Dynamic Provisioning of Container Registries in Edge Computing Infrastructures

Lucas Roges[1], **Tiago Ferreto**[1]

[1]School of Technology
Pontifical Catholic University of Rio Grande do Sul (PUCRS), Porto Alegre, Brazil

roges.lucas@edu.pucrs.br, tiago.ferreto@pucrs.br

Abstract. *The emergence of applications with latency-sensitive demands highlighted some limitations of cloud computing and led to the advent of edge computing. Accompanied by challenges such as mobile users and limited resources, this decentralized computing paradigm is frequently associated with container-based virtualization. However, the traditional container registry has significant limitations in properly provisioning container images on edge infrastructures. Consequently, previous authors focused on improving the application provisioning process by modifying this entity. This paper examines previous attempts at decentralizing container registries in edge computing infrastructures. Based on our initial comparison results, we propose a strategy for dynamically provisioning and deprovisioning container registries based on each server's storage and the infrastructure's application demands. The final results show that our strategy can optimize resource utilization in the infrastructure, but needs adjustments to reach the lowest overall latency for application users.*

1. Introduction

In recent decades, the position of the most dominant computing paradigm has been alternating between centralized and decentralized paradigms [Satyanarayanan 2017]. For a while now, the centralized paradigm of cloud computing has been the predominant choice to handle computing demands across multiple verticals, mainly due to its cost-effectiveness. By offering subscription-based services and using the pay-as-you-go model, cloud computing emerged as the backbone of the modern economy [Buyya et al. 2018]. However, the increasing number of smart devices connected to the Internet has made this traditional computing model no longer sufficient to support today's data processing needs [Cao et al. 2020] since long WAN (wide area network) latencies are a fundamental obstacle to adopting cloud computing to execute resource-intensive applications with strict latency requirements [Satyanarayanan et al. 2009].

In this sense, the edge computing paradigm emerged as a decentralized alternative representing an extension of the cloud infrastructure to locations near end users (e.g., urban centers). Along with the benefits of this physical proximity, such as low latency and less bandwidth congestion, there are several challenges to ensuring adequate Quality-of-Service (QoS) for edge computing users. In this context, handling user mobility [Mao et al. 2017, Rejiba et al. 2019] and orchestrating edge resources [Luo et al. 2021] are fundamental necessities for infrastructure providers due to the several use cases related to mobile entities (e.g., autonomous vehicles) and the limited resources available in such infrastructures, respectively. To fulfill these requirements,

edge computing is often associated with container-based virtualization [Ismail et al. 2015, Mansouri and Babar 2021], which offers low overhead compared to other virtualization techniques (e.g., virtual machines).

Generally, this type of virtualization is achieved through platforms that manipulate low-level components. In Docker [Merkel et al. 2014], the most popular of these platforms, applications are packaged into container images and stored in container registries. The container images are templates with software resources to run a particular application, while the container registries are a repository of container images that handle push and pull requests to receive and send these images to hosts with the Docker daemon. Although the user can rely on cache mechanisms, it depends on the container registry to deploy new applications locally. During this deployment process, image download accounts for 76% of the application provisioning time [Harter et al. 2016], turning the container registry into a significant entity in this process. Consequently, several works focus on enhancing this application provisioning process through modifications to the container registry. In edge computing, most efforts are toward distributing the container registry to obtain a lower application provisioning time.

In this context, this paper analyzes existing optimization strategies for allocating container registries on edge infrastructures and their impact on the latency of mobile users accessing composite applications. More specifically, we compare a static and a dynamic approach against baseline registry provisioning, focusing on their capacity to minimize the overall latency. Based on preliminary results, we propose modifications to the dynamic approach and evaluate the impact of these modifications on the target metrics. The remainder of the paper is organized as follows. Section 2 presents the work related to our paper. Section 3 introduces the edge computing scenario on which our research is based. Section 4 describes how we conducted the experiments of this research. In Section 5, we present the preliminary results comparing existing techniques and, based on these experiments, introduce an algorithm to improve the registry provisioning process in Section 6. In Section 7, we evaluate this algorithm and conclude the paper in Section 8.

2. Related Work

Various approaches focus on improving the container image provisioning process through optimizations to the container registry. This section focuses on work that turns the container registry into a distributed entity.

