Research Article

A Microservice Resilience Deployment Mechanism Based on Diversity

Hang Yu, Xiulei Wang, Changyou Xing, and Bo Xu

Command & Control Engineering College, Army Engineering University of PLA, Nanjing 210007, Jiangsu, China

Correspondence should be addressed to Xiulei Wang; xiuleiwang1988@126.com

Received 25 October 2021; Revised 18 February 2022; Accepted 11 May 2022; Published 1 June 2022

Academic Editor: Juhriyansyah Dalle

Copyright © 2022 Hang Yu et al. This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

The microservice architecture has many advantages, such as technology heterogeneity, isolation, scalability, simple deployment, and convenient optimization. These advantages are important as we can use diversity and redundancy to improve the resilience of software system. It is necessary to study the method of improving the resilience of software system by diversity implementation and redundant deployment of software core components based on microservice framework. How to optimize the diversity deployment of microservices is a key problem to maximize system resilience and make full use of resources. To solve this problem, an efficient microservice diversity deployment mechanism is proposed in this paper. Firstly, we creatively defined a load balancing indicator and a diversity indicator. Based on this, a diversified microservices deployment model is established to maximize the resilience and the resource utilization of the system. Secondly, combined with load balancing, a microservice deployment algorithm based on load balance and diversity is proposed, which reduces the service’s dependence on the underlying mirror by enriching diversity and avoids the overreliance of microservices on a single node through decentralized deployment, while taking into account load balancing. Finally, we conduct experiments to evaluate the performance of the proposed algorithm. The results demonstrate that the proposed algorithm outperforms other algorithms.

1. Introduction

The ability of the system to maintain the continuous running of services and ensure the completion of tasks is called system resilience [1]. In order to enhance system resilience, service providers usually adopt the idea of diversity and redundancy to deploy software [2]. Typically, the traditional monomer architecture requires that the software must be deployed as a whole, which makes diversity and redundancy implementing technologies costly [3]. In fact, a large number of user requests can be completed only by some functional modules in the software system and only a very small number of requests need all the functional modules of the software system to complete the response.

Microservice architecture [4] decouples software system functions into multiple microservice components and the microservice components can be developed by using container mirroring [5], which greatly improves the reusability of microservice components. The redundant deployment under microservice architecture is more flexible and efficient than monomer architecture. But at the same time, high reusability also means that multiple component containers deployed based on the same mirror are homogeneous. The studies have shown that security vulnerabilities are prevalent in container mirroring [6]. So once an attacker finds vulnerability in a microservice component of the target system, the attacker can then repeatedly exploit the vulnerability to successfully attack containers that use the same mirror, severely affecting the system’s ability to provide services.

Currently, researchers generally agree that diversity [7] methods such as heterogeneous are an effective way to solve homogeneity problems [2]. For systems with diversity, attackers can succeed in an attack if and only if they successfully obtain the vulnerabilities of each implementation version, which makes the attack very difficult. At the same time, as the system may jump in multiple versions of implementations each time when it provides services to users, even if the attacker succeeds temporarily, this attack is
difficult to sustain. As a result, the more versions there are, the higher the diversity metrics are, and the difficult it is for an attacker’s attack to succeed.

This paper aims to solve the problem of software system diversity deployment under microservice architecture by using virtualization technology, which eliminates the impact of infrastructure platform inconsistency and realizes the deployment of heterogeneous containers on heterogeneous infrastructure. By further enriching the diversity of software systems, attackers cannot take advantage of the homogeneity of containers to destroy and strike software systems of microservice architectures on a large scale, thereby ensuring the resilience of software systems.

The main contributions of this paper are as follows:

(1) A load balancing indicator and a diversity indicator is creatively defined. Based on this, a diversified microservice deployment model is established to maximize the resilience of the system.

(2) A microservice deployment algorithm based on load balancing and diversity is proposed, which reduces the service’s dependence on the underlying mirror by enriching diversity and avoids the overreliance of microservices on a single node through decentralized deployment, while taking into account load balancing.

(3) Finally, the experiments are designed to evaluate the performance of the proposed algorithm. The results demonstrate that the proposed algorithm outperforms other comparison algorithms.

The remainder of this paper is organized as follows. In Section 2, an overview of the related work is provided. In Section 3, the typical scenarios are shown and the problem is analyzed. In Section 4, the load balance dictator and diversity dictator are given. Based on these two indicators, the optimization model of the problem is given which is proved to be NP hard. In Section 5, a heuristic algorithm, LB-Diversity, is proposed and the performance of this algorithm is shown in Section 6. The application scenario is discussed in Section 7, followed by conclusions of this study in Section 8 and the acknowledgement in Section 9.

2. Related Work

At present, researchers have proposed some practical solutions to the service deployment problem on microservice architecture, but these researches mainly focus on efficiency. Wan [8] proposed a resource allocation algorithm EPTA (EC Placement and Task Assignment Algorithm) using the hierarchical characteristics of Docker containers. This algorithm significantly reduces deployment costs by balancing wake-up costs, installation costs, and communication costs. Fan [9] proposed scheduling algorithms which combination of latency, reliability, and load balance-aware to determine container-based microservice deployments in edge computing. By improving and optimizing the Particle Swarm Optimization (PSO), a better LRLBAS algorithm is proposed.

