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

LUCAS ROGES DE ARAUJO

DYNAMIC PROVISIONING OF CONTAINER REGISTRIES IN EDGE COMPUTING INFRASTRUCTURES

Porto Alegre
2024

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

DYNAMIC PROVISIONING OF CONTAINER REGISTRIES IN EDGE COMPUTING INFRASTRUCTURES

LUCAS ROGES DE ARAUJO

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

Advisor: Prof. Dr. Tiago Coelho Ferreto

Porto Alegre
2024

Ficha Catalográfica

A663d Araujo, Lucas Roges de

Dynamic Provisioning of Container Registries in Edge Computing Infrastructures / Lucas Roges de Araujo. – 2024.
68 p.
Dissertação (Mestrado) – Programa de Pós-Graduação em Ciência da Computação, PUCRS.

Orientador: Prof. Dr. Tiago Coelho Ferreto.

1. Container Registry. 2. Edge Computing. 3. Docker. 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

LUCAS ROGES DE ARAUJO

DYNAMIC PROVISIONING OF CONTAINER REGISTRIES IN EDGE COMPUTING INFRASTRUCTURES

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

Sanctioned on March 26th, 2024.

COMMITTEE MEMBERS:

Prof. Dr. Márcio Bastos Castro (PPGCC/UFSC)

Prof. Dr. Dalvan Jair Griebler (PPGCC/PUCRS)

Prof. Dr. Tiago Coelho Ferreto (PPGCC/PUCRS - Advisor)

ACKNOWLEDGMENTS

This work was supported by the PDTI Program, funded by Dell Computadores do Brasil Ltda (Law 8.248 / 91). The authors acknowledge the High-Performance Computing Laboratory of the Pontifical Catholic University of Rio Grande do Sul (LAD-IDEIA/PUCRS) for providing support and technological resources for this project.

PROVISIONAMENTO DINÂMICO DE REGISTROS DE CONTÊINER EM INFRAESTRUTURAS DE COMPUTAÇÃO DE BORDA

RESUMO

A proliferação de dispositivos móveis e sensores tem provocado o desenvolvimento de aplicações sensíveis à latência e com uso intensivo de recursos. Enquanto os dispositivos e sensores gerando os dados podem ter capacidade limitada de processamento e armazenamento, as infraestruturas de computação de nuvem oferecem a escalabilidade necessária para processar essas demandas. No entanto, a distância entre os usuários finais e os centros de dados de nuvem introduzem uma sobrecarga de comunicação. Nesse sentido, computação de borda surge para estender infraestruturas de nuvem para localidades próximas do usuário final. A proximidade intrínseca do paradigma melhora a latência e largura de banda para aplicações com demandas estritas. Além disso, a associação com virtualização baseada em contêineres melhora a qualidade de serviço e qualidade de experiência. O aspecto de leveza dessa tecnologia fornece um rápida provisionamento da aplicação e baixa sobrecarga de recursos. No entanto, provisionar aplicações baseadas em contêineres sofre com uma sobrecarga significativa enquanto o conteúdo é baixado dos registros de contêiner. Ainda que diversos autores tenham proposto o uso de registros distribuídos para enfrentar esse problema, suas técnicas focam exclusivamente no tempo de provisionamento, geralmente negligenciando métricas chave para infraestruturas de computação de borda, como latência e uso de recursos. Além disso, outros autores adotam técnicas complexas, como migração de registros, a qual pode afetar a utilização de rede. Em resposta a este cenário, nós propomos o provisionamento dinâmico de registros de contêiner com duas estratégias (LMDyn e MODyn) que alocam registros de contêiner em servidores de borda baseados na análise da infraestrutura e sem necessitar de migração de recursos. Nossa avaliação mostra reduções significativas no uso de recursos utilizando LMDyn e um *trade-off* balanceado entre tempo de provisionamento, latência e uso de recursos com MODyn.

Palavras-Chave: registro de contêiner, computação de borda, Docker.

DYNAMIC PROVISIONING OF CONTAINER REGISTRIES IN EDGE COMPUTING INFRASTRUCTURES

ABSTRACT

The proliferation of mobile devices and sensors has provoked the development of latency-sensitive and resource-intensive applications. Although devices and sensors that generate data might have limited processing and storage capacity, cloud computing infrastructures offer the necessary scalability to process these demands. However, the distance between end users and cloud data centers introduces communication overhead. In this sense, edge computing emerges to extend cloud infrastructures to locations near the end user. The inherited proximity of the paradigm improves latency and bandwidth for applications with strict demands. Additionally, the association with container-based virtualization improves Quality of Service (QoS) and Quality of Experience (QoE). The lightweight aspect of this technology provides faster application provisioning and lower resource overhead. However, provisioning container-based applications suffers from a significant overhead while downloading content from container registries. Although several authors propose using distributed registries to tackle this problem, their techniques focus exclusively on provisioning time, often neglecting key metrics for edge computing infrastructures, such as latency or resource usage. In addition, other authors adopt complex techniques, such as registry migration, which can affect network utilization. In response to this scenario, we propose dynamic provisioning of container registries with two strategies (LMDyn and MODyn) that allocate container registries to edge servers based on infrastructure analysis without requiring resource migration. Our evaluation demonstrates significant reductions in resource usage with LMDyn and a balanced trade-off between provisioning time, latency, and resource usage with MODyn.

Keywords: container registry, edge computing, Docker.

LIST OF FIGURES

LIST OF TABLES

LIST OF ALGORITHMS

CONTENTS

1. INTRODUCTION

The deployment of 5G technology networks accelerated the growth of edge computing due to the low latency and high bandwidth capabilities that these networks offer [50]. Besides the 5G networks, the increasing number of Internet of Things (IoT) and mobile devices also impact edge computing's growth. Although the well-established cloud computing paradigm can allocate computing resources and services for the IoT paradigm over the Internet, it has a high service response time, making this interaction infeasible for supporting interactive real-time services [39]. The resource limitations of mobile hardware lead to the use of cloud computing for its demands. However, high wide-area network (WAN) latencies from mobile devices to centralized cloud data centers are a fundamental obstacle to adopt cloud computing in this matter [43], especially for real-time applications.

In this sense, some authors envision the utilization of decentralized resources to provide the necessary support for demands originating from devices with limited capacity that cannot rely on centralized cloud data centers to meet these demands [10, 42, 43]. Although this decentralization had different goals in the early stages (e.g., mobile computing [43] or IoT [10]), Satyanarayanan [42] recognizes the importance of decentralized edge infrastructures for both use cases. The author highlights that the proximity inherited by decentralization has a fundamental role in achieving lower end-to-end latency, minimizing the cumulative bandwidth into the cloud, avoiding releasing sensitive data to the cloud, and temporarily serving as a fallback service. However, decentralization of resources poses new challenges.

While providing the necessary Quality of Service (QoS) and Quality of Experience (QoE) to the end-users, edge computing infrastructures must deal with their location's inherited challenges, such as resource limitation and user mobility. Such challenges occur because the edge infrastructure is not as abundant in computing and storage resources as cloud data centers, and because of the proximity to the end users. For positioning edge infrastructures near the end users, the space in these locations imposes a significant constraint. Furthermore, the dynamic aspect introduced by user mobility helps to degrade the quality of application service due to the varying conditions of the network, in addition to being the main reason for disconnection from the edge server [1]. In the face of these challenges, several works propose resource management strategies to cope with the aspects of edge infrastructures [46], but there are still obstacles to overcome. In addition to these strategies, some existing technologies also help to address inherited challenges.

Among the supporting technologies, container-based virtualization is one of the most notable. Although Satyanarayanan et al. [43] initially pictured virtual machines (VMs) as the basis for edge infrastructure, several works highlight the compatibility between edge computing infrastructures and containers [20, 24, 36], especially for devices

up to medium resource capability (e.g., cloudlets) [33]. Currently, Docker [35] is the most popular container platform, representing applications as container images. Although advanced users can manually build container images, the most common practice is obtaining the container images from container registries, entities responsible for storing and distributing the container images. However, while obtaining a container image from a registry, the download process could represent a bottleneck for the end user, considering that it takes about 76% of the container deployment process [23].

In this context, several works propose modifications to the container registry to improve the download process. Although part of these works includes intrusive modifications to the daemon, another part focuses on distributing the demands of this entity. This latter approach includes many strategies, such as P2P-based, container image placement, container registry placement, and container registry migration. To the best of our knowledge, all strategies improve the application provisioning time by accelerating the container image download. However, only a few of these strategies handle other relevant metrics for edge computing users: latency and resource usage. However, the only strategy that targets all these other metrics (registry migration) employs a complex process to distribute the container registries, which can affect other operations in the infrastructure.

Considering this scenario, we advocate that dynamically provisioning and deprovisioning container registries in edge computing might be an alternative to deal with the bottlenecks of provisioning container-based applications in edge computing infrastructures. More specifically, we aim to offer an adequate trade-off based on latency, provisioning time, and resource usage. To this matter, we propose two algorithms, LMDyn and MODyn, to periodically select a set of edge servers to allocate container registries without requiring the migration of a massive set of container images between container registries. LMDyn focuses on minimizing the number of container registries to improve latency and resource usage compared to the P2P-based strategy. MODyn is an improved version that considers the provisioning time as an important metric, and, for this matter, it aims to reach a trade-off between the target metrics while providing registries to cope with a target number of replicas for each container image in the infrastructure. While the LMDyn algorithm reduces latency and resource usage compared to the P2P-based strategy, the MODyn algorithm offers a competitive provisioning time with a relative reduction in resource usage and similar latency to LMDyn and other non-P2P-based strategies.

We organize the remainder of this work as follows. Chapter 2 contains the background of this work, with details on Edge Computing (Section 2.1) and Container-based Virtualization (Section 2.2). Chapter 3 presents the existing strategies for provisioning container images with distributed container registries. Chapter 4 includes the contributions of this work, which are our system model (Section 4.1) and the proposed algorithms (Section 4.2). Chapter 5 comprises the experimental results of our algorithms against

the baseline approaches extracted from the literature. Lastly, Chapter 6 concludes the manuscript with a summary of our work and future work perspectives.

2. BACKGROUND

In this chapter, we will provide the necessary context and background information for our research. First, in Section 2.1, we introduce the edge computing paradigm, an extension of cloud computing to cope with emerging application requirements. We discuss the advantages and disadvantages of using edge infrastructures and highlight the relevant features of this paradigm for our work. Then, in Section 2.2, we dive deep into containers, a lightweight technology to virtualize computational resources. We compare containers and virtual machines (VMs) slightly and focus on presenting the Docker workflow, which is the baseline of our proposal.

2.1 Edge Computing

Edge computing is an emerging paradigm that considers the existence of substantial computing and storage resources close to mobile devices or sensors [42]. The appearance of this paradigm is directly related to some constraints observed when relying on cloud infrastructures to process data from the above-mentioned and other related devices. The well-established cloud computing paradigm, known for enabling on-demand access to shared resources [34], has a centralized aspect, allowing plenty of computing and storage resources in specific locations. However, this centralization incurs long distances between end-user devices and cloud data centers, which hurts communication between these entities and impacts the dependency of resource-constrained devices on cloud infrastructures. However, end-user devices often have hardware limitations, making dependence on other resources necessary [43]. In this sense, edge infrastructures aim to extend cloud functionalities to the proximity of end-user devices, which comprises a three-tier model of computing presented by Satyanarayanan et al. [45] (Figure 2.1).

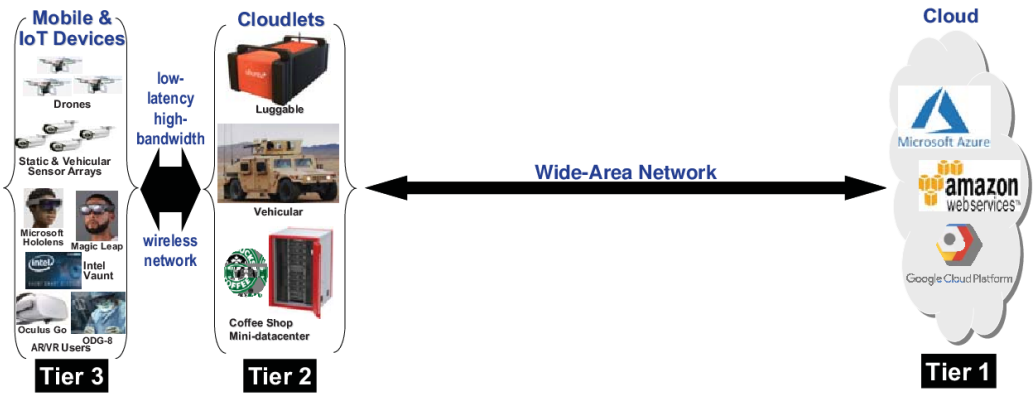


Figure 2.1: Three-tier model of computing (Satyanarayanan et al. [45]).

With the consolidation and wide spread of mobile devices and sensors, part of it associated with IoT's emergence, the proximity between end-user devices and significant computing and storage resources became more important than ever. Satyanarayanan et al. [42] listed at least four distinct ways in which the proximity of edge infrastructures helps, including (i) highly responsive cloud services, (ii) scalability via edge analytics, (iii) privacy policy enforcement, and (iv) masking cloud outages. High responsiveness is a direct consequence of proximity, as the communication between the end user does not rely on WAN connections. Scalability via edge analytics is achieved by partially processing a service on edge premises to decrease the data sent to the cloud. Privacy policy enforcement regards the ability to send sensitive data for processing on trusted edge servers instead of sending these data to unsafe cloud servers. Lastly, masking cloud outages means that cloudlets can substitute cloud infrastructures during failures and service outages. Although this last benefit is often specific to hostile environments (e.g., military operations), the previous ones have broad applicability to the applications and services currently deployed on remote servers in the cloud.

Along with proximity's benefits, using edge computing poses new challenges for managing and provisioning resources to the demands of edge computing infrastructures. The location of these infrastructures (e.g., coupled to base stations) limits the availability of resources at a single point. Carvalho et al. [12] emphasize that this limitation is not exclusive to computing resources, but also to storage capacity, which suffers from a higher degree of limitation, as highlighted in a comparative table between cloud and edge paradigms. Considering that the inherited resource limitation of edge infrastructures requires resource provisioning strategies, Shakarami et al. [46] conducted a systematic review to identify and classify current approaches for managing this issue. The authors present five different mechanisms (heuristic/metaheuristic-based, framework-based, model-based, machine learning-based, and game theory-based) with multiple internal classifications each. These mechanisms cover different performance metrics (e.g., latency, cost, and energy) and case studies (e.g., IoT, smart cities, healthcare), among other scopes. However, the resource provisioning process still has open issues for future research in addition to resource limitations.

Some authors highlight the challenge of maintaining an uncompromised QoS and QoE among the open issues. Although the authors consider latency and QoS synonyms [27], other metrics play a significant role in QoS and QoE. In this context, Shi et al. [47] discuss multiple essential optimization metrics, such as bandwidth, cost, and energy, to choose optimal allocation strategies and avoid QoS and QoE issues. Regarding prevention strategies, Vargese et al. [54] point out the need to avoid resource overload by flexibly partitioning and scheduling tasks based on usage information to maintain adequate levels of QoS and QoE. In addition to ensuring uncompromised QoS and QoE, managing user mobility is another challenge in edge computing. This challenge might be related

to QoS and QoE, since user mobility could affect the distance between users and their applications [12]. In addition, it makes the resource provisioning process more complex due to the dynamic aspect it introduces in the environment, requiring application migration or reallocation.

Independently of the advantages and challenges of using edge computing infrastructures, the paradigm has an extensive list of application use cases. Satyanarayanan et al. [45] propose a taxonomy for edge computing applications based on the three-tier computing model depicted in Figure 2.1. We present this taxonomy in Table 2.1, with "P" indicating the primary execution site and "O" indicating the optional (i.e., non-critical) use of that tier's resources. In this taxonomy, device-only application developers can use the current situation as an opportunity to refine these applications and rely on edge computing, which is currently not the case. With the optional use of edge computing, cloud-native and device-native applications use the edge infrastructure to process additional features of the application or to improve performance without a critical dependency on edge infrastructures. Lastly, edge-native applications make significant use of edge infrastructures and, thus, are custom-designed to use at least one of edge computing's features mentioned in the second paragraph of the current section. These applications are the most relevant for the context of our work and include a variety of use cases, which we discuss below.

Table 2.1: Taxonomy of application types (Satyanarayanan et al. [45]).

	Tier-3	**Tier-2**	**Tier-1**
Device-only	P		
Edge-accelerated, cloud-native		O	P
Edge-enhanced, device-native	P	O	
Edge-native	P	P	

Edge-native applications include what would be considered the "killer-apps" for consolidating edge computing: human cognitive augmentation/assistance. Many of these applications require the use of a wearable device with multiple sensors (e.g., video, audio, accelerometer, and gyroscope) to capture and stream data and use ML-based algorithms to generate output for the wearable user [44]. Additionally, these applications can scale to a smart-city-level of cognitive assistance [40], which uses sensors spread throughout the city to collaborate with cyclists, people with disabilities, and emergency responses. Furthermore, edge computing can offer support for other IoT applications [22] (e.g., smart industry, smart agriculture [58], and e-health) and Internet of Vehicles (IoV) applications [11]. The use of artificial intelligence (AI) and deep learning (DL) methods is also common in edge applications, with many opportunities to improve this combination of technologies [55, 57]. Generally, edge computing applications are related to IoT or AI, and much of the time to both [48]. The association between these three domains (i.e., edge computing, IoT, and AI) creates a powerful combination that relies on IoT devices to

collect data and AI algorithms to process the data with low latency and enough resources provided by edge infrastructures.

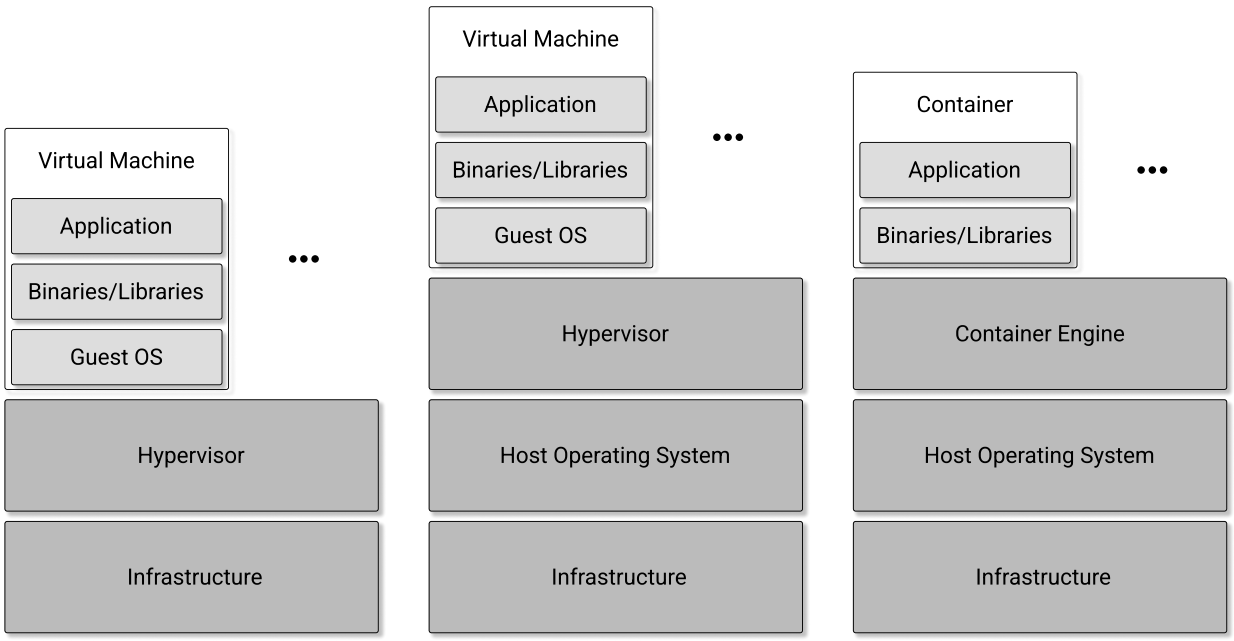
To enable the deployment of applications in edge computing environments, container-based virtualization is an attractive technology [24]. The lightweight aspect of this virtualization technique copes with the limited resources available and the strict requirements of edge computing environments. Morabito et al. [36] evaluate containers and unikernels (i.e., lightweight virtualization techniques) for IoT use cases using edge computing infrastructures. In addition to concluding that both techniques offer relevant support for the edge environment, the authors highlight the advent of Docker as the platform that made containers very popular. In addition, Gillani and Lee [20] compared Docker containers with Linux VMs for service migration in edge computing. The authors show that the containers perform better than Kernel-based Virtual Machines (KVM), reinforcing the advantage of relying on this technology in edge infrastructures. Together, these works show the importance of container-based virtualization to improve the use of edge computing resources.

