Runtime Resource Management for Microservices-Based Applications: A Congestion Game Approach (Short Paper)

Ruici Luo[1,3], Wei Ye[2,3], Jinan Sun[2,3(✉)], Xueyang Liu[2,3], and Shikun Zhang[2,3]

[1] School of Electronics Engineering and Computer Science, Peking University, Beijing, China
[2] National Engineering Research Center for Software Engineering, Peking University, Beijing, China
[3] Key Laboratory of High Confidence Software Technologies, Ministry of Education, Beijing, China
{luoruici,wye,sjn,liuxueyang,zhangsk}@pku.edu.cn

Abstract. The term "Microservice Architecture" has sprung up in recent years as a new style of software design that gains popularity as cloud computing prospers. In microservice-based applications, different microservices collaborate with one another via interface calls, but they may also compete for resources when an increase of users' need renders the resources insufficient. This poses new challenges for allocating resources efficiently during runtime. To tackle the problem, we propose a novel approach based on Congestion Game in this paper. Firstly, we use a weighted directed acyclic graph to model the inter-relationship of the microservices that compose an application. Then we use M/G/1 Queue in Queue Theory to describe the arrival process of access requests, and combine it with the above graph to calculate the arrival rate of access requests to each microservice, which in turn is used to estimate response time in a newly-designed microservice revenue function. Finally, we define resources competing problem as a congestion game where each microservice is a player aiming to maximize its revenue, and propose an algorithm to find Nash equilibrium in polynomial time. Experiment results show that our approach can effectively improve the overall performance of the system with limited resources, and outperform Binpack and Spread, two scheduling strategies used in Docker Swarm.

Keywords: Microservice architecture · Resource management · Game theory

1 Introduction

The popularity of cloud computing technology not only promotes the development of computer hardware and system software architectures, but also brings

H. Gao et al. (Eds.): CollaborateCom 2018, LNICST 268, pp. 676–687, 2019.
https://doi.org/10.1007/978-3-030-12981-1_47

about changes in the way software is developed and used. The idea of IT resource servitization is becoming increasingly popular, leading to the trend of "X as a service." The service model represented by IaaS, PaaS and SaaS has been widely adopted. In the open, dynamic and complex cloud computing environment, software systems need continuous online evolution [19] to respond quickly to users' need, therefore the complexity of software is ever growing. Considering the principles, methods, and techniques for modeling and controlling complexity in software engineering, the basic idea can be summarized as separation of concerns [14]. Microservice Architecture [7] is a design principle that aims to fight against the increasing complexity of software in the cloud environment. The core idea is to separate an application into a series of distinct microservices, such that each of them addresses a separate business logic and can run in a separate environment. It can make the boundaries between services clear and enable them to adopt lightweight mechanism to communicate. As a result, it forms a highly cohesive and loosely-coupling architecture.

In a large-scale application system based on microservice architecture, such properties as large amount of microservices, continuous online evolution and complex dependencies among microservices pose new challenges for runtime resource management. On one hand, microservices need to accomplish a specific business logic by collaboration; on the other hand, they may also compete for resources in a constrained runtime environment.

Game theory has been successfully applied to many computer resource-related optimization problems such as online price setting, flow and congestion control, network routing optimization, etc. [2]. Nash Equilibrium [13] in game theory is the most common solution: it is a state where for any participant in the game, he cannot get more benefits by changing his strategy without changing the strategies of other participants. At this point the strategy combination of all participants constitutes the Nash equilibrium state.

Considering the collaboration and competition among microservices, we propose to use game theory to model the runtime resource management problem as a congestion game, and optimize resource allocation by finding the Nash Equilibrium state of the Game, where each microservice is considered as a player to maximize its revenue. More specifically, we make the following contributions in this paper:

1. We establish a microservice application model in which we quantify call relationships among microservices based on runtime microservice interaction information.
2. We design a microservice revenue function based on the compliance level with quality of service (QoS) requirements specified in Service Level Agreement (SLA).
3. We define the competition for resources among microservices as a congestion game model and provide a polynomial-time algorithm for solving resource management optimization problem.

