Published in the Proceedings of the 2019 IEEE Symposium on Computers and Communications (ISCC 2019), Barcelona, Spain, June 29-July 3 2019. doi: https://doi.org/10.1109/ISCC47284.2019.8969607

Elastic Deployment of Software Containers in Geo-Distributed Computing Environments

Fabiana Rossi, Valeria Cardellini, Francesco Lo Presti
DICII, University of Rome Tor Vergata, Italy
{f.rossi,cardellini}@ing.uniroma2.it, lopresti@info.uniroma2.it

***Abstract*—Software containers are ever more adopted to manage and execute distributed applications. Indeed, they enable to quickly scale the amount of computing resources by means of horizontal and vertical elasticity. Most of the existing works consider the deployment of containers in centralized data centers. However, to exploit the diffused presence of edge/fog computing resources, we need new solutions that deploy containers while also considering their placement on decentralized resources.**

In this paper, we present a two-step approach that manages the run-time adaptation of container-based applications deployed over geo-distributed virtual machines. In the first step, our approach exploits Reinforcement Learning (RL) solutions to control the horizontal and vertical elasticity of the containers. In the second step, it addresses the container placement by solving a suitable integer linear programming problem or using a network-aware heuristic. A wide set of simulation results shows the benefits and flexibility of the proposed approach, which can satisfy stringent application requirements expressed in terms of response time percentiles.

***Index Terms*—Containers, Elasticity, Placement, Self-adaptation, Geographically distributed resources**

I. Introduction

Software containers are changing the way cloud applications are designed, deployed, and executed [1]. Exploiting a lightweight operating system-level virtualization, containers allow to wrap up an application with its execution environment (i.e., libraries, code), and to easily run it on any machine, physical or virtual. Furthermore, containers allow to quickly change the application deployment through horizontal and vertical scaling [2]. By combining different dimensions of elasticity, the application can react more quickly to small workload variations through fine-grained vertical scaling, as well as to sudden workload peaks through horizontal scaling. Nevertheless, so far only few works have explored the benefits of combining the two elasticity dimensions for container-based applications (e.g., [2], [3]), while most of the existing solutions consider either horizontal elasticity (e.g., [4]) or vertical elasticity (e.g., [5]). Moreover, containers are usually deployed in centralized cloud data centers, which could be distant from users, data sources, and system actuators (e.g., in a IoT context). To improve application scalability and reduce response time, the trend is to use cloud resources in combination with edge/fog computing resources located at the network edges. Such a geo-distributed environment allows to decentralize the application execution, by moving the computation closer to data sources and consumers, thus reducing the expected application response time. Nonetheless, to successfully exploit this computing environment, the non-negligible network delays among computing resources running different parts of the application should properly be taken into account. Most works on container orchestration cannot efficiently exploit the features of the emerging geo-distributed environment. Indeed, they either operate at single level of abstraction (e.g., [5]), scale containers without considering their placement (e.g., [3], [6]), or do not consider the geographic distribution of cloud/fog environments (e.g., [7]).

In this paper, we propose a two-step approach to determine and adapt the deployment of container-based applications on geo-distributed virtual machines (VMs). In the first step, our solution exploits Reinforcement Learning (RL) to horizontally and vertically scale the application containers. In the second step, it allocates the application containers on VMs interconnected with non-negligible network delays. To this end, we provide a general formulation of the application placement problem that take multiple Quality of Service (QoS) attributes into account (i.e., minimizing application performance penalty, adaptation cost, and resource cost). Since the placement problem is NP-hard, we also present a network-aware heuristic to determine more quickly the application containers placement on geo-distributed resources. Differently from previous works, we allow the application's user to specify performance requirements in terms of response time percentiles (instead of average values). In such a way, our solution can identify an adaptation policy that better satisfies the user-perceived QoS.

The main contributions of this paper are as follows. First, we design RL algorithms to control the elasticity of container-based applications taking into account the percentile bound on the application response time. We use a model-based RL solution and, for sake of comparison, we also consider the classic model-free Q-learning (Sections III-IV). Then, we present an Integer Linear Programming (ILP) formulation and a network-aware heuristic, which solve the placement problem for container-based applications (Section V). Finally, we extensively evaluate the proposed approach by means of simulations and show the flexibility and efficacy of the designed deployment heuristics (Section VI).

II. Related Work

The ability of cloud computing to provide resources on demand encourages the development of elastic applications, that can be dynamically adapted in face of changing working conditions. Almost every application that is built is made of distributed interactive components. Therefore, a key challenge is to determine *how* and *where* to deploy each application component so to meet stringent QoS requirements. The *elasticity problem* exploits horizontal and vertical scaling to adapt at run-time the number of application instances and the amount of computing resources assigned to each of them. The *application placement problem* maps each application instance to a specific computing resource. Most of the existing solutions consider the two problems separately and focus either on the placement or on the elasticity problem of applications (e.g., [8], [9]). So far, only a limited number of works have studied how to jointly solve the two problems (e.g., [2], [10], [11]).

