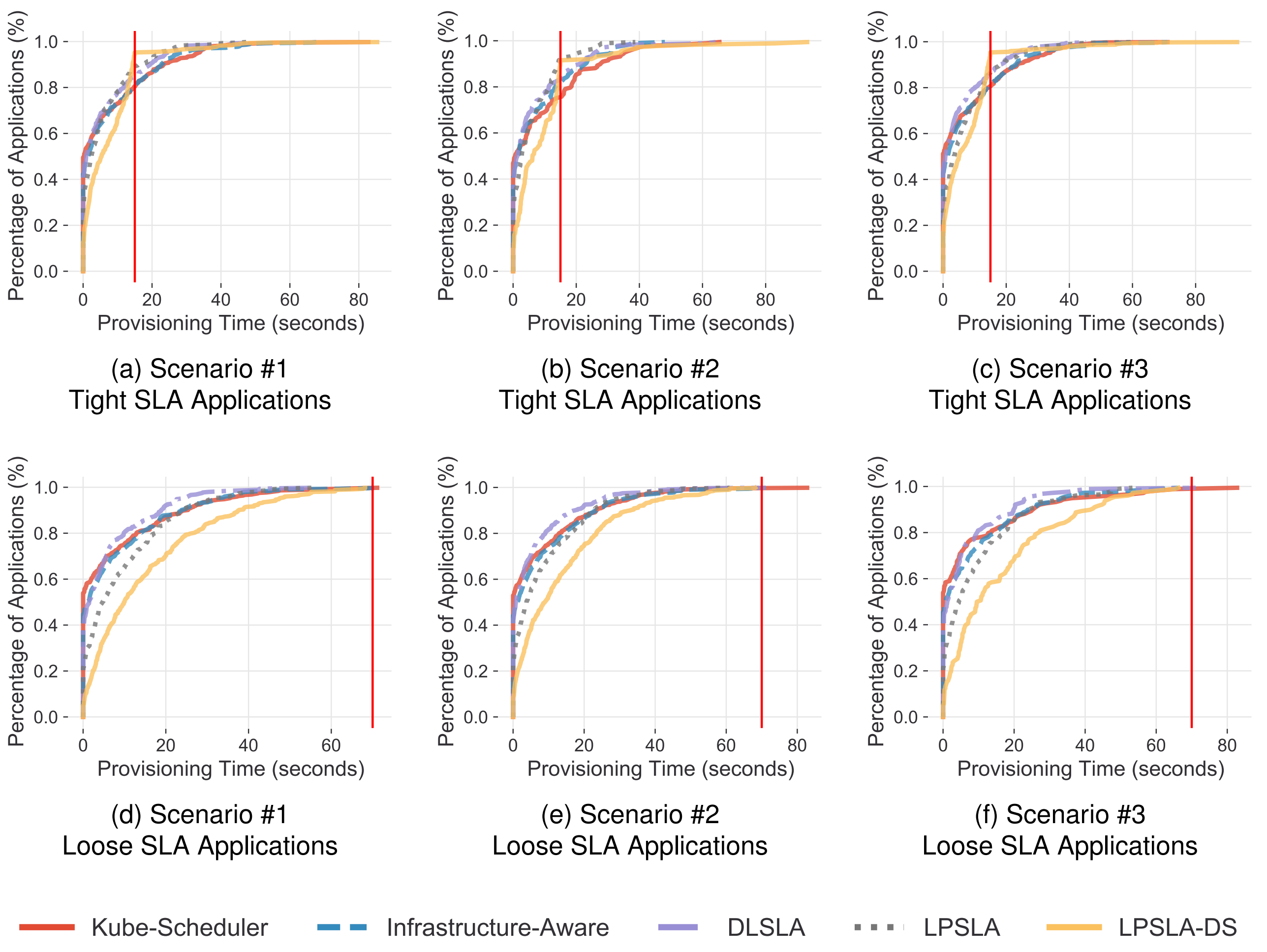
Compared to Kube-Scheduler, the proposed heuristics, LPSLA-DS and LPSLA, reduced the number of SLA violations by 56%, 72%, and 47%, respectively, in the three evaluated scenarios. While LPSLA-DS showed a reduction of 10%, 29%, and 0% compared to the LPSLA version in the three scenarios evaluated. The slight difference in the last scenario is caused by the difficulty presented in the scenario, as 80% of applications have a tight SLA. Furthermore, large amounts of latency SLA violations of the other heuristics are because the KS, IA, DLSLA approaches are network latency agnostic.

