Container-Based Performance Isolation for Multi-Tenant SaaS Applications in Micro-Service Architecture

Yu Wang\(^{1,2}\), Yi Sun\(^*\), Zhaowen Lin\(^{1,2}\), Jiangsong Min\(^3\)

\(^1\)Network and Information Center, Institute of Network Technology, Beijing University of Posts and Telecommunications, Beijing, 100876, China
\(^2\)National Engineering Laboratory for Mobile Network Security (No. [2013] 2685)
\(^3\)National Engineering Research Center for Supporting Software for Enterprise Internet Services

*Corresponding author: Yi Sun, email: sybupt@bupt.edu.cn

Abstract. With the development of cloud computing technology, SaaS has become a popular delivery mode in enterprise applications. Since multi-tenant can share a set of customized software applications deployed on the cloud, it significantly reduces the economic cost of providing software services of the cloud service provider. However, the sharing of cloud computing resources leads to service performance competition among tenants. Therefore, how to ensure that all tenants in the cloud have high-quality services becomes the most concerned issue for cloud service provider. To solve the above problem, our research aims at exploring an effective performance isolation mechanism based on container technology and micro-service architecture in multi-tenant scenario. This mechanism includes an SLA-oriented multi-tenant and multi-instance hybrid deployment scheme and a tenant performance isolation algorithm for microservice clusters, which can effectively reduce the performance competition among tenants, guarantee the SLA of tenants, and improve the cost-efficiency of cloud services. Experiments show that the request latency of the tenant working within their assigned quota can be reduced by 88.1% by applying the proposed performance isolation mechanism.

1. Introduction
In SaaS (Software as a Service) pattern, tenants order software services from cloud providers.[1] Their data will be centralized and stored in the cloud, and they will get a customized software provided by the cloud computing center[2]. From the customer's point of view, the SaaS model greatly reduces their technical costs[3]. However, the responsibility of software system maintenance is handed over to the cloud provider, which brings many new technical challenges. And the problem of multi-tenant performance isolation, which is how to ensure all tenants working within assigned quota are not affected by the other tenants that have exceeded their quotas in a multi-tenant scenario[4], has become a research focus in this area. Generally speaking, there are two main solutions, virtual machine based tenant performance isolation and application-level middleware based composite performance isolation.

In 2011 L. Wu et al.[5] proposed an SLA-oriented resource allocation strategy based on virtual machine. In 2012 Takeshi et al.[6] achieved performance isolation by deploying virtual machines independently for each tenant. Since each virtual machine consists of a separate operating system, there are high system resource consumption in VM-based SaaS applications. While achieving effective performance isolation, their schemas sacrificed cost-effectiveness.
In 2012 David Shue et al.[7] studied the performance isolation of multi-tenant databases, and proposed a Pisces system. Although there are differences between database and Web application, their architecture of the performance isolation mechanism still has great significance to our research.

In 2014 Krebs et al.[8] proposed an algorithm of performance isolation by monitoring tenant resource consumption in a single-instance SaaS application. They analyzed and compared the existing request resource consumption estimation algorithm, but this research only focused on single-instance applications.

In recent years, some scholars have conducted research on the construction of multi-tenant applications based on container technology. In 2016, Truyen et al. proposed the conception of deploying SaaS applications based on container technology, which provides significant guidance on performance isolation based on container[9].

In 2018 Sumit Kalra et al.[10] proposed a multi-tenant and multi-instance deployment scheme and multi-tenant management architecture based on micro-service and container technology. However, they only achieved performance isolation for some tenants with high performance requirements through separate containers. As for the remaining tenants, there is still performance competition among them.

Aiming at the scenario of multi-tenant SaaS applications under micro-service architecture, this paper proposes a comprehensive scheme of multi-tenant performance isolation based on container technology. The main contributions are as follows.

1) An SLA-oriented multi-tenant and multi-instance hybrid deployment scheme is designed to meet the performance requirements of different levels of tenants.

2) A tenant performance isolation algorithm for microservice clusters is implemented, which provides effective performance isolation for tenants and guarantees the quality of service.

