Enforcing Deployment Latency SLA in Edge Infrastructures through Multi-objective Genetic Scheduler

Luis Augusto Dias Knob
luis.knob@sertao.ifrs.edu.br
Federal Institute of Rio Grande do Sul - Campus Sertao
Sertão, RS, Brazil

Carlos Henrique Kayser
carlos.kayser@edu.pucrs.br
Pontifical Catholic University of Rio Grande do Sul
Porto Alegre, RS, Brazil

Paulo Silas Severo de Souza
paulo.severo@edu.pucrs.br
Pontifical Catholic University of Rio Grande do Sul
Porto Alegre, RS, Brazil

Tiago Ferreto
tiago.ferreto@pucrs.br
Pontifical Catholic University of Rio Grande do Sul
Porto Alegre, RS, Brazil

ABSTRACT

Edge Computing emerged as a solution to new applications, like augmented reality, natural language processing, and data aggregation that relies on requirements that the Cloud does not entirely fulfill. Given that necessity, the application deployment in Edge scenarios usually uses container-based virtualization. When deployed in a resource-constrained infrastructure, the deployment latency to instantiate a container can increase due to bandwidth limitation or bottlenecks, which can significantly impact scenarios where the edge applications have a short life period, high mobility, or interdependence between different microservices. To attack this problem, we propose a novel container scheduler based on a multi-objective genetic algorithm. This scheduler has the main objective of ensuring the Service Level Agreement set on each application that defines when the application is expected to be effectively active in the infrastructure. We also validated our proposal using simulation and evaluate it against two scheduler algorithms, showing a decrease in the number of applications that do not fulfill the SLA and the average time over the SLA to not fulfilled applications.

CCS CONCEPTS

• **Networks** → **Network management**; **Cloud computing**.

KEYWORDS

container management, container orchestration, edge computing, application deployment

ACM Reference Format:
Luis Augusto Dias Knob, Carlos Henrique Kayser, Paulo Silas Severo de Souza, and Tiago Ferreto. 2021. Enforcing Deployment Latency SLA in Edge Infrastructures through Multi-objective Genetic Scheduler. In *2021 IEEE/ACM 14th International Conference on Utility and Cloud Computing (UCC'21), December 6–9, 2021, Leicester, United Kingdom.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3468737.3494100

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

UCC'21, December 6–9, 2021, Leicester, United Kingdom

© 2021 Association for Computing Machinery.

ACM ISBN 978-1-4503-8564-0/21/12…$15.00
https://doi.org/10.1145/3468737.3494100

1 INTRODUCTION

Cloud computing has revolutionized the deployment of applications. However, new solutions using technologies, like augmented reality, natural language processing, and data aggregation, rely on requirements that the Cloud does not entirely fulfill, *e.g.* low latency and privacy-preserved data transfer. Multi-Access Edge Computing and Fog Computing have emerged to attend these requisites that are not available on current solutions. These paradigms, generalized as Edge Computing, aim to bring computational resources closer to its end-users through low latency, positioning awareness, and geo-distributed processing power infrastructure.

The application deployment in Edge computing scenario relies on fast, distributed, and heterogeneous solutions. Therefore, several works [16, 19] advocate the use of container-based virtualization techniques. The main advantages of using containerization are faster boot time, a smaller footprint, and scalability. However, during the deployment phase, the application images must be transferred from a central registry to one or more servers at the edge, which can result in a long provisioning time depending on the server and its bandwidth [2]. This total time to deploy a container is also called Deployment Latency [8].

This latency can significantly impact scenarios where the edge applications have a short life period, high mobility, or interdependence between different microservices. We understand that, in these cases, the deployment is usually followed by a Service Level Agreement (SLA) that defines the moment a given container needs to be working on the topology. Nevertheless, Kubernetes, one of the most popular container orchestration platforms, does not consider the time required to instantiate a new container during the scaling process. That happens because its default scheduler only considers CPU, memory, and storage resources, which can lead to a long provisioning time, the node's bandwidth overhead, or network bottlenecks.

In this paper we propose a novel container scheduler based on a multi-objective genetic algorithm. This scheduler, called DLSLA, has as primary objective ensuring the Service Level Agreement set on each application that defines the time when the application

is expected to be effectively active in the infrastructure. We also validated our proposal using simulation and evaluate it against two scheduler algorithms, the Kube-scheduler and the Infrastructure Aware [13] showing a decrease in the number of applications that do not fulfill the SLA and the average time over the SLA to not fulfilled applications.

The remainder of the paper is organized as follows. In Section 2, related work is presented. In Section 3, we describe the problem formulation following by the algorithm solution in Section 4. The simulation results are presented in Section 5. Finally, Section 6 concludes the paper.