5.3.2 Provisioning Time Analysis

The provisioning time values consist of the time taken to transfer a container-based application from a registry to a given node, represented by the Equation 4.1. As shown in Table 5.5, our proposed solution also managed to decrease the number of provisioning time SLA violations in all evaluated scenarios significantly. Figure 5.4 presents the CDF of application provisioning time on all evaluated scenarios.

In the first scenario, as shown in the Figure 5.4a, about 80%, 81%, 85%, 95% and 89% of tight SLA applications were provisioned within 15 seconds for the strategies KS, IA, DLSLA, LPSLA-DS and LPSLA, respectively. With the KS, IA, DLSLA, LPSLA-DS, and LPSLA reaching the maximum time of 83, 68, 48, 85, and 47 seconds, respectively. Considering only loose SLA applications, as shown in Figure 5.4d, all approaches were able to provision applications within 70 seconds.

The provisioning time values presented by the KS, IA, and DLSLA heuristics are caused by the impact of other policies as these may end up not prioritizing the fastest node for provisioning. In addition, the latter focuses on optimizing the node download queue, sorting it to reduce the number of provisioning time SLA violations while avoiding making a high number of changes to it.



(a) Scenario #1
Tight SLA Applications

(b) Scenario #2
Tight SLA Applications

(c) Scenario #3
Tight SLA Applications

(d) Scenario #1
Loose SLA Applications

(e) Scenario #2
Loose SLA Applications

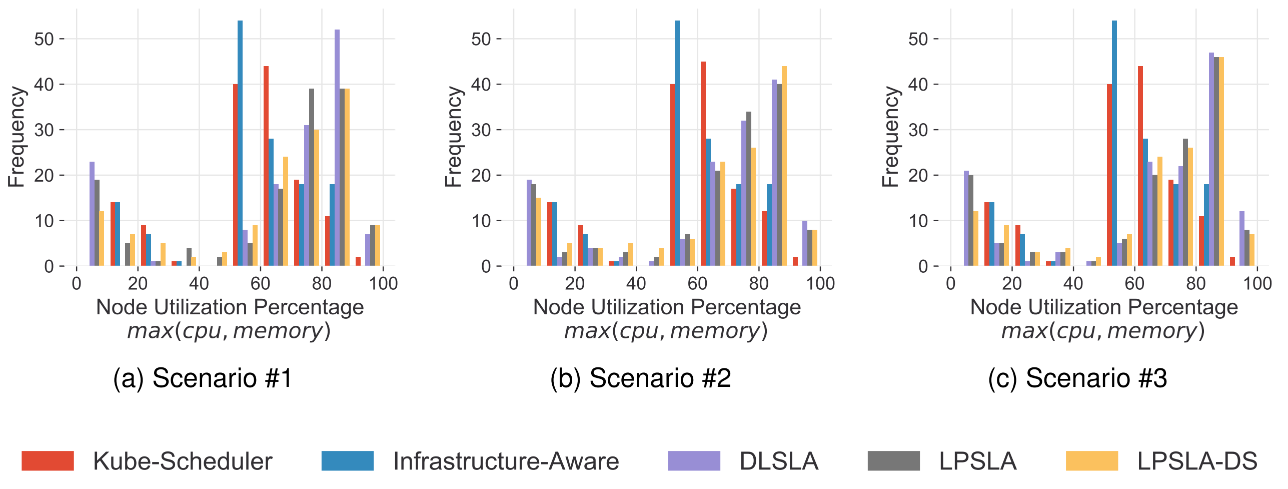
(f) Scenario #3
Loose SLA Applications

Figure 5.4: Application provisioning time CDFs for tight and loose SLA applications in all scenarios.

