Published in Proceedings of the IEEE 6th International Conference on Future Internet of Things and Cloud (FiCloud 2018),
http://dx.doi.org/10.1109/FiCloud.2018.00009

Multi-level Elastic Deployment of Containerized Applications in Geo-distributed Environments

Matteo Nardelli, Valeria Cardellini
Dept. of Computer Science and Civil Engineering
University of Rome Tor Vergata
Rome, Italy
{nardelli,cardellini}@ing.uniroma2.it

Emiliano Casalicchio
Department of Computer Science
Blekinge Institute of Technology, Sweden, and
Sapienza University of Rome, Italy
emiliano.casalicchio@bth.se

***Abstract*—Containers are increasingly adopted, because they simplify the deployment and management of applications. Moreover, the ever increasing presence of IoT devices and Fog computing resources calls for the development of new approaches for decentralizing the application execution, so to improve the application performance. Although several solutions for orchestrating containers exist, the most of them does not efficiently exploit the characteristics of the emerging computing environment.**

In this paper, we propose Adaptive Container Deployment (ACD), a general model of the deployment and adaptation of containerized applications, expressed as an Integer Linear Programming problem. Besides acquiring and releasing geo-distributed computing resources, ACD can optimize multiple run-time deployment goals, by exploiting horizontal and vertical elasticity of containers. We show the flexibility of the ACD model and, using it as benchmark, we evaluate the behavior of several greedy heuristics for determining the container deployment.

I. Introduction

Supported by Cloud and network providers [1], [2], the container technology is changing the way Cloud applications are architected, deployed, and managed, Containers exploit operating system level virtualization to realize flexibility and portability of software. They allow to wrap up an application together with its customized execution environment (i.e., run-time, libraries, code), and to easily run it on any machine, whether bare metal or virtual machine (VM). Several solutions (e.g., Amazon ECS, Google Cloud Platform, Kubernetes) suggest to run containers as a further level of virtualization on top of VMs, so to benefit from speed of the former and isolation of the latter. As a result, an application component runs within a container (e.g., Docker), which, in turn, runs on a VM that can be dynamically acquired and released. The presence of two virtualization layers allows to manage the application deployment at different levels of abstraction, thus opening a wide spectrum of adaptation strategies. These strategies can work in isolation, if they operate on a single level of abstraction (e.g., by only scaling containers), or can exploit a multi-level approach, if they jointly optimize the container deployment as well as the allocation of VMs over the distributed environment.

Funded by BigData@BTH project grant n. 20140032, Knowledge Foundation, Sweden.

The wide diffusion of IoT sensors, wearable devices, and single-board computers fosters the development of new pervasive and latency-sensitive applications, which aim to improve our everyday life (e.g., health-care, environment monitoring, smart traffic lights, and data-driven decision makers — [3], [4], [5], [6], [7]). Recent trends explore the possibility of running Big Data and IoT applications (e.g., [5], [8]) over distributed Cloud and near-edge/Fog computing resources, so to provide low latency and high throughput [9]. Nevertheless, the use of a diffused infrastructure poses new challenges that include network and system heterogeneity, geographic distribution, and non-negligible network delays among distinct nodes processing the distributed application [10]. The emerging scenario is very challenging, and most of the existing solutions for orchestrating containers may not efficiently exploit the geo-distributed computing environment. Indeed, most solutions operate at a single level of abstraction (e.g., [7], [11]), do not consider the geographic distribution of Cloud/Fog environments (e.g., [12]), place containers without jointly considering their scaling (e.g., [7], [13]) and, most importantly, provide best-effort approaches (e.g., [3], [14]) that cannot be easily tuned to optimize different deployment objectives (e.g., improve application availability, reduce response time, network usage, or cost). Moreover, there is no general formulation of the container deployment problem, which makes it difficult to analyze and compare the different existing solutions.

In this paper, we aim to fill this gap by proposing Adaptive Container Deployment (**ACD**), a general formulation of the deployment and adaptation problem of containerized applications over geo-distributed infrastructures. ACD can be equipped with different optimization objectives, e.g., aimed to minimize resource load unbalance, network traffic or cost, maximize application throughput, or a combination thereof; in this paper, we model the deployment and adaptation costs of running containers on VMs. ACD considers applications where multiple software components (e.g., microservices) should be elastically scaled at run-time, so to efficiently handle the incoming workload. Differently from existing solutions (e.g., [12], [13], [15]), ACD allocates the application containers over multiple geo-distributed VMs, so to optimize specific deployment objectives. Then, to preserve these objectives at run-time, it can horizontally and vertically scale the application containers,

while also acquiring and releasing VMs as needed. ACD explicitly models the presence of two layers of virtualization and, by providing a general formulation of the deployment problem, it represents a benchmark against which heuristics can be evaluated.

The paper is structured as follows. We model ACD as an Integer Linear Programming (ILP) problem (Sections II and III), which can be used to optimize different QoS metrics; specifically, we formalize the deployment cost and adaptation cost. We show the flexibility of the proposed solution and, using ACD as benchmark, we evaluate various well-known deployment heuristics (Section IV). We discuss related works in Section V and conclude in Section VI.

II. System Model and Problem Definition

A. System Model

Computing Infrastructure Model. We consider a geographically distributed computing environment, where multiple Cloud data centers and Fog micro-data centers provide VMs on-demand (i.e., VMs can be acquired and released at run-time as needed). A VM offers computing and memory resources that can be exploited to run containerized applications. Although in theory unlimited, we assume that the number of VMs that can be leased in a certain time period is limited (e.g., at most 50 new VMs can be acquired at once). Let V be the set of all VMs, including the active (leased) ones and those turned off but leasable. A VM $v \in V$ has the following characteristics: C_v, the total CPU capacity, expressed in number of CPU cores; M_v, the total available memory capacity; DR_v, the download data rate from an external container repository; and T_v, its boot time. As will be clarified later, the external repository (e.g., Docker Hub or Docker Store) allows to download, prepare, and launch application instances by means of containers. Since data centers are scattered on a geo-distributed environment, distinct computing resources can communicate with a non-negligible delay. We model the logical connectivity (u, v) between each pair of VMs $u, v \in V$, which is characterized by a network delay $d_{u,v}$. Such a logical connectivity between computing resources results by the underlying physical network paths and routing strategies.

