Location-Aware Maintenance Strategies for Edge Computing Infrastructures

Paulo S. Souza, Tiago C. Ferreto, *Member, IEEE*, Fábio D. Rossi, and Rodrigo N. Calheiros, *Senior Member, IEEE*

***Abstract*—Efficient server maintenance and update is essential to prevent performance and security issues in edge computing environments. Despite many initiatives in maintenance planning, state-of-the-art approaches concentrate on carrying out updates in cloud data centers, ignoring aspects of the problem that are specific to the edge computing paradigm, such as user-location awareness. In this letter, we present two maintenance strategies, called Lamp and Laxus, that consider users' locations when performing migration decisions to avoid delay bottlenecks during edge servers maintenance. Results show that the proposed strategies can reduce maintenance time by 44.27% compared to existing strategies while effectively avoiding delay bottlenecks.**

***Index Terms*—Edge computing, update, maintenance.**

I. Introduction

EDGE COMPUTING infrastructure operators have the responsibility of keeping the edge infrastructure up and running, which is not trivial as resources are scarce, and the network infrastructure is less robust than its cloud counterpart. Worse still, attacks targeting the edge are becoming increasingly frequent [1]. For instance, the Mirai virus [2] orchestrated a Distributed Denial of Service attack against edge servers that led to downtime in over 178000 domains.

During maintenance in cloud data centers, operators can evacuate servers by relocating hosted applications to other servers typically by observing if the demand of applications could be met, regardless of the new server location within the data center [3]. However, edge servers may have heterogeneous hardware configurations, meaning that some hosts in the edge infrastructure may not deliver analogous performance for applications. In addition, applications executed on edge environments usually have tight locality and delay constraints, narrowing the candidate hosts that could accommodate them while delivering acceptable response times.

There has been considerable prior work regarding maintenance in cloud environments [4] [3] [5]. While some strategies could be adapted to edge computing, they overlook users' locations when relocating applications. Although this characteristic does not significantly impact cloud data centers, it can generate a significant increase in application delay in edge environments. Therefore, migration techniques that consider users' location [6] can be adapted to maintenance scenarios, ensuring that the impact of maintenance on applications' performance remains as low as possible.

Manuscript received January 20, 2022; accepted February 6, 2022. Date of publication February 8, 2022; date of current version April 11, 2022. This work was supported by the PDTI Program, funded by Dell Computadores do Brasil Ltda (Law 8.248 / 91). The associate editor coordinating the review of this letter and approving it for publication was Y. Liu. *(Corresponding author: Paulo S. Souza.)*

Paulo S. Souza and Tiago C. Ferreto are with the School of Technology, Pontifical Catholic University of Rio Grande do Sul, Porto Alegre 90619-900, Brazil (e-mail: paulo.severo@edu.pucrs.br; tiago.ferreto@pucrs.br).

Fábio D. Rossi is with the Federal Institute Farroupilha, Alegrete 97555-000, Brazil (e-mail: fabio.rossi@iffarroupilha.edu.br).

Rodrigo N. Calheiros is with the School of Computer, Data and Mathematical Sciences, Western Sydney University, Parramatta, NSW 2000, Australia (e-mail: r.calheiros@westernsydney.edu.au).

Digital Object Identifier 10.1109/LCOMM.2022.3150243

TABLE I
Summary of Main Notations Used in This Letter

Symbol	Description
b_f	Wireless delay of base station $\mathcal{B}_f$
ξ_u	Delay of network link $\mathcal{L}_u$
$\aleph_u$	Bandwidth capacity of network link $\mathcal{L}_u$
ρ_i	Capacity of edge server $\mathcal{S}_i$
η_i	Capacity demand of edge server $\mathcal{S}_i$
μ_i	Update status of edge server $\mathcal{S}_i$
$\wp_i$	Update time of edge server $\mathcal{S}_i$
∂_i	Sanity check time of edge server $\mathcal{S}_i$
λ_j	Capacity demand of application $\mathcal{A}_j$
ω_j	Delay of application $\mathcal{A}_j$
$\hbar_j$	Delay threshold of application $\mathcal{A}_j$
$x_{i,j}$	Matrix that represents the application placement
$\Upsilon(\mathcal{A}_j, \mathcal{S}_i)$	Set of links used to migrate $\mathcal{A}_j$ to $\mathcal{S}_i$

