IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received 10 August 2023, accepted 4 September 2023, date of publication 11 September 2023, date of current version 22 September 2023.

Digital Object Identifier 10.1109/ACCESS.2023.3313643

RESEARCH ARTICLE

K-TAHP: A Kubernetes Load Balancing Strategy Base on TOPSIS+AHP

RONG GAO[1], XIAOLAN XIE[1,2], (Member, IEEE), AND QIANG GUO[2]
[1]School of Information Science and Engineering, Guilin University of Technology, Guilin, Guangxi 541006, China
[2]Key Laboratory of Embedded Technology and Intelligent System of Guangxi, Guilin, Guangxi 541006, China

Corresponding authors: Xiaolan Xie (xie_xiao_lan@foxmail.com) and Qiang Guo (guoqiang121@163.com)

This work was supported in part by the National Natural Science Foundation of China under Grant 62262011, in part by the Natural Science Foundation of Guangxi under Grant 2021JIA17013, and in part by the China Computer Federation (CCF)-Zhipu Artificial Intelligence (AI) Large Model OF 202225.

ABSTRACT Kubernetes is an orchestration platform designed for containerized applications,allows the application provider to scale automatically to match the flfluctuating intensity of processing demand. Container cluster technology is used to encapsulate, isolate, and deploy applications, addressing the issue of low system reliability due to interlocking failures. However, after running for a long time, Kubernetes clusters often suffer from uneven system load, leading to a performance decline. To address this issue, a load balancing strategy called K-TAHP, based on TOPSIS and AHP, is proposed. This strategy takes cpu, memory, and bandwidth usage as load factors and uses K-TAHP to construct load evaluation. By employing a warning module and a migration module, it migrates high-load pods from overloaded nodes to nodes with lower loads, thus improving load balancing in the Kubernetes cluster and resolving performance degradation caused by load imbalance after prolonged cluster operation. The experimental results show that the K-TAHP load balancing strategy can improve the load balancing capability of Kubernetes clusters by around 60%. It effectively resolves the issue of load imbalance that can occur after long periods of operation in Kubernetes clusters. Additionally, it ensures uninterrupted pod services during migration, thereby maintaining the overall performance of the cluster.

INDEX TERMS Cloud computing, Kubernetes cluster, container, load balancing, migration.

I. INTRODUCTION

The development of the Internet has brought about huge changes in cloud computing and has developed into an open collaborative business model that looks for services and further diversifies other energy sources.The cloud computing service system is divided into three levels: Infrastructure as a Service (IaaS), Software as a Service (SaaS), and Platform as a Service (PaaS) [1].

In complex computing environments, cloud computing can significantly improve the utilization of computing resources [2]. In recent years, virtualization has become a key technology in cloud computing [3]. A new virtualization technology is containerization, and with the rapid development of containerization technology, a vast amount of services are being migrated from monolithic architectures based on virtual machines to cloud-native architectures based on containers [4]. In containerization, an application can be packaged into a container, which supports its running on a multi-tenant host [5]. In large-scale systems, multiple containers are deployed throughout the system, which requires a container orchestration tool to deploy and manage container resources and services.

Kubernetes is one of the most popular and powerful open-source orchestration tools for container applications [6]. It provides various features for container orchestration, such as deployment, resource management, automatic scaling, and load balancing [7]. Pod is the smallest resource unit that can be created and managed in Kubernetes. It is the minimum resource object model created or deployed by users in the resource object model, and it is also the resource object running containerized applications on Kubernetes. Three types of automatic scalers are provided in Kubernetes: Horizontal Pod Autoscaler (HPA), Vertical Pod Autoscaler (VPA), and Cluster Autoscaler (CA).

The associate editor coordinating the review of this manuscript and approving it for publication was Taehong Kim.

- HPA monitors resource utilization and automatically increases or decreases the numberof pods for an application.
- VPA directly changes the requested resources for an application.
- CA increases or decreases the numberof nodes in the cluster, and CA supports commercial cloud platforms such as Google Cloud Platform (GCP) [8] and Amazon Web Services (AWS) [9].

II. RELATED WORK

In [10], Khaleq et al. proposed a cloud microservice intelligent autoscaling system for real-time applications, which automatically scales microservice applications in a cloud environment with QoS constraints. The system identifies microservice resource requests through a general autoscaling algorithm in the Google Kubernetes Engine (GKE) module, and then implements automatic scaling thresholds through a reinforcement learning agent module. However, their research did not consider the problem of load imbalance in the cluster after long-term operation, which will affect the performance of the entire intelligent autoscaling system.