Container Model. Following the Docker model, a container is an instance of a container image (or simply *image*), which represents a container snapshot and contains all the data needed for its execution. When a container has to be instantiated on a VM, the latter needs to download the container image from an external repository (e.g., Docker Hub), if the image is not already on the VM. During its execution, a container accesses resources exposed by the hosting VM and it can be configured with specific quotas that limit how much of the available CPU and memory resources it can use. In Docker, this quota configuration can be scaled at run-time with (practically) no downtime. Without loss of generality, we consider that a container can assume one of the quota configurations belonging to the finite set S. Each configuration $s \in S$ is characterized by the CPU quota c_s, expressed in number of CPU cores, the memory quota m_s, and the container boot time on a reference computing resource, T_s.

Application Model. We consider a very general application model, where the application consists of a set of components, which cooperate in order to accomplish a common goal (e.g., [4], [6]). An application component (or, simply, *component*) is a black-box entity that carries out specific tasks (e.g., perform computation, store and access data sets) and can interact with other components. To properly process increasing incoming workloads, multiple instances of the *same* component can be created and executed in parallel. In such a way, each instance works autonomously and processes a subset of the incoming requests. From an operational perspective, we assume that each component instance runs in a software container, e.g., by leveraging on Docker. An application A consists of a set of components $a \in A$ that are interconnected by logical links (a, b), with $a, b \in A$; the latter enable the data exchange between components. Each application component $a \in A$ is characterized by the following resource requirements: C_a, its CPU demand, and M_a, its memory demand. The resource demand usually changes at run-time because of the varying application workload; it follows from non-functional requirements, such as throughput, response time, and size of the data set to be stored in memory. Other resource requirements (e.g., disk capacity, network bandwidth) can be similarly considered. We assume that the resource requirements of components can be achieved by aggregating CPU and memory resources from multiple computing nodes. We rely on containers to easily deploy, manage, and run multiple instances of the application components. Therefore, each application component $a \in A$ is associated to a container image (used to launch component instances), whose size is I_a; if the container image is not locally available on the computing resource, it can be fetched from an external repository. To model the logical connectivity between the application components $a, b \in A$, we resort on a *connectivity matrix* $R : |A| \times |A| \rightarrow \{0, 1\}$, where $R_{a,b} = 1$ if a and b exchange data using the network, and $R_{a,b} = 0$ otherwise. Since communicating components can be executed on distinct computing nodes, the application also exposes, as requirement, the maximum network delay between any pair of communicating components' instances, $R_{\max}$.

B. Problem Definition

The application deployment problem consists in determining *how* and *where* to run the application components on the computing infrastructure. Due to the multiple virtualization layers, it deals with the identification of the number and type of containers that will execute the application components, as well as the placement of containers on the computing infrastructure. The overall fleet of containers and their placement should be determined so to meet the components requirements, in terms of resource demand (i.e., C_a and M_a) and network delay (i.e., $R_{\max}$).

At run-time, the application can be subject to varying workloads, which lead its components to change the demand of computing C_a and memory M_a resources. To preserve

performance, we should solve the *adaptive container deployment* problem, which determines whether the application deployment should be conveniently updated at run-time, while explicitly taking into account the adaptation costs. Because of the two virtualization layers, it represents a multi-level optimization problem. At the first level, it deals with the elastic adaptation of the number and type of containers used to run the application components, by leveraging on vertical and horizontal scaling. At the second level, it oversees the container placement on a set of VMs that can be elastically acquired and released on demand.

As regards container scaling, we define the following operations. A *vertical scale* changes the container configuration, thus increasing or reducing the quota of resource that is granted to the application component. This operation can be performed with practically no downtime. A *horizontal scale* changes the number of containers used to run the application components, by adding (scale-out) or removing (scale-in) a component instance. Differently from vertical scaling operations, performing a scale-out introduces an adaptation cost, which considers, e.g., the time needed to launch the new containers and, if needed, acquire new VMs.

III. Adaptation Problem

In this section, we define the QoS metrics that drive the application deployment and its adaptation. Then, we formulate the ACD problem.

A. ACD Variables

We model the application deployment with integer variables $x_{a,s,v} \in \mathbb{N}$ representing the number of containers of type $s \in S$ deployed on the VM $v \in V_a$ that run an instance of the application component $a \in A$. We denote the application deployment as $\boldsymbol{x} = \langle x_{a,s,v} \rangle$, with $a \in A$, $s \in S$, and $v \in V$.

We find convenient to introduce the binary variables z_v with $v \in V$, which define whether the VM v is active, i.e., turned on (condition expressed as $z_v = 1$). To determine when the application should be more conveniently redeployed, e.g., in face of changing incoming workload, we introduce a set of variables that keep track of the current deployment (if exists). We define the variables $x'_{a,s,v}$ and z'_v, which maintain the value of $x_{a,s,v}$ and z_v, respectively, as computed at the previous optimization step by ACD. In case of initial deployment, these variables store an empty deployment, whereas at run-time they store the application deployment as computed by ACD during its latest execution. We denote the previous application deployment as $\boldsymbol{x}' = \langle x'_{a,s,v} \rangle$, with $a \in A$, $s \in S$, $v \in V$.