This letter presents two maintenance strategies (Lamp and Laxus) that update edge infrastructures while avoiding Service Level Agreement (SLA) violations, which occur when the delay of applications exceeds a threshold agreed between infrastructure providers and users. To the best of our knowledge, this is the first attempt to tackle the challenge of conducting infrastructure maintenance on edge infrastructures considering users' location when performing migration decisions. Results show that the proposed strategies can update the edge infrastructure 44.27% faster than existing solutions while effectively mitigating SLA violations during maintenance.

II. System Model

This section describes the edge maintenance scenario approached in this work. First, we describe the elements of the edge infrastructure. Then, we formulate the steps that comprise the maintenance process. Table I summarizes the notations.

We represent the environment as in Aral *et al.* [7], dividing the map into several hexagonal cells. The edge infrastructure comprises a set of interconnected base stations $\mathcal{B}$ equipped with edge servers $\mathcal{S}$, positioned at each map cell. While base stations provide wireless connectivity to a set of users $\mathcal{U}$, edge servers host the user applications $\mathcal{A}$. We represent a base station as $\mathcal{B}_f \leftarrow \{b_f\}$, where b_f is the base station's wireless delay, and a network link as $\mathcal{L}_u \leftarrow \{\xi_u, \aleph_u\}$, where attributes ξ_u and $\aleph_u$ represent the link delay and bandwidth, respectively.

We model an edge server as $\mathcal{S}_i = \{\rho_i, \eta_i, \mu_i, \wp_i, \partial_i\}$. Attributes ρ_i and η_i represent the capacity and demand of server $\mathcal{S}_i$, respectively. More specifically, η_i is the sum of the demand of all applications hosted by $\mathcal{S}_i$. The update status of $\mathcal{S}_i$ is denoted by μ_i, which is set to 1 when $\mathcal{S}_i$ is updated and to 0 otherwise. Patching $\mathcal{S}_i$ takes $\wp_i$ units of time. After patching $\mathcal{S}_i$, we execute sanity checks that validate its integrity after the update, which takes ∂_i units of time.

An application has the following properties $\mathcal{A}_j = \{\lambda_j, \omega_j, \hbar_j\}$. Here, λ_j represents the application's capacity demand, which is considered when choosing which server will host it. The delay ω_j of application $\mathcal{A}_j$ is calculated as in Equation 1, considering the wireless delay of the base station used by $\mathcal{A}_j$'s user (denoted as $\mathcal{B}_f$) summed to the aggregated delay of network links used to communicate $\mathcal{A}_j$ to its user. As $\mathcal{A}_j$'s user moves around the map, a handoff process switches him from one base station to another. In such a scenario, if $\mathcal{A}_j$'s user is not connected to the same base station whose server hosts $\mathcal{A}_j$, the connection between them is made by routing $\mathcal{A}_j$'s data through a set of network links called θ. We define the set of links θ through the Dijkstra shortest path algorithm [8] using the link delays as weight. An SLA violation occurs when the application's delay ω_j exceeds its delay threshold $\hbar_j$. The application placement is given by a binary matrix x, where $x_{i,j}$ receives 1 if $\mathcal{S}_i$ hosts $\mathcal{A}_j$ and 0 otherwise.

$$\omega_j \leftarrow b_f + \sum_{v=1}^{|\theta|} \xi_v \tag{1}$$

The maintenance scenario considered in this work focuses on the update of each server $\mathcal{S}_i \in \mathcal{S}$, where the maintenance process is divided into a set of batches $\mathcal{Q} = \{\mathcal{Q}_1, \mathcal{Q}_2, \ldots, \mathcal{Q}_{|\mathcal{Q}|}\}$ as in the model by Zheng *et al.* [4]. The maintenance continues over a number of batches until $\sum_{i=1}^{|\mathcal{S}|} \mu_i = |\mathcal{S}|$, meaning that all servers have been updated. Our goal is to update servers as soon as possible while performing as few migrations as possible and avoiding SLA violations.

