Journal of Cloud Computing:
Advances, Systems and Applications

RESEARCH

Open Access

Load balancing scheduling mechanism for OpenStack and Docker integration

Jiarui Qian[1], Yong Wang[2], Xiaoxue Wang[1], Peng Zhang[2] and Xiaofeng Wang[1,3*]

Abstract

With the development of cloud-edge collaborative computing, cloud computing has become crucial in data analysis and data processing. OpenStack and Docker are important components of cloud computing, and the integration of the two has always attracted widespread attention in industry. The scheduling mechanism adopted by the existing fusion solution uses a uniform resource weight for all containers, and the computing nodes resources on the cloud platform is unbalanced under differentiated resource requirements of the containers. Therefore, considering different network communication qualities, a load-balancing Docker scheduling mechanism based on OpenStack is proposed to meet the needs of various resources (CPU, memory, disk, and bandwidth) of containers. This mechanism uses the precise limitation strategy for container resources and a centralized strategy for container resources as the scheduling basis, and it generates exclusive weights for containers through a filtering stage, a weighing stage based on weight adaptation, and a non-uniform memory access (NUMA) lean stage. The experimental results show that, compared with Nova-docker and Yun, the proposed mechanism reduces the resource load imbalance within a node by 57.35% and 59.00% on average, and the average imbalance between nodes is reduced by 53.53% and 50.90%. This mechanism can also achieve better load balancing without regard to bandwidth.

Keywords Cloud computing, OpenStack, Docker, Scheduling mechanism, Load balancing, Collaborative computing

Introduction

With the great increase in data traffic, cloud-edge collaborative [1] computing can effectively address network congestion and transmission delay, thereby improving performance. In cloud-edge collaborative computing, scheduling optimization can also be used to reduce the overall computing service delay and improve service reliability [2]. Cloud computing [1, 3] plays a critical role in cloud-edge collaborative computing. It undertakes computing-intensive tasks and takes advantage of its superior computing power and high efficiency to reduce reducing computing power costs.

Virtualization technology and cloud computing technology [4] are the two core technologies in the field of cloud computing. In recent years, lightweight virtualization technology, represented by Docker, has become a research hotspot because of its advantages such as faster deployment and delivery, lower performance expenses, and higher resource utilization compared with traditional virtualization technology [4, 5]. Cloud platform technology can integrate different virtualization technologies to achieve the unified provisioning and elastic scaling of resources. Currently, OpenStack has become the standard for Infrastructure as a Service (IaaS) with its excellent open-source, flexible, and scalable performance [6]. The convergence of OpenStack and Docker is inevitable for the development of collaborative computing. OpenStack can provide fine-grained, large-scale container management capabilities for Docker, while Docker can enrich the OpenStack ecosystem and optimize the overall performance of cloud platforms [7].

*Correspondence:
Xiaofeng Wang
wangxf@jiangnan.edu.cn
[1] School of Artificial Intelligence and Computer Science, Jiangnan University, Wuxi, China
[2] China Key System & Integrated Circuit Co., Ltd., Wuxi, China
[3] Peng Cheng Laboratory, Shenzhen, China

In view of the advantages of the integration of OpenStack and Docker, performing Docker scheduling based on OpenStack to achieve cloud platform resource load balancing has become an urgent problem to be solved. Load balancing is an essential mechanism in Docker scheduling. An efficient load balancing mechanism can improve the reliability and scalability of cloud computing systems, enhance service availability, optimize cloud platform resource utilization, and improve service quality [8, 9]. Representative works include: Nova-docker [10], which is oriented toward the CPU, memory, and disk resource requirements in the container. It adopts the filtering and weighing scheduling mechanism of the Nova component in OpenStack for the virtual machine. In the filtering stage, the nodes that meet the resource requirements of the container are selected among all the computing nodes of OpenStack as candidate nodes. In the weighing stage, the optimal computing node is selected for the container by calculating the weight of each candidate node. Yun [11] is oriented toward the CPU, memory, disk, and bandwidth resource requirements in the container. It uses a container scheduling mechanism based on resource utilization awareness and NUMA awareness to achieve better load balancing while improving the container performance compared to that obtained with Nova-docker. However, the scheduling mechanisms adopted by Nova-docker and Yun use uniform resource weights for all containers, and different resource requirements of the containers will lead to the unbalanced utilization of cloud platform resources [12–14].

This paper mainly aim at addressing the lack of load balancing in the scheduling mechanism in Nova-docker and Yun, a Load-Balancing Scheduling Mechanism (LBSM) is proposed for the differentiated resource requirements of containers. Balancing the CPU, memory, and disk resource requirements for containers can provide users with high-quality computing and storage services. At the same time, the network serves as the hub connecting all computing and storage resources in the OpenStack cloud platform, and implementing load balancing for the CPU, memory, disk, and bandwidth requirements of containers can further reduce network congestion and improve communication quality [15, 16]. Therefore, the LBSM schedules a container considering the container's CPU, memory, disk, and bandwidth resource requirements. The following are the main contributions of this study:

- We accurately limit the container disk resources to ensure the accuracy of the cloud platform resource scheduling and avoid resource competition between containers.
- An adaptive weight algorithm is proposed to generate proprietary resource weights for the differentiated resource requirements of containers.
- Aiming at load balancing, we optimize OpenStack and Docker integration and brings no additional time consumption.

Related work

At present, load balancing mechanism mainly applies in the fields of cloud-edge collaborative computing, virtual machines and container scheduling. In this section, we provide an overview of the existing load balancing scheduling mechanisms and the difference between the load balance mechanisms in OpenStack and Kubernetes.

References [2, 17], and [18] mainly introduce load balancing in cloud-edge collaborative scheduling. To reduce the computing service delay of an in-vehicle network, reference [2] proposes a task differentiation and scheduling algorithm, develops a collaborative computing method, and builds a cloud-edge collaborative computing framework. Reference [17] designs an intelligent air quality monitoring system combining cloud and edge computing based on container virtualization. In the intelligent air quality monitoring system, OpenStack provides virtualization services, and the edge provides service applications. The method of reference [18] mainly makes optimization decisions for edge computing offloading based on the real-time state of the network and the attributes of tasks to address the load imbalance of servers between autonomous edge subnets. References [2] and [17] perform little research on load balancing in cloud-edge collaborative computing. Reference [18] mainly studies the load balancing of edge computing in a cloud-edge collaborative computing system, and the load balancing in cloud computing is less complex.