2 RELATED WORK

Several works present scheduling strategies considering the SLA of deployments [12, 21]. In [12], the authors propose an SLA-driven scheduling strategy for VM placement in order to maximize the revenue of edge infrastructure-as-a-service (IaaS) providers and minimizing SLA violations, fairly between the various service providers using the Lyapunov optimization. The simulation-based results present benefits compared to the First Fit algorithm.

Yao and Ansari [21] propose a Weighted Best Fit Decreasing (WBFD) algorithm to tackle a resource provisioning problem at the edge of the network, considering the possibility of resource failures happening while minimizing the system cost incurred by resources rentals without violating the SLA requirement. The resource provision problem is formulated as an Integer Linear problem (ILP). Simulation results show that the proposed heuristic algorithm performs close to the optimal solutions of ILP with lower computational complexity.

Some works propose evolutionary algorithms to improve the placement process at the edge of the network [1, 15] and cloud [9]. In [9], the authors propose the utilization of the evolutionary algorithm Non-dominated Sorting Genetic Algorithm II (NSGA-II) for multi-objective container allocation optimization in cloud computing. Some of the objectives of the proposed strategy are: a) balanced cluster utilization; b) threshold distance; c) system failure; and d) reduction of the network overheads. Compared to the Kubernetes scheduler mechanism, the proposed strategy presents improvements in relation to the objectives addressed.

In [15], the authors propose a multi-objective genetic algorithm based on Biased Random-Key Genetic Algorithm (BRKGA) and NSGA-II to enhance the service placement and load distribution in an Internet of Things (IoT) and Edge Computing environment. For this, a Mixed-Integer Linear Programming (MILP) problem optimization problem is formulated to minimize the potential occurrence of SLA violations. The efficiency of the proposed algorithm is analyzed through simulation, and the proposed algorithm achieves values close to the optimum of the MILP formulation.

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

3 PROBLEM FORMULATION

In this section, we describe the edge application provisioning problem targeted in this work. Firstly, we describe the main elements of the edge infrastructure considered in our modeling. Then, we formulate the several steps that comprehend provisioning applications in the edge nodes alongside our optimization objectives. Notations used in this paper are summarized in Table 1.

Table 1: Summary of notation used in this paper.

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

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

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

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

Although container layers may contain application-specific settings, many are generic, used in common by different applications. Accordingly, when provisioning a containerized application replica $\mathcal{R}_j^k$, an edge node $\mathcal{N}_i$ checks if $\mathcal{R}_j^k$ layers $\mathcal{L}_j^k$ have not been recently

downloaded and are accessible in its cache c_i. Consequently, only not cached layers are downloaded from the registry node β, avoiding unnecessary traffic in the network and potentially shortening applications' provisioning time. We can check whether container layer $\mathcal{L}_{j,k}^{v}$ is available in cache c_i through a $\mathcal{N} \times \mathcal{L}$ matrix ϑ, where:

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

We assume that edge nodes download container layers from the registry sequentially given their queues. Therefore, a replica $\mathcal{R}_j^k$ located at position ρ in a download queue q_i has to wait for w_j^k units of time before getting downloaded, where w_j^k represents the time needed to download all previous items in q_i. When its turn comes, downloading replica $\mathcal{R}_j^k$ takes d_j^k units of time, as denoted in Equation 1.

$$d_j^k = \sum_{v=1}^{|\mathcal{L}_j^k|} \frac{t_v^{j,k}}{b_i} \cdot (1 - \vartheta_{i,j,k,v}) \tag{1}$$

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

Our goal consists in defining the placement of application replicas and the arrangement of the edge nodes' download queues to minimize the number of SLA violations due to prolonged provisioning times. Accordingly, the objective function can be formulated as in Equation 2, where constraint 1 (Equation 3) guarantees that each replica is only provisioned once, constraint 2 (Equation 4) sets the lower bound of provisioning times, and constraint 3 (Equation 5) certifies that edge nodes are not overloaded.

$$\textbf{Minimize} \sum_{j=1}^{u} [\partial_j \; > \wp_j] \tag{2}$$

Subject To:

$$\sum_{j=1}^{u} \sum_{k=1}^{r_j} \varkappa_{i,j,k} = 1, \; \forall_i \in \{1, \; 2, \; ..., \; n\} \tag{3}$$

$$\partial_j \; \geqslant 0, \; \forall_j \in \{1, \; 2, \; ..., \; u\} \tag{4}$$

$$\sum_{j=1}^{u} \sum_{k=1}^{r_j} h_j^k \cdot \varkappa_{i,j,k} \leqslant z_i, \; \forall_i \in \{1, \; 2, \; ..., \; n\} \tag{5}$$