Reference [11] proposed a hybrid autoscaling approach for containerized applications. The article uses a machine learning-based prediction method to predict the future requirements of applications and combines it with a burst identification module to make autoscaling decisions more effective. To avoid pods service interruption during vertical autoscaling, they proposed a rolling update module. During the vertical autoscaling process, the information of corresponding pods is obtained, and when the resource request of pods changes, the pods are redeployed on the node with normal working pods. After deployment, the traffic on the original Pods is forwarded to the newly deployed pods, and finally, the original pods are deleted. However, their research did not consider the problem of load balancing in the cluster, and load imbalance after a long time of cluster operation will lead to a decrease in module performance.

Reference [12] proposed a HANSEL system based on the Kubernetes platform. The system uses a Bi-LSTM load prediction algorithm to accurately predict the load of microservices and implements active elastic scaling by combining passive and active methods through reinforcement learning to optimize the horizontal autoscaling strategy of Kubernetes.

Reference [13] designed a Kubernetes autoscaler based on the pod replica prediction to improve resource scaling efficiency. Both [12] and [13] also did not consider the problem of load balancing of the entire cluster and pods, leading to a decrease in cluster performance.

Reference [14] proposed an internal architecture based on Kubernetes and Docker containers that can dynamically autoscale based on SLO requirements to improve resource utilization and ensure QoS.

Reference [15] proposed a general-purpose system to dynamically adjust the Kubernetes cluster scale to improve cluster resource utilization and ensure QoS by automatically determining cpu utilization threshold to meet the requirements of specific applications and providing a cluster autoscaling algorithm to obtain the ideal number of nodes in a Kubernetes cluster. However, it cannot guarantee the entire cluster resource utilization for specified node deployment applications.

Reference [16] did not use the default system metrics from the measuring server component in Kubernetes, but mentioned several application-level metrics that affect the performance of HPA while showing the direction for optimizing HPA.

Reference [17] proposed a new model for collaborative robots (CoBots) in robotics, artificial intelligence, and IoT devices. CoBots should develop a work ecosystem system and work together to achieve maximum productivity and consistency. This model is also applicable to machine-to-machine collaboration management in cloud environments and IoT devices.

Reference [18] proposed a novel intelligent control network to improve microgrid communication performance, which is used to solve typical drawbacks of a single SDN controller. Combining Kubernetes with SDN microservices can eliminate single-point of failure in the hierarchical control, shorten application recovery time, and enhance container security and portability.

Reference [19] designed and implemented a Kubernetes simulator named K8sSim, which can quickly obtain scheduling results of different scheduling algorithms, greatly reducing their scheduling time.

In the paper [20], the authors proposed a Kubernetes Scheduler (KubeSC-RTP) to reduce processing time and improve user satisfaction in heterogeneous environments. They introduced a machine learning-based approach using runtime prediction to better select appropriate cpu or gpu resources. Similarly, in the paper [21], a Kubernetes scheduling platform (KubCG) was proposed, which manages the deployment of Docker containers in heterogeneous clusters. The platform implements a new scheduler that reduces the completion time for different tasks to 64% of the original time.

Manzoor et al. proposed a suitable wireless load definition and analyzed the performance of SD-WiFi by varying the load conditions [22]. Additionally, they also introduced a QoS-aware load balancing strategy (QALB) for software-defined wireless networks (SD-WiFi) as a solution to address the Wi-Fi congestion among OpenFlow-enabled access points (OAPs) [23].

These studies did not focus on the problem that load imbalance in a Kubernetes cluster after long-term operation will lead to a decrease in cluster performance.

III. INTRODUCTION TO KUBERNETES

Kubernetes is an open-source container orchestration platform used to automate the deployment, scaling, and

management of containerized applications. In the Kubernetes platform, a pod is the smallest deployable unit.

A. ARCHITECTURE OF KUBERNETES

A Kubernetes cluster consists of master nodes and worker nodes, as shown in Figure 1. Each node can run on a physical machine or on a virtualized environment. By default, there is usually one master node that controls the entire cluster, but multiple master nodes can be deployed to achieve high availability of the cluster.

The master node consists of kube-apiserver,kube-scheduler,kube-controller-manager, and etcd [6].

- kube-apiserver is used to control the entire Kubernetes cluster. It can communicate with all other components in the cluster and receive requests from them. kube-apiserver also interacts with worker nodes through kubelet. Additionally, users can pass commands to kube-apiserver in the master node through kubectl.
- kube-controller-manager monitors and ensures the running state of the cluster. For example, if an application creates 10 replicas and one of them is deleted or lost, kube-controller-manager must ensure that a new replica is created.
- kube-scheduler is responsible for finding unscheduled pods and scheduling them to appropriate nodes in the cluster.
- etcd is a high-availability, distributed, and consistent backend database used to store data in the cluster.