As the software system is implemented based on microservice architecture, which has the advantage of loose coupling, it can be divided into several independent parts and each part works independently. Combined with container technology, individual parts can be packaged into different instances as components to build a complete software system. This provides great potential for resilience enhancement of software systems. Based on this, the application of redundancy technology can be realized to improve the availability of the system.

This redundancy strategy has been widely used in the deployment of Service Function Chain (SFC). Hmaity et al. [10] proposed three redundancy protection schemes in the redundancy protection mechanism of SFC, namely, (1) end-to-end redundancy, (2) virtual node redundancy protection, and (3) virtual link redundancy protection. Although these three schemes can protect nodes and links in a targeted manner, such extensive protection will lead to waste of resources, since the author did not study the number of redundancy. At the same time, the virtual link protection scheme proposed by the author may cause both the primary virtual link and the backup virtual link to be mapped to the same physical link, and this may result in weaker resilience. Han [11] conducted performance analysis experiments for workloads to obtain refined resource requirements and performed microservice deployment based on a greedy heuristic algorithm. The innovation of the algorithm lies in the use of refined resource requirements derived from the analysis results to consider application performance, but the algorithm only considers the case of a single service chain. Joseph [12] proposed a novel heuristic method, IntMA (Interaction-aware Microservice Allocation), to deploy microservices in an interactively aware manner with the help of the interaction information obtained from the interaction graph. However, none of the above work on deployment issues has been discussed based on diversity.

Due to the extensive use of redundancy to ensure availability, most of the microservices (components) in the software system of the microservice architecture are homogeneous. Systems built on Docker container technology introduce the homogeneity brought by using the same base image, which leads to a large number of shared vulnerabilities being introduced into the system [6, 13]. Therefore, the system will be vulnerable to multi-step attacks [14, 15] that exploit the same vulnerability in multiple microservices. Diversity is considered as the most effective means of mitigating the shared vulnerability problem. After the introduction of diversity, how to deploy multi-version microservice components in a server cluster to make the system more secure and efficient has become a problem that researchers need to pay attention to.

At present, some research work is carried out in combination with diversity. Taking into account the security risks that may be caused by the isomorphism of operating systems, Zhang [16] proposed a security deployment strategy for virtual machines based on diversity by using heterogeneous operating systems, which effectively reduced the attacker’s unit attack revenue, but since only the secure deployment of virtual machines is considered, resource
constraints and other issues are not considered. Xie [17] proposed a deployment method based on the heterogeneity of nodes, aiming to ensure the heterogeneity of nodes during redundancy backup and remapping. Torkura [18] applied Moving Target Defense (MTD) technology to the microservice architecture for the first time. Using the MTD mechanism, risk analysis was performed on microservices to detect and prioritize vulnerabilities, and then automatic code generation technology was used to convert the programming language and container images of microservices. Minimizing shared vulnerabilities through diversified deployment of security strategies brings uncertainty to attackers and reduced the attack ability of the system under the microservice architecture, thereby overcoming the security risks brought by homogeneous microservices. Alleg [2] proposed a placement solution of Service Function Chains (SFC) modeled as a Mixed Integer Linear Program (MILP) designed to meet the availability level of the target SFC and reduce the cost escalation due to diversity and redundancy. This approach can allocate fewer resources to backup instances while avoiding service disruption. Qing [19] proposed an adaptive spatiotemporal diversity joint scheduling strategy, SASTD (Self Adaptive Spatiotemporal Diversity). The strategy combines spatiotemporal diversity and coarse-grained intrusion detection, overcomes the defects of high defense cost and poor effect under a single strategy, and improves the defense capability of the system at a lower cost. However, the article does not discuss the impact of resource consumption, nor does it take into account the deployment location.

In addition to this, researchers have proposed various security strategies such as Address Space Layout Randomization (ASLR) [20], instruction-level [21], and basic-block-level [22] transformations to address such challenges. The deployment problem of services under the microservice architecture can be understood as an optimization problem under certain constraints. This problem is an NP-hard problem. In terms of algorithm selection for solving the deployment scheme, researchers usually choose genetic algorithm [23], simulated annealing algorithm [24], list search algorithm [25] and other heuristic algorithms. In addition, the Viterbi algorithm [17, 26] has also been mentioned by researchers.

We provide a comparison of the work mentioned in this section with ours in Table 1. As shown in Table 1, our proposed method considers both diversity and redundancy. Compared with other method, more factors are considered during optimization. In addition, the Docker we use is finer in granularity and better cloud-native.

### 3. Problem Statement

Based on existing researches, it can be seen that the resilience and damage resistance of software system can be effectively improved through diversity technology, but under the existing monolithic architecture, the realization of software system diversity is facing great challenges. The emergence of microservice technology provides a new way to solve this problem; this paper mainly studies how to establish a diversified microservice deployment model under the constraints of resources, in order to maximize the resilience of the system.

### 3.1. Failure Scenario

#### 3.1.1. Node Failure

Node failure is an unavoidable problem, and when a node in a cluster fails to work, all containers deployed on that node lose the ability to provide services, which will greatly affect the associated microservice delivery capabilities.

1. If the load of the node is too high, the failure will lead to a significant decrease in the overall service capacity of the system.
2. If the instances of a single microservice are centralized deployed on a node, it will cause the service performance to decrease and become a bottleneck of the whole system.