4 DEPLOYMENT LATENCY SLA ENFORCEMENT SCHEDULER

Defining placement schemes for application replicas and finding proper arrangements for edge nodes' download queues, which is a variant of the Application Scheduling Problem [20], is an NP-hard optimization problem. For that reason, approximation algorithms represent viable alternatives to find acceptable solutions within a bounded time. As finding the optimal solution is infeasible given the problem complexity, we calculate a Pareto Front to find a set of non-dominated solutions (i.e., none of the solutions found beat them in all objectives) [7]. Figure 1 presents a visual representation of a Pareto Front in a sample bi-objective optimization.

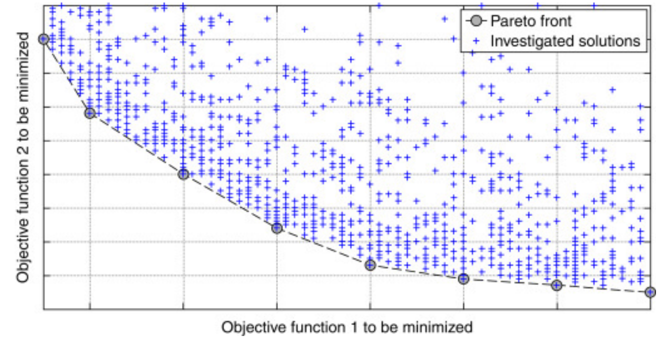


Figure 1: Visual representation of a Pareto Front [3].

There are several single and multi-objective algorithms that can find pareto-optimal solutions [5, 6, 10]. We employ the Non-dominated Sorting Genetic Algorithm II (NSGA-II) [6], as it reaches superior results compared to other meta-heuristics [22]. In the remaining of this section, we present a novel scheduling algorithm called Deployment Latency SLA Enforcement Scheduler (DLSLA), which leverages NSGA-II functionality to minimize the number of SLA violations due to overextended provisioning time of multi-replica applications in edge computing environments.

4.1 Population Initialization

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

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

4.2 Download Queue Implementation

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

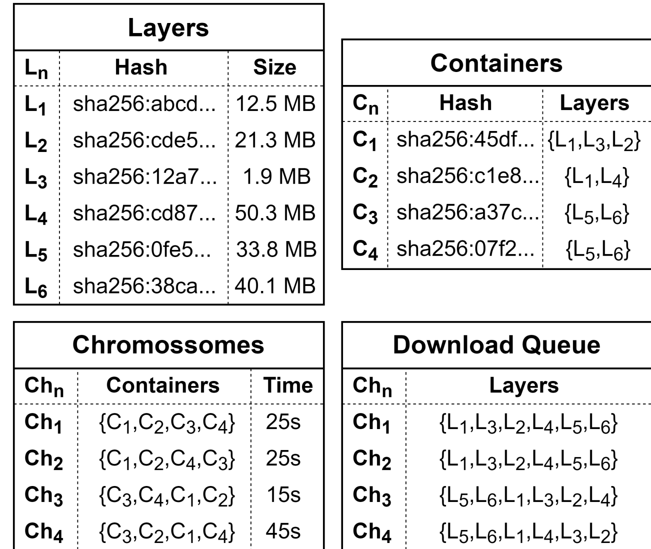
So, in an overcrowded cluster, with many applications and lifecycle operations, this can generate situations where large applications (like Java-based or databases) will take several seconds or even minutes to be deployed, halting the application instantiation in a set of nodes. In cloud computing, with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

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

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

4.3 Chromosome Representation

We represent each possible solution (so-called chromosome) as an array, called *containers*, which can be correlated with the node's deployment queue. Each value on the queue, called *gene*, is set by the unique identifier representing the containers that need to be downloaded and deployed. No container can appear more than once on the queue, and all containers need to be present on each chromosome. If several applications use the same container, the algorithm will put only the one with the smallest SLA value since all applications that use the same container will start together after the image download. Figure 2 presents a graphic representation from the chromosomes and the relation between the containers images, layers, download queue, and the chromosome.



Layers

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

Containers

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

Chromossomes

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

Download Queue

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

Figure 2: Chromosome Representation.

4.4 Fitness Function

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

Algorithm 1 presents the fitness function implementation, where we first calculate, to a given queue, the amount of time needed to deploy all the containers. We also validate how many SLA violations that a given chromosome will generate. Finally, we verify the number of changes between the original queue and the queue presented on the chromosome.