B. QoS Metrics

We are interested in determining an application deployment on a geographically distributed computing infrastructure, which minimizes the deployment cost. Moreover, at run-time, we want to conveniently reconfigure the application deployment so to efficiently handle varying workloads.

Deployment cost. We define the deployment cost $Z(\cdot)$ as the overall number of VMs leased for running all the application components in A. Given the application deployment vector $\boldsymbol{x}$, the deployment cost can be defined as:

$$Z(\boldsymbol{x}) = \sum_{v \in V} z_v \tag{1}$$

where the variable $z_v \in \{0, 1\}$ denotes whether the VM $v \in V$ is turned on (condition expressed as $z_v = 1$). We can readily determine the value of z_v by leveraging on $\boldsymbol{x}$; indeed, v is active if it hosts at least one application component. A linear formulation of this condition can be expressed as follows:

$$\frac{\sum_{a \in A} \sum_{s \in S} x_{a,s,v}}{\Gamma} \leq z_v \leq \sum_{a \in A} \sum_{s \in S} x_{a,s,v} \tag{2}$$

where Γ is a large constant. Note that, although we provide a simple model of deployment cost, others definitions can be similarly considered, e.g., so to control the monetary cost for the allocation of VMs over different billing time units.

Adaptation cost. We define the adaptation cost $D(\cdot)$ as the time needed to reconfigure the application deployment. This term takes into account the time needed to spawn new VMs, retrieve the application container images, and finally start the containers. Given the deployment vectors $\boldsymbol{x}$ and $\boldsymbol{x}'$, we have:

$$D(\boldsymbol{x}, \boldsymbol{x}') = \max_{v \in V} (T_v \cdot \delta_v) + \max_{v \in V} \sum_{a \in A} \left(\frac{I_a}{DR_v} \cdot b_{a,v} \cdot \dot{x}_{a,v} \right) + \\ + \max_{a \in A} \max_{s \in S} (T_s \cdot \delta_{a,s}) \tag{3}$$

where the first term on the right hand side models the longest boot time of any VM used for A, the second term represents the time needed to download the container images of the application components on v, and the third term represents the longest boot time of any container used for A. The binary constant $b_{a,v}$ indicates if the component image I_a is not yet available on v and has to be downloaded from the external repository (in such a case, $b_{a,v} = 1$).

The binary variables δ_v indicate whether a new VM $v \in V$ should be turned on and used to execute some application components. We define δ_v as follows:

$$z_v - z'_v \leq \delta_v \leq \frac{z_v + (1 - z'_v)}{2} \tag{4}$$

The variable $\dot{x}_{a,v} \in \{0, 1\}$ denotes whether at least one component $a \in A$ should be executed on $v \in V$ (in such a case, $\dot{x}_{a,v} = 1$). Formally, we have:

$$\frac{\sum_{s \in S} x_{a,s,v}}{\Gamma} \leq \dot{x}_{a,v} \leq \sum_{s \in S} x_{a,s,v} \tag{5}$$

The binary variable $\delta_{a,s}$ determine whether a scale-out operation should be performed for $a \in A$, by adding a new container of type $s \in S$: $\delta_{a,s} = 1$ if exists any $v \in V$ such that $x_{a,s,v} - x'_{a,s,v} > 0$, $\delta_{a,s} = 0$ otherwise. A linear definition of $\delta_{a,s}$ can be formulated as $\Gamma \cdot \delta_{a,s} \geq x_{a,s,v} - x'_{a,s,v}$, which must hold true $\forall v \in V$.

Vertical scaling operations are implicitly captured by $\boldsymbol{x}$ and $D(\boldsymbol{x}, \boldsymbol{x}')$. A component $a \in A$ is vertically scaled if, given

$v \in V$, the container configuration of a on v is changed from r to s, with $s \neq r \in S$ (i.e., if $\exists s, r \in S$, with $s \neq r$, such that $x_{a,r,v} < x'_{a,r,v}$ and $x_{a,s,v} > x'_{a,s,v}$). This reconfiguration does not introduce an adaptation cost in $D(\boldsymbol{x}, \boldsymbol{x}')$.

Application constraints. Since the application components can be executed on VMs disseminated over a geo-distributed environment, the application requires to limit the network delay between the communicating application components. To satisfy this requirement while computing the application deployment, we model the network delay between the application components and require it to be below the upper bound $R_{\max}$, i.e., $R_{a,b} \cdot d_{u,v} \cdot y_{(a,b)(u,v)} \leq R_{\max}$, $\forall a, b \in A, u, v \in V$, where $R_{a,b}$ indicates if the components a and b, with $a, b \in A$ exchange data, $d_{u,v}$ is the network delay between $u, v \in V$, and $y_{(a,b)(u,v)}$ is a binary variables, whose value is 1 if at least one instance of a should be deployed on u and at least one instance of b should be deployed on v. By definition, we have that $y_{(a,b)(u,v)} = \dot{x}_{a,u} \cdot \dot{x}_{b,v}$, with $a, b \in A$ and $u, v \in V$.

C. Adaptation Problem Formulation

The ACD formulation determines the application deployment on geo-distributed computing resources and, at run-time, evaluates benefits and costs of its adaptation[1]. To select the best deployment among all the feasible ones, ACD uses an objective function that minimizes the deployment and adaptation cost. We formalize the objective function as follows:

$$F(\boldsymbol{x}, \boldsymbol{x}') = w_z \frac{Z(\boldsymbol{x})}{Z_{\max}} + w_a \frac{D(\boldsymbol{x}, \boldsymbol{x}')}{D_{\max}} \tag{6}$$

where $w_z, w_a \geq 0$, $w_z + w_a = 1$, are weights for the different QoS attributes, and $Z_{\max}$ and $D_{\max}$ denote respectively the maximum value for the overall expected deployment and adaptation cost.