In our modeling, servers need to be rebooted for patches to take effect. Therefore, only servers hosting no applications can be updated as a means to avoid application downtime. At each maintenance batch $\mathcal{Q}_e \in \mathcal{Q}$, all outdated edge servers hosting no applications are updated. Then, migrations take place, relocating applications out of the remaining outdated servers so they can be updated in the next batch (this migration process is called "server draining" as the goal of migrations is "emptying" the outdated servers).[1] This process is repeated for a number of iterations (i.e., maintenance batches) until all servers are updated.

We assume that no shared storage exists in the infrastructure. Therefore, migrating an application $\mathcal{A}_j$ to an edge server $\mathcal{S}_i$ implies in transferring $\mathcal{A}_j$'s capacity demand λ_j from its current host to $\mathcal{S}_i$ through a set of links $\Upsilon(\mathcal{A}_j, \mathcal{S}_i) \subseteq \mathcal{L}$ that interconnect the edge servers' base stations. We define $\Upsilon(\mathcal{A}_j, \mathcal{S}_i)$ through the Dijkstra shortest path algorithm (link bandwidths are used as weight) [8]. In this context, the time it takes to migrate $\mathcal{A}_j$ to $\mathcal{S}_i$ is given by the ratio between $\mathcal{A}_j$'s capacity demand λ_j and the bandwidth available for the migration, as depicted in Equation 2. As the edge network infrastructure may be heterogeneous, the set of links $\Upsilon(\mathcal{A}_j, \mathcal{S}_i)$ may have different bandwidth capacities. Accordingly, the lowest available bandwidth between the links in $\Upsilon(\mathcal{A}_j, \mathcal{S}_i)$ is considered the actual bandwidth for the migration.

$$\kappa(\mathcal{A}_j, \mathcal{S}_i) = \frac{\lambda_j}{min\{\aleph_u \mid u \in \Upsilon(\mathcal{A}_j, \mathcal{S}_i)\}} \tag{2}$$

[1]Migrations are performed sequentially as in Zheng *et al.* [4].

III. Proposed Strategies

A. Lamp

At the beginning of each maintenance batch, Lamp updates all outdated servers that are not hosting applications (Alg. 1, lines 3–6). Then, if there are still outdated servers in the infrastructure, it starts to migrate applications to drain the servers that still need to be updated (Alg. 1, lines 8–20).

Algorithm 1: Lamp Maintenance Heuristic.

```
1  while There are outdated servers in $\mathcal{S}$ do
2      $F \leftarrow$ Outdated servers in $\mathcal{S}$
3      foreach $F_i \in F$ do
4          if $F_i$ hosts no application then
5              Update $F_i$
6              Remove $F_i$ from $F$
7      if $F$ is not empty then
8          Sort the elements of $F$ by Eq. 3 (asc.)
9          $T \leftarrow \{\}$
10         foreach server $F_i \in F$ do
11             $N \leftarrow$ Applications in $F_i$ sorted by demand (desc.)
12             $Y \leftarrow \mathcal{S} - \{T \cup F_i\}$
13             if checkCapacity($Y$, $N$) $= |N|$ then
14                 foreach application $N_j \in N$ do
15                     $Y \leftarrow$ Servers in $Y$ sorted by Eq. 4 (asc.)
16                     foreach server $Y_i \in Y$ do
17                         if $\rho_i - \eta_i \geq \lambda_j$ then
18                             Migrate $N_j$ to $Y_i$
19                             break
20                 $T \leftarrow T \cup \{F_i\}$
```

To select the order in which servers are drained, Lamp sorts outdated servers according to a cost function ϖ (Equation 3), which considers the normalized capacity, demand, and time required to update the outdated servers (we normalize variables that have different scales with Min-Max Normalization [9] to ensure that equations are not unbalanced). While patching larger servers means being able to accommodate a larger number of applications on updated servers early, prioritizing less occupied servers with shorter patching time ensures the infrastructure has up-to-date resources early.

$$\varpi(\mathcal{S}_i) \leftarrow norm\left(\frac{1}{\rho_i}\right) + norm\left(\eta_i\right) + norm\left(\wp_i + \partial_i\right) \tag{3}$$

Before performing any migration to drain a given server $\mathcal{S}_i$, Lamp calls a method described in Algorithm 2 to check if the other servers that are not being drained in the current batch have enough capacity to host all the applications hosted by $\mathcal{S}_i$ (Alg. 1, line 13). In this way, it avoids migrations that will not result in servers being drained in the current batch, shortening the duration of maintenance batches, which leads to servers being updated early.