2. Preliminaries

2.1 Container Technology

Compared with virtual machine, container has lower resource consumption and is easier for migration and expansion. As an operating system virtualization technology, container technology is based on the shared host operating system kernel[11]. Obviously, container technology brings new solutions for performance isolation in SaaS applications. Existing researches of container technology confirm that container technology can achieve improved cost-efficiency and lower performance isolation in comparison to VM[12][13]. Furthermore, compared with SaaS applications deployed in physical machine environment, container technology provides good performance isolation[14]. Therefore, container technology provides cloud providers with the opportunity to trade-off between performance isolation and cost-efficiency.

The idea of building SaaS applications based on container technology was proposed by Truyen et al. [9]. They proposed three container-based SaaS applications deployment strategies: shared container, no shared container, and shared container per SLA class. Shared container strategy is easy to deploy and expand, but it can not achieve performance isolation between tenants. No shared container strategy can effectively achieve the performance isolation among tenants, but it reduces the cost efficiency. Shared container per SLA class strategy provides improved security isolation between different level of tenants’ SLA. At the same time, it is more cost-effective than the second strategy, but this strategy can’t achieve performance isolation between tenants with the same SLA level.

2.2 Micro-Service Architecture

The micro-service architecture is an architectural solution for building software applications. A Large-scale application is separated into a group of loosely coupled applications according to core functions[15]. If such an architecture is applied to SaaS application, the applications can be developed and deployed independently in the form of components, which can better meet the differentiated function customization and performance requirements of tenants[16].
In the study of Sumit et al., a multi-tenant deployment scheme for micro-service based applications was proposed[10]. This model is based on micro-service architecture which can better meet the differentiated function customization and performance requirements of tenants. According to the isolation requirements and functional requirements of tenants, container-based micro-service instances are allocated to tenants. However, they failed to take full account of the differences in tenant SLA and could not use system resources effectively. Besides, the performance isolation mechanism of the tenants sharing the micro-service instance was not implemented.

3. Performance Isolation Mechanism for Multi-Tenant SaaS Applications

3.1 System Architecture

The architecture of performance isolation mechanism is shown in figure 1.

Access Layer: SaaS service gateway is used to select container clusters according to tenant SLA level. It achieves performance isolation between tenants with different SLA levels.

Control layer: Based on the performance isolation algorithm proposed in our research, access control mechanism is implemented here. The concurrent scheduler regularly collects the estimated resource consumption of requests from service layer.

Service Layer: Load balancer[17] is based on reverse proxy technology to schedule tenant requests to corresponding micro-service instances. Each micro-service instance is deployed in a docker container. Request resource demand estimation module is developed by Spring AOP technology[18].

3.2 SLA-oriented Multi-tenant and Multi-Instance Hybrid Deployment Scheme

In this study, we proposed an SLA-oriented multi-tenant and multi-instance hybrid deployment scheme. This scheme has improved the shared container per SLA class schema combined with the micro-service architecture. As shown in figure 2, the container is used to deploy micro-service instance. The resource allocation of tenants with different SLA level is achieved by allocating different resource quotas to the containers. Service providers can classify the tenants’ SLA into several levels in terms of data size, request concurrency, response delay, number of users and so on, such as standard, professional and enterprise.

This strategy provides improved security isolation between different level of tenants’ SLA. Compared with the schema proposed by Sumit et al., our schema takes full account of the differences in tenant SLA and improves cost efficiency. As shown in the figure 2, T1, T2 and T3 are divided into SLA-1 and T4 is divided into SLA-2 according to SLA level. The performance isolation between two SLA level is achieved by the different quota of containers.
3.3 Performance Isolation Algorithm for Microservice Clusters

Considering that requests from tenants with the same SLA level will still reach the same container cluster, the performance competition of these tenants still exist. When some tenants submit many requests in a certain period, their requests will consume a lot of system resource. As a result, the requests of other tenants will be interfered, and higher request latency, higher error rate and lower throughput will be observed [4].

In HTTP protocol-based SaaS applications, once users' requests are accepted, it is difficult to limit their resource consumption. Therefore, the method widely used in current research is admission control mechanism. The resource consumption of tenants is controlled by stopping or delaying requests of tenants who have exceed their quotas.