Aside from the edge computing context, most works present solutions based on peer-to-peer (P2P) protocols, such as BitTorrent (BT). Generally, these solutions present requirements unsuitable for edge computing infrastructures, such as the need for dedicated resources for specific tasks (e.g., controlling the communication between peers and seeds). For example, this is the case for HDID [Liang et al. 2016] and FID [Kangjin et al. 2017], two BT-based approaches to registry provisioning focused on cloud data centers. While HDID encompasses an adaptive strategy that does not build torrents for small image layers due to the overhead of this process, FID focuses on scalability and fault tolerance by presenting an architecture with distributed management entities. In the industry, Kraken [Uber 2022] and Dragonfly [Alibaba 2022], proposed by Uber and Alibaba, respectively, are open-source projects focused on accelerating image provisioning through P2P distribution entities.

In edge computing, the variety of alternative solutions is more extensive. Based on the network bottleneck between edge nodes from the same locality that request common layers to a registry server in the cloud, Gazzetti et al. [Gazzetti et al. 2017] propose a strategy for edge nodes to share these container images, similar to a P2P model. In this solution, every request for a new container image passes through a gateway that checks which layers are already available on other infrastructure nodes and which ones need to be downloaded from the registry in the cloud. Also focused on taking advantage of locally available container images, Becker et al. [Becker et al. 2021] highlight the challenges of adopting BT-based solutions in edge computing environments. Thus, the authors propose EdgePier, a P2P image distribution tool based on the InterPlanetary File System (IPFS), which is not dependent on centralized or dedicated services, such as BT. In their scenario, the container registry in the cloud starts provisioning container images, and then, through IPFS, nodes can share container images among themselves.

Considering that the image provisioning process occurs entirely in the edge infrastructure, Knob et al. [Knob et al. 2021] look into the problem of decreasing the application deployment latency on such infrastructures. To avoid significant alterations in the container orchestration process, the authors propose a community-based heuristic solution to find the best nodes for placing fully replicated container registries. With distributed nodes acting as container registries, the heuristic decreases the chances of network bottleneck for provisioning container images from a distant node. Compared to previous strategies aimed at edge environments, this strategy has a static aspect (i.e., container registries are provisioned once with the same set of container images each).

Given our focus on the edge computing scenario, we consider the community registry strategy and a naive version of the P2P registry strategy to provision container registries in our experiments. We also consider a baseline registry provisioning approach, which does not use distributed resources. Our primary interest is understanding each strategy's behavior and getting insights into new improvements for registry allocation decisions. In addition to previous works, we consider the impact on resource utilization of allocating multiple container registries in the infrastructure and its applications. This aspect is disregarded, or at least implicit in the previous works.

3. System Model

This section presents the edge computing scenario in which container registry provisioning strategies are evaluated with respect to their capacity to minimize overall latency when transferring container images to reallocate composite applications in the infrastructure. The following paragraphs describe the notation representing the infrastructure elements and their relationships. This notation is summarized in Table 1.

The edge infrastructure comprised in this paper is based on the WAN infrastructure of cellular networks [Klas 2017]. There is a set of base stations $\mathcal{B}$ modeled as $\mathcal{B}_n = \{a_n\}$, in which a_n represents the wireless latency for the users connected to $\mathcal{B}_n$. These base stations are responsible for wireless connectivity in a particular area, represented as a hexagonal cell, based on the map model of Aral et al. [Aral et al. 2021]. Furthermore, there is a set of network links ($\mathcal{L}$) to interconnect the base stations and allow communication between distant entities. In this set $\mathcal{L}$, each link is modeled as $\mathcal{L}_f = \{b_f, l_f\}$, representing the link bandwidth (b_f) and link latency (l_f).

