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

LUIS AUGUSTO DIAS KNOB

IMPROVING CONTAINER DEPLOYMENT LATENCY IN DISTRIBUTED EDGE INFRASTRUCTURES

Porto Alegre
2022

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

IMPROVING CONTAINER DEPLOYMENT LATENCY IN DISTRIBUTED EDGE INFRASTRUCTURES

LUIS AUGUSTO DIAS KNOB

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

Advisor: Prof. Tiago Coelho Ferreto

Porto Alegre
2022

Ficha Catalográfica

K72i Knob, Luis Augusto Dias

Improving container deployment latency in distributed edge infrastructures / Luis Augusto Dias Knob. – 2021.
122 f.
Tese (Doutorado) – Programa de Pós-Graduação em Ciência da Computação, PUCRS.

Orientador: Prof. Dr. Tiago Coelho Ferreto.

1. Edge Computing. 2. Container management. 3. Orchestration. 4. Container scheduling. I. Ferreto, Tiago Coelho. II. Título.

Elaborada pelo Sistema de Geração Automática de Ficha Catalográfica da PUCRS
com os dados fornecidos pelo(a) autor(a).
Bibliotecária responsável: Clarissa Jesinska Selbach CRB-10/2051

LUIS AUGUSTO DIAS KNOB

IMPROVING CONTAINER DEPLOYMENT LATENCY IN DISTRIBUTED EDGE INFRASTRUCTURES

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

Sanctioned on August 30, 2021.

COMMITTEE MEMBERS:

Inicialmente, gostaria de agradecer a minha família pelo total apoio e incentivo durante minha caminhada acadêmica. Em especial, a minha mãe, Dinah, pela educação e valores ensinados, tendo sido sempre um exemplo para mim. Para você mãe, meu eterno agradecimento.

Agradeço também ao amor da minha vida, Natália, pelo apoio, pela compreensão e pela infinita paciência durante este período. Obrigado por ser minha amada companheira tanto nos momentos alegres, quanto naqueles não tão felizes. Saiba que te amo incondicionalmente.

Agradeço ao professor Tiago Ferreto pela oportunidade de realizar um doutorado em uma das melhores instituições de ensino do país. Também agradeço por todas as lições, ensinamentos e momentos de conversa que sem dúvidas foram essenciais na minha formação acadêmica. Também agradeço pela confiança em mim depositada, durante os diversos trabalhos que realizamos juntos neste período e que espero que continuemos realizando. Aos demais professores do PPGCC, agradeço pela qualidade das disciplinas ministradas e por sempre esperarem o melhor de mim. Tenho muito orgulho de ter sido aluno de um grupo tão distinto de professores.

Sou muito grato a todos os amigos que conquistei durante esses cinco anos, principalmente aqueles que tiveram que me aguentar nos primeiros anos de disciplinas obrigatórias e visitas esparsas ao laboratório. Obrigado Paulo, Ângelo, Felipe, Carlos e todos os outros pelo companheirismo. Agradeço também aos demais colegas de PUCRS que, embora não estejam aqui nomeados, foram fundamentais para esse momento.

Sono grato anche alla Fondazione Bruno Kessler, che mi ha accolto in uno dei momenti più complicati della storia e che, anche durante la quarantena, mi ha fatto avere la città di Trento come seconda casa. In particolare, ma non esclusivamente, ringrazio Domenico Siracusa, Silvio Cretti e Francescomaria Faticanti.

Finalmente, gostaria de agradecer a todos que de alguma forma incentivaram ou participaram, direta ou indiretamente de minha formação acadêmica.

“Isn’t it enough to see that a garden is beautiful without having to believe that there are fairies at the bottom of it too?”
(Douglas Adams)

ACKNOWLEDGMENTS

This thesis 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 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.

This thesis was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior – Brasil (CAPES) – Finance Code 001.

APRIMORANDO A LATÊNCIA NA INSTANCIAÇÃO DE CONTÊINERES EM INFRASTRUTURAS DE BORDA DISTRIBUÍDAS

RESUMO

Novos serviços, como realidade aumentada e processamento de linguagem natural, necessitam de níveis de processamento e comunicação que não são alcançáveis com Computação em Nuvem. Novos paradigmas, como *Multi-Access Edge Computing* e Computação em Névoa, ou genericamente Computação na Borda, surgem como solução para atender os requisitos destas aplicações. Entretanto, este paradigma apresenta diversos desafios, como o rápido e contínuo provisionamento de aplicações distribuídas geograficamente em equipamentos heterogêneos na borda, muitas vezes com recursos limitados. Atualmente, existem algumas estratégias para diminuir o tempo de provisionamento de aplicações em infraestruturas baseada em contêineres. Entretanto, as especificidades de um cenário utilizando Computação na Borda e os diversos componentes presentes nestas topologias possuem questões que precisam ser otimizadas antes da larga adoção deste paradigma. Desta forma, esta tese apresenta as seguintes contribuições. Primeiro, é apresentado um simulador baseado em eventos para orquestração de contêineres na borda. Depois, são apresentadas três contribuições em diferentes componentes destas infraestruturas, um algoritmo de posicionamento utilizando comunidades fluidas para os repositórios de contêineres, uma nova prioridade para o *kube-scheduler* baseada na disponibilidade de rede e, por último, um novo escalonador com foco no nível de garantia no tempo total de instanciação de contêineres utilizando um algoritmo genético multiobjetivo.

Palavras-Chave: Gerenciamento de Contêineres, Computação na Borda, Orquestração, Escalonamento de Contêineres.

IMPROVING CONTAINER DEPLOYMENT LATENCY IN DISTRIBUTED EDGE INFRASTRUCTURES

ABSTRACT

New services, such as augmented reality and natural language processing, require some network and processing thresholds that aren't possible with Cloud Computing. New paradigms near the end-user, like Multi-Access Edge Computing and Fog Computing, or generically speaking Edge Computing, emerged to bring these requisites to such applications. 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 few strategies to decrease application deployment time in container-based infrastructure. However, the specificities of an Edge scenario and the several components presents in these topologies have several points that need to be optimized before a large adoption of this paradigm. With that in mind, this thesis presents four main contributions. First, the development of an event-driven simulator to edge container orchestration. After, we give three contributions on distinct components, a fluid communities placement for the container registries, a new priority to the kube-scheduler based on the network availability, and a new Deployment SLA-driven scheduler using a multi-objective genetic algorithm.

Keywords: Container Management, Edge Computing, Orchestration, Container Scheduling.

LIST OF FIGURES

LIST OF TABLES

LIST OF ALGORITHMS

CONTENTS

1. INTRODUCTION

1.1 Container Orchestration on Edge Infrastructures

Edge Computing is an enabler technology to the new 5G networks. These networks seek to implement an infrastructure that allows applications, like augmented reality and natural language processing, to be used in real-time through low latency, positioning awareness, and geo-distributed processing power infrastructure. Edge applications usually need to be instantiated in highly distributed and heterogeneous scenarios, far from the well-managed Cloud Providers, increasing the complexity and the management cost.

Although having distinct architectures and being standardized by several consortia with different names, like Fog Computing[Bonomi et al., 2012] and Multi-Access Edge Computing (MEC)[Hu et al., 2015], generally speaking, Edge Computing aims at pushing computational capacity closer to the end-user. Despite its similarities with Cloud Computing, this creates a scenario that shows several new challenges in management and orchestration. Today, both academia and industry are working hard to implement solutions that improve the deployment of applications on the edge [Santoro et al., 2017][Wang et al., 2021].

Initially implemented on Virtual Machines (VMs) [Satyanarayanan et al., 2009], edge applications are rapidly changing to use containerization, enabling faster deployment, smaller footprint, and scalability. However, the geographic distribution, the heterogeneity of the physical infrastructures, and the applications' demands in these scenarios present management requirements that are not fully met by any solution currently available. In several works [Carella and Magedanz, 2016][Schiller et al., 2018], 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, within 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.

Typically, container images are stored on registries as small reusable parts or layers requested by worker nodes when a container image or layer not cached needs to be deployed. This download phase mostly consists of the time needed to instantiate a container, and applications that have a large set of microservices or replicas may require several downloads on different nodes simultaneously. That, in edge scenarios, is the main reason for the deployment latency, which lies in the image download from an external registry and can take several seconds on constrained-resource nodes.

1.2 Challenges to Decrease the Deployment Latency on Edge Infrastructures

There are many characteristics of edge computing that hinder the decrease in deployment latency [Fu et al., 2020, Wong et al., 2019], including but not limited to: i. resource-constrained and heterogeneous infrastructure; ii. geo-distributed nodes with several milliseconds round-trip time; iii. distinct necessities on scheduler and runtimes; iv. difficulties in validating new solutions. These issues can be summarized in the three challenges below.

- **Challenge 1: Validation of new solutions for container orchestration in large-scale infrastructures.** Simulation is a common ground on the experimentation of new solutions on infrastructure that are near impossible to be evaluated in real topologies or testbeds. Cloud and Fog computing are broadly attended by several simulators, mainly focused on resource utilization, like CPU, memory, and energy [Calheiros et al., 2011] [Varga, 2010]. However, container orchestration is not easily replicated on these simulators, largely because it is the pivot on the messages exchange between the several components on the infrastructure. Therefore, researchers usually develop custom simulators [Fu et al., 2020], that increase the difficulties on simulating new algorithm implementations and functionalities in this scenario.

- **Challenge 2: Implementation of a scheduler and runtime optimization algorithm to reduce the time needed to deploy and maintain applications on edge computing.** Some methods to decrease the deployment latency were already proposed in the literature, with a focus on the cache usage [Darrous et al., 2019] or changes on the container runtime [Ahmed and Pierre, 2018]. In addition, new scheduling algorithms were also presented by [Santos et al., 2019] and [Fu et al., 2020]. These works extend the Kube-scheduler adding new priorities based on latency and cache usage, showing promise results in constrained scenarios.

- **Challenge 3: Evaluation of registry placement and new image distribution strategies.** Some approaches propose improvements on how the containers are stored in or distributed to nodes. Pulling simultaneously the same image by multiple registries is presented in [Nathan et al., 2017], which also discusses a cooperative implementation of a set of registries. Solutions based on peer-to-peer communication are also presented in [Uber, 2021] [Kangjin et al., 2017]. However, these implementations rely on powerful worker nodes placed on high-speed networks, usually not replicable on edge scenarios, with geo-distributed topologies. Furthermore, adding a new daemon on a resource-constrained node can generate bottlenecks, as shown in [Ahmed and Pierre, 2020], where even simple applications running on it can generate high latency in the deployment of new applications.

1.3 Goals, Research Questions, and Approaches

1.3.1 Goals

On the basis of the challenges listed in Section 1.2, we define the set of research goals of this thesis as follows:

> ***Goal 1:*** *To investigate the container deployment process on edge computing, learn how the schedulers' solutions works, and understand how other components can impact this operation*

> ***Goal 2:*** *To investigate if there is any solution to simulate or emulate the orchestration of container-based large-scale edge infrastructure, and if necessary, evaluate the requirements to implement a simulator*

> ***Goal 3:*** *To improve the deployment process on edge infrastructure through new solutions on several phases of the deployment*

All these goals have as main focus, the improvement of the Deployment Latency in Edge Computing.

1.3.2 Research Questions and Approaches

In the first goal, we expressed that we want to investigate container orchestration's state of the art, mainly in edge scenarios. This leads to our first research question:

> ***RQ 1:*** *What are the differences between the cloud and the edge on the application deployment process? Are the actual solutions adapted to this largely heterogeneous and constrained-resource scenario?*

We address RQ 1 in all chapters of this thesis.

Our next research question is a direct consequence to the first one, where after understanding the differences in the deployment process and the edge computing specific necessities, we investigate if:

> ***RQ 2****: Is it possible to optimize the deployment process and reduce the latency created by them? If yes, what components should be optimized?*

We address RQ 2 in Chapters 4, 5, and 6.

After, based on the second goal, we want to research the main methods that can be used to validate the solutions implemented in this scenario. This leads us to the third research question:

> ***RQ 3****: How can we evaluate distinct solutions on edge scenarios? Is there any simulator that can be used? What are the main requisites for the simulation?*

We address RQ 3 in Chapter 3.

The next three questions are related to the third goal and are all generated as a result of the first two research questions, being them:

> ***RQ 4****: How the registry placement influence the deployment latency? How can we distribute the network load between several registries on the topology?*

We address RQ 4 in Chapter 4.

> ***RQ 5****: Does the scheduler uses any network information as input to schedule applications? Is it possible to implement a solution that uses the available bandwidth as input to the application schedule? How this impacts the other's priorities?*

We address RQ 5 in Chapter 5.

> ***RQ 6****: How the download queue on a node can be optimized to improve the deployment latency? Can the scheduler use the queue manipulation to ensuring service level agreements on the deployment total time?*

We answer RQ 6 in Chapter 6.

1.4 Organization and Key Contributions

This thesis is organized into seven chapters. In the following, we provide a summary for each chapter with corresponding publications and key contributions.

Chapter 2: *Background on Edge Infrastructures and Container Orchestration*

In this chapter, we explain the basic concepts for the best understanding of the remainder of the text. It starts with an overview of Edge computing technologies and related frameworks. After, we present a background to container virtualization and its orchestration.

Chapter 3: *Container orchestration simulation on large distributed infrastructures*

In this chapter, we underline the requirements to simulate container orchestration on a large distributed infrastructure after we briefly discuss why the current simulators are not fit for the majority of our scenarios. Finally, we present ECOS, a simulator focused on the container orchestration and communication between the several components, such as network abstractions, container nodes, and registries.

Chapter 4: *Registry placement to speed up application deployment*

This chapter discusses the importance of registry placement on the deployment latency on a distributed infrastructure. Furthermore, we present a Community-based placement strategy to instantiate on the topology a given number of registries. To evaluate its effectiveness, we carried a series of experiments on realistic and random networks with distinct container schedulers.

This chapter is based on the part of the following peer-reviewed paper:

Luis Augusto Dias Knob, Francescomaria Faticanti, Tiago Ferreto, Domenico Siracusa. *Community-based placement of registries to speedup application deployment on Edge Computing*. 9th IEEE International Conference on Cloud Engineering (IC2E 2021).

Chapter 5: *Container scheduling based on network bandwidth availability*

In this chapter, we investigate the use of network bandwidth availability as a priority on the scheduling process. In addition, we propose 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. 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.

This chapter is based on the part of the following peer-reviewed paper:

Luis Augusto Dias Knob, Carlos Henrique Kayser, Tiago Ferreto. *Improving Container Deployment in Edge Computing Using the Infrastructure Aware Scheduling Algorithm*. 26th IEEE Symposium on Computers and Communications (ISCC 2021).

Chapter 6: *Ensuring Service Level Agreement on Application Deployment in an Edge Infrastructure*

Chapter 6 aims to ensure a soft Service Level Agreement (SLA) to the total time needed to instantiate each application on a multi-level edge infrastructure. To accomplish that, we implement a new scheduler using a multi-objective genetic algorithm. We also modify how the container node implements the download queue, allowing containers that have not started to be downloaded to shift their orders. Finally, we validated our solution through a series of experiments using the Ipê Network.

This chapter is based on the part of the following peer-reviewed paper:

Luis Augusto Dias Knob, Carlos Henrique Kayser, Paulo Silas Severo de Souza, Tiago Ferreto. *Ensuring SLA Deployment Latency on Container Edge Infrastructure*. 14th IEEE/ACM International Conference on Utility and Cloud Computing (UCC 2021).

Chapter 7: *Final Considerations*

The last chapter summarizes the conclusions of the studies which have led to this thesis. We also revisit the goals and re-state the answers to the research questions. In addition, we outline some possible directions for future work.

2. BACKGROUND ON EDGE INFRASTRUCTURES AND CONTAINER ORCHESTRATION

This chapter describes the existing paradigms in the space between the cloud and the edge devices, or Cloud-to-Things continuum, to demonstrate the application's management complexity increase in a much broader spectrum than it is currently available. In addition, this chapter seeks to provide a better understanding on the relationship between those paradigms and how they can benefit from new technologies in the implementation of new and modern applications.

2.1 Cloud Computing

Cloud Computing is a model that allows ubiquitous, and on-demand access of a configurable set of computational resources that can be provided with a minimum of management effort [Mell and Grance, 2011]. This model has generated a significant impact in the IT industry through several companies, like Google, Amazon, IBM, and Microsoft that, when creating cloud platforms, sought to promote a more powerful, efficient, and reliable environment for new applications that may benefit from this model [Armbrust et al., 2010].

These platforms, called public clouds, are offered by major providers and are based on a model where computing capacity, storage, networks, and other resources are paid on demand. The private clouds designation is used to describe private infrastructures not available to the general public, but which have the characteristics presented by public clouds, such as allocation on demand for resources. Halfway, there are hybrid clouds, which represent the integration of services between a public and a private cloud managed by a company, and community clouds, where several companies share the same cloud infrastructure, which may or may not be managed by a third-party agent [Mell and Grance, 2011].

Regarding the business model, we can divide the services available in Cloud Computing into three categories: Software as a Service (SaaS), Platform as a Service (PaaS), and Infrastructure as a Service (IaaS). In the SaaS model, the application is made available to the user through the Internet and can be accessed from several devices using a simple interface, usually a browser. The user does not control or manage the application in this model, leaving this responsibility to the application provider. In the PaaS model, the user manages only the implementation of the application over the infrastructure maintained by the cloud. The management authority over the operating system and the libraries required rests with the platform operator. Finally, in the IaaS model, the cloud provider makes available the CPU, storage, and network infrastructure, usually through virtualization methods, and the user is responsible for maintaining the operating system, dependencies, and the application [Mell and Grance, 2011]. Figure 2.1 shows the differences between the three business models presented, taking into account management concerning on-premises infrastructure.

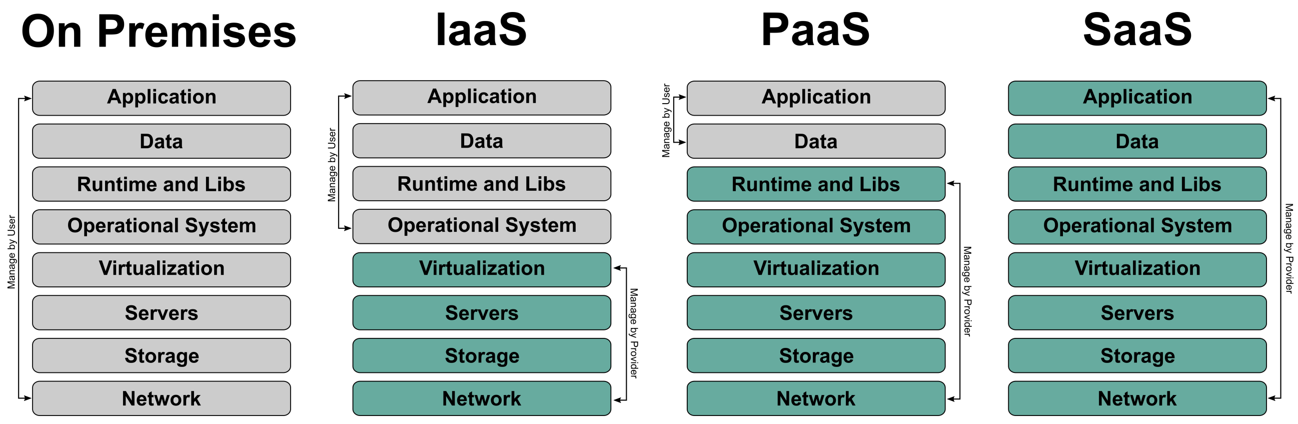


Figure 2.1: Services business model and management responsibility (adapted from [Mell and Grance, 2011])

Resource virtualization is one of Cloud Computing's key technologies, mainly because virtualization enables the scalability necessary for the deployment of services and applications in these infrastructures. However, because they are instantiated in isolate virtual resources, virtual machines (VM) require memory and disk space for the libraries and binaries of the hosted operational system OS. As a result, if several VMs are instantiated with the same OS or base library, e.g., Python or Java, this can generate redundancy on the hypervisor. It should also be noted that VMs usually have a slow startup and generally run a single process/service per instance [Pahl, 2015]. For this reason, several other solutions were presented aiming to improve the implementation of services in the cloud, such as containerization and serverless computing.

For the successful implementation of an application in the cloud, the orchestration and management of available resources have significant importance, both for scalability success and fast instantiation. However, resource management in the cloud is not trivial, as each platform uses different ways to describe resources, services, and tools. Thus, migrating an application between platforms is a task with great complexity since there is no standard for orchestrating and migrating resources. Recent efforts try to unify or approximate the implementation of resources between different cloud providers, such as Terraform[1], Apache Libcloud[2], and AWS CloudFormation[3].

Cloud Computing has revolutionized the deployment of applications, enabled several new market models, and accelerated the implementation of new technologies. However, some characteristics that can present downgrades in Cloud Computing, such as privacy, misuse, or lack of use of resources at the edge [Garcia Lopez et al., 2015], latency, and massive use of the network backbone communication by applications used

[1]Terraform, https://www.terraform.io
[2]Apache Libcloud, https://libcloud.apache.org
[3]AWS CloudFormation, https://aws.amazon.com/cloudformation

by many users. Hence, several authors and organizations [Garcia Lopez et al., 2015] [Satyanarayanan et al., 2009] [ETSI MEC-ISG, 2020] argue that there is a new decentralization era in the infrastructure, focusing on the use of equipment resources at the edge or using underused resources on the path between the cloud and the end-user.

In the following sections, several proposals are presented, both from industry and academia, which seek to improve the application infrastructure by integrating the cloud with technologies and equipment closer to the end-user.

2.2 Edge-Centric Computing

While Cloud Computing seeks to centralize computing power and application processing control in dense environments with an elastic infrastructure that could be expanded indefinitely, several studies [Garcia Lopez et al., 2015] [Satyanarayanan et al., 2009] emphasize that a new decentralization era can emerge by the constant growth of the computing power on users' equipment, sensors, and actuators. Privacy and control of the data generated by the end-user are among the main objectives mentioned by these works, since there is no guarantee that the network environment between the edge and the Cloud cannot be attacked, denied, or intercepted.

In the study presented by Lopez et al. [Garcia Lopez et al., 2015], the authors describe a new paradigm, where the processing and transfer of information between different equipment must occur primarily at the edge of the network, the cloud being used as computational support only when necessary. This concept, named *Edge-centric Computing*, predicts that proximity, intelligence, trust, and control of applications are characteristics that are located at the edge. Thus, with the constant increase of computational power in the actuators, sensors, and user equipment, it makes sense that they will again perform control and processing functions in addition to the cloud. Ultimately, the main feature that differentiates Edge-Centric Computing from technologies such as P2P is that services must be implemented jointly between edge devices and the cloud, while maintaining privacy and control in the userspace.

2.3 Cloudlets or Small Clouds

Cloud Computing presents several challenges that are impossible to be solved when used to instantiate applications that require low latency and high bandwidth, mainly because it has a high point-to-point latency, traffic congestion points, and high cost of communication over long distances [Wang et al., 2018]. Thus, for satisfactory implementation of these solutions, it is necessary to bring the application closer to the user through small servers or data centers.

Under those circumstances, Satyanarayanan introduced the concept of cloudlets [Satyanarayanan et al., 2009], which corresponds to a set of computers that are reliable, resource-rich, connected to the Internet, and available close to the edge, that serve as intermediaries between users and the Cloud. Positioned at strategic points, such as Shopping Centers, Regional Centers, or even offices, cloudlets function as a 'Datacenter in a box or small clouds, offering to process only to users close to them. Thus, when requesting an application that need a real-time interaction, with low latency or high bandwidth, the device searchs for the nearest cloudlet, which reduces the bandwidth used to communicate with the Cloud, the execution time, and consequently saving energy.

Based on virtual machines, a cloudlet can use two different approaches for managing applications. First, when necessary, the final device suspended a virtualized application and sent it to the cloudlet, which continues the application execution from there. Second, the user device maintains only the difference in the state between the template virtual machine already stored in the cloudlet and its unique data. Thus, the device only sent the delta between the image and the template running on the cloudlet, which refactors the VM and instantiates it.

Although it is one of the first works to present the use of computing at the edge of the network, cloudlets offer several implementation and scalability problems, in addition to relying on storage and constant interaction with mobile devices. Some of these problems are investigated in the literature [Fesehaye et al., 2012] [Jararweh et al., 2014].

2.4 Cloud Radio Area Network (C-RAN)

Like in Cloud Computing, the Radio Area Network (RAN) evolution converges towards the centralization of processing and services. A new architecture called Cloud Radio Area Network (C-RAN), presented in Figure 2.2, seeks to aggregate the computational resources of a set of base stations in a centralized virtualized pool. In this scenario, we can divide the base station services into two: Remote Radio Heads (RRH), which are the antennas responsible for receiving the mobile frequencies, performing signal amplification and digital/analog and analog/digital conversion; and the Baseband Units (BBU) that process these packets and forward them to the core of the data network [Wu et al., 2015].

Being a fundamental technology to the bandwidth increase needed in 5G networks, C-RAN can implement smaller access cells while consuming less energy and making the connection environment denser. The use of smaller cells has two main advantages: increased connection bandwidth and more efficient use of radio wave spectrum. However, denser networks can cause problems such as interference, and for this reason, the man-

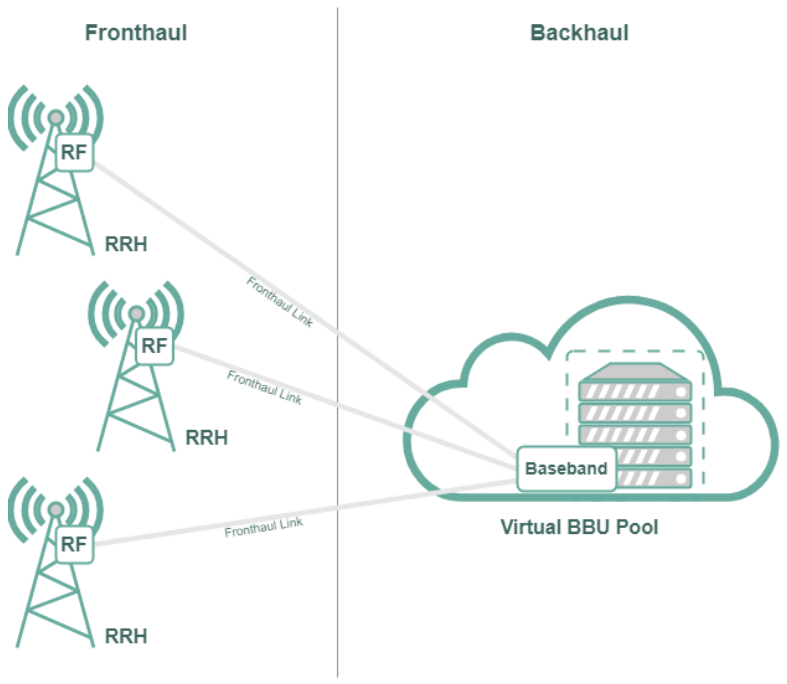


Figure 2.2: C-RAN Architecture (adapted from [Rost et al., 2014])

agement and processing solutions of nearby cell resources need to operate as a single infrastructure [Rost et al., 2014].