References [9, 19], and [20] mainly introduce load balancing in virtual machines. Reference [9] proposes a multi-objective load balancing mechanism based on machine learning for determining the placement of virtual machines in cloud data centers. It comprehensively considers three resource types, CPU, memory, and bandwidth, to achieve load balancing within each host of the cloud data center and among the hosts. For the load balancing problem between virtual machines and physical hosts in the cloud environment, reference [19] first uses the K-means algorithm based on Bayesian optimization and the artificial neural network (ANN) algorithm to divide the virtual machines into low load collections and overloaded collections. Then, it schedules user tasks to a collection of low-load virtual machines for load balancing. Reference [20] proposes a dynamic load balancing algorithm based on deep

reinforcement learning (DRL) under the constraints of a service level agreement (SLA), which effectively reduces the load imbalance and task rejection rate of virtual machines. These works are all based on achieving better load balancing in virtual machines, and there is less research on Docker scheduling based on cloud computing.

References [14] and [21] mainly introduce load balancing in container scheduling. Reference [14] proposes a Docker cluster scheduling strategy based on a genetic algorithm, which takes the load of the CPU, memory, hard disk I/O, and network traffic into consideration and effectively improves the load balancing performance of the cluster and the efficiency of multi-task concurrent scheduling. However, the model for predicting the load values of container tasks needs to manually set the initial parameters of the task, which increases the difficulty of practical application of container scheduling. Reference [21] proposes an intelligent container scheduler and improves it from the perspective of load balancing, latency, etc., but this work only discusses some suitable techniques.

References [22] introduces Kubernetes scheduling mechanism. Kubernetes scheduling includes filtering phase and optimization phase. The filtering phase traverses all working nodes and filters them by filtering rules. In the optimization stage, the optimal strategy is used to calculate the score of all candidate nodes, and the node with the highest score is selected for scheduling. However, Kubernetes' default scheduling policy only considers CPU and memory, and is scored based on mono-criterion criterion, which fails to improve resource utilization and load balancing.

It can be seen that there are few researches on load balancing in the integration of OpenStack and Docker. The load balancing mechanism mentioned above take CPU, memory, disk, and bandwidth into account. An efficient load balancing mechanism can improve the service processing capability of cloud computing system, solve network congestion problems to provide better access quality for users, and improve resource utilization. Therefore, our work is necessary.

LBSM

An efficient load balancing scheduling mechanism can achieve high availability of cloud computing systems and optimize the service quality of cloud platform users. However, the scheduling mechanism adopted by the existing OpenStack and Docker fusion solutions use a uniform resource weight for all containers, which cannot meet the resource load balancing requirements of cloud platform computing nodes under differentiated resource requirements of containers. The LBSM is shown in Fig. 1.

The LBSM is based on the precise limitation strategy for container resources and a centralized strategy for container resources when it is oriented toward various resource demand scenarios of containers. The precise limit strategy for container resources aims to precisely limit the usage of the corresponding resources of the container to avoid resource preemption between containers. The container resource centralization strategy considers CPU and memory centrally to avoid frequent

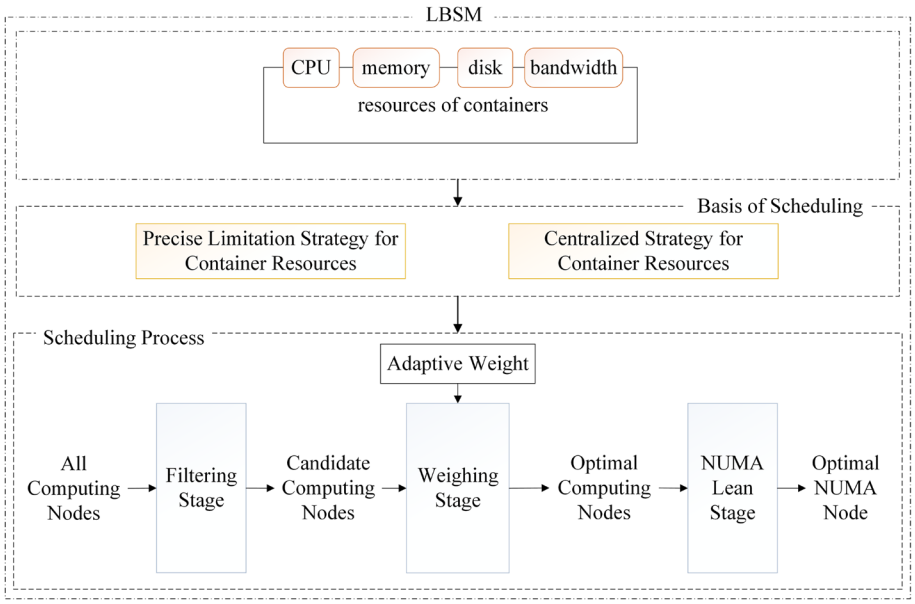

Fig. 1 LBSM

switching of the CPU and memory and remote memory access across NUMA nodes during container operation.

The scheduling process of the LBSM is divided into three stages: the filtering stage, the weighing stage, and the NUMA lean stage. First, through the filtering stage, the nodes whose remaining resources do not meet the container resource request specification are filtered. The goal of this stage is to select nodes that meet the container resource request specifications from among all the Docker computing nodes of OpenStack as candidate computing nodes to avoid nodes that cannot deploy containers due to insufficient resources. Then, the optimal computing node is selected for the container from among all candidate computing nodes through the weighing stage based on weight adaptation, and the optimal NUMA node is selected from among all NUMA nodes of the optimal computing node through the NUMA lean stage. The goal of these two stages is to generate exclusive weights for containers when facing differentiated resource requirements of containers, deploy containers to optimal computing nodes, and occupy resources that are allocated to them by optimal NUMA nodes.

LBSM implements resource load balance within a node and between nodes, considering CPU, memory, disk, and bandwidth. LBSM implements resource load balancing within a node. If the CPU utilization reaches 90% and the memory utilization reaches only 20%, computing node will cannot deploy new containers due to insufficient CPU resources, even though memory resources are sufficient. As a result, computing node wastes memory resources. In addition, the container running on the computing node cannot achieve good performance due to high CPU load. LBSM implements resource load balancing between nodes. If the CPU utilization of computing node 1 reaches 85% and the CPU utilization of computing node 2 reaches only 15%, computing node 1 will undertake a large number of computing-intensive tasks, resulting in high CPU load. In this way, the stability of OpenStack cloud platform and the availability of user service cannot be guaranteed. LBSM can provide high quality computing and storage services for computing-intensive and memory-consuming containers when it is oriented toward CPU, memory, and disk resource requirements. LBSM can meet the container's high requirements on network communication quality and reduce network congestion when it is oriented toward CPU, memory, disk, and bandwidth resource requirements.