We formulate ACD as an ILP problem as follows:

min $\bar{F}(\boldsymbol{x}, \boldsymbol{x}')$

subject to:

$$\mathcal{V} \geq T_v \cdot \delta_v \qquad \forall v \in V \quad (7)$$

$$\mathcal{A} \geq \sum_{a \in A} \left(\frac{I_a}{DR_v} \cdot b_{a,v} \cdot \dot{x}_{a,v} \right) \qquad \forall v \in V \quad (8)$$

$$\mathcal{S} \geq T_s \cdot \delta_{a,s} \qquad \forall a \in A, s \in S \quad (9)$$

$$C_v \geq \sum_{a \in A} \sum_{s \in S} c_s \cdot x_{a,s,v} \qquad \forall v \in V \quad (10)$$

$$M_v \geq \sum_{a \in A} \sum_{s \in S} m_s \cdot x_{a,s,v} \qquad \forall v \in V \quad (11)$$

$$C_a \leq \sum_{s \in S} \sum_{v \in V} c_s \cdot x_{a,s,v} \qquad \forall a \in A \quad (12)$$

$$M_a \leq \sum_{s \in S} \sum_{v \in V} m_s \cdot x_{a,s,v} \qquad \forall a \in A \quad (13)$$

$$R_{\max} \geq R_{a,b} \cdot d_{u,v} \cdot y_{(a,b)(u,v)} \qquad \forall a, b \in A, \forall u, v \in V \quad (14)$$

$$\dot{x}_{a,u} = \sum_{v \in V} y_{(a,b)(u,v)} \qquad \forall a, b \in A, u \in V \quad (15)$$

$$\dot{x}_{b,v} = \sum_{u \in V} y_{(a,b)(u,v)} \qquad \forall a, b \in A, v \in V \quad (16)$$

In the problem formulation, we replaced the objective function $F(\boldsymbol{x}, \boldsymbol{x}')$, which is not a linear expression in $\boldsymbol{x}$ due to the maximum operations in $D(\boldsymbol{x}, \boldsymbol{x}')$, with a linear function $\bar{F}(\boldsymbol{x}, \boldsymbol{x}')$. The latter is obtained from $F(\boldsymbol{x}, \boldsymbol{x}')$ by replacing the adaptation cost $D(\boldsymbol{x}, \boldsymbol{x}')$ with the auxiliary function $\bar{D}(\boldsymbol{x}, \boldsymbol{x}') = \mathcal{V} + \mathcal{A} + \mathcal{S}$, where $\mathcal{V}$, $\mathcal{A}$, and $\mathcal{S}$ are defined in (7), (8), and (9), respectively. To explain why this formulation works, we consider the term $\mathcal{V} \geq T_v \cdot \delta_v$. Since $\mathcal{V}$ must be larger or equal than its right-hand side, $T_v \cdot \delta_v$, and at the optimum $\mathcal{V}$ is minimized, then $\mathcal{V} = \max_{v \in V} (T_v \cdot \delta_v)$. Similar arguments apply also to $\mathcal{A}$ and $\mathcal{S}$. The constraints (10) and (11) limit the placement of application components on a VM $v \in V$ according to its available resources. Equations (12) and (13) guarantee that the demand of resources by each application component $a \in A$ is satisfied. Constraint (14) limits the network delay between any pair of component instances to the upper bound $R_{\max}$. Finally, constraints (15) and (16) model the logical AND between the deployment variables, that is, $y_{(a,b)(u,v)} = \dot{x}_{a,u} \cdot \dot{x}_{b,v}$.

We can demonstrate that ACD is an NP-hard problem; indeed, it is a generalization of the Partition problem, which is known to be NP-hard. Due to space limitations, we omit the proof.

IV. Experiments

We evaluate the ACD model through a set of numerical experiments that aim at demonstrating the formulation flexibility. In Section IV-B, we analyze how ACD allows us to optimize different QoS metrics, such as deployment cost, adaptation cost, and a combination thereof. Then, in Section IV-C, we use ACD as a benchmark against which we compare three greedy heuristics.

A. Experimental Setup

We run the experiments on an Amazon EC2 VM (c4.xlarge: 4 vCPU and 7.5 GB RAM). To solve ACD, we use CPLEX© (version 12.6.3), the state-of-the-art solver for ILP problems.

We consider an application with 5 components $a \in \{0, \cdots, 4\}$ that exchange traffic on bidirectional links; the resulting non-zero values of the connectivity matrix R are $R_{0,1} = R_{0,2} = R_{2,3} = R_{3,4} = 1$. Due to the lack of reference workloads for geo-distributed containerized applications, we generate a synthetic resource demand for each application component as follows. At time t, the demand of computing C_a and memory M_a resources by $a \in A$ is defined as $f_a(t) = f_a(t - \Delta_a) \pm r_a$, where Δ_a and r_a are stochastic variables. We extract r_a from a Poisson distribution ($\mathcal{P}$, with parameter λ) and limit the resulting $f_a(t)$ in $[f_a^{\min}, f_a^{\max}]$. The rate at which the resource demand changes, Δ_a, can be extracted from either an exponential distribution (Exp, with parameter μ) or a lognormal distribution (Lgn, with parameters μ and σ). Table I reports the parameters used for each component. The resulting resource demand is represented in Figure 1; we readily see that the application includes 3

[1]Note that ACD realizes the Plan component within the MAPE reference model for autonomous systems [16]. Indeed, it is in charge of determining the adaptation actions with the aim of meeting specific optimization objectives.

