A Resource Allocation Framework with Qualitative and Quantitative SLA Classes

Tarek Menouer[1(✉)], Christophe Cérin[1], Walid Saad[1,2], and Xuanhua Shi[3]

[1] Université Paris 13, Sorbonne Paris Cité, Paris, France
{tarek.menouer,christophe.cerin,walid.saad}@lipn.univ-paris13.fr
[2] ENSIT, LATICE Laboratory, University of Tunis, Tunis, Tunisia
[3] Huazhong University of Science and Technology, Wuhan, China
xhshi@hust.edu.cn

Abstract. This paper presents a new resource allocation framework based on SLA (Service Level Agreements) classes for cloud computing environments. Our framework is proposed in the context of containers with two qualitative and two quantitative SLAs classes to meet the needs of users. The two qualitative classes represent the satisfaction time criterion, and the reputation criterion. Moreover, the two quantitative classes represent the criterion over the number of resources that must be allocated to execute a container and the redundancy (number of replicas) criterion. The novelty of our work is based on the possibility to adapt, dynamically, the scheduling and the resources allocation of containers according to the different qualitative and quantitative SLA classes and the activities peaks of the nodes in the cloud. This dynamic adaptation allows our framework a flexibility for efficient global scheduling of all submitted containers and for efficient management, on the fly, of the resources allocation. The key idea is to make the specification on resources demand less rigid and to ask the system to decide on the precise number of resources to allocate to a container. Our framework is implemented in C++ and it is evaluated using Docker containers inside the Grid'5000 testbed. Experimental results show that our framework gives expected results for our scenario and provides with good performance regarding the balance between objectives.

Keywords: Scheduling and resource management
Optimization · Performance measurement and modelling
New economic model · Cloud computing
Containers to support high performance computing
and industrial workloads

1 Introduction

Nowadays, different forms of cloud computational resources exist such as virtual machines (VMs), containers, or bare-metal resources, having each their own characteristics. Container technology is relatively new in production systems

G. Mencagli et al. (Eds.): Euro-Par 2018 Workshops, LNCS 11339, pp. 69–81, 2019.
https://doi.org/10.1007/978-3-030-10549-5_6

but it is not a new concept. Container is a light-weight OS-level virtualization technique that allows to run an application and its dependencies in a resource-isolated process.

This paper presents a new opportunistic scheduling and resource allocation system based on an economic model related to different classes for SLAs (Service Level Agreements). The objective is to address the problems of companies that manage a private infrastructure of machines i.e. a cloud platform, and would like to optimize the scheduling of several containers submitted online by users. Each container is executed using a set of computing resources.

To specify the user desired SLAs classes, we propose to modelled each class by 3 services. This choice is motivated by our experience with the Fonds Unique Interministériel (FUI) Wolphin project [6]. It is based on the observation that hosting solutions do not allow manufacturers or cloud providers to offer to their users a fair or accurate invoice, i.e. a precise invoice with respect to the waiting time, the nodes reputation, consumption of resources and the number of replicas. The AlterWay company, coordinator of the Wolphin project, noticed that the project must respond to the following usages with regard to the deployed services: (i) Premium service is designed to users who want to get a 'high quality' service; (ii) Advanced service is designed to users who want to get an 'average quality' service; and (iii) Best effort service is designed to users who want to get a 'low/less quality' service.

In this work, we decompose the scheduling and allocation problems into 4 steps, namely selection of a container in a queue, selection of candidate nodes, computation of resources and allocation on a node. One can view the scheduler has a program that repeats forever these 4 steps. The third step is new, compared to the existing state-of-art research because, to the best of our knowledge, none of the existing cloud scheduler computes, dynamically, the number of resources allocated to a container. This is the first contribution of the paper. The user do not request for a fixed number of resources. The second contribution is related to the new economic model sustained by the 4 SLA classes regrouped in 2 qualitative and 2 quantitative SLA classes. The third contribution of the paper is the experiments that we conduct with Docker containers. We have implemented a new scheduler, based on the Docker API, for the creation of containers and we execute traces representative from the High Performance Computing (HPC) world and traces representative of Web hosting companies.

The organization of the paper is as follows. Section 2 presents some related works. Section 3 describes our framework architecture. Section 4 presents our qualitative and quantitative SLA classes. Section 5 describes how the SLA classes are used by our framework. Section 6 introduces exhaustive emulation that allows the validation of our proposed framework. A last, a conclusion and some future works are given in Sect. 7.