The implementation of C-RAN offers the potential to decrease energy costs for the operator, mainly due to the decrease in the number of BBUs compared to traditional base stations. In addition, during periods of low traffic (e.g., at dawn), some BBUs can be turned off without affecting network performance. Hence, savings with others infrastructure costs, such as cooling and hardware maintenance, can also be noted. The paper [Checko et al., 2015] shows that it is possible to decrease CAPEX and OPEX in the implementation of the structure by about 70%.

2.5 Multi-access Edge Computing

Multi-access Edge Computing (MEC), or until 2016, Mobile Edge Computing, is defined by a set of standards organized by the European Telecommunications Standards Institute (ETSI) and Industry Specification Group (ISG). Emerged in 2014, this framework provides an environment for applications, with the same characteristics as the Cloud Computing environment, together with the Radio Access Network (RAN) and close to the end-user. In addition, MEC is seen, together with Software-Defined Networks (SDN) and Network Function Virtualization (NFV), by the European 5GPPP (5G Infrastructure Public Private Partnership) research group, as one of the key emerging technologies for creating new 5G mobile networks.

Although the change from Mobile to Multi-access has brought a broader meaning to the standards defined by ETSI, the main focus of the specifications is on the availability of these MEC environments by mobile phone companies, which, together with technologies such as C-RAN, can provide access for applications by third parties to unused infrastructure close to the base stations [et al., 2018]. Among the benefits of using MEC as a component of an application's infrastructure, we can describe the reduction in latency and Internet access backbone bandwidth usage. In addition, it opens new opportunities for the development of applications, usually called local context awareness applications, that can benefit from information about the infrastructure such as user's position, moving pattern, or network traffic characteristics [Taleb et al., 2017].

The first white paper published by ETSI on MEC [Patel et al., 2014], presents the characteristics that define this new paradigm, namely:

- On-premises: A MEC platform can run in isolation from the rest of the network, only with access to local resources;
- Proximity: Because it is located close to the user, MEC is particularly useful for data processing and big data;
- Lower latency: Because it is located close to the user, it is possible to decrease the access time to a service, in addition to decrease congestion on the network core;
- Location awareness: By usually working close to distributed network devices, the network can leverage low-level signaling information to assist locating devices on the network;
- Network context information: Applications providing information about the network can take advantage of the MEC application model, since they can estimate the use and congestion of radio cells close to the user.

Among the several guiding documents made available by ETSI as a method to assist the development of Multi-access Edge Computing solutions, the Mobile Edge Computing (MEC); Framework and Reference Architecture (GS MEC 003) [ETSI MEC-ISG, 2020] presents a generic architecture and framework definition that determines a MEC infrastructure. Both are described in detail below.

The MEC framework consists of three levels: system, host, and network, as illustrated in Figure 2.3. They compose the entities and functions existing in the ecosystem of the infrastructure. At the host level, the MEC host provides the virtualized infrastructure for implementing MEC applications and the management platform. There are different access networks to the MEC infrastructure at the network level, such as local networks, mobile networks, and the Internet in general. At the top level, it describes the communication between the management system, user equipment, and third-party software.

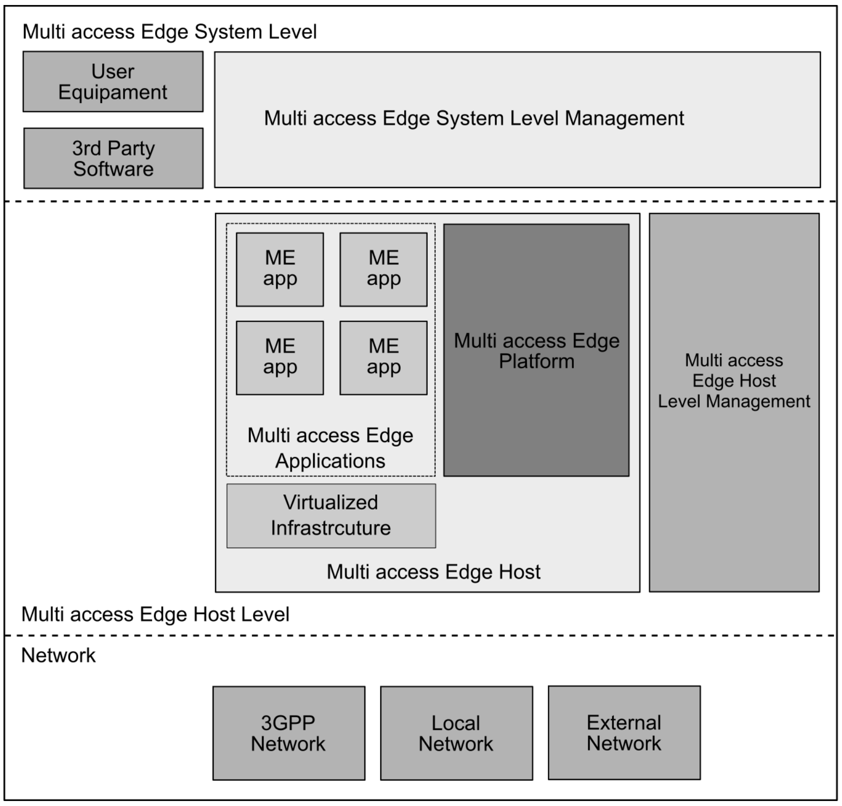

Multi access Edge System Level
User Equipament
3rd Party Software
Multi access Edge System Level Management
ME app
ME app
ME app
ME app
Multi access Edge Applications
Multi access Edge Platform
Multi access Edge Host Level Management
Virtualized Infrastrcuture
Multi access Edge Host
Multi access Edge Host Level
Network
3GPP Network
Local Network
External Network

Figure 2.3: MEC Framework (adapted from [ETSI MEC-ISG, 2020])

Figure 2.4 presents a simplified version of the architecture proposed by ETSI [ETSI MEC-ISG, 2020]. The main objective of the architecture is to present the relationship between the different agents/modules present in the framework to facilitate its modeling and implementation. First, the entities are grouped in two different levels, system and host. At the system level, there are the operational support system (OSS/BSS), the Multi-access Edge (ME) orchestrator, the CFS portal, and the applications on users' equipment, with the first two being responsible for generating an interface between the user and the infrastructure on the host MEC [Sabella et al., 2016]. Meanwhile, the host level is formed by the MEC host itself, the MEC management platform, and the virtualized infrastructure platform. The MEC platform is located on the host, which the features required for the applications, called ME services. This set of features is responsible for instantiating, controlling ME applications (ME Apps), and interacting with the virtualized infrastructure. This layer also presents the interaction interface of the MEC platform with the same module of another host, used for the instantiation of services between multiple domains or equipment.

With the definition of the architecture and framework for a MEC infrastructure, ETSI also provides other guiding documents related to this technology. Among the available documents, are the definitions of requirements for management APIs, use cases, and interactions with other technologies, such as NFV. In addition, MEC and similar technologies, like Fog Computing, share several characteristics. With that, new standards and research in the area are constantly published and updated.

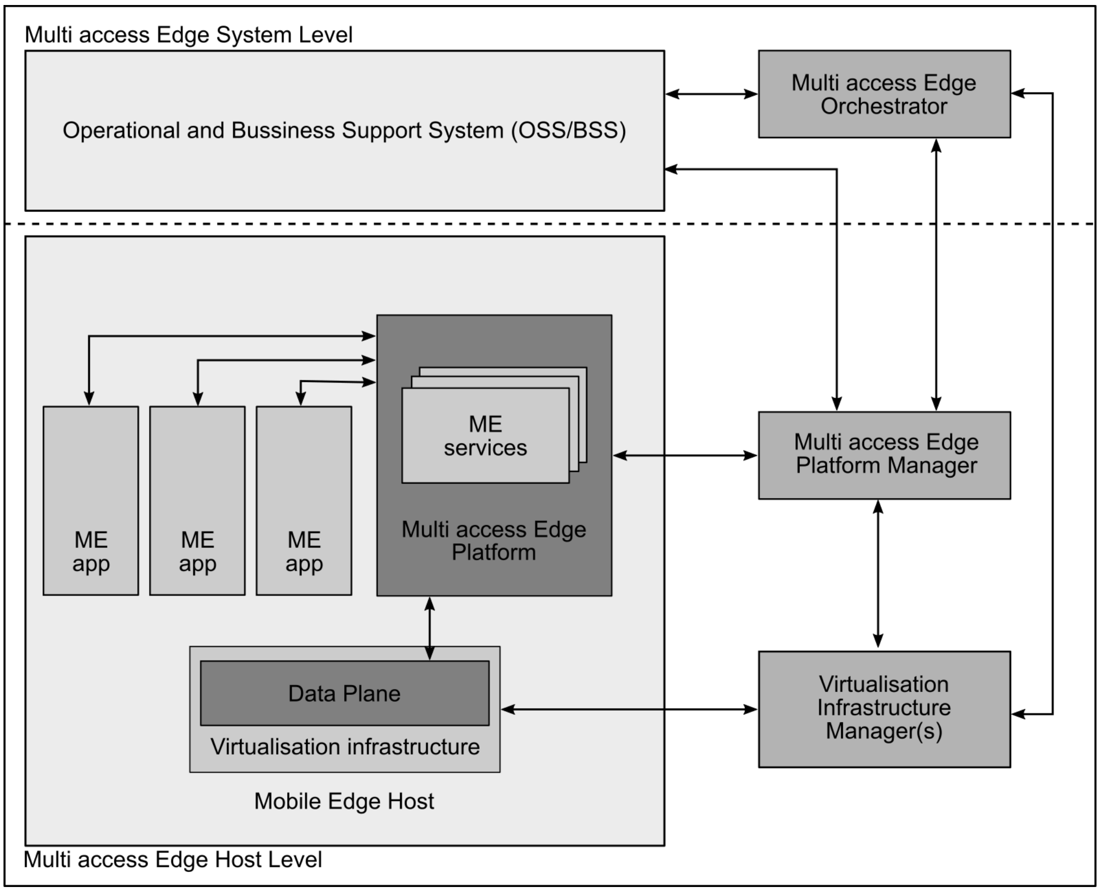

Multi access Edge System Level
Operational and Bussiness Support System (OSS/BSS)
Multi access Edge Orchestrator
ME services
ME app
ME app
ME app
Multi access Edge Platform
Multi access Edge Platform Manager
Data Plane
Virtualisation infrastructure
Virtualisation Infrastructure Manager(s)
Mobile Edge Host
Multi access Edge Host Level

Figure 2.4: MEC Architecture (adapted from [ETSI MEC-ISG, 2020])

2.6 Fog Computing

Fog Computing, a terminology introduced by Cisco in 2012, extends the Cloud Computing paradigm to the edge of the network, allowing a new set of applications and services. Serving as an intermediary between the Cloud and users, Fog Computing is defined by characteristics of low latency, location awareness, broad geographic distribution, mobility, predominant wireless access, and heterogeneity [Bonomi et al., 2012].

Although they present similar services, such as processing, storage, and network services, the Fog and the Cloud have significant differences. Fog Computing seeks to attend a specific geographic space, solves communication problems in real-time, and the application's location awareness. In comparison, although it presents a greater availability, Cloud Computing is usually centralized and distant from the end-user, struggling to serve applications with these characteristics.

Thus, Figure 2.5 presents the architecture that illustrates the role of Fog Computing in the connection between the Cloud and a Sensor Network or IoT. This configuration uses its intermediate position in the communication to allow the Fog to be monitored in real-time, allowing data analysis with low latency. In addition, there is a decrease in traffic that is

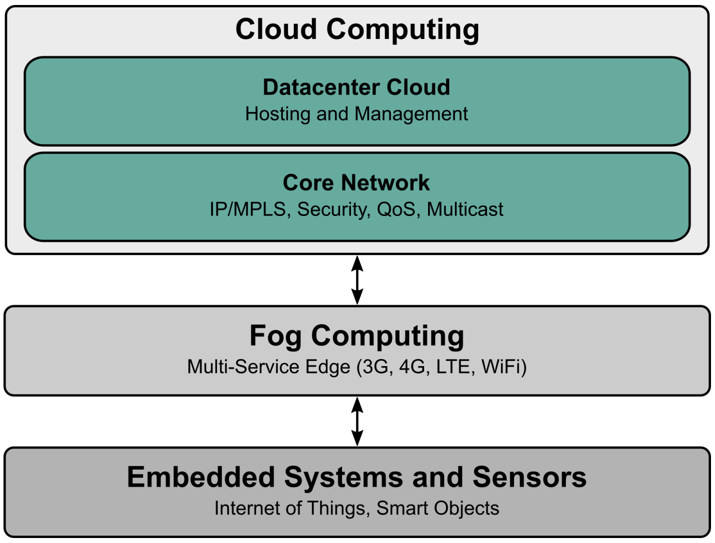

Cloud Computing
Datacenter Cloud
Hosting and Management
Core Network
IP/MPLS, Security, QoS, Multicast
Fog Computing
Multi-Service Edge (3G, 4G, LTE, WiFi)
Embedded Systems and Sensors
Internet of Things, Smart Objects

Figure 2.5: Fog Computing Architecture

forwarded to the Cloud by the core of the network, reducing the load and enabling new applications.

Other authors have expanded this initial definition of Fog Computing. Vaquero [Vaquero and Rodero-Merino, 2014] defines Fog Computing as a scenario where many heterogeneous and decentralized devices communicate and potentially cooperate with the network to provide storage and data processing without human intervention. Although this new definition may be questionable, it extends the concept of Fog, not only to auxiliary equipment interconnected to wireless network systems, but also to complete data centers on the edge of mobile and Internet networks. Therefore, there is a certain similarity with the concepts of Multi-access Edge Computing previously seen.

2.7 Virtualization

Virtualization is one of the oldest ways to share infrastructures, having fundamental importance for implementing Cloud Computing and other paradigms close to the edge. Satyanarayanan [Satyanarayanan et al., 2009], in one of the first works that deal with process computing on Edge, uses the distribution and maintenance of virtual machines (VMs) to provide services on small servers in offices, shopping malls, or wireless access points. Likewise, guiding documents for both Multi-access Computing and Fog Computing describe the use of a virtualized infrastructure, mainly through VMs.

Thus, several works that seek to solve problems in the instantiation and development of applications on edge, when linked to infrastructure issues, use virtualization solutions. The work presented by Osanaiye [Osanaiye et al., 2017] discusses different types of algorithms for live migration in Xen, presenting an optimized proposal for Fog Computing, based on the pre-copy of information to the destination infrastructure, taking into account the resource prediction and the provisioning algorithm used. In the same way, Ha [Ha et al., 2017] presents a solution that tries to predict which will be the next access point that the user will use to accelerate the VM migration process, while Zhang [Zhang et al., 2018] presents a solution that optimizes the migration of VMs between different devices performing the caching of the image in advance.

In addition to the migration and instantiation of resources, some works show solutions to other questions, like the use of accelerators, such as NetFPGAs and GPGPUs, in edge infrastructures [Varghese et al., 2018a], the use of Blockchain for the distribution of applications at the edge [Varghese et al., 2018b], and VM placement algorithms [Aryal and Altmann, 2018] [Jain and Tata, 2017]. Other technologies that can improve the instantiation of resources in the cloud can also be used at the edge. In the paper presented by Xavier [Xavier et al., 2016], for example, the difference in the provisioning time for VMs, Containers, and Unikernels is evaluated.

2.8 Containerization

Virtualization through Hypervisors provides complete isolation between services and applications using virtual machines. So, the network services to information exchange between applications are similar to those available between physical machines. Meanwhile, modern operating systems present weak isolation solutions, usually through process abstraction, but that allow through some mechanisms, the sharing of information, the file system, or access to global processes [Soltesz et al., 2007]. This trade-off, between complete isolation and the abstraction of processes, has always had a central point, the impact that each process could generate on others by dividing the same kernel. Thus, recent advances in both Linux, with the cgroups and namespaces [Tosatto et al., 2015], and the new versions of Windows Server, have consolidated an existing technology that is present for more than ten years, with the tools Linux-VServer [4] and OpenVz [5], called Containerization, or virtualization at the operating system level.

When instances of virtual machines are used to provide services in cloud environments, they are allocated through large isolated files, which usually run a single service. Although isolation guarantees a high level of security, the necessary cost for this is the complete instantiation of a new operating system. That happens even if this operating system has already been installed in another VM on the same server, causing the highest consumption of both RAM, CPU time, and disk. In addition, virtual machines are slow to start and

[4]Linux- VServer, http://linux-vserver.org
[5]OpenVz, https://openvz.org/

may take more than 10 minutes to boot up [Pahl, 2015]. Figure 2.6 presents the architectural differences between VMs and Containers.

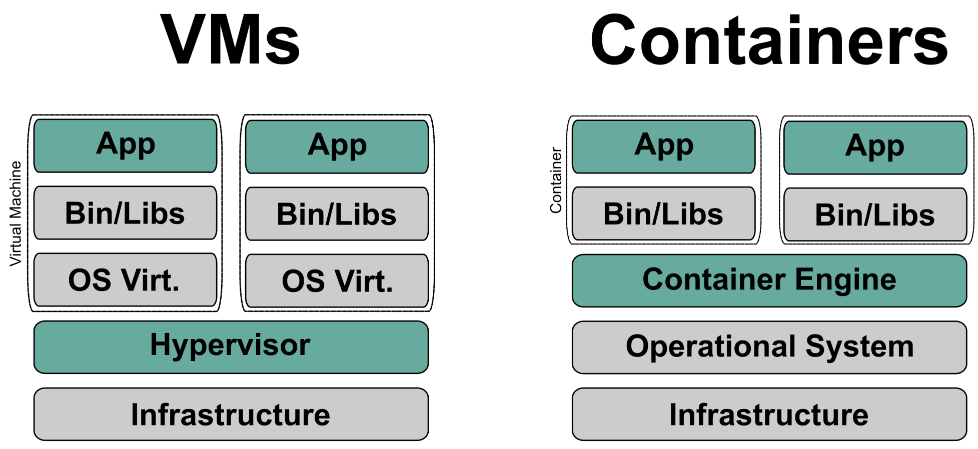


Figure 2.6: Differences between VMs and Containers (adapted from [Tosatto et al., 2015])

As can be seen in Figure 2.6, unlike virtual machines, when containers are used, applications share the same operating system, which leaves their images significantly smaller than when instantiating via Hypervisor. That allows the provisioning of hundreds of containers on a single host compared to the limited number of VMs that usually can be maintained in a single node. Because they use the same operating system as the host and, consequently, already have the kernel and shared libraries already allocated in memory, the instantiation time of a container is also shorter, only requiring to initialize the application inside the container [Bernstein, 2014].

Container provisioning provides several benefits for instantiating a heterogeneous IoT environment, allowing the implementation of services on demand through the availability of nodes and the dynamic configuration of devices. Among the main advantages of using a container to provide services at the edge, there is the low overhead of the applications, the fast instantiation of new services, and the high density of different applications on a single host [Morabito et al., 2017].

2.8.1 Docker

Docker is an open-source project started in 2013 by the dotCloud company, which worked primarily with platform development as a service solution. Developed in Go, a language made by Google, the project soon grew with the community's support, and after a few months, the company joined the Linux Foundation and changed its name to Docker Inc. Docker is also one of the largest projects hosted on Github [Merkel, 2014]. Currently, the organization maintains client versions for Linux, Windows, and macOS, with constant development on all platforms. Recently, in addition to the default Linux Container, it started the development of the Windows Server Container in partnership with Microsoft, which uses the same concept, but running on a Windows [Casalicchio, 2019] kernel.

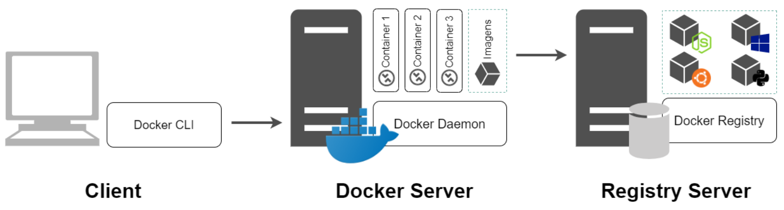


Figure 2.7: Docker Communications Components

The Docker functionalities are based on the container engine, called Docker Engine, which is responsible for the containerization technology and includes all the software components for managing the Docker containers. In addition, it provides a series of APIs, which, among other factors, is one of the reasons for the high adoption of this technology [Morabito et al., 2017]. Figure 2.7 presents the different layers of the architecture implemented by Docker. After defining the necessary settings for creating the container, the Docker Engine sends the settings to the containerd, which is responsible for the system-level management of the containers. It then uses one of the runtimes to execute it. Although runC is traditionally used, other projects such as Kata Containers [6], seek to present alternatives, mainly in terms of security and service isolation.

Each container in Docker is based on an image and a set of settings stored in a configuration file called Dockerfile. Images are static snapshots containing a series of layers, being each layer stored in read-only mode and upper layers work incrementally through a file system called UnionFS [Tosatto et al., 2015]. Each time a container is started, a new layer with read and write permission is added, which keeps the files changed at the time of execution. By default, data is not kept after the destruction of a container. So, it is necessary to link a static volume to the Docker container for stateful storage. This volume can be either a folder on the host, a network map, or a disk partition. In this way, the layered disk image system allows for a significant saving of disk space used by the containers, since each image is loaded only once in the system, through the read-only common layers, even if used by several containers.

Another important feature of Docker is that all images can be stored in a central server, called Registry. This server keeps the distinct layers for each application and sends them to the container node when requested. The Docker officially maintains the Docker Hub, a community library, where the world's largest companies keep containers for various services, such as Apache, Oracle DB, or NodeJS. It is also possible for a company to maintain closed container repositories, or even a private Registry [Truyen et al., 2016]. Figure 2.8 presents a view of the three actors in the life cycle of a container, the Docker Client, Host Docker, and Docker Registry.

[6]Kata Containers, https://katacontainers.io/.

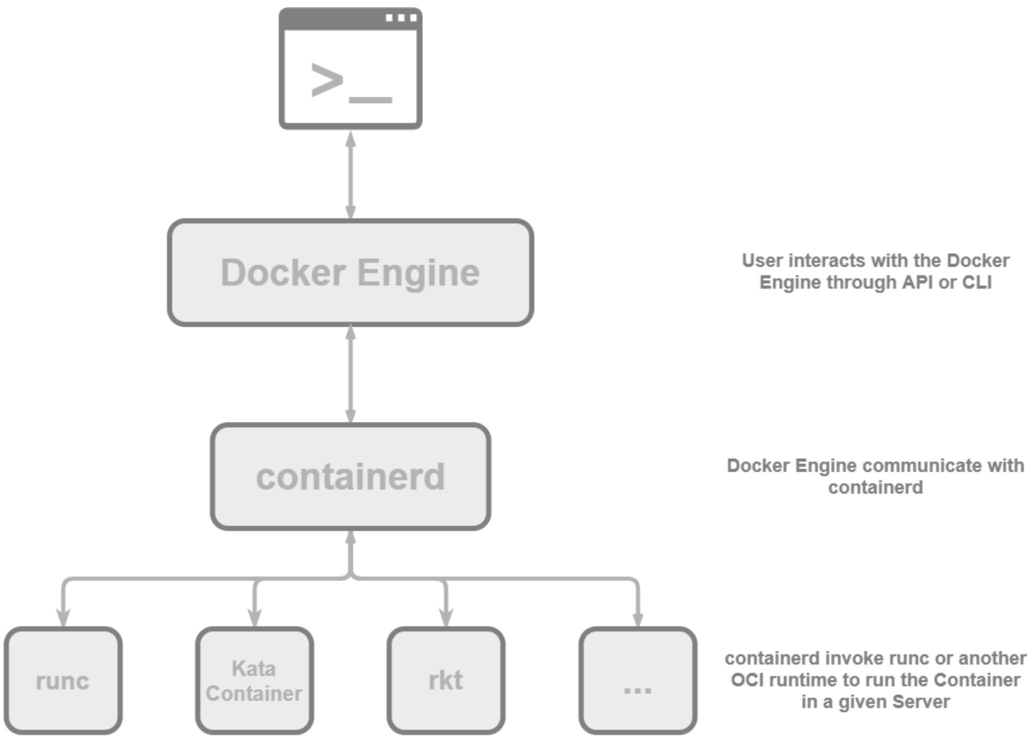

>_
Docker Engine
User interacts with the Docker Engine through API or CLI
containerd
Docker Engine communicate with containerd
runc
Kata Container
rkt
...
containerd invoke runc or another OCI runtime to run the Container in a given Server

Figure 2.8: Docker Architecture

2.8.2 Docker Container Orchestration

Docker container orchestration systems represent a central element in the instantiation and management of multiple containerized applications distributed on different hosts, whether physical or virtual, especially in Data Center environments [Taleb et al., 2017]. That happens due to the characteristics of containerized applications, which are traditionally elastic and replicable. The most popular solutions for Container orchestration are Kubernetes [7], Docker Swarm [8], and Apache Mesos [9].

These solutions, although they have independent characteristics and management using objects with different models for each service, have as common point the objective of providing mechanisms that allow instantiation, maintenance, and scale of applications between multiple hosts, facilitating access to them through a proxy, regardless of the host the container is hosted on [Velasquez et al., 2018]. This is usually done through a structure with two well-defined logical entities: the manager node, responsible for maintaining the state of the cluster and distributing the containers among the multiple hosts, and the worker node, responsible for receiving and instantiating the containers in the infrastructure.