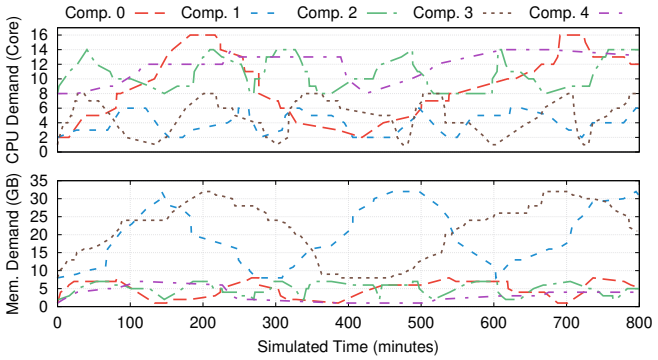

Fig. 1. Periodic workload generated for each application component $a \in A$.

CPU-intensive and 2 memory-intensive components. Each component $a \in A$ defines a container image whose size is 330 MB (value obtained as the average size of 25 popular Docker images). Moreover, the application requires that the maximum network delay between any two components that exchange data is $R_{\max} = 80$ ms.

We define four different container configurations in S, with quota $(c_s, m_s) = \{(1,1), (2,4), (2,8), (4,4)\}$, respectively, where c_s is expressed in CPU cores and m_s in GB. The container boot time, T_s, is selected uniformly in $[8.5, 11.5]$ s. The computing infrastructure includes two different types of VMs, where one type is twice the other: specifically, type A with $C_v = 4$ vCPU and $M_v = 16$ GB RAM, and type B with $C_v = 8$ vCPU and 32 GB RAM. Each VM has a boot time T_v extracted with a uniform probability in $[85, 115]$ s, i.e., one order of magnitude greater than the container boot time. We also consider that the VM v has a download data rate from the container repository, DR_v, equal to 100 Mbps. We consider a geographically distributed infrastructure, where VMs are interconnected with a non-negligible network delay uniformly distributed in $[10, 100]$ ms.

To adapt the application deployment at run-time, aiming to satisfy the varying resource demand, we periodically solve the ACD model, every 5 minutes of simulated time.

B. Optimizing QoS Metrics

The ACD model allows us to compute the application deployment while optimizing different QoS metrics, whose importance depends on the utilization scenario. In this experiment, we show the effect of different optimization objectives on the application run-time performance, expressed in terms of deployment cost and adaptation cost (defined in (1) and (3), respectively). We first solve ACD by optimizing a single QoS metric. For example, to optimize the deployment cost, we set the weights as $w_z = 1$ and $w_a = 0$ in (6). Then, we optimize the multi-objective function by uniformly weighting each metric contribution (i.e., $w_z = w_a = 0.5$); from previous experiments, we obtain the following normalization factors, that we use in (6): $D_{\max} = 152.90$ s and $Z_{\max} = 12$.

Figure 2 reports the application performance in terms of different QoS metrics: the deployment cost measures the

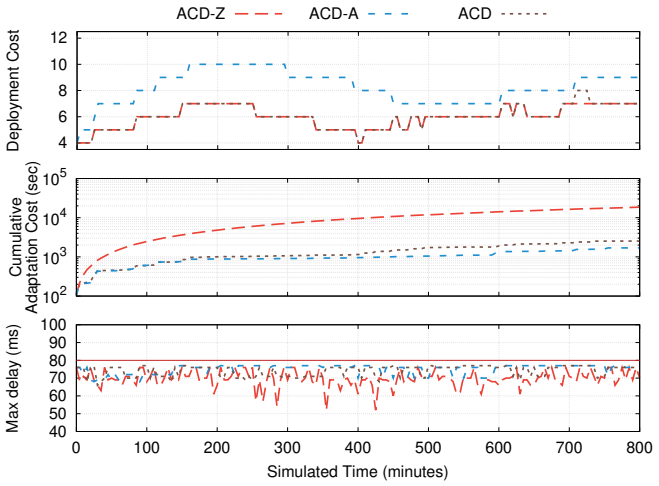

Fig. 2. Impact of different optimization objectives on the application QoS metrics.

number of active VMs; the cumulative adaptation cost[2] represents the overall time spent for adapting the application deployment, since $t = 0$ s; and max delay is the maximum network delay experienced between any two communicating application components, whose value should be lower than $R_{\max}$ (the latter is represented as a horizontal red line).

When ACD optimizes the deployment cost (namely, *ACD-Z*), it always uses as few VMs as possible. Therefore, whenever possible, ACD reconfigures the application deployment so to consolidate its containers on a reduced number of nodes. Since ACD-Z continuously starts containers and VMs (as the incoming workload increases), it obtains a rather high cumulative adaptation cost. Table II reports different percentiles of the adaptation cost during the whole experiment. We can see that, on average, the adaptation cost of ACD-Z is 115 s.

When ACD optimizes the adaptation cost (namely, *ACD-A*), the latter QoS metric is reduced of about 91%. Since ACD-A does not consolidate containers on fewer VMs, we can see from Figure 2 that it uses up to 10 VMs (on average, 3 VMs more than ACD-Z). As expected, ACD-A under-uses the resources available on the active VMs, down to 76.4% at 400 minutes (the lowest value).

When ACD optimizes both the QoS metrics (labeled as *ACD*), the application experiences deployment and adaptation costs that are very close to the values obtained for the single-objective optimization (see Figure 2). Observe that, soon after 700 minutes, ACD uses a slightly higher number of VMs than ACD-Z, because it prefers to not reconfigure the application deployment so as to consolidate it on fewer VMs. Indeed, this operation would introduce an adaptation cost. As regards network delay, we can see from Figure 2 that, independently from the optimization objective, ACD always meets the application requirement.

[2]In Figures 2 and 3, we represent the cumulative adaptation cost instead of the adaptation cost so to better show the overall resulting performance.

TABLE I
PARAMETERS USED TO DEFINE C_a AND M_a FOR EACH APPLICATION COMPONENT $a \in A$.