The rest of the paper is organized as follows. Section 2 summarizes related works. Section 3 provide some background information regarding resource man-

agement under microservices scenario. Section 4 presents our proposed congestion game model of resource management for microservices. Section 5 discusses the algorithm to find the Nash Equilibrium of the congestion game. Section 6 presents the results of experiments done to validate our model, and finally Sect. 7 concludes the paper.

2 Related Works

The methods of resource planning, scheduling and management in cloud computing environment can be divided into three categories: optimization method, adaptive method and game theory method.

Optimization is a crucial method to solve the scheduling and allocation problem of computing resources in a cloud environment. [18] solved deployment optimization problem by greedy algorithm under the scenario where a service joins in and leaves dynamically. [11] tried to minimize the cost of computing resources during service deployment by studying dynamic backpack problem in which the goal is to minimize the sum of the cost of all backpacks. [6] studied the scenario of multiple IaaS and targeted at minimizing the overall consumption, and proposed to solve virtual machine deployment optimization problem by random integer programming method.

A classic MAPE-K feedback loop can be formed by monitoring and analyzing the runtime data of an application, planning the resources the application needs and then executing, combined with a shared knowledge base [9]. A self-adaptive software using such method usually consists of two parts: the managed element and the managing element. The former refers to the application logic that can be dynamically adjusted during runtime; the latter refers to the adaptive logic that can regulate the application logic through a feedback loop. Considering that the applications in a cloud environment are usually deployed in a mixed operating environment of IaaS and Paas, [4] proposed a scheduling method for two kinds of virtualization resources (virtual machine instance of IaaS layer and container of PaaS layer) through a cybernetic feedback loop.

Because of the resemblance between the resource competition behavior among applications in a cloud computing environment and the economic competition behavior in a free market, game theory can be used to describe the competitive relationship in resource management. In existing literatures, it is common to consider resource-related roles as players in a game where each player gets the corresponding resources by adopting a certain strategy to optimize its own revenue/cost. [3] studied the scenario where multiple SaaS providers run applications in the same IaaS. After establishing a mathematical model to measure the benefits and costs of SaaS and defining the decision space of IaaS and SaaS in the game, the authors found the optimal solution for IaaS/SaaS resource pricing/acquisition by calculating the Nash equilibrium. [20] described the competitive characteristics of parallel computing tasks on resources at the business level by game theory and proposed a scheduling algorithm based on the dual constraints of completion time and cost, considering both optimization and fairness.

Most existing game theory-based methods focus on scheduling of infrastructure (virtual machine) resources, but do not consider more granular scheduling units and their collaboration at the application level.

3 Background

Different microservices have different requirements for computing resources, load, and QoS. Take a social application for example. The activity feed service displays contents such as topics, articles, and videos of interest to users, and is used to display advertisements. It is the service that has the greatest impact on user experience and application revenue. In comparison, the user service stores a huge amount of social relationship data, which is important for data analysis and recommendation tasks, but is less important than the activity feed service from the perspective of application revenue. It is conceivable that the application runtime resource management needs to consider the importance of each service from the business perspective.

In addition, the satisfaction level of SLA will affect the application revenue as well. The response time of a user's access request is the most important factor to measure the satisfaction level. The shorter the response time, the higher the SLA satisfaction level, and the higher the revenue (the revenue agreed in the SLA, the revenue from user experience, etc.). The longer the response time, the lower the SLA satisfaction level, and the lower the revenue (default penalty, user loss, etc.).

Therefore, taking both the business importance and response time into account, we come up with the following revenue function for one access request to microservice i:

$$\theta_i = v_i + m_i r_i \tag{1}$$

where v_i is the revenue when response time is zero, and r_i is the actual response time of the access request. Note that the slope of the function $m_i < 0$, which means that the shorter the response time, the higher the revenue and vice versa. For each microservice i, a different v_i and m_i can be set to reflect its business importance. For example, an activity feed microservice in a social application, corresponding to a smaller primary function slope, indicates that the smaller the response time, the more revenue it will receive than the benefits of other services.