2.2 Container-based Virtualization

The type of virtualization used by the operating system determines the functionality of containers. They depend on the operating system kernel to create isolated spaces with specific dependencies, environment variables, libraries, and other resources. In comparison to virtual machines, containers are considered lighter because they can be deployed over an existing operating system without requiring the boot of a new system within each. Figure 2.2 depicts this architectural difference, considering both types of hypervisors that virtual machines can use.

Historically, according to Bernstein [9], some previous developments inspired the deployment of this technology. The oldest of these developments is the *chroot*[1] command from Unix, which changes the root directory of the current process to a new given directory. It was launched in 1979 and enabled the isolation of the file system from a given process and its subprocesses. Later significant developments are *jails* [18], launched in 1998, and *zones* [49], launched in 2004. Jails extended the isolation ability from the chroot command to other resources beyond the file system, such as the networking subsystem. Zones enabled essential features in the operating system-level virtualization domain, such as snapshot and ZFS clone. The latest container technologies (e.g., Linux Containers [31] and Docker [35]) leverage two Linux process resource management solutions: *namespaces* and *cgroups*. Namespaces[2] wraps a global system resource (e.g., user, process ID,

[1] https://www.freebsd.org/cgi/man.cgi?query=chroot&sektion=2&format=html
[2] https://man7.org/linux/man-pages/man7/namespaces.7.html



(a) Type 1 Hypervisor (b) Type 2 Hypervisor (c) Containers

Figure 2.2: Architectural comparison between virtualization types.

network, and mount) in an abstraction that resembles an isolated instance of the given resource. Cgroups [28] allow limiting the access of a group of processes to the resources available in the system (e.g., CPU, disk, memory, and network).

Nowadays, Docker is the most popular platform for deploying applications and services through container-based virtualization. Merkel [35] states that Docker unifies existing technologies (e.g., namespaces, cgroups, and copy-on-write filesystem) to create a tool greater than the sum of its parts. With a significant focus on the software development process, Docker facilitates the maintenance and deployment of multiple environments (e.g., development, test, and production). In addition, it is very suitable for the microservice architecture, which separates multiple services from an application. Docker allows building and deploying each service from a microservice-based application with its dependencies and without conflicting with other services. Besides, the usage of Docker containers allows scaling these services independently according to the application's current demands with an insignificant impact on user performance.

Another advantage of Docker is the layered structure of the Docker images. After creation, an image is a read-only template based on a Dockerfile [3], a text document with proper syntax to indicate the steps to build an image. Figure 2.3 depicts this structure and highlights that the Docker image consists of a set of commands executed and transformed into Docker layers. This structure enables different images to share the same content as long as the order of previous layers is the same. This layer-sharing feature is an opportunity to optimize container images and improve the use of storage resources, which is

[3] https://docs.docker.com/engine/reference/builder/

essential in environments with resource limitations. Once the image is available to use, the last step is to turn it into a container, a process that adds a writable layer on top of the image's read-only layers. This writable layer stores all the modifications made while the containerized application is executed. Docker containers can also take advantage of multiple resources that Docker offers, such as custom networks to communicate between containers and volumes to persist content outside of the container.

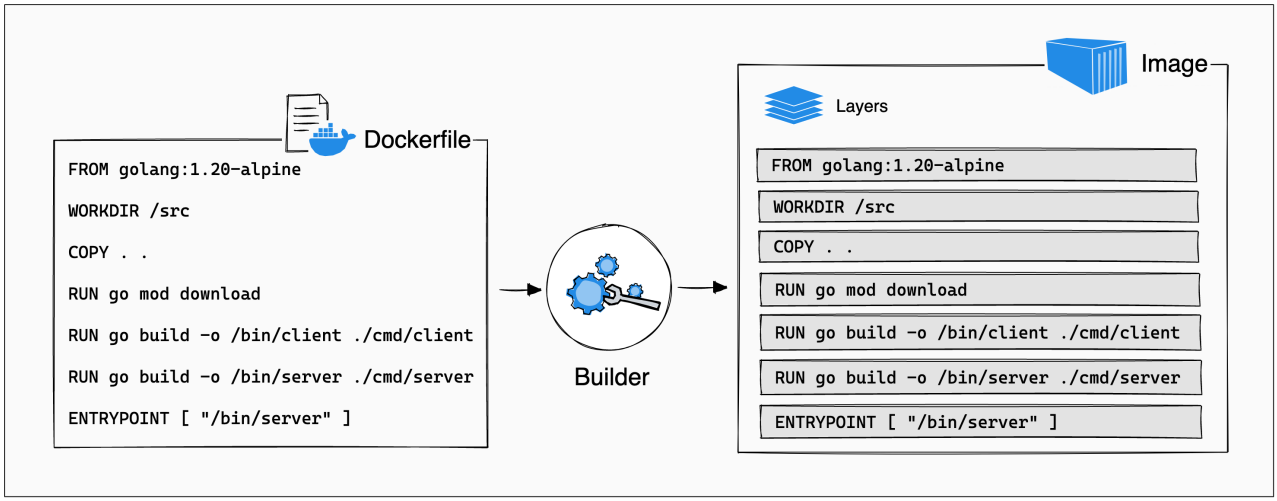


Figure 2.3: Docker layered structure of images [16].

To interact with the above-mentioned objects, Docker has multiple components making up its architecture. Figure 2.4 shows an overview of the Docker architecture consisting of some components (Docker client, Docker daemon, and Docker registry) communicating to manipulate images and containers.

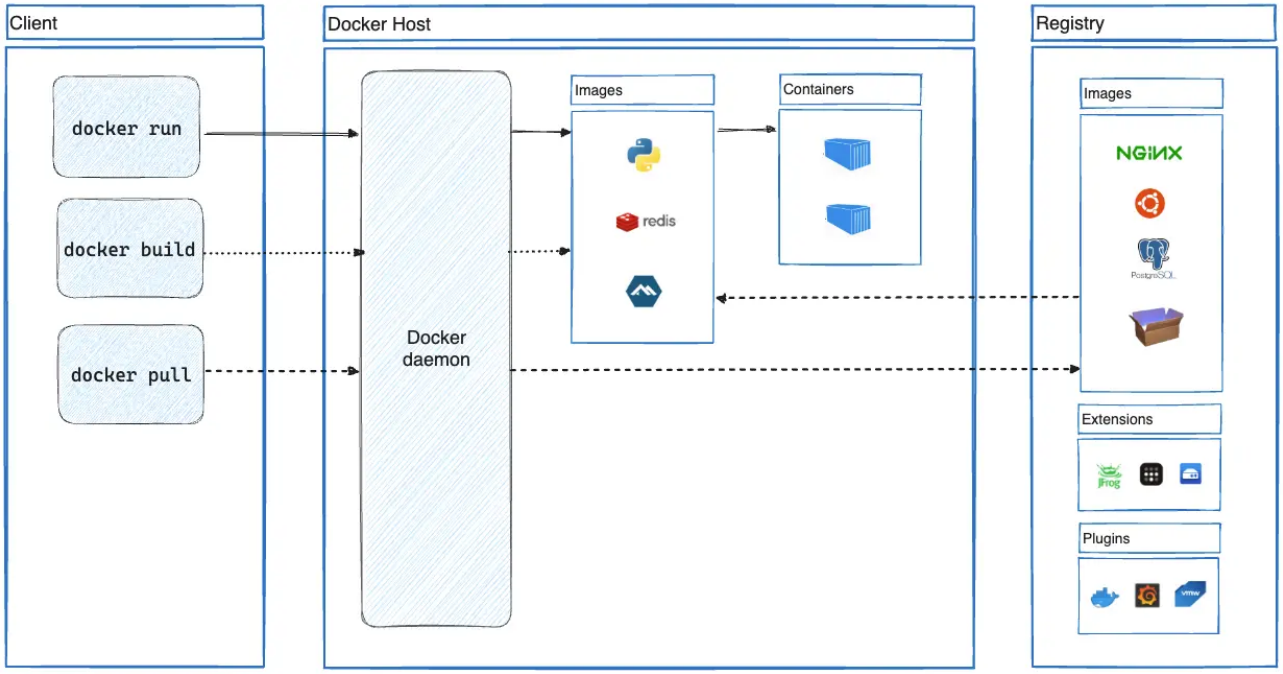


Figure 2.4: Docker architecture [15].

In this architecture, the Docker client is the primary way users interact with Docker. This component supports several commands that trigger specific actions at the Docker daemon, the core component of Docker. This communication occurs via API, so the user might rely directly on API requests to interact with the daemon following the documentation [4]. However, the client wraps this API and facilitates the interaction with Docker. Meanwhile, the Docker daemon receives these requests and acts upon them internally or by communicating with other components (e.g., the registry). More specifically, the daemon interacts with the Docker registry if the request is to download or run a locally unavailable container image. The registry is a repository that stores container images and distributes them upon download requests. It also receives requests for upload to store images created locally. Users can take advantage of public registries (e.g., Docker Hub[5]) for their demands, or they can deploy a private registry by using the official registry container image[6]. The public registry offers easy access to container images, but private registries provide more security and control over access to their content.

Figure 2.4 also includes Docker's main workflows, which we describe below. Users can obtain a container image through the *build* command, which allows the user to build a new image locally based on a custom Dockerfile, and through the *pull* command, which searches in a target container registry for the image to download it. While the first approach executes the commands to create the image and store it on the disk, the second approach verifies which layers are unavailable in the local storage and downloads them in gzip format to decompress them later. The *run* command adds a writable layer over the read-only layers of the container image component and starts the containerized application. If an image instantiated in this command is locally unavailable, the *pull* command is indirectly triggered to search for the given container image in the registry. Lastly, the user can use *push* to send a locally built image to a registry so that other users can download it.

Analyzing the container deployment workflow, Harter et al. [23] identify a significant overhead in the pull process download stage, which accounts for about 76% of the command execution time. The authors enhance the importance of mechanisms to decrease the time spent in this stage, which relies directly on the container registry. Although they propose a new storage driver to tackle this overhead, other works propose less intrusive modifications to the container image provisioning process to improve the download stage. In the next chapter, we detail the approaches that consider distributing the demands of the container registry to address this problem.

[4] https://docs.docker.com/engine/api/v1.44/
[5] https://hub.docker.com/
[6] https://hub.docker.com/_/registry

3. RELATED WORK

This chapter presents solutions focused on enhancing the container image provisioning operation through distributed container registries (i.e., multiple compute nodes acting as container registries). With the popularization of container-based virtualization and Docker, several works attempt to optimize the container image distribution time by using a set of distributed container registries instead of using it as a single centralized entity. Other works have used different approaches, such as registry design optimizations [5, 13, 23] and distributed file systems [2, 17, 59]. However, we do not detail these approaches, as we understand that these works are more intrusive to the Docker daemon, unlike the architectures and algorithms we present in the following sections.

We separate the works into two sections: Section 3.1 contains strategies aimed at cloud computing environments, and Section 3.2 contains strategies aimed at edge computing environments. We highlight the relevant aspects of the proposed architecture or algorithm in each selected work. In Section 3.3, we conclude the chapter by summarizing the related works in a comparative table and presenting the limitations that led to the development of our proposal.

3.1 Distributed Registries in the Cloud Computing Paradigm

Cloud infrastructures were the first target of improvements to enhance the container image provisioning process with distributed registries. Most solutions for these environments employ peer-to-peer (P2P) protocols, such as BitTorrent (BT), to distribute the registry's demands. These solutions are particularly suitable for cloud data centers, given their availability of dedicated resources for important tasks of the protocol, such as the control of communication between seeds and peers by the BT tracker.

Liang et al. [30] propose HDID, a hybrid Docker image distribution system for data centers tool that adaptively uses the BT protocol or the default container registry to provision container images. Before designing the solution, the authors conducted a study to understand the features of the top 29 most commonly used images in Docker Hub at that time. In this study, the authors analyze aspects such as the image/layer size distribution and layer sharing to conclude that BT is a natural fit to improve the downloading speed of container images. In addition, the authors understand that the original registry is still convenient, especially for provisioning small layers, due to the overhead of making a torrent for these pieces of container images. Based on these insights, they present the architecture shown in Figure 3.1.

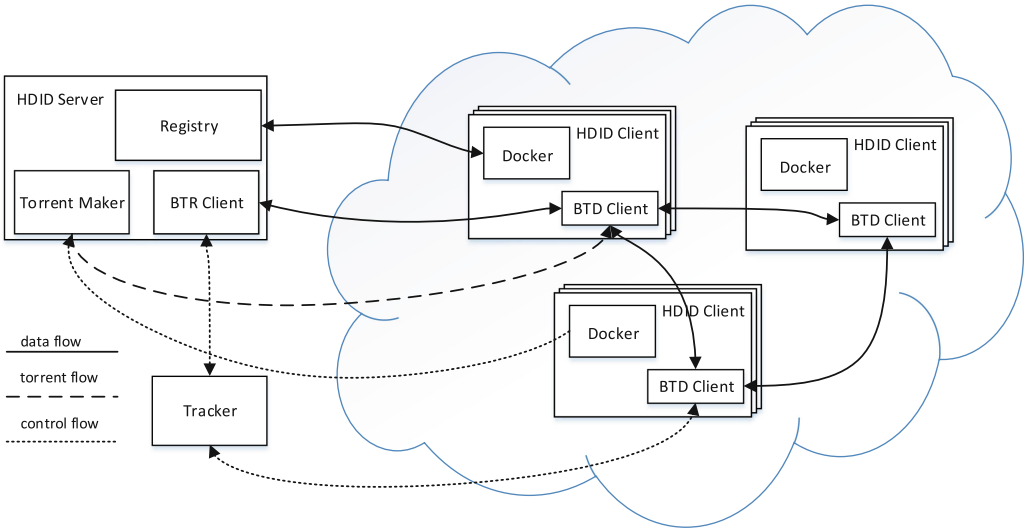


Figure 3.1: HDID architecture (Liang et al. [30]).

The three main components of HDID's architecture are the HDID server, the HDID client, and the tracker. The HDID server is responsible for provisioning small layers through the registry component, making torrents for large layers (sizes larger than 15MB), and enabling the provisioning through BT. On the other hand, the HDID client has a Docker daemon to receive content from the Docker registry and a BitTorrent-Docker client that downloads and uploads torrents. Lastly, the tracker communicates with the BTR and BTD clients to update them on the network's peers and seeds.

Similarly to the previous work, Kangjin et al. [26] claim that the container registry is prone to become a bottleneck for large-scale container deployments. For this matter, the authors propose FID, a faster image distribution system for the Docker platform. Based on the goal of reducing image distribution time in high concurrency scenarios, FID introduces concerns about scalability and fault tolerance. To match these concerns, the architecture of FID, shown in Figure 3.2, considers the deployment of multiple BT trackers and P2P registry servers supported by a back-end storage.

The server and BT tracker components have similar functionalities to their correspondent component of HDID, whereas clients in FID architecture have an agent to handle the image downloading through BT. The agent emerges to avoid modifying Docker's source code and has two working modes: *load mode* and *proxy mode*. In load mode, the user sends requests to an API instead of executing the Docker pull command. Each request triggers the agent to obtain the image manifest and then the blob identifier of the missing layers, which enables the agent to get the corresponding torrent files. In proxy mode, the agent takes advantage of Docker's configurable proxy and acts only when it identifies a blob request from the data gathered in the requests. The latter mode turns out to be more lightweight since it acts only in part of the whole process.

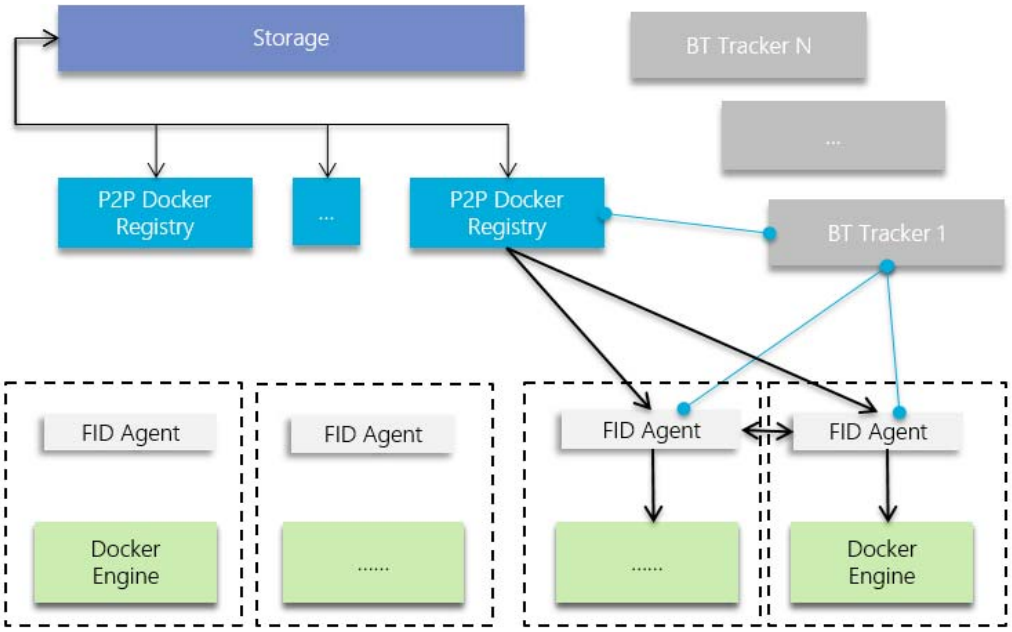


Figure 3.2: FID architecture (Kangjin et al. [26]).