Coupled with some base stations, there is a set of edge servers $\mathcal{E}$. Each edge server is modeled as $\mathcal{E}_i = \{c_i, r_i, d_i\}$, with the three attributes representing a capacity of the server: c_i represents the CPU capacity, r_i represents the RAM capacity, and d_i represents the disk capacity. Among the entities that take advantage of the edge server's capacity, the container registries $\mathcal{R}$ are each modeled as $\mathcal{R}_l = \{w_l, g_l\}$. Both attributes represent the demands of the registry: w_l is the CPU demand, and g_l is the RAM demand. Furthermore, the registry placement matrix is represented by $x_{i,l,t}$ and is detailed in Equation 1.

$$x_{i,l,t} = \begin{cases} 1 & \text{if edge server } \mathcal{E}_i \text{ hosts registry } \mathcal{R}_l \text{ at time step } t \\ 0 & \text{otherwise.} \end{cases} \tag{1}$$

A set of users $\mathcal{U}$ is distributed across the infrastructure, where each user accesses a single application from $\mathcal{A}$, modeled as $\mathcal{A}_j = \{u_j, s_j, p_{j,t}, \delta(\mathcal{A}_j, t)\}$. The user that accesses $\mathcal{A}_j$ is represented by u_j, while s_j is an ordered set of services from S that comprise the application. The communication path matrix $p_{j,t}$ is made up of communication paths between u_j and s_j last service, with each path comprising network links that connect these two entities in a given time step t from $\mathcal{T}$. Each path is calculated using Dijkstra's shortest path algorithm [Dijkstra 1959] (link latency as weight), and the application latency, at the time step t, is represented by $\delta(\mathcal{A}_j, t)$. A_j's latency is calculated considering the wireless latency a_n of $\mathcal{B}_n$ (the base station in which u_j is located) and the aggregated latency of the set of network links in $p_{j,t}$, as depicted in Equation 2.

$$\delta(\mathcal{A}_j, t) = a_n + \sum_{\mathcal{L}_f \in p_{j,t}} l_f \tag{2}$$

As described, applications are composed of an ordered service chain. The whole set of services in the infrastructure is represented by $\mathcal{S}$, and each service is modeled as $\mathcal{S}_k\{h_k, d_k, v_k, z_k\}$. The CPU and RAM demands of $\mathcal{S}_k$ are represented by h_k and d_k, respectively, while z_k represents $\mathcal{S}_k$'s container image. Furthermore, $y_{i,k,t}$ is the service placement matrix detailed in Equation 3. The set of container images $\mathcal{C}$ represents the necessary container images to support the applications used in the infrastructure. Each container image has a layered structure with $\mathcal{I}_o$'s set of layers represented by e_o, a subset of layers from $\mathcal{C}$. This organization allows users to use caching mechanisms for entire container images or partially with a subset of container layers.

$$y_{i,k,t} = \begin{cases} 1 & \text{if edge server } \mathcal{E}_i \text{ hosts service } \mathcal{S}_k \text{ at time step } t \\ 0 & \text{otherwise.} \end{cases} \tag{3}$$

Our experiments aim to understand the capacity of each registry provisioning strategy to minimize overall latency throughout time (Equation 4), a crucial demand in edge environments. As depicted in this equation, we consider the mean latency of all applications during a set of time steps to calculate this target metric. Additionally, we consider that the constraints of Equations 5 and 6 should be met. These constraints ensure that each service is placed only once on a single edge server and that the edge servers' capacities are not exceeded, respectively, during all time steps of the set $\mathcal{T}$.

$$\text{Min: } \frac{\sum_{j=1}^{|\mathcal{A}|} \delta(\mathcal{A}_j, t), \forall t \in \{1, ..., |\mathcal{T}|\}}{|\mathcal{A}| * |\mathcal{T}|} \tag{4}$$

subject to:

$$\sum_{i=1}^{|\mathcal{E}|} y_{i,k,t} = 1 \qquad \forall k \in \{1, ..., |\mathcal{S}|\},\ \forall t \in \{1, ..., |\mathcal{T}|\} \tag{5}$$

$$\left[c_i \leqslant \sum_{l=1}^{|\mathcal{R}|} t_l \cdot x_{i,l,t} + \sum_{k=1}^{|\mathcal{S}|} h_k \cdot y_{i,k,t} \right] + \left[r_i \leqslant \sum_{l=1}^{|\mathcal{R}|} g_l \cdot x_{i,l,t} + \sum_{k=1}^{|\mathcal{S}|} d_k \cdot y_{i,k,t} \right] = 0, \tag{6}$$
$$\forall i \in \{1, ..., |\mathcal{E}|\},\ \forall t \in \{1, ..., |\mathcal{T}|\}$$

4. Methodology

The experiments were performed using EdgeSimPy [Souza et al. 2023b], a simulator for modeling and evaluating resource management policies in the edge. In EdgeSimPy, the entities detailed in Section 3 are agents interacting with each other and the environment over time according to communication and decision-making rules. The simulator's paper includes a verification process to show the correctness of EdgeSimPy's conceptual model.

