Community-based Placement of Registries to Speed up Application Deployment on Edge Computing

Luis Augusto Dias Knob*†, Francescomaria Faticanti‡§, Tiago Ferreto*, Domenico Siracusa‡
*Pontifícia Universidade Católica do Rio Grande do Sul, Porto Alegre, Brazil
†Federal Institute of Education, Science and Technology of Rio Grande do Sul, Sertão, Brazil
‡Fondazione Bruno Kessler, via Sommarive, 18 I-38123 Povo, Trento, Italy
§University of Trento, via Sommarive, 9 I-38123 Povo, Trento, Italy
luis.knob@sertao.ifrs.edu.br, tiago.ferreto@pucrs.br, {ffaticanti, dsiracusa}@fbk.eu

2021 IEEE International Conference on Cloud Engineering (IC2E) | 978-1-6654-4970-0/21/$31.00 ©2021 IEEE | DOI: 10.1109/IC2E52221.2021.00029

***Abstract*—The use of virtualization techniques, such as containerization, is rapidly changing how the deployment of applications is performed at the network edge. Indeed, container images enable fast instantiation and small footprint. However, although having smaller size than VMs virtual disks, container images continue to have hundreds of megabytes and can take several seconds to be downloaded in an edge node. In fact, the heterogeneity and resource-constrained infrastructure, typical of an edge computing scenario, can also increase this latency, by the several bottlenecks that may occur on the network topology. We advocate that the use of well-positioned container registries on the topology can significantly improve the deployment process. To prove that, in this paper we focus our analysis on the network requirements of large amounts of container deployments, and the impact generated on two distinct edge topologies. We also present a new registries placement solution based on a fluid communities algorithm. We validated our proposal using simulation and results show that it validates the model and generality of the proposed solution, showing enhanced performance even with biased schedulers with large amounts of deployments in a concentrated set of nodes.**

***Index Terms*—Container Management, Edge Computing, Orchestration, Deployment, Container Registry**

I. Introduction

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

Although having distinct architectures and being standardized by several consortia with different names, like Fog Computing [1] and Multi-Access Edge Computing (MEC) [2], generally speaking, Edge Computing aims at pushing computational capacity closer to the end-user. Despite its similarities with Cloud Computing, this creates a scenario that shows several new challenges in management and orchestration. Today, both academia and industry are working hard to implement solutions that improve the deployment of applications on the edge [3] [4].

Initially implemented on Virtual Machines (VMs) [5], edge applications are rapidly changing to use containerization, enabling faster deployment, smaller footprint, and scalability. Typically, container images are stored on registries as small reusable parts or layers requested by worker nodes when a container image or layer not cached needs to be deployed. This download phase mostly consists of the time needed to instantiate a container, and applications that have a large set of microservices or replicas may require several downloads on different nodes simultaneously.

It is common sense that current container orchestration solutions do not present an optimal method for distributing the applications on Edge Computing. Hence, several works aim at optimizing it by reducing the amount of time needed to instantiate, or diminishing the load generated on the infrastructure, through better usage of the nodes based on peer-to-peer communication [6], distributed caches [7] or new placement algorithms [8]. However, these solutions usually propose many changes in the current orchestration frameworks that are not so easy to achieve in a multi-tenant infrastructure with several distinct actors. Moreover, no one of them considers the importance of the registry placement in this distribution and the load that it generates on the infrastructure.

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

To the best of our knowledge, this work is the first to consider the impact of the application deployment in large-scale distributed topologies, exploring the possible bottlenecks that this network load can generate in the infrastructure. Besides, it is also the first attempt to tackle the challenges of registries placement in an edge computing environment

978-1-6654-4970-0/21/$31.00 ©2021 IEEE
DOI 10.1109/IC2E52221.2021.00029

using infrastructure's constraints, seeking to decrease the total amount of time needed, namely the deployment latency, to fully implement these services on the topology.

The remainder of the paper is organized as follows. In Section II related work is presented. In Section III, we describe the problem formulation following by the algorithm solution in Section IV. The simulation results are presented in Section V. Finally, Section VI concludes the paper.

II. Related Work

Current solutions that seek to decrease the container deployment latency distributed their efforts on several distinct areas, mainly with large modifications on the container runtime [9] or new service that needs to be added topology [10].