Industry players have also proposed solutions to improve container image provisioning through P2P-based techniques. Kraken [53], proposed by Uber, and Dragonfly [4], from Alibaba, are open-source projects to accelerate image distribution using a P2P-based approach. A significant advantage of these solutions is that they are constantly receiving contributions and updates, as seen in their GitHub repositories[1,2]. Initially, Kraken employed a BT driver, but currently, the tool uses a custom implementation of a P2P driver to enhance the control over performance optimizations. Regarding Dragonfly, although it offers advantages over the original BT protocol, such as dynamic block size, Kraken claims to be more scalable since Dragonfly might have its throughput limited by one or a few nodes that coordinate the data transfer process.

Alongside P2P-based solutions, Nathan et al. [37] propose CoMICon, a cooperative management system for Docker container images. Although not directly based on any P2P protocol, CoMICon aims to create a pool of local Docker hosts that can share the container images among themselves. CoMICon enables lower container image provisioning time by allowing nodes to pull an image from multiple nodes and partially store, transfer, and delete container images. In addition, it addresses high availability requirements by placing multiple copies of the container images across the set of hosts configured with CoMICon. Figure 3.3 shows the architecture of the solution.

The CoMICon system components act upon the receipt of three different inputs. Information about a new node is the first input type, which the Node Registration handles by adding the information (e.g., IP address, a list of stored layers) to a database. Data of

[1]https://github.com/uber/kraken
[2]https://github.com/dragonflyoss/Dragonfly2

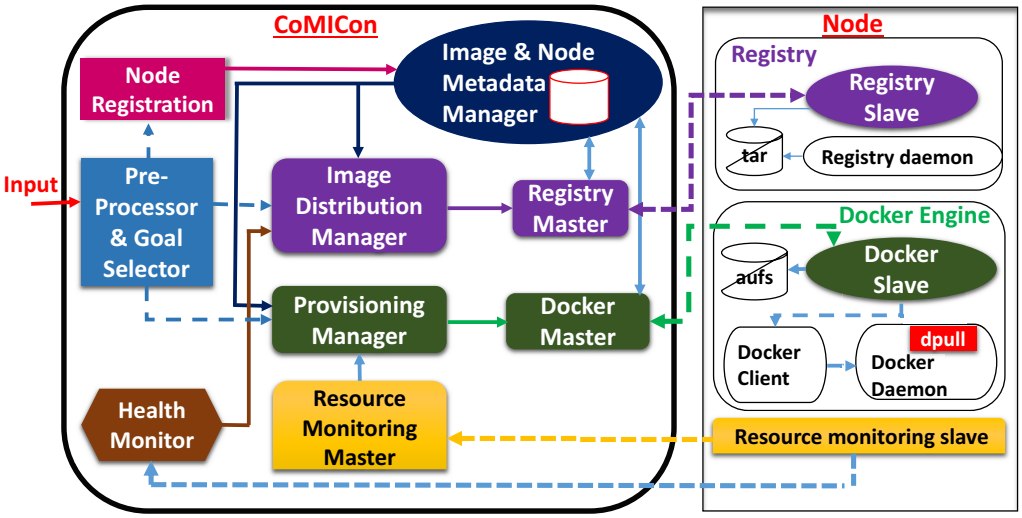


Figure 3.3: CoMICon system and node architecture (Nathan et al. [37]).

new container images is the second input type, and the Image Distribution Manager handles this input by mapping the layers to the nodes based on a custom heuristic algorithm. Lastly, applications for provisioning in the infrastructure are the third input type, and the Provision Manager handles them by trying to maximize the cache usage and accomplish application resource requirements when allocating these applications.

Regarding the node architecture, we highlight that the main difference between CoMICon and the P2P-based solutions is that the registry and the Docker engine have separate storages, so the registry cannot distribute all the images available in its node. Thus, a node only distributes images initially assigned to the node's registry storage. Among open research problems, the authors highlight the need to adapt the system to environments with varying network conditions, such as edge infrastructures.

3.2 Distributed Registries in the Edge Computing Paradigm

Edge computing infrastructures are commonly known for adopting container-based virtualization. Consequently, the image distribution process in these environments has also been the target of optimizations. However, the above-mentioned solutions focus on cloud data centers because they depend on resource-intensive dedicated services and stable network conditions. Thus, in the following paragraphs, we present different solutions to optimize the image distribution process through distributed registries, considering the constraints of edge computing infrastructures.

Gazzetti et al. [19] target the problem of getting redundant data (i.e., the same container images/layers) from a cloud-based container registry for multiple edge nodes within the same locality. The authors consider that nodes in the same locality have connections with lower latency and bandwidth costs compared to edge-cloud communica-

tion. In this sense, the authors propose the Stream Deployment (SD) approach based on P2P provisioning. This solution allows nodes in the same locality to share container images/layers instead of downloading redundant content from the cloud. Figure 3.4 depicts the SD architecture.

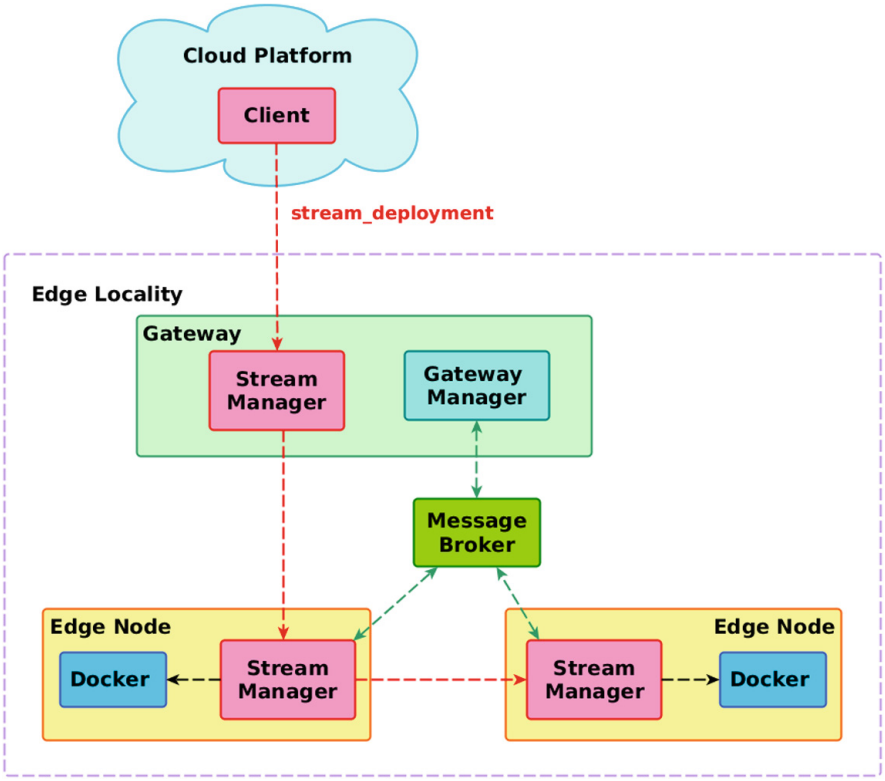


Figure 3.4: Stream Deployment architecture proposed by Gazzetti et al. [19].

In this architecture, the gateway node is responsible for pulling images from the cloud platform and distributing them to the remaining nodes, given their requests. Inside the gateway, there is also a manager responsible for keeping information about the current state of the nodes, so the stream manager only sends content to working nodes. The nodes within the locality receive the images through the stream manager, which communicates with the Docker daemon to make the images available to the end user. Meanwhile, there is a message broker for coordination between the nodes and the gateway.

Becker et al. [8] propose EdgePier, another P2P-based approach for provisioning container images. The authors claim that existing solutions generally use BT-based technologies to enable P2P image distribution. However, these technologies are unsuitable for edge computing environments due to the overhead of dedicated components (e.g., BT tracker) and the limited resources available in edge infrastructures. Therefore, the authors build an architecture based on the fully decentralized InterPlanetary File System (IPFS), which uses a distributed hash table to find the peers' addresses and hosted objects. This decentralized approach has the relevant advantage of not requiring dedicated components for P2P image provisioning. In addition, EdgePier aims to have a non-intrusive

deployment without adapting the container runtime. Figure 3.5 shows the workflow to pull an image using EdgePier.

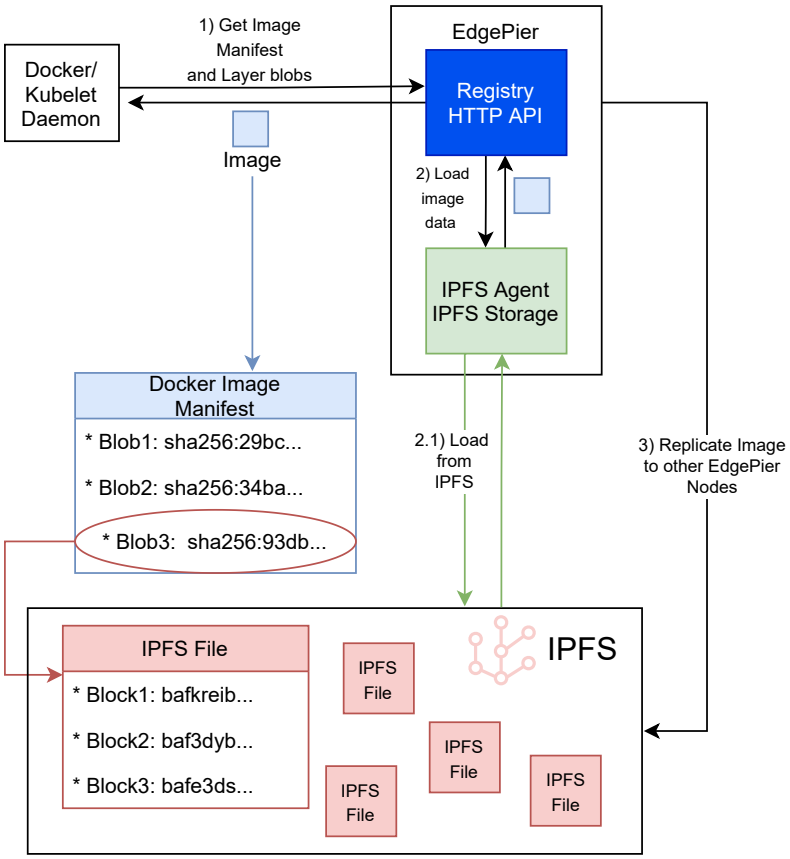


Figure 3.5: EdgePier workflow (Becker et al. [8])

On the schedule of an application, the daemon attempts to pull the image from the local registry. EdgePier checks if the layers are locally available to get them from the IPFS local storage. If not available, it locates the data in the IPFS network and downloads them from the selected peers. By default, Docker splits the images into layers, allowing different images to share the same layer. Similarly, IPFS divides the layers into smaller pieces called blocks. This process makes it easy to distribute the demand for download across multiple peers. Furthermore, similar to the work of Gazzetti et al. [19], the authors consider nearby nodes to have a common purpose. In this sense, EdgePier replicates an image stored on other network nodes at the same edge site.

Darrous et al. [14] propose two container image placement algorithms to reduce image provisioning time. Although an image placement algorithm does not directly modify the container registry, it makes multiple nodes serve as a registry. The authors propose the k-Center-Based Placement (KCBP) and KCBP-Without-Conflict (KCBP-WC) algorithms, which are responsible for placing sets of container images and layers across a set of fully connected nodes. These algorithms aim to reduce the maximum retrieval time of images and layers to any edge server of the infrastructure.

To design the first algorithm, the authors formalize the problem of minimizing the retrieval time for all layers on all nodes as the MaxLayerRetrievalTime problem. Then, they compare this problem, if using a single layer, to the k-Center problem, which aims to place facilities on a set of nodes to minimize the distance from any node to the closest facility. Finally, they use a solver for k-Center in KCBP to generate valid layer placements in the infrastructure. Subsequently, to design the second algorithm, they formalize the problem for retrieving complete container images as the MaxImageRetrievalTime problem. To this problem, the authors defend that two layers of the same image should not be placed at the same node to avoid sharing bandwidth. However, it is hard to accomplish this constraint considering the capacity of the nodes and the need to avoid these conflicts. To solve this condition, the authors propose that in KCBP-WC, which also uses the solver for the k-Center problem, a percentage of the largest layers of an image cannot be placed at the same node. Thus, the algorithm avoids sharing bandwidth when provisioning large layers, minimizing the effects of this event.

Knob et al. [29] advocate that the use of strategically placed container registries in an edge computing infrastructure can improve the application deployment process. To address this, the authors propose an algorithm to split the infrastructure into communities and select an edge server to host a container registry in each community. In this work, the authors assume that the registries are only in the edge infrastructure (i.e., the image provisioning does not rely on cloud-based registries). In this sense, each server with a community-based registry has a copy of all the necessary container images in the storage for provisioning applications on the edge infrastructure. This replication occurs because each registry needs to cover a whole region (i.e., a community of nodes), so an edge server should not rely on a registry from another community. This fact highlights the importance of picking an adequate number of communities to avoid overhead in specific regions, besides avoiding maintaining registries that spend most of the time inactive, given the storage cost.

The strategy for placing the container registries starts with the fluid communities (FluidC) algorithm for partitioning the graph into the input number of communities. This algorithm initially selects the communities randomly and runs iteratively until no better communities are found, based on the available bandwidth in each community, or until it reaches the termination condition. Then, for each community, the eigenvector centrality algorithm is applied. This algorithm gives information about the centrality and connectivity of the nodes with respect to the other community nodes. With this information, the algorithm selects which node in each community will host the container registry.

More recently, Temp et al. [52] propose a registry migration strategy to avoid Service Level Agreement (SLA) violations. Although P2P-based strategies for edge computing environments have a certain degree of mobility awareness, the authors claim that the algorithm is the first to dynamically migrate container registries according to user mo-

bility. Accordingly, a relevant aspect of the strategy is to use user mobility to provision new registries preemptively. Another relevant aspect is that it assumes fully replicated registries, as Knob et al. [29] did. This practice enhances the ability to spread the provisioning demands across the infrastructure but might represent a limitation in terms of resource usage, especially regarding disk demands.

As the strategy relies upon SLAs, each application has a delay SLA (i.e., the maximum acceptable delay between the user and the app's server) and a provisioning time SLA (i.e., the maximum acceptable time for provisioning the application's container image from the registry to a target server). The algorithm uses thresholds to make anticipated decisions and avoid SLA violations. These thresholds are user-defined and represent a percentage of the delay SLA and provisioning time SLA. First, the algorithm iterates over the applications to migrate only those with the delay to the end user above the delay SLA threshold to a server closer to the end user and with the capacity to host the application, if there is any. Then, the algorithm deprovisions all container registries distant from users. Finally, it iterates over the servers with resource capacity to host a container registry until the registries are provisioned close enough to a set of users or until no server can host a registry.

3.3 Final Remarks

In the previous sections, we presented several works with different approaches to improve the container image provisioning operation with a set of distributed container registries. Generally, these approaches aim to improve the provisioning time of the container images from the registry to the demanding server. Speeding up this process allows allocating applications sooner and ensures some service-level objectives (SLOs) to keep the QoS and QoE in a given system. However, some aspects of edge computing infrastructures (e.g., user mobility and resource limitation) can make this process more difficult, besides demanding more than simply ensuring better provisioning time. Along with provisioning time, latency becomes a key target metric based on its relevance in edge computing scenarios. In addition, computing and storage resources must be well-allocated to avoid the waste of resources and impact on user latency due to the lack of nearby resources.

The most common approach is to use P2P registries to exploit cached images on nodes that are not initially a registry. Some of the highlighted approaches focus on data centers [4, 26, 30, 53] with abundant resources and stable networks to integrate with the requirements of BT-based solutions. Meanwhile, there are a few strategies [8, 19] that propose alternatives to BT-based approaches to comply with the constraints of edge infrastructures. However, the main problem with P2P registries is that they consider every single node of the network a registry, as long as the nodes have container images in

their storage. Registry functionality demands computing resources, which could be more valuable for applications with stringent latency demands. Thus, using P2P registries might harm resource usage and the application's user latency.

Next, we have the container image placement approaches [14, 37]. These strategies distribute container images across multiple servers and create an infrastructure with distributed registries. Although the image placement is based on algorithms to optimize the image distribution, they are placed statically on the infrastructure. Therefore, as users move through the edge infrastructure, initial placement may be outdated and provisioning time can be affected. In addition, both strategies for image placement do not seem to have a mechanism to control or limit the number of servers that act as container registries, which can elevate resource usage and negatively affect the application's user latency.

Lastly, we discuss the remaining approaches aimed at container registries in edge computing infrastructures: registry placement [29] and migration [52]. The registry placement strategy has a notable dependency on the ideal number of registries, which is not pre-defined. Experiments with this strategy have shown that a relevant number of registries (around one-fourth of the edge servers) is required to improve the provisioning time. Although this elevated number of registries does not seem to affect the application's user latency, it certainly impacts the storage demands the edge infrastructure must have because container images are completely replicated in each registry. The registry migration strategy follows the dynamic aspect of P2P-based approaches. However, this approach makes decisions based on user mobility, which is not the case with P2P approaches. This strategy also aims to achieve latency SLAs while keeping a number of registries provisioned based on the current demands. Even though it is very aware of the target metrics we specified, migrating the server storage to another, even partially, can significantly impact the image provisioning process and the communication flows between the users and their applications due to network overhead.

To conclude this chapter, we summarize the related works in Table 3.1. Given the lack of compliance of most existing strategies with all the target metrics and the complexity of frequently migrating registries in edge infrastructure from Temp et al.'s strategy [52], we propose to dynamically provision and deprovision P2P registries in edge computing infrastructures, taking advantage of the already cached images in the servers. Our goal with this technique is to maintain an adequate number of container registries to minimize image provisioning time and avoid resource wastage, benefiting applications that need to improve their latency by reallocating to other servers.

Table 3.1: Comparative table between this work and related strategies.

			Target metrics		
Work	**Paradigm**	**Approach**	**Prov. time**	**App. latency**	**Resource usage**
[30]	Cloud	P2P Registry	✓	×	×
[26]	Cloud	P2P Registry	✓	×	×
[53]	Cloud	P2P Registry	✓	×	×
[4]	Cloud	P2P Registry	✓	×	×
[37]	Cloud	Image placement	✓	×	×
[19]	Edge	P2P Registry	✓	×	×
[8]	Edge	P2P Registry	✓	×	×
[14]	Edge	Image placement	✓	×	×
[29]	Edge	Registry placement	✓	✓	×
[52]	Edge	Registry migration	✓	✓	✓
This Work	Edge	Dynamic registry provisioning	✓	✓	✓

4. CONTRIBUTIONS

In this chapter, we discuss the contributions of our work. Section 4.1 details the system model, which includes the notation to represent the elements that make up the infrastructure, the equations used to facilitate the explanation of the algorithms, the constraints that we have considered, and the equations to measure our target metrics. Then, Section 4.2 includes a custom algorithm to select the registry to download container layers in a distributed manner, based on the proposal by Nathan et al. [37], along with the two algorithms we designed for dynamic provisioning and deprovisioning container registries in edge computing infrastructures: LMDyn and MODyn.

4.1 System Model

The map of the edge computing infrastructure represented in this work comprises multiple adjacent hexagonal cells that represent the coverage area managed by a single base station, as depicted by Aral et al. [6] and considered a typical representation of mobile cellular networks [21]. Figure 4.1 outlines this infrastructure and its main elements, which we detail in the following paragraphs.

