Computer Communications 158 (2020) 73–84

Contents lists available at ScienceDirect

Computer Communications

journal homepage: www.elsevier.com/locate/comcom

Resource scheduling for piano teaching system of internet of things based on mobile edge computing

Yu Xia

Department of Music and Dance college, Huaihua University, China

ARTICLE INFO

Keywords:
Edge computing
Internet of things
Piano teaching
System resources
Resource scheduling

ABSTRACT

The effective operation of the piano teaching system of the Internet of Things requires the effective support of virtualization technology. In particular, on the basis that the edge computing standards and systems are not yet mature, the resource scheduling problem of edge computing needs to be studied from the actual point of view. In order to improve the effective operation of the piano teaching system of Internet of Things, this study analyzes the resource scheduling of delay-sensitive applications, sets the resource scheduling mode based on the space–time difference of the edge container load in a multi-cluster environment, and proposes a cross-cluster scheduling strategy. Simultaneously, this study uses simulation experiments to analyze the performance of the strategy proposed in this paper. The research results show that the strategy proposed in this paper can perform delay-insensitive application scheduling during system operation, achieve multi-cluster collaborative scheduling goals, and make the load between clusters more balanced.

1. Introduction

The Internet of Things is a core component of the intelligent teaching information platform. It uses advanced sensing technology, network technology, computing technology, control technology, and intelligent technology to comprehensively perceive the teaching process through cameras and other devices. Moreover, it aggregates its collected environment and status information, etc., and conducts large-scale, large-capacity data transmission and interaction between multiple systems to analyze and process the teaching information in a timely and sufficient manner, thereby supporting full control of each student, and ultimately effectively improving teaching efficiency.

In the past ten years, mobile cloud computing (MCC) technology has experienced rapid development. Due to high reliability, good scalability, and strong computing power, the technology is used as the basic support for mobile applications, from leading giants to startup small and micro enterprises. For example, the professional cloud service of international e-commerce giant Amazon (Amazon Web Services, AWS) has provided cloud platform support to thousands of enterprises in more than 190 countries and regions by deploying data centers around the world and more than 2 million cloud servers. Nowadays, on the new front of the industry 4.0 era, mobile applications are facing increasingly stringent service quality requirements [1] (Quality-of-Service, QoS). Moreover, the era of big data 2.0 will fuel the blowout growth of massive data [2]. In addition, the maturity and implementation of 5G bearer technology will inevitably make the core network more severely tested [3]. In this situation, MCCs that provide centralized services will likely be trapped, and it is difficult to achieve the ambitious vision of a millisecond response [4]: On the one hand, real-time communication is difficult to achieve due to network congestion caused by massive requests rushing to data centers. On the other hand, the heavy load of the data center makes it very easy to become a service bottleneck that delays the response. What is more, the high expansion cost of data centers is obviously not suitable for the exponentially rising demand scale. Therefore, the design needs of the next-generation cloud service system are imminent and eager, which has stimulated the enthusiasm of researchers at home and abroad [5]. In recent years, a new computing method called Mobile Edge Computing (MEC) has dawned and is expected to take on the responsibility of the next generation of cloud service systems. The European Telecommunications Standardization Organization defines MEC as "a cloud platform that provides services directly in the wireless access network close to users and provides high-bandwidth, low-latency computing and storage services to mobile users near data sources" [6]. In the future plan of MEC, the central cloud function of MCC will be "submerged" to tens of billions of network edge devices, making it "edge cloud server". These servers not only have the basic network functions of network access and forwarding traffic, but also undertake hosting services and cloud computing functions for processing requests and provide mobile users with "edge cloud services" that can be enjoyed with a single hop network distance [7]. With the blessing of 5G bearer technology, MEC will definitely show great promise in the fields of emerging mobile applications, smart cities, industrial Internet of things, and smart homes [8]. From the perspective

E-mail address: yqzheng19@lzu.edu.cn.

https://doi.org/10.1016/j.comcom.2020.04.056
Received 7 February 2020; Received in revised form 29 March 2020; Accepted 27 April 2020
Available online 29 April 2020

of the network structure, the MEC service system located near the terminal access network not only offloads the workload for the MCC data center, relieves network congestion, and improves service quality for the MCC data center, but also makes up the shortcomings of the MCC in computing response speed and content distribution efficiency, and improves the user experience. Compared with substitution and innovation, MEC is complementary to MCC and has complementary advantages, and the two will work together to create a rich and diverse cloud service ecosystem.

Today, the "second half" of the mobile Internet industry has begun: Pervasive computing is in the ascendant, the Internet of Everything is beginning to flourish, artificial intelligence is about to fall, and complex and emerging applications are increasingly mobile. What is more, resource-intensive, delay-sensitive computing tasks such as view processing, big data technology, and reinforcement learning have imposed severe requirements on computing device storage, storage, and battery life. However, consumer-grade mobile terminals are limited by the shortage of computing power resources and it is difficult to meet high standards of service alone.

In order to resolve the contradiction between increasing application requirements and limited resources, an effective solution is to enable users to submit some or all of the computing tasks to resource-rich cloud servers for processing. The aforementioned delegated execution process is called Computing Offloading. In the near future, due to its deployment advantages and geographical advantages, MEC is bound to become a mainstay in providing computing offload services. Computing offloading technology has important value: For mobile users, computing offloading technology expands the computing capabilities of mobile devices, saves energy consumption costs, extends its running time, and improves user experience. For operators and service providers, computing offloading technology makes full use of network resources, improves production efficiency, and can create huge benefits. In the field of MEC research, computational offloading technology has received much attention due to its higher research value. To sum up, the research topic in this paper is both practical and theoretical. Moreover, this study conducts research and analysis on resource scheduling of the piano teaching system of the Internet of Things based on mobile edge computing.