After ensuring that a server $\mathcal{S}_i$ can be drained in the current maintenance batch, Lamp uses a cost function ϕ (Equation 4) to sort the servers that can accommodate each of the applications hosted by $\mathcal{S}_i$ according to their update status, occupation, and delay to the user that accesses the application being migrated, represented by $\Re$ (Alg. 1, line 15). In this way, Lamp tries to guarantee that applications are

Algorithm 2: Lamp's Server Capacity Checking Method.

```
Function checkCapacity($Y$, $N$):
    $Y' \leftarrow$ List of servers in $Y$
    $N' \leftarrow$ List of applications in $N$
    $\varkappa \leftarrow 0$
    foreach $N'_j \in N'$ do
        foreach $Y'_i \in Y'$ do
            if $\rho_i - \eta_i \geq \lambda_j$ then
                Host application $N'_j$ on edge server $Y'_i$
                $\varkappa \leftarrow \varkappa + 1$
                break
    return $\varkappa$
```

placed on up-to-date servers close enough to users to avoid SLA violations. Finally, Lamp goes through the ordered list of candidate servers, migrating applications to the first server found with enough capacity to host them (Alg. 1, lines 14–19).

$$\phi(\mathcal{S}_i) \leftarrow (1 - \mu_i) + norm\,(\rho_i - \eta_i) + norm(\Re) \tag{4}$$

B. Laxus

Laxus starts each maintenance batch by updating outdated edge servers that are not hosting applications (Alg. 3, lines 3–6). After that, if the infrastructure still has outdated edge servers, Laxus performs migrations in an attempt to drain those servers (Alg. 3, lines 8–12). Laxus makes migration decisions through a function called $NSGAII$, consisting of a metaheuristic called Non-Dominated Sorting Genetic Algorithm II (NSGA-II) [10]. We use NSGA-II as it can be more effective than other algorithms when finding Pareto-optimal solutions for multi-objective problems with an $O(MN^2)$ time complexity, where M represents the number of objectives the algorithm has and N is the population size [10].

Algorithm 3: Laxus Maintenance Heuristic.

```
1  while There are outdated servers in $\mathcal{S}$ do
2      $F \leftarrow$ Outdated servers in $\mathcal{S}$
3      foreach server $F_i \in F$ do
4          if $F_i$ hosts no application then
5              Update $F_i$
6              Remove $F_i$ from $F$
7      if $F$ is not empty then
8          $\mathcal{W} \leftarrow NSGAII(\mathcal{S}, \mathcal{V})$
9          $\upsilon \leftarrow$ Placement $\in \mathcal{W}$ w/ the lowest $\varrho$ (Equation 10)
10         for $k \leftarrow 1$ to $|\upsilon|$ do
11             if edge server $\upsilon_k$ does not host application $\mathcal{V}_k$ then
12                 Migrate $\mathcal{V}_k$ to $\upsilon_k$
```

1) Genetic Encoding: [2] The NSGA-II algorithm used by Laxus creates and evolves chromosomes representing migration plans to drain outdated servers. Each chromosome $\mathcal{P}_v$ of population $\mathcal{P}$ is a vector of size $|\mathcal{V}|$, where $\mathcal{V}$ denotes the set of applications hosted by the outdated servers. Each index j of $\mathcal{P}_v$ represents one of the applications in $\mathcal{V}$, and the value of each index represents the suggested edge server to host these applications. Thus, $\mathcal{P}_{v,j} \leftarrow \mathcal{S}_i$ indicates that $\mathcal{V}_j$ should be migrated to $\mathcal{S}_i$. No migration is made if $\mathcal{S}_i$ already hosts $\mathcal{V}_j$.