Applications are deployed on worker nodes. Each worker node is managed by master node and consists of kubelet, container runtime (such as Docker [24]), and kube-proxy components [6].

- kubelet is responsible for managing the containers running on the machine. It reports the current status of the worker node to the Master and operates pods based on the instructions of kube-apiserver in the master node.
- Docker is the most commonly used container runtime in Kubernetes.
- kube-proxy maintains network rules that allow communication with pods from within or outside the cluster. Each pod is assigned a unique IP address when created. kube-proxy uses these IP addresses to forward traffic to pods.

B. KUBERNETES SERVICE

In a Kubernetes cluster, each pod can be accessed internally since it has a unique IP address. However, the lifecycle of pods in Kubernetes is ephemeral, as pods can be created or destroyed at any time, and a new IP address is assigned every time a new pod is created. Therefore, using the IP of pods in Kubernetes is not a good solution, as the IP addresses assigned to pods can only be accessed internally within the cluster and not from outside the cluster.

Kubernetes Service is an abstraction for a group of pods that allows access to deployed pods inside and outside the cluster. There are three types of Kubernetes Service:

- ClusterIP is the default type of Service created, and it can only be accessed within the cluster.
- NodePort is a reserved port on each node that a Service is exposed on. For example, if a NodePort Service is created for a group of pods with the label "App A", the NodePort would be 32321, and these pods can be accessed using NodeIP:32321. kube-proxy captures traffic arriving at port 32321 through corresponding iptables rules and forwards it to the ClusterIP, which eventually distributes the traffic to the backend pods.
- LoadBalancer is provided by specific cloud service providers.

Kubernetes provides several objects for managing replicas of applications.Deployment are used to manage statele-ss applications, while StatefulSets are used to manage stateful applications. Stateless applications do not require real-time data storage, meaning that data does not need to be synchronized in real-time between replicas. On the other hand, stateful applications require realtime data storage, meaning that the data needs to be synchronized in real-time between replicas.

IV. K-TAHP LOAD BALANCING STRATEGY

In order to achieve Kubernetes load balancing after the Kubernetes cluster runs for a long time, a load balancing strategy K-TAHP based on TOPSIS+AHP is proposed.

A. BUILDING LOAD EVALUATION BASED ON TOPSIS+AHP

The K-TAHP load balancing strategy requires the ability to evaluate the load status of each working node and pod. Therefore, the cpu, memory, and bandwidth usage of each worker node and pods are obtained from the monitoring module and used as load factors. These load factors are evaluated using Analytic Hierarchy Process (AHP) to obtain load weight vectors for each factor. Finally, the TOPSIS technique is used to obtain the final load evaluation value based on the cpu, memory, and bandwidth usage values obtained.

To begin with, a pairwise comparison matrix of order n is constructed. This matrix is used to represent the relative superiority of the selected indicators. The general form of the matrix is as follows:

$$\mathrm{A} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & & a_{2n} \\ \cdots & \cdots & \cdots & \cdots \\ a_{n1} & a_{n1} & \cdots & a_{nn} \end{bmatrix}$$

where a_{ij} represents the importance comparison result of a_i to a_j. Table 1 shows the 9 importance levels and their corresponding values.

To obtain the relative weights of the three load factors by solving the judgment matrix A, the steps are as follows:

1. The column vectors of the normalized matrix A.:

$$A' = a_{ij} / \sum_{i=1}^{n} a_{ij} \tag{1}$$

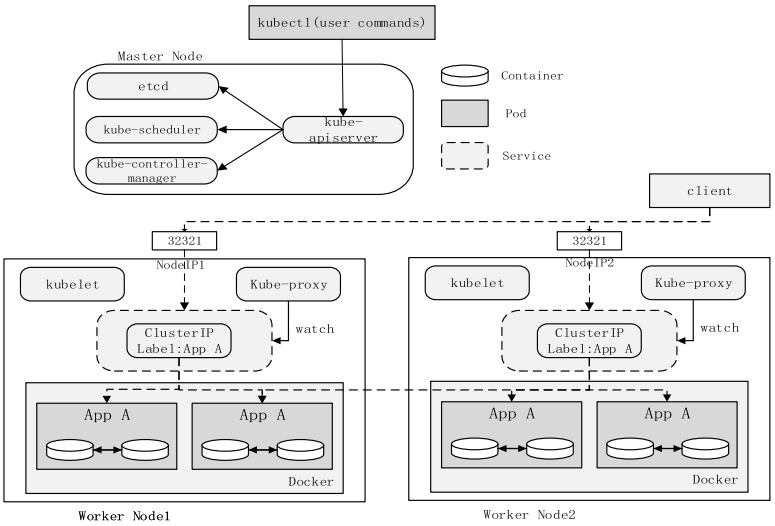


FIGURE 1. The architecture of kubernetes.