We consider an infrastructure with a homogeneous mesh network topology [Aral et al. 2021] with 261 links, each with bandwidth = 100 Mbit/s and latency = 5. The wireless latency of the base stations is also 5. There are 24 edge servers on the map distributed with the K-Means algorithm [MacQueen 1967], configured with real CPU and RAM specifications [Ismail and Materwala 2021], uniformly based on three models: (i) 32 cores and 32 GB RAM, (ii) 48 cores and 64 GB RAM, and (iii) 36 cores and 64 GB RAM. In addition, all servers have a default disk size of 128GB.

Container registries are organized according to each registry provisioning strategy. For central and community registry strategies, each registry contains a copy of all container images distributed in the infrastructure, and their positioning is static (i.e., it does not change over time). In the central registry strategy, only one server hosts a registry. At the same time, we define the k number of communities in the community strategy as six (25% of the edge servers have a container registry). For the P2P registry strategy, only one registry has all container images distributed in the infrastructure, and the remaining distribute only container images from services that have been or are currently hosted on that server. In addition, new container registries can be created over time in the P2P strategy. Regarding registry CPU and RAM demands, we consider two variations based on hardware requirements to support the Docker Trusted Registry (DTR)[1]. The variation in minimal requirements demands two cores and 8 GB of RAM. In contrast, the variation of the recommended requirements demands four cores and 16 GB of RAM.

[1] `https://docs.docker.com.zh.xy2401.com/v17.12/datacenter/ucp/2.2/guides/admin/install/system-requirements/`

Table 1. Summary of notations used in this paper.

Symbol	Description
$\mathcal{B}$	Set of base stations
$\mathcal{L}$	Set of network links
$\mathcal{E}$	Set of edge servers
$\mathcal{R}$	Set of container registries
$\mathcal{U}$	Set of users
$\mathcal{A}$	Set of applications
$\mathcal{S}$	Set of services
$\mathcal{I}$	Set of container images
$\mathcal{C}$	Set of container layers
$\mathcal{T}$	Set of time steps
a_n	$\mathcal{B}_n$'s wireless latency
b_f	$\mathcal{L}_f$'s bandwidth
l_f	$\mathcal{L}_f$'s latency
c_i	$\mathcal{E}_i$'s CPU capacity
r_i	$\mathcal{E}_i$'s RAM capacity
d_i	$\mathcal{E}_i$'s disk capacity
w_l	$\mathcal{R}_l$'s CPU demand
g_l	$\mathcal{R}_l$'s RAM demand
$x_{i,l,t}$	Container registry placement matrix
u_j	$\mathcal{A}_j$'s user
s_j	$\mathcal{A}_j$'s service chain
$p_{j,t}$	$\mathcal{A}_j$'s communication path matrix
$\delta(\mathcal{A}_j, t)$	$\mathcal{A}_j$'s latency at time step t
h_k	$\mathcal{S}_k$'s CPU demand
d_k	$\mathcal{S}_k$'s RAM demand
z_k	$\mathcal{S}_k$'s container image
$y_{i,k,t}$	Service placement matrix
e_o	$\mathcal{I}_o$'s container layers

In our simulation, 36 randomly distributed users are moving on the map using the Pathway mobility model [Bai and Helmy 2004]. They move from one base station to another in 60 time steps, and, according to this mobility, applications and their services are reallocated using the Follow User strategy [Yao et al. 2015] to maintain the lowest possible latency. To reallocate a service, the strategy initially checks if the container image or some of its layers are cached on the target server, and the remaining data, if there is any, that must be downloaded from the nearest container registry with the data available, using the minimum network latency as the weight to select the registry. Each user is accessing a different application, and the applications are composed of a different number of services: 1, 2, or 3 services. Each service corresponds to different layers of the application. Services have CPU and RAM demands uniformly defined, as in Souza et al. [Souza et al. 2023a], using the following combinations of values (CPU demand/RAM demand): 2 cores/2 GB, 4 cores/4 GB, 8 cores/8 GB, and 16 cores/16 GB. The container images that comprise these services are extracted from the most popular images in the

Docker Hub. The container images vary between programming languages, stream processing tools, databases, and others.