#### 3.1.2. Shared Vulnerabilities Problem

When using containers to build a software system, for some microservice, if it uses the same basic image in a large number, and once the image has a high-risk vulnerability and it is exploited by the attacker, the attacker can easily use this vulnerability to launch repeated attacks on such containers in the system, making them lose service capability, so as to greatly reduce the performance of the microservice.

### 3.2. Problem Analysis

To solve the node failure problem described in Section 3.1.1, there are two ideas. The first one requires the system to have good load balancing performance, which can avoid the load being too concentrated on a certain node. The second one requires every microservice to be deployed as evenly as possible among nodes, which is also referred to as the diversity of microservices deployment on nodes. Two illustrative examples are shown as follows.

**Example 1.** When a large number of instances of service $X$ are concentrated on node $N$, once node $N$ fails to work due to failure or attack, the service capability of service $X$ will suffer huge losses, thus affecting the overall service performance.

**Example 2.** When node $N$ hosts a large number of service instances of the system, once node $N$ fails to work due to failures or attacks, the overall service capability of the system will be severely degraded.

To solve the sharing vulnerability problem in Section 3.1.2, it requires any microservice in the system to be deployed with multi-version instances to take full advantage of diversity. At the same time, it is necessary to achieve the balance of the requests of the same microservice between different versions, that is, the diversity of deployment versions of microservices, so as to avoid a single service being too dependent on a certain version implementation, resulting in attacks caused by the mirror vulnerability of this
version. The performance of the next microservice suffers a serious loss. An illustrative example is shown as follows.

**Example 3.** Suppose a total of 20 instances of service X are deployed in the system, 18 of which use the same Alpine-based container template, and the other two instances use other container templates to start. Once the Alpine container template has fatal flaws, it is exploited by attackers to carry out attacks. Through lateral infection, an attacker can cause service X to lose 90% of its service capacity.

According to the analysis, the problem can be parsed and is shown in Figure 1.

### 4. Indicator Design and System Model

**4.1. Symbols.** According to the analysis in Section 3.2, node failure can be solved by load balancing and shared vulnerability problems can be mitigated by version diversity. In order to analysis and model the problem, the relevant indicators, optimization objective, and constraints of the problem should be defined first.

The symbols required for modeling in this paper are shown in Table 2.

<table>
<thead>
<tr>
<th>Research work</th>
<th>Solution</th>
<th>Instance unit</th>
<th>Optimization values</th>
<th>Redundancy</th>
<th>Diversity</th>
</tr>
</thead>
<tbody>
<tr>
<td>[9]</td>
<td>Heuristic (PSO)</td>
<td>Container</td>
<td>Performance, load balance, availability level, latency</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>[10]</td>
<td>ILP</td>
<td>VNF</td>
<td>Latency, cost</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>[12]</td>
<td>Heuristic (IntMA)</td>
<td>Container</td>
<td>Internode communications, latency</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>[16]</td>
<td>Heuristic</td>
<td>VM</td>
<td>Diversity level</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>[17]</td>
<td>Heuristic</td>
<td>VNF</td>
<td>Cost, diversity level</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>[18]</td>
<td>—</td>
<td>Container</td>
<td>Security risk level</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>[2]</td>
<td>—</td>
<td>VNF</td>
<td>Availability level, cost</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>[19]</td>
<td>SASTD (self-adaptive spatiotemporal diversity)</td>
<td>COTS (commercial off-the-shelf)</td>
<td>Diversity level, security risk level</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Our work</td>
<td>Heuristic (LB-Diversity)</td>
<td>Docker</td>
<td>Performance, availability level, load balance, diversity level, cost</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

The symbols required for modeling in this paper are shown in Table 2.

\[
H_{i,j}^{x} = \begin{cases} 
1, & \text{if } MS_{x,i} \text{ is deployed on node } N_j, \\
0, & \text{others,}
\end{cases}
\]

Therefore, for a single deployment node in cluster, the node CPU resource usage is the sum of CPU resource usages for all instances deployed on that node, which can be expressed as shown in formula (3).

\[
L_{j}^{CPU} = \sum_{x=1}^{s} \sum_{i=1}^{n} H_{i,j}^{x} (h_{i,k}^{x} R_{x,k}^{CPU}).
\]

Similarly, the node memory resource usage is equal to the sum of memory resource usages for all instances deployed on the node and can be expressed as shown in formula (4).

\[
L_{j}^{MEM} = \sum_{x=1}^{s} \sum_{i=1}^{n} H_{i,j}^{x} (h_{i,k}^{x} R_{x,k}^{MEM}).
\]

The node load rate can then be written as the ratio of the current node load to the total number of node resources, as shown in formulas (5) and (6).

\[
u_{j}^{CPU} = \frac{L_{j}^{CPU}}{R_{N_j}^{CPU}}.
\]

\[
u_{j}^{MEM} = \frac{L_{j}^{MEM}}{R_{N_j}^{MEM}}.
\]

In order to eliminate the influence of inconsistency in the scale, the calculation results are normalized by formula (7) and the normalized deployment nodes CPU and memory resource load balance degree are obtained as \(U_{j}^{CPU}, U_{j}^{MEM}\).

\[
U = \frac{u - \text{Min}(u)}{\text{Max}(u) - \text{Min}(u)}.
\]