Basis of scheduling

Precise limitation strategy for container resources

The precise limitation strategy for container resources is used as the basis for scheduling by the LBSM. A precise resource limit can ensure the accuracy of resource scheduling in the cloud platform and avoid resource competition among containers, thus improving the quality of service. In the filtering stage, the LBSM determines whether the remaining computing node resources can meet the request specifications of the container's corresponding CPU, memory, disk, and bandwidth resources. Therefore, the precise limitation strategy for container resources will precisely limit the utilization of corresponding container resources according to the container request specifications for CPU, memory, disk, and bandwidth resources to ensure the accuracy of resource scheduling in the filtering phase.

The LBSM encapsulates the CPU and memory resource limit API interfaces officially provided by Docker, and it implements precise limits on the container CPU and memory resources. The container disk resource limit can be implemented based on the devicemapper storage driver by calling the disk limit API interface officially provided by Docker. However, loop-lvm is only suitable for the test environment and has poor performance. Direct-lvm is suitable for the production environment, but its configuration is complicated. Additionally, the default basic device size of devicemapper is 10 GB. If the disk request size of the container is less than 10 GB when creating a container, the container will not be successfully created. In view of the deficiencies in the precise limitation of container disk resources, the LBSM changes the storage driver to overlay2. The LBSM implements precise restrictions on container disk resources through the following steps:

Docker uses Union File System technology to obtain a layered stack of container images. The precise container resource restriction strategy uses overlay2 as the Docker storage driver to implement the Union File System technology, and it uses the feature that the files in the container exist in the "container layer" to convert restrictions on container disk resources to restrictions on the "container layer".

Centralized strategy for container resources

Due to its high aggregate memory bandwidth and good system scalability [23, 24], NUMA is currently the most common physical server architecture and has become the mainstream in the fields of high-performance computing and cloud computing. As shown in Fig. 2, the NUMA architecture divides the CPU and memory resources of the system into multiple independent NUMA nodes. Each NUMA node consists of a socket and its adjacent memory nodes. In each socket, multiple logical CPUs share the same integrated memory controller (IMC), and the IMC is connected to its local memory node. Different

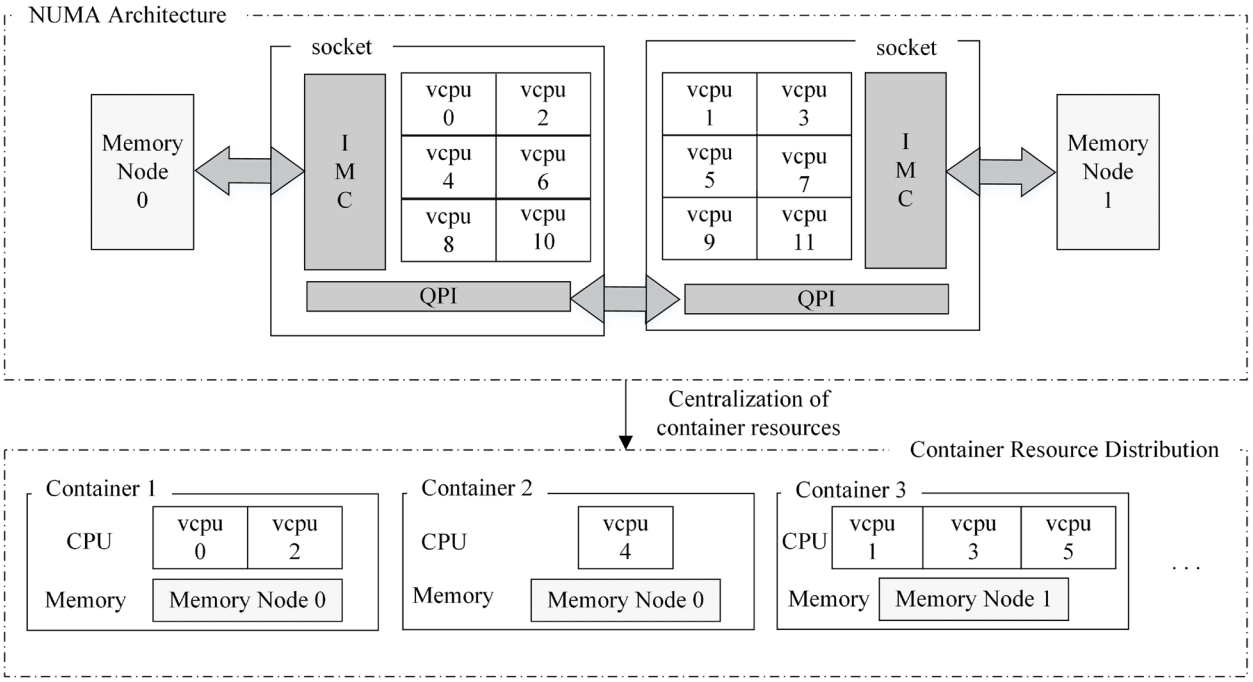

Fig. 2 Centralized strategy for container resources

sockets are interconnected through links (such as Intel's Quick Path Interconnect, QPI).

Therefore, the LBSM is based on the NUMA architecture and introduces a container resource centralization strategy. Concentrating the logical CPU and memory of the container on the same NUMA node avoids frequent switching of the CPU and memory and remote memory access across NUMA nodes during the running process of the container.

Scheduling process

Filtering stage

In the filtering stage, five filters for CPU, memory, disk, bandwidth, and NUMA are introduced for the CPU, memory, disk and bandwidth resource requirements of the containers to filter all computing nodes in OpenStack. The CPU, memory, and disk filters are filtered by the remaining amount of the corresponding resources on the computing node. For example, if the remaining amount of CPU resources on the current computing node is less than that in the container CPU resource request specification, then the node will be filtered directly. The bandwidth filter uses the real-time bandwidth utilization of the current computing node as the filtering criterion, and if the bandwidth utilization of the computing node exceeds the set threshold (defined as 90% in this paper), the node will be filtered directly. The LBSM adopts a centralized strategy for container resources, which concentrates both the CPU and memory of the container into the same NUMA node. Therefore, the NUMA filter is introduced, and if the current computing node does not have a NUMA node that satisfies the container's CPU and memory resource requests, the node will be filtered directly. Eventually, the nodes that have passed all the filters will be taken as the candidate nodes in the weight-based adaptive weighing phase.

Weighing phase based on weight adaptation

For the CPU, memory, disk, and bandwidth resource requirements of containers, four scales of CPU, memory, disk, and bandwidth are introduced in the weighing phase. As shown in Fig. 3, for the differentiated resource requirements of containers, the LBSM comprehensively considers the resource requirement characteristics of the containers and the resource characteristics of the computing nodes in the weighing stage. Through node weighing and weight adaptation, the computing node with the highest weight score is selected from among all the candidate computing nodes obtained in the filtering stage as the optimal computing node.