Some approaches propose improvements on how the containers are stored in or distributed to nodes. Pulling the same image by multiple registries simultaneously is presented in [6], which also discusses a cooperative implementation of a set of registries. Solutions based on peer-to-peer communication are also presented in [11] [12]. However, these implementations rely on powerful worker nodes placed on high-speed networks, usually not replicable on edge scenarios, with geo-distributed topologies. Furthermore, adding a new daemon on a resource-constrained node can generate bottlenecks, as shown in [13], where even simple applications running on it can generate high latency in the deployment of new applications.

The use of distributed file systems to share images between several nodes is presented with slight changes in [14], and [15]. In both papers, the nodes share a given folder that acts as a centralized cache for the images. This implementation usually shows a significant improvement on the total amount of time needed to instantiate a new container, together with a small redundancy on the layers, since all the shared images are stored in the same place. However, that solution also relies on high-speed networks, with negligible downtime and near-allocated nodes.

There are also proposals of new placement algorithms focusing on geo-distribution [16], and the sharing of microservices between cloud and edge [8] to decrease the deployment time, using latency or bandwidth as input to define the application's placement. Nevertheless, these works' primary objective is to improve the number of containers that can stay active in the topology. However, no one considers the deployment latency on the topology and how this latency can cause congestion on the network and affect the placement results.

In [7], the authors discuss that in edge computing, due to the limited node storage, it is impossible to have many images locally stored. Hence, to improve the deployment latency, it is necessary to retrieve the images on the topology faster. Therefore, they propose a sharing algorithm called KCBP, which finds the closest node with a given layer and uses it to forward it to the requester. Although this paper is the first to discuss the positioning of the layers on the topology and its impact on the deployment latency, its implementation has several limitations. Some of them include the necessity of direct connections between nodes and assumptions that are not feasible in the real world, such as nodes sharing images.

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

Table I: Main notation used throughout the paper

Finally, changes on how the Docker runtime download and start new images are also proposed in [9], where the authors suggest a set of tweaks to improve the several steps executed on the deployment focusing on resource constrained nodes.

III. Problem Formulation

A. System Model

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

B. Registries Placement Problem

This paper's main objective is to design an efficient algorithmic solution for the placement of container registries among the network nodes to speed up the download time in each node from the placed registries. The main objective is represented by the minimization of the maximum download time from the registries to each node of the network. Deeply thinking to this problem, the reader may think to a *vertex k-center problem* [17], where a set of facilities must be deployed on a complete graph in order to minimize the maximum distance between each node and its closest facility. In our case, the concept of distance can be viewed as the download time from a container registry and each node in the network. However, there are some differences concerning the problem we need to solve. The first is the type of graph we are dealing with. Indeed, the vertex k-center requires a complete graph, whilst, in our case, we do not always have such a case. The second main difference is represented by the computation of the download time between each node and the facilities placed on the graph. In fact, this time is not easily computable since it depends on the placement of facilities and the bandwidth occupation of each link leading to a combinatorial explosion of possible configurations.

For this reason, we tackle the problem from a different perspective. We aim to partition the network graph placing a

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

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

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

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

$\forall v \in V$.

2) Objective: The objective function is represented by the standard deviation of the total bandwidth available in each community:

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

where

$$\alpha_k = \sum_{(u,v)|u,v \in V_k} B_{u,v} \quad \forall k \in \mathcal{K}, \tag{2}$$

and

$$\bar{\alpha} = \frac{\sum_{k \in \mathcal{K}} \alpha_k}{K}. \tag{3}$$

Summarizing, the problem we aim to solve is the following

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

subject to:

$$\alpha_k = \sum_{(u,v) \in E} B_{u,v}\, x_{u,k}\, x_{v,k}, \quad \forall k \in \mathcal{K}, \tag{4}$$

$$\alpha_k > 0, \quad \forall k \in \mathcal{K}, \tag{5}$$

$$\sum_{k \in \mathcal{K}} x_{u,k} = 1, \quad \forall u \in V, \tag{6}$$

$$x_{u,k} \in \{0,1\}, \quad \forall u \in V, \forall k \in \mathcal{K}, \tag{7}$$

where constraint (5) ensures that each community has at least two adjacent nodes, i.e., the total bandwidth inside the community is greater than zero. Constraint (6) imposes that each node of the network is assigned to exactly one community.