Component	r_a	$C_a^{\min}$	$C_a^{\max}$	r_a	$M_a^{\min}$	$M_a^{\max}$	Δ_a distribution
0	$\mathcal{P}(1)$	2	16	$\mathcal{P}(1)$	1	8	EXP(0.001)
1	$\mathcal{P}(1)$	2	6	$\mathcal{P}(2)$	8	32	EXP(0.080)
2	$\mathcal{P}(1)$	8	14	$\mathcal{P}(1)$	2	7	EXP(0.060)
3	$\mathcal{P}(1)$	1	8	$\mathcal{P}(2)$	8	32	LGN(5.70, 1.00)
4	$\mathcal{P}(1)$	8	14	$\mathcal{P}(1)$	1	7	LGN(7.50, 1.00)

TABLE II
AVERAGE AND PERCENTILE VALUES OF THE APPLICATION QOS METRICS: DEPLOYMENT COST $Z(\boldsymbol{x})$, I.E., NUMBER OF ACTIVE VMS; ADAPTATION COST $D(\boldsymbol{x}, \boldsymbol{x}')$, EXPRESSED IN SECONDS.

	$Z(\boldsymbol{x})$						$D(\boldsymbol{x}, \boldsymbol{x}')$					
	avg	min	50th	75th	95th	max	avg	min	50th	75th	95th	max
ACD-Z	5.99	4	6	7	7	7	115.26	10.91	117.77	123.4	124.03	124.11
ACD-A	8.30	4	8	9	10	10	10.09	0	0	9.73	104.92	125.11
ACD	6.01	4	6	7	7	8	15.22	0	0	9.80	103.64	116.70
Greedy	8.61	6	8	10	11	11	10.88	0	0	9.22	106.75	123.46
Greedy H	10.74	5	11	12	12	12	9.21	0	0	9.73	10.91	124.06
Greedy V	6.76	6	7	7	7	7	2.87	0	0	0	10.02	123.46

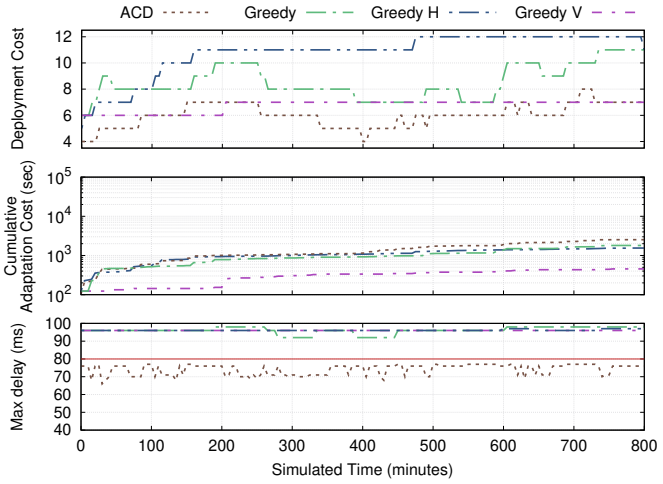

Fig. 3. Comparison of Greedy First-fit heuristics against ACD.

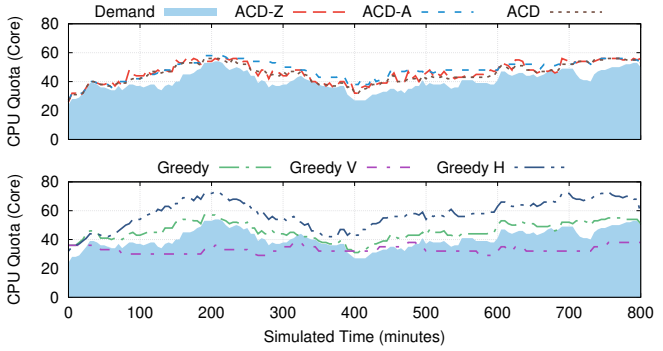

Fig. 4. Overall amount of CPU resource, in number of cores, demanded by the application components and allocated by the different deployment policies.

C. Evaluating Greedy Heuristics

In this experiment, we use ACD as a benchmark against which we evaluate different heuristics. We consider Greedy First-fit, i.e., one of the most popular heuristics used to solve the bin-packing problem and often adopted by open-source products to address the placement problem.When new resources are needed, *Greedy First-fit* first scales vertically and then, if needed, scales horizontally the component containers. Conversely, when resources should be released, Greedy First-fit first scales-in horizontally and then, if possible, vertically on the running containers. We also consider other two greedy heuristics, namely *Greedy H* and *Greedy V* that perform only horizontal and vertical scaling operations, respectively. Figure 3 reports the application QoS metrics. We use ACD that optimizes the multi-objective function as benchmark; note that ACD adapts the application deployment by means of vertical and horizontal scaling operations.

Greedy V performs only vertical scaling, meaning that it cannot add new containers to run the application; however, it can migrate the existing ones on other VMs, where more resources can be exploited. From Figure 3 we can see that Greedy V obtains the lowest possible adaptation costs (on average, 2.9 s) and uses at most only 7 VMs to run 12 containers. Nevertheless, since Greedy V cannot run in parallel more than 12 containers, it often fails to accommodate the resource demand by all the application containers. For this heuristic, determining the initial number of containers is crucial to accommodate run-time workload fluctuations. Figure 4 allows to better visualize this limitation; due to space limitations, the figure reports only the overall demand and allocated quota of CPU cores by each deployment strategy.

Greedy H can change the number of application containers

at run-time. Therefore, as represented in Figure 3, it results in higher values of deployment and adaptation costs: on average, it uses 11 VMs and imposes an average adaptation cost of 9.2 s. Differently from Greedy V, Greedy H properly allocates the CPU demand (see Figure 4), even though it reserves more resources than needed. This depends on the criteria used to select the container configuration: when the application component has heterogeneous requirements (e.g., CPU and memory resources), determining the best container configuration to be used is not trivial.