[7]Kubernetes, https://kubernetes.io
[8]Docker Swarm, https://docs.docker.com/engine/swarm
[9]Apache Mesos, http://mesos.apache.org

Although the benefits of using containers are already seen in several applications and researches when used as an auxiliary technology for the implementation of Fog/Edge Computing [Ismail et al., 2015] [Yousefpour et al., 2019]. Several issues remain open in the application's orchestration in highly distributed environments, such as the implementation of live migration, monitoring [Varghese et al., 2016], data persistence, container distribution, and access between multiple regions and federation resources.

2.8.3 Container Lifecycle Operations

The container orchestration relies on a series of lifecycle operations to manage the application on the infrastructure, usually maintained by a controller responsible for the application from its deployment until the complete deletion from the topology. The three main operations are the deployment, the update, and the delete [Ahmed and Pierre, 2018]. The last is straightforward, and after the orchestrator receives the command, it removes every information and configuration related to the application on the infrastructure.

However, the deployment has a more elaborated path before the execution of the container runtime. First, one common mistake is believing that all replicas from an application are treated as a single instance. Each replica will have a separate scheduler and lifecycle that is distinct from the application one. After selecting the node that will receive a given replica, the orchestrator sends a message to this node, starting the deployment process.

The deployment process on the container node usually has a series of steps as follows:

- Find on cache the container manifest from the application, if not find, download it from the registry;
- Find on cache each layer from the given image, if not find, download it from registry sequentially;
- If more than three layers need to be download, and it creates a queue that waits until the next download slot be available;
- Each downloaded layer are downloaded as a gzip file that needs to be decompress;
- When all layers are available, create the read-write layer and start the container;
- Applies all the resources constraints and creates the network bonds;
- Warn the orchestrator that the deployment was complete.

From all these steps, the more network-intensive is the one that downloads the layers from the registry to the node, and the most CPU-intensive process is the gzip files decompress [Ahmed and Pierre, 2018]. Since cloud infrastructure usually relies on powerful servers with large bandwidth, these processes are typically neglected on the total cost from the deployment. However, on edge resource-constrained nodes with small bandwidth, the deployment process can take several seconds or even minutes to be done. This total time to deploy a container is also called Deployment Latency [Fu et al., 2020]. Several small tweaks were already developed to decreased this latency, on the runtime [Ahmed and Pierre, 2020], on the image creation [Huang et al., 2019] and even on the orchestrator scheduler [Darrous et al., 2019]. Still, we understand that this latency continues to be a majorly open problem on edge infrastructures.

Finally, the update process works basically as an integration between the delete and deployment. Since there is no update on the running container, each newly updated replica needed to be created as a new application, been the old one deleted after it starts.

2.8.4 Kubernetes

Kubernetes is an open-source orchestration system for automating deployment, scaling, and management of containerized applications, initially developed by Google and maintained by the Cloud Native Computing Foundation (CNCF) [Kubernetes, 2021b]. Kubernetes is built from a set of composable modules through a standard API that can be extended by the users, alongside the core components [Burns et al., 2016], allowing the development of several applications, like function-as-a-service frameworks[Ellis, 2021], and multi-cluster management [Rancher-Labs, 2021].

Kubernetes uses controllers to manage their objects, composed of three basic fields: metadata, specification, and status. The object metadata contains information about the object, such as UID, name, and labels. The specification describes the desired state of the object, and the status provides read-only information about the current state of the object [Burns et al., 2016]. Thus, these controllers attempt to ensure the desired state described in the object specification and, when necessary, return the current state in the object status field.

The smallest deployable object on Kubernetes is the *Pod*. *Pod* is a group of one or more containers that share resources, shown as a single access point for the other objects in the cluster. *Pods* are designed to be ephemeral and represent a single instance of an application in Kubernetes. They are rarely created as an individual object and usually use a more complex controller, such as *StatefulSets* and *Deployments*, to manage their instantiation and replication on the cluster nodes.

Deployment is the main controller used by Kubernetes to create elastic stateless applications. The Deployment Controller organizes *Pods*, automatically adjusts the number of instances, using a *ReplicaSet*, based on the desired state described in the configuration object YAML file. Deployments also control the update of the template used to create Pods through two different techniques, recreate and rolling update, and enabling rollback to an early deployment revision, if necessary.

For stateful applications, Kubernetes has the *StatefulSet* controller, that differently from the *Deployment*, maintains the order and uniqueness of each Pod created. To do that, the *StatefulSet* does not use *ReplicaSet* to control the desired number of replicas for each application, doing it by itself. To ensure data integrity, each *Pod* is instantiated with its own Persistent Volume Claim (PVC) and maintains its state. So, in an update, for every old Pod that terminates, a new *Pod* is created. But, since it does not use a *ReplicaSet*, it is impossible to rollback to an early *StatefulSet* revision after the update.

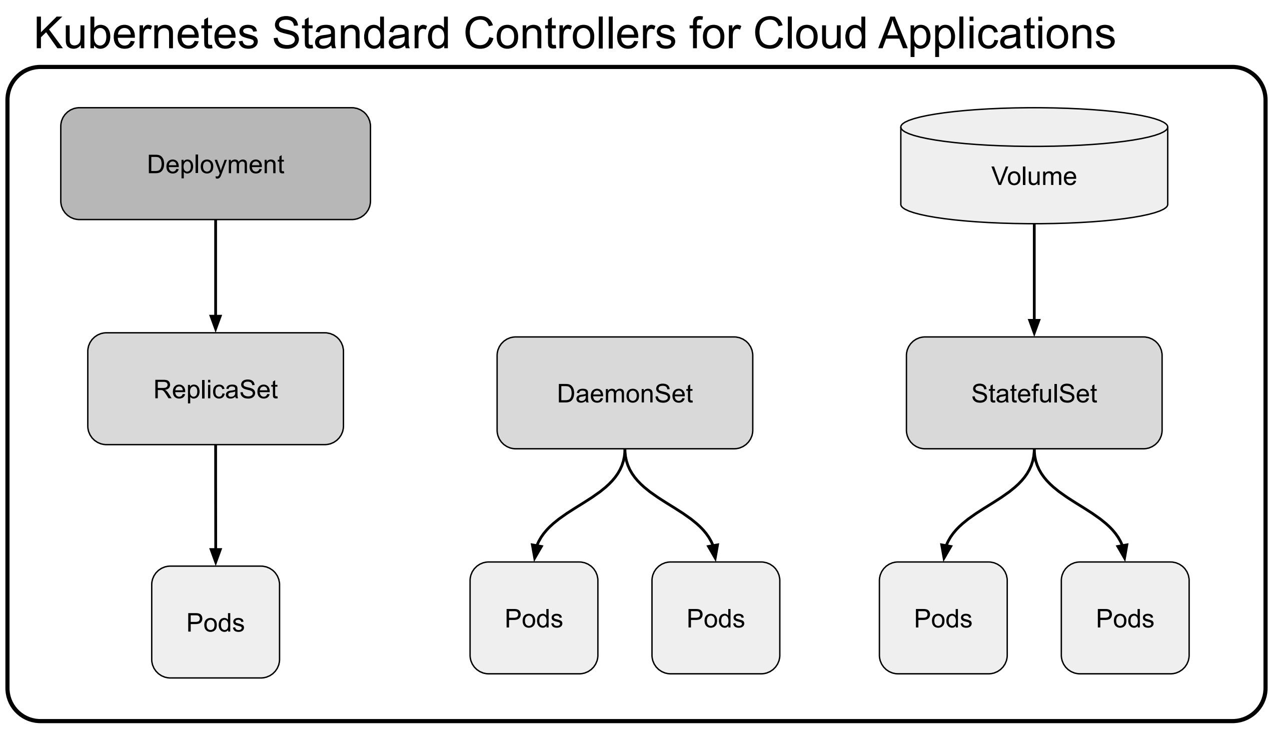


Figure 2.9: Kubernetes default controllers

For applications that need to be instantiated and maintained on all nodes, such as networking, monitoring, or logging applications, Kubernetes introduces a third controller named *DaemonSet*. These three controller types, with their peculiarities, are presented in Figure 2.9. Kubernetes standard controllers are designed for a Cloud Computing environment with low latency and high bandwidth between nodes, and several papers present issues in using Kubernetes for distributed and heterogeneous environments, such as Edge Computing [Fahs and Pierre, 2019][Faticanti et al., 2019][Wobker et al., 2018].

These works describe solutions related to the access of services through distributed points in the network [Fahs and Pierre, 2019], the placement of the applications [Faticanti et al., 2019], the definition of the hardware requirements, and the implementation of these applications in different processor architectures. However, no work presents a solution for instantiating applications in a distributed topology, allowing to manage the number of replicas in different regions from a single controller.

2.9 Closing Remarks

This chapter discussed the main technologies that make up the Cloud-to-Things continuum, from the Cloud to the framework and technologies closest to the user, such as Edge Computing and Fog Computing. In addition, we also present the several components from the container deployment process. Based on these technologies and paradigms, the next chapters seek to present a series of solutions to speed up the container deployment at the edge.

3. CONTAINER ORCHESTRATION SIMULATION ON LARGE DISTRIBUTED INFRASTRUCTURES

3.1 Introduction

Edge computing is one of the key technologies for new applications, like augmented reality and natural language processing. However, given its high complex and heterogeneous infrastructure, it becomes onerous to emulate or implement an extensive and close-to-real testbed to validate new solutions quickly. Simulators are highly tailor-made tools that enable rapid changes in configurations for modeling and analyzing a diversity of policies and options. These tools are fundamental to new research, since they simplify the validation step, and may be used to stress new algorithms before a more costly validation stage on real scenarios.

While application scheduling and node utilization simulators have already been extensively discussed in academia with several stable and well-documented solutions [Calheiros et al., 2011] [Nikdel et al., 2017], container orchestration still lacks better support. This happens mainly because the container orchestration relies on the middle ground between network and node utilization, and a large amount of simultaneous application scheduling with specific configurations, like image overlay and registry placement. So, actual solutions are hard to adapt or become highly complex to use in this type of scenario. Some authors (e.g., [Fu et al., 2020] and [Fahs and Pierre, 2019]) prefer to implement customs simulators to validate their works. However, this approach still increases the validation time since a well-implemented simulator takes time to be developed.

In this chapter, we present a discrete-event simulator focused on container scheduling in highly heterogeneous and distributed infrastructures, called ECOS (Edge Container Orchestration Simulator). First, we rapidly describe the solutions on container and edge simulation scenarios, detailing their main limitations. After, we formalize the problem definition, describe the design, and present details on the implementation of each component of our simulator. The simulator is used in the experiments to validate the contributions presented in Chapters 4, 5, and 6.

Also, in this chapter, we want to answer the following research questions:

- *How can we evaluate distinct solutions on edge scenarios? Is there any simulator that can be used? What are the main requisites for the simulation?*

3.2 Related Work

Simulation is a common ground in cloud and network research. Based on distinct objectives and features, several simulators are widely used like CloudSim [Calheiros et al., 2011] and OMNet++ [Varga, 2010]. These simulators tend to focus on the core functionalities of their respective scenarios, delegating new features to extensions or derived works. So, it is not strange that both simulators present related works that address container usage in their contexts.

ContainerCloudSim [Piraghaj et al., 2016] is an extension to the CloudSim Simulator that implements containers management on a Cloud infrastructure, mainly focusing on the node resource management and usage after deployment. As well as CloudSim, this work presents several functionalities that simulate aspects like memory, CPU time-sharing, and storage usage. Although it has a scheduling module, some implementations such as image download are not implemented. Meanwhile, based on the iCanCloud OMNet++ simulator, the DockerSim presented in [Nikdel et al., 2017], is a simulator that seeks to simulate the Docker behavior in a well-defined and fully integrated network scenario, implementing all TCP/IP stacks, daemon functionalities, and communication between containers on the network. However, like the ContainerCloudSim, this simulator does not implement orchestration functionalities, being focus on the containerization as a solution to sharing a giving node between several applications. Both simulators does not have, for example, the communication between the container host and the registry where the image are download. This behavior is fundamental to us, since the bandwidth between the registry and the node is one of the most important factor to the deployment latency.

YAFS, presented in [Lera et al., 2019], is a simulator of Fog applications, focusing mainly on the transfer of messages between applications, latency, and mobility between nodes. However, YAFS does not abstract the application management process, making it impossible to measure the orchestration cost and delays related to the application deployment or update. Another limitation of YAFS is in the network management, since it provides little control over the sharing of the network infrastructure between messages. This can be a problem when a large number of applications is instantiated at the same time. This creates limitations impossible to be surpassed, making it impossible to use to simulate application orchestration.

In [Fu et al., 2020], the authors implemented a Kubernetes cluster simulator to validate a scheduler based on dependencies. This is one of the first simulations related to large container orchestration. Among the reasons to develop a custom simulator described by the authors was the fined-grained control on parameters to improve the validation. However, the simulator developed has several limitations, with no concurrent transmission and direct connection between all nodes. Although the authors made available the source code, it is strongly bound to the scheduler developed, making it almost impossible to generalize it to other schedulers or scenarios.

After reviewing the main simulators present in the literature, it is clear that the available simulators are not adequate to our needs in a container orchestration scenario without requiring extensive modifications and abstractions. We can cite as limitations from the current simulators: i.) The container deployment process implementation, with the registry communication, including the cache validation, layers control, and version availability on the registries nodes; ii.) A well-defined scheduling template enables complex algorithms, like the default implementation from Kubernetes (Kube-scheduler); and, iii.) The lack of maintenance in the current solutions. Therefore, this work aims to simulate aspects on several objects and steps from the deployment process, and the only way to guarantee that is by implementing a new simulator.

3.3 Problem Definition

The simulation scenario is a container edge infrastructure composed of a set of nodes, whether bare metal or virtual, which are geographically distributed. Bidirectional links connect the nodes, and the communication between any two nodes uses the shortest path algorithm based on the bandwidth available between the nodes. We abstract the orchestrator as a global entity with direct communication to each node, and a registry node is locally connected to the infrastructure. The infrastructure is managed in a multi-tenant single-orchestrator mode and populated by a set of applications.

Periodically, a set of applications' lifecycle operations is performed. Usually, each application's first operation will be scheduled and may have $a = 1, \ldots, \infty$ number of replicas and generate a load on the network. This load occurs because a new image has to be deployed from the registry to each node defined by the scheduling algorithm. On Data centers, this problem is dismissed by the virtually unlimited network resource available. Still, in an edge scenario, with the heterogeneity on the network links and edge nodes, and the application location awareness, bottlenecks can be created on the infrastructure, increasing the expected amount of time needed to instantiate multiple containers simultaneously. Even a single congested link can consistently degrade all operations on the infrastructure. Although running applications can generate background traffic that may reflect on the updates, in our base simulation scenario, we disregard this traffic from the model, taking into account only the management load used by these operations.

3.4 Formalization

Table 3.1 presents a description of each term used in the problem formalization.

Term	Description
N	set of nodes
s	image registry node
L	set of network links
$T = (N, L)$	graph representing the network composed by N nodes and L links
(x, y, bw)	represents of a link $l \in L$ from nodes x and y $\in N \cup s \mid x \neq y$, with bandwidth bw
R	set of lifecycle operation requests
(t, D, m)	represents of a operation request $r \in R$, with t equals to the timestamp, D a set of node $\in N$ and m the container image
tt_r	the *total time* to instantiate the request r
$ct_{r,d}$	the *amount of time* need to provision request r on node d
$l_{r,t}$	represents the set of r operation that pass-trough a given link l on time t
$r_{bw,t}$	the *total bandwidth* available to the operation r on time t
n_{dque}	the active *download queue* on node n

Table 3.1: Terms used in the problem formalization

Let N be the set of edge nodes deployed by the applications and s the image registry in the topology. A set of bidirectional links connects all nodes L. Each link $l \in L$ is defined by (x, y, bw), where x and y are nodes $\in N \cup s | x \neq y$ and bw being the bandwidth capacity in both directions. We consider the edge topology defined by $T = (N, L)$ as an undirected graph with no self-loops and parallel links. Figure 3.1 shows a topology example based on the formal definition.

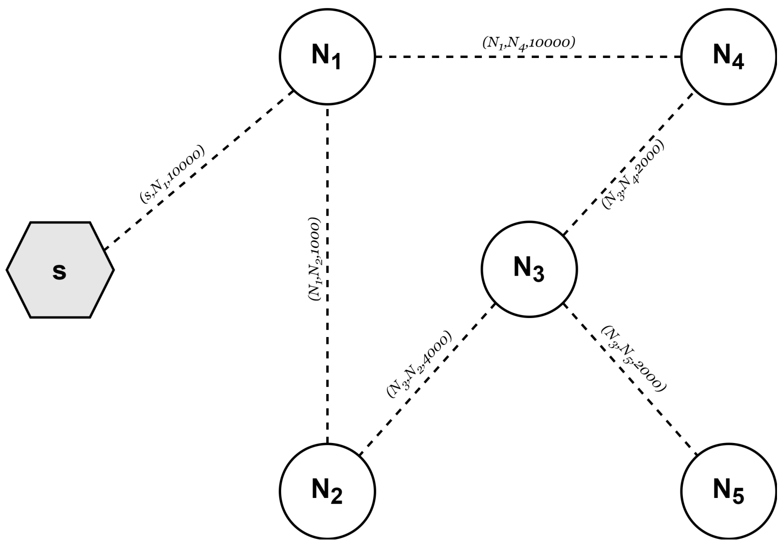


Figure 3.1: Topology example

Periodically, a set of lifecycle operations R are provisioned on the infrastructure. Each $r \in R$ is defined by (t, D, m), where t is the time when the request is submitted, D is the set of nodes $\in N$ where the operation will be executed, and m is the container image. Finally, the path between a node n to the registry s is defined by $P_d = (l | l \in L)$ where $d \in N$ uses the shortest path algorithm.

We assume that tt_r is the total time for provisioning request r and $ct_{r,d}$ is the time when the provisioning of request r is completed for node d where $tt_r = max(\{ct_{r,d} | \forall d \in D_r\})$. This total time is influenced by the amount of operations that need to be executed simultaneously. For each $l \in L$ we can have a set of $r = 0, ..., \infty$ active in a given time. We understand that the best way to divide the bandwidth between several operations is a fairness distribution between all active operations in a link at certain time, or $\exists r \in l_{r,t} | r_{bw,t} = l_{bw} \div len(l_{r,t})$. However, this behavior can generate a network sub-utilization, because each P_d can have distinct links and each link will generate a different value to $r_{bw,t}$. To solve this problem, we use a max-min fairness algorithm to fairly share all the links on the topology taking into account all r active on a given time t. More information about this implementation can be see on Section 3.6.1.

Finally, when provisioning lifecycle operations, if one node has more than one operation performed at the same time, it creates a queue based on the order in which the operations arrive, called n_{dque}. This queue works in a First-In-First-Out (FIFO) mode, where older operations are first attended. So, a large image can maintain the infrastructure busy for several seconds or even minutes, delaying the other operations. The validation of solutions that decreases the amount of time an operation r stays on this queue is one of the main objectives of this thesis.

3.5 Simulator Implementation

Aiming to validate the formal description presented before, we implemented a discrete-event simulator called ECOS (Edge Container Orchestration Simulator) using Python 3. Figure 3.2 presents the ECOS framework, showing the relationship between the several modules that compose the simulator framework.

The main element present in the framework is the *Simulation* module that implements the discrete-event model and glue together all the other ones. This module is also responsible for maintaining the simulation time and the looping until the end of the simulation. The next three are the *Scheduler, Infrastructure, and Application* modules, responsible respectively for managing the different scheduler algorithms, controlling the network sharing and resource management, and managing the status of the applications, replicas' instantiation, and version control.

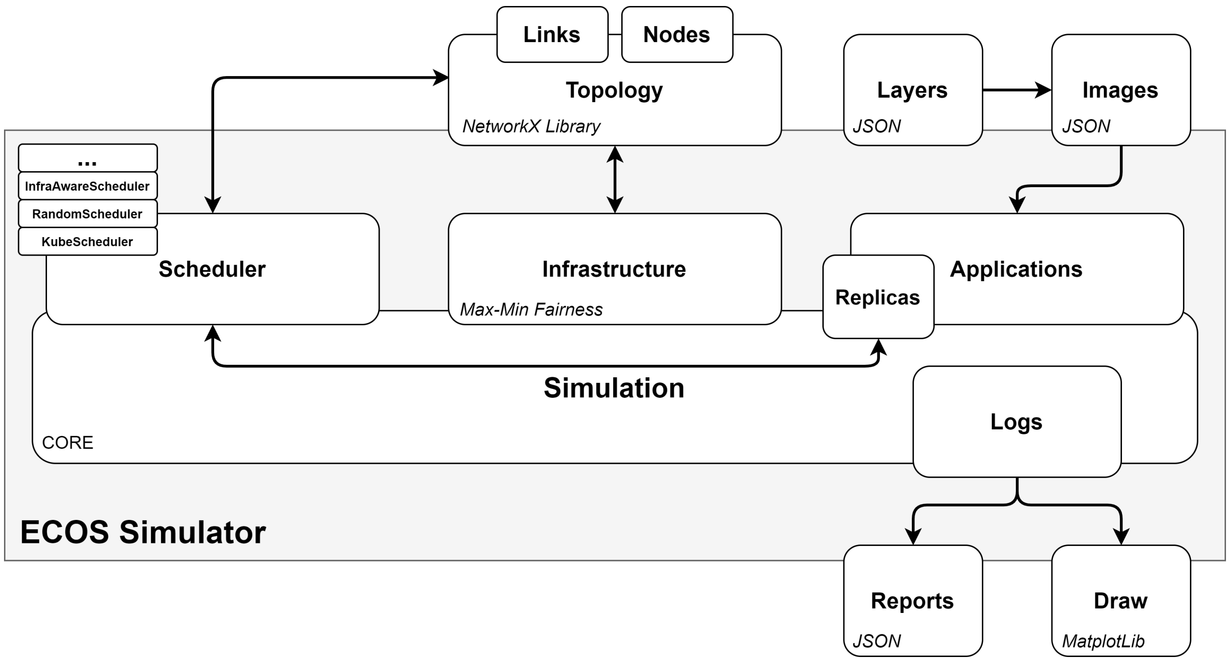


Figure 3.2: ECOS Framework

The module *Topology* uses the NetworkX library to implement the connected graph that represents the network topology and store the information that will be used either by the *Infrastructure* or the *Scheduler* modules. The *Images and Layers* are usually imported from an external source, like the DockerHub API, to the simulator and are used by the application to define the container that needs to be deployed. Finally, all modules generate logs that are stored in the *Logs Modules* and can be exported as reports or images using respectively, JSON and MatplotLib. In the following sections, each module from the framework is described in detail.

3.6 Infrastructure Module

The *Infrastructure* module has two main functions: first, simulate the network behavior regarding the links' sharing between the several flows deployed on the topology, and second, manage the node resources, like CPU, memory, and storage. The node, link configurations, and attributes are implemented as a directed graph via the NetworkX library to facilitate the implementation. We execute all management functions over this graph object, which is also used to calculate the best path between the nodes, using the bandwidth as weight, through the Dijkstra algorithm.

3.6.1 Modeling the Network Behavior

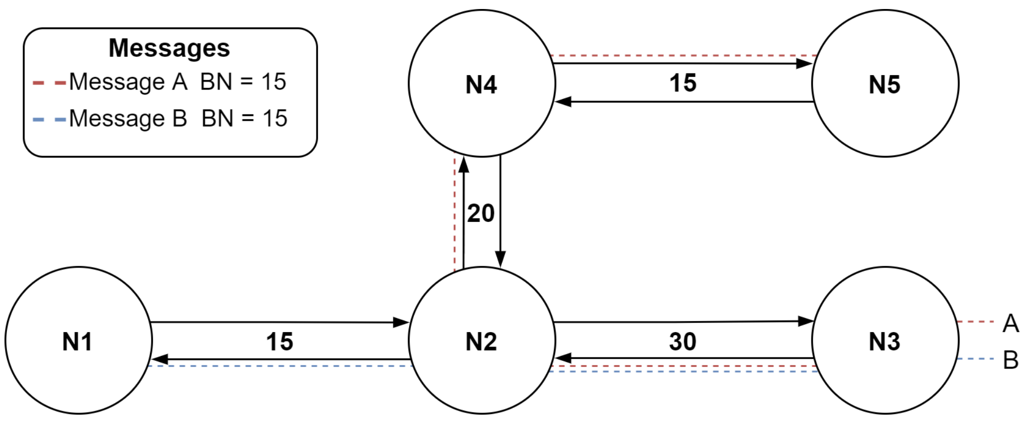
One of the main limitations of Edge Computing simulators is the network sharing between several flows with distinct sources and destinations. These communications can generate bottlenecks that are usually disregarded in cloud scenarios with large bandwidth and fast switching equipment. However, on Edge scenarios with long route paths and constrained resource connections, that is a question that cannot be set aside. So, we implement an option on the infrastructure to define the network sharing policy, an implement a default algorithm based on the Max-Min Fairness.