TABLE 1. a_{ij} value ratio and meaning.

a_{ij} value ratio	meaning
1	The load factor of the former and latter are equally important.
3	The load factor of the former is slightly more important than the latter.
5	The load factor of the former is relatively more important than the latter.
7	The load factor of the former is more strongly important than the latter.
9	The load factor of the former is more critical than the latter.
2, 4, 6, 8	The importance level is somewhere between the above mentioned values.

2. To sum up the matrix A′ by rows.:

$$W' = \left(\sum_{j=1}^{n} a_{1j}, \sum_{j=1}^{n} a_{2j}, \ldots, \sum_{j=1}^{n} a_{nj}\right)^{T} \quad (2)$$

3. To normalize W′ to obtain the weight vector W:

$$W = (w_1, w_2, \ldots, w_n)^{T} \quad (3)$$

4. To obtain the maximum eigenvalue of the judgment matrix A:

$$\lambda_{max} = \sum_{i=1}^{n} ((AW)_i / nw_i) \quad (4)$$

The consistency ratio (CR) is used to test the consistency of the judgment matrix A, and its formula is shown in equation (5):

$$CR = \frac{CI}{RI} \quad (5)$$

TABLE 2. The random consistency indicator RI.

n	1	2	3	4	5	6
RI	0	0	0.58	0.90	1.12	1.24

where CI is the consistency index introduced in AHP. When CI is larger, the inconsistency of the judgment matrix is more severe. When CI= 0, the judgment matrix is completely consistent. To measure the size of CI, the random consistency indicator (RI) is introduced.

The formula for calculating CI is shown in equation (6):

$$CI = \frac{\lambda_{\max} - n}{n - 1} \quad (6)$$

The values of RI are as shown in Table 2:

When the consistency ratio CR is less than 0.1, it indicates that the inconsistency of the judgment matrix A is within an acceptable range, and it can pass the consistency test. Otherwise, it is necessary to reconstruct the judgment matrix A to improve its consistency.

In Kubernetes, cpu, memory, and bandwidth usage can be used as load factors for monitoring and managing clusters. A judgment matrix A can be constructed to represent the relative importance between these load factors using AHP:

$$A = \begin{bmatrix} 1 & 2 & 4 \\ 1/2 & 1 & 2 \\ 1/4 & 1/2 & 1 \end{bmatrix}$$

Here, a_i represents load factors, a_1 represents cpu usage, a_2 represents memory usage, and a_3 represents bandwidth usage.

Through equations (1), (2), and (3),obtain the weight vector of A:W=(0.5714,0.2857,0.1429) and by using equations (4), (5), and (6), we obtained CR $= 7.0385 * 10^{-8} < 0.1$, indicating that the judgment matrix A satisfies consistency.

To obtain the load evaluation of each node using the TOPSIS method, the following steps can be taken:

1. Standardize the obtained load data of each node:

$$Z_i = \frac{b_i}{\sqrt{\sum_{i=1}^{n} b_i^2}} \tag{7}$$

In this context, where b_i represents the load data of a node, b_1 represents the cpu usage, b_2 represents the memory usage, and b_3 represents the bandwidth usage.

2. Obtain load evaluation for each node:

$$L_{cn} = (L_1, L_2, \ldots, L_n) \tag{8}$$

There, L_{cn} is closer to 1, it indicates a higher node load, and L_{cn} is smaller, it indicates a lower node load.The calculation method for L_i is as follows:

$$L_i^+ = \sqrt{\sum_{i=1}^{n} (w_i \times (b_i - 1)^2)} \tag{9}$$

$$L_i^- = \sqrt{\sum_{i=1}^{n} (w_i \times b_i^2)} \tag{10}$$

$$L_i = \frac{L_i^-}{L_i^+ + L_i^-} \tag{11}$$

At the same time, the load evaluation L_{cp} of pods is calculated. A larger L_{cp} indicates a larger pod load, while a smaller L_{cp} indicates a smaller pod load.The calculation formula is shown in (12):

$$L_{cp} = b \times W \tag{12}$$

B. THE IMPLEMENTATION OF K-TAHP LOAD BALANCING STRATEGY

The K-TAHP load balancing strategy involves migrating high load Pods from highly loaded worker nodes to corresponding low load worker nodes in order to balance the load and ensure overall load balancing of the cluster. Therefore, a warning module and a migration module are designed and implemented to respectively detect and alert unbalanced load in the cluster and execute the migration operation.