4.5 Genetic Operators

Selection. As DLSLA is based on NSGA-II, it uses three concepts to select the best chromosomes in the population [9]. The first is dominance, where a chromosome dominates another if the fitness values for all objectives is better than the dominated. Second, the optimal fronts that group the chromosomes non-dominated by other ones using the Pareto distribution. Finally, the crowding distance is calculated by the average distance along each objective between the chromosome in the same front. This average distance is used

Algorithm 1: Fitness function.

```
Input  : node_queue: Node download queue; bandwidth:
         Network bandwidth available on node; inital_time:
         Simulation time where the new application need to
         be instantiate; chromosome: Chromosome that
         needs to compute; original_app_queue: Current
         application queue on node
Output : Fitness score to the chromosome
1  sla_violations ← 0;
2  total_time ← {};
3  changes_on_queue ← 0;
4  for gene ∈ chromosome_genes do
5      for digest, size ∈ gene_layers do
6          if digest ∉ node_queue then
7              total_time+ = size ÷ bandwidth;
8              node_queue.append(digest);
9      if total_time > gene_sla then
10         sla_violations+ = 1;
11 for app ∈ original_app_queue do
12     temp_index ← node_queue.index(app);
13     node_queue.pop(temp_index);
14     changes_on_queue+ = temp_index;
15 fitness ← {sla_violations, total_time,
     changes_on_queue};
16 return fitness
```

to sort the chromosomes. After these three operations, we have a population with $len(population) * 2 - 1$ chromosomes. Then, the algorithm selects the $len(population)$ best chromosomes.

Crossover. After selecting the fittest chromosomes, DLSLA employs a mating process (called crossover) to evolve the population. The crossover process used by DLSLA can be seen on Algorithm 2. First, we run the algorithm $len(population) - 1$ times, selecting the chromosomes $population_x$ and $population_{x+1}$ as parents for each new child. Then, for each child gene, we randomly select a gene in the same position from one of the parents. We also store the gene not chosen, so in case of conflict or if the gene is already on the child, we pop one storage gene and complete the queue with a unique gene since no gene appears more than one time on the chromosome. This crossover function guarantees that if a gene is equal in both parents, it stays the same on the child. So, only distinct genes between parents are randomly selected for the child.

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

4.6 Scheduler Score and Ranking

After the GA runs for a given number of generations, we return the high classified chromosome as the best solution to that given node. After executing the scoring algorithm to all nodes, be it the

Algorithm 2: Crossover function.

```
Input  : population: population of chromosomes;
Output : New population with original chromosomes plus
         children
1  for x < len(population) − 1 do
2      child ← ∅;
3      father1 ← population_x;
4      father2 ← population_{x+1};
5      cache ← ∅;
6      for y < len(father1_genes) do
7          gene1, gene2 ←
             random(father1_genes[y], father2_genes[y]);
8          if gene1 ∉ child_genes then
9              child_gene[y] ← gene1;
10             if gene1 ∈ cache then
11                 cache.delete(gene1);
12             if gene2 ∉ cache ∧ gene2 ∉
                 child_genes ∧ gene1 ≠ gene2 then
13                 cache.append(gene2);
14         else if gene2 ∉ child_genes then
15             child_gene[y] ← gene2;
16             if gene2 ∈ cache then
17                 cache.delete(gene2);
18         else
19             child_gene[y] ← cache.pop();
20     if random() < mutation_rate then
21         child ← mutation(child);
22     child_fitness ← fitness(child);
23     population.append(child);
24 return population
```

simplified version or the GA, we rank all nodes by the following weights:

$weight \leftarrow 0$

$weight += 10 - sla_violations * 0.5$

$weight += min_time/time_on_chost * 10$

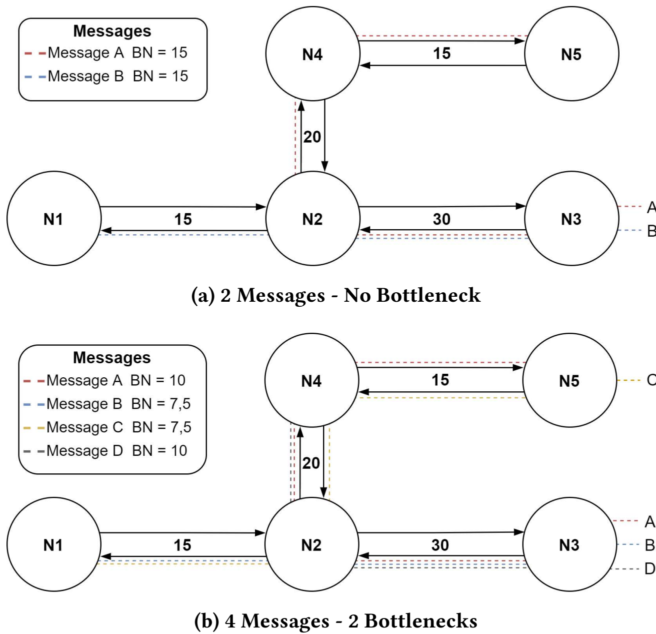
$weight += 10 - number_of_apps_on_node$

$weight += 10 - changes_on_queue * 0.5$

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

5 EVALUATION

This section presents an evaluation of the DLSLA scheduling algorithm. The algorithm is compared to the Kube-scheduler and the Infrastructure-Aware Scheduler [13] in a simulated scenario based on the Brazilian research network topology using Docker Hub images. We choose these two algorithms as a baseline because the first is the default scheduler enabled on Kubernetes. The second implements network availability as a priority, decreasing the



(a) 2 Messages - No Bottleneck

(b) 4 Messages - 2 Bottlenecks

Figure 3: Examples of Max-min Fairness Flow Model

deployment latency without considering the SLA. The metrics used for comparison include the number of applications that do not fulfill the SLA deployment latency, scheduling distribution, among others.

5.1 Simulator

In order to evaluate the strategies in an edge computing environment, we propose a discrete event edge computing simulator written in Python and NetworkX library. The simulator models the deployment process, application image distribution, and transfer across a configurable network topology. With the simulator, it is possible to implement different scheduling strategies (e.g., kube-scheduler, random scheduler, best-fit scheduler) and evaluate its performance in terms of metrics. Some are application image cache hit and cache miss, workload distribution across the network, node utilization, application provisioning time, and SLA violations.

About the Kubernetes scheduler, we simplify the implementation presented on the official source code [14]. Our implementation also enables fine-grained control of the simulation scenario, allowing to configure the weights of the scheduler predicates and priorities and then enabling an understanding of how these policies and their weights affect scheduling decisions.

5.1.1 Max-min Fairness. The network capacity and variability are based on a fair sharing schedule policy based on the max-min fairness (MMF) algorithm, more precisely using the algorithm proposed by [4], which focuses on sharing an infrastructure based on an equal distribution between flows and maximum bandwidth usage to each link. This algorithm is similar to the TCP congestion control algorithm fairness, that is, the MMF presents a reasonable approximation with the normal network behavior.

Initially, the algorithm initializes the bandwidth available to each flow as zero. Then, the MMF is calculated for all active interfaces and updates the bandwidth equally to each transmission until one network link becomes saturated or the total amount of data to a given communication is satisfied. The over-utilized links cause a bottleneck for all transmissions 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. With a set of network links and its bandwidth and a set of paths, it is possible to calculate the available bandwidth for each transmission in a given time using a progressive filling algorithm that respects the MMF model.

The major features of the MMF are a) flows (i.e., network packets, network messages) have the same priority over the available bandwidth; b) the network link bandwidth is equally shared between the flows, and c) flows are always using the maximum bandwidth possible based on the active links. Figures 3a and 3b present two examples of the MMF model behavior. The first one presents two flows in execution, one from node N3 to N1 (message A) and the other one from node N3 to N5 (message B). In this example, no bottlenecks occur on the network; thus, each flow can use the total bandwidth for the small link in its path.

The latter scenario presents four flows in execution, one from the node N3 to N5 (message A), one from the node N3 to N1 (message B), one from the node N5 to N1 (message C), and one from the node N3 to N4 (message D). However, two edge links (N2-N1 and N2-N4) provoke a bottleneck on the network, limiting the bandwidth usage by edges N4-N5 and N3-N2.

5.2 Simulation Scenario

To simulate an edge computing topology, we used the Brazilian Research Network, called Ipê. This topology interconnects all Brazilian universities and research institutes through 28 Points of Presence (PoPs) distributed over the country (Figure 4). The topology also connects to several international research networks, such as Clara (Latin America), Internet2 (United States), and Géant (Europe). In the experiments, the actual bandwidth and latency for each link were used, as described in [18].

To compare our solution with the other two schedulers, we deployed a set of container nodes on the Ipê topology. Each PoP included a large node named Worker Node and five small nodes named Edge Nodes. The main difference between the nodes is the bandwidth available to each one. For instance, the worker node has 100 Mbps, and the smaller nodes have between 10 and 60 Mbps of bandwidth (distributed uniformly). For simplicity, the simulation only considers the network utilization for container provisioning, from the registry to the worker or edge nodes. Other communications that may occur in a real network are ignored.

