On the Optimality of Concurrent Container Clusters Scheduling over Heterogeneous Smart Environments

A. Asensio, X. Masip-Bruin, J. Garcia, S. Sánchez

Advanced Network Architectures Lab, Universitat Politècnica de Catalunya, CRAAX-UPC, Spain.
{aasensio, xmasip, jordig, sergio}@ac.upc.edu

***Abstract*— Smart environments rely on the cloud for most computation activities; however, leveraging the availability of resources at the edge could complement the cloud capabilities to provide a resources continuum spectrum that takes advantage of the benefits of both technologies, cloud and edge computing. In this scenario, applications are usually decomposed for execution in sets of tasks, which are in turn encapsulated into virtual components such as containers. Containers are lightweight implementations of virtual machines, improving efficiency and portability in distributed applications. Traditional containers scheduling in cloud is a well-known problem, but when the environment is heterogeneous, as it is in the scope of edge computing and edge-cloud systems, the problem becomes more challenging. In this paper we present the Concurrent Container Clusters Scheduling problem (C3S) aimed at optimizing the problem of placing containers in clusters of heterogeneous nodes satisfying a set of resource requirements, quality of service limitations, and considering additional stringent constraints in terms of applications execution in isolation for security guaranteeing. The C3S problem has been formulated using Integer Linear Programming with the dual objective of minimizing the number of applications rejected while minimizing the number of nodes used for computation. We have evaluated the optimality of this approach, analyzed the performance in terms of solving time and, finally, created a heuristic approach to solve the problem in realistic high demanding scenarios.**

Keywords*— edge computing, edge-cloud systems, optimal scheduling, heterogeneous system, smart environment.

I. INTRODUCTION

Smart environments are envisioned as a technological context aimed at bringing in a high impact on the whole society (e.g., enhancing people's quality of life, improving industrial systems efficiency, optimizing energy consumption), to be deployed in a wide range of domains, including smart cities, smart manufacturing or smart homes. Typically, these smart environments involve a massive sensors network spread along the whole environment, connected through low power wide area networks to the cloud data center, where smart services leverage the collected data to process complex analyses and provide some sort of intelligent utilities.

One of the core components in any smart environment is the cloud infrastructure. Cloud computing provides unlimited computing and storage capabilities, ubiquity, elasticity and efficiency, benefiting from an economy of scale model. Data generated through the sensors network are collected and stored at cloud before being processed. However, moving all data to cloud, which presumably will be physically far from the sensing devices location, adds several inconveniences, such as network overloading, high communication latencies, and high security and privacy risks [1]. To mitigate these obstacles, new solutions must be sought intended to take advantage of locality. In this arena, fog [2] and edge [3] computing came up. There is no wide consensus on using a unique wording to accommodate such locality concept, and many references may be found in the literature proposing solutions for both fog and edge computing. In this context, and with no aim to dig into this discussion, in this paper we will use the term edge to refer to such a locality concept. Edge computing provides a virtualized environment leveraging the capabilities of available resources at the edge of the network, thus bringing computing resources closer to data sources. Interestingly, although the computing capacity of edge devices is much lower than that of the cloud, in some scenarios, especially smart environments, the computing potential at the edge could be enormous. For instance, in [4] authors estimate the potential computing capacity in a major European city, showing a theoretical aggregated peak performance up to 10 or even 20 times higher than the largest Amazon Web Service (AWS) cloud data center. Edge computing has not been conceived to replace the cloud, as shown in [5], rather, both cloud and edge could be seen as a continuum global heterogeneous environment, combining the advantages of both technologies and, therefore, providing enhanced options for efficient smart services execution [6].

In the cloud, services are usually encapsulated using a virtualization technology and then distributed for execution throughout several nodes of the data center. Indeed, although virtual machines are extensively used in the context of cloud computing, there are several limitations yet remaining unsolved, such as poor performance, long boot time, and complex scheduling. As an alternative, a container-based technology provides a much lighter virtual environment, largely improving efficiency and portability in distributed applications deployment. This technology is becoming increasingly popular in cloud computing. In fact, most cloud providers have created their own container-based infrastructure (Amazon Elastic Container Service, ECS [7], Google Kubernetes Engine, GKE [8], or Microsoft Azure Container Services [9]) to facilitate the creation, deployment and management of users' distributed services.

Containers scheduling in cloud is a well-known and largely-studied problem. Containers are usually managed through a queue, from which the scheduler fetches on-the-fly one container at a time. This strategy poses several limitations for selecting an effective placement of the entire workload due to the lack of a global view. Alternatively, concurrent container scheduling provides a global workload view, but solving an optimal scheduling strategy may become highly complex, notably affecting the service's quality requirements. As illustrated in [10], by analyzing the Google cluster traces the scheduler needs to make in peak hours hundreds of container placement decisions per second. Furthermore, in the presence of heterogeneous clusters (as they are in smart environments), efficient containers orchestration becomes more complex yet. Containers require a combination of different resources features (e.g., CPU, memory, network) and nodes in the cluster may have diverse capacities and availabilities. Finally, when considering distributed applications demanding affinities among containers, placing the set of containers on the appropriate nodes can substantially enhance data locality, reduce inter-node communication and latency, and balance the containers execution among nodes. In such context, however, finding an appropriate concurrent containers scheduling is still an open challenge. In this paper, we present the Concurrent Container Clusters Scheduling (C3S) problem. It tackles the optimization problem of placing containers into the nodes that satisfy their resource requirements and guarantee communication between containers while also satisfying certain Quality of Service (QoS). In detail, we consider the Google Kubernetes [11] approach assuming that: *i*) one or more containers can be grouped in a pod; and *ii*) application's pods need to be placed in suitable nodes, all communicated to ease the application deployment. Despite considering Google Kubernetes, it is worth recalling that our proposal is not limited to it. In fact, the C3S problem is applicable to containers, generally speaking. Moreover, the C3S problem considers stringent constraints not only in terms of resources required and QoS guarantees but also in terms of execution of applications in isolation, for security reasons, when required. We model the C3S problem using an Integer Linear Programming (ILP) formulation intended to minimize the number of applications rejected while minimizing the number of nodes utilized. In addition, we propose a heuristic algorithm to solve the problem in realistic scenarios and compare its performance against that of the ILP in different contexts.

The main contributions of this paper are:

- A formal description of the C3S problem; which considers not only resources allocation constraints but also QoS requirements impacting on the container clusters creation and including additional container isolation constraints.
- An ILP formulation to model the C3S problem; which can be considered to validate and evaluate other placement solutions before implementing them in real testbeds.
- A scalable iterative algorithm that can obtain near-optimal solutions in short times.

The remainder of this paper is organized as follows. In Section II background on the reference architecture considered is described, and relevant and recent works that can be found in the literature related to the container scheduling problem are reviewed. Next, in Section III, the C3S problem is presented. Section IV presents the formal description of the problem including an ILP formulation; whereas Section V describes the proposed heuristic algorithm. Then, Section VI presents the scenarios considered in this paper to validate and compare the proposed mathematical model and the heuristic algorithm and shows the obtained results. Finally, sections VII and VIII present the future work considered by the authors and conclusions, respectively.

II. Background and Related Work

Aimed at reinforcing the technological basis that supports the problem presented in this paper (which requires a centralized scheduler with global knowledge about computing resources availability and applications' requirements), this section starts describing a tentative architecture aimed at facilitating the deployment of such scheduler. Moreover, related work is reviewed and main differences to the paper contribution are highlighted.

A. Management architecture

Ideally, in order to optimally place containers, schedulers should be aware of both all concurrent applications' requests and the real availability of each individual resource. Unfortunately, it is difficult to assume the existence of a centralized scheduler in current real-world systems that may know all nodes' computing resources availability and even the needs of all potential applications. Indeed, dynamics of computing and applications make this assumption impractical. In theory, a decentralized architecture may be highly beneficial to face these limitations, hence let us consider an edge-cloud scenario based on a hierarchical and distributed architecture, similarly to that presented in [12].

In the considered architecture, nodes are grouped in clusters, by layers, and each node is assigned a role; thus imposing control and management functions and relationships among them [13]; e.g., nodes acting as cluster heads implement functions to manage and control resources from the underlying nodes and to communicate with nodes in the immediate upper layer through a platform manager [14]). Fig. 1 shows an example of a 4-layered architecture. In this example, the logical topology of an 11-nodes network is depicted. Each node represents a set of resources and it is assigned into a cluster and a layer; e.g., nodes N7 and N8 form a cluster jointly with N4, which acts as the cluster head; similarly, nodes N4, N5 and N2 (cluster head) form another cluster. In this architecture, cluster heads store information about resources of the underlying nodes.