Max-min Fairness

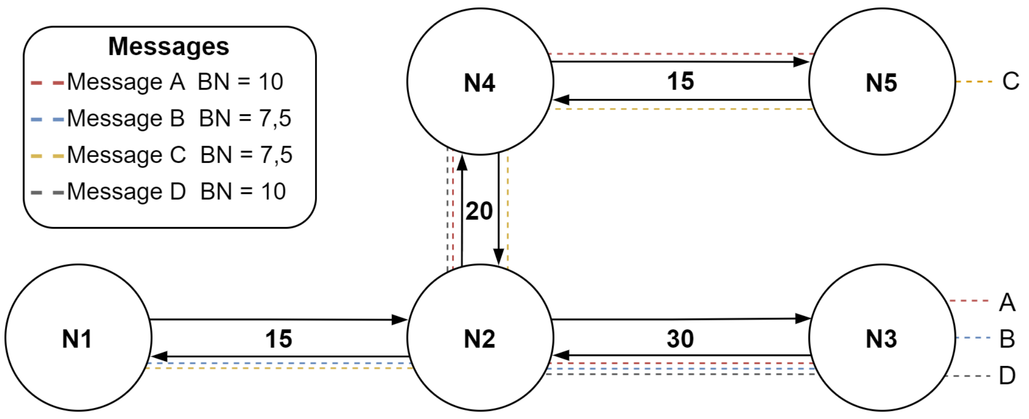
We implement a fair sharing schedule policy based on max-min fairness (MMF) as a default network sharing algorithm. The MMF control algorithm presents similarities with the TCP congestion control algorithm fairness and a good approximation with the normal network behavior[Bertsekas et al., 1992]. The main properties from MMF are: i) flows have the same priority over the available bandwidth, ii) flows get an equal share of the link bandwidth, iii) links are always using the maximum bandwidth possible based on the active flows. Figures 3.3a and 3.3b present two examples of the MMF model. The first one presents two flows, one from N3-N1 and the other from N3-N2. There are no bottlenecks on the network and each flow can use the total bandwidth for the small link in its path. In the second scenario, with four flows, two edges (N2-N1 and N2-N4) act as bottlenecks on the network, limiting the bandwidth use by edges N4-N5 and N3-N2.

Given a set of network links with respective bandwidth and a set of paths, it is possible to obtain the available bandwidth for each transmission in a given time using a progressive filling algorithm that respects the MMF model. In our implementation, we use a solution close to that presented in [Bertsekas et al., 1992]. The algorithm initializes the bandwidth available to each flow with 0. Then, it calculates the MMF to all interfaces with active flows and updates the bandwidth equally to each transmission until one link becomes saturated or the total amount of data to a given communication is satisfied. The saturated links serve as a bottleneck for all transfers using them, and the transmissions that do not need all the bandwidth available transfer the free space to the biggest ones. The algorithm executes until all links are saturated or all flows are satisfied.

We use the max-min fairness to set the max bandwidth available between events with different times on the simulation. To calculate the amount of time until the next event, we find the minimum value between three situations: next schedule application, minimum time until one download finishes, and minimum time until one deadline is defined. The pseudocode of our network sharing algorithm is shown in Algorithm 3.1.



(a) 2 Messages - No Bottleneck



(b) 4 Messages - 2 Bottlenecks

Figure 3.3: Examples of Max-min Fairness Flow Model

In lines 1-12, we prepare the data that will be used by the max-min fairness algorithm, from the topology graph. We select all the links that have active flows, the available bandwidth per link, and the number of flows per link. After this information is set, in line 13, we run the max-min fairness and set the array *R* with the available bandwidth per flow. Finally, in lines 14-25, we calculate the minimum time to the next event on the simulation and consolidate the transfer data on each flow based on the multiplication between the time and the bandwidth available per flow.

One of the main advantages of this approach is that the network sharing algorithm is decoupled from the simulation core and can be altered with a simple extension on the infrastructure module. Finally, although we use only the management flows as an entry on the simulation in the following chapters, the actual solution is capable of processing flows of different applications. It can also be used to simulate network noise or concurrent background traffic.

3.6.2 Modeling the Node Management

We also implemented a simple node management on the infrastructure module. Figure 3.4 presents the attributes that need to be managed on each node and link present on the Topology implemented on the NetworkX graph. Related to the nodes, we have three different nodes types: one generic representing the network abstractions like switches, routers, and autonomous systems in the middle of the network. This type of node is only used to determine the best route between the node and the registries and has no attribute, like CPU or memory.

Input : $G = (V, E)$: network graph; M: messages active on topology; mat: Next time that a new message need to be started
Output: Time spent until next event and update transfer data from each message

```
1  sA ← $\emptyset$
2  C ←
3  sP ← $\emptyset$
4  l ← $\emptyset$
5  for $m \in M$ do
6      for $v \in path(m)$ do
7          $a \leftarrow e \in E(u, w)_{(u,v) \in V | u=v \wedge w=next(v)}$
8          l.$append((m, a))$
9          if $a \notin sA$ then
10             sA.$append(a)$
11             $C_a = bandwidth(a)$
12         sP.$append(m)$
13 $\{R_m\}_{m \in M} \leftarrow MaxMin(sA, C, sP, l)$
14 $o \leftarrow \emptyset$
15 $d \leftarrow \emptyset$
16 for $m \in M$ do
17     $o_m \leftarrow dataToTransmit(m)/R_m$
18     $d_m \leftarrow deadline(m)$
19     $o.append(o_m)$
20     $d.append(d_m)$
21 $nft \leftarrow min(o)$
22 $mdt \leftarrow min(d)$
23 $time \leftarrow min(nft, mdt, mat)$
24 for $m \in M$ do
25     $transferData(m, R_m * time)$
26 return $time, M$
```

Algorithm 3.1: Network Simulation

The second one, called ContainerHost, is responsible for deploying the container working as the runtime machine on the containerization infrastructure. So, ContainerHost is the set of nodes available to the scheduler, and all the lifecycle operations happen on these nodes. The main attributes in the ContainerHost nodes are the CPU, memory, and storage, which are managed as a container with a given size. As the simulator has the mainly focus on the applications' orchestration phase, the resource's usage management is simplified. However, this module part is also decoupled from the simulator core and can be extended in the future. In this node type, the infrastructure also manages the network configurations like the registries, the number of simultaneous downloads and maximum download queue sizes, and the running application' replicas (or containers).

The last node type is called Registry and works as an abstraction of the registry server responsible for storing the images and layers before the ContainerHosts requests. All ContainerHosts need to have at least one Registry configured on their network configurations to download the images when required. These nodes only have the image and layers available as an attribute and, does not have storage, CPU, and memory limitations.

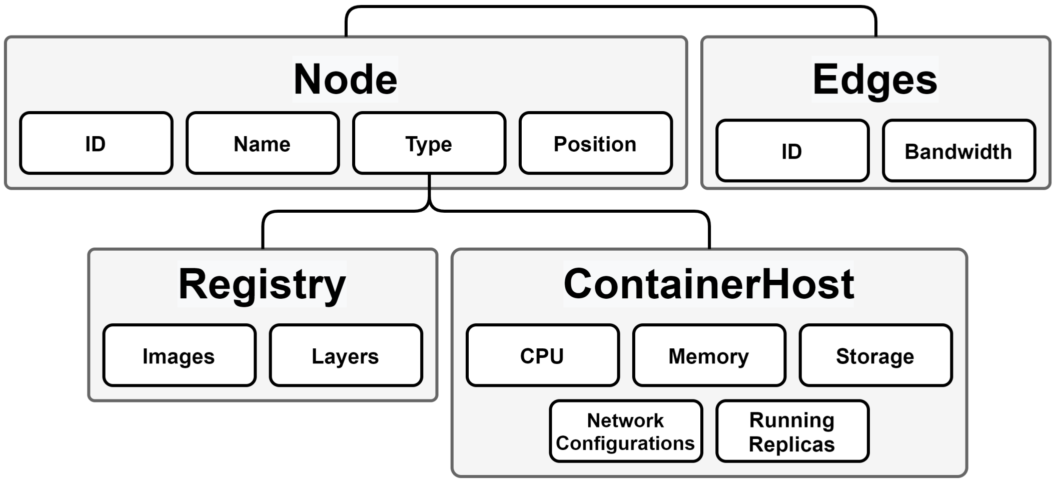


Figure 3.4: Node and Edge Attributes

Finally, the Link is the connection between the distinct type of nodes and has only attributed the bandwidth available to him. All calculations made by the network abstraction use this information to determine the amount of bandwidth available to each flow.

3.7 Image and Layer Abstraction

The container images and layers are implemented as an abstraction from the data available on the default Docker registry implementation. Figure 3.5 shows the relationship between the JSON image index and JSON image manifest pulled from the Docker Registry API, and the Image and Layer objects on the simulator implementation.

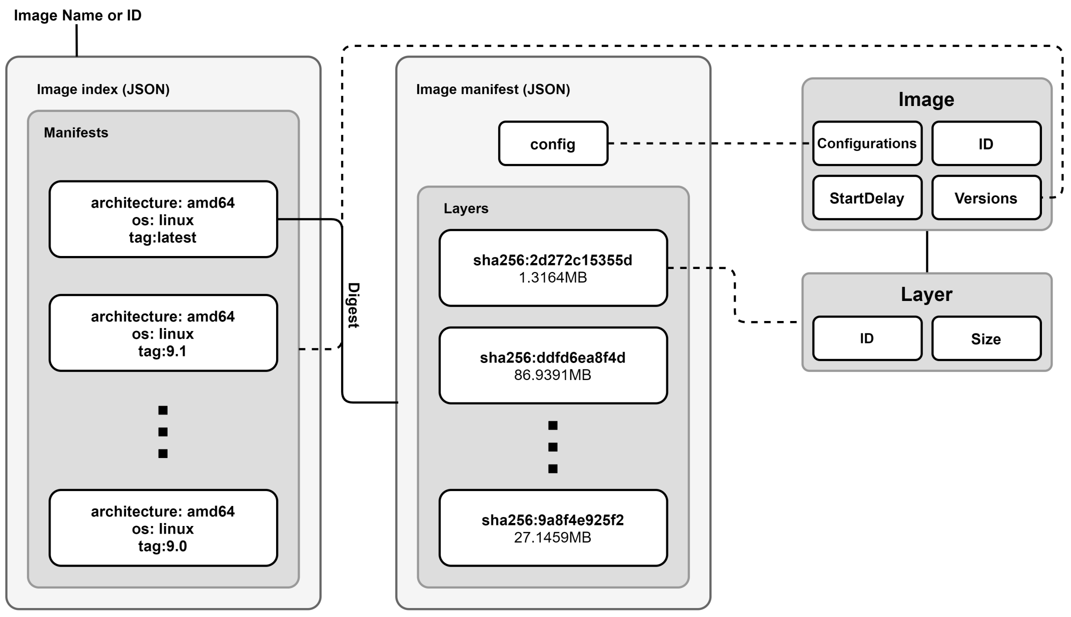


Figure 3.5: Relationship between the DockerHub API and the Image and Layer Abstraction

The first manifest pulled from the API contains the images available from each architecture, operational system, and tag from a given repository. We use this information to set all the versions available from a given image. After, we get all the images manifest available from the image index and use the latest tag to get the configuration options. We also set an estimate *startupDelay* for the application based on how much time is need to get the application running after completed download. All the layers presented on each image manifest are also created on the simulation, and we reuse the layers with the same hash ID.

3.8 Applications' Abstraction

As explained in Section 2.8.4, Kubernetes implements applications in several ways depending on the deployment needs. However, the default instantiation controller, called *Deployment*, is responsible for almost every application in a common cluster. The main advantages of using the *Deployment* controller are the rollback options, and the fast upgrade and downgrade on the number of replicas on stateless applications. For stateful applications, the default controller is the *StatefulSet*, which has the main difference in the persistent storage for each container, making rollback impossible.

However, both rely on the same steps from instantiation in a given node, and as a way to simplify our simulator, all applications present a single controller, called Application. The Application controller keeps the replica number recorded for a given application, the current image, and additional options like CPU and memory usage for each replica. As the simulator does not take care of the application after the deployment, so there is no practical difference between stateless and stateful applications.

Finally, all the applications are controlled by lifecycle operations, like deployment or update, described in the simulation configuration file. All lifecycle operations have a time used as an entry for the simulator, which means the moment when this action should be executed. One lifecycle example can be update a given application increasing the replica number by one. These operations acting as time controllers from the simulation, and when there are no more operations to be performed, the simulation is ended.

3.9 Modeling the Container Orchestration

In the default container orchestration model, one node, usually called "master", is responsible for storing all the worker nodes' information and maintaining the scheduler and applications logs. To simplify our implementation and take into account that we do not use latency between the communication as input in our simulation, we do not implement this node. All tasks that are usually running on the master are implemented through the infrastructure module. The description of the scheduler and the operation logs are described in the following sections.

3.9.1 Scheduler Implementation

The Scheduler module is responsible for implementing the algorithms used to place the containers on the infrastructure. We develop a generic scheduler interface, where the scheduler algorithm needs only to deliver for each container a given node where it can be deployed. This is the only pattern that the scheduler policy needs to follow, and the scheduler module has access to all the infrastructure and applications' data. It is important to note that the scheduler algorithm is always running over the replicas object and not on the application.

Kubernetes Scheduling Strategy

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

Kube-scheduler [Kubernetes, 2021b] 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. In the ECOS Simulator, we implemented all the predicates and priorities that are previously presented, based on the source code available on the Kube-scheduler GitHub Repository [Kubernetes, 2021a].

Other Strategies

We also already implemented other generic strategies, like a simple random and a worst-fit scheduler. As previously said, the main objective of this module is to create an interface for the development of new scheduler algorithms, so this module is highly customizable, and new options can also be added to the configuration file to provide fine-grained control to the simulation execution.

3.10 Simulation Input

To configure the simulator, we use a JSON file with all the objects and options we want to use as input. The Listing 3.1 presents the minimum configuration needed to run a simulated scenario. The file is divided into several arrays, each one gives information related to some module or object on the framework. Lines 2-11 present the nodes and links configuration, lines 12-29 show the image and layers options. Lines 30-39 present the applications managed during the simulation, with their respective lifecycle operations. Finally, the last lines (40-50) describe the infrastructure configuration, like the scheduler used, the log level, and the reports that will be generated at the end of the simulation.

Listing 3.1: Simulation Configuration File

```
1  {
2      "nodes": [
3          {"type":"netabs", "name":"switchA"},
4          {"type":"registry", "name":"registry", "position": [0, 0],
5          {"type":"chost", "name":"worker1", "registry": ["registry"], "position": [0, 0], "
               cpuCapacity": 8000,
6              "memoryCapacity": "32GB", "storageCapacity": "10GB", "simultaneousDownload":1}
7      ],
8      "links":[
9          {"bandwidth":"100Mb", "from":"switchA", "to":"registry", "bidirectional":true},
10         {"bandwidth":"100Mb", "from":"switchA", "to":"worker1", "bidirectional":true}
11     ],
12     "layers": [
13         {"size":"50MB", "digest":"1111"},
14         {"size":"50MB", "digest":"3333"},
15         {"size":"50MB", "digest":"2222"},
16         {"size":"50MB", "digest":"4444"}
17     ],
18     "images": [
19         {
20             "name":"postgres", "startDelay": 15,
21             "versions":[ {"digest":"6666",  "tag":"latest", "layers": ["2222","1111"]} ]
22
23         },
24         {
25             "name":"mysql", "startDelay": 15,
26             "versions":[ {"digest":"7777", "tag":"latest", "layers": ["3333","4444"]} ]
27
28         }
```

```
    ],
    "applications": [
        {"name": "postgres", "image": "postgres", "cpuRequired": 500, "memoryRequired": "256MB"},
        {"name": "mysql", "image": "mysql", "cpuRequired": 500, "memoryRequired": "256MB"},
    ],
    "lifecyleOperations": [
        {"application": "postgres", "name": "deployment", "type": "creation", "tag": "latest",
            "numReplicas": 1, "time": 0, "labels": { "sla": 20}},
        {"application": "mysql", "name": "deployment", "type": "creation", "tag": "latest",
            "numReplicas": 1, "time": 2, "labels": { "sla": 25}}
    ],
    "infrastructure": {
        "scheduler": {
            "name": "KubeScheduler",
            "config": {
                "predicates": [],
                "priorities": []
            }
        },
        "logLevel": "DEBUG",
        "log": [ "netstat" ]
    }
}
```

3.11 Simulation Logs

The *Logs* module is responsible for storing all information that needs to be kept until the end of the simulation. Each other module can generate a specific log object that can be configured as required. A generic log is generated during the simulation based on the log level specified on the configuration file. Lastly, the simulator uses these logs to create reports and graphs at the end of the simulation.

3.11.1 Reports and Graphs

There are two main ways to export information from the simulation to other applications. The first one is via the reports generation in JSON using the log generated by the simulation. Several reports are implemented on the simulator. Examples can be seen below:

1. **LinkUsage**: This report returns the usage of all links during the simulation. Each link has an array of tuples that store the time and the bandwidth usage every time a new value is set by the network sharing policy algorithm.
2. **CacheStatus**: This report returns the number of cache miss and cache hit by ContainerHost;

3. **RegistryUsage**: This report shows the number of times a registry was used to download a container;

4. **ContainerTimes**: This report presents the time usage to all steps on the deployment time, showing how much time the container takes to be download, on the download queue, and active;

The second option is to generate graphs using the matplotlib direct from the simulation. The primary usage of these options is the creation of network maps and flow visualization.

3.12 Closing Remarks

In this chapter, we presented ECOS, a simulator for edge infrastructures. ECOS focuses on orchestrating container applications and trying to meet several objectives that are not present on the other available simulators. We implemented a complete modular framework focused on three components: infrastructure, scheduler algorithm, and application lifecycle. The first enables the network and node sharing management, making possible fine-grained control over the topology, easing to find bottlenecks and resource limitation on nodes. The second creates an extended interface that allows the experimentation of a new scheduler solution using any information available on the topology. The last one is used to validate new components on the orchestration infrastructure, placement, and network cost. In the following chapters, ECOS is used to validate new solutions to decrease the deployment latency on edge container infrastructure.

4. REGISTRY PLACEMENT TO SPEED UP APPLICATION DEPLOYMENT

4.1 Introduction

It is common sense that current container orchestration solutions do not present an optimal method for distributing the applications on Edge Computing. Hence, several works aim at optimizing it by reducing the amount of time needed to instantiate, or diminishing the load generated on the infrastructure through better usage of the nodes based on peer-to-peer communication [Nathan et al., 2017], distributed caches [Darrous et al., 2019] or new placement algorithms [Faticanti et al., 2019]. However, these solutions usually propose many changes in the current orchestration frameworks that are not so easy to achieve in a multi-tenant infrastructure with several distinct actors. Moreover, no one of them considers the importance of the registry placement in this distribution and the load that it generates on the infrastructure.

We present a novel approach to strive the deployment latency generated on edge infrastructures. The proposed approach can be used without significant alterations in the topology or container orchestration. To achieve that, we focus on a specific component from the container architecture, i.e., the registry nodes, which are responsible for storing the images used to deploy the containers on worker nodes. Further, we study how the registries placement on edge infrastructure can affect the network load and generate bottlenecks. Given the NP-hardness of the registries placement problem, we propose a heuristic solution based on fluid communities to find the best place to set a *k* number of registry nodes. To evaluate our solution's effectiveness, we carried a series of experiments on realistic and random networks with distinct container schedulers.

This chapter want to answer the main research question:

- *How the registry placement influence the deployment latency? How can we distribute the network load between several registries on the topology?*

This chapter is based on our previously published paper "Luis Augusto Dias Knob, Francescomaria Faticanti, Tiago Ferreto, Domenico Siracusa. *Community-based placement of registries to speedup application deployment on Edge Computing*. 9th IEEE International Conference on Cloud Engineering (IC2E 2021)".

4.2 Related Work

Current solutions to decrease the container deployment latency are distributed on distinct areas, mainly with large modifications on the container runtime [Ahmed and Pierre, 2018] or new services that need to be added to the topology [Harter et al., 2016].

Some approaches propose improvements on how the containers are stored in or distributed to nodes. Pulling the same image by multiple registries simultaneously is presented in [Nathan et al., 2017], which also discusses a cooperative implementation of a set of registries. Solutions based on peer-to-peer communication are also presented in [Uber, 2021] and [Kangjin et al., 2017]. However, these implementations rely on powerful worker nodes placed on high-speed networks, usually not replicable on edge scenarios, with geo-distributed topologies. Furthermore, adding a new daemon on a resource-constrained node can generate bottlenecks, as shown in [Ahmed and Pierre, 2020], where even simple applications running on it can generate high latency in the deployment of new applications.

The use of distributed file systems to share images between several nodes is presented with slight changes in [Littley et al., 2019], and [Zheng et al., 2018]. In both papers, the nodes share a given folder that acts as a centralized cache for the images. This implementation usually shows a significant improvement on the total amount of time needed to instantiate a new container, together with a small redundancy on the layers, since all the shared images are stored in the same place. However, that solution also relies on high-speed networks, with negligible downtime and near-allocated nodes.

There are also proposals of new placement algorithms focusing on geo-distribution [Rossi et al., 2020], and the sharing of microservices between cloud and edge [Faticanti et al., 2019] to decrease the deployment time, using latency or bandwidth as input to define the application's placement. Nevertheless, these works' primary objective is to improve the number of containers that can stay active in the topology. However, no one considers the deployment latency on the topology and how this latency can cause congestion on the network and affect the placement results.

In [Darrous et al., 2019], the authors discuss that in edge computing, due to the limited node storage, it is impossible to have many images locally stored. Hence, to improve the deployment latency, it is necessary to retrieve the images on the topology faster. Therefore, they propose a sharing algorithm called KCBP, which finds the closest node with a given layer and uses it to forward it to the requester. Although this chapter is the first to discuss the positioning of the layers on the topology and its impact on the deployment latency, its implementation has several limitations. Some of them include the necessity of direct connections between nodes and assumptions that are not feasible in the real world, such as nodes sharing images.

Symbol	*Meaning*
$G = (V, E)$	network graph
$B_{u,v}$	available bandwidth on link $(u, v) \in E$
$\mathcal{K}$	set of communities on graph G
$K = \|\mathcal{K}\|$	number of communities in the graph
$\{V_k\}_{k\in\mathcal{K}}$	partition of graph G identified by the communities

Table 4.1: Main notation used throughout the chapter

Finally, changes on how the Docker runtime download and start new images are also proposed in [Ahmed and Pierre, 2018], where the authors suggest a set of tweaks to improve the several steps executed on the deployment focusing on resource constrained nodes. However no one of the works consider the impact of the application deployment in large-scale distributed topologies, exploring the possible bottlenecks that this network load can generate in the infrastructure. So, it is, to the best of our knowledge, the first attempt to tackle the challenges of registries placement in an edge computing environment using infrastructure's constraints, seeking to decrease the total amount of time needed, namely the deployment latency, to fully implement these services on the topology.

4.3 Problem Formulation

4.3.1 Registries Placement Problem

This chapter's main objective is to design an efficient algorithmic solution for the placement of container registries among the network nodes to speed up the download time in each node from the placed registries. The main objective is represented by the minimization of the maximum download time from the registries to each node of the network. When trying to solve this problem, usually we may think to a *vertex k-center problem*[Hsu and Nemhauser, 1979], where a set of facilities must be deployed on a complete graph in order to minimize the maximum distance between each node and its closest facility. In our case, the concept of distance can be viewed as the download time from a container registry to each node in the network. However, there are some differences concerning the problem we need to solve. The first is the type of graph we are dealing with. Indeed, the vertex k-center requires a complete graph, whilst, in our case, we do not always have such a case. The second main difference is represented by the computation of the download time between each node and the facilities placed on the graph. In fact, this time is not easily computable since it depends on the placement of facilities and the bandwidth occupation of each link leading to a combinatorial explosion of possible configurations.

For this reason, we tackle the problem from a different perspective. We aim to partition the network graph placing a container registry in each subset (community) of the partition. The graph partitioning should be performed in such a way that the total bandwidth of the graph is fairly balanced among the partition. The graph partitioning and the bandwidth load balancing should be performed in order to: i) speed up the download time for each node inside each community and ii) avoid bottlenecks in the network, such as putting containers registries in a few and close nodes of the network. Hence, the main question that we want to answer in this problem is the following: *where can we place container registries in order to have load-balanced distribution, in terms of bandwidth, of containers among the network's nodes?*

In order to tackle the problem described above, we assume that the desired number of communities in the graph is given as input of our problem. In order to have a load-balanced solution in terms of bandwidth, we use the concept of standard deviation between the total bandwidth of each community. In Figure 4.1 we present the notation used throughout the chapter. Above, we provide a formal description of the problem.

4.3.2 System Model

We consider a network infrastructure consisting of a set of nodes V and a set of edges E. Therefore, we represent the network topology as a weighted graph $G = (V, E)$, where $E \subseteq V \times V$. Each edge $(u, v) \in E$ of the graph is characterized by the bandwidth available on the link, namely $B_{u,v}$. Further, we indicate a partition of the graph with $\{V_k\}_{k\in\mathcal{K}}$, i.e., a family of subsets of V where $V_k \subseteq V, \quad \forall k \in \mathcal{K}$, and $V_k \cap V_{k'} = \emptyset, \quad \forall k \neq k'$. For the sake of a clear explanation of the algorithmic solution, we call these subsets *communities*. The cardinality of $\mathcal{K}$, $|\mathcal{K}| = K$, also indicates the number of desired container registries to be deployed on the network as the algorithm will place a container registry for each community of the input graph.

Variables

We introduce the following decision variables for each node of the network:

$$x_{v,k} = \begin{cases} 1, & \text{if node } v \text{ is placed in community } k \\ 0, & \text{otherwise,} \end{cases}$$

$\forall v \in V$.

Objective

The objective function is represented by the standard deviation of the total bandwidth available in each community (Equations 4.1 - 4.3):

$$\sqrt{\frac{\sum_{k\in\mathcal{K}}(\alpha_k - \bar{\alpha})}{K-1}}, \tag{4.1}$$

where

$$\alpha_k = \sum_{(u,v)|u,v\in V_k} B_{u,v} \quad \forall k \in \mathcal{K}, \tag{4.2}$$

and

$$\bar{\alpha} = \frac{\sum_{k\in\mathcal{K}} \alpha_k}{K}. \tag{4.3}$$

Summarizing, the problem we aim to solve is the following

$$\text{minimize} \quad \sqrt{\frac{\sum_{k\in\mathcal{K}}(\alpha_k - \bar{\alpha})}{K-1}} \tag{4.4}$$

subject to:

$$\alpha_k = \sum_{(u,v)\in E} B_{u,v}\, x_{u,k}\, x_{v,k}, \quad \forall k \in \mathcal{K}, \tag{4.5}$$

$$\alpha_k > 0, \quad \forall k \in \mathcal{K}, \tag{4.6}$$

$$\sum_{k\in\mathcal{K}} x_{u,k} = 1, \quad \forall u \in V, \tag{4.7}$$

$$x_{u,k} \in \{0,1\}, \quad \forall u \in V, \forall k \in \mathcal{K}, \tag{4.8}$$

where constraint (4.6) ensures that each community has at least two adjacent nodes, i.e., the total bandwidth inside the community is greater than zero. Constraint (4.7) imposes that each node of the network is assigned to exactly one community.