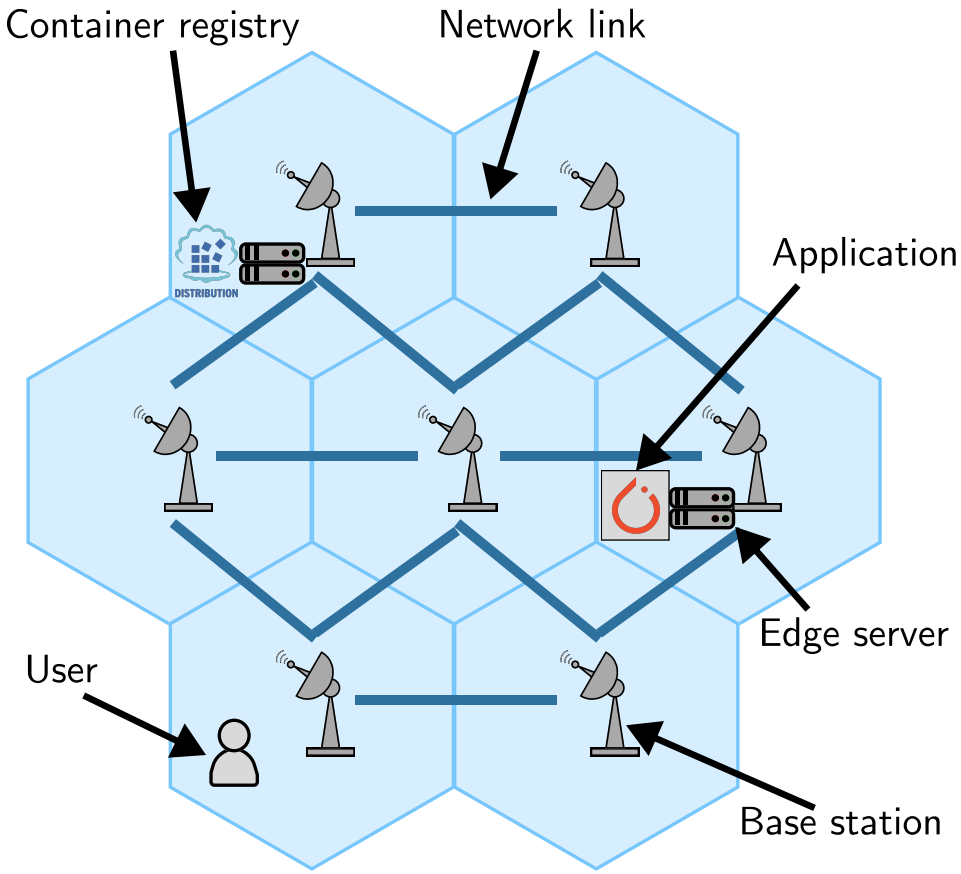


Figure 4.1: Example of infrastructure based on the system model.

In the infrastructure, each base station in the set of base stations B has fixed wireless latency for the entire coverage area, and there is no overlap between the base stations. Furthermore, a set N of network links interconnect these base stations and allow communication between distant entities. Each network link $N_f = \{b_f, l_f\}$ has a bandwidth b_f and a latency l_f. Coupled with some base stations in B, we have edge servers to provide computing and storage capabilities. The set of edge servers E has each server represented as $E_i = \{c_i, r_i, d_i, \dot{d}_{i,t}\}$, in which the first three elements represent a different capacity of the edge server: c_i is the CPU capacity, r_i is the RAM capacity, and d_i is the disk capacity.

While container registries and applications consume CPU and RAM, the container images and their pieces (i.e., container layers) consume the disk of the edge servers. Although each application reflects a specific container image, the disk demand of an application depends on which layers the target server to host this application already has. Thus, we represent the disk demand $\dot{d}_{i,t}$ as an attribute of the edge server that varies over time. Meanwhile, CPU and RAM demands of applications and container registries are fixed and represented directly in these entities. In addition to the attributes mentioned above, $x_{i,k}$ represents the edge server placement matrix, which works as detailed in Equation 4.1 for every time step $t \in T$.

$$x_{i,k} = \begin{cases} 1 & \text{if edge server } E_i \text{ is coupled to base station } B_k \\ 0 & \text{otherwise.} \end{cases} \tag{4.1}$$

One of the entities hosted by edge servers is the container registry. Container registries store container images and layers for distributing these elements to other servers. The set of container registries R has each element represented as $R_l = \{\dot{c}_l, \dot{r}_l\}$. For each registry, R_l, $\dot{c}_l$ represents the CPU demand and $\dot{r}_l$ represents the RAM demand. Previous work ignored these demands for container registries, but they are significant in edge computing scenarios because of the limited resources in these infrastructures. Furthermore, $y_{i,l,t}$ represents the container registry placement matrix, which works as detailed in Equation 4.2.

$$y_{i,l,t} = \begin{cases} 1 & \text{if edge server } E_i \text{ hosts registry } R_l \text{ at time step } t \\ 0 & \text{otherwise.} \end{cases} \tag{4.2}$$

Along with container registries, applications also take advantage of the computing resources provided by edge servers. The set of applications A contains multiple entities modeled as $A_j = \{\hat{c}_j, \hat{r}_j, z_j, u_j, p_{j,t}, \delta(A_j, t), h_{j,t}\}$. Similarly to the container registry, each application has CPU and RAM demands represented by $\hat{c}_j$ and $\hat{r}_j$, respectively. Each application reflects a container image z_j from the set of container images I, which contains elements represented as $I_o = \{q_o\}$, in which q_o is the set of container layers from L con-

tained in this image. The matrix $z_{i,j,t}$ represents the placement of the application and works according to Equation 4.3.

$$z_{i,j,t} = \begin{cases} 1 & \text{if edge server } E_i \text{ hosts application } A_j \text{ at time step } t \\ 0 & \text{otherwise.} \end{cases} \tag{4.3}$$

Furthermore, each application has a single user u_j, which has a wireless latency w_j to communicate with the nearest base station. Each application A_j also has a communication path list $p_{j,t}$ that might change over time. For a given time step $t \in T$, this list contains the set of network links that connect the base station u_j' to the base station B_k, so that E_i hosts A_j (i.e., $z_{i,j,t} = 1$) and E_i is associated with B_k (that is, $x_{i,k} = 1$). Equation 4.4 represents the latency between u_j and the application A_j for a given time step. It consists of summing the wireless latency $w_{j,t}$ with the latency's sum of all network links in $p_{j,t}$. Lastly, each application has a history of provisioning times ($h_{j,t}$), which lists the time intervals it took to complete each of the finished reallocations of A_j'.

$$\delta(A_j, t) = w_{j,t} + \sum_{N_f \in p_{j,t}} l_f \tag{4.4}$$

We also consider some constraints that the scenario must meet to maintain the evaluation valid. The first constraint, depicted in Equation 4.5, ensures that each application is placed only once on a single edge server during all time steps of the set T. Then, equation 4.8 ensures that the CPU, RAM, and disk capacities do not exceed during all time steps of the set T.

$$\sum_{i=1}^{|E|} z_{i,j,t} = 1 \qquad \forall j \in \{1, ..., |A|\},\ \forall t \in \{1, ..., |T|\} \tag{4.5}$$

$$cpu\_demand(i, t) = \sum_{l=1}^{|R|} \dot{c}_l \cdot y_{i,l,t} + \sum_{j=1}^{|A|} \hat{c}_j \cdot z_{i,j,t} \tag{4.6}$$

$$ram\_demand(i, t) = \sum_{l=1}^{|R|} \dot{r}_l \cdot y_{i,l,t} + \sum_{j=1}^{|A|} \hat{r}_j \cdot z_{i,j,t} \tag{4.7}$$

$$\begin{gathered} [c_i \leq cpu\_demand(i, t)] + [r_i \leq ram\_demand(i, t)] + \left[d_i \leq \dot{d}_{i,t}\right] = 0, \\ \forall i \in \{1, ..., |E|\},\ \forall t \in \{1, ..., |T|\} \end{gathered} \tag{4.8}$$

Considering this scenario and the constraints, our goal with this work is to understand if dynamically provisioning and deprovisioning container registries in edge computing infrastructures only using the cached container images for distributing can optimize the latency and provisioning time of applications, besides the resource usage (i.e., CPU,

RAM, and disk usage) of edge servers, during an interval of time, or at least offer an adequate trade-off to be considered a relevant choice in some scenarios. For this matter, we model four equations to measure these different optimization goals independently. Equation 4.9 considers the mean latency between users and A's applications during a set of time steps T. The mean latency value refers to the latency of one application $A_j \in A$ in a single time step $t \in T$. Meanwhile, Equation 4.10 considers the mean provisioning times of A's applications during a set of time steps T. The mean provisioning time value refers to the time it takes to make a single provisioning of an application A_j considering its entire history of provisioning times $h_{j,|T|}$.

$$\frac{\sum_{j=1}^{|A|} \sum_{t=1}^{|T|} \delta(A_j, t)}{|A| * |T|} \tag{4.9}$$

$$\frac{\sum_{j=1}^{|A|} \sum_{n=1}^{|h_{j,|T|}|} h_{j,|T|}[n]}{|A| * \sum_{j=1}^{|A|} |h_{j,|T|}|} \tag{4.10}$$

The following two equations are related to resource usage objectives. First, we have Equation 4.11 considering the mean number of provisioned registries per time step during a set of time steps T. This equation measures the use of computing resources (i.e., CPU and RAM) because applications consume a fixed amount of CPU and RAM. Meanwhile, the dynamic consumption of these resources depends only on the number of provisioned container registries over time (i.e., more provisioned registries equals a greater consumption of computing resources). Then, we have Equation 4.12 considering the mean disk demand per edge server and time step during a set of time steps T. This equation measures the use of storage resources (i.e., disk) by edge servers, which is more elevated in strategies requiring multiple fully replicated registries than in strategies that only rely on cache-based container images.

$$\frac{\sum_{t=1}^{|T|} |R^t|}{|T|} \tag{4.11}$$

$$\frac{\sum_{t=1}^{|T|} \sum_{i=1}^{|E|} \dot{d}_{i,t}}{|E| * |T|} \tag{4.12}$$

Table 4.1 summarizes the notation of the elements and their attributes presented in this section. This notation is especially useful to describe the pseudo-code of our algorithms in the next section.

Table 4.1: Summary of notations used in this paper.

Symbol	Description
B	Set of base stations
N	Set of network links
E	Set of edge servers
R^t	Set of container registries at time step t
A	Set of applications
U	Set of users
I	Set of container images
L	Set of container layers
T	Set of time steps
b_f	N_f's bandwidth
l_f	N_f's latency
c_i	E_i's CPU capacity
r_i	E_i's RAM capacity
d_i	E_i's disk capacity
$\dot{d}_{i,t}$	E_i's disk demand at time step t
$x_{i,k}$	Edge server placement matrix
$\dot{c}_l$	R_l's CPU demand
$\dot{r}_l$	R_l's RAM demand
$y_{i,l,t}$	Container registry placement matrix
$\hat{c}_j$	A_j's CPU demand
$\hat{r}_j$	A_j's RAM demand
z_j	A_j's container image
q_o	l_o's container layers
$z_{i,j,t}$	Application placement matrix
u_j	A_j's user
$w_{j,t}$	u_j's wireless latency at time step t
$p_{j,t}$	A_j's communication path
$\delta(A_j, t)$	A_j's latency at time step t
$h_{j,t}$	A_j's history of provisioning times
$\sigma(\epsilon_1, \epsilon_2, t)$	Latency between elements ϵ_1 and ϵ_2 at time step t

4.2 Algorithms

This section showcases the algorithms that we have developed as our contributions. In Section 4.2.1, we present an algorithm that enables pulling different layers of the same container image from multiple registries, based on the work of Nathan et al. [37] and considered an adequate practice in P2P-based registries. In Section 4.2.2, we present the two algorithms we design for dynamically provisioning and deprovisioning container registries in edge computing infrastructures. The interval between each algorithm's execution can be custom for both strategies, which take advantage of the container images and layers already cached in the servers' storage. Thus, they only allocate new registries and remove existing ones without requiring any migration of resources.

4.2.1 Distributed Pull Algorithm

In Section 3.2, we describe CoMICon [37], a cooperative management system for Docker container images. In their work, they highlight that the existing P2P approaches are unaware of the layered aspect of Docker container images and, thus, do not take advantage of this (i.e., they enforce the download of an entire image from a single source registry). Consequently, the authors propose to modify the Docker daemon to introduce a distributed pull of images in which the layers of a single image can come from different hosts. Since then, P2P strategies for provisioning container images have evolved and already support this prominent operation, which decreases the provisioning time by avoiding overloading a single container registry. Therefore, we built a simple algorithm to select the target registry to pull a container layer following the distributed pull proposition. Algorithm 4.1 depicts the implemented strategy. The goal is to collect container layers within a single container image from different container registries.

Algorithm 4.1: Distributed Pull Algorithm.

Input 1: *target_layer* – target layer to be downloaded
Input 2: *target_server* – target server for the download of *target_layer*
Input 3: *ts* – current time step
1: R' = container registries that contain *target_layer* in its server's storage at the time step *ts*
2: **for** $R_l \in R'$ **do**
3: $R_l[downloads]$ = number of layers being downloaded from R_l at the time step *ts*
4: $R_l[latency] = \sigma(R_l, target\_server, ts)$
5: **end for**
6: $registry = min(R', (downloads, latency))$
7: Start downloading *target_layer* from *registry*

As input to the algorithm, we have *target_layer*, *target_server*, and *ts*. These inputs indicate that *target_server* wants to download *target_layer* to allocate a given application considering the state of the infrastructure at the time step *ts*. The first step of the algorithm is to filter which container registries have *target_layer* available for download (line 1). The algorithm then iterates over these filtered registries to obtain two metrics regarding each registry (lines 2–5). The first metric is the current number of layers' downloads, while the second is the latency between the registry and *target_server*. The algorithm's idea is to perform a round-robin scheduling operation to select the registry based on the current number of active downloads. The latency is a tiebreaker for the first parameter for determining the priority of nearby registries. Based on these two metrics, the registry is selected, prioritizing the registry with the lowest number of active layer downloads and the closest registry if there is a tie with the first metric (line 6). Finally, the download of *target_layer* from the selected registry to *target_server* is started (line 7). Using this algorithm instead of relying exclusively on the closest registry improved the provisioning time of the target strategies (i.e., P2P-based strategies) for this distributed pull algorithm.

4.2.2 Dynamic Registry Provisioning Algorithms

LMDyn: Layer-Matching-based Dynamic Registry Provisioning

During preliminary experiments to compare existing registry provisioning strategies, we identified that the P2P strategy for provisioning container registries keeps allocating new edge servers to host container registries (as show in Figure 5.6) under the premise that a server having at least a single container layer stored and with CPU and RAM resources will join the P2P network. This practice leads to the growing consumption of resources in these registries and a conflict between the registries and the applications for these resources. Consequently, the unavailability of resources might prevent reallocating an application that can be closer to its end-user. With this in mind, we design LMDyn, an algorithm for dynamically choosing which edge servers should host P2P-based registries until the following algorithm is executed, based on the layer-matching percentage between the servers' storage and nearby applications. Algorithm 4.2 presents the proposed heuristic to tackle this problem. The algorithm's primary goal is to reduce the overall mean latency by provisioning a controlled number of P2P-based registries instead of allocating every possible edge server, which is the P2P strategy.

Algorithm 4.2: LMDyn Algorithm.

Input 1: R_c – base registry (initial registry with all container images)
Input 2: ts – current time step

E' = edge servers with capacity to host a registry at time step ts
for $E_i \in E'$ **do**
  for $A_j \in A$ **do**
    if $\sigma(u_j, E_i, ts) \leqslant \sigma(u_j, R_c, ts)$ **then**
      lm = percentage of A_j's service layers that are in E_i's disk at time step ts
      if $lm > 0$ **then**
        $E_i[lm] += lm$
        $E_i[pr] += 1$
      end if
    end if
  end for
end for
$pr_{mean} = int(\frac{\sum_{i=1}^{|E'|} E_i[pr]}{|E'|})$
E'' = edge servers in E' with possible recipients above pr_{mean}
$lm_{mean} = \frac{\sum_{i=1}^{|E''|} E_i[lm]}{|E''|}$
E''' = edge servers in E'' with layer matching above lm_{mean}
for $E_i \in E'$ **do**
  if $E_i \in E'''$ && $\sum_{l=1}^{|R|} y_{i,l,ts} == 0$ **then**
    Provision a registry on E_i
  end if
  if $E_i \notin E'''$ && $\sum_{l=1}^{|R|} y_{i,l,ts} == 1$ **then**
    Deprovision the registry allocated on E_i
  end if
end for

For any P2P-based approach, we always consider that there exists a central registry located in the edge infrastructure to ensure that at least one server has all the necessary container images and layers to distribute them to the other servers. The central registry R_c and the current time step *ts* are the inputs of this algorithm. The first step is to filter which servers can host container registries (line 1), which includes servers that already have a registry at the time step *ts* and servers that do not have a registry at the time step *ts* but have CPU and RAM resources to host a registry. Then, the algorithm iterates over these filtered servers and, for each server, calculates two scores (lines 2–12): layer matching and possible recipients. The algorithm iterates over all applications hosted in the infrastructure to calculate these scores, and, for each application, checks if the application user is closer to the current edge server than to the central registry (lines 3–4). If it is, the layer matching between the server and the application's container image is calculated and added to the layer matching score (lines 4–7). Additionally, if any layer matches, the score of possible recipients is incremented by one (line 8). Then, there are two subsequent cuts in the list of candidate servers, one after another, to minimize the number of selected servers. First, the algorithm discards edge servers with a score of possible recipients below the mean value (lines 13–14), and then servers with a layer-matching score below the mean value of this score are ignored (lines 15–16). Finally, the algorithm iterates over all candidate servers to provision and deprovision the necessary container registries (lines 17–24).

MODyn: Multi-Objective Dynamic Registry Provisioning

After analyzing the performance of the LMDyn algorithm, we observed a significant improvement in the application's latency and the usage of the infrastructure's resources. However, the time required to provision the missing layers of an application to a target server to complete a reallocation based on user mobility was not satisfactory. This behavior occurs because fewer container registries are available, which means fewer options to download a container image. This can lead to network bottlenecks and increase the time it takes to download missing layers and complete an application's reallocation. Although these metrics can conflict with each other (e.g., more provisioned registries decrease the provisioning time but occupy more resources and then increase the latency), a lower provisioning time can contribute to reducing the latency (e.g., provisioning an application faster speeds up the reallocations and updates the latency to a lower value faster than with fewer registries available).

Thus, based on this problem, we propose a second strategy for dynamically provisioning and deprovisioning container registries with multiple objectives: MODyn. This algorithm also aims to decrease the overall mean latency of applications by minimizing the resource usage in the infrastructure. The purpose of this system is also to maintain an adequate provisioning time. In this regard, we introduce the goal of replicating the con-

tainer images and keep provisioning registries until there are available resources or until we reach the desired number of replicas, given a few constraints. In addition, we prioritize provisioning registries on edge servers with higher amounts of free resources and fewer possible future demands. The strategy is depicted in Algorithm 4.3 and is explained below.

Algorithm 4.3: MODyn Algorithm.