[2]At the beginning of the NSGA-II's execution, $\mathcal{P}$ is generated randomly.

2) Constraints and Fitness Functions: In our modeling, a solution is only considered valid if $\sum_{i=1}^{|\mathcal{S}|} [\eta_i > \rho_i] = 0$, meaning that the demand of none of the edge servers exceeds its capacity. We use three fitness functions α, β, and γ that aim at minimizing: (i) the number of outdated servers hosting applications; (ii) the migration cost; and (iii) the number of SLA violations, respectively.

The first fitness function α (Equation 5) quantifies the effectiveness of a solution $\mathcal{P}_v$ in draining outdated servers. As the maintenance continues until all edge servers are up to date, solutions that manage to drain a larger number of outdated edge servers per batch tend to finish the maintenance early. Besides, in many maintenance scenarios, updating servers as soon as possible is essential. For instance, when servers must get security patches, draining more of them sooner implies these can be updated early, reducing the attack surface on the infrastructure [5]. Therefore, α accounts for the number of outdated edge servers hosting applications.

$$\alpha(\mathcal{P}_v) \leftarrow \sum_{j=1}^{|\mathcal{P}_v|} 1 - \mu_{\mathcal{P}_{v,j}} \tag{5}$$

The second fitness function β (Equation 6) quantifies the migration cost imposed by a solution $\mathcal{P}_v$. First, β considers the average migration time (given by the function $\tau(\mathcal{P}_v)$, in Equation 7) so that solutions that perform long migrations are penalized. In addition, β also considers the function $\zeta(\mathcal{P}_v)$ (Equation 8), which penalizes solutions that make undesired migrations. More specifically, function $\zeta(\mathcal{P}_v)$ treats as undesired those migrations that meet at least one of the following criteria: (i) the current server of the application being migrated hosts other applications that will not be relocated (meaning that, despite the migration, that server will not be drained in that maintenance batch); (ii) the application is being migrated to an outdated server.

$$\beta(\mathcal{P}_v) \leftarrow \tau(\mathcal{P}_v) \cdot max(1, \zeta(\mathcal{P}_v)) \tag{6}$$

$$\tau(\mathcal{P}_v) \leftarrow \frac{1}{|\mathcal{P}_v|} \sum_{j=1}^{|\mathcal{P}_v|} \kappa(\mathcal{V}_j, \mathcal{P}_{v,j}) \tag{7}$$

$$\begin{aligned}\zeta(\mathcal{P}_v) \leftarrow \sum_{j=1}^{|\mathcal{P}_v|} \left[x_{\mathcal{P}_{v,j},j} = 0\right] \cdot \big([\{\mathcal{S}_i \in \mathcal{S} | x_{i,j} = 1\} \subset \mathcal{P}_v] \\ + 1 - \mu_{\mathcal{P}_{v,j}} \big)\end{aligned} \tag{8}$$

The third fitness function γ (Equation 9) quantifies the number of SLA violations produced by allocation decisions made by a solution $\mathcal{P}_v$. Thus, solutions that overlook users' location and migrate applications to servers too distant from users so that the access delay exceeds the application SLAs are penalized for sacrificing the delivered quality of service.

$$\gamma(\mathcal{P}_v) \leftarrow \sum_{j=1}^{|\mathcal{P}_v|} [\omega_j > \hbar_j] \tag{9}$$

3) Non-Dominated Sorting: After assigning fitness scores to the population, individuals are arranged on different fronts based on their dominance over other individuals in the population. In addition, each individual receives a score called crowding distance [10], which corresponds to the distance of the individual to its neighboring solutions on the Pareto front. Once the population is arranged on different fronts and the crowding distance of each individual is calculated, the population for the next generation is chosen based on the fronts structure (individuals on the first fronts and with larger crowding distances are prioritized). After this sorting process, only the $|\mathcal{P}|$ best individuals are selected to be part of population $\mathcal{P}$. We define a fixed number of generations ψ as the stopping criterion for the NSGA-II algorithm. Therefore, the evolution process is repeated until the maximum number of generations ψ is reached.