1) Node weighing: Node weighing is performed based on the real-time resource utilization information and information on the amount of remaining resources of each computing node. The amount of remaining resources of all candidate computing nodes is $R_t = \{r_t^1, r_t^2, \ldots, r_t^m\}$, and the real-time utilization

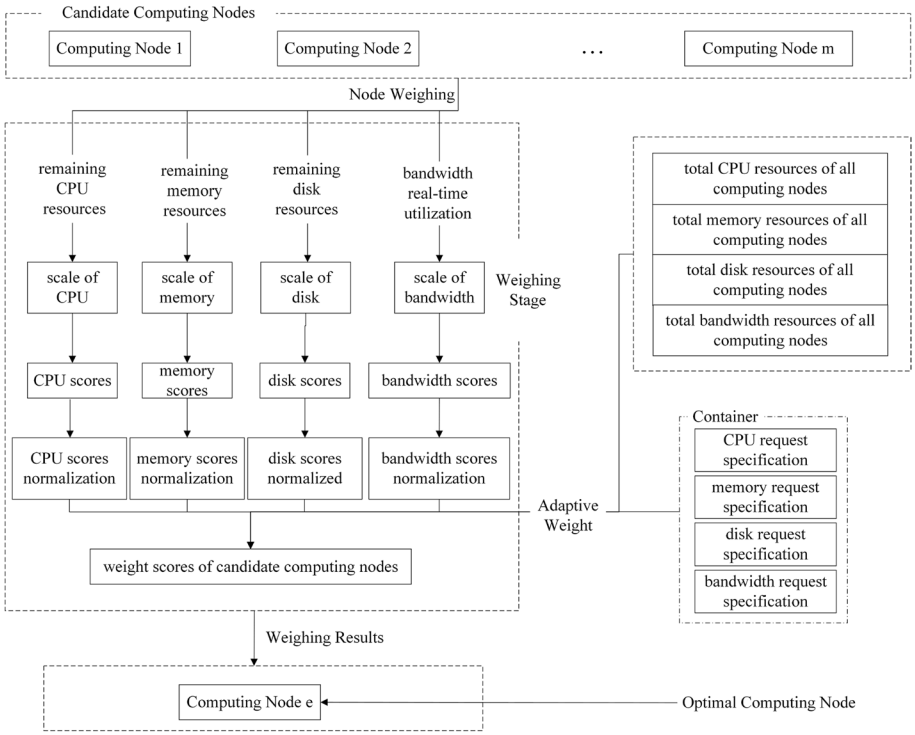

Fig. 3 Weighing phase based on weight adaptation

of resources t is $U_t = \{u_t^1, u_t^2, \ldots, u_t^m\}$, where m represents the total number of candidate computing nodes. Then, the scores of all candidate computing nodes $X_t = \{x_t^1, x_t^2, \ldots, x_t^m\}$ against resource t can be calculated by Eq. (1).

$$x_t^i = \begin{cases} r_t^i, & t \in \{cpu, memory, disk\} \\ 1 - u_t^i, & t \in bandwidth \end{cases}, i \in \{1, 2, \ldots, m\} \tag{1}$$

The score y_t^i of candidate computation node i normalized against resource t can be calculated by Eq. (2), where x_t^{max} and x_t^{min} respectively denote the maximum and minimum values of X_t.

$$y_t^i = \frac{x_t^i - x_t^{min}}{x_t^{max} - x_t^{min}}, i \in \{1, 2, \ldots, m\} \tag{2}$$

2) Adaptive Weight: For the differentiated resource requirements of containers, proprietary resource weights are generated for containers based on the total resource information of each computing node in OpenStack and the container resource request specifications. If the total number of resources t of all computing nodes in OpenStack is $A_t = \{a_t^1, a_t^2, \ldots, a_t^c\}$ and the request specification of container j for resource t is d_t^j, then the weight w_t^j of container j against resource t can be calculated by Eq. (3), where c represents the total number of all computing nodes in OpenStack and n represents the number of resource types.

$$tmp_t^j = \frac{d_t^j}{\sum_{i=1}^{c} a_t^i}, w_t^j = \frac{tmp_t^j}{\sum_{t=1}^{n} tmp_t^j} \tag{3}$$

Then, the weight score $S = \{s_1, s_2, \ldots, s_m\}$ of all candidate computing nodes can be calculated by Eq. (4). Finally, the candidate computing node e with the highest weight score will be the optimal computing node for deployment container j.

$$s_i = \sum_{t=1}^{n} \left(w_t^j \times y_t^i \right), i \in \{1, 2, \ldots, m\} \tag{4}$$

NUMA lean stage

The NUMA lean stage mainly consists of two stages: the NUMA filtering stage and the NUMA weighing stage, as shown in Fig. 4. The NUMA filtering stage uses a centralized strategy for container resources as the scheduling basis to filter all NUMA nodes on the optimal computing node. Only when the remaining CPU and memory resources of the current NUMA node are greater than the request specifications of the corresponding resources of the container will the NUMA node be used as a

