Re-deploying Microservices in Edge and Cloud Environment for the Optimization of User-Perceived Service Quality

Xiang He(✉), Zhiying Tu(✉), Xiaofei Xu(✉), and Zhongjie Wang(✉)

School of Computer Science and Technology, Harbin Institute of Technology, Harbin, China
september_hx@outlook.com,
{tzy_hit,xiaofei,rainy}@hit.edu.cn

Abstract. Deploying microservices in edge computing environment shortens the distance between users and services, and consequently, improves user-perceived service quality. Because of resource constraints of edge servers, the number and Service Level Agreement (SLA) of microservices that could be deployed on one edge server are limited. This paper considers *user mobility*, i.e., location changes of massive users might significantly result in deterioration of user-perceived service quality. We propose a method of looking for an optimized microservice re-deployment solution by means of *add*, *remove*, *adjust*, and *switch*, to make sure service quality that massive users perceive always conforms to their expectations. Three algorithms are adopted for this purpose, and an experiment in real-world edge-cloud environment is also conducted based on *Kubernetes* to re-deploy microservice systems automatically.

Keywords: Microservices · Edge and cloud environment · Service system re-deployment · Service quality · User mobility

1 Introduction

Recently, lots of research have been conducted on *Cloud - Edge - Mobile devices* architecture in Edge Computing. In such architecture, the distance between users and services can be shortened by deploying services on edge servers. Microservices architecture and container technology have been adopted so that the services can be easily deployed, and services can be migrated to a cloud-native architecture [1], which makes the system adapt to the user demand changes.

User demands cannot remain unchanged all the time. *User mobility*, *functional requirement changes*, and *quality expectation changes* are typical changes in user demands. Deployment of microservices in cloud and edge environment should make changes accordingly to keep users satisfied. In this paper, we consider *user mobility* as the trigger of microservice system re-deployment.

In this paper, the following factors are considered: (1) *Multi-services*: A service system is composed of many services with different functionalities and SLAs,

S. Yangui et al. (Eds.): ICSOC 2019, LNCS 11895, pp. 555–560, 2019.
https://doi.org/10.1007/978-3-030-33702-5_42

and a user might need to request two or more services to satisfy his needs; (2) *Multi-SLAs*: Each microservice in the system might offer different SLAs; at a certain time, its SLA is deterministic, but it would switch to another SLA after re-deployment; (3) *Multi-users*: Massive users are simultaneously requesting services, and it is necessary to keep the quality of service that a service system offers always above their expectations; (4) *Resource constraints*: Computing resources offered by each edge server is different and limited.

Our work is to look for an optimized re-deployment solution, and the main contributions of the paper are listed below:

- We define the optimization problem of microservice system evolution which takes multi-services, multi-SLA, and multi-users into consideration. This extends traditional placement research to make it more fit for real world.
- We use Genetic algorithm (GA), heuristic algorithm (HA) and Artificial Bee Colony algorithm (ABC) to look for an optimized re-deployment solution. A set of experiments are conducted under four representative user mobility scenarios and the results have validated algorithm performance.
- We develop a tool based on docker and Kubernetes to execute a re-deployment solution in real-world edge-cloud computing environment, which empowers a service system the capacity of automatic and continuous re-deployment.

The remainder of this paper is organized as follows. Section 2 introduces definitions. Section 3 describes algorithms. Section 4 details the experimental and protosystem. Section 5 reviews related work. Section 6 concludes the paper.

2 Problem Formulation

Definition 1 (Service). The set of services are describes as S. A service s is described as a set of triple $\{(l_{sla}, r, n)\}$, and for each $s \in S$: id is the unique id which is used to distinguish different instances of the same service with the same SLA on one server; r is how much computing resources s needs to offer the quality level l_{sla}; n is the maximum number of users that one instance of s with l_{sla} can serve concurrently.

Definition 2 (Cloud/Edge Server Node). E stands for a set of server nodes, and a node is described as $e = (type, r, loc)$: $type \in \{EDGE, CLOUD\}$ is the type of e (might be an edge server or a cloud server); r is the total computing resources e can be provided for service instances; loc is the geographic location of e (*latitude* and *longitude*). It is important to notice the difference between cloud and edge nodes: computing resources in a cloud node is much more sufficient than an edge node.

Definition 3 (Re-deployment operations on microservice instances). $O^I = \{adjust, add, remove, retain\}$ is used to describe four types of re-deployment operations on a microservice instance: *adjust* means adjusting the quality level of an instance; *add* means creating a new instance; *remove* means deleting an instance; *retain* means keeping unchanged.

Definition 4 (Re-deployment operations on users). $O^U = \{switch, keep\}$: *switch* stands for switching a user's request on a service to another; *keep* means keeping a user on the same instance before and after.