The users have the exact initial positioning in each registry provisioning strategy's dataset. Their applications are allocated as close as possible, based on their latency, with slight differences due to the registries' resource usage. We executed five seeds for each dataset, each corresponding to different user mobility due to the random seed. This variation aims to understand whether changing how users move on the map has a relevant impact on the target metrics. The bar charts of Sections 5 and 7 represent the mean value of these five executions for each strategy. On the contrary, the line graph represents the execution of a single seed to facilitate visualization. Lastly, this paper's material and algorithms are available in a GitHub repository[2].

5. Preliminary Results

Based on the overall latency results during the simulation, shown in Figure 1, the community registry provisioning strategy has the lowest mean latency using minimal and recommended hardware requirements to host the container registry. Although we expected that the P2P registry provisioning strategy would provide better or similar results regarding mean latency, given its dynamic aspect, the fully replicated and strategically placed registries from the communities were more effective. Observing the number of service reallocations that occurred during the simulation, which is shown in Figure 2, it caught our attention that most of the reallocations are using cached images (i.e., the container image necessary to deploy the service in the reallocation target host is already available on its disk). Furthermore, the community registry provisioning strategy appears to have the best cache awareness among the three strategies, since it has the largest mean number of total service reallocations, but the lowest mean number of reallocations without using the cache (i.e., reallocations that need to download container images from a container registry through the network).

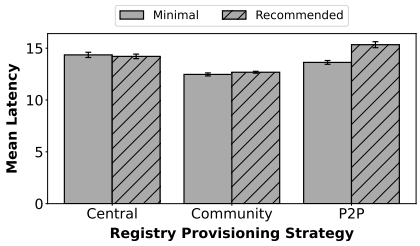

Figure 1. Overall Latency.

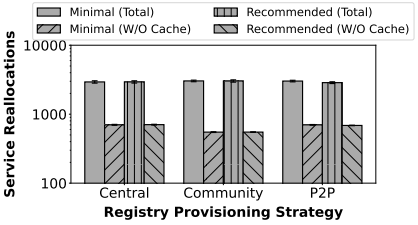

Figure 2. Serv. Reallocations.

Both scenarios of hardware requirements experience lower latency with the community registry provisioning strategy, but the other strategies behave differently. Taking into account the minimal requirements scenario, the P2P registry follows the community registry with a mean latency lower than the central registry. On the other hand, the P2P registry has the worst mean latency when we consider the recommended requirements. This behavior occurs due to the significant resources demanded to allocate a registry when we consider the recommended requirements and the large number of container registries

[2]https://github.com/lucasroges/wscad-registry

allocated in the infrastructure, as shown in Figure 3. When the simulation starts, the applications and services are spread over the edge servers in the infrastructure, meaning that any of these servers that have CPU and RAM capacity to host a container registry will already host one from that point, which means that this strategy allocates a registry in each edge server very early in the simulation.

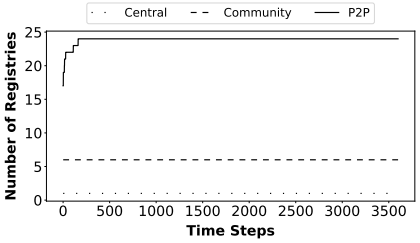

Figure 3. Allocated Registries.

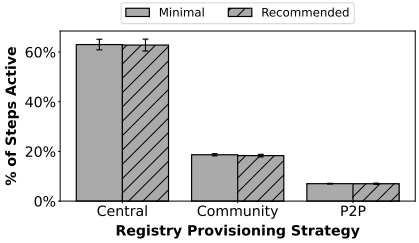

Figure 4. Registry Usage.