4 Microservice Oriented Model for Resource Management Issue

Therefore in this section, we will (1) build a microservice application model based on the runtime interaction information of microservices; (2) design a microservice revenue model based on the revenue function defined in Sect. 3 (Eq. (1)); (3) define a congestion game model for microservice runtime resource management.

4.1 Microservice Application Model

An **Application** is a triplet (S, E, T) where:

- S is the collection of microservice that make up an application. For any microservice $s \in S$, there is a collection of access endpoints $E_s = \{e_1, e_2, \ldots, e_n\}$.
- $E = \bigcup_{s \in S} E_s$ is the collection of access endpoints for all microservices.
- T represents the interaction between endpoints, $\forall t \in T, t = <e_o, e_q, p>$, $e_o, e_q \in E, p \in \mathbb{R}^+$

Essentially, the definition above can be regarded as a directed acyclic graph (DAG) composed of an endpoint set and a call request set. For each edge in the graph, it associates two endpoints to represent a call relationship, and uses a positive real number to represent the expected number of accesses that a access to the source endpoint will cascade to the target endpoint. $p < 1$ means only some requests will trigger sub-request (e.g. A branch judgement occurs in the program). $p \geq 1$ means a request will trigger more than one sub-requests on average (e.g. Loops, branches or other situations occur in the program).

4.2 Microservice Revenue Model

The number of microservice requests per unit time has several properties: (1) the number of requests is large enough; (2) a single request has little impact on the overall system performance and resource consumption; (3) all requests arrive independently. Therefore, the number of access requests for microservice per unit time can be described by Poisson distribution:

$$P(X = k) = \frac{e^{-\lambda}\lambda^k}{k!} \tag{2}$$

Where λ represents the average number of microservice access requests that arrive per unit time. In order to cope with a large number of access requests, each logical microservice can physically have multiple instances (containers) running at the same time, distributed on different virtual machines. Provided that access requests are balanced to each microservice running instance, we can use a queuing system to describe how each microservice processes its access requests, assuming that the following properties hold:

- The request arrival time is in accordance with the Poisson distribution, i.e. the exponential probability density distribution.
- Process time for each request is the same.
- For requests that arrive at the same time, the microservice processes them with time-sharing policy.

Therefore, processing access requests by each microservice accords with the condition of M/G/1 Queue [5]. For microservice i, the expectation of access request response time γ can be calculated as:

$$E[\gamma_i] = \frac{1}{C_i\mu_i - \lambda_i} \tag{3}$$

- C_i is the maximum computing resource that the corresponding microservice container can consume.
- μ_i is the service efficiency of the corresponding microservice container, which is the number of requests processed per unit time per unit of computing resource.
- λ_i is the access request arrival rate at each container, which is equivalent to the total request arrival rate of microservice i divided by its number of physical units, i.e. containers.

For the parameter λ_i required in Eq. (3), we need to make predictions based on history data. Using Microservice Application Model defined in Sect. 4.1, $\forall i \in S$, the arrival rate collection is defined as $\omega_i = \{\lambda_{ie_1}, \lambda_{ie_2}, \ldots, \lambda_{ie_n}\}$, where $e_1, e_2, \ldots, e_n \in E_i$. The total arrival rate of microservice i can be represented as $\lambda_i = \sum_{e \in E_i} \lambda_{ie}$. For any microservice, it is possible to cascade its access requests to the microservices it depends on. Using the call graph T in Microservice Application Model, we can easily derive the T-based arrival rate update algorithm by traversing the graph.

Assuming that all microservices run on a finite resource collection $R = \{1, \ldots, r\}$, and the access request arrival rate of microservice i on resource r per unit time is λ_{ir}, the total number of microservices that occupy resource r is x_r, the microservice revenue from resource r per unit time can be written as follows:

$$d_r(x_r) = \lambda_{ir}\theta_i \tag{4}$$

Substitute θ_i with Eq. (1) from Sect. 3, and we have:

$$d_r(x_r) = \lambda_{ir}(\nu_i + m_i E[\gamma_i]) \tag{5}$$

Note that we have used the expected value of γ_i to approximate γ_i.