To determine or adapt at run-time the placement of application instances (using containers or VMs), the existing approaches recur to two main methodologies: mathematical programming and heuristics. Mathematical programming approaches consider the initial deployment (e.g., [12]) as well as its run-time adaptation (e.g., [8]). Mao et al. [12] present an IP formulation of the initial container placement aiming to maximize the available resources in each hosting machine. Arkian et al. [8] solve a Mixed-ILP problem to deploy application components (i.e., VMs) to fog nodes to satisfy end-to-end delay constraints. The mostly used placement heuristics range from meta-heuristics (e.g., [6]), to threshold-based heuristics (e.g., [13]), to specifically designed solutions (e.g., [14], [15]). For example, Huang et al. [14] model the mapping of IoT services to edge/fog devices as a quadratic programming problem, that, although simplified into an ILP formulation, may suffer from scalability issues. To the best of our knowledge, only Tang et al. [15] exploit RL techniques to solve the application placement problem. After defining a multi-dimensional Markov Decision Process to minimize communication delay, power consumption and migration costs, a Q-learning algorithm is proposed to control the migration of application components in a fog environment.

We can classify the most used approaches to drive the application elasticity in: custom solutions (e.g., [16]), threshold-based (e.g., [5]), and RL-based solutions. Netto et al. [16] present a state machine approach for replicating Docker containers in Kubernetes aiming to provide high availability, integrity, and strong consistency. Al-Dhuraibi et al. [5] propose ELASTICDOCKER, which employs a threshold-based policy to vertically scale CPU and memory resources assigned to each container. RL has mostly been applied to devise policies for VM allocation and provisioning (e.g., [9], [17]) and, in a limited way, to manage containers (e.g., [4]). Arabnejad et al. [17] combine Q-learning and SARSA RL algorithms with a fuzzy inference system that drives VM auto-scaling. Horovitz et al. [4] propose a threshold-based policy for horizontal container elasticity using Q-learning to adapt the thresholds. To tackle the slow convergence rate of classic model-free algorithms [18], Tesauro et al. [9] propose a hybrid RL method to dynamically allocate homogeneous servers to multiple applications. We present in [3] RL policies to control the horizontal and vertical elasticity of containers so to satisfy the average application response time. Building on this work, we here propose a fully-fledged deployment policy that manages the container placement on decentralized computing resources.

Algorithm 1 Two-step deployment adaptation policy

```
1: $R_{\max}$: requirement on the application response time
2: for each discrete time step $i$ do
3:     $S_{i-1}$: current deployment; $M_i$ = monitored performance
4:     $k_i$ = scale the application considering ($S_{i-1}$, $M_i$, $R_{\max}$)
5:     $S_i$ = place the $k_i$ containers on geo-distributed VMs
6: end for
```

Few works jointly solve the elasticity and placement problem of container-based applications. Guan et al. [10] present a LP formulation to determine the number of containers and their placement on a static pool of physical resources; nevertheless, vertical scaling operations are not considered. Nardelli et al. [2] propose an ILP formulation of the elastic provisioning of VMs for container deployment, while Guerrero et al. [11] present a genetic algorithm for container horizontal scaling and allocation on physical machines; both works take explicitly into account the network delay between machines.

Differently from these works, we propose a two-step approach to dynamically adapt the deployment of container-based applications in geo-distributed environments. The first step relies on RL to horizontally and vertically scale the application containers, thus changing the amount of assigned computing resources [3]. As second step, we propose an ILP formulation and a novel network-aware heuristic to determine the container placement. Differently from existing solutions (including [3]) that assume average QoS requirements, we consider more stringent ones and guarantee performance on the 95th percentile of the application response time, which better expresses the user-perceived QoS [19].

III. SOLUTION OVERVIEW AND SYSTEM MODEL

Running elastic applications over a geo-distributed computing infrastructure requires to determine where and how to allocate computing resources. Moreover, being applications subject to varying workloads and exposing stringent QoS requirements, their deployment should also be properly adapted at run-time. In this paper, we propose a general deployment adaptation solution (see Algorithm 1), that proceeds in two steps. First, we change the amount of computing resources allocated to the application, exploiting both vertical and horizontal elasticity of containers controlled by means of RL-based policies (line 4). Then, we determine the container placement on geo-distributed VMs (line 5). To this end, we rely on an ILP formulation as well as on a novel network-aware placement heuristic. The proposed solution is general, meaning that it can be tuned to optimize different QoS metrics. In the following, we focus on identifying deployment solutions that minimize the amount of allocated computing resources, while meeting QoS requirements on the response time percentiles.

Application Model: We consider the application as a black-box entity that carries out specific tasks and exposes stringent QoS requirements in terms of response time: its 95th percentile should not exceed a target value $R_{\max}$. To simplify management operations, the application is executed using containers. Following the Docker model, a container is an instance of a container image (structured as a series of layers), which represents an application snapshot; it embeds all the dependencies needed for the execution. The container image layers can be downloaded from an external repository, before running the application instance on a new hosting VM. We refer to the set of application containers as E. Each container $e \in E$ is characterized by: c_e, the amount of required CPU shares on the hosting VM; $T_e \in [T_{\min}, T_{\max}]$, its startup time; and I, the set of container image layers. Each image layer $i \in I$ adds specific functionality and has size l_i.

Resource Model: We consider VMs distributed over a geo-distributed infrastructure. A VM offers computing resources for the execution of containerized applications. Let V be the set of VMs. A virtual machine $v \in V$ is characterized by: C_v, the amount of available computing resources (e.g., CPU cores); DR_v, the download data rate from the container image repository; and I_v, with $I_v \subseteq I$, the set of image layers available in v. The (logical) network link (u, v) between each pair of VMs $u, v \in V$ has a network delay $d_{u,v}$.