1) THE WARNING MODULE

In order to determine whether the Kubernetes cluster load is unbalanced, it is necessary to dynamically and real-time obtain the cpu, memory, and bandwidth usage of each worker node from the monitoring module.To avoid immediate pods migration when a worker node reaches a peak overload, a warning threshold is set, and the threshold calculation is performed every 5 minutes. Because application workloads may reach a peak in a short period, there is no need to perform pods migration during this time because it generates additional system overhead, such as cpu and bandwidth. When a worker node exceeds the warning threshold calculated three times, it is identified as a highly loaded node and added to the list of highly loaded nodes. At the same time, the migration module is notified to perform pods migration. The formula (13) is used to calculate the warning threshold:

$$Threshold = \delta \times \left(\sum_{i=1}^{n} L_{ni}/n\right) \tag{13}$$

In the formula, δ represents the load balancing factor, which is used to evaluate whether the worker node is a highly loaded node or a low loaded node. To ensure that the highly loaded node is selected and there are sufficient lists of low-loaded nodes, δ is set to 1.25 and 0.75 for highly loaded nodes and low-loaded nodes respectively.

2) THE MIGRATION MODULE

The migration module is responsible for moving high-load pods from the high-load node list to the low-load node list to ensure load balancing in the cluster. The main implementation steps are as follows:

1) Get the list of high-load worker node L_{high} using formula (13), and if the list is not empty, proceed to step 2;
2) Get the list of low-load worker node nodes L_{low} using formula (13), and if the list is not empty, proceed to step 3;
3) Iterate through the L_{high} list, calculate the load of each pod on each high-load node using formula (12), add it to the L_{pl} list, and then sort the resulting L_{pl} list in descending order of pod load;
4) To ensure that high-load pods deployed on nodes specified by label tags can be migrated, select the first 1/3 length of the L_{pl} list as L'_{pl}. This can ensure that pods deployed on nodes specified by label tags can be migrated;
5) Iterate through the L_{pl}' list and, in order to minimize the number of pod migrations as much as possible, perform parallel pod migrations based on the length of L_{low} list (S_{low}) to reduce redundant calculations. This ensures the efficiency of pod migration. In order to so-lve the problem of service interruption of pods during migration, before migrating a pod, a copy of the pod is made and the copied pod is deployed on the low-load node to be migrated to. After the copy is deployed, the original pod on the original node is deleted;
6) Throughout the migration process, prioritize migrating pods deployed on non-specified nodes to the low-load node list. Add pods deployed on the specified nodes with label tags to the list L_{np} with tags. After the iteration is complete, proceed to step 7;
7) Determine whether to migrate pods deployed on non spe-cified nodes with the number of S_{low}. If not, iterate through the L_{np} list and migrate pods deployed on the specified nod-es until enough pods to meet the S_{low} number are migrated.

Algorithm 1 is K-TAHP Load Balancing Algorithm pseudocode.

V. EXPERIMENTAL RESULTS AND ANALYSIS

In order to verify the K-TAHP load balancing strategy, a Kubernetes cluster was deployed with Kubernetes

version 1.18.0 and Docker version 18.06.1. Prometheus [25] and Grafana [26] monitoring components were deployed in the cluster, with Prometheus used to real-time monitoring the cpu, memory, and bandwidth usage of each node, and Grafana used to visualize the data. The cluster consists of one master node and three worker nodes, and the configuration information for each node is listed in Table 3.

Algorithm 1 K-TAHP Load Balancing Algorithm

Input:cpu, memory and bandwidth usage
Output:pod migration to worker node information
1: L_{high} = getHighNode(Nodes) && L_{low} = getLowNode(Nodes) //get a list of high load and low load nodes
2: if notNull(L_{high}, L_{low})
3: while i ∈[0,M]
4: loadPod(pods) //calculate pods load
5: end while
6: while j ∈[0,N]
7: while k ∈[0,S]
8: if labelPod(pod_k)
9: add(pod_k)
10: else
11: copy(pod_k)
12: migrate(pod_k) //migrate pods with no label
13: end if
14: end while
15: end while
16: while l ∈[0,Q]
17: migrate(labelPod$_l$) //migrate pods with label
18: end while
19: end if

TABLE 3. Cluster nodes configuration.

Nodename	CPU	Memory	OS
master	4	2GB	CentOS7
node1	4	2GB	CentOS7
node2	4	2GB	CentOS7
node3	4	2GB	CentOS7

20 pods were deployed in the cluster, with 15 of them deployed on node1. Each pod requested 200m of cpu resources and 100Mi of memory resources. A Service object was created to expose these pods externally. To verify the effectiveness of the K-TAHP load balancing strategy, the Hey [27] testing tool was used to perform a load test on the deployed pods, making the workload of the entire cluster unbalanced.