3) Computational Complexity: Looking at the formulation of Prob. 1, it is easy to see that our problem falls under the class of graph partitioning problems [18]. These particular problems are typically NP-hard requiring approximated solutions to be solved. Furthermore, defining the optimal number of communities to cover all the network graph adds another complexity dimension to the problem. The investigation of the optimal number of communities is left for future works.

IV. Algorithmic Solution

Given the NP-hardness of the problem described in the previous section, we propose a heuristic method based on general search algorithms such as Hill-Climbing [19].

For the graph partition problem we resort on the fluid communities algorithm [20], namely `FluidC`. The algorithm takes in input a graph $G = (V, E)$ and the desired number K of communities. Initially, it randomly selects K vertices from V. These single vertices form the initial K communities. Each community has a density $\delta \in (0, 1]$. For each community k, its density is given by

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

At each step of the algorithm, the community of each vertex is updated and when the assigned communities to vertices do not change for two consecutive steps the procedure terminates. The update rule for each vertex v computes the community with maximum total density within the neighbourhood of v (including v as well). If more than one community is found for a vertex v, then a random community is chosen within the candidate ones [20].

As shown in [20], the main advantage of the `FluidC` algorithm is the scalability. Indeed, this algorithm provides good communities for large graph in reasonable time. However, this kind of solution is not thought to work with weighted graphs. For this reason, we perform a Hill-Climbing search to find the partition of the graph that has the best standard deviation of the total bandwidth available on each community. The pseudocode of our main algorithm is shown in Algorithm 1.

In the first part of Algorithm 1, lines 5-17, the `FluidC` algorithm is repeatedly applied to the input graph until no better communities are found in terms of bandwidth available on each single community. This part follows the Hill-Climbing search approach where we always move towards a better configuration until a termination condition is met. In the second part of the algorithm, lines 18-25, for each community computed in the previous step, a particular node for the registry placement is selected. In this case, the algorithm selects the node with the highest *eigenvector centrality* degree [21]. This kind of measure provides, for each node, an indication about the centrality of the node and the connectivity of the node towards nodes with high centrality degree within the same community.

Time Complexity. The complexity of the algorithm is mainly dominated by the number of iterations required to reach the last local maximum from the starting point (as it can be noticed from Algorithm 1, if after 100 iterations no better move is performed the *while* loop is terminated) in the first part. However, all the operations inside the *while* loop are polynomially

Algorithm 1: `FluidC-Placement`

Input: $G = (V, E)$: network graph; k: desired number of communities

Output: Set of nodes where to place container registries

```
place ← ∅;
final_comm ← ∅;
sd ← +∞;
it ← 0;
while True do
    b̄ ← [];
    {V_k}_{k∈K} ← FluidC(G, K);
    for k ∈ K do
        B_k ← Σ_{(u,v)|u,v∈V_k} B_{u,v};
        b̄.append(B_k);
    if stdv(b̄) < sd then
        final_comm ← {V_k}_{k∈K};
        it ← 0;
    else
        it ← it + 1;
    if it ≥ 100 then
        break;
for k ∈ final_comm do
    c̄ ← [];
    G_k ← (V_k, {(u, v) ∈ E|u, v ∈ V});
    for v ∈ V_k do
        c_v ← eigenvector_centrality(G_k);
        c̄.append(c_v);
    u ← sort(c̄).pop();
    place ← place ∪ {u}
return place
```

bounded. Indeed, the `FluidC` algorithm presents a linear cost equal to $O(E)$ [20]. In the second part of the algorithm, to compute the eigenvector centrality, it is sufficient to solve a linear system of equation of the size of subgraph G_k. Finally, the `sort` operation has a cost of $O(|V| \log(|V|))$ in the worst case. Hence, assuming a constant number of iterations required to reach a local maximum, Algorithm 1 presents a polynomially bounded time complexity. This confirms the scalability of the proposed algorithm.