Problem Definition. A service system evolves from time t to $t + \delta$ by a set of operations on users $OU = \{o^U | o^U \in O^U\}$ and a set of operations on microservice instances $OI = \{o^I | o^I \in O^I\}$. The δ means that the service system doesn't keep evolving all the time, only be triggered when most of the user demands are not satisfied. The optimization problem is described below:

$$minC_e = min(\sum_{o^I \in OI} cost(o^I) + \sum_{o^U \in OU} cost(o^U))$$

$$\text{s.t.} \begin{cases} f(u_i, eu_{ij}) * sla(eu_{ij}) >= sla(u_i, s_j), & \forall s_j \in S, \forall u_i \in U(t+\delta) \\ \sum_{inst\ on\ e_k} r(inst) <= r_{design}(e_k), & \forall e_k \in E(t+\delta) \\ 1 \leq ns(inst) \leq ns_{design}(inst), & \forall inst \in Inst(t+\delta) \end{cases} \quad (1)$$

where *Inst* denotes the set of instances, eu_{ij} is an instance of service s_j and a user u_i's request on s_j is to be satisfied by eu_{ij}. $f()$ is a function for *attenuation coefficient* of quality level *w.r.t.* the distance between a user and a service instance. $r()$ get the amount of computing resources that *inst* requires, and $ns()$ gets the actual number of users that *inst* is serving, while $ns_{max}()$ gets the maximal number of users that *inst* can serve concurrently.

The first constraint assures that the quality level that each user could be satisfied. The second constraint assures that the total resources do not exceed the maximal resource offering by the node. The number of users that each instance serves cannot exceed the maximal number that the instance can serve concurrently, which is assured by the last constraint.

3 Algorithms

In ABC, to initialize the population, one server node is picked up randomly from the candidate list for every service that each user requests. An instance with the lowest cost to accept the user will be chosen, and the result is treated as the *nectar*. In the *employed bees* phrase, a non-empty service instance will be chosen, and all users it serves will be dispatched to other instances. In the *onlooker bees* phrase, some nectar will be picked randomly and dispatched to other instances on nodes randomly. The abandoned food sources will be replaced by solutions randomly generated in the scout bees phrase.

In GA, the initialization process is the same as ABC. The gens represent the instances that are chosen for the user demand for every service. Some of the genes will be randomly chosen, and they will be adapted to other instances on the nodes which are in their candidate lists, and exchanging parts of the gens between two solutions is treated as the crossover.

The heuristics algorithm is based on the following heuristic rules: (1) Assign each user to the server node that is the most closest to him; (2) Existing instances

will be considered first. Or the cost of *add* and *adjust* is compared, and the operation with the lower cost will be chosen; (3) User demands that cannot satisfied by the closest server node will be assigned to the next closest node; (4) When there are no enough computing resources, existing instances will be merged, and instances that have no user will be removed.

4 Experiments and Prototype

4.1 Experiments Setup

In the experiments, the cellular layout is used to place edge servers. The costs of all operations come from the average time (seconds * 10) of necessary Kubernetes operations. It is noteworthy that Kubernetes doesn't support dynamic resource allocation for pods, the *adjust* operation has to be split into one *remove* and one *add*. The cost of *switch* operation is calculated by the time that 1 MB data needed to transfer with 100 Mbps. The costs for *add*, *remove*, *adjust*, *retain*, *switch*, and *keep* used here are 68, 25, 94, 0, 0.8, and 0.

There are three main scenarios: (1) *Group to Group*: Users are gathered in some specific locations (i.e., they are in the form of groups;) and after their moves, they are in re-grouped; (2) *Random to Group*: Users are distributed randomly, and after their moves, they are gathered in groups; (3) *Group to Random*: Users are gathered in groups, and then they disperse all over the area.

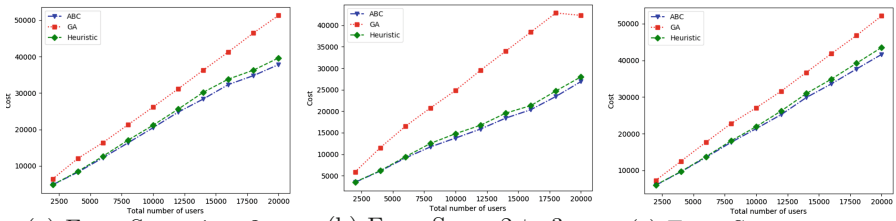
4.2 Scenario 1: Group to Group

In this experiment, we evaluate our algorithms with the scenario 1. We generate three basic scenes: Scene 1, Scene 2 and Scene 3. They are three different situations that users gather together. Three experiments were conducted: moving from Scene 1 to 2, from Scene 2 to 3, and from Scene 3 to 1. The results are shown in Fig. 1. The x-axis stands for the number of users in the experiment, and the y-axis is the cost of the evolution plan that algorithm generated.

As shown in Fig. 1, the cost of evolution is linearly and positively correlated with the number of users. As the number of users grows, more user connections should be switched from the old server node to the new one, and more service instances must be deployed on server nodes. It shows that our ABC algorithm performs better than the GA and HA in all three situations. Both HA and ABC have a huge improvement compared to GA.