Despite the highest provisioning time value being from the LPSLA-DS heuristic, according to Table 5.5, this heuristic reduced the total violations by 77%, 68%, and 76% compared to the worst heuristic (KS) in the first, second and third scenario, respectively. It happens because LPSLA-DS prefers the nodes with the highest provisioning time as long as it does not violate any SLA. However, suppose no node can provision the application without violating any SLA. In that case, this heuristic ends up preferring to allocate it to the worst nodes since the SLA would have been violated anyway.

5.3.3 Resource Utilization Analysis

The resource utilization analysis goal is to understand the behavior of the evaluated heuristics in allocating container-based applications. This analysis makes it possible to determine how distributed the applications are in the infrastructure topology. The values presented below are the highest values between CPU and memory of the nodes, as we understand that it is possible to use 100% of a resource (e.g., CPU), making it impossible to provision more applications while the other resource is available (e.g., memory).



(a) Scenario #1 (b) Scenario #2 (c) Scenario #3

Figure 5.5: Edge node resource utilization percentage.

It is possible to observe that KS and IA distributed the applications among the nodes since the frequency of use of the nodes is higher between 50% and 90% in all evaluated scenarios. It happens because these strategies have internal policies that tend to spread applications across nodes. In comparison, DLSLA, LPSLA-DS, and LPSLA tend to centralize the applications on specific nodes due to the high frequency of little-used nodes and the high frequency of heavily used nodes.

The centralization of applications in a set of nodes caused by the heuristics DLSLA, LPSLA, and LPSLA-DS, occurs due to the prioritization of nodes with faster network links and or image layers already in the nodes' cache. It happens because nodes that already have some layers or even entire images locally can provide a particular application faster, justifying the best results obtained concerning the provisioning time.

5.4 Final Remarks

Based on the experiment results, it is possible to verify that the proposed heuristics effectively minimizes the applications provisioning time SLA violations while also avoiding network latency SLA violations in an edge computing scenario. In order to achieve these results, the proposed heuristics were designed to prioritize nodes that can provision applications without causing SLA violations based on an estimated application provisioning time (considering the cache and application queue waiting to be installed on the nodes) and network latency between the user of the application and nodes. Consequently, the proposed heuristics centralize the applications in a set of nodes.

It is worth mentioning that the results of the evaluated heuristics presented in this work, IA and DLSLA, may vary from those presented in its original papers [12, 23] since the evaluation scenario changed and no optimization or sensitivity analysis of the parameters was performed, which could have positive impacts on its results.

6. CONCLUSIONS AND FUTURE WORK

Finding the most suitable edge node to deploy a given application in a totally heterogeneous scenario like edge computing is a variant of the Application Scheduling Problem [43]. In addition to the heterogeneity of edge nodes, several characteristics must be considered during the application scheduling so that the edge computing benefits are not affected. That is, for example, the network latency between the edge and the user for latency-sensitive applications such as augmented reality.

Previous investigations proposed various strategies to improve the application schedule at the network's edge, through improvements to the Kubernetes scheduler, Kubernetes-based schedulers, as well as improvements in the distribution of images at the edge of the network through the placement of registries in the infrastructure or improvements in the Docker image repositories. However, to the best of our knowledge, none of them considered the application provisioning time and the network latency concerning the users of such applications.