Taking into account the distinct amount of registries allocated for each strategy, we also analyzed the usage of these registries during simulation time, as shown in Figure 4. The behavior and mean values are very similar for both hardware requirement scenarios. Since the central registry strategy only allocates a single container registry to provision container images to the entire infrastructure, this registry spends more than 60% of the simulation time actively provisioning container images. Given the distributed aspect of the remaining strategies, this percentage is lower for both. Each community registry spends almost 20% of the simulation time provisioning images, while each P2P registry spends only around 6% of the time in an active state. These values indicate a possible resource underutilization, especially if it impacts the mean latency in the case of the P2P registry.

6. Dynamic Registry Provisioning Algorithm

Focusing on the possible resource underutilization pointed out in the previous section, we extend the P2P registry provisioning with a simple strategy to dynamically provision and deprovision container registries in the edge infrastructure. Compared to P2P registry provisioning, the strategy considers that the use of common hosts as container registries is still possible. However, our goal is to dynamically select only a few hosts to maintain as a container registry until the subsequent execution of the algorithm.

The strategy is depicted in Algorithm 1 and receives the base registry and the current time step as input. Initially, the algorithm filters the edge servers that can host a container registry (line 1). This filter includes both servers already hosting, or not, a registry, but with at least one container image and CPU/RAM resources available. Then, we iterate for each edge server to gather metrics regarding its capacity to host a potentially active container registry. To this end, there is an inner loop over every application in the infrastructure in which we check if the application user is closer to the current server than to the base registry. Since the registry selection for provisioning a container image is based on latency proximity, we use this condition to ignore the next steps for servers distant from the current application user in the inner loop.

For users near the server, we check the percentage of container layers from their services that are already available on the server (line 5). If there is any layer stored, we add

this percentage to the *layer matching* attribute of the edge server ($\mathcal{E}_i[lm]$), and consider that application as a *possible recipient* ($\mathcal{E}_i[pr]$) (lines 6 - 8). After gathering the *layer matching* and *possible recipients* metrics for the edge servers in $\mathcal{E}'$, we select the servers with the highest number of possible recipients and layer matching to the applications in the infrastructure, sequentially, using the average of each of these metrics as the cutoff (lines 9 - 12). Lastly, we provision registries on qualified servers that do not have a registry yet (lines 14 - 15), and we deprovision registries from unqualified servers (lines 16 - 17).

Algorithm 1: Dynamic Registry Provisioning Algorithm.

Input: $\mathcal{R}_c$ – base registry (initial registry with all container images)
t – current time step

1 $\mathcal{E}' =$ edge servers with capacity to host a registry
2 **foreach** $\mathcal{E}_i \in \mathcal{E}'$ **do**
3 **foreach** $\mathcal{A}_j \in \mathcal{A}$ **do**
4 **if** u_j *is closer to* $\mathcal{E}_i$ *than to* $\mathcal{R}_c$ **then**
5 $lm =$ percentage of $\mathcal{A}_j$'s service layers that are in $\mathcal{E}_i$'s disk
6 **if** $lm > 0$ **then**
7 $\mathcal{E}_i[lm] += lm$
8 $\mathcal{E}_i[pr] += 1$
9 $pr_{mean} = int(\frac{\sum_{i=1}^{|\mathcal{E}'|} \mathcal{E}_i[pr]}{|\mathcal{E}'|})$
10 $\mathcal{E}'' =$ edge servers in $\mathcal{E}'$ with possible recipients above pr_{mean}
11 $lm_{mean} = \frac{\sum_{i=1}^{|\mathcal{E}''|} \mathcal{E}_i[lm]}{|\mathcal{E}''|}$
12 $\mathcal{E}''' =$ edge servers in $\mathcal{E}''$ with layer matching above lm_{mean}
13 **foreach** $\mathcal{E}_i \in \mathcal{E}'$ **do**
14 **if** $\mathcal{E}_i \in \mathcal{E}'''$ && $\sum_{l=1}^{|\mathcal{R}|} x_{i,l,t} == 0$ **then**
15 Provision a registry on $\mathcal{E}_i$
16 **if** $\mathcal{E}_i \notin \mathcal{E}'''$ && $\sum_{l=1}^{|\mathcal{R}|} x_{i,l,t} == 1$ **then**
17 Deprovision the registry allocated on $\mathcal{E}_i$

7. Final Results

Figure 5 shows that the strategy from Algorithm 1 improves the mean latency compared to the P2P registry strategy, especially considering the recommended hardware requirements. However, the community registry provisioning strategy still presents an advantage in this target metric. According to Figure 6, the number of service reallocations without using the cache is similar between P2P and dynamic registry strategies, so the community registry strategy also has this advantage against the proposed strategy. The dynamic strategy performs significantly better than the P2P registry strategy with respect to the number of allocated registries and registry usage. Generally, the number of registries allocated by this strategy is often lower than the number of registries of the community registry strategy (Figure 7), while the percentage of time steps that each registry spends in an active state is near 15% (Figure 8), both metrics indicating a lower resource underutilization.