Computational Complexity

Looking at the minimization presents in the Equation 4.4, it is easy to see that our problem falls under the class of graph partitioning problems [Andreev and Racke, 2004]. These particular problems are typically NP-hard requiring approximate solutions to be solved. Furthermore, defining the optimal number of communities to cover all the network graph adds another complexity dimension to the problem. The investigation of the optimal number of communities is left for future works. So, here we want to define, given a number of registries, the best placement to each one of them.

4.4 Algorithmic Solution

Given the NP-hardness of the problem described in the previous section, we propose a heuristic method based on general search algorithms such as Hill-Climbing [Russell and Norvig, 2002].

For the graph partition problem, we resort on the fluid communities algorithm [Parés et al., 2017], namely `FluidC`. The algorithm takes as input a graph $G = (V, E)$ and the desired number K of communities. Initially, it randomly selects K vertices from V. These single vertices form the initial K communities. Each community has a density $\delta \in (0, 1]$. For each community k, its density is given by

$$\delta(k) = \frac{1}{|v \in V_k|}.$$

At each step of the algorithm, the community of each vertex is updated and when the assigned communities to vertices do not change for two consecutive steps, the procedure terminates. The update rule for each vertex v computes the community with maximum total density within the neighbourhood of v (including v as well). If more than one community is found for a vertex v, then a random community is chosen within the candidate ones [Parés et al., 2017].

As shown in [Parés et al., 2017], the main advantage of the `FluidC` algorithm is the scalability. Indeed, this algorithm provides good communities for large graph in reasonable time. However, this kind of solution is not thought to work with weighted graphs. For this reason, we perform a Hill-Climbing search to find the partition of the graph that has the best standard deviation of the total bandwidth available on each community. The pseudocode of our main algorithm is shown in Algorithm 4.1.

In the first part of Algorithm 4.1, lines 7-17, the `FluidC` algorithm is repeatedly applied to the input graph until no better communities are found in terms of bandwidth available on each single community. This part follows the Hill-Climbing search approach where we always move towards a better configuration until a termination condition is met. In the second part of the algorithm, lines 18-25, for each community computed in the previous step, a particular node for the registry placement is selected. In this case, the algorithm selects the node with the highest *eigenvector centrality* degree [Latora et al., 2017]. This kind of measure provides, for each node, an indication about node centrality and connectivity towards nodes with high centrality degree within the same community.

Input : $G = (V, E)$: network graph; k: desired number of communities
Output: Set of nodes where to place container registries

```
place ← $\emptyset$
final_comm ← $\emptyset$
sd ← $+\infty$
it ← 0
while $True$ do
    $\bar{b} \leftarrow [\,]$
    $\{V_k\}_{k\in\mathcal{K}} \leftarrow$ FluidC($G, K$)
    for $k \in \mathcal{K}$ do
        $B_k \leftarrow \sum_{(u,v)|u,v\in V_k} B_{u,v}$
        $\bar{b}$.append($B_k$)
    if $stdv(\bar{b}) < sd$ then
        final_comm ← $\{V_k\}_{k\in\mathcal{K}}$
        it ← 0
    else
        it ← it + 1;
    if $it \geq 100$ then
        break
for $k \in final\_comm$ do
    $\bar{c} \leftarrow [\,]$
    $G_k \leftarrow (V_k, \{(u, v) \in E | u, v \in V\})$
    for $v \in V_k$ do
        $c_v \leftarrow$ eigenvector_centrality($G_k$)
        $\bar{c}$.append($c_v$)
    $u \leftarrow$ sort($\bar{c}$).pop()
    place ← place $\cup \{u\}$
return $place$
```

Algorithm 4.1: FluidC-Placement

Time Complexity

The complexity of the algorithm is mainly dominated by the number of iterations required to reach the last local maximum from the starting point (as it can be noticed from Algorithm 4.1, if after 100 iterations no better move is performed the *while* loop is terminated) in the first part. However, all the operations inside the *while* loop are polynomially bounded. Indeed, the `FluidC` algorithm presents a linear cost equal to $O(E)$ [Parés et al., 2017]. In the second part of the algorithm, to compute the eigenvector centrality, it is sufficient to solve a linear system of equations of the size of subgraph G_k. Finally, the `sort` operation has a cost of $O(|V| \log(|V|))$ in the worst case. Hence, assuming a constant number of iterations required to reach a local maximum, Algorithm 4.1 presents a polynomially bounded time complexity. This confirms the scalability of the proposed algorithm.

4.5 Evaluation

4.5.1 Simulator

To evaluate the `FluidC` algorithm, we use the ECOS simulator presented on the Chapter 3. Before starting the simulation, first we run the registry placement algorithm (Random or the FluidC) and calculate the best path between the nodes and all the registry node, or the registries if more than one is configured. After choosing the closest registry, all the container downloads occurs from the same Registry Node.

4.5.2 Simulation Scenarios

To perform the simulations, we used two topologies. The first one is a random topology with 100 nodes using the Erdös - Rényi model [Erdos, 1961]. The topology is fully connected, and each node has a downlink and uplink with the same bandwidth (1 Gbps). This topology is also highly connected with 242 links, and each registry added in the topology has bandwidth available 10 times bigger than the worker nodes (10 Gbps). This guarantees that registries do not represent a bottleneck for the simulation.

We also validated the placement algorithm with the Italian education and research network (GARR) topology. The GARR topology is composed by 70 operational zones and 112 links and covers all the Italian territory. The topology is presented in Figure 4.1. We used the GARR topology as an isolated network, and we assumed that each operational zone is a worker node available to deploy a new application. Each operational zone can

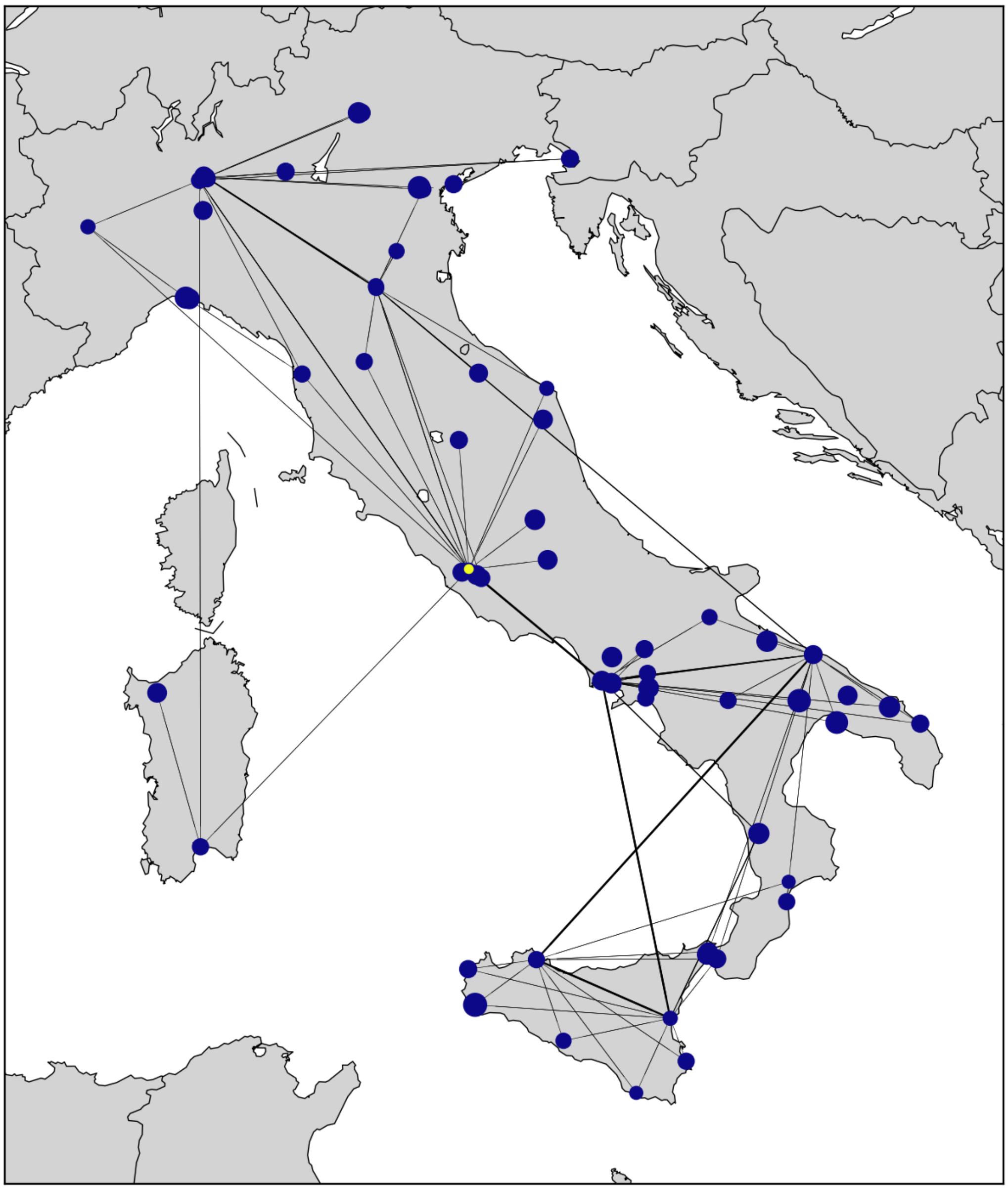


Figure 4.1: GARR Network Topology

also be used to deploy a registry using the `FluidC` algorithm. We also set the bandwidth available to the registries node as 10 Gbps. Besides, we limited the total capacity used by the communication between the nodes to 10% of the total bandwidth described in [Consortium-GARR, 2021], and even with that limitation, the average size to each link is close to 2Gbps.

Each node has a cache of fixed size that reduces the amount of data that needs to be transmitted. As we want to verify the impact of a large amount of containers deployed on the topology, we set a cache size of 600MB in all the nodes in all the simulations. For the sake of simplicity, each container image has only one layer, and all the images have the same size (200 MB). In these scenarios, the impact caused by an application on a node is not considered, so we used a small number of specific applications (13). With that, the cache corresponds to 23% of the total (2.6GB). The policy usage to replace the current images on cache is the *LeastRecentlyUsed*. Whenever a new container download is requested, if the image is already in the cache, the application starts instantaneously, and the cache is updated.

One of the limitations of our simulation is that the cache takes into account only the download phase, regardless of whether the application continues to run on the node or not. The parameters used for all the simulations are summarized in Table 4.2.

Parameter	*Value*
Number of distinct containers	13
Size of each container	200 MB
Cache size	600 MB
Cache policy	LRU
Worker node bandwidth	1 Gbps
Registry node bandwidth	10 Gbps

Table 4.2: Simulation parameters

4.5.3 Application Scheduler

To understand the impact of a high density of applications on the topology, we use two distinct schedulers to instantiate an application on the edge nodes. In both cases, we run the scheduler by one simulated hour (3600 seconds) with an average number of requests close to 5 operations per second.

Each deployment may have a deadline time, that is, the instant wherein the application no longer needs the container. By default, the smallest deadline used in our simulation was 25 seconds. If this value is not present, we understand that this container will run until the end of the simulation. If the simulation time is longer than the deadline to a given container, it returns as an error to the simulation, and the container is counted as non-started.

The placement algorithms selected to choose each container's destination node are: Random and the PESS Scheduler presented in [Doriguzzi-Corin et al., 2020]. The Random scheduler tries to show a non-bias distribution on the topology with a fair amount of containers between all worker nodes. With the PESS Scheduler, we want to validate the same algorithm with a more realistic scenario focused on the application placement at the Edge, improving the node utilization. The PESS Scheduler, was initially developed to Security Function Chains, where based in a heuristic solution, takes into account security and QoS requirements of user applications, while ensuring that computing and network resources are accurately utilised. However, in our scenario we use this scheduler to provide a series of chained-applications acting like microservices from an edge application.

4.5.4 Experimental Results

We executed three simulations. The first one was the Random Topology with a Random Scheduler. Then, we run both PESS and Random Scheduler with the GARR Topology. In all cases, we executed our `FluidC`-based algorithm in contrast with a random selection of nodes to distribute 1, 2, 4, and 8 registries on the network, and on the GARR Topology scenarios we used the same placement results to both schedulers. This number of registries was manually defined to understand the impact of an increasing number of registries on several points, such as the total deployment time, the distribution of requests among distinct registries, and the bottlenecks generated on the network. When the simulation runs with more than one registry, each worker node chooses the registry with the shortest path based on the available bandwidth. If the shortest path return a draw between several registries, it randomly chooses one.

Random Topology

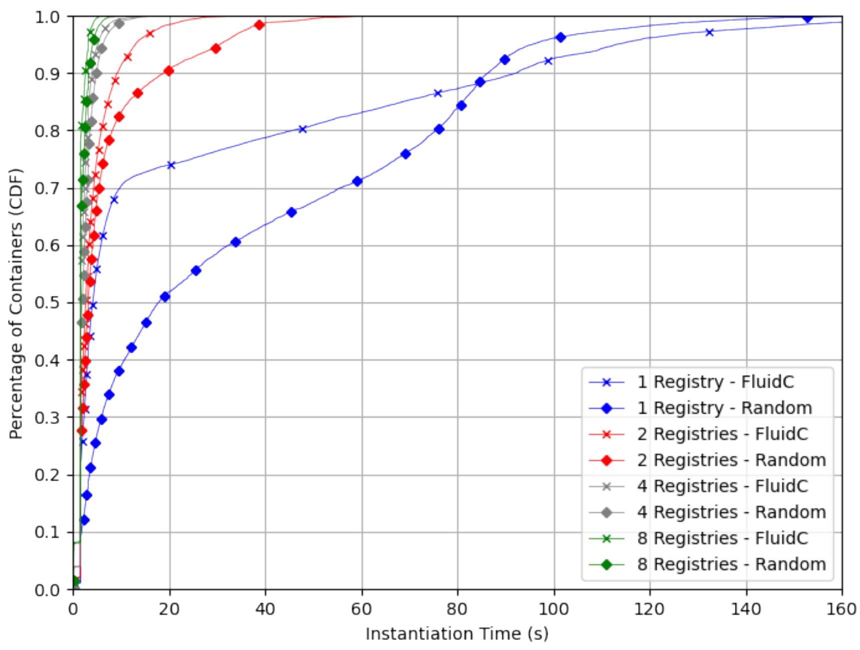


Figure 4.2: Random Topology - Random Scheduler

In Figure 4.2 we have depicted the CDF of the container deployment latency according to the Random Scheduler. As expected, when the number of registries on the topology increases in both cases, random or `FluidC` algorithm, we have an improvement on the

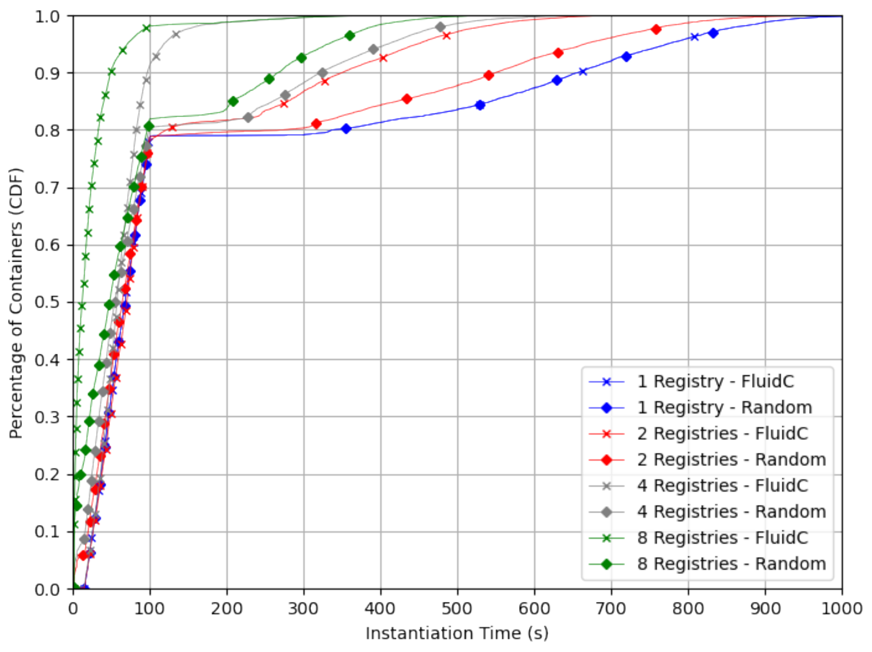


Figure 4.3: GARR Topology - Random Scheduler

deployment latency, and the difference between them decreases as we increase the number of registries that are located on the network. However, the `FluidC` algorithm presents a deployment latency of 1.59 and 1.55 times smaller in scenarios with 1 or 2 registries and 12.5% in the eight registries scenario. As the Random Topology is highly connected, in a scenario with more than four registries, more than 90% of the containers start with less than 10 seconds. However, it is important to note that we used all the available bandwidth to deploy the containers, which is not feasible in real infrastructures.

We also summarize the simulation results in Table 4.3, showing that in this scenario, with four or more registries, the number of non-started containers is zero, and `FluidC` has slightly better results with two or one registries. Finally, we can see that the high number of connections on the random topology enables a good distribution of the deployment requests between the registries with the `FluidC` performing better in all number of registries.

GARR Topology

In Figure 4.3 and Figure 4.4, we present the results from the simulations made on the GARR Topology. The first one used the random scheduler to distribute the containers. We can see that the faster 80% requests are deployed in less than 100 seconds in all scenarios until four registries, in a pretty similar distribution. However, the network generates specific bottlenecks as we place the registries with both placement heuristics, showing that the 20% slower requests have better results consistently with the `FluidC` in contrast with the random placement.

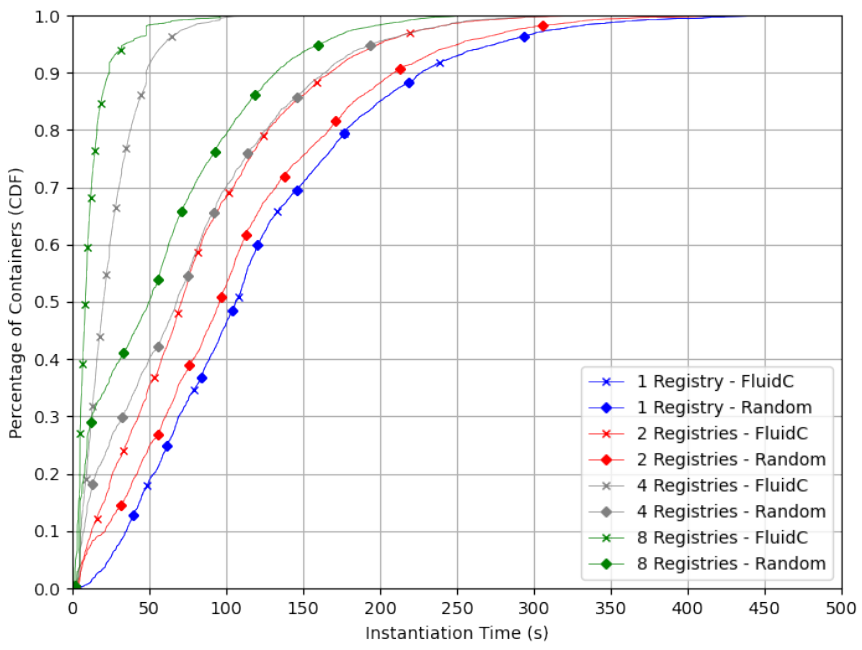


Figure 4.4: GARR Topology - PESS Scheduler

Furthermore, with four and eight registries, the `FluidC` almost mitigates the bottlenecks on the network, showing a decrease of 58% and 78%, respectively, in deployment time. At last, with one registry, both placements present the same results. In these cases, the bottleneck is the connection between the node where the registry is connected with the rest of the network.

The PESS Scheduler tends to schedule the application on the center of the network with the largest bandwidth connection possible. In fact, in our simulation, 67% of all containers are scheduled in only 6 worker nodes. Even with that small distribution on the network core, as shown in the Figure 4.4, `FluidC` presents consistently better results than the random scheduler, that besides the scenario with one registry, the deployment time was 24%, 65%, and 80% smaller using 2, 4, and 8 registries.

Finally, from these figures, we can notice our placement algorithm's independence with respect to the scheduling algorithm. Indeed, we can notice that the curves of `FluidC` related to 4 and 8 registries in both cases (Random Scheduler and PESS Scheduler) are pretty similar. This confirms that our container registries placement algorithm is agnostic to the specific scheduler used to distribute applications among the network. This confirms the possibility of improving the applications' instantiation time without modifying the orchestration mechanisms of the applications.

We summarize both experiments in Table 4.3, presenting results similar to the random topology, showing the decrease of non-started containers and a shorter average deployment time in every scenario using `FluidC`. We can highlight the cases with 4 and 8 registries that have in average 32% better results with the random scheduler, and 71% with the PESS scheduler.

4.6 Closing Remarks

In this chapter, we present a novel deployment community-based placement to distribute container registries on an edge topology. Our solution optimizes the registries' distribution on the topology like communities in a relation graph. To do that, we implement a two-phase algorithm that first generates a set of communities based on the fluid communities algorithm and then chooses, in each community, the most central node that will be used as the host for the new registry. We validated the solution with a series of simulations using two distinct topologies, and a random and realistic application scheduler, showing enhanced performance in both cases. The total instantiation time was optimized in the best case in more than 70%, and the small number of non-started containers can be noted even with the best placement of just two registries.

Scenario			*Number of Deployments*	*StdDev Req. Registries (un.)*	*Non-started Containers (%)*	*Average Deployment Time (s)*
Random Topology Random Scheduler	Random	1 Registry	19790	-	19.37	21.43
		2 Registries		3583.6	1.11	5.14
		4 Registries		1088.45	0.00	2.09
		8 Registries		1371.54	0.00	1.53
	FluidC	1 Registry		-	13.26	13.46
		2 Registries		929.1	0.12	3.31
		4 Registries		1065.86	0.00	1.81
		8 Registries		385.3	0.00	1.34
GARR Topology Random Scheduler	Random	1 Registry	19649	-	47.86	273.62
		2 Registries		10820.9	44.64	207.8
		4 Registries		5694.0	38.09	106.72
		8 Registries		3882.0	33.42	75.65
	FluidC	1 Registry		-	47.86	273.62
		2 Registries		5831.5	44.35	132.72
		4 Registries		857.3	31.34	45.42
		8 Registries		897.23	7.93	15.89
GARR Topology PESS Scheduler	Random	1 Registry	19129	-	40.52	75.15
		2 Registries		11857.5	35.23	64.93
		4 Registries		7046.5	24.29	46.06
		8 Registries		4835.7	17.61	38.25
	FluidC	1 Registry		-	40.52	75.15
		2 Registries		8556.7	24.95	49.91
		4 Registries		3307.9	3.18	16.34
		8 Registries		2111.8	0.87	7.76

Table 4.3: Additional statistics from the simulations

5. CONTAINER SCHEDULING BASED ON NETWORK BANDWIDTH AVAILABILITY

5.1 Introduction

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 chapter 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 [Fu et al., 2020] 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 [Docker, 2021].

Besides that, we want so answer the following research question:

- *Does the scheduler uses any network information as input to schedule applications? Is it possible to implement a solution that uses the available bandwidth as input to the application schedule? How this impacts the other's priorities?*

This chapter is based on our previously published paper "Luis Augusto Dias Knob, Carlos Henrique Kayser, Tiago Ferreto. *Improving Container Deployment in Edge Computing Using the Infrastructure Aware Scheduling Algorithm*. 26th IEEE Symposium on Computers and Communications (ISCC 2021)."

5.2 Container Schedule Strategies

The default Kubernetes scheduling strategy, called Kube-scheduler, has a generic and modular implementation that can be use in several distinct cloud topologies. However, in an edge computing scenario composed of heterogeneous devices, these policies may not be enough to deploy container-based applications. For instance, they do not consider potentially scarce resources at the edge, such as network bandwidth, to meet the requirements of applications without compromising their quality of service. In this section we describe new strategies proposed in the literature to schedule edge applications.

5.2.1 Dependency Aware Strategy

Fu et al. [Fu et al., 2020] propose 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 propose 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 gives to 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 already allocated 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.

5.2.2 Others Edge Schedulers

Other schedulers were proposed based on distinct objectives that can also affect an edge infrastructure. For example, in [Santos et al., 2019] the authors introduce 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 [Faticanti et al., 2019], 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 increase the total amount of time needed to deploy the applications.

5.3 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 5.2, 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 5.1 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 sets 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).