This work presents a new container-based application scheduler which focus on enforcing the application latency and provisioning time SLA compliance called LPSLA. This heuristic prioritizes nodes by estimating the application provisioning time considering the nodes' cache, the application queue waiting to be downloaded by the node (download queue) and the node's bandwidth. In addition, the network latency between possible nodes and the application user is also considered. We also proposed a variation of the heuristic called LPSLA-DS, that prioritizes nodes that have the highest values of latency and provisioning time as long as they do not violate application SLAs.

The proposed solutions, LPSLA and LPSLA-DS, were evaluated in a simulated edge computing scenario with different application SLA values. We also evaluated and compared the proposed solutions with Kube-Scheduler [24], Infrastructure Aware [12] and DLSLA [23]. Simulation-based results indicate that the proposed solution overcomes these strategies on latency and provisioning time SLAs across all scenarios.

6.1 Future work

In this work, we consider only one user per application; however, more than one user could need to access a particular application in real scenarios. In this way, as future work, we indent to schedule applications based on the location of these multiple users to make the applications available within limits provided by the SLA, whether provisioning the application close to all its users or even replicating the application in other regions.

We did not consider migrating the services already provisioned, which is also a possibility for future works in order to balance the use of resources, for example. Furthermore, strategies that prioritize one of the objectives over another could also be formulated in future works.

In addition to implementing the proposed strategy in the Kubernetes scheduler in order to evaluate the behavior of the algorithms in real scenarios, as future work, the proposed solution could also be evaluated in other edge network infrastructures with different characteristics, such as the position of one or more registries, different applications, edge node capabilities and connection links.

6.2 Achievements

In addition to the contributions mentioned above, during our research period we participated in other studies in relation to container-based applications provisioning in the context of edge computing and resource management on cloud computing. These publications are listed next:

- **Improving Container Deployment in Edge Computing Using the Infrastructure Aware Scheduling Algorithm**
  Luis Knob, Carlos Kayser, Tiago Ferreto
  IEEE Symposium on Computers and Communications (ISCC), 2021.

- **Enforcing deployment latency SLA in edge infrastructures through multi-objective genetic scheduler**
  Luis Knob, Carlos Kayser, Paulo Souza, Tiago Ferreto
  IEEE/ACM International Conference on Utility and Cloud Computing (UCC), 2021.

- **Predicting and Avoiding SLA Violations of Containerized Applications using Machine Learning and Elasticity**
  Paulo Souza, Miguel Neves, Carlos Kayser, Felipe Rubin, Conrado Boeira, João Moreira, Bernardo Bordin, Tiago Ferreto
  International Conference on Cloud Computing and Services Science (CLOSER), 2022.

REFERENCES

[1] Ahmed, A.; Pierre, G. “Docker container deployment in fog computing infrastructures”. In: IEEE International Conference on Edge Computing (EDGE), 2018, pp. 1–8.

[2] Bahl, V. “Emergence of micro datacenter (cloudlets/edges) for mobile computing”, *Microsoft Devices & Networking Summit*, vol. 5, 2015, pp. 2–1.

[3] Becker, S.; Schmidt, F.; Kao, O. “Edgepier: P2p-based container image distribution in edge computing environments”. In: IEEE International Conference on Performance, Computing and Communications (IPCCC), 2021, pp. 1–8.

[4] Bernstein, D. “Containers and cloud: From lxc to docker to kubernetes”, *IEEE Cloud Computing*, vol. 1–3, 2014, pp. 81–84.

[5] Bertsekas, D. P.; Gallager, R. G.; Humblet, P. “Data networks (2nd Edition)”. Prentice-Hall International New Jersey, 1992, 556p.

[6] Bilal, K.; Khalid, O.; Erbad, A.; Khan, S. U. “Potentials, trends, and prospects in edge technologies: Fog, cloudlet, mobile edge, and micro data centers”, *Computer Networks*, vol. 130, 2018, pp. 94–120.

[7] Bonomi, F.; Milito, R.; Zhu, J.; Addepalli, S. “Fog computing and its role in the internet of things”. In: Workshop on Mobile Cloud Computing (MCC), 2012, pp. 13–16.