V. Evaluation

A. Simulator

To evaluate the `FluidC` algorithm, we developed a simulator in Python 3 and NetworkX library that models an orchestrated container infrastructure and simulated the message exchange between nodes and registries. We also implemented the image distribution as layers on the topology and the simultaneous download and cache size parameters of each node.

We used a fair sharing schedule policy based on max-min fairness (MMF) to simulate the network behavior. This flow control algorithm presents an acceptable degree of approximation with real networks [22], based on the following properties: i) flows have the same priority over the available bandwidth, ii) flows get an equal share of the link bandwidth, iii) links are always using the maximum bandwidth possible based on the active flows. Figure 1 presents an example of the MMF model, where four flows need to be transmitted simultaneously, generating bandwidth limitations between routers R3, R4, and R5.

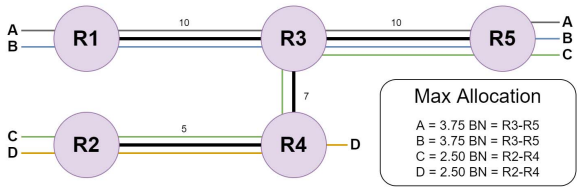


Figure 1: Max-min Fairness

Given a set of network links with respective bandwidths, and a set of paths, it is possible to obtain the available bandwidth for each transmission in a given time using a progressive filling algorithm which respects the MMF model. In our implementation, we use a solution close to that presented in [22]. The algorithm initializes the bandwidth available to each flow with 0. Then, it proceeds to calculate the MMF to all interfaces with active flows and updates the bandwidth equally to each transmission until one link becomes saturated or the total amount of data to a given transmission is satisfied. The saturated links serve as a bottleneck for all transfers using them, and the transmissions that do not need all the bandwidth available transfer the free space to the biggest ones. The algorithm executes until all links are saturated, or all flows are satisfied.

We use events to control the simulation and each time a new event occurs (start or finish of a given flow), the MMF needs to be recalculated in the network. The simulation runs until there is no more scheduled flow to be added to the topology.

B. Simulation Scenarios

To perform the simulations, we used two topologies. The first one is a random topology with 100 nodes using the Erdös - Rényi model [23]. The topology is fully connected, and each node has a downlink and uplink with the same bandwidth (1 Gbps). This topology is also highly connected with 242 links, and each registry added in the topology has bandwidth available 10 times bigger than the worker nodes (10 Gbps). This guarantees that registries do not represent a bottleneck for the simulation. We also validated the placement algorithm with the Italian education and research network (GARR) topology. The GARR topology is composed by 70 operational zones and 112 links and covers all the Italian territory. The topology is presented in Figure 2. We used the GARR topology as an isolated network, and we assumed that each operational zone is a worker node available to deploy a new application. Each operational zone can also be used to deploy a registry using the `FluidC` algorithm. We also set the bandwidth available

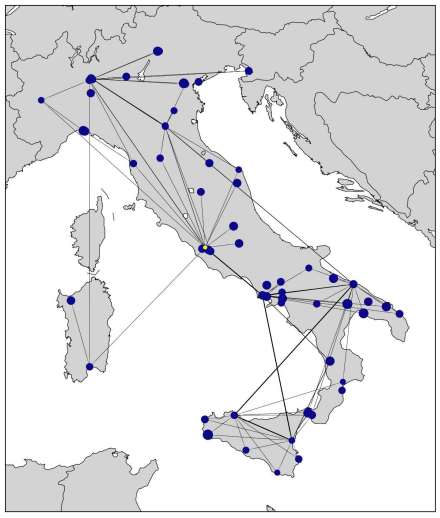


Figure 2: GARR Network Topology

to the registries node as 10 Gbps. Besides, we limited the total capacity used by the communication between the nodes to 10% of the total bandwidth described in [24], and even with that limitation, the average size to each link is close to 2Gbps.