2 Related Work

In the literature, all problems of resources allocation or resources management refer to the same class of scheduling problems. They consist generally in associ-

ating a user's request to one or several computing cores. Most of these problems are *NP*-hard [13].

In the context of containers scheduling on cloud computing, there exists several studies, as those presented in [10,14,16]. However, to the best of our knowledge, all frameworks schedule containers according to a fixed configuration in term of computing resources. From an industrial point of view, we may cite, as examples, the schedulers inside Google Kubernetes [16], Docker Swarm [10] and Apache Mesos [14].

Google Kubernetes [16] is a scheduler framework which represents an orchestration system for containers based on pods concept. Pods are a group of one or more containers. They are always co-located and co-scheduled and run in a shared context. Moreover, they will be run on the same physical or virtual machine. The principle of Kubernetes scheduling can be summarized in two steps. First, filter all machines to remove machines that do not meet certain requirements of the pod. Second, classify the remaining machines using priorities to find the best fit to execute a pod.

Docker Swarm [10] is an important container scheduler framework developed by Docker. Docker is the technology used by the FUI Wolphin project [6] which is the support of our work. The Swarm manager is responsible for scheduling the containers on the agents or nodes. Swarm also has two steps to finally selecting the node that will execute the container. First, it uses filters to select suitable nodes to execute the container. Then, it uses, according to a ranking strategy, the most suitable one. Actually, Swarm has three ranking strategies: (i) Spread strategy which executes a container on the node having the least number of containers, (ii) Bin packing strategy, in contrast with spread, chooses the node with the most packed containers on it, and (iii) Random strategy which chooses a node randomly.

The field of Virtual Machines (VMs) scheduling may also serve as a reference for containers scheduling. Various approximation approaches are applied in the work of Tang et al. [12]. Authors propose an algorithm that can produce high-quality solutions for hard placement problems with thousands of machines and thousands of VMs within 30 seconds. This approximation algorithm strives to maximize the total satisfied application demand, to minimize the number of application starts and stops, and to balance the load across machines.

Targeting the energy efficiency and SLA compliance, Borgetto et al. [2] present an integrated management framework for governing Cloud Computing infrastructures based on three management actions, namely, VM migration and reconfiguration, and power management on physical machines. By incorporating an autonomic management loop, optimized using a wide variety of heuristics ranging from rules over random methods, the authors demonstrated that the proposed approach can save energy up to 61.6% while keeping SLA violations acceptably low.

In contrast to these related and above-mentioned studies, our proposed framework combines scheduling and allocation strategies with qualitative and quantitative SLA classes. The SLA classes are proposed to answer the needs of

different users. The benefit of our framework consists to use the different SLA classes to: (i) select the first container that must be executed; (ii) decide the cloud node that must execute the selected container; (iii) compute dynamically the number of resources allocated to the considered container and (iv) decide the number of replicas for a container and choose nodes which execute the considered container and it's redundancy replicas.

A preliminary work has been published in [9]. In this paper, we consider the following improvements: (i) we consider 4 SLA classes instead of 2 SLA classes to have an economic model with several classes; (ii) the general scheduling schema is composed of 4 steps instead of 3 to satisfied all the SLA classes; and (iii) experiments are emulation on Grid'5000 testbed, with Docker containers, instead of simulations. In other words, this work introduces a more general and realistic framework compared to [9].

3 Architecture

The goal of our framework is to give answers to the problem stated as follows: *in cloud computing environment, how to use a set of qualitative and quantitative SLA classes to optimize the global scheduling of containers submitted online by users?*

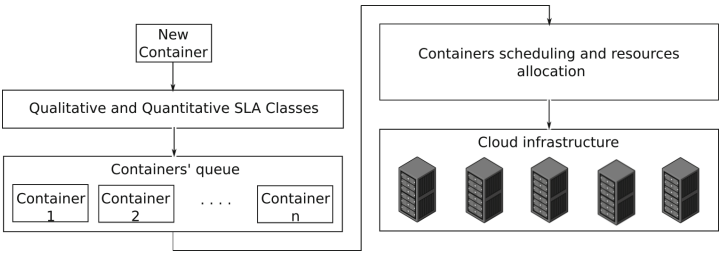

Fig. 1. Framework's architecture

Figure 1 depicts the architecture of our framework. Each time a new container is submitted online, the user must firstly select its services in the qualitative and quantitative SLA classes. Then, the new submitted container is inserted in the containers' queue. After that, our framework, schedules and allocates resources to each container according to its configuration in term of SLA classes. Finally, the submitted container is executed in the most appropriate cloud node.