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  $aL \leftarrow \{\}$
2  for $layer \in application_{image}$ do
3      $aL_{digest} \leftarrow size(layer)$
4  $ttInst \leftarrow \{\}$
5  for $c \in chosts$ do
6      $mL \leftarrow \{\}$
7      for $app \in c_{scheduleApps}$ do
8          for $layer \in app_{layers}$ do
9              $mL_{app} \leftarrow size(layer)$
10     $cL \leftarrow \{\}$
11     for $image \in c_{cache}$ do
12         for $layer \in image_{layers}$ do
13             $cL_{image} \leftarrow size(layer)$
14     $s \leftarrow \sum_{i \in (aL \setminus mL \setminus cL)} i$
15     $ttInst_c \leftarrow s \div bandwidth(c)$
16 $minTime \leftarrow \min_{\forall t \in ttInst} t_{time}$
17 $maxTime \leftarrow \max_{\forall t \in ttInst} t_{time}$
18 if $minTime = maxtime$ then
19     for $c \in chosts$ do
20         $c_{score} += w$
21 else
22     for $c \in chosts$ do
23         $score \leftarrow (w - ((w - 0) \div (maxTime - minTime) \times (ttInst_c - minTime))$
24         $c_{score} += score$
25 return chosts
```

Algorithm 5.1: Least Congested Node Priority

5.4 Evaluation

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 does not 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.

This section presents an evaluation of the Infrastructure Aware scheduling algorithm. The algorithm is compared to the Kube-scheduler, and the algorithms presented in [Fu et al., 2020] (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, cache hit, and application distribution between the nodes.

5.4.1 Simulator

In order to perform the evaluation, we use the ECOS simulator presented in Chapter 3. This experiment's main focus is the deployment process and image distribution from a registry to the edge nodes considering network topology behavior. This approach, based on the max-min fairness, results in a more realistic simulation considering the network bottlenecks when provisioning containers in edge nodes.

The ECOS 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 official implementation [Kubernetes, 2021a], where we can add and remove predicates, priorities, and set 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. We also implemented the priority Infrastructure-Aware, that can be activate or deactivate as any other default priority on the Kube-scheduler.

5.4.2 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, including all the points of presence (PoPs), network connections, link' speeds and latency. This topology interconnects all Brazilian universities and research institutes through 28 Points of Presence (PoPs) distributed over the country (Figure 5.1). 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 [RNP, 2021b].

To enable the comparison with the other scheduling algorithms (Kube-scheduler, Image and Layer locality [Fu et al., 2020]), 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 [RNP, 2021a]. It was given a bandwidth of 10Gbps to avoid having the Registry Node as a bottleneck on the simulation.

5.4.3 Workload

The workload used in the simulation is based on real images on the Docker Hub [Docker, 2021]. 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).

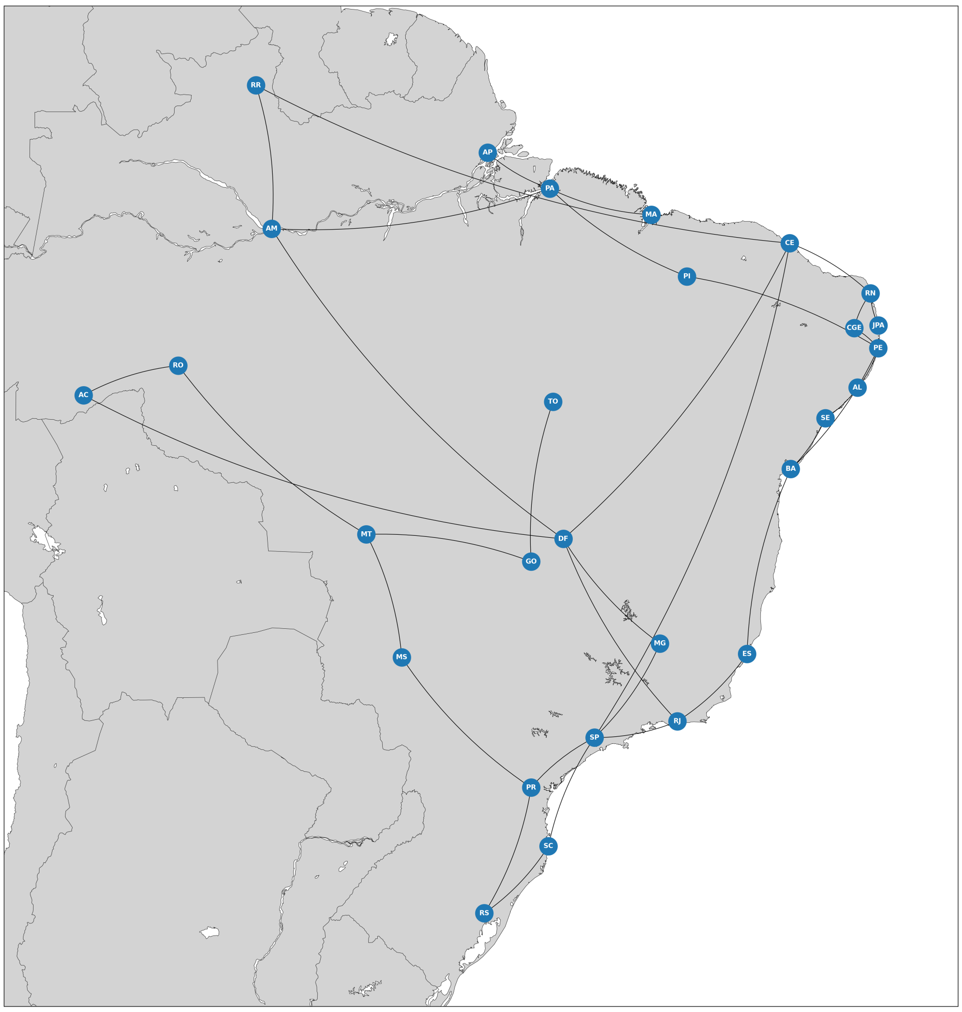


Figure 5.1: Ipê Brazilian Research Network Topology

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) is configured. 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 5.1 presents the parameters used in the experiments.

5.4.4 Results

Deployment Latency

Figure 5.2 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

Parameter	***Value***
Server Nodes	*28*
Server Links	*100 Mbps*
Edge Nodes	*140*
Edge Links	*10 - 60 Mbps*
Number of Images	*24*
Number of Applications	*350*
Number of Replicas	*1250*

Table 5.1: Simulation parameters

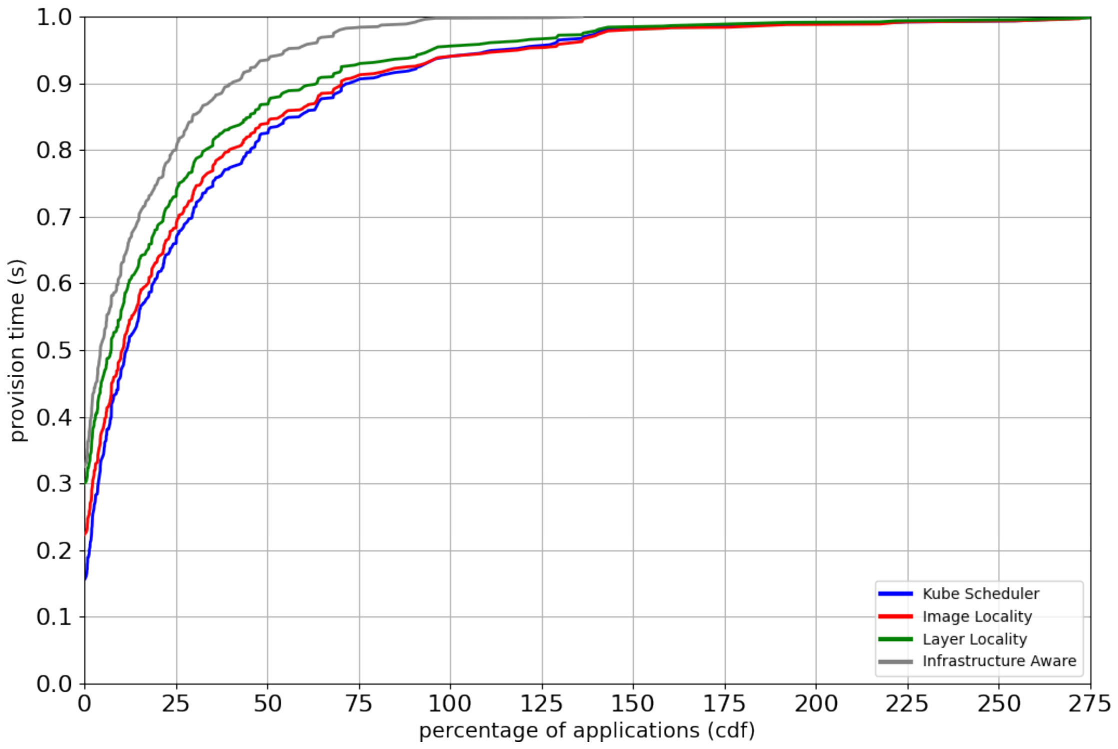


Figure 5.2: Provision time by scheduling algorithm

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 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 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 be deployed with Layer Locality and Kube Scheduler.

It is important to notice that the Image and Layer locality 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.

Node Storage Utilization

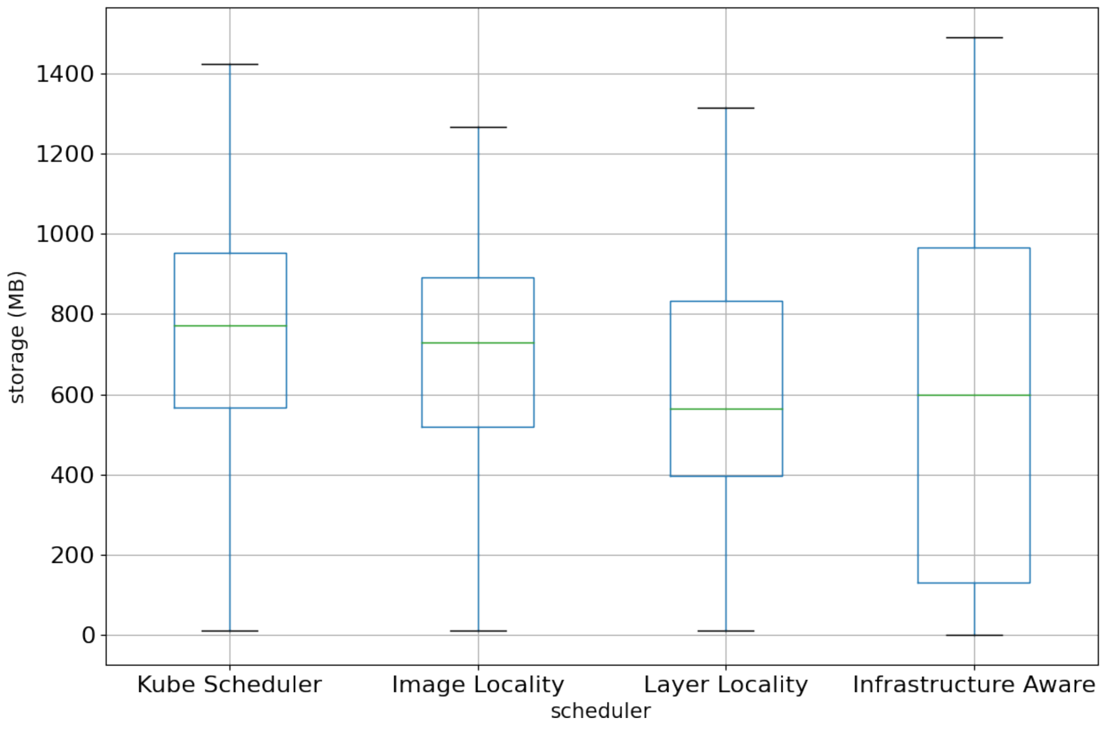


Figure 5.3: Node storage utilization by each scheduling algorithm

Usually, Kube-scheduler has as default behavior to equally distribute containers between the nodes that surpass the filtering phase on the scheduler. In Figure 5.3, 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 Kube-scheduler than 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 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.

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 5.2 summarizes the results obtained in the simulation.

5.5 Closing Remarks

This chapter 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 [Fu et al., 2020], 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 average network usage on the policy. Besides that, we also want to improve the simulation scenarios, adding more constraints and node limitations, like limited cache size, and implement the predicate on Kubernetes to evaluate our scheduler in a real cluster.

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 5.2: Additional statistics from the simulations

6. ENSURING SERVICE LEVEL AGREEMENT ON APPLICATION DEPLOYMENT IN AN EDGE INFRASTRUCTURE

6.1 Introduction

This chapter investigates the requirements to implement a solution that ensures a maximum time to deploy an application on the Edge. To achieve that, we propose a novel container scheduler based on a multi-objective genetic algorithm. This scheduler has the main objective of ensuring the Service Level Agreement set on each application that defines the time when the application is expected to be effectively active in the infrastructure. We also evaluate the scheduler against the Kube-scheduler and the Infrastructure-Aware presented in Chapter 5 using the ECOS simulator.

Besides that, we want so answer the following research question:

- *How the download queue on a node can be optimized to improve the deployment latency? Can the scheduler use the queue manipulation to ensuring service level agreements on the deployment total time?*

This chapter is based on our previously published paper "Luis Augusto Dias Knob, Carlos Henrique Kayser, Paulo Silas Severo de Souza, Tiago Ferreto. *Ensuring SLA Deployment Latency on Container Edge Infrastructure*". 14th IEEE/ACM International Conference on Utility and Cloud Computing (UCC 2021).

6.2 Related Work

Several works present scheduling strategies considering the SLA of deployments [Katsalis et al., 2016, Yao and Ansari, 2018]. In [Katsalis et al., 2016], the authors propose an SLA-driven scheduling strategy for VM placement in order to maximize the revenue of edge infrastructure-as-a-service (IaaS) providers and minimizing SLA violations, fairly between the various service providers using the Lyapunov optimization. The simulation-based results present benefits compared to the First Fit algorithm.

Yao and Ansari [Yao and Ansari, 2018] propose a Weighted Best Fit Decreasing (WBFD) algorithm to tackle a resource provisioning problem at the edge of the network, considering the possibility of resource failures happening while minimizing the system cost incurred by resources rentals without violating the SLA requirement. The resource provision problem is formulated as an Integer Linear problem (ILP). Simulation results show that the proposed heuristic algorithm performs close to the optimal solutions of ILP with lower computational complexity.

Some works propose evolutionary algorithms to improve the placement process at the edge of the network [Maia et al., 2021, Abbasi et al., 2020] and cloud [Guerrero et al., 2017]. In [Guerrero et al., 2017], the authors propose the utilization of the evolutionary algorithm Non-dominated Sorting Genetic Algorithm II (NSGA-II) for multi-objective container allocation optimization in cloud computing. Some of the objectives of the proposed strategy are: a) balanced cluster utilization; b) threshold distance; c) system failure; and d) reduction of the network overheads. Compared to the Kubernetes scheduler mechanism, the proposed strategy presents improvements in relation to the objectives addressed.

In [Maia et al., 2021], the authors propose a multi-objective genetic algorithm based on Biased Random-Key Genetic Algorithm (BRKGA) and NSGA-II to enhance the service placement and load distribution in an Internet of Things (IoT) and Edge Computing environment. For this, a Mixed-Integer Linear Programming (MILP) problem optimization problem is formulated to minimize the potential occurrence of SLA violations. The efficiency of the proposed algorithm is analyzed through simulation, and the proposed algorithm achieves values close to the optimum of the MILP formulation.

However, this solution aims to speed up the provisioning time of container-based application at the edge of the network avoiding provisioning time SLAs violations.

6.3 Problem Formulation

In this section, we describe the edge application provisioning problem targeted in this work. Firstly, we describe the main elements of the edge infrastructure considered in our modeling. Then, we formulate the several steps that comprehend provisioning applications in the edge nodes alongside our optimization objectives. Notations used in this chapter are summarized in Table 6.1.

We consider an edge computing network infrastructure modeled as an undirected graph $\mathcal{G} = (\mathcal{N}, \mathcal{S})$, where $\mathcal{N} = \{\mathcal{N}_1, \mathcal{N}_2, ..., \mathcal{N}_n\}$ represents a set of n edge nodes, in which the capacity of an edge node $\mathcal{N}_i$ is given by z_i, and $\mathcal{S} = \{\mathcal{S}_1, \mathcal{S}_2, ..., \mathcal{S}_m\}$ is the set of m links connecting the edge nodes. The set of u applications deployed in the edge nodes is represented by $\mathcal{A} = \{\mathcal{A}_1, \mathcal{A}_2, ..., \mathcal{A}_u\}$, where an application $\mathcal{A}_j \in \mathcal{A}$ has a provisioning time SLA $\wp_j$ and is comprised by r_j replicas $\mathcal{R}_j = \{\mathcal{R}_j^1, \mathcal{R}_j^2, ..., \mathcal{R}_j^{r_j}\}$, and a replica $\mathcal{R}_j^k$ has a demand h_j^k. The placement of application replicas on edge nodes is represented by a $\mathcal{N} \times \mathcal{A} \times \mathcal{R}$ tensor $\varkappa \in \{0, 1\}$, where:

Table 6.1: Summary of notation used in this chapter.

Notation	Description
$\mathcal{G}$	Network topology, comprised of edge nodes and links
$\mathcal{N}$	Set of n edge nodes in the infrastructure
z_i	Capacity of edge node N_i
q_i	Download queue of edge node $\mathcal{N}_i$
c_i	Cache memory of edge node $\mathcal{N}_i$
b_i	Bandwidth available for edge node $\mathcal{N}_i$
$\mathcal{S}$	Set of m links comprising the network infrastructure
$\mathcal{A}$	Set of u applications
$\wp_j$	Provisioning time SLA of application $\mathcal{A}_j$
∂_j	Provisioning time of application $\mathcal{A}_j$
$\mathcal{R}_j$	Set of r_j replicas from application $\mathcal{A}_j$
h_j^k	Demand of replica $\mathcal{R}_j^k$
$\varkappa_{i,j,k}$	Replicas placement scheme
β	Container registry node
$\mathcal{L}_j^k$	Set of container layers from replica $\mathcal{R}_j^k$
$t_v^{j,k}$	Size of container layer $\mathcal{L}_v^{j,k}$
$\vartheta_{i,j,k,v}$	Matrix that informs whether layer $\mathcal{L}_{j,k}^v$ is available in cache c_i or not
w_j^k	Waiting time of replica $\mathcal{R}_j^k$
d_j^k	Download time of replica $\mathcal{R}_j^k$

$$\varkappa_{i,j,k} = \begin{cases} 1 & \text{if edge node } \mathcal{N}_i \text{ hosts replica } \mathcal{R}_j^k \\ 0 & \text{otherwise.} \end{cases}$$

As instances of containerized applications, replicas are built on top of container layers that provide specific functionalities. For instance, a database replica could be made of 2 layers, one containing the operating system (e.g., *"ubuntu:latest"*) and the other the database itself (e.g., *"mysql:latest"*). As edge nodes can receive provisioning requests from multiple applications, a download queue q_i defines the order in which container layers of each replica hosted by an edge node $\mathcal{N}_i$ will be downloaded from the registry edge node β.

Although container layers may contain application-specific settings, many are generic, used in common by different applications. Accordingly, when provisioning a containerized application replica $\mathcal{R}_j^k$, an edge node $\mathcal{N}_i$ checks if $\mathcal{R}_j^k$ layers $\mathcal{L}_j^k$ have not been recently downloaded and are accessible in its cache c_i. Consequently, only not cached layers are downloaded from the registry node β, avoiding unnecessary traffic in the network and potentially shortening applications' provisioning time. We can check whether container layer $\mathcal{L}_{j,k}^v$ is available in cache c_i through a $\mathcal{N} \times \mathcal{L}$ matrix ϑ, where:

$$\vartheta_{i,j,k,v} = \begin{cases} 1 & \text{if layer } \mathcal{L}^{v}_{j,k} \text{ is available in } c_i \\ 0 & \text{otherwise.} \end{cases}$$

We assume that edge nodes download container layers from the registry sequentially given their queues. Therefore, a replica $\mathcal{R}^k_j$ located at position ρ in a download queue q_i has to wait for w^k_j units of time before getting downloaded, where w^k_j represents the time needed to download all previous items in q_i. When its turn comes, downloading replica $\mathcal{R}^k_j$ takes d^k_j units of time, as denoted in Equation 6.1.

$$d^k_j = \sum_{v=1}^{|\mathcal{L}^k_j|} \frac{t^{j,k}_v}{b_i} \cdot (1 - \vartheta_{i,j,k,v}) \tag{6.1}$$

More specifically, d^k_j accounts for the time needed to download all container layers $\mathcal{L}^k_j$ of replica $\mathcal{R}^k_j$ not available in c_i from the container registry β. The download time of an uncached layer $\mathcal{L}^{j,k}_v$ depends on its size $t^{j,k}_v$ and b_i, which denotes the available bandwidth for edge node $\mathcal{N}_i$. We assume that the provisioning of an application $\mathcal{A}_j$ is only complete when all its replicas $\mathcal{R}_j$ are successfully provisioned in the infrastructure. Therefore, the overall provisioning time of $\mathcal{A}_j$ can be described as $\partial_j = \sum_{k=1}^{r_j} w^k_j + d^k_j$.

Our goal consists in defining the placement of application replicas and the arrangement of the edge nodes' download queues to minimize the number of SLA violations due to prolonged provisioning times. Accordingly, the objective function can be formulated as in Equation 6.2, where constraint 1 (Equation 6.3) guarantees that each replica is only provisioned once, constraint 2 (Equation 6.4) sets the lower bound of provisioning times, and constraint 3 (Equation 6.5) certifies that edge nodes are not overloaded.

$$\textbf{Minimize} \sum_{j=1}^{u} [\partial_j > \wp_j] \tag{6.2}$$

Subject To:

$$\sum_{j=1}^{u} \sum_{k=1}^{r_j} \varkappa_{i,j,k} = 1, \; \forall_i \in \{1,\ 2,\ ...,\ n\} \tag{6.3}$$

$$\partial_j \geqslant 0, \; \forall_j \in \{1,\ 2,\ ...,\ u\} \tag{6.4}$$

$$\sum_{j=1}^{u} \sum_{k=1}^{r_j} h^k_j \cdot \varkappa_{i,j,k} \leqslant z_i, \; \forall_i \in \{1,\ 2,\ ...,\ n\} \tag{6.5}$$

6.4 Deployment Latency SLA Enforcement Scheduler

Defining placement schemes for application replicas and finding proper arrangements for edge nodes' download queues, which is a variant of the Application Scheduling Problem [Topcuoglu et al., 2002], is an NP-hard optimization problem. For that reason, approximation algorithms represent viable alternatives to find acceptable solutions within a bounded time. As finding the optimal solution is infeasible given the problem complexity, we calculate a Pareto Front to find a set of non-dominated solutions (i.e., none of the solutions found beat them in all objectives) [Fard et al., 2014]. Figure 6.1 presents a visual representation of a Pareto Front in a sample bi-objective optimization.

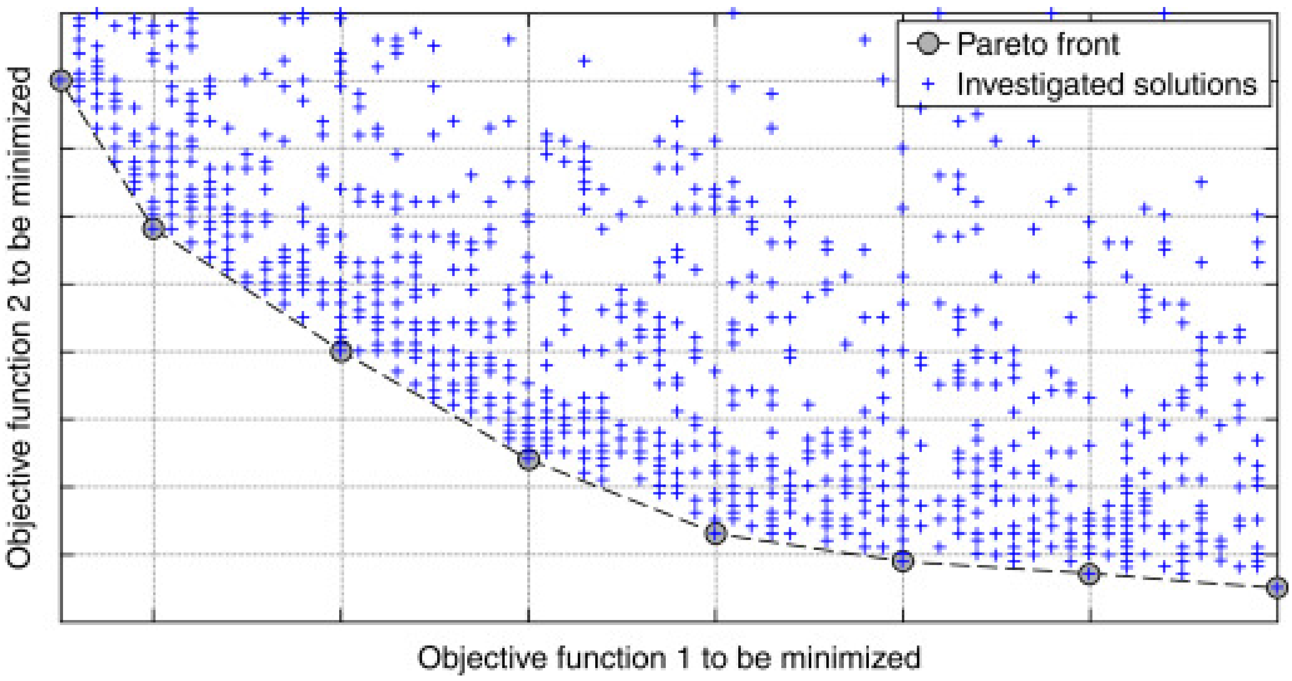


Figure 6.1: Visual representation of a Pareto Front [Ascione et al., 2018].

There are several single and multi-objective algorithms that can find pareto-optimal solutions [Deb et al., 2002, Hao et al., 2006, Deb and Jain, 2013]. We employ the Non-dominated Sorting Genetic Algorithm II (NSGA-II) [Deb et al., 2002], as it reaches superior results compared to other meta-heuristics [Zambrano-Vega et al., 2017]. In the remaining of this section, we present a novel scheduling algorithm called Deployment Latency SLA Enforcement Scheduler (DLSLA), which leverages NSGA-II functionality to minimize the number of SLA violations due to overextended provisioning time of multi-replica applications in edge computing environments.

6.4.1 Population Initialization

Our scheduling algorithm takes as input each container node that passes from the Kube-scheduler Filter stage. First, we evaluate the download queue from the node, and if the download queue has less than three containers, we run a score function based on a simplified fitness function implementation to define the scoring values. To queues with 3 or more applications, we send the node to DLSLA, setting the number of population and generation to $minqueue\_size - 1)!, 100$ and $min(queue\_size - 2)!, 100)$ respectively.

Running DLSLA to nodes with a download queue smaller than three containers is not cost-effective, so we only run it when necessary. After receiving the scoring values to all nodes, we send them to a custom-made ranking and bind implementation that chooses the node that will deploy the given application replica.

6.4.2 Download Queue Implementation

When a container node receives several container lifecycle operations simultaneously or close in time, it needs to create a queue to manage the order in which operations will be processed. This queue works in a FIFO (First-In-First-Out) model, where each container layer, when not in the cache, will be downloaded based on the manifest order (usually first the base layer until the top layer). Typically, the container runtime can manage three simultaneous downloads that will share the connection. However, this value is configurable so that it can be decreased to one, for example. After finishing all layers for one container, the runtime starts to download the next one.

So, in an overcrowded cluster, with many applications and lifecycle operations, this can generate situations where large applications (like Java-based or databases) will take several seconds or even minutes to be deployed, halting the application instantiation in a set of nodes. In cloud computing, with large bandwidth links, usually, it is not a problem. Still, edge computing with small links and highly-shared infrastructure can generate bottlenecks (for example, 1GB image in a 20Mbps network will take more than 5 minutes to be downloaded using all available bandwidth). Hence, driving the need for availability and mobility between nodes at the edge, we understand that some applications need to be deployed faster than others, and one way to do this is to take a better position on a busy node's queue.

By default, in the current runtime implementations, this queue can not be altered. So the only way to manage the order of waiting operations is to remove them from the scheduler and re-add them in a new order. However, these operations will redo the scheduler process and can return a new set of distinct nodes. Furthermore, sometimes we only want to change the queue order in one busy node and not in the managed cluster. Therefore, we propose a new implementation of the operations queue, where the queue can be altered respecting the currently active operation. We implemented a simple modification where, when we schedule a new operation to a node, we send the new order for the waiting operations, which can be entirely different from the currently active order.

This provides fine-grained control over the lifecycle operations and enable new policies and scheduler algorithms to ensure the amount of time needed to instantiate or update an application, improving the overall cluster utilization. So, together with the scoring values, the DLSLA returns to the ranking and bind modules, the best waiting queue to the node. If the scheduler selects the node with the container bind, it also reconfigures the waiting queue based on that return by the GA.

6.4.3 Chromosome Representation

We represent each possible solution (so-called chromosome) as an array, called *containers*, which can be correlated with the node's deployment queue. Each value on the queue, called *gene*, is set by the unique identifier representing the containers that need to be downloaded and deployed. No container can appear more than once on the queue, and all containers need to be present on each chromosome. If several applications use the same container, the algorithm will put only the one with the smallest SLA value since all applications that use the same container will start together after the image download. Figure 6.2 presents a graphic representation from the chromosomes and the relation between the containers images, layers, download queue, and the chromosome.

6.4.4 Fitness Function

Since DLSLA is based on NSGA-II, which is designed for multi-objective optimization problems, the expected result from the fitness function is a dictionary containing the values for each optimization objective. In our case, the fitness value is represented by the three functions to be minimized, namely *sla_violations*, *total_time*, *changes_on_queue*. Since we already receive the node after the Filter stage, it is impossible that a given constraint has an infinite or negative result, so we do not define default values for any criteria.

Algorithm 6.1 presents the fitness function implementation, where we first calculate, to a given queue, the amount of time needed to deploy all the containers. We also validate how many SLA violations that a given chromosome will generate. Finally, we verify the number of changes between the original queue and the queue presented on the chromosome.

Layers

L_n	Hash	Size
L_1	sha256:abcd...	12.5 MB
L_2	sha256:cde5...	21.3 MB
L_3	sha256:12a7...	1.9 MB
L_4	sha256:cd87...	50.3 MB
L_5	sha256:0fe5...	33.8 MB
L_6	sha256:38ca...	40.1 MB

Containers

C_n	Hash	Layers
C_1	sha256:45df...	$\{L_1,L_3,L_2\}$
C_2	sha256:c1e8...	$\{L_1,L_4\}$
C_3	sha256:a37c...	$\{L_5,L_6\}$
C_4	sha256:07f2...	$\{L_5,L_6\}$

Chromossomes

Ch_n	Containers	Time
Ch_1	$\{C_1,C_2,C_3,C_4\}$	25s
Ch_2	$\{C_1,C_2,C_4,C_3\}$	25s
Ch_3	$\{C_3,C_4,C_1,C_2\}$	15s
Ch_4	$\{C_3,C_2,C_1,C_4\}$	45s

Download Queue

Ch_n	Layers
Ch_1	$\{L_1,L_3,L_2,L_4,L_5,L_6\}$
Ch_2	$\{L_1,L_3,L_2,L_4,L_5,L_6\}$
Ch_3	$\{L_5,L_6,L_1,L_3,L_2,L_4\}$
Ch_4	$\{L_5,L_6,L_1,L_4,L_3,L_2\}$

Figure 6.2: Chromosome Representation.

6.4.5 Genetic Operators

Selection. As DLSLA is based on NSGA-II, it uses three concepts to select the best chromosomes in the population [Guerrero et al., 2017]. The first is dominance, where a chromosome dominates another if the fitness values for all objectives is better than the dominated. Second, the optimal fronts that group the chromosomes non-dominated by other ones using the Pareto distribution. Finally, the crowding distance is calculated by the average distance along each objective between the chromosome in the same front. This average distance is used to sort the chromosomes. After these three operations, we have a population with $len(population)*2-1$ chromosomes. Then, the algorithm selects the $len(population)$ best chromosomes.

Crossover. After selecting the fittest chromosomes, DLSLA employs a mating process (called crossover) to evolve the population. The crossover process used by DLSLA can be seen on Algorithm 6.2. First, we run the algorithm $len(population)-1$ times, selecting the chromosomes $population_x$ and $population_{x+1}$ as parents for each new child. Then, for