Define the node resource load standard deviation \(\sigma_{CPU}^{CPU}\) and \(\sigma_{MEM}^{CPU}\), which represent the difference between the resource load rate on one deployment node and the average of
all deployment nodes, as shown in formulas (8) and (9), where $U_{\text{cpu}}^{j}$ and $U_{\text{mem}}^{j}$ represent the CPU and memory resource load of all deployed nodes after normalization, respectively.

$$\sigma_{\text{cpu}} = \sqrt{\frac{1}{N} \sum_{j=1}^{N} (U_{\text{cpu}}^{j} - U_{\text{avg}}^{\text{cpu}})^2}, \quad (8)$$

$$\sigma_{\text{mem}} = \sqrt{\frac{1}{N} \sum_{j=1}^{N} (U_{\text{mem}}^{j} - U_{\text{avg}}^{\text{mem}})^2}. \quad (9)$$

Because of the large difference between node CPU and memory resource data base scale, the direct addition of $\sigma_{\text{cpu}}$ and $\sigma_{\text{mem}}$ cannot effectively reflect the load balancing level of the cluster as a whole; in order to eliminate the impact of the measurement scale and scale between the two indicators, this paper uses the coefficient of variation for calculation. The total load deviation degree of the cluster can be measured by the sum of the coefficients of variation of the node CPU and memory resources, and the smaller the value of the indicator, the smaller the load difference between the nodes, and the better the load balancing effect. To align the cluster load balancing metric with the increase in load balancing performance, the load balancing metric is defined as a negative number of load deviations for the total load of the cluster, which can be expressed in formula (10).

$$E_{LB} = -\left( \frac{\sigma_{\text{cpu}}}{U_{\text{avg}}^{\text{cpu}}} + \frac{\sigma_{\text{mem}}}{U_{\text{avg}}^{\text{mem}}} \right). \quad (10)$$

4.3. Diversity Indicator. Diversity is reflected in two main aspects: the deployment node diversity and the version diversity. In order to measure the richness of diversity, this paper refers to the relevant work [6, 14]. According to the measure of biodiversity, a measure of systematic diversity is proposed using the Shannon formula.

4.3.1. Node Diversity. For the diversity of deployment nodes for a single microservice, the mathematical formula can be expressed as

$$D_{1}^{x} = -\sum_{j=1}^{n_{x}} p_{j}^{x}\log_{2}p_{j}^{x}. \quad (11)$$

where $D_{1}^{x}$ represents the node diversity metric for the microservice, $n_{x}$ represents the number of cluster deployment nodes, and $p_{j}^{x}$ represents the proportion of the total number of instances deployed on node $N_{j}$ by microservice $MS_{x}$.

4.3.2. Version Diversity. Similarly, for the diversity of deployment versions of a single microservice, the mathematical formula can be expressed as

$$D_{2}^{k} = -\sum_{k=1}^{m_{x}} \frac{q_{k}^{x}\log_{2}q_{k}^{x}}, \quad (12)$$

where $D_{2}^{k}$ represents the node diversity metric for the microservice, $m_{x}$ represents the number of implemented versions of microservices $MS_{x}$, and $q_{k}^{x}$ represents the ratio which indicates the number of deployments of version $k$ to the total number of microservice $MS_{x}$ deployments.

In summary, the diversity indicator can be summarized as shown in formula (13). Single microservice diversity metric is contributed by the deployment node diversity metric for microservices and the deployment version diversity metric, which can be represented as $(D_{1}^{x} + D_{2}^{x})/2$. Overall diversity indicator $E_{D}$ is the mean of the degree of diversity of all microservices.

$$E_{D} = \frac{\sum_{x=1}^{s}(D_{1}^{x} + D_{2}^{x})/2}{s}. \quad (13)$$

4.4. The Goal of Optimization. Load balance degree $E_{LB}$ and diversity degree $E_{D}$ both play an important role in solving the problems discussed in this paper; i.e., load balancing and diversity can be considered as two important means of solving the problems raised and enhancing system resilience in this paper, so this paper combines these two factors into one resilience indicator $E$, as shown in formula (14).