Interestingly, assuming this architecture, a set of concurrent requests received by a set of nodes can be forwarded to the selected cluster head in upper layers to be processed, thus defining resources view. That is, let us assume that nodes N7 and N8 receive an application request each; named to as *Request 1* and *Request 2* in the figure. The first scheduler deployment option would be N7 and N8 to process *Request 1* and *Request 2* respectively, considering only their local resources. This would turn into a low available resources and low delay approach. The second option would consider to forward both requests up in the

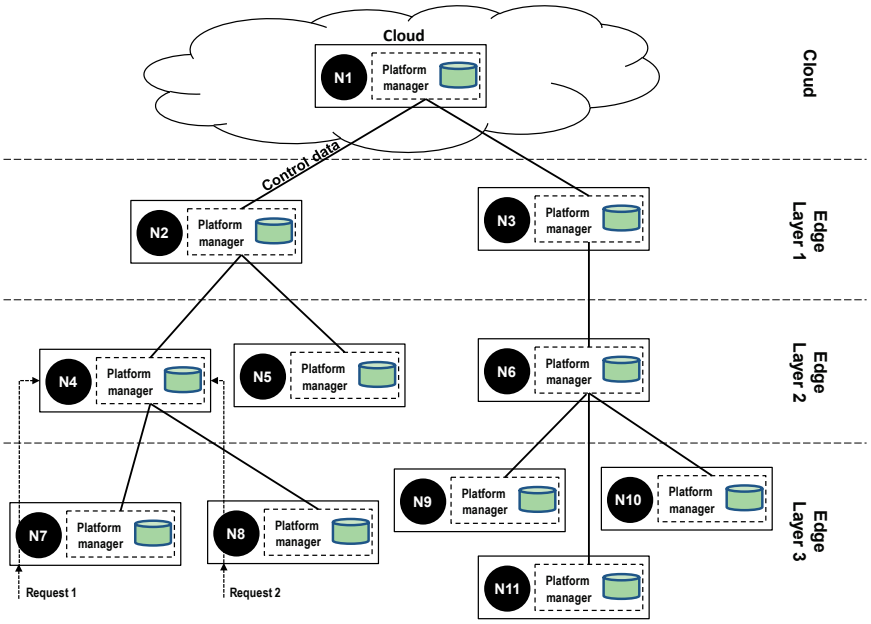

Fig. 1. Logical topology considering a hierarchical an distributed architecture.

hierarchy, reaching out to the cluster head N4, where a scheduler can process that set of requests jointly, being aware of both information about resources availability from the nodes in the cluster and information from the platform management databases. This approach would turn into a not so low delay but with a substantial increment in the set of available resources. Furthermore, this second deployment option may keep forwarding the requests upstream till having access to a large set of resources. For example, by delegating the scheduling execution to node N2, in the upper layer, the view of resources can be increased, including not only resources from nodes N7, N8 and N4, but also from nodes N5 and N2. Therefore, service providers managing container-based services could benefit from implementing their resource schedulers at different layers to balance their needs in terms of having information of few local resources at near-real time or increasing resources information at expenses of increasing delay. It is clear that, a scheduler in cloud will be penalized by the highest delay but there will be few or none limitation regarding resources availability; differently, a scheduler placed in an edge layer (edge layers 1, 2 and 3 in the figure), will have information of limited resources but at reduced delay.

B. Related work

Although orchestration platforms for container-based infrastructure, such as Google Kubernetes and Docker Swarm [15], provide their own algorithms, container scheduling has become a hot topic in the recent years attracting the interest of the research community.

Google Kubernetes and Docker Swarm schedulers are usually considered as the state-of-the-art algorithms to make container placement decisions. The default algorithm in Kubernetes tends to balance resource usage, assuming multi-resource (i.e., different types of resources) requirements [10]; whereas the binpack strategy in Swarm tends to consolidate the load into the most loaded machines. These two approaches perfectly match the underlying concepts of two well-known strategies in cloud computing: *i*) load balancing [16] and *ii*) task consolidation [17]; being the latter of particular interest for energy-saving and resource usage maximization. However, more strategies can be found far beyond these two approaches. Indeed, Swarm provides not only one but three container scheduling strategies; namely, spread, binpack and random. In brief, the spread strategy aims at deploying containers in nodes having the freest CPU and memory, being a node either a physical machine or a virtual machine (VM). Opposite to spread, the binpack strategy aims at deploying containers in those nodes having the highest CPU and memory usage. Finally, the random strategy will distribute the containers randomly, among the entire set of nodes.

Considering the aforementioned strategies (i.e., spread, binpack and random), the authors in [18] propose a multi-objective container scheduling algorithm and compare their algorithm against Swarm. The proposed algorithm considers the following factors: nodes' CPU and memory usage, time to transmit images in the network, association between containers and nodes, and clustering of containers. Although similitudes with our work can be found (e.g., both consider nodes' CPU and memory usage, association between containers and nodes, and clustering of containers), in this manuscript we propose an ILP formulation considering concurrent container scheduling as well as security-related constraints. Moreover, our objective function is totally different from their proposal.

Also compared to Swarm's spread and random strategies, the authors in [19] propose a container scheduling algorithm based on Particle Swarm Optimization (PSO) to balance resource usage, intended to improve applications' performance. Our work clearly differs, since it focuses on a different strategy (consolidation), proposes an ILP formulation as well as a heuristic algorithm instead of PSO. Furthermore, in the studies carried out in [18] and [19], both the number of nodes and containers is very low; i.e., 6 and 5 node clusters are considered in their experiments, respectively, what notably hides the scope of these contributions.

Interestingly, authors in [10] analyze the concurrent container scheduling problem, against the mostly utilized container-by-container scheduling approaches Kubernetes and Swarm algorithms use as reference. Notice that container scheduling approaches (container-by-container and concurrent) are later described for the sake of understanding, in Section III. In that work, the authors model the concurrent container scheduling problem as a minimum cost flow problem, turning into a strategy so-called most-loaded heuristic, inspired by vector bin packing algorithms and aligned with the previously described idea of task (container) consolidation. Indeed, heuristics based on the bin packing problem and focusing on VM placement or VM consolidation have been widely studied in the literature, even in recent works, see [20], [21]. However, those works do not focus on the container-based infrastructure and they do not either face challenges arising in novel scenarios such as those considering not only heterogeneous machines but also edge and cloud resources.

Although in our work we face the concurrent container scheduling problem, substantial differences with the work above reviewed can be found. The most relevant differences are the following. In our approach, an ILP model and an iterative heuristic algorithm are proposed; our proposal to manage container clusters imposes additional constraints than those considered by the approaches previously cited (in [18] and [10]). Specifically, the container affinity approach tries to place containers having affinity in the same machine. However, in our approach, we assume that containers having affinity create a cluster that poses certain QoS requirements among its containers. Therefore, assuming different QoS levels between containers becomes both realistic and flexible enough to facilitate the deployment of a variety of container clusters requiring different QoS and to guarantee that the required QoS is not violated even in the case that containers of the same cluster are placed in distant machines. Moreover, we also consider security-related constraints in terms of isolation of containers [22]. A detailed description on the container clusters approach and the isolation requirements considered in this paper are provided in the next section.

Although the work in [23] focuses on the particular case of optimizing container scheduling for data-intensive in serverless edge computing scenarios, it shares some similitudes with ours and describes certain challenges and limitations in their proposal and in state-of-the-art edge systems that are of interest for this paper. Similar to in our work, in that paper Kubernetes pods are considered and the proposed algorithm is based on scoring; authors define a set of priority functions and describe a method to optimize the weights of each priority function based on three particular scenarios. Among the current limitations, it is worth highlighting that current container-based systems lack of knowledge about clients location with respect to nodes; thus making difficult the selection of nodes close to the clients to reduce latency between client and nodes. Interestingly, the assumed architecture could easily solve this drawback. In addition, container-based systems may arise some drawbacks with respect to isolation and multitenancy; therefore, in our constraints we also include intuitive constraints related to isolation.

Moreover, although some of the works reviewed remark the idea that these kind of scheduling problems in cloud computing are NP-hard ([10], [19]), the authors believe that providing a mathematical model may contribute not only to formally state the problem, but also to validate future container scheduling algorithms before being implemented in real testbeds. In fact, ILP has been considered before formally stating the VM placement problem in cloud computing. Interestingly, in a recent work [24], authors focus on the utilization of ILP for deriving test suites and evaluating the quality of VM placement implementations; and, similarly, authors in [25] propose a mixed ILP formulation to determine an efficient placement on mobile edge servers, which poses an additional constraint to the scheduling problem.

Finally, it is worth highlighting that reviewed works considering a container-based infrastructure do not pay attention to a potential lack of resources in the machines forming the infrastructure to serve the incoming requests. Certainly, assuming enough resources in the container-based infrastructure may be realistic in some cluster configurations, e.g., large-scale clusters. However, the lack of resources may be of paramount importance in those clusters having local and limited resources, –in fog-based [2] or fog-cloud-based [12] scenarios–, where several services requiring to take advantage of fog computing capabilities (such as low latency) need to be deployed in a set (or even in a subset) of shared resources in a smart scenario. Moreover, although container-by-container scheduling has the advantage to get parallelized [26], it is worth mentioning that parallelization can be also explored in our proposal.