4) Pareto-Optimal Selection: Instead of looking for a single optimal solution in the search space, NSGA-II returns a Pareto Set $\mathcal{W}$ comprising non-dominated solutions in the Pareto Front. Accordingly, as soon as the algorithm's stopping criterion is reached, we must choose which Pareto-Optimal solution will be employed in the maintenance process. For this, we evaluate each solution $\mathcal{W}_v \in \mathcal{W}$ according to a cost function ϱ (Equation 10), selecting the solution with the lowest ϱ (Alg. 3, line 9).

$$\varrho(\mathcal{W}_v) \leftarrow norm(\alpha(\mathcal{W}_v)) + norm(\beta(\mathcal{W}_v)) + norm(\gamma(\mathcal{W}_v)) \tag{10}$$

IV. Performance Evaluation

A. Experiments Description

1) Dataset: [3] We consider an edge computing infrastructure with 40 edge servers interconnected by a Barabási-Albert network topology [11] with links containing delays = {5, 10} and bandwidths = {5, 10}. We assume that servers have heterogeneous capacities = {200, 250}. We update the edge servers with two patches with duration = {250, 350}. Each patch type has sanity checks with duration = {300, 400}. We consider a set of 90 users distributed randomly across the environment and 90 applications with demands = {20, 40, 60} and delay SLAs = {45, 90}. The initial placement of applications is defined according to a Closest-Fit heuristic described in Algorithm 4.

Algorithm 4: Initial Application Placement Heuristic.

```
1 $\mathcal{A}' \leftarrow$ List of applications in $\mathcal{A}$ arranged randomly
2 foreach $\mathcal{A}'_j \in \mathcal{A}'$ do
3     $\mathcal{U}_r \leftarrow$ User that accesses application $\mathcal{A}'_j$
4     $\mathcal{S}' \leftarrow$ List of edge servers in $\mathcal{S}$ sorted by delay from $\mathcal{U}_r$ (asc.)
5     foreach edge server $\mathcal{S}'_i \in \mathcal{S}'$ do
6         if $\rho_i - \eta_i \geq \lambda_j$ then
7             Host application $\mathcal{A}'_j$ on edge server $\mathcal{S}'_i$
8             break
```

2) Baseline: To the best of our knowledge, we are the first to propose maintenance strategies considering the requirements of edge computing infrastructures. Thus, we compare Lamp and Laxus against two maintenance strategies designed to update servers in cloud data centers. The first baseline strategy, called Greedy Least Batch (GLB) [4], aims to reduce the number of maintenance batches needed to update a group of servers. The second baseline strategy, called Salus [5], focuses on security patch scenarios, where the main goal is to reduce the period that servers remain outdated (i.e., vulnerable to attacks) during maintenance. We chose these baseline strategies because they also model the maintenance process in batches while considering most of the performance metrics we evaluate.

[3]Network, edge server, and application parameters are assigned uniformly.

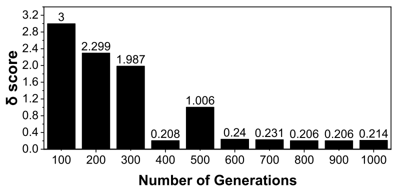

Fig. 1. Sensitivity analysis of Laxus parameters.

3) Metrics: We compare the analyzed strategies in terms of maintenance time, number of migrations, Vulnerability Surface (VS) [5] (which quantifies how long servers remain outdated during maintenance), and number of SLA violations. While the first three metrics assess specific maintenance goals, the number of SLA violations measures the impact of migrations on the quality of service delivered to end-users.

4) Testbed and Reproducibility: We conducted our experiments on a virtual machine with Ubuntu 20.04.1 LTS containing 8 CPU cores and 32GB of RAM. The virtual machine was configured with Python 3.7.10 (PyPy 7.3.5 with GCC 7.3.1 20180303). We strive to follow the reproducible research and open science principles in our investigation. The companion material hosted in a public GitHub repository[4] contains the source code, dataset, and instructions to reproduce our results.