IV. Horizontal and Vertical Elasticity

Aiming to run elastic applications, we want to dynamically adapt the amount of allocated computing resources at run-time by exploiting horizontal and vertical elasticity of containers. To drive the adaptation actions, we rely on a RL agent whose goal is to minimize an expected long-term cost. To this end, the RL agent estimates the so-called Q-function. It consists in $Q(s, a)$ terms, which represent the expected long-term cost that follows the execution of action a in state s.

We define the application state at time i as $s_i = (k_i, u_i, c_i)$, where k_i is the number of application instances (i.e., the cardinality of E), u_i is the CPU utilization, and c_i is the CPU share granted to each container. We denote by $\mathcal{S}$ the set of all the application states. We assume that $k_i \in \{1, 2, ..., K_{\max}\}$, $u_i \in \{0, \bar{u}, ..., L\bar{u}\}$, and $c_i \in \{\bar{c}, 2\bar{c}, ..., M\bar{c}\}$, where $\bar{u}$ and $\bar{c}$ are suitable quanta. For each state $s \in \mathcal{S}$, we define the set of adaptation actions $\mathcal{A}(s) = \{-r, -1, 0, 1, r\}$, where $\pm r$ denotes a vertical scaling (i.e., to add/remove r CPU shares), ± 1 denotes a horizontal scaling (i.e., to scale-out/in the number containers), and 0 is the *do nothing* decision. Observe that not all the actions are available in any application state, because of the upper and lower bounds on the number of CPU shares and instances per application.

We associate an immediate cost function $c(s, a, s')$ to each tuple (s, a, s'), which captures the cost of carrying out action a when the application state transits from s to s'. The cost function includes three different contributions: the performance penalty c_{perf}, the resource cost c_{res}, and the adaptation cost c_{adp}. The *performance penalty* c_{perf} is paid whenever the 95th percentile of the application response time exceeds the target value $R_{\max}$. The *resource cost* c_{res} is proportional to the number of application instances and assigned CPU share. The *adaptation cost* c_{adp} captures the time needed to reconfigure the application deployment $D(\cdot)$ (i.e., adaptation time). Formally, we define $c(s, a, s')$ as the weighted sum of the costs, normalized in the interval $[0, 1]$; the different weights allow us to express the relative importance of each term:

$$\begin{aligned} c(s, a, s') \quad &= \quad w_{\text{perf}} \mathbb{1}_{\{R(k+\tilde{k}, u', c+\tilde{c}) > R_{\max}\}} + \\ &+ w_{\text{res}} \frac{(k+\tilde{k})(c+\tilde{c})}{K_{\max}} + w_{\text{adp}} \frac{D}{D_{\max}} \end{aligned} \tag{1}$$

where $\mathbb{1}_{\{\cdot\}}$ is the indicator function, w_{adp}, w_{perf} and w_{res}, $w_{\text{adp}} + w_{\text{perf}} + w_{\text{res}} = 1$, are non negative weights for the different costs, and $R(k, u, c)$ is the 95th percentile of the application response time in $s = (k, u, c)$. Furthermore, we decompose action a in terms of number of containers added/removed, $\tilde{k}$, and amount of CPU share increased/decreased, $\tilde{c}$.

We consider two different RL approaches for estimating the Q-function, namely *Q-learning* and *Model-based* algorithms.

Q-learning. Q-learning is a *model-free* algorithm that requires no knowledge of the application dynamics [18]. At time i, Q-learning observes the application state s_i and selects a_i using an ϵ-greedy policy on $Q(s_i, a_i)$; the application transits in s_{i+1} and experiences an immediate cost c_i (see (1)). Then, Q-learning updates $Q(s_i, a_i)$ using a simple weighted average:

$$Q(s_i, a_i) \leftarrow (1-\alpha) Q(s_i, a_i) + \alpha \left[c_i + \gamma \min_{a' \in \mathcal{A}(s_{i+1})} Q(s_{i+1}, a') \right]$$

where $\alpha \in [0, 1]$ is the *learning rate* parameter and $\gamma \in [0, 1)$ the *discount factor*.

Model-Based RL. In the model-based approach, we exploit the known system dynamics and compute the Q-function using directly the Bellman equation:

$$Q(s, a) = \sum_{s' \in \mathcal{S}} p(s'|s, a) \left[c(s, a, s') + \gamma \min_{a' \in \mathcal{A}} Q(s', a') \right] \quad {\scriptstyle \forall s \in \mathcal{S}, \atop \forall a \in \mathcal{A}(s)} \tag{2}$$

We observe that the paid cost and the next state transition depend on known and unknown application dynamics. So, we replace the unknown transition probabilities $p(s'|s, a)$ and the unknown cost function $c(s, a, s')$, $\forall s, s' \in \mathcal{S}$ and $a \in A(s)$, by their empirical estimates. We estimate $p(s'|s, a)$ as the relative number of times the CPU utilization changes from state $j\bar{u}$ to $j'\bar{u}$ in the interval $\{1, \ldots, i\}$.

For the estimates of the immediate cost $c(s, a, s')$, we observe that it can be written as the sum of two terms, named as the known and the unknown cost. The known cost $c_k(s, a)$ depends on the current state and action; in our case, it accounts for the adaptation and resource costs. To quickly approximate the adaptation cost, we observe that $c_{\text{adp}} \neq 0$ only when either a scale-out or a scale-up operation should be performed (i.e., $a \in \{+1, +r\}$), because it can require to download container images or waiting containers boot time.