III. Problem Description

In this section, we start describing the container scheduling problem from two distinct approaches: *i*) container-by-container scheduling and *ii*) concurrent container scheduling. Then, we introduce the concept of container clusters and, next, based on the concurrent container scheduling extended with containers clustering, we present the Concurrent Container Clusters Scheduling (C3S) problem tackled in this manuscript, which imposes stringent constraints to schedule the containers.

A. Container scheduling

Since cloud computing was introduced in [27], the explosion of cloud services has become a reality. Indeed, such explosion can be quantified with the large number of requests to deploy applications at cloud, see [28]. Obviously, the tremendous number of requests directly impacts on how container schedulers perform and surely on the strategies to be defined to guarantee an optimal services performance in such highly demanding scenario. Two different approaches can be found to schedule containers: *i*) container-by-container scheduling and *ii*) concurrent container scheduling.

In brief, container-by-container scheduling assumes the processing of one single request at a time and is responsible for deciding its placement. Although container-by-container scheduling can be executed in parallel, placement decisions are made without knowledge from the other concurrent requests. Therefore, any placement decision may impact negatively on placement decisions of other concurrent requests.

Differently from the container-by-container approach, concurrent container scheduling can mitigate the impact of container placement decisions on concurrent requests, by providing knowledge about requirements from a set of requests to the scheduling algorithm. It is worth noting that, in both approaches, container-by-container and concurrent container scheduling, placement decisions must guarantee resources enough as required. Without loss of generality, in this paper we assume three different resources to be guaranteed: *i*) CPU, *ii*) memory, and *iii*) runtime environment; i.e., the scheduler needs to guarantee that candidate nodes to host containers do not violate the accepted SLA for these resources when making placement decisions.

For readability purposes, we refer to container-by-container and to concurrent container scheduling approaches, in general, without focusing on a particular container-based infrastructure orchestrator. However, it is worth introducing the naming utilized from here on out in this paper. Specifically, aligned with the evolution described in [22] from containers relying on Linux kernel containment features (LXC) [29] to Docker containers to Kubernetes, we assume the naming from Kubernetes and refer to pod as the unit to deploy. Each pod consists of one or more containers and it is assigned its own IP address, thus facilitating containers clumping turning into container clusters.

B. Container clusters

To deploy an application, a number of containers may be required. These containers can be gathered into a single pod or they can be part of different pods. Moreover, affinity among those containers impose certain needs in terms of communication requirements among them. In addition, these needs may become a must in some novel scenarios, specifically, those requiring real-time or near real-time processing or those being data-intensive to prevent high load in the network.

These requirements may be included in the form of SLA parameters, mapping the required quality to be delivered into a user deploying an application. Consequently, different QoS levels can be defined among the working nodes in the cluster (e.g., to represent network delay among them), and users can request a certain QoS level to be guaranteed among the containers of their applications. In this case, containers requiring a specific QoS level cannot be placed in any node having enough resources but only in those nodes having enough resources and also satisfying the desired QoS level among them.

C. The C3S problem

Undoubtedly, cloud computing has led and facilitated the development of new services. Interestingly, the advent of edge computing and the wide adoption of the Internet of Things (IoT) concepts, have opened new opportunities to service providers and application developers, specifically for those services and applications requiring, among others, stringent constraints imposed by real-time applications. Moreover, considering both cloud and edge computing, services should be able to select either a resource or a group of them from the entire set of resources, which has been referred to as cloud-to-thing continuum ([30], [31]) or resources continuum [32].

As described in [10], resources required by containers need to be guaranteed by the schedulers. Interestingly, in the cloud-to-thing continuum, resources can be found either at the edge of the network or at the cloud. In such scenario, where a wide variety of services can be deployed, from real-time services to computing intensive services (e.g., data analytics), placement decisions cannot be based only on resource guarantees but also other SLA parameters must be met to guarantee the expected QoS. In addition, the container placement problem can also face other highly restrictive constraints, for example those related to security issues. Indeed, in [22], the author analyses security related aspects in container-based infrastructures, ending up in proposing, as also highlighted in [33], the deployment of one container per host (physical machine or VM) as a fundamental rule to provide real security isolation. In this paper we also adopt this security approach and consequently, we assume that, should an application require such level of security, all required containers (pods) must be deployed in hosts or VMs with no containers (pods) belonging to other applications.

In summary, we assume a set of applications to be deployed, each having several pods (having, at least, one container each), and an active scheduler on the desired layer as described in Section II.A. That scheduler makes placement decisions concurrently over the entire set of applications, to deploy the containers (pods) in physical machines or VMs, satisfying the resource requirements in terms of: CPU, memory and runtime environment. In addition, two more aspects are considered. First, we also assume that a defined QoS must be guaranteed among containers (pods) from the same request to form container clusters. Second, regarding security guarantees and container isolation, we assume that, for applications requiring isolation in the SLA, a host can only deploy containers (pods) of one single application. Moreover, in such stringent scenario we rely on a multi-objective optimization approach with a twofold objective, to deploy the maximum number of applications (main objective), and to deploy the applications in the minimum number of hosts (secondary objective). This approach is similar to the strategy deployed by Swarm binpack and the task consolidation approach in cloud computing, which have been demonstrated to be energy efficient, being also of particular interest in IoT-based and smart environments.

Fig. 2 illustrates a set of applications' requests and their requirements (Fig. 2a) and the nodes of a container-based infrastructure and their available resources (Fig. 2b). In Fig. 2a, we show 3 applications (represented in the figure by APP1, APP2, and APP3) requiring the deployment of 4 pods, each with its respective containers (C1, C2, C3, and C4). Unlike other, POD1 requires to be deployed in a specific runtime environment (we refer to environment as ENV in the figure, in both the depicted nodes and pods), and also requires to be executed in isolation. To illustrate affinity between containers, containers deployed for APP2 (i.e., containers in POD2 and POD3) require a QoS level of 1 among them; that is, these containers need to be deployed in the same node or in distant nodes that do not violate such QoS level among them.

To illustrate the problem tackled in this paper and the differences between container-by-container and concurrent container approaches both described in Section III.B, let us assume a scheduler based on the container-by-container approach. In this case, the scheduler first places POD1, then evaluates POD2 placement, etc. We consider that the scheduler places container C1

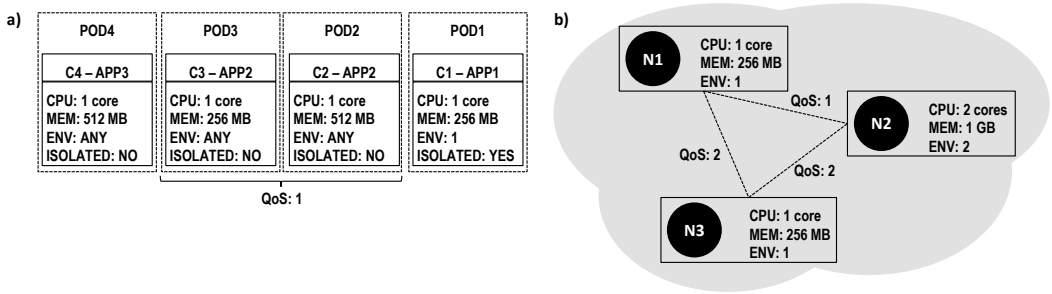

Fig. 2. Set of requests (a) and cluster of resources (b).

at node N1, since it is the first node that satisfies the required constraints in terms of available resources (CPU and memory), runtime environment, and isolation, since no other container is deployed in the node. Afterwards, containers C2 and C3 are deployed at node N2, guaranteeing the QoS level required through the affinity between containers C2 and C3 to build up the aforementioned container cluster. However, container C4 could not be deployed in this case, since there are no available nodes in the cluster that satisfy its requirements (i.e., the available node N3 does not meet APP3 requirements). Therefore, the request from APP3 would be rejected.

Differently, if the scheduler is based on a concurrent container scheduling approach, knowledge about requirements from the whole set of concurrent containers to be deployed provides highly valuable information to decide the placement of each container. Specifically, placing container C1 at node N3, container C4 at node N2, and containers C2 and C3 at nodes N2 and N1, respectively, results in a solution enabling all containers to be deployed without violating any of the requirements, neither in terms of resources required nor in terms of isolation and/or the QoS required in the container cluster formed by C2 and C3.

Although relevant works can be found in the recent literature proposing novel scheduler algorithms, to the best of our knowledge, only few works tackle the problem from the concurrent container approach. Being aware of recent works highlighting the advantages of concurrent container against container-by-container scheduling (e.g., [10]), the contribution in this paper goes far beyond them, addressing the concurrent container scheduling problem, enriched with multi-resource purposes, QoS level guarantees for container clustering as well as security guarantees based on a multi-objective function approach, as described in Section II. The authors believe that, by considering the whole set of constraints identified by the aforementioned requirements, a much more complete and realistic scenario is defined, demanding for a novel solution to describe and to model the concurrent container approach, while, to the best of our knowledge, no work faces the proposed problem considering all those constraints.

IV. Problem Formulation

In this section, first the C3S problem is formally stated. Then an ILP formulation is proposed to model it.