Greedy First-fit (referred as *Greedy* in Figure 3) exploits the strengths of Greedy H and Greedy V: it scales horizontally containers to satisfy the resource demand (see Figure 4), and recurs to vertical scaling to better exploit the available resources. Greedy First-fit uses on average 85% of the available resources on each VM, whereas Greedy H and Greedy V use 77% and 78%, respectively. Note that Greedy First-fit uses about 43% more computing resources than ACD (44% more than ACD-Z). This happens because Greedy First-fit does not consolidate the application containers on fewer VMs.

Most importantly, the Greedy heuristics are not network-aware. Hence, as represented in Figure 3, they cannot satisfy the application requirement on the maximum network delay $R_{\max}$ between its communicating components. We believe that ACD can be used to devise new deployment policies that can overcome the limitations of the Greedy First-fit heuristics and that can better exploit the available computing resources while satisfying the application requirements.

On Resolution Time. We briefly discuss the resolution time of the different deployment strategies. The Greedy First-fit heuristics are very fast, determining the application deployment in 3 ms on average. Nevertheless, they do not explicitly optimize the deployment objectives and cannot meet the application requirements on network delays. These feature are of crucial importance on distributed Cloud/Fog computing environments. The single-objective strategies that solve the ILP formulation, i.e., ACD-Z and ACD-A, experienced on average resolution time of 954 ms and 714 ms, respectively. Computing the deployment while optimizing a multi-objective function is harder, and ACD spent on average 17.26 s to determine the best solution. Note that, although higher than the other approaches, this is still a reasonable amount of time for our medium-size experimental setting. Although ACD suffers from scalability issues (being NP-hard), it provides interesting insights that can be help to design new and efficient heuristics that can overcome the limitations of the existing approaches.

V. Related Work

The fast increasing adoption of container technologies and the decentralization of computation away from Cloud data centers call for effective deployment and management strategies for containerized applications, also addressing their run-time adaptation [17]. We can classify existing research works that focus on container deployment according to the following main directions: (1) the deployment goals, (2) the span of control, (3) the actions and methodologies that can be used for determining or adapting the deployment, and (4) the distribution of the managed computing resources.

The *optimization goals* pursued by container deployment and management solutions include the improvement of application performance (e.g., [3]), load balance and resource utilization (e.g., [13], [18], [19]), energy efficiency [8], and the reduction of deployment cost (e.g., [15]). In some cases, a combination of deployment objectives is considered (e.g., to improve data locality and load balance [7]).

As regards the *span of control*, some works deal with the initial placement (e.g., [7], [12], [19]), while the majority addresses the deployment run-time adaptation (e.g., [3], [11], [19]), so to enable the containerized application to preserve its performance in face of changing working conditions.

The *actions* that control the deployment of containerized applications include the container placement on the underlying computing nodes (either physical or virtual), the container scaling (horizontal, vertical, or a combination thereof), and the container migration. When containers are placed on VMs, a second level of deployment can entail the VM allocation over the distributed environment. Most of the works consider a single level of deployment (e.g., [7], [11], [13], [14], [15]); in this paper, we consider a multi-level deployment, as also done in [3], [20].

To determine or adapt the deployment on the underlying infrastructure, the approaches proposed so far recur to two main *methodologies*: mathematical programming and heuristics. Mathematical programming approaches consider the initial placement of containers (e.g., [7], [19]) as well as their run-time deployment adaptation (e.g., [12], [15], [18]). C-Port [18] is the first example of orchestrator that deploys and manages containers across multiple Clouds using a constraint-programming model for resource selection. Since the model details are not provided, a comparison with our approach is not possible. Mao et al. [19] present an IP formulation of the initial container placement aiming to maximize the available resources in each node. To fulfill this objective at run-time, an heuristic conveniently migrates the resource-intensive containers to another node. Guan et al. [15] also consider the container scaling. They propose a LP formulation that determines the number of containers and their placement on a static pool of physical machines; however, vertical scaling operations are not considered. Nardelli et al. [12] propose an ILP formulation of the elastic provisioning of VMs for container deployment, taking explicitly into account the heterogeneity of container requirements and VMs. Comparing to them, we consider an enhanced geo-distributed model of the application deployment as well as the action of scaling vertically the containers. Furthermore, differently from our work, all these approaches do not take into account the adaptation costs.

The mostly used heuristics range from well-known approaches for solving the bin-packing problem (e.g., greedy first-fit and best-fit), to meta-heuristics [13], to specifically designed solutions [7], [19], to threshold-based heuristics [3], [11], [14], [20]. In particular, the latter are exploited to determine at run-time the elastic scaling of containers and

represent the most popular scaling methodology as for the Cloud infrastructure layer [21]. Kaewkasi et al. [13] propose an ant colony optimization algorithm that schedules containers on a static pool of resources; nevertheless, they do not consider scaling actions. Orchestration frameworks that support the container placement, such as Kubernetes and Docker Swarm, use simple heuristics (e.g., to pack or spread containers) and allow to specify placement constraints (e.g., only on a subset of nodes) and preferences (e.g., co-location). As regards container scaling, most solutions, including Kubernetes, Docker Swarm, and Amazon ECS, provide best-effort threshold-based policies based on some load metrics. Barna et al. [3] estimate performance metrics by leveraging on a layered queuing network model of the system. Casalicchio et al. [14] aim to improve the resource allocation and fulfill application response time constraints. Khazaei et al. [20] compare some thresholds against a combination of CPU, memory, and network usage so to horizontally scale both containers and VMs. ELASTICDOCKER [11] also employs a threshold-based policy that, differently from our solution, only scales vertically both the CPU and memory resources assigned to each container. Such a policy is similar to the Greedy V heuristic, so it may suffer from the same problems in accommodating the resource demand. Vertical scaling can also be used to reduce resource consumption in IoT scenarios [8].