The unknown cost $c_u(s')$ captures performance penalties (1), and is needed to compute and update $Q(s, a)$. Therefore, at time i, the RL agent observes the immediate cost c_i, computes $c_{u,i}(s') = c_i - c_{k,i}(s, a)$, and updates the estimate of the unknown cost $\hat{c}_{u,i}(s')$, as follows:

$$\hat{c}_{u,i}(s') \leftarrow (1-\alpha) \hat{c}_{u,i-1}(s') + \alpha c_{u,i}(s') \tag{3}$$

Further details on the RL-based policy can be found in [3].

V. Container Placement

At time i, the RL agents determine the number of containers k_i in E and the amount of computing resources per container c_i to allocate for running the application. Then, as second step, our approach solves the problem of determining the containers placement on geo-distributed computing resources.

We model the application container placement with binary variables $x_{e,v}$, $e \in E$, $v \in V$: $x_{e,v} = 1$ if container e is deployed on VM v and $x_{e,v} = 0$ otherwise. For short, we denote by $\boldsymbol{x}$ the placement vector for containers, where $\boldsymbol{x} = \langle x_{e,v} \rangle$, $\forall e \in E$, $\forall v \in V$. Since we solve the container placement problem at each discrete time step i, we find convenient to define binary variables $x'_{e,v}$, which store the placement of $e \in E$ on $v \in V$, as determined at time step $i-1$. Leveraging on $x'_{e,v}$, we also define the binary variables $\delta_{e,v}$, which indicate whether a container $e \in E$ has a different placement with respect to the configuration in $i-1$, i.e., $\delta_{e,v} = 1$, if e was turned off or was allocated on a VM $u \neq v$, with $u, v \in V$. Observe that $x'_{e,v}$ and $\delta_{e,v}$ enable to model the run-time re-allocation of containers on VMs.

Adaptation Time. We define the adaptation time of containers $D(\cdot)$ as the time needed to deploy every container in E. This term accounts for the time needed to retrieve container images and finally start the containers. Given the placement vector $\boldsymbol{x}$, we have:

$$D(\boldsymbol{x}) = \sum_{e \in E} \sum_{v \in V} D_{e,v}(\boldsymbol{x}) \tag{4}$$

where $D_{e,v}(\boldsymbol{x})$ denotes the time needed to deploy e on v:

$$D_{e,v}(\boldsymbol{x}) = \sum_{i \in I \setminus I_v} \frac{l_i}{DR_v} x_{e,v} + T_e \delta_{e,v}. \tag{5}$$

The term $\sum_i \frac{l_i}{DR_v}$ models the time needed to download the container image layers not yet on v (i.e., not in I_v), and T_e is the time needed to start a new container e on v.

Virtual Machines Cost. The virtual machines cost $Z(\boldsymbol{x})$ counts the number of VMs used for running the application:

$$Z(\boldsymbol{x}) = \sum_{v \in V} z_v \tag{6}$$

where the binary variables z_v denote whether $v \in V$ hosts at least one container. Formally, we have $z_v = \vee_{e \in E, v \in V} x_{e,v}$.

Application Constraints. The application runs on geo-distributed VMs and exposes strict requirements on response time (i.e., its 95th percentile should be smaller than $R_{\max}$). To meet this requirement, the RL agent conveniently scales the amount of computing resources for the application at run-time. Furthermore, the placement policy explicitly models the network delay between VMs, and allocates the application only in VMs whose network delay is below the critical value $ND_{\max}$. Formally, we have that $d_{u,v} \cdot \bar{z}_{u,v} \leq ND_{max}, \forall u, v \in V$, where $\bar{z}_{u,v}$ is a binary variable representing whether u and v run the application (i.e., $\bar{z}_{u,v} = z_u \cdot z_v$).

Optimal Placement Problem Formulation. We formulate the placement problem as an ILP model that, solved at time i, determines the optimal mapping of the k_i application containers onto the geo-distributed VMs. Our problem formulation considers an objective function that minimizes the adaptation time and the VM cost. We define the objective function $F(\boldsymbol{x})$ as the sum of the normalized QoS metrics to be minimized, as follows:

$$F(\boldsymbol{x}) = \frac{D(\boldsymbol{x})}{D_{\max}} + \frac{Z(\boldsymbol{x})}{Z_{\max}} \tag{7}$$

where $D_{\max}$ and $Z_{\max}$ denote the maximum value for the overall expected adaptation time and cost, respectively. After normalization, each metric ranges between the best possible case (i.e., 0) and the worst case (i.e., 1). The Optimal Placement problem is formulated as follows:

$$\min_{\boldsymbol{x}} F(\boldsymbol{x}) \qquad \textbf{subject to:}$$

$$\begin{aligned}
&\sum_{v \in V} x_{e,v} = 1 && \forall e \in E && (8)\\
&\sum_{e \in E} c_e x_{e,v} \leq C_v && \forall v \in V && (9)\\
&d_{(u,v)} \cdot \bar{z}_{u,v} \leq ND_{\max} && \forall u, v \in V && (10)\\
&\frac{1}{\Gamma} \sum_{e \in E} x_{e,v} \leq z_v \leq \sum_{e \in E} x_{e,v} && \forall v \in V && (11)\\
&z_u + z_v - 1 \leq \bar{z}_{u,v} \leq \frac{z_u + z_v}{2} && \forall u, v \in V && (12)\\
&\delta_{e,v} \geq x_{e,v} - x'_{e,v} && \forall e \in E, \forall v \in V && (13)\\
&\delta_{e,v} \leq \frac{(1 - x'_{e,v}) + x_{e,v}}{2} && \forall e \in E, \forall v \in V && (14)\\
&\delta_{e,v} \in \{0,1\}, x_{e,v} \in \{0,1\} && \forall e \in E, \forall v \in V && (15)\\
&z_v \in \{0,1\}, \bar{z}_{u,v} \in \{0,1\} && \forall u, v \in V && (16)
\end{aligned}$$

Equations (8) and (9) allow to find a correct placement. The former requires that a container e is allocated on one and only one VM, whereas the latter ensures that a hosting VM exposes only its available resources. Equation (10) expresses the application constraint on network delay between the hosting VMs. The latter are identified by means of z_v and $\bar{z}_{v,u}$, whose definition is provided by Equations (11) and (12). Γ is a large constant, such that $\Gamma \geq K_{\max}$. Equations (13) and (14) model whether a container e has a different allocation with respect to the placement defined in $x'_{e,v}$.