$$E = \alpha E_{LB} + (1 - \alpha)E_{D}. \quad (14)$$

Therefore, the problem raised in this paper can be modeled as an optimization problem. Under certain constraints, the aim is to get Max($E$), which is the maximum value of the resilience index $E$. The constraints of the problem are described below.
Table 2: Symbols frequently used in this paper.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s$</td>
<td>The number of types of system microservices</td>
</tr>
<tr>
<td>$n_{x}$</td>
<td>Number of $x$-th microservice deployment instances</td>
</tr>
<tr>
<td>$m_{s}$</td>
<td>Number of versions of the $x$-th microservice implementation</td>
</tr>
<tr>
<td>$n_{N_j}$</td>
<td>The number of cluster deployment nodes</td>
</tr>
<tr>
<td>$MS_{x,i-j}$</td>
<td>The $i$-th instance of the $x$-th microservice ($i \leq n_x$)</td>
</tr>
<tr>
<td>$N_j$</td>
<td>The $j$-th deployment node in the cluster ($j \leq n_{N_j}$)</td>
</tr>
<tr>
<td>$h_{x,i-j}^{k}$</td>
<td>Bool value, indicating whether the $i$-th instance of the microservice $MS_{x}$ uses the $k$-th mirror</td>
</tr>
<tr>
<td>$H_{x,i-j}^{k}$</td>
<td>Bool value, indicating whether the $i$-th instance of the microservice $MS_{x}$ is deployed at the $j$-th deployment node</td>
</tr>
<tr>
<td>$p_j^x$</td>
<td>The ratio of the number of instances belonging to microservice $MS_{x}$ deployed on $N_j$ to the total number of microservice $MS_{x}$ deployments</td>
</tr>
<tr>
<td>$q_k^x$</td>
<td>The ratio of the number of version $k$ instances of microservice $MS_{x}$ deployed on $N_j$ to the total number of microservice $MS_{x}$ deployments</td>
</tr>
<tr>
<td>$D_z^x$</td>
<td>Node diversity indicator for microservice $MS_{x}$</td>
</tr>
<tr>
<td>$D_z^s$</td>
<td>Version diversity indicator for microservice $MS_{x}$</td>
</tr>
<tr>
<td>$R_{x,k}^{CPU}$</td>
<td>CPU resources required to deploy the $k$-th version instance of the microservice $MS_{x}$</td>
</tr>
<tr>
<td>$R_{x,k}^{MEM}$</td>
<td>Memory resources required to deploy the $k$-th version instance of the microservice $MS_{x}$</td>
</tr>
<tr>
<td>$R_{N_j}^{CPU}$</td>
<td>The CPU resource of the deployment node ($x \leq n$)</td>
</tr>
<tr>
<td>$R_{N_j}^{MEM}$</td>
<td>The memory resource of the deployment node $N_j$</td>
</tr>
<tr>
<td>$L_{CPU,j}^x$</td>
<td>The CPU load of the deployment node $N_j$</td>
</tr>
<tr>
<td>$L_{MEM,j}^x$</td>
<td>The memory load of the deployment node $N_j$</td>
</tr>
<tr>
<td>$u_{CPU,j}^x$</td>
<td>The CPU load rate of the deployment node $N_j$</td>
</tr>
<tr>
<td>$u_{MEM,j}^x$</td>
<td>The memory load rate of the deployment node $N_j$</td>
</tr>
<tr>
<td>$U_{CPU,j}$</td>
<td>The normalized CPU load rate of the deployment node $N_j$</td>
</tr>
<tr>
<td>$U_{MEM,j}$</td>
<td>The normalized memory load rate of the deployment node $N_j$</td>
</tr>
<tr>
<td>$U_{CPU,avg}$</td>
<td>Average load rate of CPU resource for all deployment nodes</td>
</tr>
<tr>
<td>$U_{MEM,avg}$</td>
<td>Average load rate of memory resource for all deployment nodes</td>
</tr>
<tr>
<td>$\sigma_{CPU}$</td>
<td>The standard deviation of CPU load for the deployment node $N_j$</td>
</tr>
<tr>
<td>$\sigma_{MEM}$</td>
<td>The standard deviation of memory load for the deployment node $N_j$</td>
</tr>
</tbody>
</table>

4.5. Constraints. For any one microservice instance $MS_{x,i-j}$, it can only be deployed on one node, so constraint (15) can be obtained.

$$\sum_{j=1}^{n_{N_j}} H_{i,j}^{x} = 1, \; \forall x, \forall i.$$ (15)

Similarly, microservice instance $MS_{x,i-j}$ can only use one version mirror; that is,

$$\sum_{k=1}^{m_{s}} h_{i,j}^{x,k} = 1, \; \forall x, \forall i.$$ (16)

In addition, the ratio $p_j^x$ of the number of instances deployed on $N_j$ to total deployment number of microservice $MS_{x}$, as well as the ratio $q_k^x$ of the number of instances deployed with version $k$ to total number of microservice $MS_{x}$, also has constraints (17) and (18).

$$\sum_{j=1}^{n_{N_j}} p_j^x = 1, \; 0 \leq p_j^x \leq 1, \forall x,$$ (17)

$$\sum_{k=1}^{m_{s}} q_k^x = 1, \; 0 \leq q_k^x \leq 1 \forall x.$$ (18)

From a resource perspective, both CPU and memory resource constraints need to be met during deployment, and their constraints can be expressed as follows:

$$\sum_{x=1}^{s} \sum_{i=1}^{n_x} \sum_{k=1}^{m_{s}} H_{i,j}^{x}(h_{i,j}^{x} R_{x,k}^{CPU}) \leq R_{N_j}^{CPU}, \; \forall j,$$ (19)

$$\sum_{x=1}^{s} \sum_{i=1}^{n_x} \sum_{k=1}^{m_{s}} H_{i,j}^{x}(h_{i,j}^{x} R_{x,k}^{MEM}) \leq R_{N_j}^{MEM}, \; \forall j.$$ (20)

Formulas (19) and (20) indicate that the sum of the CPU and memory resources consumed by containers deployed on the node does not exceed the maximum resources owned by the node.

Formulas (15)–(20) form the set of constraints for model optimization problem solving in this paper.

5. Algorithm

5.1. Algorithmic Analysis. According to the modeling process, it can be learned that this problem is a problem about the optimal combination of service version and deployment location, which is a typical NP difficult problem. For the
solution of such problems, the academic community usually uses heuristic algorithms to solve, such as GA (genetic algorithms), PSO (particle swarm optimization), etc. However, there are common problems of such methods like high time complexity and long convergence time. Industry, on the other hand, has opted for performance compromises, not maximizing the target function, but simply using strategy-based solutions such as Round Robin and Random to get shorter response delays.