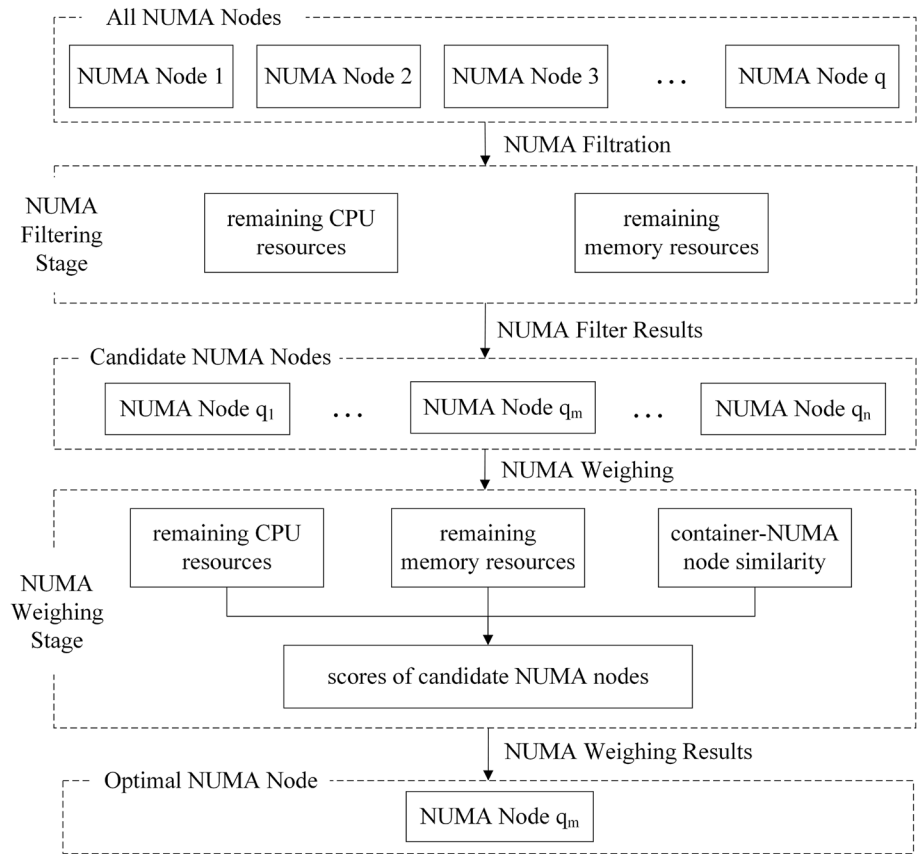

Fig. 4 NUMA lean stage

candidate NUMA node and enter the NUMA weighing module.

The NUMA weighting stage assigns a score to NUMA nodes based on the remaining CPU and memory and the similarity between the container and the current NUMA node. The CPU surplus of all candidate NUMA nodes is $R_{cpu} = \{r^1_{cpu}, r^2_{cpu}, \ldots, r^q_{cpu}\}$ and the amount of memory remaining is $R_{mem} = \{r^1_{mem}, r^2_{mem}, \ldots, r^q_{mem}\}$, where q represents the total number of candidate NUMA nodes. The request specification of container j for CPU resources is d^j_{cpu} , the request specification of memory resources is d^j_{mem}, and the resource unit of the container is the same as that of the NUMA nodes (e.g., the container memory unit and the NUMA node memory unit are both in MB). We use Eqs. (5) and (6) to normalize the resource request specification of the container and the resource residual of the NUMA nodes to the same metric, respectively.

$$D^j_{cpu} = \frac{d^j_{cpu}}{\sum_{k=1}^{q} r^k_{cpu} + d^j_{cpu}}, D^j_{mem} = \frac{d^j_{mem}}{\sum_{k=1}^{q} r^k_{mem} + d^j_{mem}} \tag{5}$$

$$R^k_{cpu} = \frac{r^k_{cpu}}{\sum_{k=1}^{q} r^k_{cpu} + d^j_{cpu}}, R^k_{mem} = \frac{r^k_{mem}}{\sum_{k=1}^{q} r^k_{mem} + d^j_{mem}} \tag{6}$$

D^j_{cpu} denotes the CPU resource request specification of the normalized container j, D^j_{mem} denotes the memory resource request specification of the normalized container j, R^k_{cpu} denotes the CPU resources remaining in the normalized NUMA node k, and R^k_{mem} denotes the memory resources remaining in the normalized NUMA node k. Then, the value of the similarity SIM^j_k between container j and NUMA node k can be calculated by Eq. (7). Equation (7) uses cosine similarity to measure similarity, which makes the weight interval small.

$$SIM^j_k = \frac{D^j_{cpu} \times R^k_{cpu} + D^j_{mem} \times R^k_{mem}}{\sqrt{\left(D^j_{cpu}\right)^2 + \left(D^j_{mem}\right)^2} \times \sqrt{\left(R^k_{cpu}\right)^2 + \left(R^k_{mem}\right)^2}} \tag{7}$$

Then, the scores $G = \{g_1, g_2, \ldots, g_q\}$ of all candidate NUMA nodes can be calculated by Eq. (8). Finally, the candidate NUMA node with the highest score will be

chosen as the optimal NUMA node for deployment container j.

$$g_k = R^k_{cpu} + R^k_{mem} + SIM^j_k, k \in \{1, 2, \ldots, q\} \tag{8}$$

Experimental results and analysis

To verify the effectiveness of the LBSM, an experimental environment was built based on OpenStack Mitaka, containing one control node, one network node, and three Docker computing nodes. The operating system of the control and network nodes is CentOS 7.2, the operating system of each computing node is CentOS 7.8, and the Docker version is 17.06.0-ce. The information for each Docker computing node is shown in Table 1.

Verification of precise limits on container disk resources

This experiment mainly verifies that the LBSM can meet precise limitations on container disk resources. This experiment uses Nova-docker, Yun, and the LBSM to deploy containers with disk request specifications of 1 GB, 2 GB, 4 GB, 8 GB, 12 GB, 16 GB, and 20 GB. All containers aim to create a test file with a size of 30GB to check the actual disk usage of each container. The experimental results are shown in Table 2.

As can be seen from Table 2, Nova-docker cannot implement precise limits on container disk resources. For containers with disk request specifications greater than 10GB, Yun can implement precise resource limits. But when the disk request specification is less than 10GB, the container cannot be created successfully. The LBSM uses the overlay2 storage driver, which can successfully create containers with different disk request specifications and accurately limit disk resources.

Verification of load balancing

We measure the resource load balance of OpenStack computing nodes in terms of both the intranode and internode resource load imbalance [9]. Equations (9) and (10) use the coefficient of variance (CV) theory to calculate the amount of load imbalance. In probability theory, variance is often used to measure the degree of deviation between a random variable and the mean.

1) Intranode resource load imbalance. The intranode resource load imbalance in OpenStack is denoted as L_{intra} and is calculated by Eq. (9), where t represents the resource type, n represents the number of resource types, u^i_t represents the utilization of resource t of computing node i, cl_i represents the load imbalance of computing node i, and c represents the total number of computing nodes in OpenStack. The smaller L_{intra} is, the more balanced the utilization of resources in each dimension within the node in OpenStack.

$$cl_i = \sqrt{\frac{1}{n}\sum_{t=1}^{n}\left(u^i_t - \frac{1}{n}\sum_{t=1}^{n}u^i_t\right)^2}, L_{intra} = \frac{1}{c}\sum_{i=1}^{c}cl_i \tag{9}$$

2) Internode resource load imbalance. The internode resource load imbalance in OpenStack is expressed as L_{inter} and calculated by Eq. (10), where rl_t denotes the load imbalance of resources t. The smaller L_{inter} is, the more balanced the utilization of resources in each dimension among the nodes in OpenStack.

$$rl_t = \sqrt{\frac{1}{c}\sum_{i=1}^{c}\left(u^i_t - \frac{1}{c}\sum_{i=1}^{c}u^i_t\right)^2}, L_{inter} = \frac{1}{n}\sum_{t=1}^{n}rl_t \tag{10}$$

In this experiment, we constructed 9 container resource request specifications, as shown in Table 3, for the CPU, memory, and disk resource requirements of containers and the CPU, memory, disk, and bandwidth resource requirements of containers. It should be noted that Yun uses devicemapper as the storage driver for Docker, and the default disk size that Docker can use is 107.4 GB. To ensure that Nova-docker, Yun, and the LBSM are all compared based on the experimental environment shown in Table 1, we set Nova-docker, Yun, and the LBSM using overlay2 as the storage driver for Docker; the disk space available for Docker was 120 GB.