Theorem. The Optimal Placement problem is NP-hard: In order to verify the NP-hardness of the Optimal Placement problem, it suffices to prove that the corresponding decision problem is NP-hard. To prove the NP-hardness of this decision problem, let us consider the special case where: k is the number of application containers and c the amount of resources requires by each container. We suppose that: c is an integer; the network has only two nodes, $V = \{u, v\}$, with capacity $C_u = C_v = (k \cdot c)/2$; there are no network delays, that is $d_{u,v} = d_{v,u} = 0$ (this allows us to ignore the variables $\bar{z}_{v,u}$, z_u and z_v); and containers can be placed on both VMs. It is easy to realize that the resulting problem is the well known Partition problem which is known to be NP-hard. Since this special case is NP-hard, the general decision problem is NP-hard as well. And since the original optimization problem is at least as hard as the decision problem, it follows that Optimal Placement problem is NP-hard as well.

Network-aware Greedy Heuristic. The ILP formulation models an NP-hard problem; so, it does not scale well as the problem instance increases in size. To overcome this limitation, we propose a *network-aware greedy heuristic*, which exploits the deployment objective function and the system model to determine the placement of the application containers. The heuristic solves a variant of the bin-packing problem. For each application container to deploy, the heuristics selects the hosting VMs from a sorted list using a greedy approach. The list is sorted in ascending order, using the objective function as distance metric: the first VMs of the list minimizes the adaptation time. Moreover, to satisfy the application requirements, the heuristic selects only the VMs having network delay from the active VMs below $ND_{\max}$.

VI. Experimental Results

We evaluate the proposed deployment adaptation solutions by means of simulations. First, we compare the model-based RL approach against the model-free approach, aiming to identify a suitable application elasticity policy. Then, we consider the placement problem and investigate the behavior of the deployment adaptation policy, when the RL policy is coupled with the placement policies proposed in Section V.

At each discrete time step i, we consider the reference application modeled as an M/D/k_i queue, because we can reasonably assume that: the application receives random and independent (M) requests, its service time is deterministic (D), and the number of servers is equal to the number of containers (k_i). Each application instance can serve $\mu = 200 \cdot c_i$ requests/s, where $c_i \in (0,1]$ is the assigned CPU share. Furthermore, each application container image uses 3 layers, each of which has size l_i uniformly defined in $[200, 800]$ MB. The container boot time T_e is selected uniformly in $[8.5, 11.5]$ s. The application requires that the 95th percentile of its response time is below $R_{\max} = 50$ ms. We consider that the application receives a number of requests that changes over time according to the workload pattern shown in Figure 1. The RL algorithms use the following parameters: $K_{\max} = 10$, discount factor $\gamma = 0.99$ and, for Q-learning, learning rate $\alpha = 0.1$ and $\epsilon = 1/i$. To discretize the application state, we use $\bar{u} = 0.1$ and $\bar{c} = 10\%$. We run the simulation on a machine with an Intel Xeon E5504 (2.00 Ghz) and 16 GB of RAM. To solve the optimal ILP formulation, we use CPLEX 12.8.

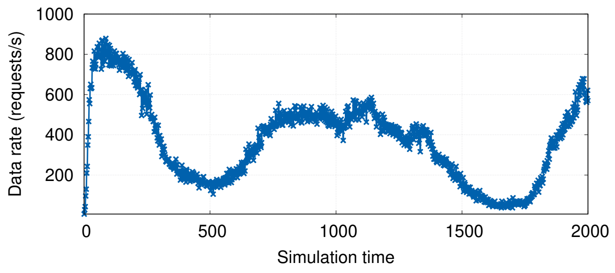


Fig. 1: Application workload used in simulation.

Application Elasticity. In the first set of experiments, we compare the model-based RL policy against Q-learning, when different cost function weights are considered (see Eq. 1). To this end, containers are allocated on VMs which are interconnected with a negligible network delay. Moreover, we use the optimal ILP formulation to compute the containers placement. Table I reports the experimental results.

When we consider the set of weights $w_{\text{perf}} = 0.90$, $w_{\text{res}} = 0.09$ and $w_{\text{adp}} = 0.01$, optimizing the application response time is more important than saving resources or avoiding reconfigurations. Q-learning frequently changes the application deployment (i.e., 78.11% of the time); as a consequence, the application response time exceeds $R_{\max}$ for 30.93% of the time (see Table I). Taking advantage of the system knowledge, the model-based solution drastically reduces to 3.15% the number of $R_{\max}$ violations. Differently from Q-learning, the model-based policy uses a higher number of medium-size containers and, in general, learns a more robust adaptation strategy, as suggested by the small number of deployment reconfigurations reported in Table I.