The problem in this paper consists of two parts, namely, load balancing and diversity. The Industry’s Round Robin strategy is a very mature solution to the problem of load balancing, whereby the next node is selected for deployment at each deployment location in a certain order of nodes. Although such a strategy is easy to implement, there is a problem of grainy. Because of the differences in resource consumption between different types and versions of microservices, load balancing is not precise. Decisions should be made based on the resource usage of the services that the node has deployed.

In the optimization of diversity, according to formula (13), we can know that if we want to get the maximum value of diversity indicator $ED_i$, we need to meet each $D_i$ take the largest. However, by formulas (1) and (2), it can be found that there is a consistent description of the node diversity indicator $D_1$ and the version diversity indicator $D_2$, as follows:

$$D = - \sum_{i=1}^{d} p_i \log_2 p_i. \tag{21}$$

According to Shannon’s first theorem, the indicator $D$ can get the extreme value when and only if $p_1 = p_2 = \ldots = p_d = 1/d$, and the maximum value is $\log d$. Thus, when deploying based on redundancy and diversity, there should be as two “even distributions” as possible for each microservice:

1. For any microservice $MS$, the number of microservice instances deployed is distributed as evenly as possible among the nodes, thus maximizing the node diversity indicator $D_1$.
2. For any microservice $MS$, the number of microservice instances deployed is distributed as evenly as possible among the versions, thus maximizing the version diversity indicator $D_2$.

5.2. Description of the Algorithm. According to the analysis of Section 4.1, in order to solve the problems, a microservice deployment algorithm based on load balancing and diversity LB-Diversity is proposed. The algorithm realizes the average distribution of microservices on the version through a polling-like mechanism, then uses the current node deployment information for node filtering, realizes the average distribution of services between nodes, and selects the deployment nodes according to the load conditions of the filter results, so as to achieve better load balancing.

The pseudocode of the deployment algorithm is given in Algorithm 1, and its input includes microservice information, node information, microservice list, service version information, node list, number of deployments, output as node deployment results, and microservice deployment results.

In Algorithm 1, line 4 indicates that all versions of the microservices $ms$ are reordered based on resource conditions. Line 6 indicates that, with the same number of deployments, deployment takes precedence over less resource-intensive versions for deployment. Lines 7–11 indicate that when you select deployment nodes, you first filter out nodes with fewer current service deployments based on the current deployment status and then find the nodes with the lowest load rate and resources that meet the deployment needs as the final deployment node. In this algorithm, the total number of microservices is $s$, each type of microservice contains $m_i$ based on different mirrors generated version, and each type of microservices needs to deploy $m_i$ instances, there are $n_N$ deployment nodes in the cluster; then the time complexity of the algorithm can be expressed as

$$O\left( s \left[ O(m_x \log m_x) + n_{\text{deploy}} (n_N O (n_N \log n_N)) \right] \right). \tag{22}$$

Because of microservices type limit, $O (m_x \log (m_x))$ is obviously less than $n_{\text{deploy}} \times (n_N O (n_N \log (n_N)))$, so formula (22) can also be written as shown in formula (23):

$$O\left( sn_{\text{deploy}} n_N^2 \log n_N \right). \tag{23}$$

6. Experiment

First of all, the paper makes a simple implementation of the system described in the model and verifies its partial performance. In order to better evaluate the performance of the model, the paper simulates the situation under more complex conditions.

6.1. Implementation. In a software system network based on a Software Definition Network (SDN), deployment control functions can be integrated into the SDN controller, making it the control center of the entire software system. As shown in Figure 2, the SDN controller is able to customize the deployment policy based on deployment tasks and resource constraints and distribute the deployment policy to each server in the cluster. In the prototype system designed in this paper, there are two types of servers in the cluster, depending on whether virtual machines are enabled: deploy microservice instances directly as service deployment nodes, and launch virtual machines on the server as service instance deployment nodes, as in Server A in Figure 2.

Diversity can be achieved in many ways, such as code implementation in different languages, using different basic image containers, using different instruction sets for infrastructure environments, etc. Thanks to the development of container technology, using different base images has become the most convenient way. The diversity implementation used in this article is to use different images (such as Ubuntu, CentOS, Debian) as the basis for building template containers when building container templates. The diversity of functional components is obtained by running component scripts on different base containers.
We use Docker container technology to construct the system because Docker provides virtualization technologies that enable research to ignore the differences in the underlying platform. In this paper, hosts that directly deploy containers, such as Server A, VM1, and VM2, are considered deployment nodes. For business instances deployed on a node, an instance is a container with multiple versions of the implementation of a container that belongs to a microservice, a container template with the same business functionality built with different container mirrors. For example, for containers that implement Web front-end functionality, container builds use the underlying images of Ubuntu, Debian, CentOS, and so on, with different versions of the implementation carrying different resource overheads.

In the software system involved in this scenario, the system business is decoupled into several microservices, including Web front end, database query, and encryption components, and different microservices coordinate with each other to form a complete system service.
The prototype system includes three types of microservices: Web, Database, and Encrypt. Each type of microservice has three implementations based on Ubuntu, CentOS, and Debian. The resource consumption of 9 types of containers is shown in Table 3.