Figures 2 and 3 show the changes in cpu and memory usage on each node in the cluster using the K-TAHP load balancing strategy. As shown in Figures 2 and 3, before using the K-TAHP load balancing strategy, the cpu and memory usage on node1, node2, and node3 in the cluster was

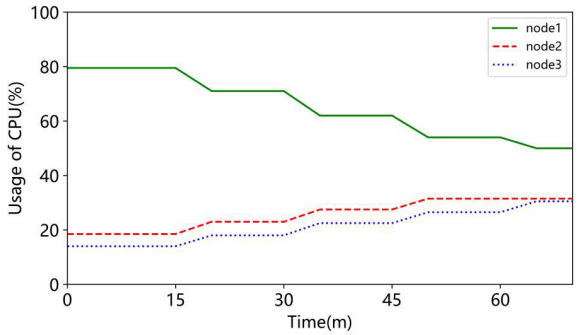


FIGURE 2. Cpu usage of each node in the cluster.

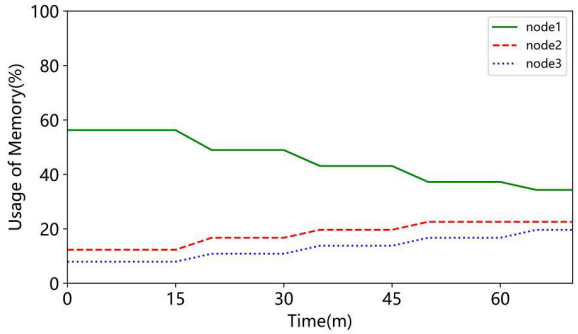


FIGURE 3. Memory usage of each node in the cluster.

TABLE 4. Number of pods with label in cluster nodes.

nodename	default	K-TAHP
node1	15	10
node2	0	2
node3	0	3

79.20%, 18.58%, and 12.32%, 56.27%, 14.00%, and 7.93%. The entire cluster was in an unbalanced load state.However, after using the K-TAHP load balancing strategy, the cpu and memory usage on node1, node2, and node3 in the cluster was 50.00%, 31.50%, and 30.50%, 34.30%, 22.58%, and 19.65%, respectively. It can be seen that the K-TAHP load balancing strategy can ensure that the entire Kubernetes cluster is in a load-balanced state.

To verify that the K-TAHP load balancing strategy can migrate pods with label, 15 pods with label were deployed on node1 using label tags. Table 4 shows the number of pods with label on node1, node2, and node3 in the cluster before and after using the K-TAHP load balancing strategy. According to the data provided in Table 4, it can be observed that the K-TAHP load balancing strategy can migrate pods with label to nodes with lower loads.

To validate whether the K-TAHP load balancing strategy can ensure uninterrupted service during pod migration, Hey was used to test the throughput of pod during the

migration process. Figure 4 presents the throughput of pod during migration under different numbers of clients. It can be observed that as the number of clients increases, the throughput generated during pod migration is nearly the same as the throughput generated under default conditions. Therefore, Figure 4 verifies that the K-TAHP load balancing strategy can guarantee uninterrupted service during pod migration, thereby preserving the performance of pod.

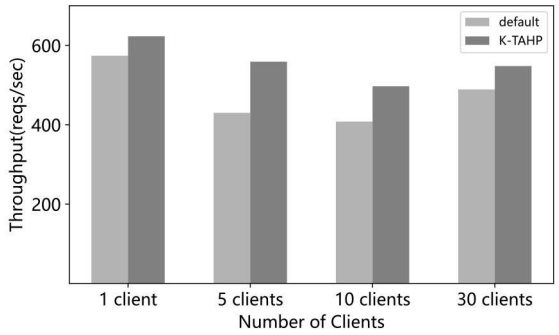


FIGURE 4. Pod throughput.

The K_{std} metric is used to measure the degree of load balancing in a cluster system, with a value of K_{std} approaching 0 indicating greater stability in the cluster system. The calculation formula for K_{std} is as follows:

$$K_{std} = \sqrt{\frac{\sum_{i=1}^{n} (L_i - \frac{\sum_{i=1}^{n} L_i}{n})^2}{n}} \tag{14}$$

Figure 5 illustrates the load balancing degree of the entire cluster system under default conditions, using the AHP strategy, and using the K-TAHP strategy. From Figure 5, it can be observed that as the number of deployed pods increases, the K-TAHP load balancing strategy is more effective in improving the load balancing degree of the cluster system compared to the AHP strategy. This ensures the overall performance of the cluster.