The registry, where all nodes request the images, is placed on PoP-São Paulo. This PoP is one of the most connected in the topology and is the primary connection to cloud providers [17]. We also set the bandwidth of the Registry Node to 10Gbps to avoid it to represent a bottleneck on the simulation.

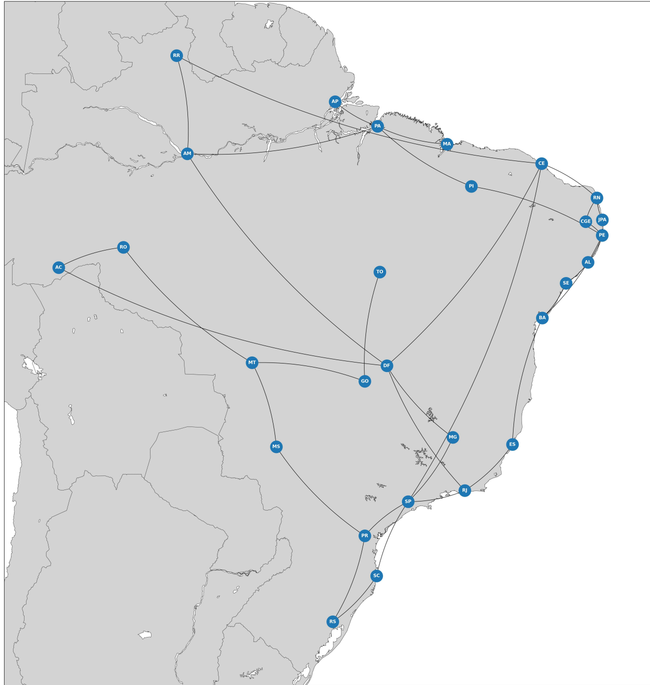


Figure 4: Ipê Topology.

5.3 Workload

The workload used in the simulation is based on the Docker Hub [11] twenty four most downloaded images, excluding base images. The images have a total size of 3436.45 MB (an average of 143.19 MB per image). However, since several images share layers, the maximum amount a given node needs to download to have all applications is 2152.78 MB (37% of similarity between images). We create a random number of applications between 5 and 25 (distributed uniformly) for each image. And for each application, we deploy a random number of replicas between 2 and 5 (distributed uniformly). Finally, each application has a random scheduling time between 0 and 1000 seconds (distributed uniformly), which defines the exact moment the application needs to be scheduled in the topology.

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

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

It is important to note that this SLA is not hard defined, so the application is always deployed, even if it is not fulfilled. After preparing the scenarios, we run the simulation 30 times for each algorithm (DLSLA, Infrastructure-Aware, and Kube-scheduler). All results present next use the arithmetic average between the runs. Table 2 presents the parameters used in the experiments.

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

Table 2: Simulation parameters.

5.4 Results

With the simulation, we want to evaluate three main variables: the number of applications that do not fulfill the SLA; on this applications' set, we want to know how much was the average time over the SLA; and finally, we want to understand how the applications' scheduling distribution was an impact between the worker and edge nodes. Since a complete centralization on the worker nodes, probably will decrease the time needed to deploy the application, but will decrease the total utilization from the topology. We also summarize the simulation results in Table 3, and present additional information like average time to all applications and number of replicas that do not fulfill the SLA.

We understand that one application does not achieve the SLA if any replica that composes this application does not start until the expected time defined by the SLA. We also want to clarify that, as the SLA is not hard ensured, the download queue created on each node with more than one container deployed simultaneously will cumulatively impact all the new applications' schedule. With that in mind, in Figure 5 we present the average number of applications that do not fulfill the SLA on each scenario.

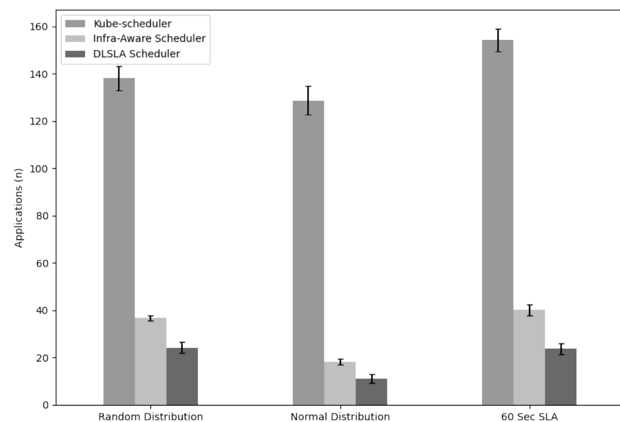


Figure 5: Number of applications that do not fulfill the SLA.