3.3.1 Resource Demand Estimation Algorithm. In a SaaS application, we need to adopt effective algorithms to estimate the resource consumptions of various request types. Brosig et al. proposed a request resource estimation algorithm based on request latency for Java web applications [19]. For each type of request $i$, the estimated resource consumption is $C_i$. Since the monitoring tool can only obtain the total resource occupation $U$ of the application, it is assumed that the response time of the request is approximately proportional to its resource demand. As a result, we can estimate the resource consumption of each request by the following formula.

$$C_i = U \cdot \frac{T_i \lambda_i}{\sum_{r=1}^{R} T_r \lambda_r}$$

(1)

Where $I$ indicates the number of request types, $T_i$ is the average delay of the request, and $\lambda_i$ is the throughput rate. The overall resource usage $U$ of the applications is:

$$U = \alpha \cdot U_c + \beta \cdot U_M + \gamma \cdot U_B$$

(2)

Where $U_c$ is the CPU usage rate, $U_M$ is the memory usage rate, and $U_B$ is the bandwidth usage rate.

3.3.2 Performance Isolation Algorithm. In this study, the algorithms are divided into tenant priority scheduling algorithm and micro-service load balance algorithm. The tenant priority scheduling algorithm is shown as Algorithm 1.

**Algorithm 1. Tenant Priority Scheduling**

**Input:** tenant_queue: tenant priority queue

**Variable:**
request_window: size of sending window  
current_num: request number in the cluster  
time_interval: interval of weight update  
request_cost: resource demand estimation function

**Function:**

```plaintext
if current_num < request_window then
    tenant_stash = new_list()

while true do
    t = priority_queue_pop(tenant_queue)
    tenant_stash.add(t)
    if t.queue.size > 0 then
        request = t.queue.dequeue()
        priority_queue_push(tenant_stash)
        send_request(request)
        t.addWeight(request_cost.getCost(request))
        break
    else
        continue

if current_time - last_time > time_interval then
    for t in tenant_queue
        t.weight = 0
    last_time = current_time
```

The system maintains a FIFO request buffer queue for each tenant. When the sending window is not full, the scheduler selects the tenant who is occupying the least resources in the cluster through the tenant priority queue. Then it dequeues a request from the selected tenant's request buffer and hand over the request to subsequent services. The priority queue is implemented by using the heapsort algorithm, which can reduce the time complexity of finding the target tenant to \( O(\log_2 n) \). This algorithm can not only achieve performance isolation, but also improve the utilization of system resources. Compared to the tenants who have already sent many requests to the cluster, a tenant that just starts sending requests is preferred due to its priority. In a scenario where a tenant has already exceeded his quota, his requests will be still be selected if there is no other tenant is going to sending requests.

The micro-service load balance is achieved by weighted round robin algorithm\[20\]. The cluster of micro-service instances is selected according to the request URI. The weight of instance is calculated by request latency. The weight calculation formula is as follows.

\[
W_i = \frac{\sum_{k=1}^{n} l_k - L_i}{\sum_{k=1}^{n} l_k}
\]  

(3)

Where \( L_i \) is the average request latency of micro-service instance \( i \). \( W_i \) is the weight of instance \( i \). The instance with low average request latency is more likely to be selected, which fully guarantees the stability of cluster performance.

4. Experiments and Analysis

4.1 Environment Setup

The hosts used in this section are configured as Intel Core i7-9750H CPU 2.60GHz 6 core 12 Logical Processors, 16 GB memory, 64-bit Ubuntu 16.04 operating system. Golang 1.12 and JAVA 1.8 is used as development languages. Mysql 5.7 is used to persist tenant data. Docker is used as Container. Spring Cloud Dalston is used as micro-service framework. In this scenario, we use Jmeter as a performance testing tool\[21\]. Each tenant has a thread pool to simulate the its users. The arrival rate of user requests is Poisson distribution.
4.2 Results and Analysis

4.2.1 SLA Level Oriented Performance Isolation. As shown in Table 1, two groups of tenants with different SLA levels were set up in the experiment. According to their SLA levels, they were allocated to two containers with different resource quota.

| Tenant | CPU (core) | Memory (GB) |
|--------|------------|-------------|
| 1      | 2          | 2           |
| 2      | 4          | 4           |