Input 1: *replicas* – number of replicas per image
Input 2: *ts* – current time step
1: E' = edge servers with capacity to host a registry at time step *ts*
2: **for** $E_i \in E'$ **do**
3: $E_i[free\_cpu] = c_i - (\sum_{l=1}^{|R|} \dot{c}_l \cdot y_{i,l,ts} + \sum_{j=1}^{|A|} \hat{c}_j \cdot z_{i,j,ts})$
4: $E_i[free\_ram] = r_i - (\sum_{l=1}^{|R|} \dot{r}_l \cdot y_{i,l,ts} + \sum_{j=1}^{|A|} \hat{r}_j \cdot z_{i,j,ts})$
5: $E_i[free] = geo\_mean(E_i[free\_cpu], E_i[free\_ram])$
6: **for** $A_j \in A$ **do**
7: **if** $\delta(A_j, ts) > \sigma(u_j, E_i, ts)$ **then**
8: *nd* = A_j's normalized amount of CPU and RAM demand
9: $E_i[pd]$ += *nd*
10: **end if**
11: **end for**
12: **end for**
13: E'' = edge servers in E' sorted by $(E_i[pd] - E_i[free])$ (asc.)
14: **while** replication target not reached and $|E''| > 0$ **do**
15: $E_i = E''$.pop(0)
16: n_acc = number of images in E_i's storage that accomplished the replication target *replicas*
17: n_unacc = number of images in E_i's storage that have not accomplished the replication target *replicas*
18: **if** n_unacc >= n_acc **then**
19: **if** $\sum_{l=1}^{|R|} y_{i,l,ts} == 0$ **then**
20: Provision a registry on E_i
21: **end if**
22: Updates replication data
23: Continue
24: **end if**
25: **if** $\sum_{l=1}^{|R|} y_{i,l,ts} == 1$ **then**
26: Deprovision E_i's registry
27: **end if**
28: **end while**
29: Deprovision registries from the remaining servers in E''

In this algorithm, we consider the number of replicas each container image available in the infrastructure should have. It is not a constraint for the conclusion of a single algorithm's execution, but it is desirable to have the replication target achieved for all the container images. Following this approach, we have the *replicas* input to determine the minimum number of replicas for each image and the *ts* input to denote the current time step. The algorithm starts by filtering the edge servers that can host a container registry at the time step *ts* (line 1). Then, it iterates over all selected servers and calculates the first score of the edge server, which is the amount of free normalized computing resources (i.e., CPU and RAM) through a geometric mean (lines 3–5). The other score is named *possible demand* and represents the normalized amount of computing resources from all applications that could be reallocated to the server because it offers better latency

(lines 6–11). Before selecting edge servers, we sort the filtered servers with fewer possible demands and more free resources (line 13). Then, while we do not reach the replication target set by *replicas* and still have edge servers to evaluate, we check if each server must be a registry (lines 14–28). This process includes analyzing whether most of the container images on the edge server reached the input replication target. If most images do not reach the target, the algorithm checks if there is a registry on the server, and if there is not, it allocates a registry there before updating the replication data (18–24). Otherwise, that server is not qualified to host a container registry, and if there is a registry there, we deprovision the existing registry (lines 25–27). If the algorithm achieves the replication target before iterating over all servers in E'', we ensure to remove the possible remaining registries after the loop (line 29).

In the next chapter, we evaluate all of these strategies, taking into account these two strategies and some baseline approaches from the literature.

5. EVALUATION

This chapter evaluates our strategies and baseline approaches to allocating container registries in edge computing infrastructures. Section 5.1 details the simulator used to perform our experiments. In Section 5.2, we describe the dataset and the parameterization of the evaluation process. Lastly, in Section 5.3, we present and discuss the results in various target metrics.

5.1 Simulator

To evaluate our work, we used EdgeSimPy [51], a verified Python-based simulator to model and evaluate resource management policies in edge computing environments. The increasing relevance of edge computing motivated the development of multiple simulators to address particular aspects of edge computing scenarios, such as user mobility. However, EdgeSimPy's creators analyzed the existing simulators for edge computing environments and concluded that they all lacked a few essential features that are relevant to this context. In this sense, EdgeSimPy has three exclusive built-in features compared to existing simulators: maintenance operations, container lifecycle management, and container registry management. While the first is outside the scope of our work, the other two are essential for our evaluation and, thus, justify the selection of EdgeSimPy to conduct experiments with strategies for allocating container registries. Figure 5.1 depicts the entire EdgeSimPy architecture, which includes container-related entities and other relevant entities described in our system model (Section 4.1).

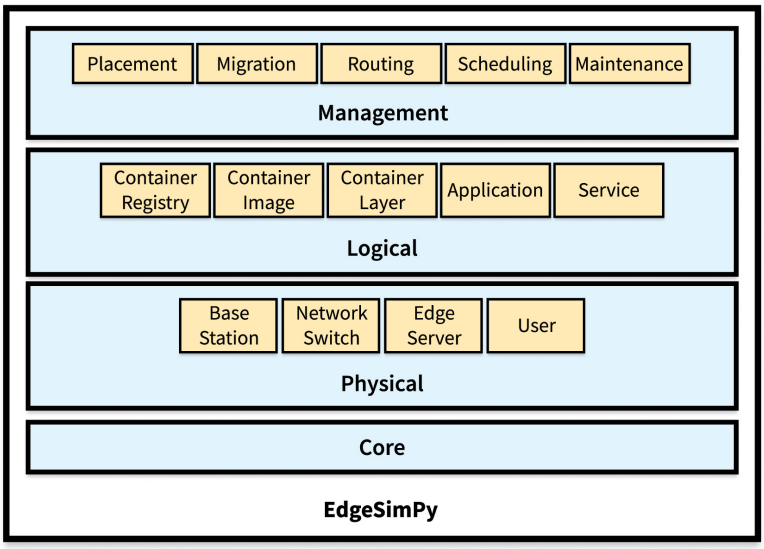


Figure 5.1: EdgeSimPy architecture (Souza et al. [51]).

5.2 Experiments Description

To evaluate our work, we consider two variable parameters that lead to multiple variations of the dataset and the opportunity to explore different scenarios. The first variable parameter we consider is the infrastructure size, which aims to evaluate the algorithms' performance under different infrastructure proportions. Thus, we build a dataset variation with 24 edge servers spread across 100 nodes interconnected by 261 network links and another variation with 48 edge servers spread across 196 nodes interconnected by 533 links. In both variations, there is a homogeneous mesh network topology [6], and each network link has a latency of 10 time units and a bandwidth of 1 Gbps. Each node represents a base station with a wireless latency of 10 time units within the coverage area. Edge servers are spread across nodes using the K-Means algorithm [32], and they have real CPU and RAM specifications obtained from Ismail et al. [25] along with a default disk size large enough to store all necessary container layers at once, as specified in Table 5.1.

Table 5.1: Edge server models' specification obtained from Ismail et al. [25].

Model	**CPUs**	**RAM**	**Disk**
Model 1	32 cores	32768 MiB	128 GiB
Model 2	48 cores	65536 MiB	128 GiB
Model 3	36 cores	65536 MiB	128 GiB

Generally, our goal is to keep the normalized occupation of the resources near 60%, according to Equation 5.1. Thus, the variation with 100 nodes has 128 users and applications, and the variation with 196 nodes has 256 users and applications.

$$\frac{\sum_{j=1}^{|A|} geo\_mean(k_j, m_j)}{\sum_{i=1}^{|E|} geo\_mean(c_i, r_i)} \tag{5.1}$$

Regarding the applications, their CPU and RAM demands are uniformly divided in the values depicted in Table 5.2, which covers a range of applications with lower demands to applications with higher demands for both computing aspects.

Table 5.2: Application demand specifications.

CPU Demand	**RAM Demand**
2 cores	2 GB
4 cores	4 GB
8 cores	8 GB

The users move on the map following the pathway mobility model [7]. This model contains regular and random components, which coincides with the fact that mobile users do not move completely random, nor completely deterministic ([56], as cited in [3]). The

algorithm to represent this model places users randomly across the base stations and repeats the following steps until the simulation ends: it selects a destination node to move for and chooses the shortest path (in our work, this path relies on the network links between the base stations) to go to the target node. Figure 5.2 shows an example of mobility in which the model randomly places the user in the hexagonal cell six and then randomly selects the hexagonal cell two as the next target. After selecting the target, the algorithm calculates the shortest path considering the connection through the network links. The solid path represents the shortest path to leave hexagonal cell six and arrive at hexagonal cell two by crossing hexagonal cell four. Meanwhile, the dashed paths represent a random path between these cells, which the pathway mobility model completely ignores.

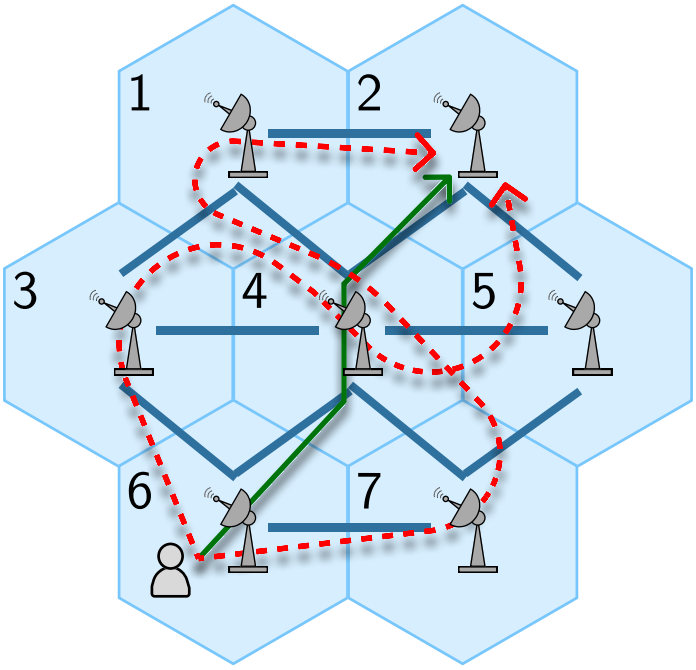


Figure 5.2: Pathway mobility model example in the hexagonal cell map with mesh network.

In addition, users stay in each hexagonal cell for an interval that depends on the speed at which each user moves. In this context, we consider two typical types of mobile users to reflect a realistic environment [38]: 6/7 of mobile users are pedestrians with a uniform speed distribution between 0.5 m/s and 1.5 m/s, and the remaining mobile users are drivers with a uniform speed distribution between 2.7 m/s and 11.1 m/s (considering each hexagonal cell with 500 meters of diameter to determine the time it takes to pass through a cell).

The domain of applications that we consider in this work is ML-based applications. Thus, we extracted 64 container images from the AWS Deep Learning container repository [1], which contains a set of container images aimed at training and serving AI models with different frameworks (e.g., TensorFlow and PyTorch). Most images have a version to run with CPU and a version to execute with GPU. However, due to a constraint

[1] https://github.com/aws/deep-learning-containers/

related to the edge servers considered, we only consider the CPU-aimed images in the scope of this work. Figure 5.3 and Table 5.3 contain relevant information regarding the 64 CPU-aimed container images. Figure 5.3 depicts the image size distribution of the images, which in their majority have less than 1 GiB and some others have between 1 GiB and 3 GiB, with a single image having near 6 GiB. Meanwhile, Table 5.3 shows the layer sharing information of the 64 chosen container images, which shows that most of the content in these 64 container images is exclusive to a single image. This aspect makes it difficult to reach partial cache hits when reallocating an application.

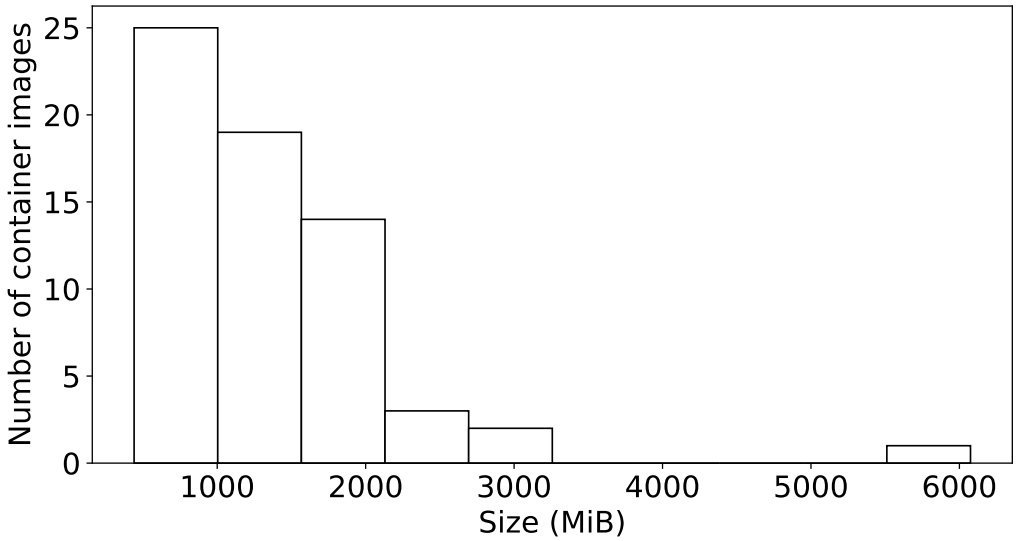


Figure 5.3: AWS Deep Learning container images size distribution.

Table 5.3: AWS Deep Learning layer-sharing information.

Number of images sharing	Total size (MiB)	Percentage of total size (%)
1	76728.261	96.523
2	2480.980	3.121
3	57.158	0.072
4	82.974	0.104
5	28.581	0.036
7	57.151	0.072
8	28.581	0.036
12	28.579	0.036
Total	79492.265	100.000

The second variable parameter we introduce is related to the applications and the container images that these applications comprise. More specifically, there is a variation in which 1/8 of the applications and users use the same container image and another in which 1/32 of the applications and users use the same container image. The first variation has fewer unique container images because more applications and users use the same image. The opposite happens to the second variation. With this parameterization, we aim

to understand the impact of the cache usage on the algorithms' performance, since we expect the scenario with fewer container images to take more advantage of this feature. Table 5.4 summarized the two values of the variable parameter and the four variations they create.

Table 5.4: Dataset variations and parametrization.

Variation	Infrastructure size	Unique container images
1	100 nodes and 128 users	8 container images
2	100 nodes and 128 users	32 container images
3	196 nodes and 256 users	16 container images
4	196 nodes and 256 users	64 container images

For each variation depicted in Table 5.4, we compare three baseline approaches for registry provisioning with the two algorithms presented in Section 4.2. Baseline approaches are the central registry, the community registry, and the P2P registry. The first approach only has a single container registry on the edge infrastructure, which is placed using the closeness-centrality algorithm to select the edge server to host the container registry. The second approach follows the strategy depicted by Knob et al. [29], in which there are multiple edge servers with fully-replicated container registries. To explore different configurations of this strategy and point out some trade-offs, we consider a version in which community registries cover one-eight (12.5%) of servers and another in which community registries cover one-fourth (25%) of edge servers. The third approach considers the existence of a central registry to ensure that at least one edge server has all the necessary container images to provision the applications in the infrastructure. In addition, this third approach considers that any edge server with at least one container image in the storage and computing resources to allocate a registry will have a registry. In addition to that, these P2P-based registries only provide container images already in the server storage, and more registries can be dynamically allocated during the simulation.

Taking into account these three approaches, we create four datasets for each variation shown in Table 5.4. These four datasets comprise a dataset with a central registry, two with community-based registries (one with one-eight of the servers allocating registries and the other with one-fourth), and a dataset with a central registry and P2P-based registries in qualified servers. This latter dataset is the base dataset to execute our two algorithms. Figures 5.4 and 5.5 represent the four different datasets created for variations 1 and 4 of Table 5.4, respectively. To allocate container registries on the edge servers, we rely on the existence of the official registry container image [2] in the edge server storage to deploy the registry there. As our system model specifies, the container registry has CPU and RAM demands to work well. In this case, we rely on the CPU and RAM specifications based on the minimal requirements of the Docker Trusted Registry, which are 2 two cores and 8 GiB of RAM [41].

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

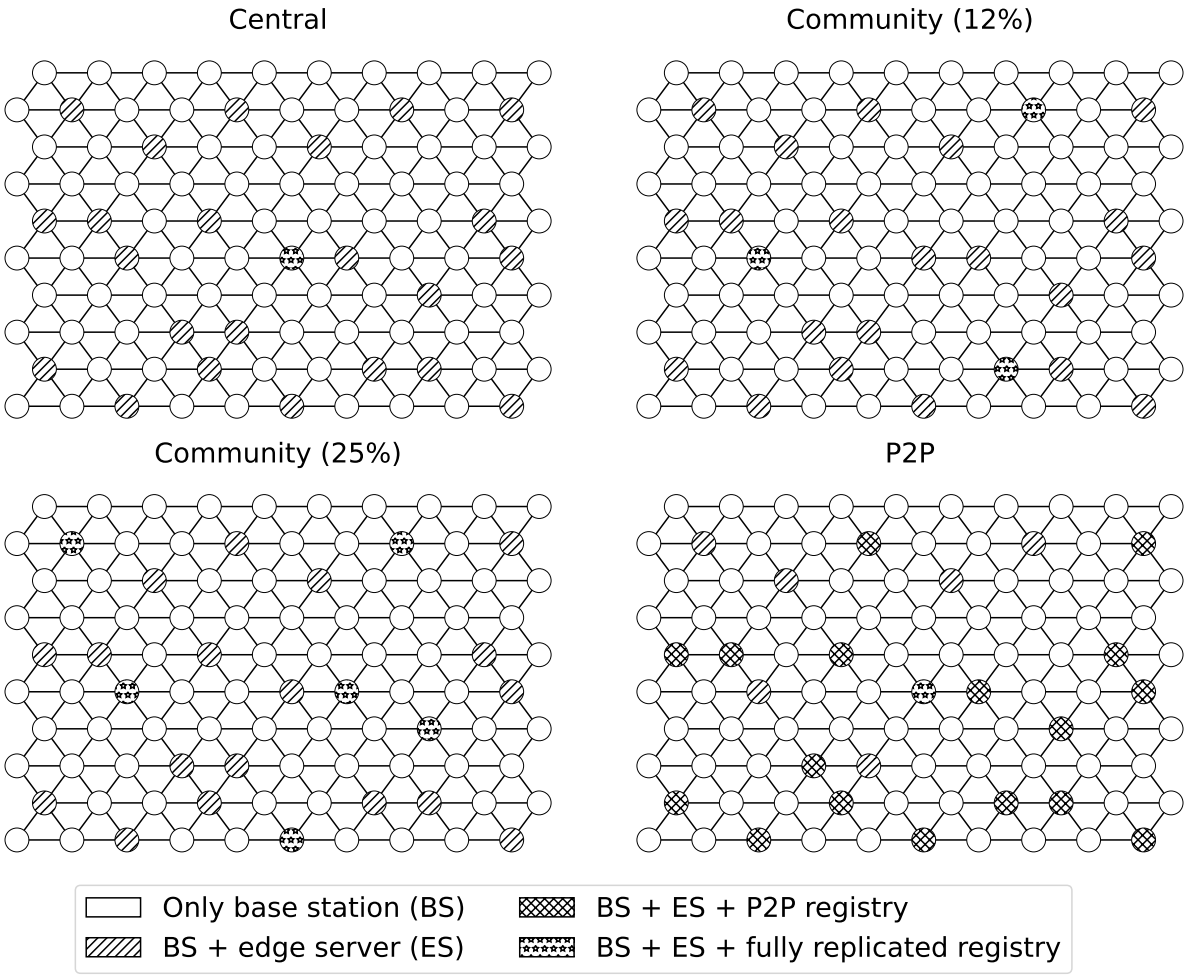


Figure 5.4: Datasets created for the variation with 100 nodes and 08 unique container images.