As last classification factor, we consider the *distribution of the computing resources* on which the deployment takes place. Most solutions cited so far are designed for a single cluster or data center; therefore, they neglect to consider communication delays that may affect the performance of multi-container applications deployed on geo-distributed infrastructures (e.g., federated and distributed Clouds, Fog systems). Network traffic as a driver for scaling is considered by Zhao et al. [7]. They study how to allocate local I/O-intensive and network-intensive containerized applications over a Cloud infrastructure. Interestingly, their solution tries to grasp the effect on resource contention by multiple concurrent applications. Nevertheless, it only addresses the initial placement problem. Nathan et al. [22] propose to reduce the deployment time by exploiting a cooperative management of Docker images. Hoque et al. [10] assess how container orchestration tools meet Fog requirements for IoT applications and propose an architecture for a Fog-based container orchestrator.

VI. CONCLUSIONS

We have presented ACD, a formulation of the deployment and adaptation problem for containerized applications over geo-distributed environments. ACD is a flexible formulation that can be conveniently configured to optimize different QoS metrics; in this paper, we have considered the deployment cost and the adaptation cost. By leveraging on numerical experiments, we have validated our solution and shown the model flexibility, which can optimize single- and multi-objective functions. We have also shown that ACD can be used as a benchmark framework against which to compare other allocation strategies.

As future work, we plan to extend the proposed formulation of ACD so to model other QoS metrics (e.g., response time, availability) and adaptation actions (i.e., migration). ACD solves an NP-hard problem, so it suffers from scalability issues. Therefore, starting from the drawbacks of existing heuristics, we plan to develop efficient heuristics to deal with large problem instances.

REFERENCES

[1] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes, "Borg, Omega, and Kubernetes," *ACM Queue*, vol. 14, pp. 70–93, 2016.

[2] R. Cziva and D. P. Pezaros, "Container network functions: Bringing NFV to the network edge," *IEEE Commun. Mag.*, vol. 55, no. 6, pp. 24–31, 2017.

[3] C. Barna, H. Khazaei, M. Fokaefs, and M. Litoiu, "Delivering elastic containerized cloud applications to enable DevOps," in *Proc. SEAMS '17*, 2017, pp. 65–75.

[4] F. Bonomi, R. Milito, J. Zhu, and S. Addepalli, "Fog computing and its role in the Internet of Things," in *Proc. MCC '12*. ACM, 2012, pp. 13–16.

[5] W. Gerlach, W. Tang, K. Keegan, T. Harrison *et al.*, "Skyport: Container-based execution environment management for multi-cloud scientific workflows," in *Proc. DataCloud '14*. IEEE, 2014, pp. 25–32.

[6] S. Yi, C. Li, and Q. Li, "A survey of Fog computing: Concepts, applications and issues," in *Proc. Mobidata '15*. ACM, 2015, pp. 37–42.

[7] D. Zhao, M. Mohamed, and H. Ludwig, "Locality-aware scheduling for containers in cloud computing," *IEEE Trans. Cloud Comput.*, 2018.

[8] A. Asnaghi, M. Ferroni, and M. D. Santambrogio, "DockerCap: A software-level power capping orchestrator for Docker containers," in *Proc. IEEE EUC '16*, 2016, pp. 90–97.

[9] B. Varghese and R. Buyya, "Next generation cloud computing: New trends and research directions," *Future Gener. Comput. Syst.*, vol. 79, pp. 849–861, 2018.

[10] S. Hoque, M. S. d. Brito, A. Willner, O. Keil, and T. Magedanz, "Towards container orchestration in fog computing infrastructures," in *Proc. IEEE COMPSAC '17*, vol. 2, 2017, pp. 294–299.

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

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

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

[14] E. Casalicchio and V. Perciballi, "Auto-scaling of containers: The impact of relative and absolute metrics," in *Proc. IEEE 2nd Int'l Workshops on Foundations and Applications of Self* Systems*, 2017, pp. 207–214.

[15] 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.

[16] J. O. Kephart and D. M. Chess, "The vision of autonomic computing," *IEEE Computer*, vol. 36, no. 1, pp. 41–50, 2003.

[17] C. Pahl, "Containerization and the PaaS cloud," *IEEE Cloud Comput.*, vol. 2, no. 3, pp. 24–31, 2015.

[18] M. Abdelbaky, J. Diaz-Montes, M. Parashar, M. Unuvar, and M. Steinder, "Docker containers across multiple clouds and data centers," in *Proc. IEEE/ACM UCC '15*, 2015, pp. 368–371.

[19] Y. Mao, J. Oak, A. Pompili, D. Beer *et al.*, "DRAPS: dynamic and resource-aware placement scheme for docker containers in a heterogeneous cluster," in *Proc. IEEE IPCCC '17*, 2017.

[20] H. Khazaei, H. Bannazadeh, and A. Leon-Garcia, "SAVI-IoT: A self-managing containerized IoT platform," in *Proc. IEEE FiCloud 2017*, 2017, pp. 227–234.

[21] T. Lorido-Botran, J. Miguel-Alonso, and J. A. Lozano, "A review of auto-scaling techniques for elastic applications in cloud environments," *J. Grid Comput.*, vol. 12, no. 4, pp. 559–592, 2014.

[22] S. Nathan, R. Ghosh, T. Mukherjee, and K. Narayanan, "CoMICon: A co-operative management system for Docker container images," in *Proc. IEEE IC2E '17*, 2017, pp. 116–126.