B. Sensitivity Analysis

Among the compared strategies, Laxus is the only one containing configurable parameters. Without loss of generality, we define the population size $|\mathcal{P}| = 120$ and mutation probability to $\frac{1}{|\mathcal{V}|}$. We conduct a sensitivity analysis to determine the best settings between different number of generations ψ = {100, 200, 300, …, 1000} and crossover probabilities = {25%, 50%, 75%, 100%}. After testing each combination σ among these parameters, we choose the best configuration based on a score function $\delta(\sigma)$ (Equation 11), which considers the normalized sum of each evaluated metric.

$$\delta(\sigma) \leftarrow norm\left(\sigma^{time}\right) + norm\left(\sigma^{migr}\right) + norm\left(\sigma^{vs}\right) + norm\left(\sigma^{viol}\right) \tag{11}$$

Figure 1 presents the δ score for each number of generations ψ considering the best crossover probability among the tested configurations. The configurations that obtained the best (i.e., lowest) δ score were ψ = {800, 900} with crossover probability = 100%. We used $\psi = 800$ with crossover probability = 100% when comparing Laxus against the other strategies.

C. Comparison With Baseline Heuristics

1) Maintenance Time: Figure 2(a) shows the results in terms of maintenance time. We can observe that Laxus updates servers faster than other strategies, followed by Lamp, Salus, and GLB, respectively. The total time spent on migrations during maintenance was the factor that most influenced the maintenance time during the experiments. While Salus and

[4]Experiment assets: https://github.com/paulosevero/lamp_laxus

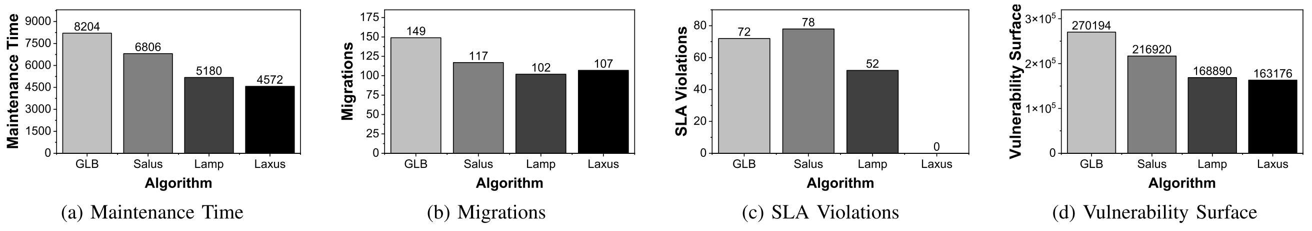

Fig. 2. Experiments results comparing the proposed heuristics (Lamp and Laxus) against strategies from the literature.

GLB spent 4556 and 5954 units of time with migrations, Laxus and Lamp completed all their migrations in only 2322 and 2930 units of time, respectively.

The reduction in the total time spent on migrations results from Laxus and Lamp's location awareness. As these strategies avoid placing applications far away from their users, they perform fewer long migrations than the other strategies. This characteristic is reflected in the average migration time of Laxus and Lamp (8.18 and 9.61 respectively) compared to GLB and Salus (14.91 and 15.29 respectively).

2) Migrations: Figure 2(b) shows the number of migrations performed by the strategies during the tests. As we can see, GLB is the one that performs the most migrations during maintenance. This behavior occurs because GLB is the only strategy that does not take into account the demand of applications when performing migrations, which opens room for potential waste of resources compared to other strategies that prioritize migrating larger applications. Although Laxus does not order applications by demand directly, its ability to evolve over a number of generations allows it to find the best packing of applications to drain servers with the lowest migration cost.

3) SLA Violations: Figure 2(c) shows the number of SLA violations that occurred while executing the compared strategies. We can observe that the baseline strategies (GLB and Salus) obtained similar results regarding the number of SLA violations (72 and 78, respectively), as they are both designed to perform maintenance on cloud data centers, performing migrations regardless of users' locations.