With these datasets and their variations, we execute simulations using EdgeSimPy with 3600 time steps each. This simulated time is equivalent to one hour of actual time. Although central and community strategies are static, the regular P2P approach checks if it is possible to provision a new registry at each time step. In addition, dynamic registry provisioning approaches update the registry allocation at each 60 time step. After the simulation, we processed the data generated by these simulations and compiled them into a set of metrics that we present in the next section. The source code to build the datasets, run the experiments and generate the set of metrics used in this work is available on GitHub [3] to ensure the reproducibility of our work. For each parameter variation, we evaluate the central registry, community registry, P2P registry, and dynamic registry provisioning strategies. The community registry strategy has the Comm* version with one-eight of the existing edge servers allocating registries and another (Comm+) with one-fourth of the edge servers allocating registries. The dynamic registry provisioning strategy comprises two algorithms described in Section 4.2: LMDyn and MODyn. In

[3] https://www.github.com/lucasroges/dynamic-registry-provisioning

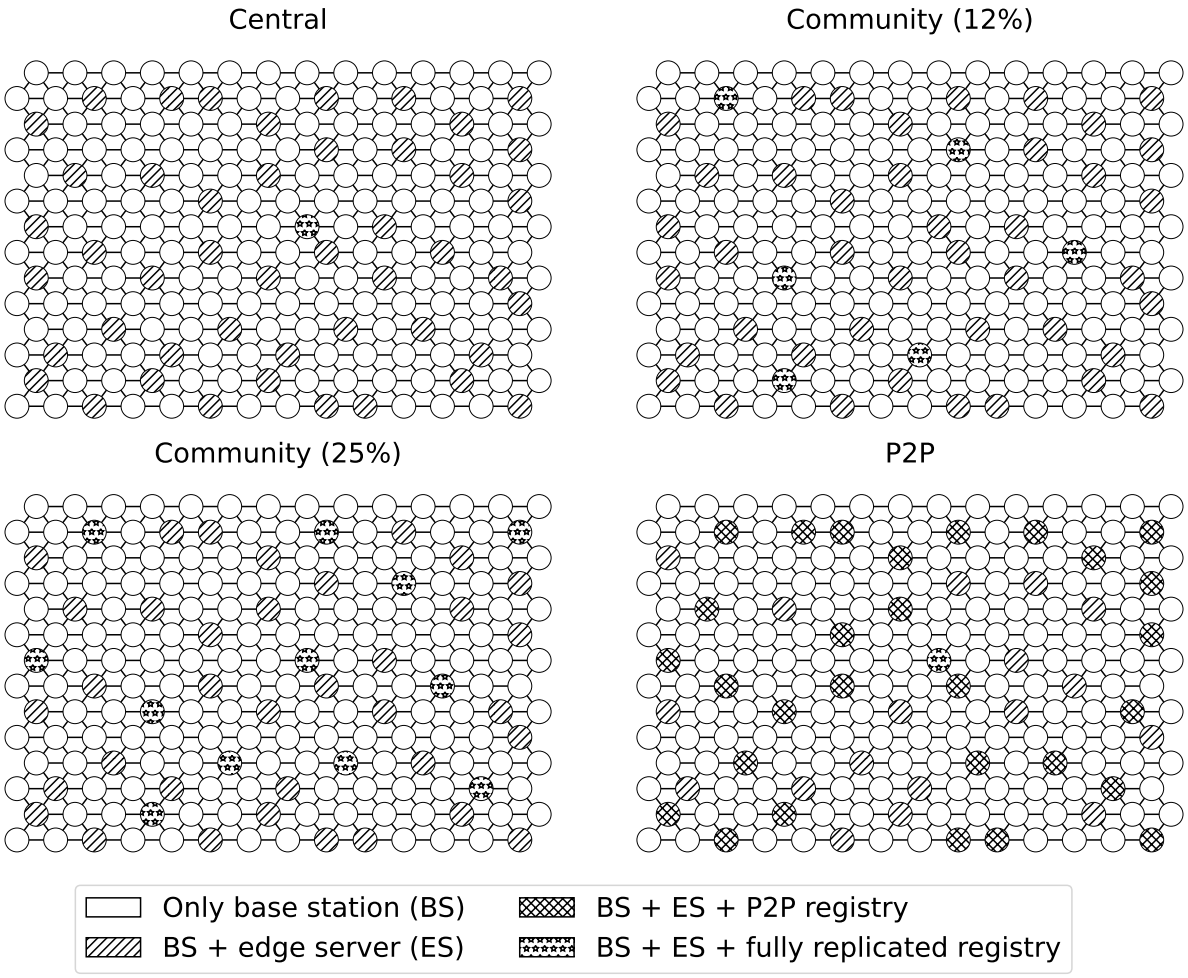


Figure 5.5: Datasets created for the variation with 196 nodes and 64 unique container images.

addition, we include four input versions for the replication input of the MODyn algorithm, specifying between 1 and 4 replicas as the target of the algorithm.

5.3 Results

In this section, we show the comparative results between the evaluated strategies. We start the evaluation by analyzing the mean latency results (Subsection 5.3.1) to understand each strategy's impact in this relevant metric for applications and users relying on edge computing infrastructures. Then, we look at the resource usage metrics (Subsection 5.3.2), focusing on CPU and RAM. This set of metrics includes the number of provisioned registries over time, the mean number of registries per time step, and the normalized server utilization. Subsequently, we analyze the applications' mean provisioning time (Subsection 5.3.3), which is the most common target metric in related work. To understand some patterns in the provisioning time, we also look into the reallocations per type (e.g., reallocation only using cache). Later, we aim at resource usage in terms of disk

utilization (Subsection 5.3.4). In this regard, we show the mean disk utilization per edge server and time step, in addition to the total disk utilization of the infrastructure over time. Although we have seen an impact of the elevated use of computing resources in other metrics, such as latency, disk utilization only indicates that some strategies may not be adequate for edge computing in some scenarios due to excessive disk allocation. Lastly, we analyze two metrics to understand some of the improvements offered by dynamic registry provisioning compared to P2P-based registries (Subsection 5.3.5): the percentage of time the registry is active (i.e., distributing container images) and the replication of container images.

5.3.1 Latency

Most of the related work discussed in Chapter 3 focuses on the provisioning time metric (i.e., the time it takes for the registry to distribute one or multiple images to some servers). Although we understand the importance of this metric and that it is the most profited by the use of distributed registries, we also recognize that both latency and resource usage are critical in edge computing infrastructures. Thus, we begin discussing the overall mean latency, in time units, obtained by the target strategies during our evaluation. Table 5.5 shows the results of the overall mean latency obtained using Equation 4.9.

Table 5.5: Overall mean latency in time units (Equation 4.9).

Nodes	100		196	
Unique images	08	32	16	64
Central	19.99	20.06	19.74	20.05
Comm*	20.19	20.11	19.91	19.88
Comm+	20.36	20.33	19.95	20.02
P2P	20.72	20.69	*20.76*	*20.80*
LMDyn	20.04	20.10	19.87	19.93
MODyn (1)	19.99	19.99	19.75	19.81
MODyn (2)	19.99	20.03	19.71	19.79
MODyn (3)	19.99	20.07	**19.70**	**19.76**
MODyn (4)	19.99	20.09	19.71	19.80

Generally, the latency values are similar, even between different variations of the dataset (different columns). The most significant difference occurs between the P2P algorithm for registry provisioning and the remaining approaches. This difference between the P2P algorithm and the remaining algorithms reaches around 5% in the variations with 196 nodes, which can affect the QoE required from latency-sensitive applications that rely on edge computing environments. A good explanation relies on the usage of resources from the provisioned P2P registries over time.

5.3.2 Computing Resources Usage

As we have detailed previously, these entities require CPU and RAM, with a significant amount of RAM required for each registry. Thus, there might be a dispute about computing resources between registries and applications. Following the P2P approach, a registry is allocated indefinitely to an edge server once it has at least one container image in its storage, wants to join the P2P network, and has computing resources to host a registry. That means that overloaded servers with container registries might be unable to provide resources for allocating an application that could have its latency to its end-user benefited from such a reallocation. Figure 5.6 depicts this critical behavior occurring in all simulated variations: the number of P2P registries tends to grow over time and reach the total number of servers at some point in the simulated time.

Regarding the other strategies, the central and community registry strategies have a fixed number of registries over time, while the dynamic registry provisioning has a variable number of provisioned registries. The number of provisioned registries by the LMDyn approach is more limited and variates between the two community use cases. Meanwhile, the number of provisioned registries by the MODyn approach versions follows a decreasing movement over time. This behavior occurs because user mobility spreads the container images across the edge servers. Then, fewer servers are required to achieve the replication target. In addition, lower replication targets (e.g., 1 and 2 container image replicas) demand fewer edge servers to allocate container registries than higher replication targets (e.g., 3 and 4 container image replicas). Table 5.6 summarizes these observations.

Table 5.6: Mean number of provisioned registries per time step (Equation 4.11).

Nodes	100		196	
Unique images	08	32	16	64
Central	1.00	1.00	1.00	1.00
Comm*	3.00	3.00	6.00	6.00
Comm+	6.00	6.00	12.00	12.00
P2P	*22.94*	*22.94*	*44.41*	*44.43*
LMDyn	3.35	3.69	7.06	7.62
MODyn (1)	2.74	5.02	4.59	9.59
MODyn (2)	4.67	8.84	7.44	16.36
MODyn (3)	6.04	12.12	10.11	21.02
MODyn (4)	7.52	14.59	12.76	25.07

Figure 5.6 and Table 5.6 also highlight a difference between variations with fewer unique images (08 and 16 unique images) and variations with more unique images (32 and 64 unique images). The MODyn approach provisions fewer container registries with

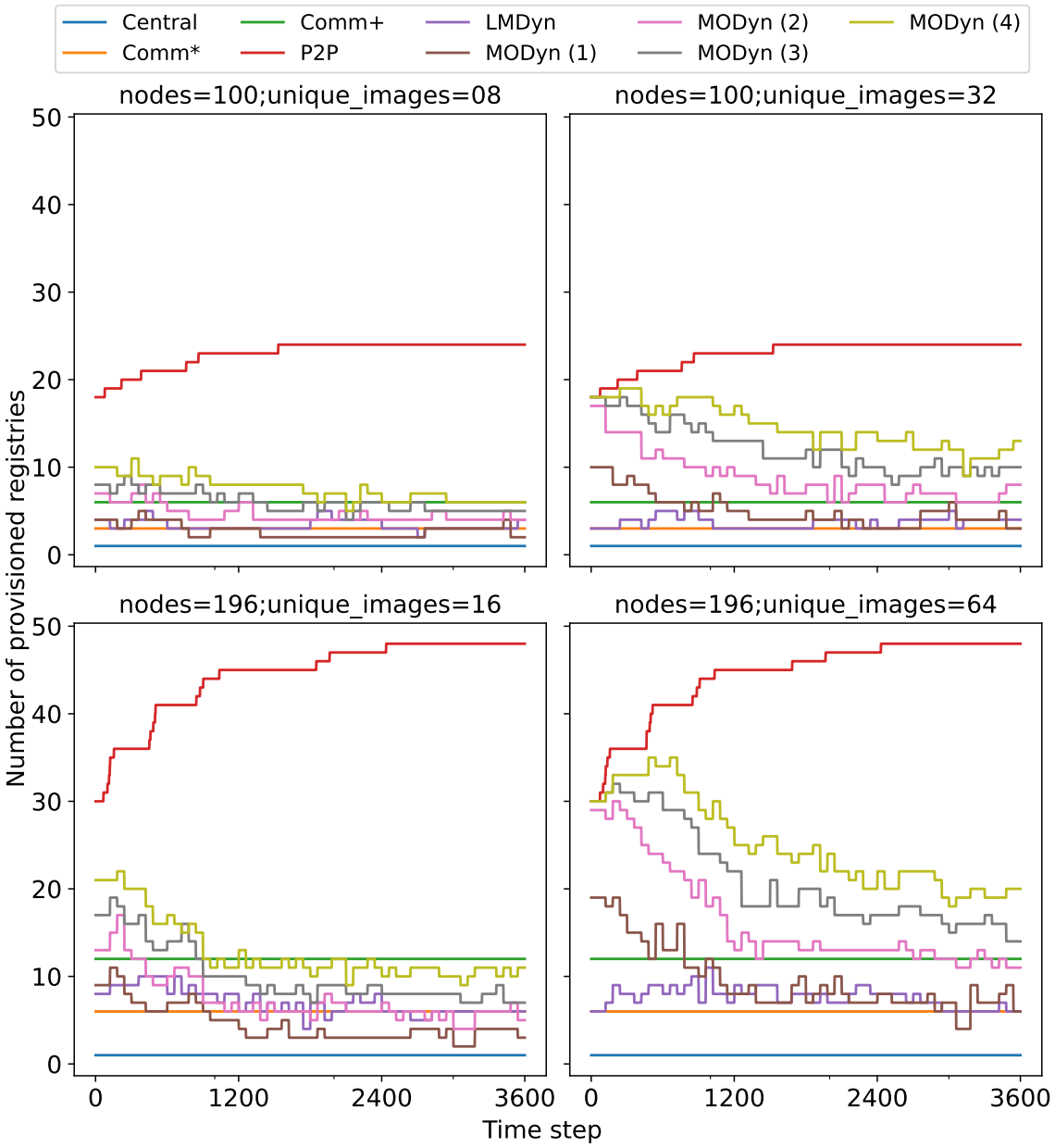


Figure 5.6: Quantity of provisioned registries per time step along the simulation time.

fewer unique images because it is easier to achieve the replication target with fewer images. As we can observe in the two subfigures on the left side of Figure 5.6, the number of registries provisioned by the MODyn approach can be slightly higher than the Comm+ version at the beginning of the simulation, but then it gets equal or lower. This pattern also appears in the mean number of provisioned registries in Table 5.6, in which only the use case with four replicas has more provisioned registries than the Comm+ version. On the other hand, with more unique images in the infrastructure, the MODyn approach requires more time to start decreasing the number of provisioned registries, and it still has more provisioned registries than the Comm+ version when a higher number of replicas is required (3 or 4 container image replicas). To finish the evaluation regarding resource usage, we present the normalized resource usage of the edge servers over time. Equa-

tion 5.2 shows the normalized resource usage calculation for a given time step t, which considers the computing resources (CPU and RAM).

$$geo\_mean(\sum_{i=1}^{|E|} \frac{cpu\_demand(i,t)}{c_i}, \sum_{i=1}^{|E|} \frac{ram\_demand(i,t)}{r_i}) \tag{5.2}$$

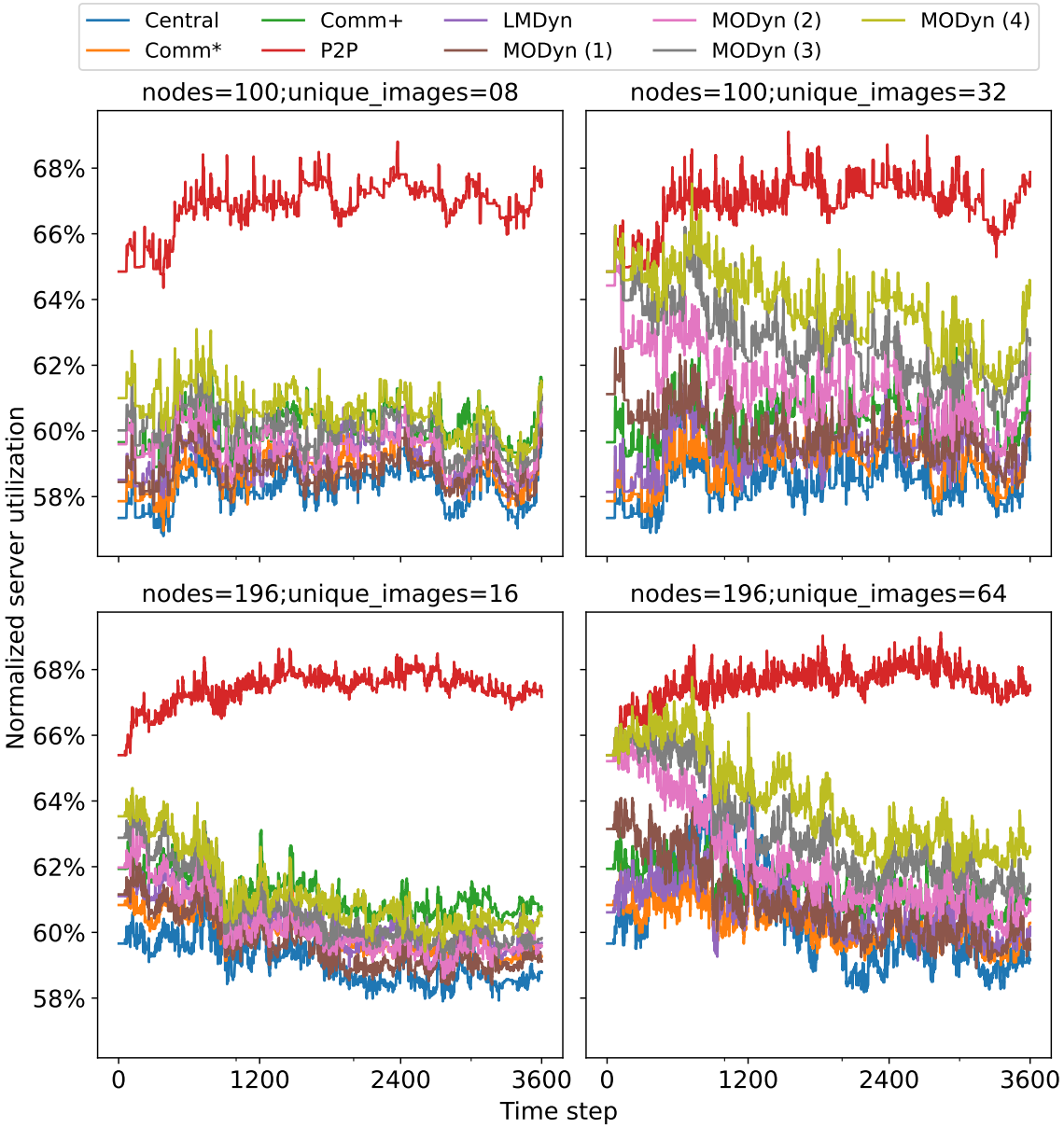


Figure 5.7: Mean normalized server utilization per time step along the simulation time.

Generally, the number of provisioned registries impacts the server utilization the most. Both graphs for each variation follow the same pattern (Figures 5.6 and 5.7). The most significant difference in server utilization is the slight changes that occur because of the reallocations. During an application's reallocation, it occupies CPU and RAM resources in two edge servers, the current one and the target server of the reallocation. The current server continues to allocate the application to avoid downtime to the end user, and the

target server needs to reserve the resources to ensure a successful reallocation when it receives the application container image.