[8] Byers, C.; Zahavi, R.; Zao, J. K. “The edge computing advantage”, Technical Report, Industrial Internet Consortium, Needham, MA, USA, 2019, 11p.

[9] Celesti, A.; Mulfari, D.; Fazio, M.; Villari, M.; Puliafito, A. “Exploring container virtualization in iot clouds”. In: IEEE International Conference on Smart Computing (SMARTCOMP), 2016, pp. 1–6.

[10] Darrous, J.; Lambert, T.; Ibrahim, S. “On the importance of container image placement for service provisioning in the edge”. In: International Conference on Computer Communication and Networks (ICCCN), 2019, pp. 1–9.

[11] Dias Knob, L. A.; Faticanti, F.; Ferreto, T.; Siracusa, D. “Community-based placement of registries to speed up application deployment on edge computing”. In: IEEE International Conference on Cloud Engineering (IC2E), 2021, pp. 147–153.

[12] Dias Knob, L. A.; Kayser, C. H.; Ferreto, T. “Improving container deployment in edge computing using the infrastructure aware scheduling algorithm”. In: IEEE Symposium on Computers and Communications (ISCC), 2021, pp. 1–6.

[13] Dijkstra, E. W.; et al.. “A note on two problems in connexion with graphs”, *Numerische mathematik*, vol. 1–1, 1959, pp. 269–271.

[14] Docker. “Docker documentation”. Source: https://docs.docker.com/, December 2021.

[15] Docker. “Docker Hub”. Source: https://hub.docker.com/, December 2021.

[16] Docker. “Docker Swarm”. Source: https://docs.docker.com/engine/swarm/, December 2021.

[17] Fu, S.; Mittal, R.; Zhang, L.; Ratnasamy, S. “Fast and efficient container startup at the edge via dependency scheduling”. In: USENIX Workshop on Hot Topics in Edge Computing (HotEdge), 2020, pp. 1–7.

[18] Han, J.; Kamber, M.; Pei, J. “Data Mining: Concepts and Techniques (3rd Edition)”. Morgan Kaufmann Publishers Inc., 2011, 696p.

[19] Ismail, B. I.; Goortani, E. M.; Ab Karim, M. B.; Tat, W. M.; Setapa, S.; Luke, J. Y.; Hoe, O. H. “Evaluation of docker as edge computing platform”. In: IEEE Conference on Open Systems (ICOS), 2015, pp. 130–135.

[20] Kaur, K.; Garg, S.; Kaddoum, G.; Ahmed, S. H.; Atiquzzaman, M. “Keids: Kubernetes-based energy and interference driven scheduler for industrial iot in edge-cloud ecosystem”, *IEEE Internet of Things Journal*, vol. 7–5, 2019, pp. 4228–4237.

[21] Khan, W. Z.; Ahmed, E.; Hakak, S.; Yaqoob, I.; Ahmed, A. “Edge computing: A survey”, *Future Generation Computer Systems*, vol. 97, 2019, pp. 219–235.

[22] Knob, L. A. D. “Improving container deployment latency in distributed edge infrastructures”, Ph.D. Thesis, Programa de Pós-Graduação em Ciência da Computação - PUCRS, 2021, 122p.

[23] Knob, L. A. D.; Kayser, C. H.; de Souza, P. S. S.; Ferreto, T. “Enforcing deployment latency sla in edge infrastructures through multi-objective genetic scheduler”. In: IEEE/ACM International Conference on Utility and Cloud Computing (UCC), 2021, pp. 1–9.

[24] Kubernetes. “Kubernetes Website”. Source: https://kubernetes.io/, December 2021.

[25] Luo, Q.; Hu, S.; Li, C.; Li, G.; Shi, W. “Resource scheduling in edge computing: A survey”, *IEEE Communications Surveys & Tutorials*, vol. 23–4, 2021, pp. 2131–2165.