```
Input  : node_queue: Node download queue; bandwidth: Network bandwidth available on
         node; inital_time: Simulation time where the new application need to be instantiate;
         chromosome: Chromosome that needs to compute; original_app_queue: Current
         application queue on node
Output: Fitness score to the chromosome
1  sla_violations ← 0
2  total_time ← {}
3  changes_on_queue ← 0
4  for $gene \in chromosome_{genes}$ do
5      for $digest, size \in gene_{layers}$ do
6          if $digest \notin node\_queue$ then
7              total_time+ = $size \div bandwidth$
8              $node\_queue.append(digest)$
9      if $total\_time > gene_{sla}$ then
10         sla_violations+ = 1
11 for $app \in original\_app\_queue$ do
12     $temp\_index \leftarrow node\_queue.index(app)$
13     $node\_queue.pop(temp\_index)$
14     changes_on_queue+ = $temp\_index$
15 fitness ← {sla_violations, total_time,
     changes_on_queue}
16 return $fitness$
```

Algorithm 6.1: Fitness function.

each child gene, we randomly select a gene in the same position from one of the parents. We also store the gene not chosen, so in case of conflict or if the gene is already on the child, we pop one storage gene and complete the queue with a unique gene since no gene appears more than one time on the chromosome. This crossover function guarantees that if a gene is equal in both parents, it stays the same on the child. So, only distinct genes between parents are randomly selected for the child.

Mutation. We mutate chromosomes generated in the crossover process to avoid local optimum. Our mutation function is applied in a random number of new individuals, executing *number_of_elements*/2 swaps on the queue order.

6.4.6 Scheduler Score and Ranking

After the GA runs for a given number of generations, we return the high classified chromosome as the best solution to that given node. After executing the scoring algorithm to all nodes, be it the simplified version or the GA, we rank all nodes by the following weights:

$weight \leftarrow 0$