A. Problem statement

The C3S problem can be formally stated as follows:

Given:

- A network topology represented by the graph $G(E, N)$, where E represents the set of edges and N the set of nodes. Parameter $\Delta_{nn'}$ represents the QoS level at edge $e_{nn'}$ between nodes $n, n' \in N$, in terms of delay.
- Each node $n \in N$ represents either a physical or a virtual machine. A node n is defined by the tuple: $<n, T_n^{CPU}, T_n^{MEM}, \Gamma_n^{CPU}, \Gamma_n^{MEM}, \Phi_n, \rho_n>$, where n represents the node, T_n^{CPU} represents its CPU resources, and T_n^{MEM} represents its total memory. Similarly, Γ_n^{CPU} and Γ_n^{MEM} represent available CPU and memory resources in the node, respectively. Φ_n represents the runtime environment in the node; $\Phi_n \in O$, being O the set of runtime environments. Parameter ρ_n represents if there is any pod already deployed at node n.
- A set of applications A.
- A set of pods P_a representing the pods to deploy application $a \in A$. As part of the QoS to be guaranteed, each pod is defined by the tuple $<p, \gamma_p^{CPU}, \gamma_p^{MEM}>$, where p represents the pod $p \in P_a$, γ_p^{CPU} represents the required amount of CPU, and γ_p^{MEM} represents the required amount of memory.
- Parameter ξ_a indicates if any runtime environment can be considered (ξ_a =0) or the one represented by parameter ϕ_a must be selected (ξ_a =1).
- Parameter ψ_a representing if the execution of application a, needs to be isolated (ψ_a=1) or it should not (ψ_a=0); i.e., only pods of the given application can be deployed in the same node.
- Parameter δ_a, representing the maximum delay allowed among pods for application a.

Objective:

- To minimize the number of applications rejected while minimizing the number of nodes utilized.

Output:

- The pod placement for each application.

B. Definitions and mathematical model formulation

The objective of the C3S problem is twofold: *i*) minimizing the number of applications rejected, and *ii*) minimizing the number of nodes utilized. The first objective is of paramount importance in scenarios requiring stringent constraints as part of the SLA parameters needed to guarantee the desired QoS and in scenarios with scarce resources. Note that an application is deployed if and only if all its pods are deployed, which increases the complexity of the scheduling problem. Moreover, the second objective is of particular interest to save energy, as described in the previous section.

In order to model the C3S problem, the following sets and subsets have been defined:

N	Set of nodes; i.e., physical machines and VMs.
A	Set of applications.
P	Set of pods (alternatively, Docker containers).
P_a	Subset of pods, $P_a \subseteq P$, that belong to application $a \in A$. $\lvert P_a \rvert$ represents the number of pods for application a.
O	Set of runtime environments.

In addition, the following parameters need to be defined:

T_n^{CPU}	Total resources of node $n \in N$ in terms of CPU.
Γ_n^{CPU}	Available resources at $n \in N$ in terms of CPU.
T_n^{MEM}	Total resources of node $n \in N$ in terms of memory.
Γ_n^{MEM}	Available resources at node $n \in N$ in terms of memory.
Φ_n	Runtime environment $\Phi_n \in O$ in node n.
ρ_n	Parameter representing if there is a pod already deployed at node n.
$\Delta_{nn'}$	Parameter representing the QoS level between nodes n and n', in terms of delay.
ξ_a	Binary parameter to represent if a specific runtime environment is required by the application (value 1) or it is not (value 0).
ϕ_a	Runtime environment required by application a if ξ_a =1.
δ_a	Parameter representing the minimum QoS level between pods; e.g., maximum delay allowed between pods of the application.
ψ_a	Parameter representing if application a must be executed in isolation; i.e., all its pods must be deployed in nodes having no pods from other applications.
γ_p^{CPU}	Required CPU to deploy pod p.
γ_p^{MEM}	Required memory to deploy pod p.
Ψ	Weighting parameter to tune the contribution to cost of the main objective; i.e., to minimize the number of rejected applications.
Λ	Weighting parameter to tune the contribution to cost of the secondary objective; i.e., to minimize the number nodes utilized.

The decision variables are defined as follows:

x_a	Binary variable representing if application a is deployed (x_a=1) or not (x_a=0).
y_{pn}	Binary variable representing if pod p is deployed at node n (y_{pn}=1); otherwise it takes value 0.
z_n	Binary variable accounting the number of nodes utilized.

Then, the ILP model for the C3S problem is defined as follows:

$$Min\ \Psi \cdot \sum_{a \in A} (1 - x_a) + \Lambda \cdot \sum_{n \in N} z_n \tag{1}$$

subject to:

$$x_a \leq \frac{\sum_{p \in P_a} \sum_{n \in N} y_{pn}}{|P_a|} \quad \forall a \in A \tag{2}$$

$$\sum_{n \in N} y_{pn} \leq x_a \quad \forall a \in A, p \in P_a \tag{3}$$

$$\sum_{p \in P} \gamma_p^{CPU} \cdot y_{pn} \leq \Gamma_n^{CPU} \quad \forall n \in N \tag{4}$$

$$\sum_{p \in P} \gamma_p^{MEM} \cdot y_{pn} \leq \Gamma_n^{MEM} \quad \forall n \in N \tag{5}$$

$$\sum_{a' \in A | a' \neq a} \sum_{p' \in P_{a'}} y_{p'n} \leq bigM \cdot \left(2 - x_a - \psi_a \cdot y_{pn}\right) \forall a \in A, p \in P_a, n \in N \tag{6}$$

$$\psi_a \cdot y_{pn} \leq \left(1 - \rho_n\right) \ \forall a \in A, p \in P_a, n \in N \tag{7}$$

$$\xi_a \cdot \phi_a \cdot y_{pn} \geq \xi_a \cdot \Phi_n \cdot y_{pn} - bigM \cdot (1 - x_a) \ \forall a \in A, p \in P_a, n \in N \tag{8}$$

$$\xi_a \cdot \phi_a \cdot y_{pn} \leq \xi_a \cdot \Phi_n \cdot y_{pn} + bigM \cdot (1 - x_a) \ \forall a \in A, p \in P_a, n \in N \tag{9}$$

$$\sum_{n' \in N} \Delta_{nn'} \cdot y_{p'n'} \leq \delta_a + bigM \cdot \left(1 - y_{pn}\right) \forall a \in A, p \in P_a, p' \in P_a, n \in N \tag{10}$$

$$z_n \geq \frac{1}{2} \cdot \left[\left(\frac{T_n^{CPU} - \Gamma_n^{CPU}}{T_n^{CPU}} \right) + \frac{\sum_{a \in A} \sum_{p \in P_a} y_{pn}}{\sum_{a \in A} |P_a|} \right] \quad \forall n \in N \tag{11}$$

The objective function (1), jointly minimizes the number of applications that cannot be deployed and the number of nodes utilized. Weighting parameter Ψ is defined to properly tune the weight of the first term (minimize the number of applications rejected) to be the primary objective; whereas weighting parameter Λ is defined aimed at tuning the contribution of the secondary objective (minimizing the number of nodes utilized in the system) by activating ($\Lambda > 0$) or deactivating ($\Lambda = 0$) it. Equation (2) computes if all pods of the application are deployed and, accordingly, the application is deployed. Equation (3) guarantees that if application *a* is deployed, then each pod *p* of the application is deployed in one and only one node. Equations (4) and (5) ensure that capacity of the node is not exceed in terms of CPU and memory, respectively. Equation (6) guarantees that if an application that requires to be executed in isolation is deployed, there are no pods from other applications co-located with its pods in the same node. Moreover, (7) guarantees that, pods from applications requiring to be isolated are not deployed in nodes having pods from other applications already deployed. Equations (8) and (9) ensure that applications deployed and requiring a specific runtime environment are deployed in nodes hosting that runtime environment. Equation (10) ensures that QoS level δ_a between pods of an application *a* is not violated. Finally, (11) accounts the number of nodes utilized, including those nodes that are already utilized and those that were not utilized initially but will be utilized to deploy the applications.

Although the proposed model finds optimal solutions to the problem, some drawbacks related to scalability arise when trying to solve the problem and the number of nodes and/or applications grows. For this reason, we propose a heuristic algorithm as described in the next section.

V. Proposed Heuristic Algorithm

In this section, an iterative algorithm is proposed aimed at: *i*) finding near-optimal solutions in short times, and; *ii*) overcoming scalability issues derived from the proposed ILP model in realistic scenarios. Moreover, at the end of this section the set of scoring procedures the proposed algorithm is based onto are described.

A. Algorithm description

The main idea behind the proposed heuristic algorithm is to define scoring functions to rank: *i*) applications and *ii*) nodes. Then, a percentage of the highest ranked applications is sorted randomly in each iteration and the algorithm tries to allocate all application's pods to resources in nodes, in an ordered manner.