Therefore, dynamically provisioning/deprovisioning container registries is more efficient than P2P registries. Over time, increasing the allocation of edge servers to host container registries demands excessive resources in the P2P registry strategy and becomes a bottleneck impacting overall latency. In contrast, the community registry provisioning

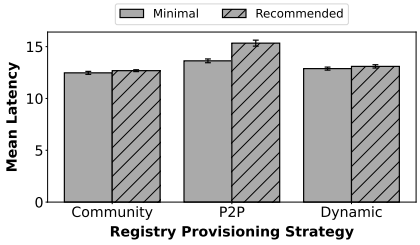

Figure 5. Overall Latency.

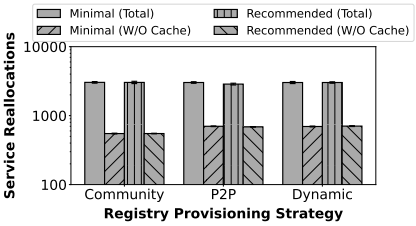

Figure 6. Serv. Reallocations.

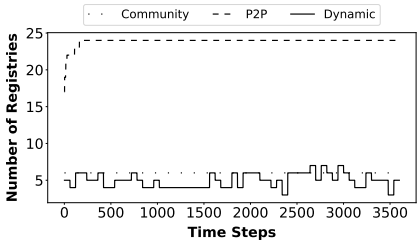

Figure 7. Allocated Registries.

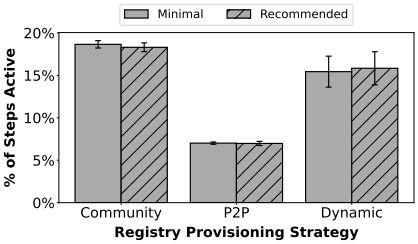

Figure 8. Registry Usage.

strategy has other advantages, such as cache awareness due to the complete replication of container images on the servers hosting registries, which leads applications to take advantage of these cached images in multiple locations of the infrastructure and avoid network traffic to decrease the overall latency.

8. Conclusions

Edge computing emerged with the primary goal of supporting applications that are latency-sensitive and compute-intensive [Satyanarayanan et al. 2009, Satyanarayanan 2017]. Due to the constraints of edge infrastructures, container-based virtualization is considered the most adequate virtualization technique [Ismail et al. 2015, Mansouri and Babar 2021]. As an essential element of the container ecosystem, the container registry has been the target of multiple optimizations to improve the application provisioning process. In this context, we evaluated techniques for distributing container registries on edge computing infrastructures regarding their capacity to minimize overall latency.

Although the results indicate that the community registry strategy can perform better than the P2P registry, we identified potential improvements to the latter's performance in our scenario. Aiming at one of these potential improvements (resource underutilization), we propose a simple strategy to verify the impact of resource usage on the overall latency. The results show that better resource utilization leads to lower overall latency, but other aspects, such as cache awareness, might also play a relevant role in this metric. For future work, we aim to investigate possible improvements to the proposed strategy, such as incorporating more metrics from the infrastructure to refine the server selection for hosting registries. We also aim to associate dynamic registry provisioning with cache awareness since this feature seems to have a significant impact on maintaining the community registry provisioning as the strategy with the lowest overall latency, besides testing the strategies with different datasets (e.g., scenarios with lower cache impact).

9. 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.

References

Alibaba (2022). dragonflyoss/dragonfly2: Dragonfly is an intelligent p2p based image and file distribution system, it also provides a variety of enterprise-level (efficiency, stability, safety, low-cost) product features.

Aral, A., De Maio, V., and Brandic, I. (2021). Ares: Reliable and sustainable edge provisioning for wireless sensor networks. *IEEE Transactions on Sustainable Computing*, 7(4):761–773.