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

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

Table 3: Additional statistics from the simulations.

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

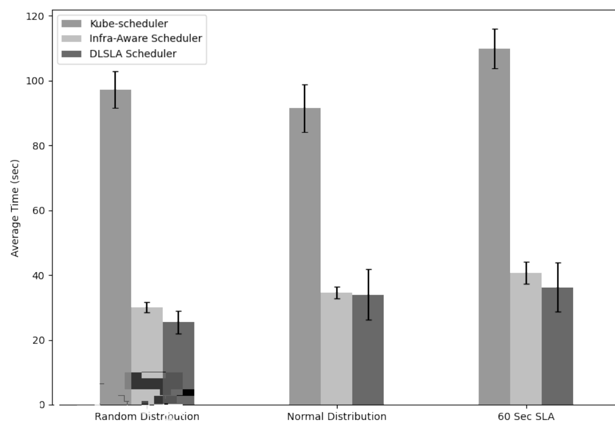


Figure 6: Average time over the SLA to not fulfilled applications.

This also reflects in our second experiment presented in Figure 6, where we plot the average time over the SLA to each application that was not deployed within the time defined by the SLA. With fewer applications ensuring the SLA, Kube-scheduler expected to have the biggest average time on each scenario, with a value close to 100 seconds in all three experiments (97.25, 91.52, 109.94). Meanwhile, the DLSLA has a better average time than the Infrastructure-Aware in all scenarios but with a bigger standard deviation over the runs. This happened for two reasons, first, the number of applications that do not fulfill the SLA is about 40% smaller using DLSLA than Infrastructure Aware. Smaller samplings will have a more significant standard deviation if the values are close to the average. Second, the infrastructure tends to be deterministic by always selecting the same nodes with more bandwidth available presenting a more consistent behavior between runs.

Finally, we want to understand the distribution impact between the worker and edge nodes based on the number of container schedules. Figure 7 presents a violin distribution to the container scheduler per node. In a worst-fit distribution, all nodes have 7.5 containers scheduled on average. Kube-scheduler shows the closest gap to this value, both on the edge and worker nodes, with a slightly bigger average on the worker node in all scenarios. While DLSLA and Infrastructure Aware present quite distinct values to the worker and edge nodes. On average, the worker nodes were selected by the Infrastructure-Aware scheduler 13.94, 14.75, and 14.21 times on average for the Random, Normal, and 60 Sec SLA, respectively. Meanwhile, the worker nodes were chosen by the DLSLA 15.29, 15.24, and 15.20 times on average.

Although the worker nodes have been chosen 2.25, 2.71, 2.72 times with the Infrastructure Aware and 2.57, 2.85, 2.99 times with the DLSLA more than the edge nodes, these nodes still allocated 68.95%, 64.79% and 64.72% of the applications with the Infrastructure-Aware scheduler and 65.98%, 63.68% and 62.51% with

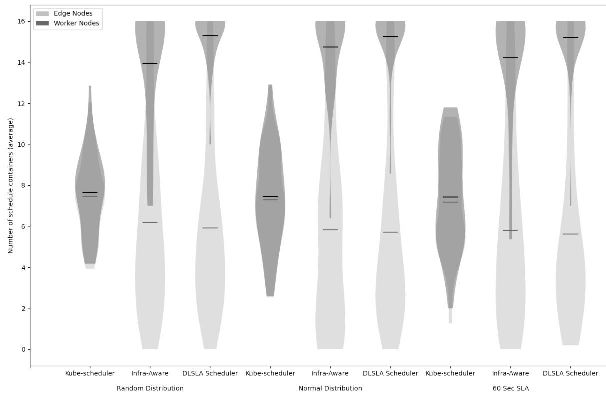


Figure 7: Average distribution between Edge and Worker Nodes.

the DLSLA. That happened because there were five times more edge nodes in the infrastructure than worker nodes. In comparison, the edge nodes with the Kube-scheduler were chosen at 82.95%, 81.05%, and 79.79% of the time. Lastly, it is possible to visualize in the distribution that in all scenarios, the Infrastructure-Aware concentrate more on a set of nodes than the DLSLA, having, for example, more edge nodes with zero container schedule (5.2, 15.8, 17.7 versus 4.2, 3.0, 1.2 on average). We summarize the experiments in Table 3, presenting more elaborated statistics for each one of the nine running sets.

6 CONCLUSION

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

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

ACKNOWLEDGMENTS

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 Nível Superior – Brasil (CAPES) – Finance Code 001.

REFERENCES

[1] Mahdi Abbasi, Ehsan Mohammadi Pasand, and Mohammad R Khosravi. 2020. Workload allocation in iot-fog-cloud architecture using a multi-objective genetic algorithm. *Journal of Grid Computing* (2020), 1–14.