We now consider the case when saving resources is more important than the other objectives, i.e., $w_{\text{perf}} = 0.09$, $w_{\text{res}} = 0.90$, and $w_{\text{adp}} = 0.01$. Intuitively, the agent should learn how to improve resource utilization at the cost of a high application response time (i.e., violating $R_{\max}$). From Table I, we can see that the model-based solution successfully does it. It runs the application with 1.20 instances, on average, that can access only to 11.71% of CPU resources. It also avoids run-time adaptations. As a consequence, the application is overloaded, and the resulting median response time is in the order of 10 seconds. Instead, Q-learning continuously reconfigures the application deployment and uses on average a higher number of containers that can access to more computing resources.

As third case, we balance the importance of three deployment goals, i.e., $w_{\text{perf}} = w_{\text{res}} = w_{\text{adp}} = 0.33$. Table I shows that the RL policies try to find a trade-off between the previous settings. Q-learning has 45.53% of $R_{\max}$ violations and 72.72% average resource utilization. Also in this case, the model-based solution learns a better adaptation strategy; indeed, it reduces $R_{\max}$ violations to 22.59%, with a 66.57% of average resource utilization. To achieve such trade-off, the model-based RL strategy adapts the application deployment less than 13% of the time.

This set of experiments has shown the importance of providing system knowledge to improve the learning task. The reduced number of application reconfigurations suggests that the model-based agent learns a robust adaptation policy, which allows to guarantee the stringent QoS requirements (i.e., 95th percentile of the application response time below $R_{\max}$).

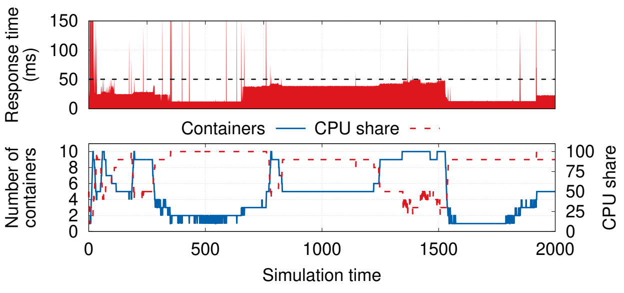
Application Elasticity and Placement. In this second set of experiments, we combine the model-based RL policy with different placement policies. We consider 20 VMs uniformly distributed across 4 data centers. Each VM has $C_v = 2$ vCPUs

TABLE I: Comparison between the RL policies to drive the application elasticity under different configurations of cost weights.

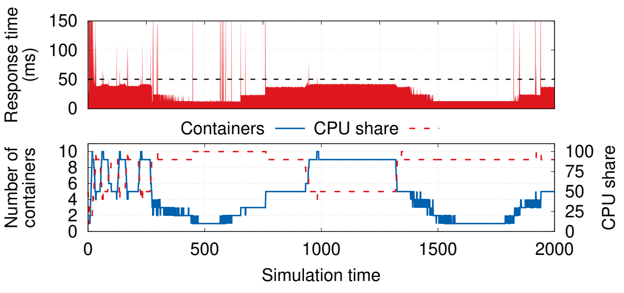
Weights	*RL Policy*	$R_{\max}$ *violations* (%)	*Average CPU utilization* (%)	*Average CPU share* (%)	*Average number of containers*	*Median R* (ms)	*Adaptations* (%)
$w_{\text{perf}} = 0.90$, $w_{\text{res}} = 0.09$, $w_{\text{adp}} = 0.01$	**Q-learning**	30.93	64.66	68.73	3.83	42.14	78.11
	Model-based	3.15	55.12	90.47	3.91	40.56	21.79
$w_{\text{perf}} = 0.09$, $w_{\text{res}} = 0.90$, $w_{\text{adp}} = 0.01$	**Q-learning**	43.18	70.07	58.35	2.81	26.00	76.91
	Model-based	99.80	99.84	11.71	1.20	9101.90	6.30
$w_{\text{perf}} = w_{\text{res}} = w_{\text{adp}} = 0.33$	**Q-learning**	45.53	72.72	64.68	3.26	44.33	77.66
	Model-based	22.59	66.57	78.74	3.74	41.56	12.69

TABLE II: Comparison among the placement policies, when the model-based RL policy manages the application elasticity.

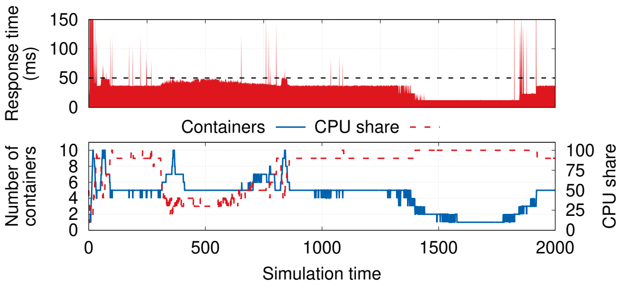
Weights	*Placement policy*	$R_{\max}$ *violations* (%)	*Average CPU utilization* (%)	*Average CPU share* (%)	*Average number of containers*	*Median R* (ms)	*Adaptations* (%)	*Average number of VMs*	$ND_{\max}$ *violations* (%)
$w_{\text{perf}} = 0.90$, $w_{\text{res}} = 0.09$, $w_{\text{adp}} = 0.01$	**OPT**	2.80	55.17	80.23	4.71	28	20.59	2.23	0.0
	NetAware	3	58.06	80.89	4.49	24.56	24.19	2.38	0.0
	Greedy First-fit	3.40	55.33	78.81	4.39	36.56	23.89	2.51	3.10
	Round Robin	6.95	26.97	99.19	7.86	51.73	12.29	7.50	99.35
$w_{\text{perf}} = w_{\text{res}} = w_{\text{adp}} = 0.33$	**OPT**	20.94	65.98	91.77	2.99	37.70	11.44	1.72	0.0
	NetAware	25.74	71.51	62.76	4.95	41	13.54	2.12	0.0
	Greedy First-fit	26.44	70.35	55.13	5.29	41.14	12.89	2.42	1.54
	Round Robin	24.59	65.65	85.92	3.34	49	12.74	3.15	26.49
$w_{\text{perf}} = 0.09$, $w_{\text{res}} = 0.90$, $w_{\text{adp}} = 0.01$	**OPT**	99.80	99.89	11.60	1.15	13949.94	6.30	1.0	0.0
	NetAware	99.80	99.87	11.71	1.20	13948.50	6.30	1.0	0.0
	Greedy First-fit	99.80	99.87	11.71	1.20	13950.15	6.30	1.0	0.0
	Round Robin	99.80	99.87	11.65	1.19	14076	6.10	1.18	4.45