The pseudo-code for the proposed heuristic algorithm is described in Algorithm 1. Specifically, the algorithm receives as input parameters: the graph, *G*(*E*, *N*); the set of applications, *A*; the number of iterations to run the algorithm, *NUM_ITERS*; and tuning parameters α, Ψ, Λ, which are defined to set the percentage of applications to be sorted randomly, and the weight of first and second terms in the objective function as described in (1), respectively. The output, *best_sol*, is the best solution that the algorithm may find; however, it is worth noting that it is not guaranteed to be the optimal solution. Moreover, each node *n* includes normalized resources' information, assuming that this information can be pre-computed in advance and included in each node object. Notwithstanding, it may be not obvious that applications' resource requirements are normalized in advance.

The algorithm first initializes *best_sol* and the current iteration (lines 1-2) and, for each application *a*, the values of the resources required by the applications are normalized. The application's score is computed in procedure *computeAppScore*(·) (lines 3-6); this procedure is later described, in Section V.B. Then, for each iteration, the set N^* is initialized with the nodes in *N*, and the set A^* is initialized utilizing procedure *randomSort*(·), which randomly sorts a percentage of the highest ranked applications if iteration is greater than 1, and sorts the remaining ones by its score in descending order (lines 7-9). The percentage of applications to be randomly sorted is defined by the input parameter α, as previously introduced. Next, for each application, its pods are evaluated: nodes are sorted and their resources evaluated to deploy the application's pods (lines 10-27). If the node under evaluation can be utilized to deploy the pod, then resource assignment is updated as part of the solution and node's score is recomputed aimed and sorting it properly to evaluate resource assignment for the next pod (lines 15-20). If

ALGORITHM 1

```
IN:  G(E, N), A, NUM_ITERS, α, Ψ, Λ
OUT: best_sol
 1:  best_sol←∅
 2:  iter=1
 3:  for a in A
 4:    normalizeApp(a,N)
 5:    a.score=computeAppScore(a)
 6:  end for
 7:  while iter ≤ NUM_ITERS do
 8:    N*←N
 9:    A*←randomSort(A,'score',DESC,α,iter)
10:    for a in A*
11:      for p in P*
12:        N*←sort(N*, 'score', DESC)
13:        allocated=false
14:        for n in N*
15:          if canDeploy(G(E,N*),a,p,n) then
16:            assignResources(p, n)
17:            n.score=computeNodeScore(n)
18:            allocated=true
19:            break
20:          end if
21:        end for
22:        if not allocated then
23:          unassignResources(Pa,N*) //Remove from solution and recompute nodes' scores
24:          break
25:        end if
26:      end for
27:    end for
28:    sol←getSolAndCost(G(E,N*),A*,Ψ,Λ)
29:    best_sol←updateBestSolution(best_sol, sol)
30:    iter++
31:  end while
32:  return best_sol
```

a pod of an application cannot be deployed, all pods of that application are not placed and they are removed from the solution; i.e., that application is not deployed and no resources are assigned to its pods (lines 22-25). Next, the cost of the solution is computed and compared against that of the current best solution found, *best_sol*. In case that a better cost is found in the current iteration, then *best_sol* is updated with the new solution found (lines 28-29). Finally, the best solution found is returned (line 32).

B. Scoring procedures

As previously described, we define a set of procedures aimed at computing scores for applications and nodes to rank them.

In brief, the *computeAppScore*(·) procedure, line 5 in Algorithm 1, sets the higher scores to the applications that require a specific runtime environment, their pods require the more resources, and have less number of pods and do not require to be isolated. As a consequence of this criterion, the applications that require the highest number of pods and those applications that require to be deployed in isolation are given the lower scores. The *computeNodeScore*(·) procedure, line 21, assigns higher score to those nodes having more percentage of resources occupied and being larger. Fully loaded nodes are set to the lowest score, since those nodes cannot be utilized to deploy any other pod due to the lack of resources in them.

VI. ILLUSTRATIVE AND NUMERICAL RESULTS

In this section, we start describing some illustrative scenarios to analyze our proposals. Next, both the mathematical model and the algorithm are validated. Then, their performance is compared in terms of the solutions obtained and solving time. Finally, results obtained when solving the problem in large-scale scenarios are shown.

A. Scenarios

Aimed at validating the proposed mathematical model and the heuristic algorithm, four different settings based on the scenarios described in [10] have been defined. Specifically, we assume two different clusters, each with 30 nodes. Moreover, one cluster is defined having a set of *homogeneous* nodes; whereas the other one is defined to have a set of *heterogeneous* nodes. The summary of the nodes found in each cluster is as follows:

- Homogeneous cluster: this cluster consists of a set of 30 nodes, each node having 2-core CPU and 6 GB of memory.
- Heterogeneous cluster: the heterogeneous cluster consists of a set of 30 nodes; the resources of each node are randomly selected from the set of resource configurations formed by the tuples <1-core, 3 GB>, <2-core, 6 GB>, and <4-core, 12 GB>.

To provide variability on the runtime in the nodes and QoS guaranteed between nodes, the former is randomly selected from a set of 3 runtime environments; the latter is randomly selected from a set of 3 different levels of QoS; i.e., 0, 1 and 2, being 0 the most restrictive QoS level.

Regarding the applications requested to the clusters, we generate 10 different instances having a number of applications, each with different number of pods, and resulting in a total number of pods close to 100. Specifically, for each instance, we generate 32 applications and the number of pods of each application is randomly selected from 1 to 5. Each pod requires between 1/5 of a core (200 millicores) and 4/5 of a core (800 millicores) and 256 MB and 2.5 GB of memory, MEM, for applications to be deployed in the homogeneous cluster and between 1/5 of a core (200 millicores) and 1 core, and between 256 MB and 3 GB for applications to be deployed in the heterogeneous cluster. Therefore, on average, the resources required by the set of applications (in terms of CPU and memory) in these scenarios represent between 70% and 80% of the resources available in the cluster (homogeneous or heterogeneous) as shown in Fig. 3, definitely aligned with the average resource utilization shown in [10]. According to these values, CPU is slightly more required than memory resources.

In addition, we differentiate between two scenarios depending on the SLA parameters required and regarding the constraints those impose: *i*) a *relaxed* scenario in which only CPU and memory resource constraints are imposed, and *ii*) a *stringent* constraints scenario in which isolation, QoS level among pods, and runtime environment constraints can be required by the applications. The former corresponds to constraints described in (4) and (5); whereas the latter also includes constraints from (6) to (10). Specifically, to generate applications' requests requiring stringent constraints, we assume that a small percentage (5%) of the applications require to be deployed in isolation and that most of the applications (75%) require a specific runtime environment. In addition, the QoS level allowed between pods of the application is randomly selected between 3 available levels, being 0 the most restrictive and 2 the less restrictive. Therefore, applications requiring QoS level 2 among the pods forming the cluster can be deployed in nodes having 2, 1 or level 0 among them; however, requiring the most restrictive QoS level, 0, means that only nodes having level 0 among them can be selected to deploy the application.

In all the proposed experiments, the values of Ψ and Λ have been set to define as primary objective the minimization of applications rejected and as secondary objective to minimize the number of nodes utilized. Moreover, we set the number of iterations for the algorithm to 1, 5, 10 and 20, and the value of α is set to randomly sort the 50% of the highest ranked applications. It is worth noting that in this case, according to the applications' scoring function described, it can be expected that some of the applications requiring the larger number of pods and the applications requiring to be isolated will be kept the last to be evaluated, whereas the others will be randomly sorted to be evaluated.

The mathematical model and the algorithm were implemented in Python. In addition, to solve the ILP model we utilize CPLEX [34]. All experiments have been carried out in a 64-bits computer built upon 12 Intel core ES-2620 v2 at 2.1 GHz and 128 GB RAM machine.

B. Model and algorithm validation

Before comparing the mathematical model and the algorithm, we first need to validate them. To that end, we have generated several problem instances, considering homogeneous and heterogeneous clusters, as previously described and solved them using both the proposed ILP model and the algorithm.

Fig. 4 shows the results obtained in terms of applications deployed and nodes utilized. A red and a green line have been also depicted to easily show the number of applications to deploy (32 applications) and the number of nodes in the cluster (30 nodes), respectively. Fig. 4a and Fig. 4b represent the results for scenarios having relaxed constraints requirements; i.e., only CPU and memory resources requirements are taken into account. Fig. 4c and Fig. 4d represent the results for scenarios requiring stringent constraints; that is, scenarios in which applications can require, as part of their SLA parameters to guarantee the desired QoS: to be isolated, to be executed in a specific runtime environment, and to define the QoS among its pods.

As it can be seen in Fig. 4a and Fig. 4b, considering either the homogeneous cluster or the heterogeneous cluster, the optimal solutions found by the model allow all applications to be deployed only when CPU and memory resources constraints are imposed. Regarding the secondary objective, to minimize the number of nodes utilized, it can be seen that, on average, 23.7 nodes from the 30 nodes in the cluster are utilized when the homogeneous cluster is considered, while only 17.9 nodes are utilized, on average, when the heterogeneous cluster is considered, since the variety of nodes allows to consolidate the load in the nodes having more resources available.