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

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

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

Table II: Simulation parameters

C. Application Scheduler

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

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

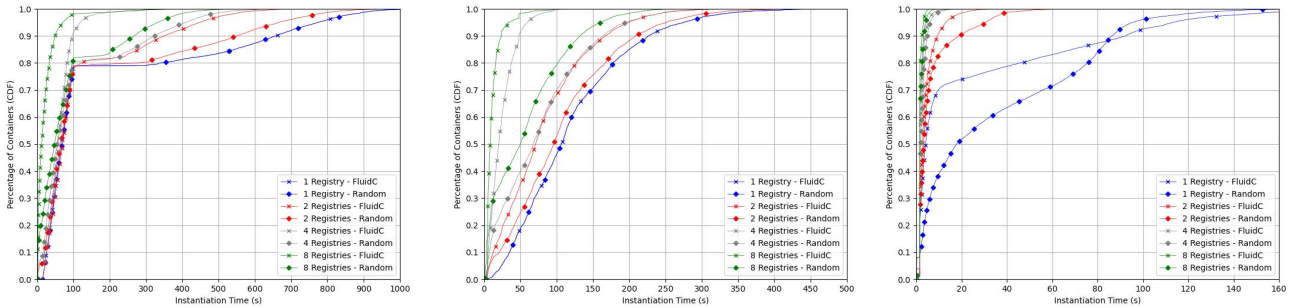
The placement algorithms selected to choose each container's destination node are: Random and the PESS Scheduler presented in [25]. The Random scheduler tries to show a non-bias distribution on the topology with a fair amount of containers between all worker nodes. With the PESS Scheduler, we want to validate the same algorithm with a more realistic scenario focused on the application placement at the Edge, improving the node utilization.

D. Experimental Results

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

1) Random Topology: In Figure 3c we have depicted the CDF of the container deployment latency according to the Random Scheduler. As expected, when the number of registries on the topology increases in both cases, random or `FluidC` algorithm, we have an improvement on the deployment latency, and the difference between them decreases as we increase the number of registries that are located on the network. However, the `FluidC` algorithm presents a deployment latency 1.59 and 1.55 times smaller in scenarios with 1 or 2 registries and 12.5% in the eight registries scenario. As the Random Topology is highly connected, in a scenario with more than 4 registries, more than 90% of the containers start with less than 10 seconds. However, it is important to note that we used all the available bandwidth to deploy the containers, which is not feasible in real infrastructures.

Results show that in this scenario, with 4 or more registries, the number of non-started containers is zero, and `FluidC` has slightly better results with two or one registries. Finally, we can see that the highly number of connections on the random



(a) GARR Topology - Random Scheduler (b) GARR Topology - PESS Scheduler (c) Random Topology - Random Scheduler

Figure 3: Instantiation time to each scenario

topology enable a good distribution of the deployment requests between the registries with the `FluidC` performing better in all number of registries.

2) GARR Topology: In Figure 3a and Figure 3b we present the results from the simulations made on the GARR Topology. The first one used the random scheduler to distribute the containers. We can see that the faster 80 percent requests are deployed in less than 100 seconds in all scenarios until four registries, in a pretty similar distribution. However, the network generates specific bottlenecks as we place the registries with both placement heuristic, showing that the 20 percent slower requests have better results consistently with the `FluidC` in contrast with the random placement.

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

The PESS Scheduler tends to schedule the application on the center of the network with the largest bandwidth connection possible. In fact, in our simulation, 67% of all containers are scheduled in only 6 worker nodes. Even with that small distribution on the network core, as shown in the Figure 3b, the `FluidC` presents consistently increase results, that besides the scenario with one registry, the deployment time was 24%, 65%, and 80% smaller using 2, 4, and 8 registries than in the random scheduler.

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

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

VI. Conclusion

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

We also consider this work as a starting point to understand the management phase cost on a shared network with many applications and how this impact can be diminished with a better allocation of layers without fundamentally changing the existing solutions on the container orchestration. In future works, we want to improve the simulation scenarios, adding more constraints and node limitations to establish the optimal number of registries for a given topology.

VII. Acknowledgements

This study was supported by the Federal Institute of Education, Science and Technology of Rio Grande do Sul (IFRS) and the PDTI Program, funded by Dell Computadores do Brasil Ltda (Law 8.248/91). The authors acknowledge the High-Performance Computing Laboratory of the Pontifical Catholic University of Rio Grande do Sul (LAD-IDEIA/PUCRS, Brazil) for providing support and technological resources, which have contributed to the development of this project and to the results reported within this research. This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nvel Superior – Brasil (CAPES) – Finance Code 001. This work has received funding from the EU H2020 R&I Programme under Grant Agreement no. 815141 (DECENTER: Decentralised technologies for orchestrated Cloud-to-Edge intelligence).