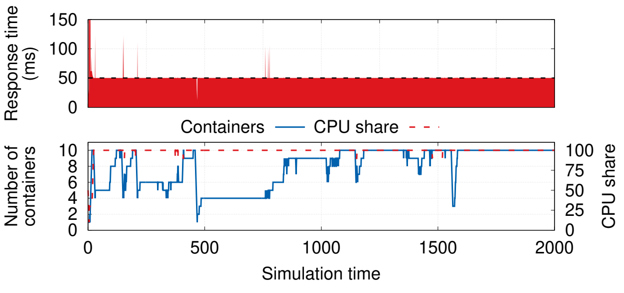
(a) Optimal placement policy (OPT)



(b) Network-aware greedy heuristic (NetAware)



(c) Greedy First-fit heuristic



(d) Round Robin heuristic

Fig. 2: Application performance and run-time deployment adaptation ($w_{\text{perf}} = 0.90$, $w_{\text{res}} = 0.09$, $w_{\text{adp}} = 0.01$).

and $DR_v = 100$ Mbps. The data centers are interconnected with a network delay that ranges in $[10, 70]$ ms interval; the intra-data center network delay is 2 ms. The application requires the network delay to be below $ND_{\max} = 40$ ms. As placement policies, we consider the optimal ILP formulation (**OPT**) and the network-aware greedy heuristic (**NetAware**) presented in Section V, as well as a simple **Greedy First-fit** and a **Round Robin** heuristic. The latter spreads uniformly the application containers across the VMs. Table II reports the experimental results. We first consider the set of weights $w_{\text{perf}} = 0.90$, $w_{\text{res}} = 0.09$, and $w_{\text{adp}} = 0.01$. Figure 2 shows the effects on performance of combining the elasticity and placement policies. First of all, we can see that the model-based RL policy is robust enough to successfully scale the application deployment even when the Round Robin placement policy is used. Figure 2 also shows the importance of using network-aware placement heuristics in a geo-distributed environment. Indeed, the learned adaptation policy differs largely when OPT or Round Robin is used. In particular, Round Robin determines a disadvantageous placement, which uses on average 7.5 VMs, selected from all the data centers. This results in a placement that always violates the $ND_{\max}$ bound and has a high number of $R_{\max}$ violations (see Table II). The Greedy First-fit heuristic inherently allocates the application containers on fewer VMs (2.5, in this case). Nevertheless, it places the application instances on VMs which can be distant with one another, thus improving the variance of the application response time (which makes it difficult to guarantee requirements on percentiles). Indeed, the median response time is higher than that obtained by OPT and NetAware. OPT and NetAware successfully deploy the application on average on 2 VMs, interconnected with network delay below $ND_{\max}$. This efficiently supports the scaling decisions by the RL agent, which registers a very low number of violations on the 95th percentile of the application response time. When the deployment goals are equally important, i.e., $w_{\text{perf}} = w_{\text{res}} = w_{\text{adp}} = 0.33$, the network-aware placement policies confirm their benefits. Indeed, they help the elasticity policy and allocate the application containers on VMs close with one another. Furthermore, they reduce the average number of used VMs to run the application, identifying a trade-off between the computing resource usage and $R_{\max}$ violations. The Round Robin tendency to spread containers negatively affects the performance also in simple scenarios (i.e., $w_{\text{perf}} = 0.09$, $w_{\text{res}} = 0.90$, $w_{\text{adp}} = 0.01$). In the initial RL learning phase, Round Robin uses up to 7 VMs; as a consequence, it violates the $ND_{\max}$ requirement for 4.45% of the time. Conversely, the other policies minimize the number of used VMs, thus resulting in less $ND_{\max}$ violations.

From these experiments, we can conclude that a good deployment policy can be achieved by combining the model-based RL policy, for dynamically scaling containers, and the NetAware heuristic, for allocating containers on geo-distributed VMs. NetAware approximates the optimal formulation behavior, limiting the delays between the VMs used to run the application and optimizing user-defined objective functions. To conclude, we observe that OPT requires to solve the ILP formulation, an NP-hard problem, whereas NetAware uses a heuristic to find an intuitively good solution. This difference clearly appears from the computational time required by the two policies. On average, in our experiments, OPT takes 50.82 ms to compute the placement solution, whereas NetAware (as the other heuristics) requires only 1 ms.