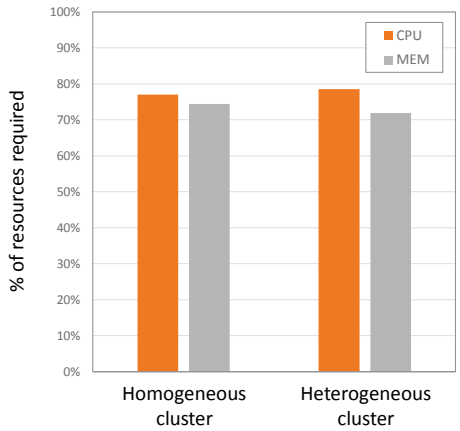

Fig. 3. Percentage of resources required in the homogeneous and heterogeneous clusters considered.

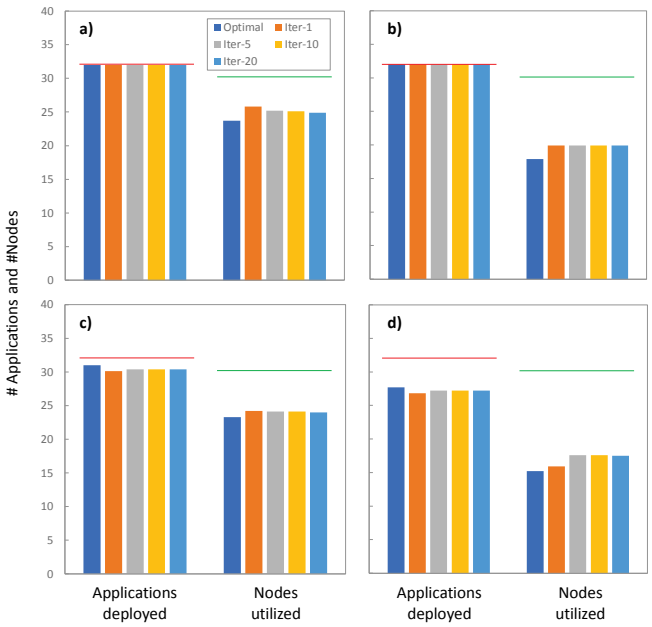

Fig. 4. Number of applications deployed and number of nodes utilized considering relaxed constraints in homogeneous (a) and heterogeneous (b) clusters and considering stringent constraints in homogeneous (c) and heterogeneous (d) clusters.

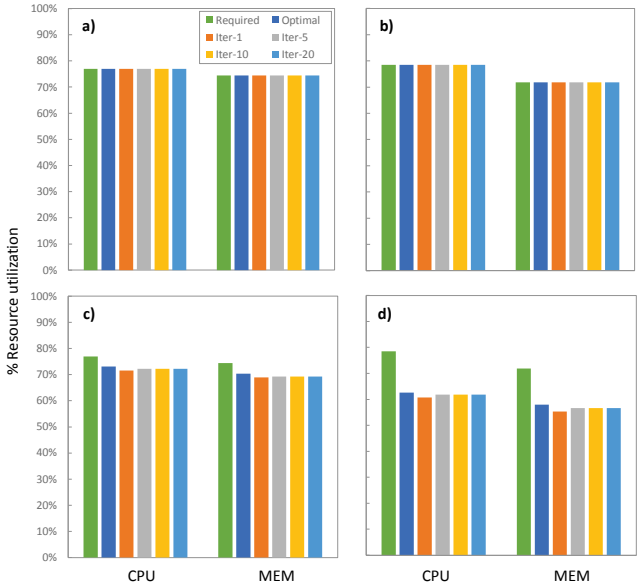

Fig. 5. Percentage of CPU and memory (MEM) resources required and occupied considering relaxed constraints in homogeneous (a) and heterogeneous (b) clusters and considering stringent constraints in homogeneous (c) and heterogeneous (d) clusters.

Interestingly, it can be observed that the algorithm performs the same in terms of number of applications deployed, since all applications can be deployed. Moreover, regarding the number of nodes utilized, it can be seen that the trend shown by the solutions obtained by the algorithm, is very similar to that of the solutions obtained by the model. However, the model performs slightly better than the algorithm, which utilizes between 24.9 and 25.8 nodes for 20 and 1 iterations, respectively, and the homogeneous cluster; and 19.9 nodes when considering the heterogeneous cluster and any number of iterations between 1 and 20.

Let us now focus on stringent constraints scenarios, in which, in addition to CPU and memory resources, applications can require stringent constraints imposed by the services they implement. Fig. 4c and Fig. 4d illustrate the results obtained in this case for homogeneous and heterogeneous clusters, respectively. As it is shown in Fig. 4c (homogeneous cluster), not all applications can be deployed due to the restrictive constraints, i.e., 1 application cannot be deployed. In fact, the impact of the stringent constraints is very clear on the results shown in Fig. 4d (heterogeneous cluster), where the number of applications deployed decreases to 27.7. However, despite the highly restrictive scenarios considered, it can be seen that the algorithm can find solutions that result in a number of applications deployed very close to that of the optimal solutions. Indeed, the number of applications deployed by the algorithm is 30.1 for 1 iteration and 30.4 for 5 to 20 iterations and homogeneous cluster (Fig. 4c) and 26.8 for 1 iteration and 27.2 for 5 to 20 iterations and heterogeneous cluster (Fig. 4d). The lowest number of nodes utilized is obtained by the model and it can be observed that, similar conclusions as for Fig. 4a and Fig. 4b are obtained. In these cases,

the optimal number of nodes utilized is 23.3 and 15.2 considering homogeneous and heterogeneous clusters, respectively; whereas the values obtained by the algorithm, when deploying the maximum number of applications, are 24 (homogeneous cluster) and 17.5 (heterogeneous cluster) in the best case.

For completeness, we also study the results obtained in terms of resources occupancy. Fig. 5 summarizes the percentage of resources required and resources utilized against the resources offered in the clusters. Fig. 5a and Fig. 5b show that all required resources are allocated to the applications, as expected; i.e., resource occupancy is the same as the resources required. Nevertheless, Fig. 5c and Fig. 5d illustrate the lack of resources in the network due to the stringent constraints considered. It is worth noting that even the optimal solutions cannot allocate all resources required by the applications. In addition, it can be seen that, in any case, optimal resource occupancy and resource occupancy resulting from the solutions obtained by the algorithm are very close.

C. Performance comparison

In view of the results shown in Section VI.B, and aimed at comparing performance of the proposed model and the algorithm, we focus on heterogeneous clusters and stringent constraints scenarios, which are the most restrictive as previously described. Moreover, to provide a complete performance comparison study we evaluate the same set of applications to deploy but in different cluster scenarios. Specifically, we extend our experiments to a set of heterogeneous clusters having different number of nodes, thus including scenarios having stringent constraints and lack of resources (similarly as described in the studies carried out in Section VI.B) and scenarios having stringent constraints but enough resources to deploy all the requested applications.

Fig. 6a illustrates normalized values for applications deployed against the number of nodes in the heterogeneous cluster. If we pay attention to the results obtained by the model (optimal solutions), for clusters having less than 45 nodes some applications are rejected. The reasoning behind this is that, in the clusters evaluated there are not enough resources to deploy all applications while satisfying their requirements in terms of: *i*) CPU and memory resources, *ii*) isolation, *iii*) runtime environment and *iv*) pod cluster's QoS level. Differently, clusters having 45 or more nodes, have enough resources to deploy all applications while guaranteeing all SLA parameters. A red dashed line has been depicted aimed at clearly differentiating these two regions. Interestingly, although in scenarios with lack of resources (30 to 45 node clusters) the algorithm performs slightly worse than the model in terms of applications deployed, it can be seen that the number of applications deployed gets closer to the optimal with the increment in the number of nodes (there are more resources available and the required SLA of applications can be more easily satisfied). Interestingly, as soon as there are enough resources to deploy all applications (45 node cluster), the algorithm deploys close to 99.7% of the applications in the worst case and all applications are deployed for 50 nodes or more in any case. It is worth highlighting that these results are aligned with the conclusions identified in Section VI.B, and the results shown for scenarios having relaxed constraints; i.e., only CPU and memory resources are required.

In addition, Fig. 6b represents the number of nodes utilized to deploy the applications against the number of nodes in the cluster. A red dashed line has been depicted to differentiate the regions in which there are not and there are enough resources. Although the algorithm finds solutions utilizing few additional nodes, it is clear that it behaves very similar to the model and, in any case, the number of nodes utilized is, undoubtedly, much lower than the number of nodes in the cluster.

Performance is also analyzed and compared in terms of time to solve the problem instances. Fig. 7 depicts the solving time, in logarithmic scale, against the number of nodes in the cluster. Although optimal solutions can be found in relative short times (from 50 to near 280 seconds) by the model, it is worth noting that the fastest configuration of the algorithm (1 iteration) finds solutions very close to the optimal ones in less than 41 ms; whereas the slowest configuration (20 iterations) takes less than 1 second. Interestingly, selecting 1 iteration of the algorithm opens the opportunity to implement the proposed algorithm in realistic scenarios to guarantee stringent constraints while obtaining near-optimal solutions, even in scenarios with lack of resources. Indeed, the authors in [10] describe a queue of 100 ms and fetching up to 100 containers and scenarios from [28] that need to process hundreds of application requests per second; which is aligned with the values in our experiments.