[2] A. Ahmed and G. Pierre. 2018. Docker Container Deployment in Fog Computing Infrastructures. In *2018 IEEE International Conference on Edge Computing (EDGE)*. 1–8. https://doi.org/10.1109/EDGE.2018.00008

[3] Fabrizio Ascione, Nicola Bianco, Claudio De Stasio, Gerardo M. Mauro, and Giuseppe P. Vanoli. 2018. 5.21 Energy Management in Hospitals. In *Comprehensive Energy Systems*, Ibrahim Dincer (Ed.). Elsevier, Oxford, 827–854. https://doi.org/10.1016/B978-0-12-809597-3.00541-1

[4] Dimitri P Bertsekas, Robert G Gallager, and Pierre Humblet. 1992. *Data networks*. Vol. 2. Prentice-Hall International New Jersey.

[5] Kalyanmoy Deb and Himanshu Jain. 2013. An evolutionary many-objective optimization algorithm using reference-point-based nondominated sorting approach, part I: solving problems with box constraints. *IEEE transactions on evolutionary computation* 18, 4 (2013), 577–601.

[6] Kalyanmoy Deb, Amrit Pratap, Sameer Agarwal, and TAMT Meyarivan. 2002. A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE transactions on evolutionary computation* 6, 2 (2002), 182–197.

[7] Hamid Mohammadi Fard, Radu Prodan, and Thomas Fahringer. 2014. Multi-objective list scheduling of workflow applications in distributed computing infrastructures. *J. Parallel and Distrib. Comput.* 74, 3 (2014), 2152–2165.

[8] Silvery Fu, Radhika Mittal, Lei Zhang, and Sylvia Ratnasamy. 2020. Fast and efficient container startup at the edge via dependency scheduling. In *3rd USENIX Workshop on Hot Topics in Edge Computing (HotEdge 20)*.

[9] Carlos Guerrero, Isaac Lera, and Carlos Juiz. 2018. Genetic algorithm for multi-objective optimization of container allocation in cloud architecture. *Journal of Grid Computing* 16, 1 (2018), 113–135.

[10] Jiang Hao, Zheng Jin-hua, et al. 2006. Multi-Objective Particle Swarm Optimization Algorithm Based on Enhanced ε-Dominance. In *2006 IEEE International Conference on Engineering of Intelligent Systems*. IEEE, 1–5.

[11] Docker Inc. [n. d.]. Docker Hub. https://hub.docker.com/

[12] Kostas Katsalis, Thanasis G Papaioannou, Navid Nikaein, and Leandros Tassiulas. 2016. SLA-driven VM scheduling in mobile edge computing. In *2016 IEEE 9th International Conference on Cloud Computing (CLOUD)*. IEEE, 750–757.

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

[14] Kubernetes. 2021. Kube-scheduler Component Configs. https://github.com/kubernetes/kube-scheduler

[15] Adyson M Maia, Yacine Ghamri-Doudane, Dario Vieira, and Miguel Franklin de Castro. 2021. An improved multi-objective genetic algorithm with heuristic initialization for service placement and load distribution in edge computing. *Computer Networks* 194 (2021), 108146.

[16] Omogbai Oleghe. 2021. Container Placement and Migration in Edge Computing: Concept and Scheduling Models. *IEEE Access* 9 (2021), 68028–68043. https://doi.org/10.1109/ACCESS.2021.3077550

[17] RNP. [n. d.]. IX.br. https://ix.br/particip/sp

[18] RNP. [n. d.]. Rede Ipê. https://www.rnp.br/sistema-rnp/rede-ipe

[19] Mahadev Satyanarayanan. 2017. The Emergence of Edge Computing. *Computer* 50, 1 (2017), 30–39. https://doi.org/10.1109/MC.2017.9

[20] H. Topcuoglu, S. Hariri, and Min-You Wu. 2002. Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE Transactions on Parallel and Distributed Systems* 13, 3 (2002), 260–274. https://doi.org/10.1109/71.993206

[21] Jingjing Yao and Nirwan Ansari. 2018. Reliability-aware fog resource provisioning for deadline-driven IoT services. In *2018 IEEE Global Communications Conference (GLOBECOM)*. IEEE, 1–6.

[22] Cristian Zambrano-Vega, Antonio J Nebro, José García-Nieto, and José F Aldana-Montes. 2017. Comparing multi-objective metaheuristics for solving a three-objective formulation of multiple sequence alignment. *Progress in Artificial Intelligence* 6, 3 (2017), 195–210.