Table 1. The container resource quota of different SLA level

| Tenant | Average Latency (ms) | Latency Standard Deviation (ms) | Throughput (sec⁻¹) |
|--------|----------------------|---------------------------------|--------------------|
| 1      | 4216                 | 773.76                          | 99.9               |
| 2      | 3010                 | 812.70                          | 139.2              |
| Sum    | 3505                 | 993.40                          | 236.0              |

Table 2. Experiment result of SLA oriented performance isolation

The results in Table 2 show that in scenario of same concurrency the tenants with higher SLA level can achieve lower request latency and higher throughput. The schema proposed in our research can provide different performance services to the tenants with different SLA levels.

4.2.2 Performance Isolation Algorithm. This experiment set two groups of tenants with the same SLA level. Two micro-services were selected, and each group of micro-services deployed two instances. All containers are configured with 4 core CPU and 4 GB memory.

Two tenants initially accessed the system concurrently with 200 threads. After 20 seconds, the concurrency of tenant2 increased gradually to 2200, simulating an abrupt increase of workload.

Firstly, we tested the scenario without tenant performance isolation algorithm.

| Tenant | Average Latency (ms) | Latency Standard Deviation (ms) | Throughput (sec⁻¹) |
|--------|----------------------|---------------------------------|--------------------|
| 1      | 6602                 | 3718.14                         | 127.4              |
| 2      | 7707                 | 3824.36                         | 175.3              |
| Sum    | 7243                 | 3819.26                         | 302.2              |

Throughput. The two tenants’ throughputs were basically the same in the initial stage. When concurrency of tenant2 started to increase, the throughput of tenant1 decreased dramatically and began to jitter dramatically, which resulted in a large fluctuation of performance perceived by users.

Request latency. The request latency of two tenants are almost the same in the whole process. When concurrency of tenant2 start to increase, the standard deviation of latency reached 3718.14ms. It is obvious that the work load of tenant2 caused performance interference to tenant1.
Then we tested the performance isolation algorithm implemented in this paper.

### Table 4. Performance comparison of performance isolation algorithm

| Tenant | Average Latency (ms) | Latency Standard Deviation (ms) | Throughput (sec⁻¹) |
|--------|----------------------|---------------------------------|--------------------|
| 1      | 788                  | 238.17                          | 161.2              |
| 2      | 8240                 | 6483.66                         | 168.3              |
| Sum    | 4627                 | 5962.52                         | 326.7              |

Throughput. The throughputs of two tenants are basically maintained at the same level. And the stability of throughput has also been greatly improved.

Request latency. After using the tenant isolation algorithm, the average request latency of tenant 1 decreased by 88.1%. And the standard deviation of tenant 1’s request latency decreased too. When the concurrency of tenant 2 increased, the request latency of tenant 1 remained at the previous level, and did not increase together with tenant 2.

5. Summary

This study proposed a comprehensive mechanism of multi-tenant performance isolation based on container technology, which implements the tenant performance isolation algorithm in the micro-service cluster scenario and meets the performance requirements of tenants with different SLA levels. Experiments show that the multi-tenant performance isolation mechanism proposed in this
paper can achieve effective tenant performance isolation.

Acknowledgment
This work is supported by the National Natural Science Foundation of China (Grant no. 61601041), the Fundamental Research Funds for the Central Universities(2019PTB-003).