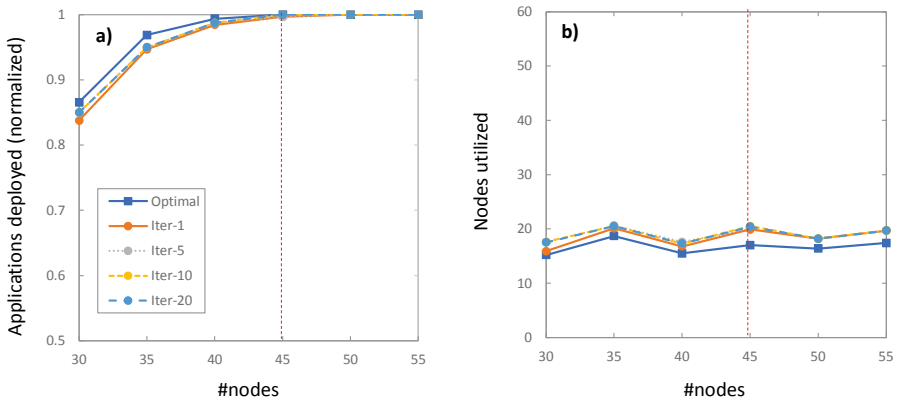

Fig. 6. Applications deployed (normalized) (a) and number of nodes utilized (b) against cluster size.

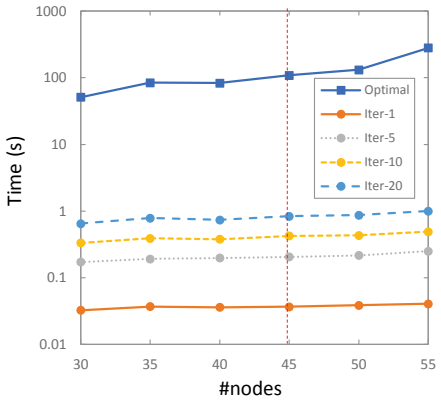

Fig. 7. Solving time against cluster size.

For completeness, aimed at deepening into the study of the secondary objective, we also analyze the results obtained considering a large enough cluster to serve all the applications in the queue (i.e., 50 node cluster) and having some of the nodes with certain level of resources utilized. Specifically, in these experiments, we set that 50% of the nodes (i.e., 25 nodes) host already deployed pods that represent about 1/5 of their CPU resources and 1/6 of their memory resources occupied; e.g., a <2-core, 6 GB> node will have 0.4 cores occupied and 1GB utilized. It is worth noting that, under this situation, there is a number of nodes that is already utilized and, therefore, when it is possible, those nodes will be preferred to deploy the pods from the new applications' requests. Interestingly, in these scenarios, constraints from (7) need to be taken into account and are activated aimed at avoiding placing pods that need to be isolated in nodes already hosting pods from different applications.

Fig. 8 illustrates the number of applications deployed, the total number of nodes utilized (which includes the number of nodes already utilized, 25, and the number of new nodes utilized) and the number of nodes utilized to deploy the new requested applications, in optimal and near-optimal solutions. A red line has been depicted to highlight the number of applications to deploy and a green line has been depicted at 25 nodes to remark the number of nodes hosting pods before deploying the new applications. As it can be seen in the figure, all applications can be deployed when solving the problem instances with the model. However, reducing the available resources and reducing the number of nodes without initial load, slightly impacts on the performance of the algorithm. Notwithstanding, this impact is very limited: more than 99.6% of applications are deployed.

Regarding the number of nodes utilized, it can be seen that optimal solutions utilize 30.4 nodes and that the new applications are deployed in 30.1; that is, the load from the new applications is distributed along the whole set of nodes utilized. Moreover, as shown in the figure, solutions found by the algorithm utilize 33 nodes in the best case and it concentrates the load of new applications in a lower number of nodes; however, in total, more nodes are utilized. Strategies based on local search [35] can be explored to minimize the number of nodes utilized. Notwithstanding, it is clear that the number of nodes utilized is much lower than the number of nodes in the cluster.

D. Large-scale clusters

Finally, aimed at evaluating the scalability of the algorithm to large-scale clusters, we generate 1,000 nodes and 5,000 nodes clusters with similar criteria as previously described for the heterogeneous cluster. Moreover, we generate a number of applications resulting in about 300 pods in addition to the previously described 100 pods scenarios.

In these scenarios, it is clear that the competition for resources is minimized with respect to the previous scenarios evaluated. Aligned with this idea, we found that all applications can be deployed. However, it is interesting to study the solving time. Fig. 9 illustrates the solving time taken by the algorithm against the number of iterations and different number of pods and the 1,000 nodes cluster (Fig. 9a) and 5,000 nodes cluster (Fig. 9b) mentioned above. As it can be seen, time increases with the number of pods (applications) and with the number the nodes in the cluster. Obviously, increasing the number of iterations of the algorithm has a remarkable impact on the solving time.

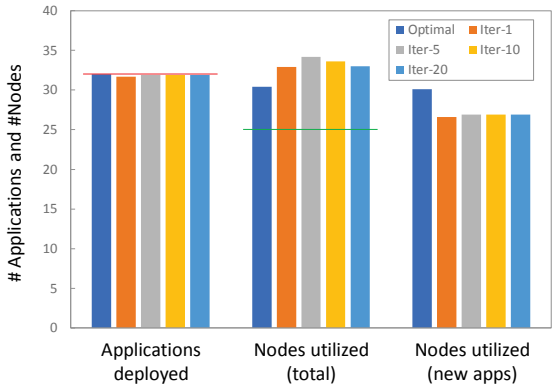

Fig. 8. Number of applications deployed and number of nodes utilized, in total and to deploy the requested applications.

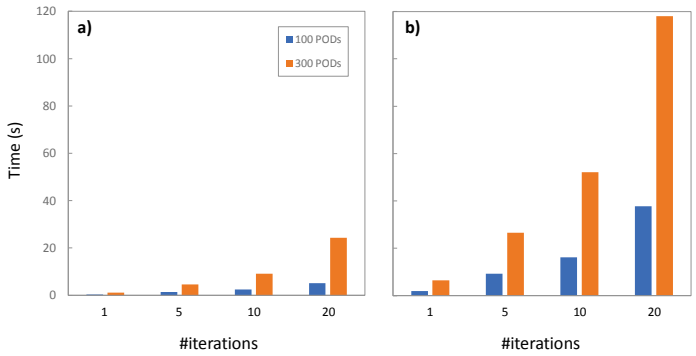

Fig. 9. Solving time against number of iterations in 1,000 nodes cluster (a) and 5,000 nodes cluster (b).

However, in such scenarios where there is few or no competition for resources, the benefits of executing the algorithm more than 1 time become not relevant, since with 1 iteration good enough solutions can be found; thus benefiting from the time reduction to solve the problem in large-scale clusters. Specifically, for a 1,000 nodes cluster, solving time is below 0.32 s and below 1.14 s for 100 and 300 pods, respectively; whereas for 5,000 nodes clusters, times are below 2 s and 6.45 s; which are promising results that demonstrate that is worth to deepen into exploring the algorithm, even though in this analysis we focus only on the time to execute the algorithm.

VII. Future Work

In view of the results shown in the previous section, the authors believe that the proposed approach is worth to continue to be explored because: *i*) the solutions obtained by the algorithm in the different scenarios evaluated are close to the optimal and *ii*) the time to obtain those solutions is very short, even in large-scale scenarios. However, three main lines worth to explore are identified:

- *Experimental validation*. In this paper we compare the performance of the proposed algorithm against that of the ILP model. Although state-of-the-art algorithms and novel algorithm proposals reviewed in Section II do not consider all constraints in our problem, it would be of high interest to implement our proposal in a real testbed and compare results obtained against them.
- *Algorithm improvement*. To improve the solutions found, specifically, in terms of nodes utilization (secondary objective), an additional stage in the algorithm focused on local search may be included (e.g., Greedy Randomized Adaptive Search Procedures [35] may be worth considering). Furthermore, by running each iteration in parallel one may think that better results could be obtained in the shortest time. This is also an improvement that can be considered.
- *Model and algorithm extension*. Both the model and the algorithm can be extended aimed at considering a larger set of resources, *r* resources, instead of focusing only on CPU and memory. Moreover, different objective functions can be designed aimed at focusing on objectives different than consolidation; e.g., for load balancing purposes.

We consider these three main improvements as the basis for our future work in container scheduling research.

VIII. Conclusions

Since the advent of cloud computing, VM placement and task consolidation have been topics of interest for both the industry and the research community. However, in the recent years, container-based infrastructures have become more relevant in front of VM-based infrastructures and it is currently attracting the interest of the research community. Indeed, to deploy applications, containers can be placed either in physical or VM.