5.3.3 Provisioning Time

Looking at the relation between latency and provisioning time, lowering the provisioning time might help decrease the overall mean latency because the applications are reallocated faster, and thus their latency to the end users decreases faster. However, the use of container registry resources plays a significant role in this scenario, turning these two metrics into conflicting metrics. Although allocating more registries decreases the provisioning time, it might affect the latency of the applications. Table 5.7 depicts the overall mean provisioning time in seconds. This metric denotes the mean time it takes to download a container image from a registry to a target server to reallocate an application.

Table 5.7: Overall mean provisioning time in seconds (Equation 4.10).

Nodes	100		196	
Unique images	08	32	16	64
Central	*1.18*	*3.31*	*3.18*	*16.80*
Comm*	*1.07*	*2.78*	*2.19*	*5.39*
Comm+	0.95	2.50	1.91	4.54
P2P	0.88	**2.26**	**1.63**	**4.14**
LMDyn	0.92	*2.73*	1.94	*5.46*
MODyn (1)	0.97	*2.75*	2.11	*5.56*
MODyn (2)	0.91	2.42	1.96	4.87
MODyn (3)	**0.85**	2.30	1.72	4.56
MODyn (4)	0.86	2.31	1.69	4.45

The main pattern observed with respect to provisioning time is that it decreases as the number of provisioned registries increases. In this sense, the P2P strategy offers the lowest provisioning times by allocating more registries over time. Following the P2P strategy, the MODyn approach with 3 or 4 replicas and the Comm+ version present the best results. The remaining strategies, including LMDyn and MODyn with 1 or 2 replicas, allocate a limited number of container registries, which limits their ability to decrease the provisioning time of container images. Similarly to the pattern observed with the number of provisioned registries, the provisioning time also differs between the variations with fewer unique images (08 and 16 unique images) and the variations with more unique images (32 and 64 unique images). This behavior is related to the cache hits and misses observed when reallocating applications. To this end, we present Figure 5.8, which depicts the different types of reallocation that occur during the simulation, including (i) only using the cache, (ii) partially using the cache, and (iii) not using the cache. The first type includes

reallocations to edge servers with all layers of the container image in their storage (cache hit), and the second type contains reallocations to edge servers with some layers of the container image in their storage (partial cache hit). The third type includes reallocations that require downloading all the container layers to the target server (cache miss). The second type can happen due to layer sharing between different images and when two applications with the same image are reallocated to the same server.

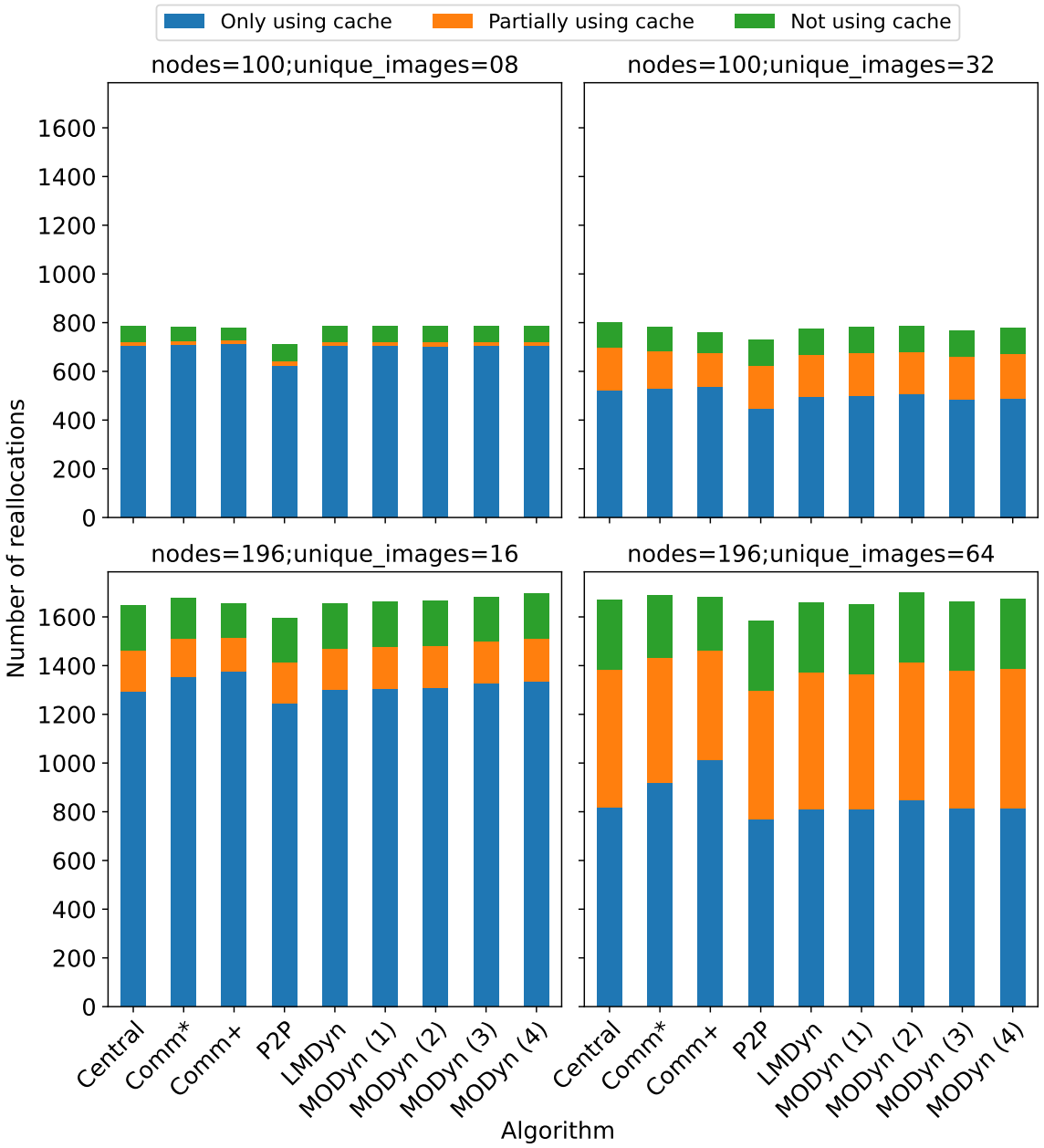


Figure 5.8: Total number of reallocations per type.

Considering these different reallocation types, Table 5.7 depicts a lower provisioning time in the variations with fewer unique container images. A compelling explanation for this behavior is the elevated number of reallocations only using cache in these variations, as depicted in the two subfigures on the left side of Figure 5.8. Meanwhile, in

the other variations, this number is lower, and the application's reallocation requires more network usage, leading to a longer provisioning time.

5.3.4 Storage Resources Usage

So far, the main focus has been to compare the P2P strategy with the dynamic approaches proposed in our work. In addition, as we have seen, the community strategy also has good results in terms of latency and provisioning time, besides having controlled resource usage considering computing resources (CPU and RAM) because the number of registries is constant and limited. However, allocating fully replicated registries instead of relying on the existing cache of the edge servers makes the community approach demand a lot of storage space. This behavior can become a constraint for adopting the community registry approach, especially in edge computing infrastructures, as highlighted by Carvalho et al. [12]. To understand the impact of this behavior in our experiments, we present the mean disk utilization per edge server and time step in Table 5.8 and the total disk utilization of the infrastructure along the simulation time steps in Figure 5.9.

Table 5.8: Mean disk utilization per edge server and time step in MiB (Equation 4.12).

Nodes	100		196	
Unique images	08	32	16	64
Central	7072.55	12330.50	11625.24	19738.34
Comm*	7206.98	13978.16	12599.16	25591.02
Comm+	*7469.51*	*16816.42*	*13837.69*	*33091.04*
P2P	7119.09	12401.09	11203.21	18796.29
LMDyn	7101.45	12317.10	11582.04	19607.08
MODyn (1)	7089.41	12283.91	11672.04	19648.21
MODyn (2)	7081.26	12343.06	11695.82	19695.32
MODyn (3)	7090.27	12389.74	11657.26	19663.90
MODyn (4)	7089.23	12318.27	11730.25	19650.14

Disk utilization also follows the pattern with a relevant difference between variations with fewer unique images (08 and 16 unique images) and variations with more unique images (32 and 64 unique images). In this case, the pattern is directly related to the fact that fewer unique container images demand less storage space on servers with fully replicated registries. Meanwhile, with more unique container images, the storage space required is larger, increasing disk utilization, especially in the community registry approach. The mean disk utilization per edge server and time step reaches 68% more than the other strategies in the Comm+ version. Disk utilization is lower in the Comm* version, but still higher than in the other approaches. Furthermore, the uniform increase in overall disk utilization over time, shown in Figure 5.9 on variations with 32 and 64 unique

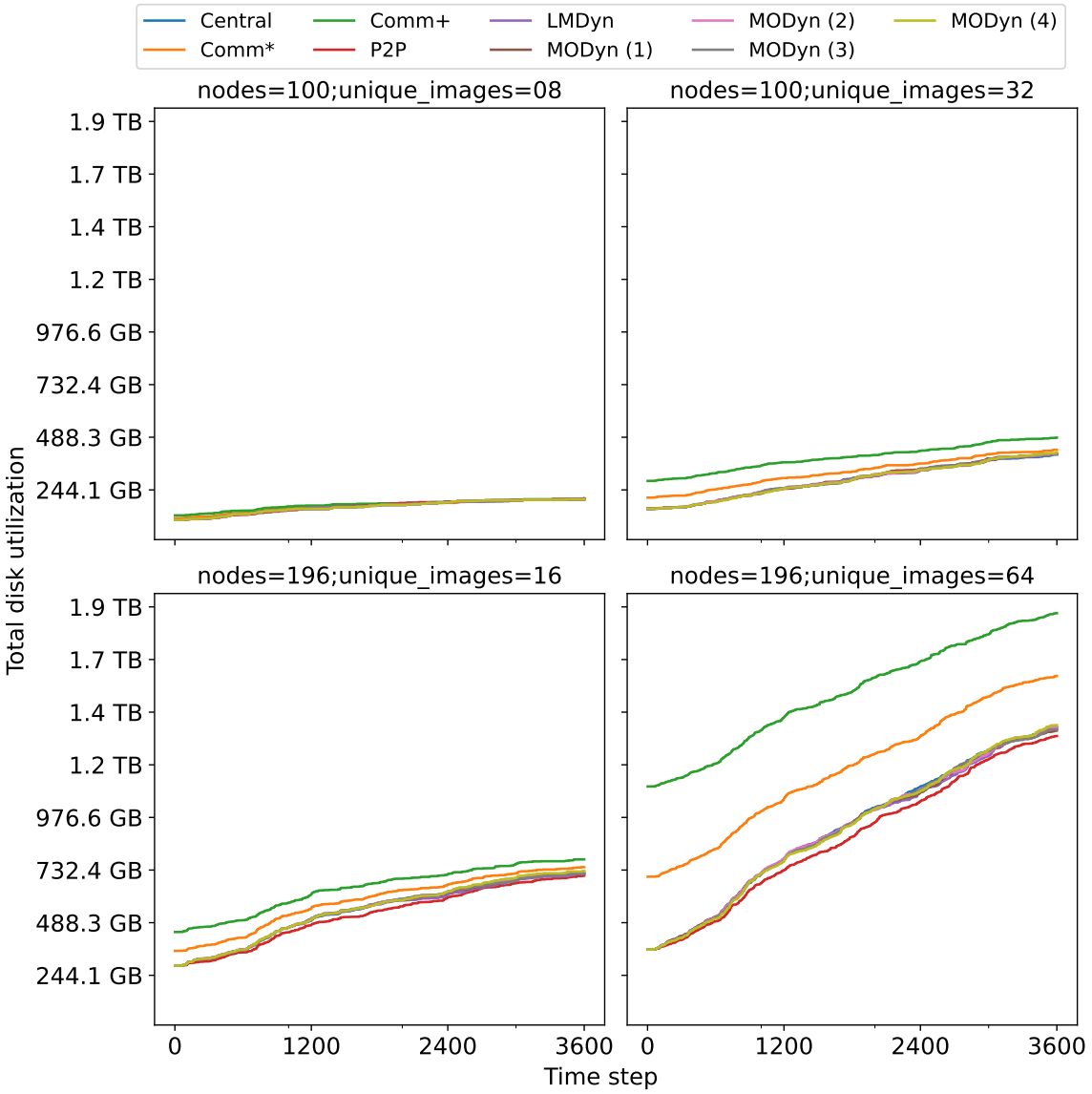


Figure 5.9: Total disk utilization per time step along the simulation time.

container images, indicates that the community strategy is not scalable in this regard. Our dataset contains only a maximum disk occupation of 79 GiB, as depicted in Table 5.3. However, in a scenario with more unique container images and larger container images (e.g., AWS Deep Learning container images for GPUs), this elevated disk utilization might be a significant constraint to use this strategy on edge computing infrastructures.

5.3.5 Additional Metrics

In addition to the main metrics targeted by this work (i.e., latency, provisioning time, and resource usage), we also analyze some secondary metrics to assess the effectiveness of dynamic registry provisioning approaches. In this sense, we start analyzing

the time the registries spend active (i.e., provisioning container images to other edge servers). We summarize the results with respect to this metric in Figure 5.10.

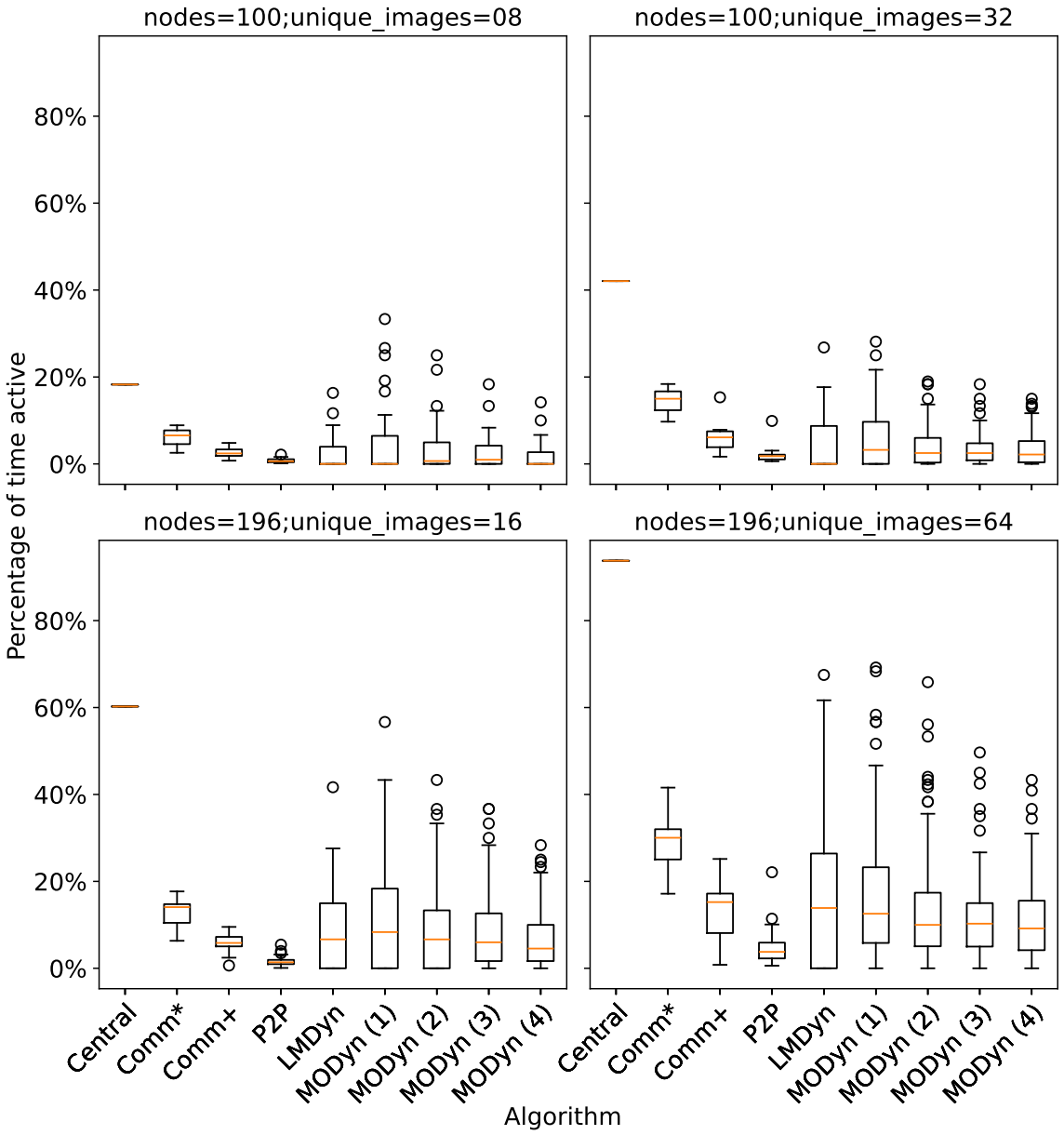


Figure 5.10: Distribution of the time percentage that the provisioned registries spend active.

One of the observed patterns is the duality of active time between the central registry and the P2P registries. The central registry strategy only has a single entity that reaches 90% of the simulated time active in the variation with the highest workload (196 nodes and 64 unique container images). On the other hand, we have seen that the P2P registry strategy allocates an excessive number of registries, which distribute the demands and present a low mean of time active, with a few outliers in the variations with more unique container images (32 and 64). In between, the community registry strategy achieves an adequate percentage of active time, especially in the Comm* version, due to the lower number of provisioned registries.

The dynamic registry provisioning strategy is also between the central and P2P strategies. It has a controlled number of provisioned registries, but is generally higher than the community strategy. By limiting the number of provisioning registries, these versions achieve a higher percentage of active time. Aside from the mean percentage of time active, calculated considering all provisioned registries, some outliers (circles above the boxes) indicate that some registries spend more time active than the other strategies. However, since we do not take any decision based on predicting user mobility or something similar, it still happens to provision registries that have no activity or near-zero activity.

The next metric that we analyze is related to container image replication. We compare P2P and dynamic registry strategies because we only consider cache-based registries to calculate the replication value. Thus, we do not consider the container image hosted in the central registry provisioned in the dataset used by these strategies. The analysis of this metric aims to understand the behavior of the distribution of container image replicas during the simulation. Figure 5.11 summarizes this information for the variation with 100 nodes and 32 unique container images, which behaves similarly to other variations.