Bai, F. and Helmy, A. (2004). A survey of mobility models. *Wireless Adhoc Networks. University of Southern California, USA*, 206:147.

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

Buyya, R., Srirama, S. N., Casale, G., Calheiros, R., Simmhan, Y., Varghese, B., Gelenbe, E., Javadi, B., Vaquero, L. M., Netto, M. A., et al. (2018). A manifesto for future generation cloud computing: Research directions for the next decade. *ACM computing surveys (CSUR)*, 51(5):1–38.

Cao, K., Liu, Y., Meng, G., and Sun, Q. (2020). An overview on edge computing research. *IEEE access*, 8:85714–85728.

Dijkstra, E. (1959). A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(1):269–271.

Gazzetti, M., Reale, A., Katrinis, K., and Corradi, A. (2017). Scalable linux container provisioning in fog and edge computing platforms. In *European Conference on Parallel Processing*, pages 304–315. Springer.

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

Ismail, B. I., Goortani, E. M., Ab Karim, M. B., Tat, W. M., Setapa, S., Luke, J. Y., and Hoe, O. H. (2015). Evaluation of docker as edge computing platform. In *2015 IEEE conference on open systems (ICOS)*, pages 130–135. IEEE.

Ismail, L. and Materwala, H. (2021). Escove: energy-sla-aware edge–cloud computation offloading in vehicular networks. *Sensors*, 21(15):5233.

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

Klas, G. (2017). Edge computing and the role of cellular networks. *Computer*, 50(10):40–49.

Knob, L. A. D., Faticanti, F., Ferreto, T., and Siracusa, D. (2021). Community-based placement of registries to speed up application deployment on edge computing. In *2021 IEEE International Conference on Cloud Engineering (IC2E)*, pages 147–153. IEEE.

Liang, M., Shen, S., Li, D., Mi, H., and Liu, F. (2016). Hdid: An efficient hybrid docker image distribution system for datacenters. In *National Software Application Conference*, pages 179–194. Springer.

Luo, Q., Hu, S., Li, C., Li, G., and Shi, W. (2021). Resource scheduling in edge computing: A survey. *IEEE Communications Surveys & Tutorials*, 23(4):2131–2165.

MacQueen, J. (1967). Classification and analysis of multivariate observations. In *Proceedings of the 5th Berkeley Symposium on Mathematical Statistics and Probability*, pages 281–297.

Mansouri, Y. and Babar, M. A. (2021). A review of edge computing: Features and resource virtualization. *Journal of Parallel and Distributed Computing*, 150:155–183.

Mao, Y., You, C., Zhang, J., Huang, K., and Letaief, K. B. (2017). A survey on mobile edge computing: The communication perspective. *IEEE communications surveys & tutorials*, 19(4):2322–2358.

Merkel, D. et al. (2014). Docker: lightweight linux containers for consistent development and deployment. *Linux j*, 239(2):2.

Rejiba, Z., Masip-Bruin, X., and Marín-Tordera, E. (2019). A survey on mobility-induced service migration in the fog, edge, and related computing paradigms. *ACM Computing Surveys (CSUR)*, 52(5):1–33.

Satyanarayanan, M. (2017). The emergence of edge computing. *Computer*, 50(1):30–39.

Satyanarayanan, M., Bahl, P., Caceres, R., and Davies, N. (2009). The case for vm-based cloudlets in mobile computing. *IEEE pervasive Computing*, 8(4):14–23.

Souza, P., Kayser, C., Roges, L., and Ferreto, T. (2023a). Thea-a qos, privacy, and power-aware algorithm for placing applications on federated edges. In *2023 31st Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, pages 136–143. IEEE.

Souza, P. S., Ferreto, T., and Calheiros, R. N. (2023b). Edgesimpy: Python-based modeling and simulation of edge computing resource management policies. *Future Generation Computer Systems*.

Uber (2022). uber/kraken: P2p docker registry capable of distributing tbs of data in seconds.

Yao, H., Bai, C., Zeng, D., Liang, Q., and Fan, Y. (2015). Migrate or not? exploring virtual machine migration in roadside cloudlet-based vehicular cloud. *Concurrency and Computation: Practice and Experience*, 27(18):5780–5792.