Substitute $E[\gamma_i]$ with Eq. (3), and we have:

$$d_r(x_r) = \lambda_{ir}(\nu_i + \frac{m_i}{\frac{C}{x_r}\mu_i - \lambda_{ir}}) = \lambda_{ir}\nu_i + \frac{m_i}{\frac{C\mu_i}{x_r\lambda_{ir}} - 1} \tag{6}$$

Here we have substituted C_i in Eq. (3) with $\frac{C}{x_r}$, where we assume that all containers on resource r share the computing resource equally, and that each resource has the same constant computing capacity C.

4.3 Microservice Congestion Game Model

Congestion games are used to describe scenarios where players share resources in a game, in which every player maximizes its own revenue by strategically selecting resources. The revenue generated by the resource is related to the number of players who choose this resource, which means that the more players that have chosen this resource, the less revenue each player can earn from this resource. The formal definition of a congestion game is as follows:

- A finite set of players $M = \{1, \ldots, n\}$

- A finite set of congestible resources $R = \{1, \ldots, r\}$
- A finite set of strategies Σ_i for each player, where each strategy $P \in \Sigma_i$ is a subset of resource set R. We use $\Sigma = \prod_{i=1}^{n} \Sigma_i$ to denote the joint strategy space and $\sigma = (\sigma_1, \ldots, \sigma_n) \in \Sigma$ is a strategy vector in which the player i chooses the strategy σ_i
- For each resource $r \in R$ and a strategy vector $\sigma = (\sigma_1, \ldots, \sigma_n) \in \Sigma$, the load (i.e. congestion number, or the number of times this resource is selected) $x_r(\sigma) = \#\{i : r \in \sigma_i\}$, where # means the size of the set.
- For each resource $r \in R$, a revenue function $d_r : \mathbb{N} \to \mathbb{R}$ describes the relationship between the number of times this resource is selected x_r and the revenue every player can earn. d_r is a monotonically decreasing function on x_r.
- For a given strategy vector $\sigma = (\sigma_1, \ldots, \sigma_n) \in \Sigma$, player i's total revenue $S_i = \sum_{r \in \sigma_i} d_r(x_r(\sigma))$. For Congestion Game Model, a strategy vector $\sigma^* = (\sigma_1^*, \ldots, \sigma_n^*)$ is a Nash Equilibrium of the congestion game if and only if

$$\sum_{r \in \sigma_i^*} d_r(x_r(\sigma^*)) \geq \sum_{r \in \sigma_i} d_r(x_r(\sigma^{'})), \forall i \in N, \forall \sigma_i \in \Sigma_i, \sigma^{'} = (\sigma_i, \sigma_{-i}^*) \tag{7}$$

We further apply Microservice Revenue Model in Sect. 4.2 to the congestion game. For any microservice i and its non-empty decision vector $\sigma = (\sigma_1, \ldots, \sigma_n) \in \Sigma$, its revenue function is defined as follows:

$$S_i(\sigma) = \sum_{r \in \sigma_i} d_r(x_r(\sigma)) \tag{8}$$

$$d_r(x_r(\sigma)) = \lambda_{ir}\nu_i + \frac{m_i}{\frac{C\mu_i}{x_r(\sigma)\lambda_{ir}} - 1} \tag{9}$$

We can see that $d_r(x_r)$ is a monotonically decreasing function on x_r (note that $m_i < 0$), which satisfies the conditions of Congestion Game Model.

5 Nash Equilibrium of the Congestion Game

The existence of Nash equilibrium can be shown by constructing a potential function [15] that assigns a value to each outcome. Moreover, the construction will also show that iterated best response finds a Nash equilibrium.

To solve the Nash equilibrium of the congestion game, we propose an algorithm that adopts an incremental optimization scheme. We set up an empty set as the strategy vector initially. At each iteration, only one player is allowed to change its current strategy and choose its best response against the strategy vector. For n microservices, there are n steps to achieve the Nash equilibrium, where in each step only one other player enters. When no player can change his strategy solely to gain more utility, the Nash equilibrium in the step is achieved.