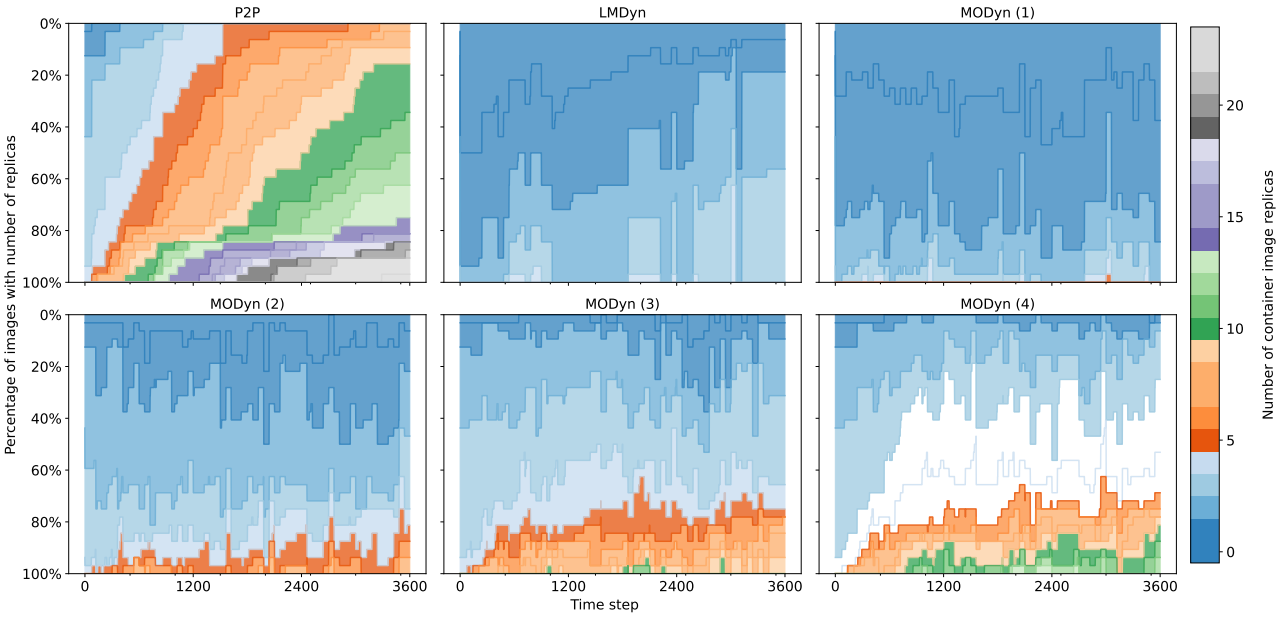


Figure 5.11: Number of container image replicas along the simulation time (variation 2).

Although both dynamic registry provisioning strategies have a similar color map, indicating similar behavior, the P2P strategy is significantly different. The P2P strategy's behavior is to have an increasing number of container image replicas available for provisioning on P2P registries (i.e., cache-based registries). This happens because the number of provisioned registries only grows as the simulation time passes, and the container images keep getting distributed to these different edge servers based on user mobility.

Having more than 80% container images with ten or more replicas for distribution in the infrastructure shows the waste of resources that is part of this strategy.

On the other hand, due to the controlled number of P2P registries provisioned during the simulation by the dynamic registry provisioning variations, the number of container image replicas is sufficient to provide the necessary fault tolerance and avoid bottlenecks during container image provisioning. Although the LMDyn approach does not have an image replication parameter, the number of provisioned registries limits the number of replicas in the infrastructure. In the MODyn approach, there are slight changes from one input to another because the input is directly related to the replication of container images. In these different inputs, most container images have at least the minimum number of specified replicas. However, we cannot ensure that all container images have the specified number of replicas because the algorithm skips edge servers to which most images reach the target. On the other hand, we cannot ensure that all container images do not overcome the specified number of replicas. However, the number of replicas is significantly lower even in the input with four replicas.

The results presented in this section corroborate many of the hypotheses about existing strategies and the possibility of dynamically provisioning and deprovisioning container registries in edge computing infrastructures. In the next chapter, we conclude this work by making some final observations about the advantages and disadvantages of the evaluated strategies by considering the different scenario variations that we targeted in our experiments. Additionally, we analyze the remaining research opportunities related to edge computing environments, especially with regard to the registry provisioning process.

6. CONCLUSION

The increasing number of mobile devices and sensors is leading to the decentralization of computing resources. The consolidated cloud computing paradigm struggles to cope with the latency and bandwidth requirements of emerging applications based on these devices, mainly due to the WAN distance that separates the data centers from the devices. In this sense, the edge computing paradigm emerges as an alternative to allocate these latency and bandwidth demands, with substantial computing and storage resources distributed across urban areas. This aspect allows the edge infrastructure to ensure the necessary QoS and QoE in terms of latency and bandwidth for its users, including mobile users. However, edge infrastructures can use container-based virtualization to meet these demands and limited resource availability.

Containers offer a lightweight virtualization aspect compared to VMs, which are known to virtualize resources in cloud infrastructures. To provision container-based applications, providers and users rely on the container registry to download the necessary content. However, this entity could become a bottleneck [23], especially in edge computing infrastructures. In this regard, previous studies focused on distributing container registries on multiple servers. These efforts, aimed at cloud and edge infrastructures, improved the process of downloading container images for provisioning container-based applications. However, until now, the sole focus of most of these works was only the provisioning time. Although Temp et al. [52] propose a strategy that also couples with other relevant metrics to our study (latency and resource usage), the authors employ a strategy of migrating container registries, which is not scalable to greater storage demands.

To this end, we propose two algorithms for dynamically provisioning container registries in edge computing infrastructures. These infrastructures are responsible for the allocation of latency-sensitive applications and are known to have limited resource availability. Thus, the first algorithm (LMDyn) is a preliminary proposal within our work and is focused only on edge computing requirements: latency and resource usage. After recognizing the importance of considering the provisioning time, as previous studies on distributed container registries did, we propose the second algorithm (MODyn). MODyn is the most recent proposed strategy, and it tries to allocate container registries by considering a trade-off between the target metrics. A significant advantage of both strategies is that they rely on the container images already stored in edge servers' storage without requiring any migration of container images.

As depicted in Chapter 5, both algorithms have adequate performance compared to baseline approaches, especially in some specific use cases. Although the LMDyn algorithm struggles with the provisioning time because of the limited number of provisioned registries, it maintains a similar latency to other strategies and considerably reduces the resource usage in terms of CPU and RAM to the P2P strategy and in terms of disk to the

community strategy. The MODyn algorithm has similar results, with better performance with respect to the provisioning time. Although it does not reach the performance of the P2P strategy with respect to these metrics, it also reduces the usage of CPU and RAM resources. Compared to the community strategy, it significantly reduces disk utilization, especially in scenarios with more unique container images, and keeps a better provisioning time with a higher replication target. In addition, disk utilization in the community strategy indicates that this strategy is more suitable for use cases with a limited variety of container images and layers.

6.1 Future Work

Taking into account the results of this work, we believe that a strategy that analyzes the mobility patterns of the user to predict his next movements can significantly improve the effective provisioning of container registries. Such a strategy could reduce the number of provisioned registries and improve the time each of the provisioned registries spends active, leading to better resource usage and a minor latency impact. Within the MODyn algorithm, we also understand that there are opportunities to improve the strategy using the same logic of provisioning container registries until it reaches a replication target. For instance, we could download relevant container images to a server that is considered an adequate candidate to allocate a registry but requires a few additional container images to improve its capacity for provisioning container images. In the same vein, we could also remove irrelevant container images from good server candidates to avoid excessive image replication or to improve disk utilization. In addition, we understand that conducting experiments in other scenarios (e.g., scenarios that require cache/disk replacement strategies because of storage limitations) is crucial to understanding additional bottlenecks in the evaluated strategies.

6.2 Achievements

During our research period, we submitted a contribution containing the LMDyn algorithm to the *XXIV Simpósio em Sistemas Computacionais de Alto Desempenho (WSCAD 2023)*, which is in the conference proceedings.

In addition to the contributions mentioned above, we participated in another study on resource management in federated edge computing infrastructures.

- **Thea - a QoS, Privacy, and Power-aware Algorithm for Placing Applications on Federated Edges**
  Paulo Souza, Carlos Kayser, Lucas Roges, Tiago Ferreto

Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP), 2023

REFERENCES

[1] Ahmed, A.; Ahmed, E. “A survey on mobile edge computing”. In: 10th International Conference on Intelligent Systems and Control (ISCO), 2016, pp. 1–8.

[2] Ahmed, A.; Pierre, G. “Docker image sharing in distributed fog infrastructures”. In: IEEE International Conference on Cloud Computing Technology and Science (CloudCom), 2019, pp. 135–142.

[3] Al-Ayyoub, M.; Husari, G.; Mardini, W. “Improving vertical handoffs using mobility prediction”, *International Journal of Advanced Computer Science and Applications*, vol. 7–3, 2016.

[4] Alibaba Cloud. “P2p-based intelligent image acceleration system of dragonfly”. Source: https://www.alibabacloud.com/blog/p2p-based-intelligent-image-acceleration-system-of-dragonfly_599645, Jan 2024.

[5] Anwar, A.; Mohamed, M.; Tarasov, V.; Littley, M.; Rupprecht, L.; Cheng, Y.; Zhao, N.; Skourtis, D.; Warke, A. S.; Ludwig, H.; et al.. “Improving docker registry design based on production workload analysis”. In: 16th USENIX Conference on File and Storage Technologies (FAST 18), 2018, pp. 265–278.

[6] Aral, A.; De Maio, V.; Brandic, I. “Ares: Reliable and sustainable edge provisioning for wireless sensor networks”, *IEEE Transactions on Sustainable Computing*, vol. 7–4, 1 2021, pp. 761–773.

[7] Bai, F.; Helmy, A. “A survey of mobility models”, *Wireless Adhoc Networks. University of Southern California, USA*, vol. 206, 2004, pp. 147.

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

[9] Bernstein, D. “Containers and cloud: From lxc to docker to kubernetes”, *IEEE cloud computing*, vol. 1–3, 9 2014, pp. 81–84.

[10] Bonomi, F.; Milito, R.; Zhu, J.; Addepalli, S. “Fog computing and its role in the internet of things”. In: Proceedings of the first edition of the MCC workshop on Mobile cloud computing, 2012, pp. 13–16.

[11] Bréhon-Grataloup, L.; Kacimi, R.; Beylot, A.-L. “Mobile edge computing for v2x architectures and applications: A survey”, *Computer Networks*, vol. 206, 4 2022, pp. 108797.

[12] Carvalho, G.; Cabral, B.; Pereira, V.; Bernardino, J. “Edge computing: current trends, research challenges and future directions”, *Computing*, vol. 103, 1 2021, pp. 993–1023.

[13] Chen, J. L.; Liaqat, D.; Gabel, M.; de Lara, E. “Starlight: Fast container provisioning on the edge and over the wan”. In: 19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22), 2022, pp. 35–50.

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

[15] Docker. “Docker overview”. Source: https://docs.docker.com/get-started/overview/, Jan 2024.

[16] Docker. “Layers”. Source: https://docs.docker.com/build/guide/layers/, Jan 2024.

[17] Du, L.; Wo, T.; Yang, R.; Hu, C. “Cider: A rapid docker container deployment system through sharing network storage”. In: IEEE 19th International Conference on High Performance Computing and Communications; IEEE 15th International Conference on Smart City; IEEE 3rd International Conference on Data Science and Systems (HPCC/SmartCity/DSS), 2017, pp. 332–339.

[18] FreeBSD. “Chapter 16. jails”. Source: https://docs.freebsd.org/en/books/handbook/jails/, Dec 2022.

[19] Gazzetti, M.; Reale, A.; Katrinis, K.; Corradi, A. “Scalable linux container provisioning in fog and edge computing platforms”. In: European Conference on Parallel Processing, 2017, pp. 304–315.

[20] Gillani, K.; Lee, J.-H. “Comparison of linux virtual machines and containers for a service migration in 5g multi-access edge computing”, *ICT Express*, vol. 6–1, 3 2020, pp. 1–2.

[21] Govindasamy, S.; Bergel, I. “Uplink performance of multi-antenna cellular networks with co-operative base stations and user-centric clustering”, *IEEE Transactions on Wireless Communications*, vol. 17–4, 2 2018, pp. 2703–2717.

[22] Hamdan, S.; Ayyash, M.; Almajali, S. “Edge-computing architectures for internet of things applications: A survey”, *Sensors*, vol. 20–22, 11 2020, pp. 6441.

[23] Harter, T.; Salmon, B.; Liu, R.; Arpaci-Dusseau, A. C.; Arpaci-Dusseau, R. H. “Slacker: Fast distribution with lazy docker containers”. In: 14th USENIX Conference on File and Storage Technologies (FAST 16), 2016, pp. 181–195.

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

[25] Ismail, L.; Materwala, H. “Escove: energy-sla-aware edge–cloud computation offloading in vehicular networks”, *Sensors*, vol. 21–15, 8 2021, pp. 5233.

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

[27] Kaur, G.; Batth, R. S. “Edge computing: Classification, applications, and challenges”. In: 2nd International Conference on Intelligent Engineering and Management (ICIEM), 2021, pp. 254–259.

[28] kernel, L. “Control groups — the linux kernel documentation”. Source: https://docs.kernel.org/admin-guide/cgroup-v1/cgroups.html, Dec 2022.

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

[30] Liang, M.; Shen, S.; Li, D.; Mi, H.; Liu, F. “Hdid: An efficient hybrid docker image distribution system for datacenters”. In: National Software Application Conference, 2016, pp. 179–194.

[31] Ltd., C. “Linux containers”. Source: https://linuxcontainers.org/, Dec 2022.

[32] MacQueen, J. “Classification and analysis of multivariate observations”. In: Proceedings of the 5th Berkeley Symposium on Mathematical Statistics and Probability, 1967, pp. 281–297.

[33] Mansouri, Y.; Babar, M. A. “A review of edge computing: Features and resource virtualization”, *Journal of Parallel and Distributed Computing*, vol. 150, 4 2021, pp. 155–183.

[34] Mell, P.; Grance, T. “The nist definition of cloud computing”. Source: https://www.nist.gov/publications/nist-definition-cloud-computing, Dec 2022.

[35] Merkel, D.; et al.. “Docker: lightweight linux containers for consistent development and deployment”, *Linux j*, vol. 239–2, 3 2014, pp. 2.

[36] Morabito, R.; Cozzolino, V.; Ding, A. Y.; Beijar, N.; Ott, J. “Consolidate iot edge computing with lightweight virtualization”, *IEEE network*, vol. 32–1, 1 2018, pp. 102–111.

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

[38] Ouyang, T.; Zhou, Z.; Chen, X. “Follow me at the edge: Mobility-aware dynamic service placement for mobile edge computing”, *IEEE Journal on Selected Areas in Communications*, vol. 36–10, 9 2018, pp. 2333–2345.

[39] Quy, N. M.; Ngoc, L. A.; Ban, N. T.; Hau, N. V.; Quy, V. K. “Edge computing for real-time internet of things applications: Future internet revolution”, *Wireless Personal Communications*, vol. 132–2, 7 2023, pp. 1423–1452.

[40] Rausch, T.; Hummer, W.; Stippel, C.; Vasiljevic, S.; Elvezio, C.; Dustdar, S.; Krösl, K. “Towards a platform for smart city-scale cognitive assistance applications”. In: IEEE Conference on Virtual Reality and 3D User Interfaces Abstracts and Workshops (VRW), 2021, pp. 330–335.

[41] Roges, L.; Ferreto, T. “Dynamic provisioning of container registries in edge computing infrastructures”. In: Anais do XXIV Simpósio em Sistemas Computacionais de Alto Desempenho, 2023, pp. 85–96.

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

[43] Satyanarayanan, M.; Bahl, P.; Caceres, R.; Davies, N. “The case for vm-based cloudlets in mobile computing”, *IEEE pervasive Computing*, vol. 8–4, 10 2009, pp. 14–23.

[44] Satyanarayanan, M.; Davies, N. “Augmenting cognition through edge computing”, *Computer*, vol. 52–7, 7 2019, pp. 37–46.

[45] Satyanarayanan, M.; Klas, G.; Silva, M.; Mangiante, S. “The seminal role of edge-native applications”. In: IEEE International Conference on Edge Computing (EDGE), 2019, pp. 33–40.

[46] Shakarami, A.; Shakarami, H.; Ghobaei-Arani, M.; Nikougoftar, E.; Faraji-Mehmandar, M. “Resource provisioning in edge/fog computing: A comprehensive and systematic review”, *Journal of Systems Architecture*, vol. 122, 1 2022, pp. 102362.

[47] Shi, W.; Cao, J.; Zhang, Q.; Li, Y.; Xu, L. “Edge computing: Vision and challenges”, *IEEE internet of things journal*, vol. 3–5, 6 2016, pp. 637–646.

[48] Singh, A.; Satapathy, S. C.; Roy, A.; Gutub, A. “Ai-based mobile edge computing for iot: Applications, challenges, and future scope”, *Arabian Journal for Science and Engineering*, vol. 47, 1 2022, pp. 1–31.

[49] Solaris. “Introdução ao solaris zones - guia de administração do sistema: gerenciamento de recursos do oracle solaris containers e oracle solaris zones”. Source: https://docs.oracle.com/cd/E38904_01/html/820-2978/zones.intro-1.html, Dec 2022.

[50] Sonmez, H. “State of the edge report 2023”, Technical Report, The Linux Foundation, 2023, 73p.

[51] Souza, P. S.; Ferreto, T.; Calheiros, R. N. “Edgesimpy: Python-based modeling and simulation of edge computing resource management policies”, *Future Generation Computer Systems*, vol. 148, 11 2023, pp. 446–459.

[52] Temp, D. C.; de Souza, P. S. S.; Lorenzon, A. F.; Luizelli, M. C.; Rossi, F. D. “Mobility-aware registry migration for containerized applications on edge computing infrastructures”, *Journal of Network and Computer Applications*, vol. 217, 8 2023, pp. 103676.

[53] Uber. “Introducing kraken, an open source peer-to-peer docker registry”. Source: https://www.uber.com/blog/introducing-kraken/, Jan 2024.

[54] Varghese, B.; Wang, N.; Barbhuiya, S.; Kilpatrick, P.; Nikolopoulos, D. S. “Challenges and opportunities in edge computing”. In: IEEE international conference on smart cloud (SmartCloud), 2016, pp. 20–26.

[55] Wang, F.; Zhang, M.; Wang, X.; Ma, X.; Liu, J. “Deep learning for edge computing applications: A state-of-the-art survey”, *IEEE Access*, vol. 8, 3 2020, pp. 58322–58336.

[56] Wysocki, T. A.; Dadej, A.; Wysocki, B. J. “Advanced wired and wireless networks”. Springer Science & Business Media, 2004, vol. 26, 270p.

[57] Zhang, M.; Zhang, F.; Lane, N. D.; Shu, Y.; Zeng, X.; Fang, B.; Yan, S.; Xu, H. “Deep Learning in the Era of Edge Computing: Challenges and Opportunities”. John Wiley Sons, Ltd, 2020, chap. 3, pp. 67–78, https://onlinelibrary.wiley.com/doi/pdf/10.1002/9781119551713.ch3.

[58] Zhang, X.; Cao, Z.; Dong, W. “Overview of edge computing in the agricultural internet of things: Key technologies, applications, challenges”, *Ieee Access*, vol. 8, 7 2020, pp. 141748–141761.

[59] Zheng, C.; Rupprecht, L.; Tarasov, V.; Thain, D.; Mohamed, M.; Skourtis, D.; Warke, A. S.; Hildebrand, D. “Wharf: Sharing docker images in a distributed file system”. In: Proceedings of the ACM Symposium on Cloud Computing, 2018, pp. 174–185.

Pontifícia Universidade Católica do Rio Grande do Sul
Pró-Reitoria de Pesquisa e Pós-Graduação
Av. Ipiranga, 6681 – Prédio 1 – Térreo
Porto Alegre – RS – Brasil
Fone: (51) 3320-3513
E-mail: propesq@pucrs.br
Site: www.pucrs.br