As shown in Table 4, we used Nova-docker, Yun, and the LBSM to deploy containers on three Docker

Table 1 Docker computing node information

node number	logical CPU	memory (GB)	disk (GB)	bandwidth (Gigabits)	NUMA node number	NUMA node logical CPU	NUMA node memory (MB)
1	24	31.2	120	1	0	12	7738
					1	12	24173
2	24	15.4	120	1	0	12	7739
					1	12	8045
3	24	15.4	120	1	0	12	7739
					1	12	8045

Table 2 Verification of precise disk resource limits

scheduling mechanism	disk request specification (GB)	test file specification (GB)	actual disk usage specification (GB)
Nova-docker	1	30	31
	2	30	31
	4	30	31
	8	30	31
	12	30	31
	16	30	31
	20	30	31
Yun	1	30	Creation failure
	2	30	Creation failure
	4	30	Creation failure
	8	30	Creation failure
	12	30	12
	16	30	16
	20	30	20
LBSM	1	30	1
	2	30	2
	4	30	4
	8	30	8
	12	30	12
	16	30	16
	20	30	20

computing nodes. For the CPU, memory, and disk resource requirements of the containers, A1∼A5 were based on the container resource request specifications in C1∼C4. For the CPU, memory, disk, and bandwidth resource requirements of containers, B1∼B5 were based on the container resource request specifications in C5∼C9.

As shown in Fig. 5, for the CPU, memory, and disk resource requirements of the containers, compared with Nova-docker and Yun, the LBSM performs well in terms of both intranode resource load imbalance and internode resource load imbalance. Compared to Nova-docker, the LBSM reduces the load imbalance of resources within a node by an average of 55.09%, with the highest reduction of 72.27% in the A1 group test and the lowest reduction of 41.88% in the A2 group test. Compared with Yun, the LBSM reduces the load imbalance of resources within a node by 50.68% on average, with the greatest reduction of 59.09% in the A5 group test and the smallest reduction of 43.56% in the A2 group test. Compared with Nova-docker, the LBSM reduces the resource load imbalance between nodes by 56.92% on average, with the largest

Table 3 Container resource request specifications

resource requirement of containers	container number	container resource request specification			
		CPU	memory (MB)	disk (GB)	bandwidth (Mbps)
CPU, memory, disk	C1	4	1024	5	0
	C2	1	4096	5	0
	C3	1	1024	20	0
	C4	2	2048	10	0
CPU, memory, disk, bandwidth	C5	4	1024	5	45
	C6	1	4096	5	45
	C7	1	1024	20	45
	C8	1	1024	5	180
	C9	2	2048	10	90

Table 4 Load balancing test group

test number	container deployment order
A1	C1×6→ C2×4→ C3×4→ C4×4
A2	C1×4→ C2×6→ C3×4→ C4×4
A3	C1×4→ C2×4→ C3×6→ C4×4
A4	C1×4→ C2×4→ C3×4→ C4×6
A5	C1×5→ C2×5→ C3×5→ C4×3
B1	C5×6→ C6×3→ C7×3→ C8×3→ C9×3
B2	C5×3→ C6×6→ C7×3→ C8×3→ C9×3
B3	C5×3→ C6×3→ C7×6→ C8×3→ C9×3
B4	C5×3→ C6×3→ C7×3→ C8×6→ C9×3
B5	C5×4→ C6×4→ C7×4→ C8×4→ C9×2

reduction of 66.99% in the A1 group test and the smallest reduction of 46.49% in the A3 group test. Compared with Yun, the LBSM achieves an average reduction of 40.79% in resource load imbalance between nodes, with the greatest reduction of 51.55% in the A5 group test and the smallest reduction of 30.38% in the A3 group test.

As shown in Fig. 6, for the CPU, memory, disk, and bandwidth resource requirements of containers, the LBSM also achieves better resource load balancing in terms of intranode and internode resource load imbalance than Nova-docker and Yun. Compared with Nova-docker, the LBSM achieves an average reduction of 57.35% in intranode resource load imbalance, with the greatest reduction of 73.98% in the B4 group and the smallest reduction of 38.77% in the B2 group.