Unlike cloud data centers, the edge infrastructure is distributed so that migrations between edge servers distant from one another can significantly affect the application's delay. As Lamp considers the distance between servers and users when performing migrations, it reduces the number of SLA violations by 27.78% and 33.33% compared to GLB and Salus, respectively. Still, Laxus achieves the best results, completing the maintenance without any SLA violation. Such gains come from Laxus' ability to test several placement alternatives as it evolves to ultimately select the configuration that achieves the best results.

4) Vulnerability Surface: Figure 2(d) presents the results regarding Vulnerability Surface, which assesses the time required by maintenance strategies to update servers. As we can see, GLB shows the worst results by overlooking servers' exposure during maintenance. Salus, which strives to avoid unnecessary migrations and update servers as soon as possible, manages to reduce the Vulnerability Surface by 19.72% compared to GLB.

The proposed strategies minimize the Vulnerability Surface by 23.46% on average compared to Salus. This shows the importance of performing short migrations when carrying out security patches on edge computing infrastructures, allowing servers to be updated early. Laxus achieves the best results among the evaluated strategies, updating 30 out of the 40 edge servers in the infrastructure 39.65% faster than Lamp, 49.38% faster than Salus, and 61.91% faster than GLB. These gains are primarily because of the reduced migration time.

V. Conclusion

In this letter, we present two maintenance strategies called Lamp and Laxus, representing the first steps toward performing maintenance in edge computing infrastructures while observing users' location to avoid SLA violations due to excessive delay increase in applications. While Lamp uses a cost-based heuristic procedure that reduces the application delay bottlenecks during maintenance by 31% while performing 23% fewer migrations than existing strategies, Laxus adopts a genetic algorithm that updates the edge infrastructure while causing no application delay bottleneck, performing only 5% more migrations than Lamp. As future work, we intend to consider: (i) the update of components with distinct maintenance priorities and (ii) the execution of migrations in parallel to reduce the maintenance time.

Acknowledgment

The authors would like to thank 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

[1] Y. Xiao *et al.*, "Edge computing security: State of the art and challenges," *Proc. IEEE*, vol. 107, no. 8, pp. 1608–1631, Aug. 2019.

[2] M. Antonakakis *et al.*, "Understanding the Mirai BotNet," in *Proc. USENIX Secur. Symp.*, 2017, pp. 1093–1110.

[3] C. Ying, B. Li, X. Ke, and L. Guo, "Scheduling virtual machine migration during datacenter upgrades with reinforcement learning," in *Proc. Int. Conf. Heterogeneous Netw. Qual., Rel., Secur. Robustness*. Springer, 2019, pp. 102–117. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-030-38819-5_7

[4] Z. Zheng *et al.*, "Least maintenance batch scheduling in cloud data center networks," *IEEE Commun. Lett.*, vol. 18, no. 6, pp. 901–904, Jun. 2014.

[5] P. Souza and T. Ferreto, "A heuristic algorithm for minimizing server maintenance time and vulnerability surface on data centers," M.S. thesis, School Technol., Graduate Program Comput. Sci., Porto Alegre, Brazil, 2020. [Online]. Available: http://tede2.pucrs.br/tede2/handle/tede/9522

[6] Z. Liang *et al.*, "Multi-cell mobile edge computing: Joint service migration and resource allocation," *IEEE Trans. Wireless Commun.*, vol. 20, no. 9, pp. 5898–5912, Sep. 2021.

[7] A. Aral *et al.*, "ARES: Reliable and sustainable edge provisioning for wireless sensor networks," *IEEE Trans. Sustain. Comput.*, early access, Jan. 8, 2021, doi: 10.1109/TSUSC.2021.3049850.

[8] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numer. Math.*, vol. 1, no. 1, pp. 269–271, Dec. 1959.

[9] J. Han *et al.*, *Data Mining: Concepts and Techniques*. Amsterdam, The Netherlands: Elsevier, 2011.

[10] K. Deb *et al.*, "A fast and elitist multiobjective genetic algorithm: NSGA-II," *IEEE Trans. Evol. Comput.*, vol. 6, no. 2, pp. 182–197, Apr. 2002.

[11] A. L. Barabási and R. Albert, "Emergence of scaling in random networks," *Science*, vol. 286, pp. 509–512, Sep. 1999.