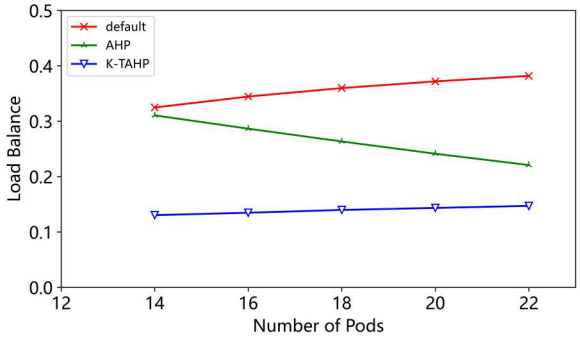


FIGURE 5. Load balancing of cluster system.

VI. CONCLUSION AND FUTURE WORK

The K-TAHP load balancing strategy utilizes cpu, memory, and bandwidth usage as load factors to construct load assessment. By employing a warning module and a migration module, it migrates high-load pods from overloaded nodes to nodes with lower loads, thus improving load balancing in the Kubernetes cluster and resolving performance degradation caused by load imbalance after prolonged cluster operation. Experimental results demonstrate that this strategy effectively resolves the problem of load imbalance in long-running Kubernetes clusters. Moreover, the strategy ensures uninterrupted service of Pods during the migration process, thereby preserving the performance of the cluster applications.

Future work will focus on the issue of elastic scaling of pods in Kubernetes, aiming to ensure the performance of pods through proactive scaling mechanisms.

REFERENCES

[1] F. Rossi, V. Cardellini, F. Lo Presti, and M. Nardelli, "Geo-distributed efficient deployment of containers with Kubernetes," *Comput. Commun.*, vol. 159, pp. 161–174, Jun. 2020.

[2] A. Botta, W. De Donato, V. Persico, and A. Pescape, "Integration of cloud computing and Internet of Things: A survey," *Future Gener. Comput. Syst.*, vol. 56, pp. 684–700, Mar. 2016.

[3] C. Arango, R. Dernat, and J. Sanabria, "Performance evaluation of container-based virtualization for high performance computing environments," 2017, *arXiv:1709.10140*.

[4] D. Gannon, R. Barga, and N. Sundaresan, "Cloud-native applications," *IEEE Cloud Comput.*, vol. 4, no. 5, pp. 16–21, Sep. 2017.

[5] C. Pahl, A. Brogi, J. Soldani, and P. Jamshidi, "Cloud container technologies: A state-of-the-art review," *IEEE Trans. Cloud Comput.*, vol. 7, no. 3, pp. 677–692, Jul. 2019.

[6] *Kubernetes*. Accessed: Jun. 10, 2023. [Online]. Available: https://www.kubernetes.io

[7] N. Nguyen and T. Kim, "Toward highly scalable load balancing in Kubernetes clusters," *IEEE Commun. Mag.*, vol. 58, no. 7, pp. 78–83, Jul. 2020.

[8] *Google Cloud Platformhttps*. Accessed: Jun. 10, 2023. [Online]. Available: https://cloud.google.com,

[9] *Amazon Web Service*. Accessed: Jun. 10, 2023. [Online]. Available: https://aws.amazon.com

[10] A. A. Khaleq and I. Ra, "Intelligent autoscaling of microservices in the cloud for real-time applications," *IEEE Access*, vol. 9, pp. 35464–35476, 2021.

[11] D.-D. Vu, M.-N. Tran, and Y. Kim, "Predictive hybrid autoscaling for containerized applications," *IEEE Access*, vol. 10, pp. 109768–109778, 2022.

[12] M. Yan, X. Liang, Z. Lu, J. Wu, and W. Zhang, "HANSEL: Adaptive horizontal scaling of microservices using bi-LSTM," *Appl. Soft Comput.*, vol. 105, Jul. 2021, Art. no. 107216.

[13] T. Hu and Y. Wang, "A Kubernetes autoscaler based on pod replicas prediction," in *Proc. Asia–Pacific Conf. Commun. Technol. Comput. Sci. (ACCTCS)*, Jan. 2021, pp. 238–241.

[14] L. M. Ruíz, P. P. Pueyo, J. Mateo-Fornés, J. V. Mayoral, and F. S. Tehas, "Autoscaling pods on an on-premise Kubernetes infrastructure QoS-aware," *IEEE Access*, vol. 10, pp. 33083–33094, 2022.