REFERENCES

[1] F. Bonomi, R. Milito, J. Zhu, and S. Addepalli, "Fog computing and its role in the internet of things," in *Proceedings of the First Edition of the MCC Workshop on Mobile Cloud Computing*, ser. MCC '12. New York, NY, USA: Association for Computing Machinery, 2012, p. 1316. [Online]. Available: https://doi.org/10.1145/2342509.2342513

[2] Y. C. Hu, M. Patel, D. Sabella, N. Sprecher, and V. Young, "Mobile edge computinga key technology towards 5g."

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

[4] K. Wang, F. Xu, Y. Ding, and L. . Xing, "Kubeedge.io," Oct 2017. [Online]. Available: https://kubeedge.io/en

[5] M. Satyanarayanan, P. Bahl, R. Caceres, and N. Davies, "The case for vm-based cloudlets in mobile computing," *IEEE Pervasive Computing*, vol. 8, no. 4, pp. 14–23, Oct 2009.

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

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

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

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

[10] T. Harter, B. Salmon, R. Liu, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Slacker: Fast distribution with lazy docker containers," in *14th USENIX Conference on File and Storage Technologies (FAST 16)*. Santa Clara, CA: USENIX Association, Feb. 2016, pp. 181–195. [Online]. Available: https://www.usenix.org/conference/fast16/technical-sessions/presentation/harter

[11] Uber, "Kraken - p2p-powered docker registry." [Online]. Available: https://github.com/uber/kraken

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

[13] A. Ahmed and G. Pierre, "Docker-pi: Docker container deployment in fog computing infrastructures," *International Journal of Cloud Computing*, vol. 9, no. 1, p. 6, 2020. [Online]. Available: https://doi.org/10.1504/ijcc.2020.105885

[14] M. Littley, A. Anwar, H. Fayyaz, Z. Fayyaz, V. Tarasov, L. Rupprecht, D. Skourtis, M. Mohamed, H. Ludwig, Y. Cheng, and A. R. Butt, "Bolt: Towards a scalable docker registry via hyperconvergence," in *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*. IEEE, Jul. 2019. [Online]. Available: https://doi.org/10.1109/cloud.2019.00065

[15] C. Zheng, L. Rupprecht, V. Tarasov, D. Thain, M. Mohamed, D. Skourtis, A. S. Warke, and D. Hildebrand, "Wharf," in *Proceedings of the ACM Symposium on Cloud Computing*. ACM, Oct. 2018. [Online]. Available: https://doi.org/10.1145/3267809.3267836

[16] F. Rossi, V. Cardellini, F. Lo Presti, and M. Nardelli, "Geo-distributed efficient deployment of containers with kubernetes," *Computer Communications*, vol. 159, pp. 161–174, 2020. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0140366419317931

[17] W.-L. Hsu and G. L. Nemhauser, "Easy and hard bottleneck location problems," *Discrete Applied Mathematics*, vol. 1, no. 3, pp. 209–215, 1979.

[18] K. Andreev and H. Racke, "Balanced graph partitioning," *Theory of Computing Systems*, vol. 39, no. 6, pp. 929–939, 2006.

[19] S. Russell and P. Norvig, "Artificial intelligence: a modern approach," 2002.

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

[21] V. Latora, V. Nicosia, and G. Russo, *Complex networks: principles, methods and applications*. Cambridge University Press, 2017.

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

[23] P. ERDOS, "On the evolution of random graphs," *Bulletin of the Institute of International Statistics*, vol. 38, pp. 343–347, 1961. [Online]. Available: https://ci.nii.ac.jp/naid/10025454140/en/

[24] C. GARR, "Consortium garr." [Online]. Available: https://www.garr.it/it/infrastrutture/rete-nazionale/infrastruttura-di-rete-nazionale

[25] R. Doriguzzi-Corin, S. Scott-Hayward, D. Siracusa, M. Savi, and E. Salvadori, "Dynamic and application-aware provisioning of chained virtual security network functions," *IEEE Transactions on Network and Service Management*, vol. 17, no. 1, pp. 294–307, 2020.