VII. Conclusions

In this paper, we have proposed a two-step approach to jointly optimize the elasticity of application containers as well as their allocation on geo-distributed computing resources. To manage elasticity, we have designed and evaluated model-free and model-based RL solutions, which exploit different degree of knowledge about the system dynamics. To define the containers placement, we have equipped the proposed RL policies with an ILP formulation, as well as with a network-aware placement heuristic. The latter defines the placement of containers on VMs that communicate with non-negligible network latencies. Our results have shown the flexibility and benefits of the proposed approach. The model-based RL approach can successfully learn the best elasticity policy, according to the user-defined deployment goals. Then, the network-aware placement heuristic allocates the application containers on VMs close each another, thus enabling to meet stringent QoS requirements expressed as the 95th percentile of the application response time.

As future work, we plan to further investigate the proposed approach according to two main directions. First, we will extend our heuristics so to efficiently control the deployment of multi-component applications (e.g., microservices). Then, we will include other QoS metrics (e.g., availability, network usage), taking into account the specific features of edge/fog computing environments.

References

[1] Y. Al-Dhuraibi, F. Paraiso, N. Djarallah, and P. Merle, “Elasticity in cloud computing: State of the art and research challenges,” *IEEE Trans. Serv. Comput.*, vol. 11, pp. 430–447, 2018.

[2] M. Nardelli, V. Cardellini, and E. Casalicchio, “Multi-level elastic deployment of containerized applications in geo-distributed environments,” in *Proc. of IEEE FiCloud ’18*, 2018, pp. 1–8.

[3] F. Rossi, M. Nardelli, and V. Cardellini, “Horizontal and vertical scaling of container-based applications using reinforcement learning,” in *Proc. of IEEE CLOUD ’19*, 2019.

[4] S. Horovitz and Y. Arian, “Efficient cloud auto-scaling with SLA objective using Q-learning,” in *Proc. of IEEE FiCloud ’18*, 2018, pp. 85–92.

[5] Y. Al-Dhuraibi, F. Paraiso, N. Djarallah, and P. Merle, “Autonomic vertical elasticity of Docker containers with ElasticDocker,” in *Proc. of IEEE CLOUD ’17*, 2017, pp. 472–479.

[6] C. Kaewkasi and K. Chuenmuneewong, “Improvement of container scheduling for Docker using ant colony optimization,” in *Proc. of KST’17*. IEEE, 2017, pp. 254–259.

[7] M. Nardelli, C. Hochreiner, and S. Schulte, “Elastic provisioning of virtual machines for container deployment,” in *Proc. of ACM/SPEC ICPE ’17 Comp.*, 2017, pp. 5–10.

[8] H. R. Arkian, A. Diyanat, and A. Pourkhalili, “MIST: Fog-based data analytics scheme with cost-efficient resource provisioning for IoT crowdsensing applications,” *J. Netw. Comput. Appl.*, vol. 82, pp. 152–165, 2017.

[9] G. Tesauro, N. K. Jong, R. Das, and M. N. Bennani, “A hybrid reinforcement learning approach to autonomic resource allocation,” in *Proc. of IEEE ICAC ’06*, 2006, pp. 65–73.

[10] X. Guan, X. Wan, B. Y. Choi, S. Song, and J. Zhu, “Application oriented dynamic resource allocation for data centers using Docker containers,” *IEEE Commun. Lett.*, vol. 21, no. 3, pp. 504–507, 2017.

[11] C. Guerrero, I. Lera, and C. Juiz, “Genetic algorithm for multi-objective optimization of container allocation in cloud architecture,” *J. Grid Comput.*, vol. 16, no. 1, pp. 113–135, 2018.

[12] Y. Mao, J. Oak, A. Pompili, D. Beer, T. Han, and P. Hu, “DRAPS: dynamic and resource-aware placement scheme for Docker containers in a heterogeneous cluster,” in *Proc. of IEEE IPCCC ’17*, 2017, pp. 1–8.

[13] C. Barna, H. Khazaei, M. Fokaefs, and M. Litoiu, “Delivering elastic containerized cloud applications to enable DevOps,” in *Proc. of IEEE/ACM SEAMS ’17*, 2017, pp. 65–75.

[14] Z. Huang, K.-J. Lin, S.-Y. Yu, and J. Y. jen Hsu, “Co-locating services in IoT systems to minimize the communication energy cost,” *J. Innovation Digital Ecosyst.*, vol. 1, no. 1, pp. 47–57, 2014.

[15] Z. Tang, X. Zhou, F. Zhang, W. Jia, and W. Zhao, “Migration modeling and learning algorithms for containers in fog computing,” *IEEE Trans. Serv. Comput.*, in press.

[16] H. V. Netto, A. F. Luiz, M. Correia, L. de Oliveira Rech, and C. P. Oliveira, “Koordinator: A service approach for replicating Docker containers in Kubernetes,” in *Proc. of IEEE ISCC ’18*, 2018, pp. 58–63.

[17] H. Arabnejad, C. Pahl, P. Jamshidi, and G. Estrada, “A comparison of reinforcement learning techniques for fuzzy cloud auto-scaling,” in *Proc. of IEEE/ACM CCGrid ’17*, 2017, pp. 64–73.

[18] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*, 2nd ed. Cambridge, MA: MIT Press, 2018.

[19] V. Cardellini, E. Casalicchio, V. Grassi, and F. Lo Presti, “Adaptive management of composite services under percentile-based service level agreements,” in *Proc. of ICSOC ’10*, 2010, pp. 381–395.