4 Qualitative and Quantitative SLA Classes

As said before, our framework is based on SLA classes to configure each new submitted container. Our SLA classes are regrouped in two qualitative and two

quantitative classes. Each class is proposed with 3 services: Premium, Advanced and Best effort.

The two qualitative classes address: (i) the satisfaction time criterion, that means, the user waiting time before the execution of the user container; and (ii) the reputation criterion, that means the node choice of the user to execute his container. Moreover, the two quantitative classes address: (i) the number of resources criterion, that means the number of resources must be allocated to execute a container; and (ii) the redundancy criterion which set the number of time that a container is executed to ensure fault tolerance.

In our context, to satisfy the user needs according to its SLAs classes, we propose to represent each service by one priority value as following: (i) Premium service: priority value = 3; (ii) Advanced service: priority value = 2; and (iii) Best effort service: priority value = 1. As we have 4 SLA classes (2 qualitative and 2 quantitative), each container is represented by 4 priorities values, each value represent the assignment of the service in one SLA class. For the first qualitative SLA class (satisfaction time), the modeling of our 3 services is motivated by the fact that users are regrouped in 3 categories:

- Premium service: It is designed for users who wish to find a solution as soon as possible without considering the price of the operation,
- Advanced service: It is designed for users that have a limited financial budget but still wish to have a solution in the smallest reasonable execution time,
- Best effort service: It is designed to users who have no time constraints, but want to pay for the minimum possible price.

For the second qualitative SLA class (reputation), our modeling based on 3 services is motivated by the fact that nodes are different according to the cloud infrastructure. Generally the differences between nodes is based on reputation criterion as: (i) security of sites; (ii) reliability of hardware; and (iii) reliability of network. In this context, users are also regrouped in 3 categories:

- Premium service: It is designed for users which execute their containers in nodes with high reputation;
- Advanced service: It is designed for users which execute their containers in nodes with an average reputation;
- Best effort service: It is designed to users who have no constraints about the reputation of the cloud nodes. The goal is to has a low cost price.

For the first quantitative SLA class (number of resources), the modeling of our 3 services is motivated by the fact that the need of resources for each user is different. Generally users are regrouped in 3 categories:

- Premium service: It is designed to users with long service that needs many computing cores.
- Advanced service: It is designed to users with short service that need some computing cores.
- Best effort service: It is designed to users with micro service that do not need many computing cores. Its service life is less than the frequency of metrics' collection.

For the second quantitative SLA class (redundancy replicas), the modeling of our 3 services is motivated by the fact that users are regrouped also in 3 categories according of the number of redundancy replicas of containers:

- Premium service: It is designed for users who execute their containers with a big number of redundancy replicas in different nodes to be sure that at the end of the execution, they get a solution;
- Advanced service: It is designed for users who execute their containers with average number of redundancy replicas in different nodes;
- Best effort service: It is designed to users who execute their containers without constraint about the number of replicas.

5 Scheduling and Resources Allocation Based on SLA Classes

As many other scheduling system proposed in the literature, we sketch to use a containers' queue to store all submitted containers. To schedule and allocate resources to containers, our framework goes through four phases according to the qualitative and quantitative SLA classes:

1. Container scheduling: It is based on a combination between qualitative and quantitative classes. To select the first container which must be executed we propose to use the PROMETHEE II (*Preference Ranking Organization METHod for Enrichment Evaluations*) algorithm;
2. Container reputation: It is based on the qualitative reputation class, to select a set of nodes that can execute the container and its redundancy replicas;
3. Container allocation: It is based on the quantitative number of resources class, to set dynamically the number of resources must be allocated to the selected container;
4. Container 'redundancy replicas': It is based on the quantitative 'redundancy replicas' class, to set the number of replicas for a container. This phase is also used to assign a container and its replicas to cloud nodes using the bin packing heuristic.