6.2. Evaluation. Our experiment was performed on the Dell T7920 workstation with Intel® Xeon Gold 6248R CPU with 3.00 GHz × 96 (dual processor), and a memory size of 512 GB, using the Ubuntu Server operating system. To verify policy performance, we used Python 3.7 for simulation.

The main simulation parameters covered in this paper are shown in Table 4.

Compare the deployment strategy used in this article with four classic algorithms, including LB-Greedy, Round-Robin, and Random. The LB-Diversity algorithm proposed in this paper is compared with the above four classical algorithms in the aspects of load balancing, shared vulnerability, and node failure.

6.2.1. Adaptability

The Number of Deployments. As you can see from the previous article, there is a multiple-variables group \{s, m, \(n_N\), \(n_{\text{deploy}}\)} in the scenario described in this article. In order to investigate the adaptability of the four deployment algorithms under different sizes, keep the first three in variables group \{s, m, \(n_N\), \(n_{\text{deploy}}\)} unchanged, and gradually increase the number of microservice instance deployments \(n_{\text{deploy}}\); i.e., \{s, m, nN, ndeploy\} = \{5,6,10, ndeploy\}.

Figure 3 shows the performance of the load balancing indicators of the four deployment algorithms under different deployment quantities. We find that the LB-Diversity proposed in this paper performs best overall. The load balancing indicator \(E_{LB}\) of LB-Diversity is always high, the volatility is smaller, and the performance index is stable compared with the other three algorithms. In addition, it can be noted that when \(n_{\text{deploy}} = 60\), there is a radical change in the LB-Diversity and LB-Greedy algorithms, showing load balancing indicator \(E_{LB} = 0.0\), which indicates that under this condition the results of both algorithm configurations bring the load of each node into line. This is due to similar load balancing by both algorithms, except that LB-Diversity uses a certain constraint to avoid deploying too many microservice components of the same type on a single node, resulting in a vulnerable node for a single microservice.

Figure 4 shows four deployment algorithm diversity indicators in different deployments and two conclusions can be drawn from the data. Firstly, as the number of microservice deployment instances increases, the overall degree of diversity of each algorithm will increase, which is the idea of using multi-version implementations to solve problems. Secondly, the LB-Diversity algorithm as a whole has a better diversity performance than the other three algorithms.

In addition, it can be found from Figure 4 that when the number of service deployments exceeds 20, as the number of deployments continues to increase, the diversity index values obtained by most deployment algorithms begin to stabilize. Among them, the results obtained by using the LB-Diversity and Round Robin algorithms, the curve begins to stabilize when the number of deployments is greater than 10. This has reference value for service operators.

Figure 5 shows the performance of four different deployment algorithms in different deployment quantities of the total performance of the resilience indicator, which is contributed by the load balancing indicator \(E_{LB}\) and the diversity indicator \(E_D\). On both indicators, LB-Diversity is the most prominent and therefore also the best performance under the overall resilience indicator.

The Number of Versions. Figures 6 to 8 show the performance of indicators \(E_{LB}\), \(E_D\), and \(E\) under the number of

### Table 3: Service instances (container) resource consumption.

<table>
<thead>
<tr>
<th>Container</th>
<th>Size (M)</th>
<th>CPU occupied</th>
<th>Memory occupied (K)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Web_Ubuntu</td>
<td>332</td>
<td>0.02</td>
<td>3116</td>
</tr>
<tr>
<td>Web_CentOS</td>
<td>477</td>
<td>0.03</td>
<td>3544</td>
</tr>
<tr>
<td>Web_Debian</td>
<td>333</td>
<td>0.02</td>
<td>3256</td>
</tr>
<tr>
<td>Database_Ubuntu</td>
<td>574</td>
<td>0.12</td>
<td>2388</td>
</tr>
<tr>
<td>Database_CentOS</td>
<td>640</td>
<td>0.20</td>
<td>2655</td>
</tr>
<tr>
<td>Database_Debian</td>
<td>583</td>
<td>0.13</td>
<td>2562</td>
</tr>
<tr>
<td>Encrypt_Ubuntu</td>
<td>177</td>
<td>0.01</td>
<td>1543</td>
</tr>
<tr>
<td>Encrypt_CentOS</td>
<td>418</td>
<td>0.02</td>
<td>2013</td>
</tr>
<tr>
<td>Encrypt_Debian</td>
<td>163</td>
<td>0.01</td>
<td>1569</td>
</tr>
</tbody>
</table>

### Table 4: Simulation parameters.

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of deployment nodes</td>
<td>8</td>
</tr>
<tr>
<td>Node CPU resource</td>
<td>References [1–4] (core)</td>
</tr>
<tr>
<td>Node memory resource</td>
<td>[1024, 2048, 3096, 4096] (M)</td>
</tr>
<tr>
<td>Container CPU occupancy</td>
<td>Average distribution on [0.01, 0.25]</td>
</tr>
<tr>
<td>Container memory occupancy</td>
<td>Average distribution on [1, 27] (M)</td>
</tr>
</tbody>
</table>

![Figure 3: Load balancing indicator performance of different deployment number.](image-url)
different microservice versions; i.e., \( \{s, m, n_N, n_{\text{deploy}}\} = \{5, m, 10, 10\} \).