In general, we consider the nth step, where former $n-1$ players have achieved equilibrium and their strategy vector is $\sigma(n-1) = (\sigma_1, \sigma_2, \ldots, \sigma_{n-1})$. It is interesting to find out, after the nth player enters, how will the system regain the

equilibrium. The player n will choose his best response strategy against the former $n-1$ players' strategies in equilibrium, which must congest some of the resources and affect the strategy vector $\sigma(n-1)$. Each affected player changes its current strategy and chooses the best response like the nth player. The Nash equilibrium is achieved at last. Formally, $S_n(\sigma_n, \sigma(n-1))$ is denoted as the utility of the player n, where player n chooses the strategy σ_i and the former $n-1$ players choose the strategy tuple $\sigma(n-1) = (\sigma_1, \sigma_2, \ldots, \sigma_{n-1})$.

To regain the Nash equilibrium strategy vector of n players, we propose Algorithm 1:

Algorithm 1. Regain to equilibrium(n)

Require: n-1 players' strategy vector $\sigma(n) = (\sigma_1, \sigma_2, \ldots, \sigma_{n-1})$
Ensure: n players' strategy vector $\sigma(n) = (\sigma_1, \sigma_2, \ldots, \sigma_n)$;
1: calculate the nth player's best strategy σ_n^*
2: initialize $\sigma(n) = (\sigma(n-1), \sigma_n^*)$
3: $i = 0$
4: **while true do**
5: **if** $i == n$ **then**
6: break;
7: **end if**
8: **if** $\forall \sigma_i \in \Sigma_i, S_i(\sigma(n)) \geq S_i(\sigma_i, \sigma_{-i}(n))$ **then**
9: $i = i + 1$;
10: **else**
11: calculate the ith player's best strategy σ_i^*
12: replace the ith players strategy: $\sigma(n)[i] = \sigma_i^*$
13: $i = 0$;
14: **end if**
15: **end while**

For every player, the time complexity of calculating best strategy is $O(r * n)$, where r is the number of resources and n is the number of players. To obtain all players' Nash equilibrium state, the iteration in Algorithm 1 would be executed at most $n * r$ times. In a word, to obtain the final strategy vector, the time complexity of the algorithm is $O(n^2 r^2)$.

6 Evaluation

6.1 Experiment Setting

We conducted experiments on an open-source project called Pwitter [1], which is a social-networking application resembling Twitter. It is a three-layer application developed in python that runs on Gunicorn and uses Redis and MySQL for data storage. We modified and extended Pwitter to microservice architectural style. After conversion, our version of Pwitter has a network of 30 microservices, and each of them is exposed to between 10 to 20 API endpoints via HTTP. We run

these microservices on virtual machine instances provided by Aliyun (a cloud provider in China). The specification of each instance is ecs.c5.x2large with 8 core CPU and 16G memory.

Docker is installed on each virtual machine instance, and each microservice runs in the form of a Docker Container. We use Swarm to administrate all the microservices, which is an official cluster management tool of Docker. To deploy a microservice, we can simply initiate a request to the Manager Node, which will place its corresponding runtime container on a suitable virtual machine instance based on the scheduling results of the chosen resource management algorithm to maximize the microservice revenue.

Docker Swarm has three inherent strategies for cluster scheduling, namely:

- Spread, the default Strategy, which picks the Worker node that currently has the least resource (CPU, memory etc.) consumption, so as to prioritize on fair usage of resources.
- Binpack, a strategy contrary to Spread, which chooses to fill up a Work node as much as possible first, so as to keep more nodes available.
- Random, which picks a Worker node at random.

To verify the effectiveness of our proposed scheduling strategy based on Congestion Game, we used this strategy to deploy the 30 microservices of Pwitter and compare the performance with that of using Spread and Binpack. After deployment, we simulated http requests and gradually increased the frequency from 1000 requests per second to 5000 requests per second at 1000 interval. To verify the effectiveness of different strategies under different resource constraints, we repeated the above experiment on 5, 10, 15, 20 and 30 virtual machine instances respectively.