5.1 Container Scheduling

To select the first container which must be executed we propose to use, in this paper and for convenience, the PROMETHEE II algorithm [11] because it is a multi-criteria decision algorithm. It is also possible to use for example a CPLEX solver [4] in order to solve the decision problem, or any other techniques. In our context, if the selected container (c_x) can not be executed because of a lack of resources for example, the container c_x wait in the container's queue and a new container is selected by the PROMETHEE II algorithm. Remind that PROMETHEE II is an algorithm which permits the building of an outranking between different alternatives [11]. It is used in this step because it is known to provide with a 'good' compromise between qualitative and quantitative criteria

and it is mathematically well founded. Indeed, the PROMETHEE II has been used with success to solve many problems [1]. It is based on a comparison, pair by pair, of possible decisions (containers) along the qualitative criteria (satisfaction time and reputation) and the quantitative criteria (number of resources and redundancy replicas). More details about the use of PROMETHEE II algorithm in our context is presented in [9]. PROMETHEE II algorithm has a complexity of $\mathcal{O}(q.n\log(n))$ [3] (where q represents the number of criteria and n the number of possible decisions (alternatives)).

5.2 Container Reputation

This step is used by our framework to select nodes that must execute a container and its copies according to the container service. Indeed, our framework classifies statically all nodes which form the cloud infrastructure in 3 categories: (i) High reputation nodes; (ii) Average reputation nodes; and (iii) Low reputation nodes.

Then, each service in the qualitative reputation class uses nodes category, as following: (i) Premium service uses the high reputation nodes category; (ii) Advanced service uses the average reputation nodes category; and (iii) Best effort service uses the low reputation nodes category.

5.3 Container Allocation

Our framework uses the quantitative number of resources class to set, for each container, the number of resources. In this step, we propose to use the same idea as the introductory work presented previously in [9], which is applied for any kind of nodes in the cloud (heterogeneous and not heterogeneous nodes). The principle is to set, for each container, a range on resources demand instead of specifying a fixed quantity of resources. It means that each service in the quantitative number of resources class has a number of resources bounded between the min and max parameters. The bound of cores for each service is proposed to be sure that each container, with low service in the number of resources class, cannot be executed with more cores than a container with a high service in the number of resources class.

To compute the bound of cores, we propose first to set N as the number of resources of the smallest machine of the infrastructure. N is set in this way to be sure that in any situation, the container is executed on one cloud node. After setting N, each service in the SLA quantitative number of resources class calculates the min and max number of resources. As we have 3 services, we propose to manage 3 intervals with the same distance as follows:

- *Best effort* class : Min number of resources = 1; and Max number of resources = $\frac{1}{3} \times N$,
- *Advanced* class : Min number of resources = Max number of resources of the Best effort service + 1; and Max number of resources = $\frac{2}{3} \times N$,
- *Premium* class : Min number of resources = Max number of resources of the Advanced service + 1; and Max number of resources = N.

After bounding the number of resources for each service in the quantitative number of resources class, we use a function that set, dynamically, the number of resources for a container (c_x) at time t using: (i) bounds cores (min and max cores) in each service; (ii) number of containers saved in the containers' queue at time t with the same reputation service as the container c_x; and (iii) number of free cores available in all the candidates nodes that can execute the container c_x at time t.

Let r_i be the number of resources that must be allocated to container c_i with quantitative number of resources priority p_i. We suppose that the containers' queue has n containers $(c_1, c_2, \cdots, c_n)$ with the same reputation service as c_i. Lest set $(p_1, p_2, \cdots, p_n)$ the quantitative number of resources priorities associated to the previous n containers saved in the queue and wc the number of waiting cores in all candidates nodes that can execute c_i (with the same reputation service as c_i). Then r_i is computed as presented in the formula 1.

$$r_i = \frac{p_i * wc}{\sum_{j=0}^{n} p_j} \tag{1}$$

The formula 1 computes, at each time t, a fair partitioning of all waiting cores between containers according to their quantitative number of resources services. Next, the system checks if $r_i >$ Max cores (Max_{cores}) of its quantitative number of resources service, then $r_i = Max_{cores}$, else if $r_i <$ (Min_{cores}) of its quantitative number of resources service, $r_i = Min_{cores}$.