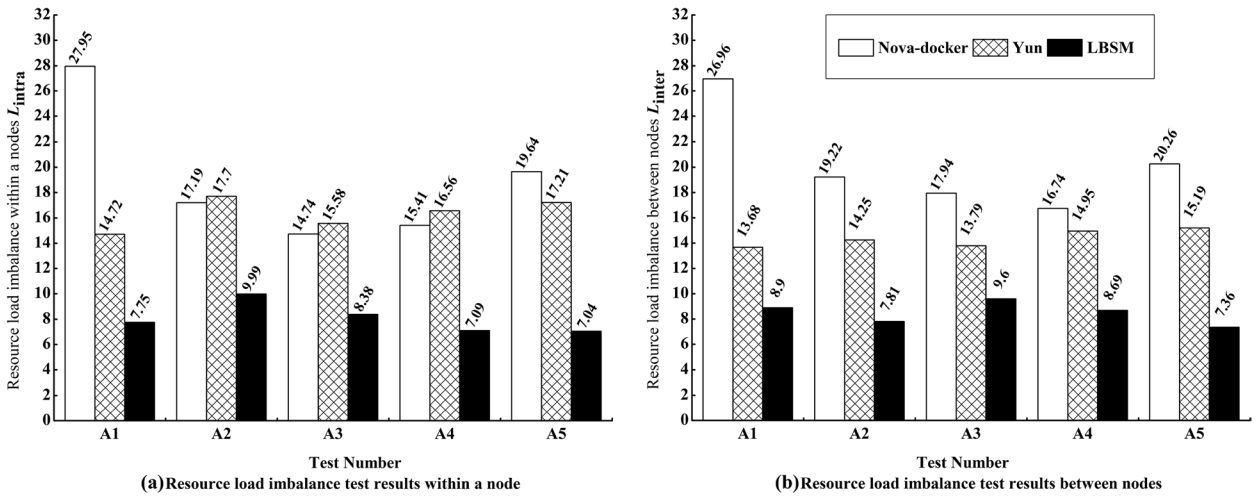

Fig. 5 Load balancing test results for three resources requested by containers

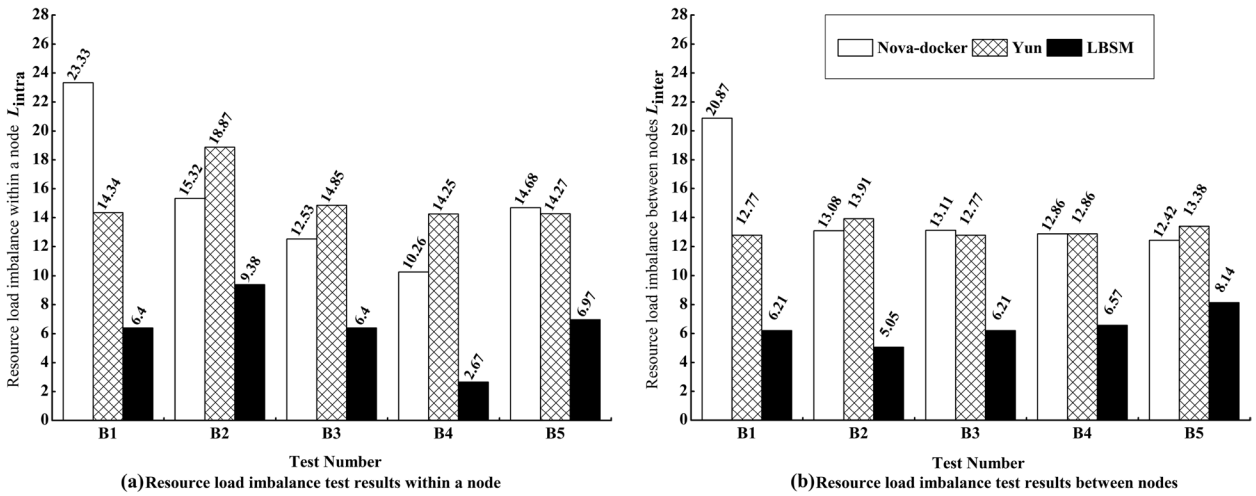

Fig. 6 Load balancing test results for four resources requested by containers

Compared to Yun, the LBSM reduces the load imbalance of resources within a node by 59.00% on average, with the greatest reduction of 81.26% in the B4 group and the smallest reduction of 50.29% in the B2 group. Compared to Nova-docker, the LBSM achieves an average reduction of 53.53% in resource load imbalance between nodes, with the greatest reduction of 70.24% in the B1 group test and the smallest reduction of 34.46% in the B5 group test. Compared with Yun, the LBSM reduces the internode resource load imbalance by 50.90% on average, with the greatest reduction of 63.70% in the B2 group test and the smallest reduction of 39.16% in the B5 group test.

Verification of container deployment time consumption

To compare the time consumption of Nova-docker, Yun, and the LBSM for deploying containers, we test the time required for Nova-docker, Yun, and the LBSM to deploy containers C1∼C4 in Table 3 to meet the CPU, memory, and disk resource requirements of the containers. A total of four sets of tests are performed. The deployment time of a container is the difference between the update time of the container and the request creation time of the container. To avoid the effects of chance on the experimental results, each group of tests is conducted three times. That is, each container specification is deployed three times, and the average value of the three deployment times is taken as the deployment time of the container for that resource request specification. The experimental results are shown in Fig. 7(a). For the CPU, memory, disk, and bandwidth resource requirements of the containers, we test the time required for Nova-docker, Yun, and the LBSM to deploy the five containers numbered C5∼C9 in Table 3. A total of five sets of tests are performed. The results of the experiments are shown in Fig. 7(b). For the CPU, memory, and disk resource requirements of the containers, Nova-docker takes 8.42 seconds to deploy containers on average. Yun takes 1.42 seconds and LBSM takes 1.58 seconds on average. For the CPU, memory, disk, and bandwidth resource requirements of the containers, Nova-docker takes 8.07 seconds to deploy containers on average. Yun takes 1.6 seconds and LBSM takes 1.27 seconds on average. Nova-docker takes more time to deploy the containers, while Yun as well as LBSM take less than 2 seconds to deploy containers. That is, the LBSM achieves better load balancing without increasing the container scheduling time consumption.

Conclusion and future work

Considering the convergence of OpenStack and Docker, this paper proposes a load-balancing scheduling mechanism for differentiated container resource demands. The scheduling mechanism is based on the precise container resource limitation strategy and a centralized strategy for container resources. The candidate computing nodes are first selected from among all computing nodes in OpenStack through a filtering phase. Then, the optimal computing node is selected for the container through a weight-based adaptive weighing phase. Finally, the optimal NUMA node is selected for the container through a NUMA leaning phase. The experiments show that, in contrast to the scheduling mechanisms adopted by Nova-docker and Yun, the scheduling mechanism proposed in this paper can achieve the precise limitation of container resources and effectively reduce the resource load imbalance degree within a node and among computing nodes in OpenStack. At the same time, this scheduling mechanism does not increase the time consumption.

The LBSM schedules containers based on their initial resource request specifications. However, users do not know the exact resource specifications required by the application and containers do not run at full capacity.

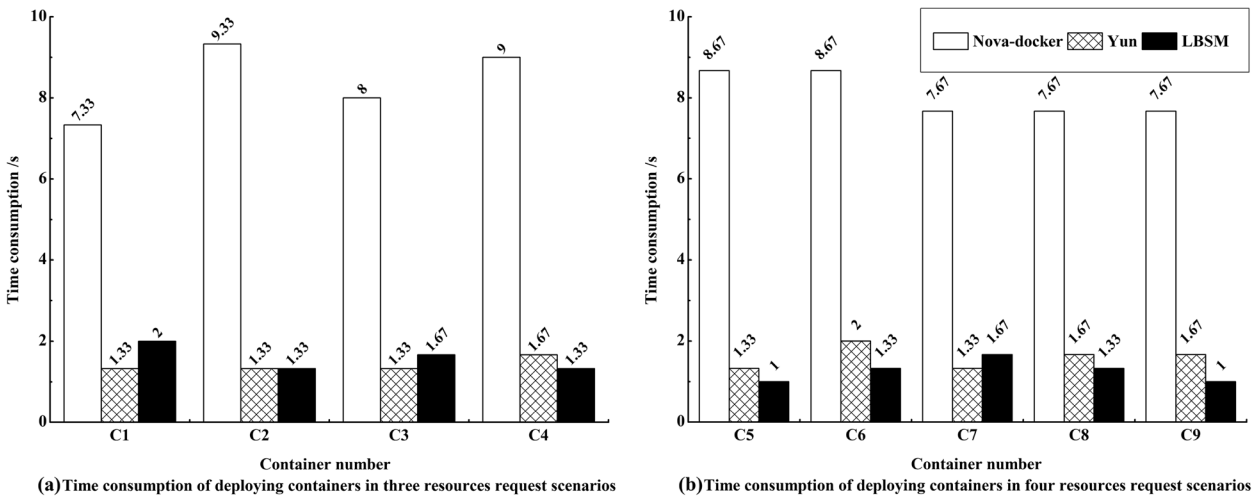

Fig. 7 Time consumption test results for deploying containers of different specifications

This method of allocating computing node resources leads to resource waste. And the alternatives of docker are also available which is highly used nowadays such as CoreOS Rocket, RKT. The next step will be to apply LBSM to OpenStack and other alternatives of Docker integration. Then, we will establish a Docker container scheduling model based on the OpenStack cloud platform and to study how to improve system resource utilization and reduce energy consumption while achieving load balancing of OpenStack computing node resources.