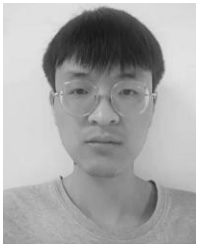
[15] Q. Wu, J. Yu, L. Lu, S. Qian, and G. Xue, "Dynamically adjusting scale of a Kubernetes cluster under QoS guarantee," in *Proc. IEEE 25th Int. Conf. Parallel Distrib. Syst. (ICPADS)*, Dec. 2019, pp. 193–200.

[16] T.-T. Nguyen, Y.-J. Yeom, T. Kim, D.-H. Park, and S. Kim, "Horizontal pod autoscaling in Kubernetes for elastic container orchestration," *Sensors*, vol. 20, no. 16, p. 4621, Aug. 2020.

[17] M. Niazi, S. Abbas, A. H. Soliman, T. Alyas, S. Asif, and T. Faiz, "Vertical pod autoscaling in Kubernetes for elastic container collaborative framework," *Comput., Mater. Continua*, vol. 74, no. 1, pp. 591–606, 2023.

[18] R. Pérez, M. Rivera, Y. Salgueiro, C. R. Baier, and P. Wheeler, "Moving microgrid hierarchical control to an SDN-based Kubernetes cluster: A framework for reliable and flexible energy distribution," *Sensors*, vol. 23, no. 7, p. 3395, Mar. 2023.

[19] S. Wen, R. Han, K. Qiu, X. Ma, Z. Li, H. Deng, and C. H. Liu, "K8sSim: A simulation tool for Kubernetes schedulers and its applications in scheduling algorithm optimization," *Micromachines*, vol. 14, no. 3, p. 651, Mar. 2023.
[20] I. Harichane, S. A. Makhlouf, and G. Belalem, "KubeSC-RTP: Smart scheduler for Kubernetes platform on CPU-GPU heterogeneous systems," *Concurrency Comput., Pract. Exper.*, vol. 34, no. 21, p. e7108, Sep. 2022.
[21] G. E. H. Ahmed, F. Gil-Castiñeira, and E. Costa-Montenegro, "KubCG: A dynamic Kubernetes scheduler for heterogeneous clusters," *Softw., Pract. Exper.*, vol. 51, no. 2, pp. 213–234, Feb. 2021.
[22] S. Manzoor, C. Zhang, X. Hei, and W. Cheng, "Understanding traffic load in software defined WiFi networks for healthcare," in *Proc. IEEE Int. Conf. Consum. Electron.*, May 2019, pp. 1–2.
[23] S. Manzoor, Z. Chen, Y. Gao, X. Hei, and W. Cheng, "Towards QoS-aware load balancing for high density software defined Wi-Fi networks," *IEEE Access*, vol. 8, pp. 117623–117638, 2020.
[24] *Docker*. Accessed: Jun. 10, 2023. [Online]. Available: https://www.docker.com
[25] *Prometheus*. Accessed: Jun. 10, 2023. [Online]. Available: https://prometheus.io
[26] *Grafana*. Accessed: Jun. 10, 2023. [Online]. Available: http://grafana.com
[27] *Hey*. Accessed: Jun. 10, 2023. [Online]. Available: https://github.com/rakyll/hey



RONG GAO received the B.E. degree from Anhui Agricultural University, Hefei, China, in 2021. He is currently pursuing the master's degree in computer technology with the Guilin University of Technology, Guilin. His research interests include cloud computing and cloud native.

XIAOLAN XIE (Member, IEEE) received the M.S. degree in computer science from Shanghai Maritime University, in 2001, and the Ph.D. degree in mechanical manufacturing and automation from Xidian University, China, in 2009. She is currently a Senior Visiting Scholar with Middlesex University, U.K. She is also the Deputy Director of the Guangxi Key Laboratory of Embedded Technology and Intelligent System and the Dean of the School of Information Science and Engineering, Guilin University of Technology. She has published more than 100 scientific research articles, including more than 50 SCI/EI papers and more than 20 Chinese core journals. Her research interests include cloud computing, big data, intelligent computing, and manufacturing informatization. She is a fellow of the American Computer Society (ACM), the Chinese Computer Society (CCF), the Cloud Computing Expert Committee of the Chinese Institute of Communications (CIC), the High Performance Computing Expert Committee of CCF, and the Collaborative Computing, Distributed Computing and Processing Expert Committee of CCF. She is also the Director of the International Institute of Engineering and Technology (IETI).

QIANG GUO received the B.E. degree from the Liaoning University of Petroleum and Chemical Technology, China, in 2001, and the master's degree in computer application technology from Guizhou University, Guiyang, China, in 2005. He is currently a Lecturer with the Information Science and Engineering School, Guilin University of Technology. His research interests include pattern recognition and cloud computing.