[26] Mell, P.; Grance, T.; et al.. “The NIST definition of cloud computing”, *NIST Special Publication*, vol. 800, 2011, pp. 1–3.

[27] Mouat, A. “Using Docker: Developing and Deploying Software with Containers”. O’Reilly Media, Inc., 2015, 350p.

[28] NetworkX. “Networkx Website”. Source: https://networkx.org/, December 2021.

[29] Patel, P.; Ranabahu, A. H.; Sheth, A. P. “Service level agreement in cloud computing”, Technical Report, Wright State University, Dayton, OH, USA, 2009, 11p.

[30] RNP. “Ix.br”. Source: https://ix.br/particip/sp, December 2021.

[31] RNP. “Rede ipê”. Source: https://www.rnp.br/sistema-rnp/rede-ipe, December 2021.

[32] Rossi, F.; Cardellini, V.; Presti, F. L.; Nardelli, M. “Geo-distributed efficient deployment of containers with kubernetes”, *Computer Communications*, vol. 159, 2020, pp. 161–174.

[33] Santos, J.; Wauters, T.; Volckaert, B.; De Turck, F. “Resource provisioning for iot application services in smart cities”. In: International Conference on Network and Service Management (CNSM), 2017, pp. 1–9.

[34] Santos, J.; Wauters, T.; Volckaert, B.; De Turck, F. “Towards network-aware resource provisioning in kubernetes for fog computing applications”. In: IEEE Conference on Network Softwarization (NetSoft), 2019, pp. 351–359.

[35] Santos, J.; Wauters, T.; Volckaert, B.; De Turck, F. “Towards delay-aware container-based service function chaining in fog computing”. In: IEEE/IFIP Network Operations and Management Symposium (NOMS), 2020, pp. 1–9.

[36] Satyanarayanan, M. “The emergence of edge computing”, *Computer*, vol. 50–1, 2017, pp. 30–39.

[37] Satyanarayanan, M. “How we created edge computing”, *Nature Electronics*, vol. 2–1, 2019, pp. 42–42.

[38] Satyanarayanan, M.; Bahl, P.; Caceres, R.; Davies, N. “The case for vm-based cloudlets in mobile computing”, *IEEE Pervasive Computing*, vol. 8–4, 2009, pp. 14–23.

[39] Senthil Kumaran, S. “Practical LXC and LXD: linux containers for virtualization and orchestration”. Springer, 2017, 180p.

[40] Shi, W.; Cao, J.; Zhang, Q.; Li, Y.; Xu, L. “Edge computing: Vision and challenges”, *IEEE Internet of Things Journal*, vol. 3–5, 2016, pp. 637–646.

[41] Shi, W.; Dustdar, S. “The promise of edge computing”, *Computer*, vol. 49–5, 2016, pp. 78–81.

[42] Smet, P.; Dhoedt, B.; Simoens, P. “Docker layer placement for on-demand provisioning of services on edge clouds”, *IEEE Transactions on Network and Service Management*, vol. 15–3, 2018, pp. 1161–1174.

[43] Topcuoglu, H.; Hariri, S.; Wu, M.-Y. “Performance-effective and low-complexity task scheduling for heterogeneous computing”, *IEEE Transactions on Parallel and Distributed Systems*, vol. 13–3, 2002, pp. 260–274.

[44] Xavier, M. G.; Neves, M. V.; Rossi, F. D.; Ferreto, T. C.; Lange, T.; De Rose, C. A. F. “Performance evaluation of container-based virtualization for high performance computing environments”. In: Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP), 2013, pp. 233–240.

MARISTA

PUCRS

Pontifícia Universidade Católica do Rio Grande do Sul
Pró-Reitoria de Graduação
Av. Ipiranga, 6681 - Prédio 1 - 3º. andar
Porto Alegre - RS - Brasil
Fone: (51) 3320-3500 - Fax: (51) 3339-1564
E-mail: prograd@pucrs.br
Site: www.pucrs.br