Abbreviations

NUMA	Non-uniform memory access
IaaS	Infrastructure as a Service
LBSM	Load-Balancing Scheduling Mechanism
ANN	Artificial neural network
DRL	Deep reinforcement learning
SLA	Service level agreement
IMC	Integrated memory controller
QPI	Quick Path Interconnect
CV	Coefficient of variance

Acknowledgements
The authors thank the editor and anonymous reviewers for their helpful comments and valuable suggestions.

Authors' contributions
Jiarui Qian and Xiaoxue Wang completed most of the writing of this manuscript and conducted the experiment. Yong Wang and Peng Zhang found the target problem from his working experience. Xiaofeng Wang took part in the discussion of the solution and gave many useful suggestions. All authors have read and approved the manuscript.

Funding
The work is supported by the National Natural Science Foundation of China (Grant Nos.62172191 and 61972182).

Availability of data and materials
Not applicable.

Declarations

Ethics approval and consent to participate
The work has not been published elsewhere nor is it currently under review for publication elsewhere.

Consent for publication
Informed consent was obtained from all individual participants included in the study.

Competing interests
The authors declare that they have no competing interests.

Received: 23 December 2022 Accepted: 16 April 2023
Published online: 28 April 2023

References

1. Ren J, Yu G, He Y, Li GY (2019) Collaborative cloud and edge computing for latency minimization. IEEE Trans Veh Technol 68(5):5031–5044
2. Li M, Gao J, Zhao L, Shen X (2020) Deep reinforcement learning for collaborative edge computing in vehicular networks. IEEE Trans Cogn Commun Netw 6(4):1122–1135
3. Ren J, He Y, Yu G, Li GY (2019) Joint communication and computation resource allocation for cloud-edge collaborative system. In: 2019 IEEE Wireless Communications and Networking Conference (WCNC), IEEE, pp 1–6
4. Al-Dhuraibi Y, Paraiso F, Djarallah N, Merle P (2017) Elasticity in cloud computing: state of the art and research challenges. IEEE Trans Serv Comput 11(2):430–447
5. Fan W, Han Z, Li P, Zhou J, Fan J, Wang R (2019) A live migration algorithm for containers based on resource locality. J Sig Process Syst 91(10):1077–1089
6. Benomar Z, Longo F, Merlino G, Puliafito A (2021) Cloud-based network virtualization in iot with openstack. ACM Trans Internet Technol (TOIT) 22(1):1–26
7. Shih WC, Yang CT, Ranjan R, Chiang CI (2021) Implementation and evaluation of a container management platform on docker: Hadoop deployment as an example. Clust Comput 24(4):3421–3430
8. Annie Poornima Princess G, Radhamani A (2021) A hybrid meta-heuristic for optimal load balancing in cloud computing. J Grid Comput 19(2):1–22
9. Ghasemi A, Toroghi Haghighat A (2020) A multi-objective load balancing algorithm for virtual machine placement in cloud data centers based on machine learning. Computing 102(9):2049–2072
10. Yang S, Wang X, An L, Zhang G (2019) Yun: a high-performance container management service based on openstack. In: 2019 IEEE Fourth International Conference on Data Science in Cyberspace (DSC), IEEE, pp 202–209
11. Yang S, Wang X, Wang X, An L, Zhang G (2020) High-performance docker integration scheme based on openstack. World Wide Web 23(4):2593–2632
12. Mao Y, Oak J, Pompili A, Beer D, Han T, Hu P (2017) Draps: Dynamic and resource-aware placement scheme for docker containers in a heterogeneous cluster. In: 2017 IEEE 36th International Performance Computing and Communications Conference (IPCCC), IEEE, pp 1–8
13. Ben Alla H, Ben Alla S, Ezzati A, Touhafi A (2021) A novel multiclass priority algorithm for task scheduling in cloud computing. J Supercomput 77(10):11514–11555
14. Lin W, Wang Z (2018) Docker cluster scheduling strategy based on genetic algorithm. J South China Univ Technol (Nat Sci Ed) 46(3):127–13
15. Shen B, Li Q, Jiang Y, Wang Y et al (2020) Research on load balancing in data center networks. J Softw 31(7):2221–2244
16. Chen G, Zhang W (2019) Elab: end-host-based congestion aware load balancing for data center network. J Commun 40(03):196–205
17. Kristiani E, Yang CT, Huang CY, Wang YT, Ko PC (2021) The implementation of a cloud-edge computing architecture using openstack and kubernetes for air quality monitoring application. Mob Netw Appl 26(3):1070–1092
18. Li Y, Qi F, Wang Z, Yu X, Shao S (2020) Distributed edge computing offloading algorithm based on deep reinforcement learning. IEEE Access 8:85204–85215
19. Negi S, Rauthan MMS, Vaisla KS, Panwar N (2021) Cmodlb: an efficient load balancing approach in cloud computing environment. J Supercomput 77(8):8787–8839
20. Tong Z, Deng X, Chen H, Mei J (2021) Ddmts: A novel dynamic load balancing scheduling scheme under sla constraints in cloud computing. J Parallel Distrib Comput 149:138–148
21. Pérez de Prado R, García-Galán S, Muñoz-Expósito JE, Marchewka A, Ruiz-Reyes N (2020) Smart containers schedulers for microservices provision in cloud-fog-iot networks. challenges and opportunities. Sensors 20(6):1714
22. Menouer T (2021) Kcss: Kubernetes container scheduling strategy. J Supercomput 77(5):4267–4293
23. Cheng Y, Chen W, Wang Z, Yu X (2015) Performance-monitoring-based traffic-aware virtual machine deployment on numa systems. IEEE Syst J 11(2):973–982
24. Wu T, Chen X, Liu K, Xiao C, Liu Z, Zhuge Q, Sha EHM (2020) Hydrafs: an efficient numa-aware in-memory file system. Clust Comput 23(2):705–724

Publisher's Note