6.2 Experiment Results

We use average request response time, ratio of request failure and microservice revenue as three metrics to compare the performance of each strategy.

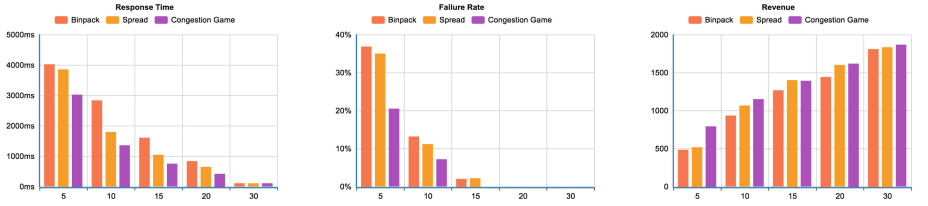

Fig. 1. Average request response time, failure rate and revenues of each strategy using different number of virtual machine instances

As shown in Fig. 1, all three strategies have relatively large response delay when only 5 virtual machine instances are used. The average response time

using Spread or Binpack is close to 4000 ms, while the average response time of Congestion Game Scheduling Strategy is 3025 ms, about 25% less. As the number of virtual machine instances increased, our strategy still has an edge compared to the other two, though the gap narrows gradually. All three strategies have roughly the same average response time when 30 virtual machine instances are used.

Congest Game Scheduling strategy also has the lowest request failure ratio compare to the other two when only 5 virtual machine are used. Its failure ratio is about 20% while Binpack has the highest failure ratio of 37%, an increase of 17%. When the number of virtual machine instances increased to 15, our strategy has no request failure at all while the other two still have about 2% failure ratio. When more than 15 virtual machine instances are used, none of the three strategies have request failure, which indicates that the virtual machine resources have satisfied the need at 5000 requests per second.

In the end, Congestion Game scheduling strategy has far more microservice revenue than the other two strategies when only 5 virtual machine instances are used. Similarly, the advantage closed out as more virtual machine instances are used. When the number reaches 30, the three strategies basically have the same revenue.

From the above results of experiments, we can see that when the number of virtual machine instances is limited, the Congestion Game scheduling strategy can optimize the performance of the system and largely outperform Binpack or Spread provided by Docker Swarm. Such gap in performance shrinks as the number of virtual machine instances increases since each instance is less pressured and more microservice containers can be run to decrease the request response time.

7 Conclusion

This paper proposes a congestion game-based resource management method under the scenario of internal resource competition among microservices in a cloud environment. Firstly, we construct a model for microservice-based applications that captures the runtime dependency among microservices that compose an application. Secondly, we propose a microservice revenue function, and then we define the problem of internal resource competition as a congestion game where each microservice is a player that tries to maximize its revenue. Finally we propose a polynomial-time algorithm to find the Nash Equilibrium of the game to solve the problem. We conducted experiments to verify the effectiveness of our approach, and results show that our proposed congestion game-based scheduling method can effectively increase the overall performance of the microservice-based application under computing resource constraints, outperforming existing strategies used by Docker Swarm.