For example, let us consider an infrastructure composed of 3 nodes with the Premium service in the reputation class, and 9 waiting cores in $node_1$, 6 waiting cores in $node_2$ and 6 waiting cores in $node_3$. The total number of waiting cores is 21. Let us also use the following three containers which have Premium service in the reputation class and have the following configuration:

- Container c_1: *Premium* service on the quantitative number of resources class (priority = 3), Min cores = 7 and Max cores = 9;
- Container c_2: *Advanced* service on the quantitative number of resources class (priority = 2), Min cores = 4 and Max cores = 6;
- Container c_3: *Best effort* service on the quantitative number of resources class (priority = 1), Min cores = 1 and Max cores = 3.

The number of resources is set as following:

- Container $c_1 : r_1 = \frac{3*21}{3+2+1} = 10$. As $10 > 9$ (max cores for *Premium* service), we set $r_1 = 9$. Then, the number of waiting cores will be equal to (9-9)+6+6 = 12. Now, in the queue, only 2 containers are saved: c_2 and c_3.
- Container $c_2 : r_2 = \frac{2*12}{2+1} = 8$. As $8 > 6$ (max cores for *Advanced* service), we set $r_2 = 6$. Then, the number of waiting cores will be equal to 0+(6-6)+6=6. Now, in the queue there is only the container c_3.
- Container $c_3 : r_3 = \frac{1*6}{1} = 6$.

5.4 Container Replicas

In our framework each container and its replicas are executed in different nodes of the same reputation category nodes. That means, we cannot execute a container and it's redundancy replicas in the same cloud node. To compute the number of redundancy replicas for each container, our framework sets an empirical value for each service in the quantitative redundancy replicas class. The unique constraint is that the highest service has the biggest value for the number of replicas. For example, we may have the following setting:

- Premium service, our framework sets 3 redundancy replicas for each container;
- Advanced service, our framework sets 2 redundancy replicas for each container;
- Best effort service, our framework sets 1 redundancy replica for each container.

We propose to add in our framework the redundancy replicas class to manage some fault tolerance issues. For example, if one cloud node ($node_x$) is stopped for different reasons, all containers who are executing on $node_x$ are also stopped. In this case, if the user chooses a high service in the redundancy replicas class, he will be granted that another copy of his container is running in another node. In reality, the usual practical assumption is that there is very low likelihood that all nodes will stop at the same time.

In our system, we guarantee that each container or its redundancy replicas are executed in fifferent nodes. To assign a container to a cloud node, our framework applies the well known bin packing principle which is a combinatorial *NP-hard* problem [5]. The principle of the bin packing heuristic consists, for each new container c_i, to assign it to the node n_j which has the less available free resources. This means that we select the node (not yet visited) that has the smallest number of idle cores and that can execute the container c_i.

The goal of using this heuristic is to minimize the number of active nodes to reduce the cost of exploiting the infrastructure.

5.5 Complexity Analysis

Based on above mentioned arguments, the overall time complexity of our approach is the complexity of the 4 steps: $\mathcal{O}(q.n\log(n))$, NP-hard problem, $\mathcal{O}(n)$ and $\mathcal{O}(n)$ respectively and for n being the node number of the architecture.

6 Experimental Evaluation

In this section, we introduce emulation result of our framework to check if it meets our expectations. For the emulation, we have used the Docker container technology inside the Grid5000 platform [7], an experimental large-scale testbed for distributed computing in France. For our experimental evaluation, we reserved an infrastructure composed of 480 computing cores distributed in 15 nodes (Intel Xeon CPU). The 15 nodes are split as following: (i) 5 nodes form

high reputation category; (ii) 5 nodes form average reputation category; and (iii) 5 nodes form low reputation category.

In this experimental evaluation, each container is submitted by one of the following three users, each user has a particular services in the SLA classes:

- Premium user: Premium service for all qualitative and quantitative SLA classes;
- Advanced user: Advanced service for all qualitative and quantitative SLA classes;
- Best effort user: Best effort service for all qualitative and quantitative SLA classes.

Each container runs a unique simple parallel application which load computing cores. The number of cores occupied by each container is set automatically by our framework as presented in Subsect. 5.3. However, each container has also a Sequential Life Time (SLT) set when the container is submitted and it is equal to 5 min. Then, according to the number of cores allocated for each container (N), the Parallel Life Time (PLT) which represent the real executing time of the container is computed as being $PLT = \frac{SLT}{N}$.