```
Input  : $population$: population of chromosomes;
Output: New population with original chromosomes plus children
for $x < len(population) - 1$ do
    child $\leftarrow \emptyset$
    $father1 \leftarrow population_x$
    $father2 \leftarrow population_{x+1}$
    $cache \leftarrow \emptyset$
    for $y < len(father1_{genes})$ do
        $gene1, gene2 \leftarrow random(father1_{genes[y]}, father2_{genes[y]})$
        if $gene1 \notin child_{genes}$ then
            $child_{gene[y]} \leftarrow gene1$
            if $gene1 \in cache$ then
                $cache.delete(gene1)$
            if $gene2 \notin cache \wedge gene2 \notin child_{genes} \wedge gene1 \neq gene2$ then
                $cache.append(gene2)$
        else if $gene2 \notin child_{genes}$ then
            $child_{gene[y]} \leftarrow gene2$
            if $gene2 \in cache$ then
                $cache.delete(gene2)$
        else
            $child_{gene[y]} \leftarrow cache.pop()$
    if $random() < mutation\_rate$ then
        child $\leftarrow mutation($child$)$
    $child_{fitness} \leftarrow fitness($child$)$
    $population.append($child$)$
return $population$
```

Algorithm 6.2: Crossover function.

$weight+ = 10 - sla\_violations * 0.5$

$weight+ = min\_time / time\_on\_chost * 10$

$weight+ = 10 - number\_of\_apps\_on\_node$

$weight+ = 10 - changes\_on\_queue * 0.5$

We select the node with the highest score to host the container. If more than one has the same final weight, we randomly select a node between them.

6.5 Evaluation

This section presents an evaluation of the DLSLA scheduling algorithm. The algorithm is compared to the Kube-scheduler and the algorithms presented in Chapter 5 (Infrastructure-Aware Scheduler) in a simulated scenario based on the Brazilian research network topology using Docker Hub images. We choose these two algorithms as a baseline because the first is the default scheduler enabled on Kubernetes. The second implements network availability as a priority, decreasing the deployment latency without considering the SLA. The metrics used for comparison include the number of applications that do not fulfill the SLA deployment latency, scheduling distribution, among others.

6.5.1 Simulation Scenario

To simulate an edge computing topology, we used the Brazilian Research Network, called Ipê. This topology interconnects all Brazilian universities and research institutes through 28 Points of Presence (PoPs) distributed over the country (Figure 5.1). 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 [RNP, 2021b]. We implemented this topology in the ECOS simulator presented in Chapter 3.

To compare our solution with the other two schedulers, we deployed a set of container nodes on the Ipê topology. Each PoP included a large node named Worker Node and five small nodes named Edge Nodes. The main difference between the nodes is the bandwidth available to each one. For instance, the worker 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 worker or edge nodes. Other communications that may occur in a real network are ignored.

The registry, where all nodes request the images, is placed on PoP-São Paulo. This PoP is one of the most connected in the topology and is the primary connection to cloud providers [RNP, 2021a]. We also set the bandwidth of the Registry Node to 10Gbps to avoiding it to represent a bottleneck on the simulation.

6.5.2 Workload

The workload used in the simulation is based on the Docker Hub [Docker, 2021] twenty four most downloaded images, excluding base images. 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 a given node needs to download to have all applications is 2152.78 MB (37% of similarity between images). We create a random number of applications between 5 and 25 (distributed uniformly) for each image. And for each application, we deploy a random number of replicas between 2 and 5 (distributed uniformly). Finally, each application has a random scheduling time between 0 and 1000 seconds (distributed uniformly), which defines the exact moment the application needs to be scheduled in the topology.

After setting the topology and the applications that need to be deployed, we create three scenarios with distinct types of *Service Level Agreements* related to the amount of time needed to full instantiate the applications on the topology:

- **Random Distribution:** We set random SLAs with values between 30 and 150 seconds for each application.
- **Normal Distribution:** We set five possible SLAs (30, 60, 90, 120, 150 seconds) with different weights (5%, 20%, 50%, 20%, 5%) for each application.
- **60 Sec SLA:** We set a 60 seconds SLA for each application without distinction.

It is important to note that this SLA is not hard defined, so the application is always deployed, even if it is not fulfilled. After preparing the scenarios, we run the simulation 30 times for each algorithm (DLSLA, Infrastructure-Aware, and Kube-scheduler). All results present next use the arithmetic average between the runs. Table 6.2 presents the parameters used in the experiments.

Parameter	***Value***
Server Nodes	*28*
Server Links	*100 Mbps*
Edge Nodes	*140*
Edge Links	*10 - 60 Mbps*
Registry Node Link	*10 Gbps*
Number of Images	*24*
Number of Applications	*350*
Number of Replicas	*1258*

Table 6.2: Simulation parameters.

6.5.3 Results

With the simulation, we want to evaluate three main variables: the number of applications that do not fulfill the SLA; on this applications' set, we want to know how much was the average time over the SLA; and finally, we want to understand how the applications' scheduling distribution was an impact between the worker and edge nodes. Since a complete centralization on the worker nodes, probably will decrease the time needed to deploy the application, but will decrease the total utilization from the topology. We also summarize the simulation results in Table 6.3, and present additional information like average time to all applications and number of replicas that do not fulfill the SLA.

We understand that one application does not achieve the SLA if any replica that composes this application does not start until the expected time defined by the SLA. We also want to clarify that, as the SLA is not hard ensured, the download queue created on each node with more than one container deployed simultaneously will cumulatively impact all the new applications' schedule. With that in mind, in Figure 6.3 we present the average number of applications that do not fulfill the SLA on each scenario.

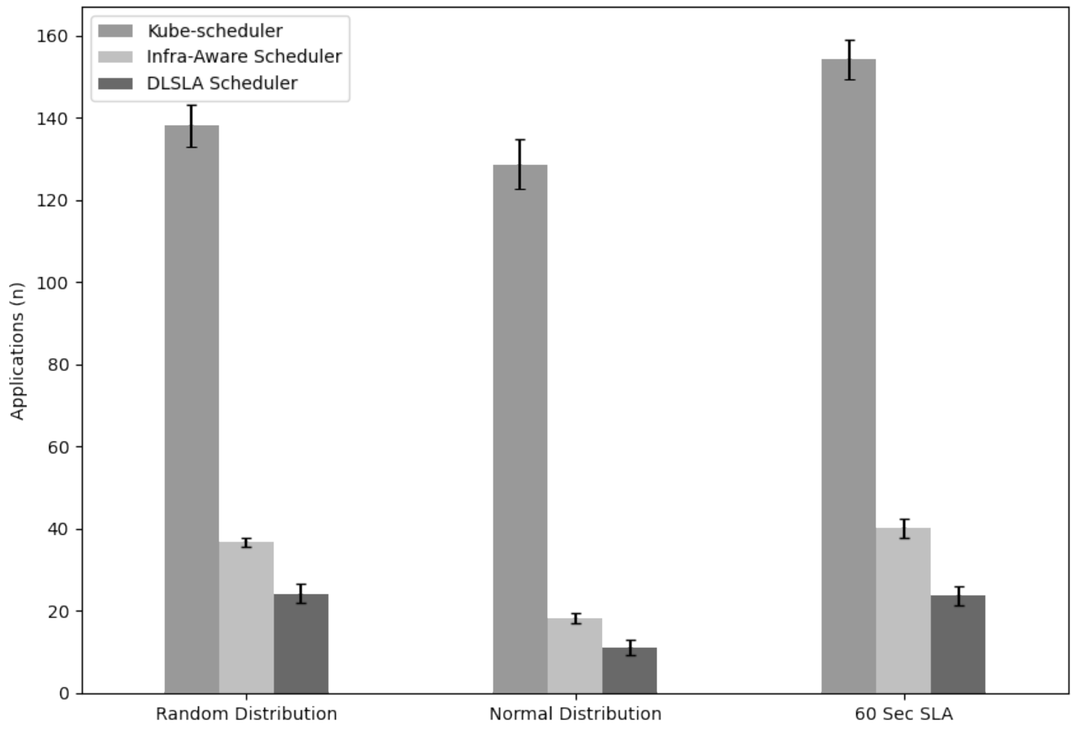


Figure 6.3: Number of applications that do not fulfill the SLA.

The results show that both the DLSLA and the Infrastructure-Aware managed to decrease the amount of not fulfilled applications in all scenarios, having as results 73.80%, 62.89%, 67.06% and 69.16%, 62.25%, 62.97% smaller, respectively to the Random, Normal Distribution and the 60 Sec SLA scenarios in comparison with the Kube-scheduler. While DLSLA has more than 90.88% of the application that achieves the SLA, the Infrastructure-Aware has a slightly inferior with 88.97% ensure SLA applications. Kube-scheduler presents that the percentage of fulfilling applications in the best scenario (Normal Distribution) was only 71.55%. This was an expected result since the Kube-scheduler does not consider the network availability or the SLA deadline time as a priority to the scheduling.

This also reflects in our second experiment presented in Figure 6.4, where we plot the average time over the SLA to each application that was not deployed within the time defined by the SLA. With fewer applications ensuring the SLA, Kube-scheduler expected to

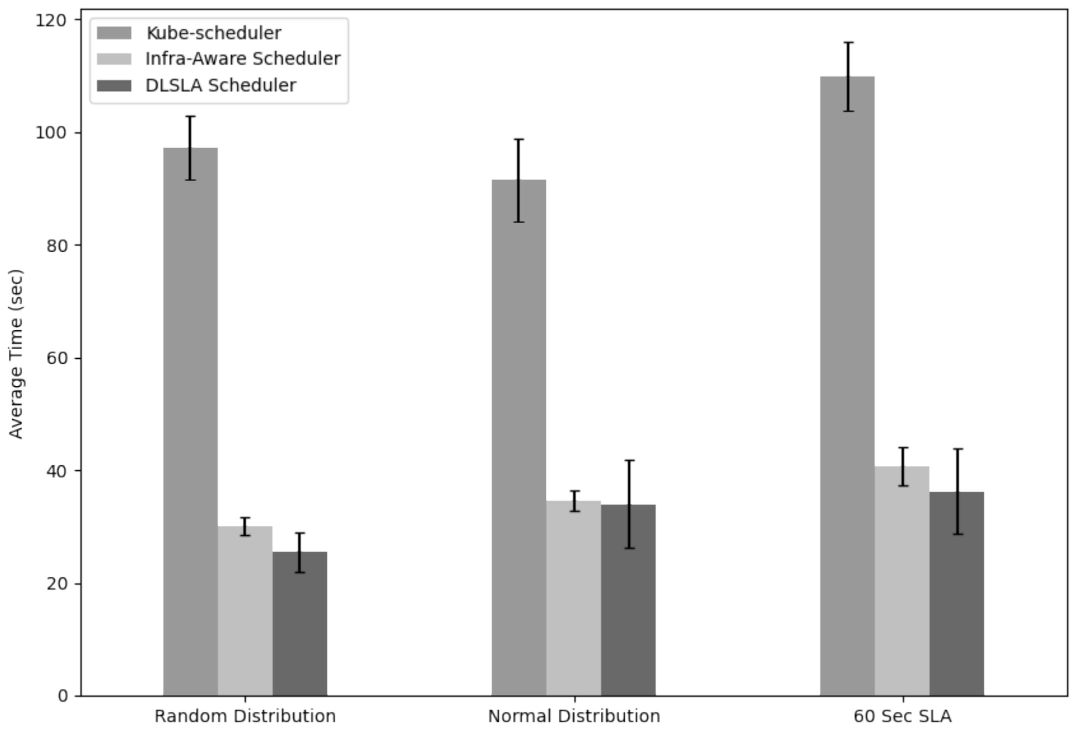


Figure 6.4: Average time over the SLA to not fulfilled applications.

have the biggest average time on each scenario, with a value close to 100 seconds in all three experiments (97.25, 91.52, 109.94). Meanwhile, the DLSLA has a better average time than the Infrastructure-Aware in all scenarios but with a bigger standard deviation over the runs. This happened for two reasons, first, the number of applications that do not fulfill the SLA is about 40% smaller using DLSLA than Infrastructure Aware. Smaller samplings will have a more significant standard deviation if the values are close to the average. Second, the infrastructure tends to be deterministic by always selecting the same nodes with more bandwidth available presenting a more consistent behavior between runs.

Finally, we want to understand the distribution impact between the worker and edge nodes based on the number of container schedules. Figure 6.5 presents a violin distribution to the container scheduler per node. In a worst-fit distribution, all nodes have 7.5 containers scheduled on average. Kube-scheduler shows the closest gap to this value, both on the edge and worker nodes, with a slightly bigger average on the worker node in all scenarios. While DLSLA and Infrastructure Aware present quite distinct values to the worker and edge nodes. On average, the worker nodes were selected by the Infrastructure-Aware scheduler 13.94, 14.75, and 14.21 times on average for the Random, Normal, and 60 Sec SLA, respectively. Meanwhile, the worker nodes were chosen by the DLSLA 15.29, 15.24, and 15.20 times on average.

Although the worker nodes have been chosen 2.25, 2.71, 2.72 times with the Infrastructure Aware and 2.57, 2.85, 2.99 times with the DLSLA more than the edge nodes, these

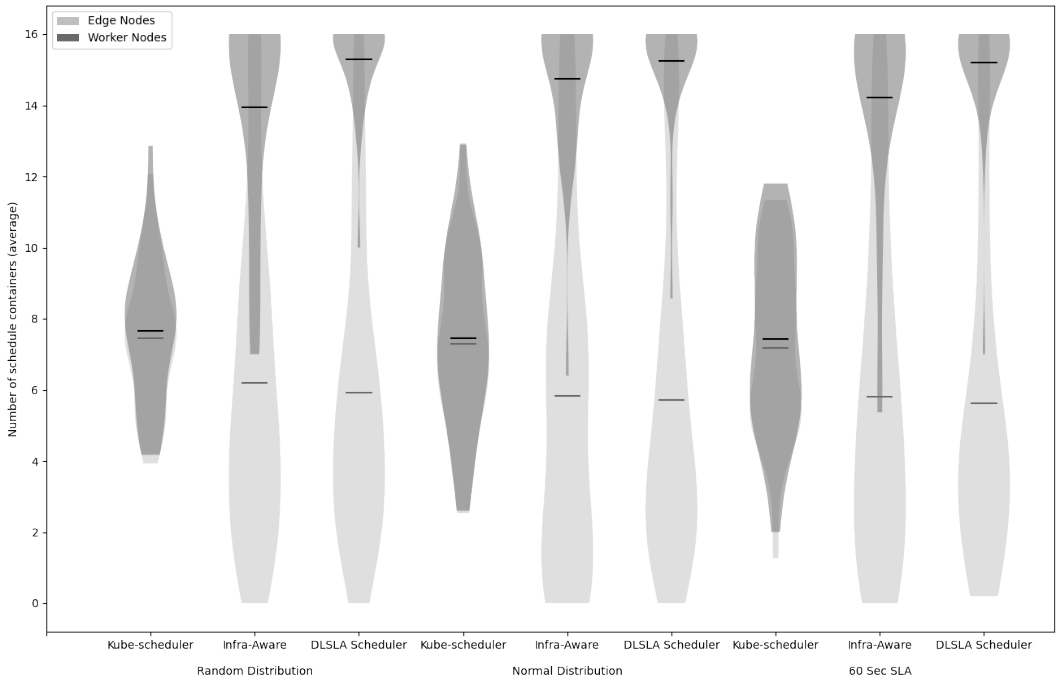


Figure 6.5: Average distribution between Edge and Worker Nodes.

nodes still allocated 68.95%, 64.79% and 64.72% of the applications with the Infrastructure-Aware scheduler and 65.98%, 63.68% and 62.51% with the DLSLA. That happened because there were five times more edge nodes in the infrastructure than worker nodes. In comparison, the edge nodes with the Kube-scheduler were chosen at 82.95%, 81.05%, and 79.79% of the time. Lastly, it is possible to visualize in the distribution that in all scenarios, the Infrastructure-Aware concentrate more on a set of nodes than the DLSLA, having, for example, more edge nodes with zero container schedule (5.2, 15.8, 17.7 versus 4.2, 3.0, 1.2 on average). We summarize the experiments in Table 6.3, presenting more elaborated statistics for each one of the nine running sets.

Scenario		**Container Scheduling**						
		Worker Nodes (total)	***Edge Nodes (total)***	***Proportion on Topology (%)***	***Worker Node (avg)***	***Worker Node (% per node)***	***Edge Node (avg)***	***Edge Node (% per node)***
Random Distribution	*Kube-Scheduler*	1043.51	214.48	82.95-17.05	7.66	0.59	7.45	0.59
	Infrastructure-Aware	867.44	390.55	68.95-31.05	13.95	1.11	6.19	0.49
	DLSLA	829.96	428.03	65.98-34.03	15.29	1.22	5.93	0.47
Normal Distribution	*Kube-Scheduler*	1019.6	238.4	81.05-18.95	8.51	0.68	7.28	0.58
	Infrastructure-Aware	815.1	442.9	64.79-35.21	15.82	1.26	5.82	0.46
	DLSLA	801.06	459.93	63.68-36.32	16.32	1.30	5.72	0.45
60 Sec SLA	*Kube-Scheduler*	1003.7	254.3	79.79-20.21	9.08	0.72	7.17	0.57
	Infrastructure-Aware	814.13	443.86	64.72-35.28	15.85	1.26	5.81	0.46
	DLSLA	786.4	471.6	62.51-37.49	16.84	1.34	5.62	0.45

Scenario		***SLA Fulfillment***		***Application Deployment Latency***			
		avg (n)	***std (n)***	***Not Fulfill Over SLA avg (sec)***	***Not Fulfill SLA std (n)***	***All Apps avg (sec)***	***All Apps std(n)***
Random Distribution	*Kube-Scheduler*	211.90	5.00	97,25	5,65	82.91	2.91
	Infrastructure-Aware	*313.21*	1.15	29,99	1,60	26.47	0.40
	DLSLA	*325.79*	2.30	25,48	3,50	25.42	0.59
Normal Distribution	*Kube-Scheduler*	221.24	6.10	91,52	7,37	84.48	3.80
	Infrastructure-Aware	*331.69*	1.20	34,55	1,86	25.04	0.30
	DLSLA	*338.93*	1.83	33,97	7,76	26.89	0.65
60 Sec SLA	*Kube-Scheduler*	195.76	4.86	109,94	6,14	88.68	3.38
	Infrastructure-Aware	*309.76*	2.29	40,71	3,46	26.84	0.85
	DLSLA	*326.17*	2.35	36,22	7,58	24.87	0.88

Table 6.3: Additional statistics from the simulations.

6.6 Conclusion

In this chapter, we have addressed the problem of container deployment time ensuring through SLA (that means ensuring the expected time that a given application needs to be running on the topology). We want to achieve that based on a three-objective optimization: (i) decrease the total time to deploy all containers, (ii) fulfill the biggest possible number of SLAs, and (iii) implement that with the smaller changes in the download queue as possible. To that, we developed a novel approach using a multi-objective genetic algorithm called DLSLA.

The results demonstrate that our approach provides a suitable solution for ensuring the SLAs, and it found optimized solutions within a reasonable population size and number of generations (100 and 200, respectively). We compared the results against Kube-scheduler and Infrastructure Aware through a set of simulations. As the Kube-scheduler does not consider the network infrastructure on the scheduler, our solution presents results of almost 200% better in ensuring the application SLA. We also show that the DLSLA scheduler presents better results than the Infrastructure-Aware while having a more consistent distribution between the edge nodes.

7. FINAL CONSIDERATIONS

This chapter summarizes and concludes our research presented in earlier chapters. First, the contributions are presented. After, we revisited the goals and research questions introduced in the first chapter. Finally, we discussed possible future works that can build on the research presented in this thesis.

7.1 Contributions

Among the contributions presented in this thesis, it is possible to identify the following items:

- Review of background technologies that define edge computing and containerization (Chapter 2).
- Review of the state-of-the-art in container orchestration focusing on decreasing deployment latency (Chapters 4, 5, and 6).
- Definition and implementation of an event-driven simulator to container edge orchestration (Chapter 3).
- Implementation of a registry placement algorithm based on fluid communities, including simulation with the GARR Topology (Chapter 4).
- Implementation of a new priority to kube-scheduler based on network availability, including simulation with the Ipê Network (Chapter 5).
- Definition and implementation of a new SLA-driven scheduler using genetic algorithm (Chapter 6).

Besides the thesis, during the Ph.D. we published four scientific papers as shown in 7.1. Also, this thesis was awarded with a PUCRS-PrInt scholarship to be developed partially in Italy. It was developed with the Fondazione Bruno Kessler - Trento. This collaboration has strengthened the relationship between the PUCRS and the FBK.

7.2 Revisiting the Goals and Research Questions

In the first chapter of this thesis, we described that one of the most critical challenges on containerization in edge computing is the deployment latency that may occur on

Year	Authors	Work Title	Conference
2018	**LAD Knob**, BG Xavier, T Ferreto	An unikernels provisioning architecture for OpenStack	ISCC
2021	**LAD Knob**, F Faticanti, T Ferreto, D Siracusa	Community-based placement of registries to speedup application deployment on Edge Computing	IC2E
2021	**LAD Knob**, C Kayser, T Ferreto	Improving Container Deployment in Edge Computing Using the Infrastructure Aware Scheduling Algorithm	ISCC
2021	**LAD Knob**, P Souza, C Kayser, T Ferreto	Ensuring SLA Deployment Latency on Container Edge Infrastructure	UCC

Table 7.1: Papers published during the PhD degree.

heterogeneous and resource-constrained scenarios. So, the main contribution of this thesis is to investigate all the main components of the container orchestration and how each one can be optimized. We also believe that the efforts to improve the deployment latency are indispensable contributions to the popularization of edge computing. Our goals, which we will revisit now, are related to this challenge. So, below, we draw our conclusions for each of the three goals presented at the start of this thesis.

> ***Goal 1:*** *To investigate the container deployment process on edge computing, learn how the schedulers' solutions works, and understand how other components can impact this operation*

The first goal focused on understanding the background technologies that depict the concept of edge computing and containerization. The background information has been described in Chapter 2. After understanding the deployment process, it became more apparent that the network has a larger impact on edge infrastructure than cloud computing. So, every contribution presented here has as main characteristic the network usage during the deployment process. Finally, more information on how each component can impact the total time to instantiate a new application can be seen in Chapters 4,5, and 6.

This goal is also highly related to the Research Questions *"What are the differences between the cloud and the edge on the application deployment process? Are the actual solutions adapted to this largely heterogeneous and constrained-resource scenario?"* and *"Is it possible to optimize the deployment process and reduce the latency created by them? If yes, what components should be optimized?"*, been the first answered by Chapter 2 and the second by the introduction and related work from Chapters 4,5, and 6.

> ***Goal 2:*** *To investigate if there is any solution to simulate or emulate the orchestration of large container edge infrastructure, and if necessary, evaluate the requirements to implement a simulator*

Large and distributed infrastructure is hard to replicate, so we knew that we would need to use simulation to validate our contributions. Our second goal was defined with that in mind. This also raised the Research Question *"How can we evaluate distinct solutions on edge scenarios? Is there any simulator that can be used? What are the main requisites for the simulation?"* that was answered in Chapter 3.

After evaluating the present solutions on edge/fog computing simulation, we decided to develop a new event-driven simulator focused on the container orchestration process. As far as we know, this was the first solution focused on this criteria. We also implemented a network sharing policy to investigate the main bottlenecks created by the deployment of simultaneous applications on distinct edge regions.

> ***Goal 3:*** *To improve the deployment process on edge infrastructure through new solutions on several phases of the deployment*

The last goal is synthesized in our last three Research Questions presented in Chapter 1: *"How the registry placement influence the deployment latency? How can we distribute the load network between several registries on the topology?"*, *"Does the scheduler uses any network information as input to schedule applications? Is it possible to implement a solution that uses the available bandwidth as input to the application schedule? How this impacts the other's priorities?"*, and *"How the download queue on a node can be optimized to improve the deployment latency? Can the scheduler use the queue manipulation to ensuring service level agreements on the deployment total time?"*. This question relies on optimization on several components and steps from the orchestration process and is answered in Chapters 4, 5, and 6.

The solutions present in these chapters are: a registry placement algorithm based on fluid communities, a priority to the kube-scheduler based on the network availability, and an SLA-driven scheduler using a genetic algorithm. These works have not fully stressed the optimizations on each of their components, and experimentation in real scenarios or testbeds still needs to be done.

7.3 Prospects for future research

In this final section of our final chapter, we discuss prospects for future research. We imagine several improvements that can be made on current works, including new simulator features and advanced techniques on infrastructure placement and application scheduler.

Implementation of bandwidth control on the worker node runtime: Today, all the QoS and limits implemented on container runtime mainly focus on the interference and requisites that can conflict when several applications run on the same node. However, the runtime itself is not limited by these configurations. Our first investigations prove that it is possible to generate a DoS attack spamming several lifecycle operations simultaneously on a orchestrate infrastructure, including application traffic. However, even if it appears promising, it needs further investigation to achieve significant results.

Dynamic registry placement: The registry placement is a challenging problem on the container orchestration since a well-positioning distribution can create a more robust infrastructure, with small latency enabling fast migration and many concurrent applications. In this thesis, we manually set the number of registries that need to be deployed in a given topology. A more dynamic scenario, may need a more fluid algorithm that can add and remove registries based on the topology operations. An improvement on our algorithm to fulfill this requirement is a future work that needs further investigation.

Improvements on the Simulator: We developed our simulator to provide an experimentation scenario to the deployment orchestration process. However, we focused on a generic solution that can be extended to distinct contexts. New modules and algorithms can be added to almost every component, improving the actual implementation. Possible solutions that can be developed are a more fine-grained control on the node resource, like CPU, memory, and energy management.

REFERENCES

Abbasi, M. Pasand, E. M. and Khosravi, M. R. (2020). Workload allocation in iot-fog-cloud architecture using a multi-objective genetic algorithm. *Journal of Grid Computing*, vol. 1, pp. 1–14.

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

Ahmed, A. and Pierre, G. (2020). Docker-pi: Docker container deployment in fog computing infrastructures. *International Journal of Cloud Computing*, vol. 9, pp. 6–27.

Andreev, K. and Racke, H. (2004). Balanced graph partitioning. *Theory of Computing Systems*, vol. 39, pp. 929–939.

Armbrust, M. Fox, A. Griffith, R. Joseph, A. D. Katz, R. Konwinski, A. Lee, G. Patterson, D. Rabkin, A. Stoica, I. and Zaharia, M. (2010). A view of cloud computing. *Communications of the ACM*, vol. 53, pp. 50–58.

Aryal, R. G. and Altmann, J. (2018). Dynamic application deployment in federations of clouds and edge resources using a multiobjective optimization AI algorithm. In: *Third International Conference on Fog and Mobile Edge Computing (FMEC)*, pp. 147–154. IEEE.

Ascione, F. Bianco, N. De Stasio, C. Mauro, G. M. and Vanoli, G. P. (2018). 5.21 energy management in hospitals. In: Dincer, I., editor, *Comprehensive Energy Systems*, pp. 827–854. Elsevier, Oxford.

Bernstein, D. (2014). Containers and cloud: From LXC to docker to kubernetes. *IEEE Cloud Computing*, vol. 1, pp. 81–84.

Bertsekas, D. P. Gallager, R. G. and Humblet, P. (1992). *Data networks*, vol. 2. Prentice-Hall International New Jersey.

Bonomi, F. Milito, R. Zhu, J. and Addepalli, S. (2012). Fog computing and its role in the internet of things. In: *First Edition of the MCC Workshop on Mobile Cloud Computing*, MCC ’12, pp. 13–16. ACM.

Burns, B. Grant, B. Oppenheimer, D. Brewer, E. and Wilkes, J. (2016). Borg, omega, and kubernetes. *Communications of the ACM*, vol. 59, pp. 50–57.

Calheiros, R. N. Ranjan, R. Beloglazov, A. De Rose, C. A. F. and Buyya, R. (2011). Cloudsim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms. *Software: Practice and Experience*, vol. 41, pp. 23–50.

Carella, G. A. and Magedanz, T. (2016). Open baton: A framework for virtual network function management and orchestration for emerging software-based 5g networks. *IEEE Newsletter*, vol. 2016, pp. 190–190.

Casalicchio, E. (2019). Container orchestration: A survey. In: Puliafito, A. and Trivedi, K. S., editors, *Systems Modeling: Methodologies and Tools*, vol. 1, pp. 221–235. Springer International Publishing, Cham, 1 ed..

Checko, A. Christiansen, H. L. Yan, Y. Scolari, L. Kardaras, G. Berger, M. S. and Dittmann, L. (2015). Cloud ran for mobile networks - 2014; a technology overview. *IEEE Communications Surveys Tutorials*, vol. 17, pp. 405–426.

Consortium-GARR (2021). Consortium garr. WebPage. https://www.garr.it/it/infrastrutture/rete-nazionale/infrastruttura-di-rete-nazionale. Date of access: 08/2021.

Darrous, J. Lambert, T. and Ibrahim, S. (2019). On the importance of container image placement for service provisioning in the edge. In: *28th International Conference on Computer Communication and Networks (ICCCN)*, pp. 1–9.

Deb, K. and Jain, H. (2013). An evolutionary many-objective optimization algorithm using reference-point-based nondominated sorting approach, part i: solving problems with box constraints. *IEEE transactions on evolutionary computation*, vol. 18, pp. 577–601.

Deb, K. Pratap, A. Agarwal, S. and Meyarivan, T. (2002). A fast and elitist multiobjective genetic algorithm: Nsga-ii. *IEEE transactions on evolutionary computation*, vol. 6, pp. 182–197.

Docker (2021). Docker Hub. WebPage. https://hub.docker.com. Date of access: 08/2021.

Doriguzzi-Corin, R. Scott-Hayward, S. Siracusa, D. Savi, M. and Salvadori, E. (2020). Dynamic and application-aware provisioning of chained virtual security network functions. *IEEE Transactions on Network and Service Management*, vol. 17, pp. 294–307.

Ellis, A. (2021). Openfaas. WebPage. https://docs.openfaas.com. Date of access: 08/2021.

Erdos, P. (1961). On the evolution of random graphs. *Bulletin of the Institute of International Statistics*, vol. 38, pp. 343–347.

ETSI MEC-ISG (2020). Mobile edge computing (mec); framework and reference architecture, etsi gs mec 003 v2.2.1. *ETSI, DGS MEC*, n$^{\underline{o}}$ 3, pp. 1–21.

Fahs, A. J. and Pierre, G. (2019). Proximity-aware traffic routing in distributed fog computing platforms. In: *19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, pp. 478–487. IEEE.

Fard, H. M. Prodan, R. and Fahringer, T. (2014). Multi-objective list scheduling of workflow applications in distributed computing infrastructures. *Journal of Parallel and Distributed Computing*, vol. 74, pp. 2152–2165.

Faticanti, F. De Pellegrini, F. Siracusa, D. Santoro, D. and Cretti, S. (2019). Cutting throughput with the edge: App-aware placement in fog computing. In: *6th IEEE International Conference on Cyber Security and Cloud Computing (CSCloud)/ 5th IEEE International Conference on Edge Computing and Scalable Cloud (EdgeCom)*, pp. 196–203. IEEE.

Fesehaye, D. Gao, Y. Nahrstedt, K. and Wang, G. (2012). Impact of cloudlets on interactive mobile cloud applications. *16th IEEE International Enterprise Distributed Object Computing Conference*, vol. 1, pp. 123–132.

Fu, S. Mittal, R. Zhang, L. and Ratnasamy, S. (2020). Fast and efficient container startup at the edge via dependency scheduling. In: *3rd USENIX Workshop on Hot Topics in Edge Computing (HotEdge 20)*, pp. 1–6. USENIX.

Garcia Lopez, P. Montresor, A. Epema, D. Datta, A. Higashino, T. Iamnitchi, A. Barcellos, M. Felber, P. and Riviere, E. (2015). Edge-centric computing: Vision and challenges. *SIGCOMM Computer Communication Review*, vol. 45, pp. 37–42.

Guerrero, C. Lera, I. and Juiz, C. (2017). Genetic algorithm for multi-objective optimization of container allocation in cloud architecture. *Journal of Grid Computing*, vol. 16, pp. 113–135.

Ha, K. Abe, Y. Eiszler, T. Chen, Z. Hu, W. Amos, B. Upadhyaya, R. Pillai, P. and Satyanarayanan, M. (2017). You can teach elephants to dance: agile vm handoff for edge computing. In: *Second ACM/IEEE Symposium on Edge Computing - SEC '17*, pp. 1–14. ACM Press.

Hao, J. Jin-hua, Z. et al. (2006). Multi-objective particle swarm optimization algorithm based on enhanced ε-dominance. In: *IEEE International Conference on Engineering of Intelligent Systems*, pp. 1–5. IEEE.

Harter, T. Salmon, B. Liu, R. Arpaci-Dusseau, A. C. and Arpaci-Dusseau, R. H. (2016). Slacker: Fast distribution with lazy docker containers. In: *14th USENIX Conference on File and Storage Technologies (FAST 16)*, pp. 181–195, Santa Clara, CA. USENIX Association.

Hsu, W.-L. and Nemhauser, G. L. (1979). Easy and hard bottleneck location problems. *Discrete Applied Mathematics*, vol. 1, pp. 209–215.

Hu, Y. C. Patel, M. Sabella, D. Sprecher, N. and Young, V. (2015). Mobile edge computing — a key technology towards 5g. *ETSI White Paper No. 11*, vol. 11, pp. 1–16.

Huang, Z. Wu, S. Jiang, S. and Jin, H. (2019). Fastbuild: Accelerating docker image building for efficient development and deployment of container. In: *35th Symposium on Mass Storage Systems and Technologies (MSST)*, pp. 28–37. SCU.

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

Jain, R. and Tata, S. (2017). Cloud to edge: Distributed deployment of process-aware IoT applications. In: *IEEE International Conference on Edge Computing (EDGE)*, pp. 182–189. IEEE.

Jararweh, Y. Tawalbeh, L. Ababneh, F. Khreishah, A. and Dosari, F. (2014). Scalable cloudlet-based mobile computing model. *Procedia Computer Science*, vol. 34, pp. 434 – 441.

Kangjin, W. Yong, Y. Ying, L. Hanmei, L. and Lin, M. (2017). Fid: A faster image distribution system for docker platform. In: *IEEE 2nd International Workshops on Foundations and Applications of Self* Systems (FAS*W)*, pp. 191–198. IEEE.

Katsalis, K. Papaioannou, T. G. Nikaein, N. and Tassiulas, L. (2016). Sla-driven vm scheduling in mobile edge computing. In: *IEEE 9th International Conference on Cloud Computing (CLOUD)*, pp. 750–757. IEEE.

Kubernetes (2021a). Kube-scheduler component configs. WebPage. https://github.com/kubernetes/kube-scheduler. Date of access: 08/2021.

Kubernetes (2021b). Production-grade container orchestration. WebPage. https://kubernetes.io/. Date of access: 08/2021.

Latora, V. Nicosia, V. and Russo, G. (2017). *Complex networks: principles, methods and applications*. Cambridge University Press.

Lera, I. Guerrero, C. and Juiz, C. (2019). Yafs: A simulator for iot scenarios in fog computing. *IEEE Access*, vol. 7, pp. 91745–91758.

Littley, M. Anwar, A. Fayyaz, H. Fayyaz, Z. Tarasov, V. Rupprecht, L. Skourtis, D. Mohamed, M. Ludwig, H. Cheng, Y. and Butt, A. R. (2019). Bolt: Towards a scalable docker registry via hyperconvergence. In: *IEEE 12th International Conference on Cloud Computing (CLOUD)*, pp. 358–366. IEEE.

Maia, A. M. Ghamri-Doudane, Y. Vieira, D. and de Castro, M. F. (2021). An improved multi-objective genetic algorithm with heuristic initialization for service placement and load distribution in edge computing. *Computer Networks*, vol. 194, pp. 108–146.

Mell, P. M. and Grance, T. (2011). The NIST definition of cloud computing. Standard publication, National Institute of Standards & Technology, Gaithersburg, MD, United States.

Merkel, D. (2014). Docker: Lightweight linux containers for consistent development and deployment. *Linux Journal*, vol. 2014, pp. 1–6.

Morabito, R. Farris, I. Iera, A. and Taleb, T. (2017). Evaluating performance of containerized IoT services for clustered devices at the network edge. *IEEE Internet of Things Journal*, vol. 4, pp. 1019–1030.

Nathan, S. Ghosh, R. Mukherjee, T. and Narayanan, K. (2017). Comicon: A co-operative management system for docker container images. In: *IEEE International Conference on Cloud Engineering (IC2E)*, pp. 116–126. IEEE.

Nikdel, Z. Gao, B. and Neville, S. W. (2017). Dockersim: Full-stack simulation of container-based software-as-a-service (saas) cloud deployments and environments. In: *IEEE Pacific Rim Conference on Communications, Computers and Signal Processing (PACRIM)*, pp. 1–6. IEEE.

Osanaiye, O. Chen, S. Yan, Z. Lu, R. Choo, K.-K. R. and Dlodlo, M. (2017). From cloud to fog computing: A review and a conceptual live VM migration framework. *IEEE Access*, vol. 5, pp. 8284–8300.

Pahl, C. (2015). Containerization and the paas cloud. *IEEE Cloud Computing*, vol. 2, pp. 24–31.

Parés, F. Gasulla, D. G. Vilalta, A. Moreno, J. Ayguadé, E. Labarta, J. Cortés, U. and Suzumura, T. (2017). Fluid communities: A competitive, scalable and diverse community detection algorithm. In: *International Conference on Complex Networks and their Applications*, pp. 229–240. Springer.

Patel, M. Naughton, B. Chan, C. Sprecher, N. Abeta, S. Neal, A. et al. (2014). Mobile-edge computing introductory technical white paper. White paper, ETSI.

Piraghaj, S. F. Dastjerdi, A. V. Calheiros, R. N. and Buyya, R. (2016). Containercloudsim: An environment for modeling and simulation of containers in cloud data centers. *Software: Practice and Experience*, vol. 47, pp. 505–521.

Rancher-Labs (2021). Run kubernetes everywhere. WebPage. https://rancher.com/. Date of access: 08/2021.

Reznik, A. et al. (2018). Cloud ran and mec: a perfect pairing. White paper, ETSI.

RNP (2021a). Ix.br. WebPage. https://ix.br/particip/sp. Date of access: 08/2021.

RNP (2021b). Rede ipê. WebPage. https://www.rnp.br/sistema-rnp/rede-ipe. Date of access: 08/2021.

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

Rost, P. Bernardos, C. J. Domenico, A. D. Girolamo, M. D. Lalam, M. Maeder, A. Sabella, D. and Wübben, D. (2014). Cloud technologies for flexible 5g radio access networks. *IEEE Communications Magazine*, vol. 52, pp. 68–76.

Russell, S. and Norvig, P. (2002). *Artificial intelligence: a modern approach*. Pearson.

Sabella, D. Vaillant, A. Kuure, P. Rauschenbach, U. and Giust, F. (2016). Mobile-edge computing architecture: The role of mec in the internet of things. *IEEE Consumer Electronics Magazine*, vol. 5, pp. 84–91.

Santoro, D. Zozin, D. Pizzolli, D. De Pellegrini, F. and Cretti, S. (2017). Foggy: A platform for workload orchestration in a fog computing environment. In: *IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, pp. 231–234. IEEE.

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

Satyanarayanan, M. Bahl, P. Caceres, R. and Davies, N. (2009). The case for vm-based cloudlets in mobile computing. *IEEE Pervasive Computing*, vol. 8, pp. 14–23.

Schiller, E. Nikaein, N. Kalogeiton, E. Gasparyan, M. and Braun, T. (2018). CDS-MEC: NFV/SDN-based application management for mec in 5g systems. *Computer Networks*, vol. 135, pp. 96–107.

Soltesz, S. Potzl, H. Fiuczynski, M. E. Bavier, A. and Peterson, L. (2007). Container-based operating system virtualization. *ACM SIGOPS Operating Systems Review*, vol. 41, pp. 275.

Taleb, T. Samdanis, K. Mada, B. Flinck, H. Dutta, S. and Sabella, D. (2017). On multi-access edge computing: A survey of the emerging 5g network edge cloud architecture and orchestration. *IEEE Communications Surveys & Tutorials*, vol. 19, pp. 1657–1681.

Topcuoglu, H. Hariri, S. and Wu, M.-Y. (2002). Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE Transactions on Parallel and Distributed Systems*, vol. 13, pp. 260–274.

Tosatto, A. Ruiu, P. and Attanasio, A. (2015). Container-based orchestration in cloud: State of the art and challenges. In: *Ninth International Conference on Complex, Intelligent, and Software Intensive Systems*, pp. 70–75. IEEE.

Truyen, E. Landuyt, D. V. Reniers, V. Rafique, A. Lagaisse, B. and Joosen, W. (2016). Towards a container-based architecture for multi-tenant SaaS applications. In: *15th International Workshop on Adaptive and Reflective Middleware - ARM*, pp. 1–6. ACM Press.

Uber (2021). Kraken - p2p-powered docker registry. WebPage. https://github.com/uber/kraken. Date of access: 08/2021.

Vaquero, L. M. and Rodero-Merino, L. (2014). Finding your way in the fog: Towards a comprehensive definition of fog computing. *SIGCOMM Computer Communication Review*, vol. 44, pp. 27–32.

Varga, A. (2010). *OMNeT++*, chap. 1, pp. 35–59. Springer Berlin Heidelberg, Berlin, Heidelberg.

Varghese, B. Reano, C. and Silla, F. (2018a). Accelerator virtualization in fog computing: Moving from the cloud to the edge. *IEEE Cloud Computing*, vol. 5, pp. 28–37.

Varghese, B. Villari, M. Rana, O. James, P. Shah, T. Fazio, M. and Ranjan, R. (2018b). Realizing edge marketplaces: Challenges and opportunities. *IEEE Cloud Computing*, vol. 5, pp. 9–20.

Varghese, B. Wang, N. Barbhuiya, S. Kilpatrick, P. and Nikolopoulos, D. S. (2016). Challenges and opportunities in edge computing. In: *IEEE International Conference on Smart Cloud (SmartCloud)*, pp. 20–26. IEEE.

Velasquez, K. Abreu, D. P. Assis, M. R. M. Senna, C. Aranha, D. F. Bittencourt, L. F. Laranjeiro, N. Curado, M. Vieira, M. Monteiro, E. and Madeira, E. (2018). Fog orchestration for the internet of everything: state-of-the-art and research challenges. *Journal of Internet Services and Applications*, vol. 9, pp. 1–23.

Wang, K. Xu, F. Ding, Y. and Xing, L. C. (2021). Kubeedge.io. WebPage. https://kubeedge.io/en. Date of access: 08/2021.

Wang, N. Matthaiou, M. Nikolopoulos, D. S. and Varghese, B. (2018). Dyverse: Dynamic vertical scaling in multi-tenant edge environments. WebPage. arXiv. http //arxiv.org/pdf/1810.04608v1:PDF.

Wobker, C. Seitz, A. Mueller, H. and Bruegge, B. (2018). Fogernetes: Deployment and management of fog computing applications. In: *NOMS - IEEE/IFIP Network Operations and Management Symposium*, pp. 1–7. IEEE.

Wong, W. Zavodovski, A. Zhou, P. and Kangasharju, J. (2019). Container deployment strategy for edge networking. In: *4th Workshop on Middleware for Edge Clouds and Cloudlets*, MECC '19, pp. 1–6, New York, NY, USA. ACM Press, Association for Computing Machinery.

Wu, J. Zhang, Z. Hong, Y. and Wen, Y. (2015). Cloud radio access network (c-ran): a primer. *IEEE Network*, vol. 29, pp. 35–41.

Xavier, B. Ferreto, T. and Jersak, L. (2016). Time provisioning evaluation of KVM, docker and unikernels in a cloud platform. In: *16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, pp. 277–280. IEEE.

Yao, J. and Ansari, N. (2018). Reliability-aware fog resource provisioning for deadline-driven iot services. In: *IEEE Global Communications Conference (GLOBECOM)*, pp. 1–6. IEEE.

Yousefpour, A. Fung, C. Nguyen, T. Kadiyala, K. Jalali, F. Niakanlahiji, A. Kong, J. and Jue, J. P. (2019). All one needs to know about fog computing and related edge computing paradigms: A complete survey. *Journal of Systems Architecture*, vol. 98, pp. 289–330.

Zambrano-Vega, C. Nebro, A. J. García-Nieto, J. and Aldana-Montes, J. F. (2017). Comparing multi-objective metaheuristics for solving a three-objective formulation of multiple sequence alignment. *Progress in Artificial Intelligence*, vol. 6, pp. 195–210.

Zhang, Y. Niu, K. Wu, W. Li, K. and Zhou, Y. (2018). Speeding up VM startup by cooperative VM image caching. *IEEE Transactions on Cloud Computing*, vol. 9, pp. 360–371.

Zheng, C. Rupprecht, L. Tarasov, V. Thain, D. Mohamed, M. Skourtis, D. Warke, A. S. and Hildebrand, D. (2018). Wharf: Sharing docker images in a distributed file system. In: *ACM Symposium on Cloud Computing*, SoCC ’18, pp. 174–185, New York, NY, USA. Association for Computing Machinery.

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