This new paradigm, jointly with novel architectures considering not only cloud resources but resource continuum from the edge up to the cloud, and the services that those architectures can support (mainly related to IoT and smart environments), reveal the need for new scheduling algorithms to optimize container placement. Traditional schedulers based on container-by-container scheduling may fall into finding solutions far from the optimal ones, even in blocking requests, since the order in which the containers are placed may impact negatively on the next placement decisions. Differently, concurrent container scheduling allows to find solutions for sets of containers, jointly, thus increasing the opportunities to place more containers or obtaining more efficient solutions.

In this paper, we proposed an ILP model to formally state the Concurrent Container Clusters Scheduling (C3S) problem and a heuristic algorithm that finds solutions close to the optimal ones while it outperforms the mathematical model in terms of time to obtain the solutions. Specifically, the time taken is below 41 ms considering a reference scenario of 30 heterogeneous nodes and about 100 pods and below 1.2 and 6.5 seconds considering 300 concurrent pods and large-scale clusters, having 1,000 and 5,000 nodes, respectively.

The proposed model can be utilized to validate new container scheduling algorithms in the described scenarios and the proposed heuristic algorithm can be considered as an alternative to current state-of-the-art algorithms, since it includes more constraints and it is oriented to minimize the applications rejected, which may become of paramount importance under certain novel scenarios considering the resource continuum from the edge up to the cloud.

ACKNOWLEDGMENT

This work has been supported by the Spanish Ministry of Science, Innovation and Universities and by the European Regional Development Fund (FEDER) under contract RTI2018-094532-B-I00.

REFERENCES

[1] A. Botta et al., "On the Integration of Cloud Computing and Internet of Things," International Conference on Future Internet of Things and Cloud (FiCloud), August 2014, doi 10.1109/FiCloud.2014.14

[2] F. Bonomi et al., "Fog computing and its role in the Internet of Things," in: Proceedings of MCC Workshop on Mobile Cloud Computing, pp. 13–16, 2012.

[3] Y. C. Hu et al., "Mobile Edge Computing: A key technology towards 5G". ETSI white paper, September 2015. ISBN: 979-10-92620-08-5

[4] J. Garcia et al., "Do we really need cloud? Estimating the fog computing capacities in the city of Barcelona". In 11th IEEE/ACM International Conference on Utility and Cloud Computing (UCC), Zurich, Switzerland, November 2018, doi: 10.1109/UCC-Companion.2018.00070

[5] C. Nguyen et al., "Why cloud applications are not ready for the edge (yet)". In Proceedings of the 4th ACM/IEEE Symposium on Edge Computing (SEC), Arlington, VA, November 2019, doi: 10.1145/3318216.3363298

[6] W.Ramírez el al., "Evaluating the Benefits of Combined and Continuous Fog-to-Cloud Architectures", Computer Communications, vol.113, pp.43-52, November 2017

[7] Amazon Elastic Cloud Service (ECS), https://aws.amazon.com/ecs/ [Accessed: Sept. 2020]

[8] Google Kubernetes Engine (GKE), https://cloud.google.com/kubernetes-engine [Accessed: Sept. 2020]

[9] Microsoft Azure Container Services, https://azure.microsoft.com/en-us/product-categories/containers/ [Accessed: Sept. 2020]

[10] Y. Hu et al., "Concurrent container scheduling on heterogeneous clusters with multi-resource constraints," Future Gener. Comput. Syst., vol. 102, pp. 562–573, 2020, doi: 10.1016/j.future.2019.08.025

[11] Google Kubernetes, https://kubernetes.io/ [Accessed: Sept. 2020]

[12] X. Masip-Bruin et al., "Foggy Clouds and Cloudy Fogs: a Real Need for Coordinated Management of Fog-to-Cloud (F2C) Computing Systems", IEEE Wireless Communications Magazine, vol. 23, no. 5, pp. 120-128, 2016, doi: 10.1109/MWC.2016.7721750

[13] A. Asensio, et al., "Designing an Efficient Clustering Strategy for Combined Fog-to-Cloud Scenarios," Future Gener. Comput. Syst., vol. 109, pp. 392–406, 2020, doi: 10.1016/j.future.2020.03.056

[14] mF2C project, http://www.mf2c-project.eu, [Accessed: Sept. 2020]

[15] Docker Swarm, https://docs.docker.com/engine/swarm/ [Accessed: Sept. 2020]

[16] K. A. Nuaimi et al., "A survey of load balancing in cloud computing: challenges and algorithms," 2012 Second Symposium on Network Cloud Computing and Applications, London, pp. 137-142, 2012, doi: 10.1109/NCCA.2012.29

[17] Y.C. Lee and A.Y. Zomaya, "Energy efficient utilization of resources in cloud computing systems," J. Supercomput. vol. 60, pp. 268–280, 2012, doi: 10.1007/s11227-010-0421-3

[18] B. Liu et al., "A new container scheduling algorithm based on multi-objective optimization," Soft. Comput. vol. 22, pp. 7741–7752, 2018, doi: 10.1007/s00500-018-3403-7

[19] Lianwan Li, Jianxin Chen, and Wuyang Yan, "A particle swarm optimization-based container scheduling algorithm of docker platform," in Proceedings of the 4th International Conference on Communication and Information Processing (ICCIP '18). Association for Computing Machinery, New York, NY, USA, pp. 12–17, 2018, doi: 10.1145/3290420.3290432

[20] S. Rampersaud and D. Grosu, "Sharing-aware online virtual machine packing in heterogeneous resource clouds," IEEE Trans. Parallel Distrib. Syst. vol. 28, no. 7, pp. 2046–2059, 2017, doi: 10.1109/TPDS.2016.2641937

[21] M. Gabay and S. Zaourar, "Vector bin packing with heterogeneous bins: application to the machine reassignment problem," Ann. Oper. Res. vol. 242, pp. 161–194, 2016

[22] D. Bernstein, "Containers and Cloud: From LXC to Docker to Kubernetes," in IEEE Cloud Computing, vol. 1, no. 3, pp. 81-84, 2014, doi: 10.1109/MCC.2014.51

[23] T. Rausch, A. Rashed, and S. Dustdar, "Optimized container scheduling for data-intensive serverless edge computing," Future Gener. Comput. Syst., vol. 114, pp. 259–271, 2021. doi:10.1016/j.future.2020.07.017. (Available online August 2020)

[24] J. López, N. Kushik, and D. Zeghlache, "Virtual machine placement quality estimation in cloud infrastructures using integer linear programming," Software Qual. J., vol. 27, pp. 731–755, 2019, doi: 10.1007/s11219-018-9420-z

[25] Tayebeh Bahreini and D. Grosu. "Efficient placement of multi-component applications in edge computing systems". In Proceedings of the 2nd ACM/IEEE Symposium on Edge Computing (SEC), San Jose, CA, October 2017, doi: 10.1145/3132211.3134454

[26] M. Stillwell et al., "Resource allocation algorithms for virtualized service hosting platforms," J. Parallel Distrib. Comput., vol. 70, no. 9 pp. 962–974, 2010, doi: 10.1016/j.jpdc.2010.05.006

[27] M. Armbrust et al., "A view of cloud computing," Commun. ACM vol. 53, no. 4, pp. 50–58, 2010, doi: 10.1145/1721654.1721672.

[28] C. Reiss et al., "Heterogeneity and dynamicity of clouds at scale: Google trace analysis," in Proceedings of the Third ACM Symposium on Cloud Computing, ACM, no. 7, pp.1-13, 2012, doi: 10.1145/2391229.2391236

[29] Linux Containers, https://linuxcontainers.org [Accessed: Sept. 2020]

[30] T. Coughlin, "Convergence through the Cloud-to-Thing Consortium," IEEE Consum. Electron. Mag., vol. 6, no. 3, pp. 14–17, 2017, doi: 10.1109/MCE. 2017.2684914

[31] IEEE 1934-2018 - IEEE Standard for Adoption of OpenFog Reference Architecture for Fog Computing, https://standards.ieee.org/standard/1934-2018.html, 2018 [Accessed: Sept. 2020]

[32] H. Gupta et al., "SDFog: A Software Defined Computing Architecture for QoS Aware Service Orchestration over Edge Devices", https://arxiv.org/pdf/1609.01190.pdf [Accessed: Sept. 2020]

[33] J. Petazzoni, "Linux Containers (LXC), Docker, and Security," 31 Jan. 2014; https://www.slideshare.net/jpetazzo/linux-containers-lxc-docker-and-security [Accessed: Sept. 2020]

[34] IBM CPLEX Optimizer, https://www.ibm.com/analytics/cplex-optimizer [Accessed: Sept. 2020]

[35] Thomas A. Feo and Mauricio G. C. Resende, "Greedy Randomized Adaptive Search Procedures,". Journal of Global Optimization, vol. 6, no. 2, pp. 109–133, 1995, doi:10.1007/BF01096763