Moreover, in this series of emulation, we introduce the performance of our framework according to the submitting containers type. In this context, we propose two types of experiments: (i) containers submitted at the same time; and (ii) containers submitted online. The first one stresses the behavior of our framework. The second one represents a "normal" operating mode.

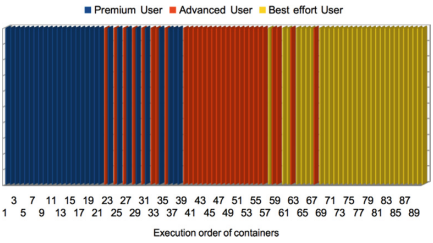

Fig. 2. Submission of 90 Docker containers at the same time

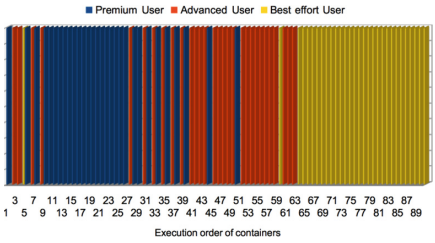

Fig. 3. Submission of 90 Docker container online with a fixed frequency

6.1 Containers Submitted at the Same Time

Figure 2 shows the order of execution of 90 containers submitted at the same time by 3 users, each user submits 30 containers. As a result, it is clear that our framework starts by the execution, firstly, of containers submitted by the Premium user, then containers submitted by the Advanced user. Finally, our framework executes containers submitted by the Best effort user. This result confirms that our framework respects the priorities of containers. We note also

that when our framework cannot execute a container which has a high service priority, as the container submitted by the Premium user, for lack of resources, our framework executes another container, for a user who has a lower service request in order to optimize the global scheduling of all containers. The goal is not to stop the scheduling process when a container is not executed and to wait.

6.2 Containers Submitted Online

Figure 3 shows the order of execution of 90 containers submitted online with a fixed frequency. Each 3 s 3 containers are submitted by 3 different users. That means, each 3 s, each user submits one container. Figure 4 shows the order of execution of 90 containers submitted according to the Google Cluster Data traces [15] patterns. The Google traces information (May 2011), are related to the submission frequency time of requests on cluster of about 12.5k machines. In our case, the 90 containers are submitted using the same submission frequency time as the first 90 requests submitted in Google traces. The 90 containers are distributed as follows: (i) 2 containers submitted by Premium user and (ii) 88 containers submitted by Advanced user. In a complementary way, Fig. 5 shows the order of execution of 90 containers submitted according to the real-world trace files of an international company called Prezi [17]. These traces represent the submission frequency time of the web oriented applications. In our case, the 90 containers are submitted using the same submission frequency time as the first 90 web oriented applications submitted in Prezi traces. The 90 containers are distributed as follows: (i) 10 containers submitted by Premium user, (ii) 13 containers submitted by Advanced user and (iii) 67 containers submitted by Best effort user. According to Figs. 3, 4 and 5, we note that there is an overlap between the execution of containers. This expected overlap is due to the fact that containers are submitted online by different users.

6.3 Comparison Between the Average Number of Cores Allocated for Each User

To the best of our knowledge, there is no framework which configures dynamically the number of cores that must be allocated to each container. This explains that it is impossible to compare the performance obtained using our framework with another state-of-art framework. However, in Table 1 we shows a comparison between the average number of cores assigned to each user. As a result, we note that our framework assigns, for each submission type, more cores to the user with highest services. We note also that the user with the low service gets always the smallest number of cores.

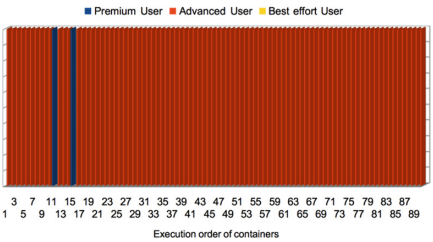

Fig. 4. Submission of 90 Docker containers according to Google traces frequency

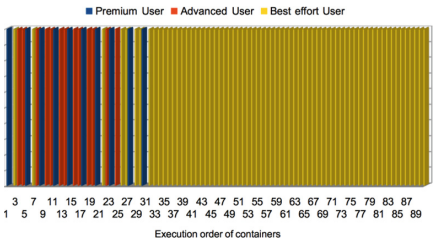

Fig. 5. Submission of 90 Docker container according to Prezi traces frequency

Table 1. Comparison between the average number of cores allocated for each user

Submitting type	Average number of cores allocated for each user		
	Premium user	Advanced user	Best effort user
Submission at the same time	27.1	15.26	6.16
Submission online with a fixed frequency	27.63	15.26	5.33
Submission online according to Google traces	32	13.7	-
Submission online according to Prezi traces	30.4	17.15	3.98