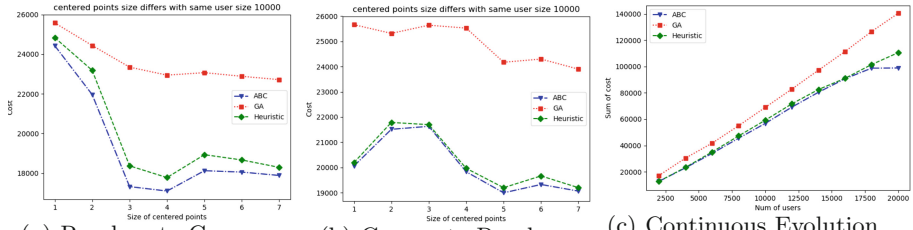
4.3 Scenario 2 and 3: Random to Group and Group to Random, and Continuous Evolution

In this experiment, we explore the situations of moving from random to group and from group to random with 10000 users and the number of server nodes that the users are grouped by differs from 1 to 7. The performance of the algorithms in the situation of *continuous evolution* is also explored.



(a) From Scene 1 to 2 (b) From Scene 2 to 3 (c) From Scene 3 to 1

Fig. 1: Re-deployment cost *w.r.t.* number of users in scenario 1



(a) Random to Groups (b) Groups to Random (c) Continuous Evolution

Fig. 2: Re-deployment cost in scenarios 2&3 and continuous evolution

The results in Fig. 2(a) and (b) show the ABC still performs better than GA and HA in these two situations. The x-axis is the number of cluster nodes and the y-axis is the cost of evolution plan generated by the algorithms.

For continuous evolution, we execute the re-deployment algorithm three times: Scene 1 to 2, then 2 to 3, and back to 1. And the cost in total is the sum of the cost. Figure 2(c) shows that ABC performs better than GA and HA. It means ABC does not overlook the global cost of the continuous evolution while trying to find the best solution to part of the problem, and the stability of the algorithm and the deployment of the service system are guaranteed.

4.4 Prototype

The prototype system is built with *Docker* and *Kubernetes*. Because the user location awareness is beyond our work, it will not detailed here.

As listed in Sect. 3, there are four types of operations that we need to implement, i.e., *add*, *remove*, *adjust* and *switch*. It's easy to implement the *switch* operation by proxy and gateway on each node, so we only illustrate how to do *add*, *remove* and *adjust* operations in K8s with the command tools *kubectl*. We assume that all the configuration files required by K8s are prepared in advance.

For *add* operation, the configuration file that related to the desired service will be used by *kubectl* with node-selector attribute. What to mention is we should label the pod with the instance that is generated by the algorithm. For *remove* operation, the pod id, which is associated with the instance id label, is passed to the command tool. Unfortunately, K8s doesn't support dynamic

resource allocation now, thereby the *adjust* operation is the combination of *add* operation and *remove* operation.

5 Related Work

Zhang et al. [2] designed a framework for dynamic service placement based on control and game theoretic models, aiming at optimizing hosting cost dynamically in Geographically Distributed Clouds. Selimi et al. [3] studied service placement in Community networks to improve the quality of experience. Mahmud et al. [4] proposed a QoE-aware application placement policy. Wang et al. [5] proposed an ITEM algorithm to solve the service placement of Virtual Reality applications with consideration about the QoS and the economic operations.

To sum up, in existing works there are no enough attentions having been paid to the changing demands of *multi-users* in a *multi-service* system that offers *multi-SLAs*. Being a very common scenario in real world and objective of our work in this paper, it is a significant extension to current research.

6 Conclusions

This paper considers user mobility to re-deploying microservices in edge and cloud environment, which can improve user-perceived service quality. Considering the challenges of *multi-services*, *multi-users* and *multi-SLAs*, and by six types of basic operations and three strategies, our methods could identify an optimized re-deployment solution effectively. And a prototype tool has been developed. Other types of user demand changes will be considered in future work.

Acknowledgment. Research in this paper is partially supported by the National Key Research and Development Program of China (No. 2017YFB1400604), the National Science Foundation of China (61802089, 61772155, 61832004, 61832014).

References

1. Balalaie, A., Heydarnoori, A., Jamshidi, P.: Microservices architecture enables DevOps: migration to a cloud-native architecture. IEEE Softw. **33**(3), 42–52 (2016)
2. Zhang, Q., Zhu, Q., Zhani, M.F., Boutaba, R., Hellerstein, J.L.: Dynamic service placement in geographically distributed clouds. IEEE J. Sel. Areas Commun. **31**(12), 762–772 (2013)
3. Selimi, M., Cerdà-Alabern, L., Sánchez-Artigas, M., Freitag, F., Veiga, L.: Practical service placement approach for microservices architecture. In: 2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID), Madrid, pp. 401–410 (2017)
4. Mahmud, R., Srirama, S.N., Ramamohanarao, K., Buyya, R.: Quality of Experience (QoE)-aware placement of applications in Fog computing environments. J. Parallel Distrib. Comput. (2018)
5. Wang, L., Jiao, L., He, T., Li, J., Mühlhäuser, M.: Service entity placement for social virtual reality applications in edge computing. In: IEEE INFOCOM 2018 - IEEE Conference on Computer Communications, Honolulu, HI, pp. 468–476 (2018)