In addition to showing the excellent implementation effect of the LB-Diversity algorithm, according to Figure 7, as the number of microservice implementation versions increases, the diversity index \( E_D \) has been significantly improved. It can be seen that increasing the diversity of microservice deployment versions is an important way to enhance overall diversity and improve system resilience.

We recommend that operators use as many versions of implementations as possible for each service within an acceptable development cost range to enhance the resilience of the software system to provide services.

6.2.2. Loss Evaluation of Shared Vulnerabilities. In response to a problem caused by an attacker exploiting a vulnerability disclosure, the effect of this issue is that the underlying mirror vulnerability is exploited by the attacker and the service instance launched based on that image is attacked.
Therefore, the decline in system service delivery capacity is in fact related to the number of underlying images that are not available due to a vulnerability attack.

As shown in Figure 9, Random has a higher loss than the other three algorithms. LB-Diversity, LB-Greedy, and Round Robin have similar results, and LB-Diversity is slightly smaller than the other two.

6.2.3. Loss Evaluation of Node Failure. For problems caused by node failures, this loss is determined by the number of instances that have been deployed on the failed node, as shown in Figure 10, which is significantly more stable and has a smaller performance loss than the other three deployment methods.

6.3. Evaluation Summary. According to the above experiments, we can conclude that, compared with the other three methods, LB-Diversity has better performance in both load balancing and diversity. This is because LB-Diversity tries to ensure that the principle of “decentralized deployment” is enforced regardless of the deployment location or deployment version. The principle of “decentralized deployment” includes two main points: “decentralized among nodes” and “decentralized among versions.”

In addition, in the failure scenario of node loss, the practice of “centralizing the same service chain on the same node” can also allow as few users as possible to be affected.

7. Application

The service architecture has been widely used at present, and many excellent open source platforms have been developed in practice. Microservice architectures are usually deployed in containers [28]; especially with the rise of Docker container technology [29] in recent years, software systems using Docker technology to build Microservice architectures have become de facto standard in the industry. The current mainstream microservice architecture platforms mainly include Kubernetes, Swarm, and Mesos. Among them, Kubernetes [30] launched by Google, with its high degree of molecularity, strong operability, and perfect self-healing mechanism, makes the deployment of containerized services simpler and more efficient [27], and has become the current benchmark in the industry.

Since Kubernetes adopts the Master-Slave mode, all control implementations related to cluster state changes will be completed on the Master node. The Kubernetes components on the Master node are highly modular, among which Schedule and Controller Manager are the main components for cluster state control.

The Scheduler is responsible for managing the service instance to be scheduled (a service instance in Kubernetes is a Pod) and all available nodes and selects the best host for the service instance to be scheduled by using the appropriate scheduling algorithm and scheduling strategy according to the cluster load information stored on Etcd. Bind the service instance to Node. Scheduler is a pluggable component that can be replaced with other schedulers.

Controller Manager is responsible for manipulating various controllers of Kubernetes and is a collection of controllers including Replication Controller, Node Controller, etc. Controller is the core abstraction in Kubernetes. After the user declares the desired state through Kube-ApiServer, the controller continuously monitors the current state in Kube-ApiServer and reacts to the difference between the current state and the desired state to ensure that the current state of the cluster is constantly approaching user-declared desired state.

When the total number of deployments and available deployment versions are known, the method proposed in this paper mainly solves two problems: (1) how many containers should be configured for each version; (2) where each container should be deployed. Considering the use of this method in Kubernetes, it is only necessary to convert the calculation result of problem (1) into a yaml file through automatic code generation for automatic deployment. Secondly, the scheduling strategy of Scheduler is replaced by the method used in this paper to determine the location of container deployment.

In addition, in the auto-scaling scenario, whenever a microservice instance is to be added or deleted, we only need to add a module to the scheduling algorithm. This module is responsible for determining the version and deployment
location of the adding/removing microservice instance, based on the comparison of the changes in the Total Resilience Indicators before and after the adding/removing, just as shown in formula (13). The auto-scaling problem is also the next research point that our team is working on, which we will show in detail in the next paper.

The above application cases demonstrate the strong adaptability of the proposed mechanism in practical applications. This shows the application value and promotion value of this mechanism in practice.

8. Conclusion

The application of microservice architecture can overcome many problems of traditional monomer architecture, but because microservices are usually deployed by instance through containers, this behavior can effectively improve productivity and module reuse rate, but also create the problem of sharing vulnerabilities between containers.

To mitigate shared vulnerabilities under the microservice architecture, we have established a diversified microservices deployment model to maximize the resilience of the system. Combined with the problem of load balancing, a microservice deployment algorithm based on load balancing and diversity is proposed, which reduces the service’s dependence on the underlying mirror by abundant diversity and avoids the overreliance of microservices on a single node through decentralized deployment, while taking into account load balancing [31].

Data Availability

The authors have pushed the source data they used and the intermediate results in the experiment to GitHub for open source. The URL of the project repository is https://github.com/goodboydan/Microservice-Deployment.

Conflicts of Interest

The authors declare that they have no conflicts of interest.

Acknowledgments

This research was supported by a research grant from the National Natural Science Foundation of China under Grant no. 61379149, the China Postdoctoral Science Foundation under Grant no. 2017M610286, and the National Science Foundation of Jiangsu Province under Grant no. SBK2020043435.

References