7 Conclusion

We have presented, in this paper, a new framework adapted for cloud computing environments in the context of containers technologies. The novelty of our framework relies on SLA classes to optimize the global scheduling and the allocation of resources for containers. Our solution proposes to users two qualitative and two quantitative SLAs classes with three services for each class (Premium, Advanced and Best effort). In our framework, the number of resources are computed, dynamically, according to the quantitative number of resources class.

As a first perspective, we propose to compute the number of resources by taking into consideration the submitted container history. It is challenging to efficiently decide when and how to reconfigure the cloud in order to dynamically adapt to the changes. Such a challenge has been identified as a MAPE-K (Monitoring, Analysis, Planning, Execution, and Knowledge) control loop by IBM, deeply investigated in [8], resulting in the concept of autonomic computing that could be used in our case.

We may also wonder if the approach is flexible enough in the context of multiple cloud providers. This question poses the problem of the adoption of our economic model. We also propose, as a perspective, to add to our framework a consolidation heuristic which allows to set dynamically the number of active

cloud nodes in the infrastructure. This means that, according to the global load of nodes, the framework decides the number of active nodes to reduce the energy consumption.

Acknowledgements. This work is funded by the French *Fonds Unique Ministériel (FUI)* Wolphin Project. We thank Grid5000 team for their help to use the testbed.

References

1. Behzadian, M., Kazemzadeh, R., Albadvi, A., Aghdasi, M.: Promethee: a comprehensive literature review on methodologies and applications. Eur. J. Oper. Res. **200**(1), 198–215 (2010)
2. Borgetto, D., Maurer, M., Costa, G.D., Pierson, J., Brandic, I.: Energy-efficient and SLA-aware management of IaaS clouds. In: International Conference on Energy-Efficient Computing and Networking, e-Energy 2012, Madrid, Spain, p. 25 (2012)
3. Calders, T., Assche, D.V.: Promethee is not quadratic: an o(qnlog(n)) algorithm. Omega **76**, 63–69 (2018)
4. IBM CPLEX solver: https://www.ibm.com/products/ilog-cplex-optimization-studio
5. Garey, M.R., Johnson, D.S.: Computers and Intractability: A Guide to the Theory of NP-Completeness. W. H. Freeman & Co., New York (1979)
6. Fui-22 wolphin project: https://lipn.univ-paris13.fr/~menouer/wolphin.html
7. Grid5000: https://www.grid5000.fr/
8. Huebscher, M.C., McCann, J.A.: A survey of autonomic computing–degrees, models, and applications. ACM Comput. Surv. **40**(3), 7:1–7:28 (2008)
9. Menouer, T., Cerin, C.: Scheduling and resource management allocation system combined with an economic model. In: IEEE International Symposium on Parallel and Distributed Processing with Applications (IEEE ISPA) Guangzhou, China (2017)
10. Peinl, R., Holzschuher, F., Pfitzer, F.: Docker cluster management for the cloud-survey results and own solution. J. Grid Comput. **14**(2), 265–282 (2016)
11. Deshmukh, S.C.: Preference ranking organization method of enrichment evaluation (promethee). Int. J. Eng. Sci. Invent. **2**, 28–34 (2013)
12. Tang, C., Steinder, M., Spreitzer, M., Pacifici, G.: A scalable application placement controller for enterprise data centers. In: Proceedings of the 16th International Conference on World Wide Web, Banff, Alberta, Canada, pp. 331–340, May 2007
13. Ullman, J.: NP-complete scheduling problems. J. Comput. Syst. Sci. **10**(3), 384–393 (1975)
14. The apache software foundation. mesos, apache. http://mesos.apache.org/
15. Google cluster data traces. https://github.com/google/cluster-data/
16. Kubernetes scheduler. https://kubernetes.io/
17. Prezi real-world traces. http://prezi.com/scale/