References
[1] Kwok, T., Nguyen, T., Lam, L. (2008). A software as a service with multi-tenancy support for an electronic contract management application. In: 2008 IEEE International Conference on Services Computing. Hawaii, USA. pp. 179-186.
[2] Bezemer, C. P., Zaidman, A. (2010) Multi-tenant SaaS applications: maintenance dream or nightmare? In: Proceedings of the Joint ERCIM Workshop on Software Evolution and International Workshop on Principles of Software Evolution Antwerp, Belgium. New York, USA. pp. 88-92.
[3] Weissman, C. D., Bobrowski, S. (2009). The design of the force. com multitenant internet application development platform. In: 2009 ACM SIGMOD International Conference on Management of Data. Rhode Island, USA. pp. 889-896.
[4] Krebs, R., Momm, C., Kounev, S. (2014) Metrics and techniques for quantifying performance isolation in cloud environments. Science of Computer Programming, 90; 116-134.
[5] Wu, L., Garg, S. K., Buyya, R. (2011) SLA-based resource allocation for software as a service provider (SaaS) in cloud computing environments. In: 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing. California, USA. pp. 195-204.
[6] Takahashi, T., Blanc, G., Kadobayashi, Y., Fall, D., Hazeyama, H., Matsuo, S. I. (2012) Enabling secure multitenancy in cloud computing: Challenges and approaches. In: 2nd Baltic Congress on Future Internet Communications. Vilnius, Lithuania. pp. 72-79.
[7] Shue, D., Freedman, M. J., Shaikh, A. (2012) Performance isolation and fairness for multi-tenant cloud storage. In: 10th USENIX Symposium on Operating Systems Design and Implementation. California, USA. pp. 349-362.
[8] Krebs, R., Spinner, S., Ahmed, N., Kounev, S. (2014) Resource usage control in multi-tenant applications. In: 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing. Illinois, USA pp. 122-131.
[9] Truyen, E., Van Landuyt, D., Reniers, V., Rafique, A., Lagaise, B., Joosen, W. (2016) Towards a container-based architecture for multi-tenant SaaS applications. In: 15th International Workshop on Adaptive and Reflective Middleware. Trento, Italy. pp. 6:1-6:6.
[10] Kalra, S., Prabhatkar, T. V. (2018) Towards Dynamic Tenant Management for Microservice based Multi-Tenant SaaS Applications. In: 11th Innovations in Software Engineering Conference. Hyderabad, India. pp. 12:1–12:5.
[11] Xavier, M. G., Neves, M. V., Rossi, F. D., Ferreto, T. C., Lange, T., De Rose, C. A. (2013). Performance evaluation of container-based virtualization for high performance computing environments. In: 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing. Belfast, UK. pp. 233-240.
[12] Dua, R., Raja, A. R., Kakadia, D. (2014) Virtualization vs containerization to support PaaS. In: 2014 IEEE International Conference on Cloud Engineering. Massachusetts, USA. pp. 610-614.
[13] Zhang, M., Marino, D., Efthathopoulos, P. (2015). Harbormaster: Policy enforcement for containers. In: IEEE 7th International Conference on Cloud Computing Technology and Science. BC, Canada. pp. 355-362.
[14] Xavier, M. G., De Oliveira, I. C., Rossi, F. D., Dos Passos, R. D., Matteussi, K. J., De Rose, C. A. (2015) A performance isolation analysis of disk-intensive workloads on container-based clouds. In: 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing. Turku, Finland. pp. 253-260.

[15] Nadareishvili, I., Mitra, R., McLarty, M., Amundsen, M. (2016) Understanding microservices. In: MacDonald, B., Bauer, H. (Eds.), Microservice architecture: aligning principles, practices, and culture. O'Reilly Media Inc., California, USA. pp. 3-21

[16] Villamizar, M., Garcés, O., Castro, H., Verano, M., Salamanca, L., Casallas, R., Gil, S. (2015). Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud. In: 10th Computing Colombian Conference. Bogota, Colombia. pp. 583-590.

[17] Namiot, D., Sneps-Sneppe, M. (2014). On micro-services architecture. International Journal of Open Information Technologies, 2: 24-27.

[18] Kumar, A. (2016). Applying separation of concern for developing softwares using aspect-oriented programming concepts. Procedia Computer Science, 85: 906-914.

[19] Brosig, F., Kounev, S., Krogmann, K. (2009) Automated extraction of palladio component models from running enterprise Java applications. In: 4th International ICST Conference on Performance Evaluation Methodologies and Tools. Pisa, Italy. pp. 1-10.

[20] Shreedhar, M., Varghese, G. (1996). Efficient fair queuing using deficit round-robin. IEEE/ACM Transactions on Networking, 4: 375-385.

[21] Wu, Q., Wang, Y. (2010). Performance testing and optimization of J2EE-based web applications. In: 2nd International Workshop on Education Technology and Computer Science. Wuhan, China. pp. 681-683.