References

1. Affetti, L.: Pwitter. https://github.com/deib-polimi/pwitter
2. Altman, E., Boulogne, T., El-Azouzi, R., Jiménez, T., Wynter, L.: A survey on networking games in telecommunications. Comput. Oper. Res. **33**(2), 286–311 (2006). https://doi.org/10.1016/j.cor.2004.06.005
3. Ardagna, D., Panicucci, B., Passacantando, M.: Generalized nash equilibria for the service provisioning problem in cloud systems. IEEE Trans. Serv. Comput. **6**(4), 429–442 (2013). https://doi.org/10.1109/TSC.2012.14
4. Baresi, L., Guinea, S., Leva, A., Quattrocchi, G.: A discrete-time feedback controller for containerized cloud applications. In: Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, pp. 217–228. ACM, New York (2016). https://doi.org/10.1145/2950290.2950328
5. Bolch, G., Greiner, S., de Meer, H., Trivedi, K.S.: Queueing Networks and Markov Chains: Modeling and Performance Evaluation with Computer Science Applications. Wiley-Interscience, New York (1998)
6. Chaisiri, S., Lee, B.S., Niyato, D.: Optimal virtual machine placement across multiple cloud providers. In: 2009 IEEE Asia-Pacific Services Computing Conference (APSCC), pp. 103–110 (2009). https://doi.org/10.1109/APSCC.2009.5394134
7. Dragoni, N., Lanese, I., Larsen, S.T., Mazzara, M., Mustafin, R., Safina, L.: Microservices: how to make your application scale. In: Petrenko, A.K., Voronkov, A. (eds.) PSI 2017. LNCS, vol. 10742, pp. 95–104. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-74313-4_8
8. Gupta, A., Garg, R.: Load balancing based task scheduling with ACO in cloud computing. In: 2017 International Conference on Computer and Applications (ICCA), pp. 174–179 (2017). https://doi.org/10.1109/COMAPP.2017.8079781
9. Hoenisch, P., Schulte, S., Dustdar, S., Venugopal, S.: Self-adaptive resource allocation for elastic process execution. In: 2013 IEEE Sixth International Conference on Cloud Computing, pp. 220–227 (2013). https://doi.org/10.1109/CLOUD.2013.126
10. Kansal, S., Kumar, H., Kaushal, S., Sangaiah, A.K.: Genetic algorithm-based cost minimization pricing model for on-demand IaaS cloud service. J. Supercomput. (2018). https://doi.org/10.1007/s11227-018-2279-8
11. Li, Y., Tang, X., Cai, W.: Dynamic bin packing for on-demand cloud resource allocation. IEEE Trans. Parallel Distrib. Syst. **27**(1), 157–170 (2016). https://doi.org/10.1109/TPDS.2015.2393868
12. Ling, Y., Yi, X., Bihuan, C., Xin, P., Wenyun, Z.: Towards runtime dynamic provision of virtual resources using feedforward and feedback control. J. Comput. Res. Dev. **52**(4), 889–897 (2015)
13. Maskin, E.: The theory of implementation in Nash equilibrium: a survey. In: Social Goals and Social Organization, pp. 173–204 (1985)
14. Mei, H., Huang, G., Zhang, L., Zhang, W.: ABC: a method of software architecture modeling in the whole lifecycle. Scientia Sinica Informationis **44**(5), 564–587 (2014)
15. Monderer, D., Shapley, L.S.: Potential games. Games Econ. Behav. **14**(1), 124–143 (1996)
16. Pahl, C., Jamshidi, P.: Microservices: a systematic mapping study. In: Proceedings of the 6th International Conference on Cloud Computing and Services Science, CLOSER 2016, vol. 1 and 2, pp. 137–146. SCITEPRESS - Science and Technology Publications, LDA, Portugal (2016). https://doi.org/10.5220/0005785501370146

17. Sheikholeslami, F., Navimipour, N.J.: Service allocation in the cloud environments using multi-objective particle swarm optimization algorithm based on crowding distance. Swarm Evol. Comput. **35**, 53–64 (2017). https://doi.org/10.1016/j.swevo.2017.02.007, http://www.sciencedirect.com/science/article/pii/S221065021730130X
18. Stolyar, A.L., Zhong, Y.: An infinite server system with general packing constraints: asymptotic optimality of a greedy randomized algorithm. In: 2013 51st Annual Allerton Conference on Communication, Control, and Computing (Allerton), pp. 575–582 (2013). https://doi.org/10.1109/Allerton.2013.6736576
19. Wang, H.M., Shi, P.C., Ding, B., Yin, G., Shi, D.X.: Online evolution of software services. Jisuanji Xuebao (Chin. J. Comput.) **34**(2), 318–328 (2011)
20. Wei, G., Vasilakos, A.V., Zheng, Y., Xiong, N.: A game-theoretic method of fair resource allocation for cloud computing services. J. Supercomput. **54**(2), 252–269 (2010). https://doi.org/10.1007/s11227-009-0318-1