2. Related work

Mobile Cloud Computing (MCC), as an example of the organic combination of communication networks and the Internet, is the most effective way to provide mobile users with cloud computing [9]. Typical business applications are such as Apple iCloud and Microsoft OneDrive. MCC service providers provide users with cloud services through a centrally operated data center, and allow users to lease resources on demand, dynamically deploy applications, and enjoy computing services and storage resources [10].

Fan Q et al. [11] used algorithms based on Lyapunov optimization to make it difficult to achieve millisecond-level real-time response in such problems as energy but high computational load and core network congestion. It may be difficult to adapt to the widely existing high real-time application requirements in the future. To make up for the shortcomings of MCC, the idea of Mobile Edge Computing (MEC) is to locally deploy computing and storage resources near users, and to provide "near-end cloud" services without going through the core network in a distributed manner. Researchers at Carnegie Mellon University in the United States first proposed and implemented the predecessor of MEC, the Cloudlet platform [12]. The fog computing model between the central cloud and on-premises was first proposed by Cisco [13]. The "White Paper on Mobile Edge Computing" was formally released by the European Telecommunications Standardization Organization, which comprehensively elaborated and clearly defined the basic concepts of the MEC field [14] and verified the technical feasibility within the following two years [15]. At this point, the prelude to the vigorous development of MEC officially kicked off.

In recent years, academic circles have been enthusiastic about research in the field of MEC: at the theoretical level, a number of literature reviews [16] summarized the existing theoretical framework and architecture of MEC and looked forward to the future development of MEC; Mao Y et al. [17] investigated MEC supporting technologies, including technologies such as virtualization and software-defined networking; Zhang K et al. [18] classified MEC scenes, models, and deployment in detail. At the application level, Yang X et al. [19] expounded the application and value of MEC in the field of Internet of Things; Huang X [20] et al. Introduced and analyzed mobile edge computing under 5G-oriented requirements in detail, and Xu X et al. [21] implemented a smart home edge computing system on a personal computer platform. The industry is equally keen on MEC.IBM issued a paper discussing the value of MEC systems in enterprise applications [22]. In September of the same year, Nokia made the first attempt to promote the MEC service platform to large enterprises, providing low-latency services including assets, monitoring, and data [23]. In China, the "Edge Computing Industry Alliance" led by the China Academy of Information and Communication Technology was established in Beijing to accelerate the incubation of MEC best practices and promote the rapid development and landing of the MEC industry in China.

Computing offloading is a process in which mobile users delegate computing-intensive tasks to cloud computing platforms for execution. It has been fully and widely applied in the field of MCC [24]. In the field of MEC, computing offloading technology is the first of the three design elements of the MEC system (the other two major elements are resource allocation and mobility management), which can not only optimize resource utilization, but also reduce service delay, extend equipment life, and improve user experience, so it is also of great research value [25]. In general, the key to the design of the calculation offloading algorithm is to decide which tasks to uninstall and which devices to offload tasks to execute. Starting from the scope of computational offloading, Yang J et al. [26] divided the computational offloading model into two basic categories: single server model and multi-server model. Under the single-server computing offloading model, Yuan J et al. [27] used a convex optimization method to achieve the optimal computational offloading decision. Moreover, Al-Hammouri A T et al. [28] made the decision on the workload of local execution and offload execution by controlling the CPU frequency and transmission rate trade-off after offloading the decision. However, the above studies did not take into account the performance advantages of edge server collaboration and resource sharing for computing offload in the multi-server model. Under the multi-server computing offloading model, Elbamby M S et al. [29] proposed a multi-server computing offloading framework consisting of resource allocation, revenue management and service provider cooperation. Moreover, Qi Z et al. [30] studied a cloud service system composed of a single edge server and a single central cloud, and shared resources through server collaboration to provide more resources for mobile users. In addition to optimizing offloading decisions, Lin Y D et al. [31] also studied the computational migration between different remote servers. When a user moves into the access range of other MEC servers, the user's computing tasks are migrated to the server. However, the above research usually assumes that mobile users offload the entire computing load to the MEC server, without taking into account the flexibility advantages brought by partial offloading (some tasks are performed locally while others are offloaded). Under the partial computational offloading model, existing work [32] is more concerned with independent rather than dependent task models. For example, in literature [33], some tasks can be offloaded to the cloud for execution to ensure delay and energy consumption are minimized, but there is no dependency between tasks. However, in practice, most computationally intensive tasks or large-scale complex systems can often be disassembled into multiple dependent tasks [34]. At the same time, precision and efficient offloading of dependent tasks can also greatly increase the degree of parallelism in computing, reduce response time and improve resource utilization.

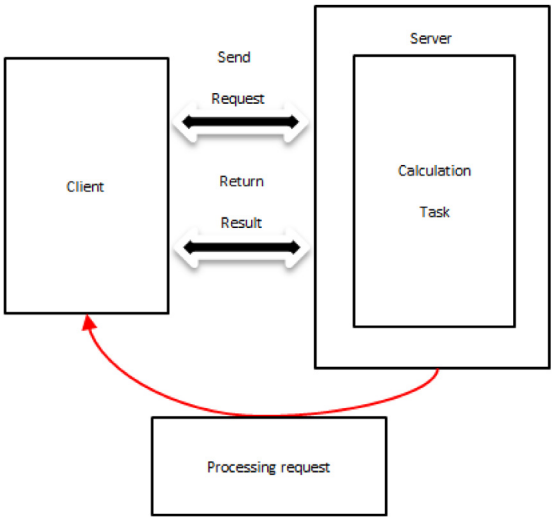

Fig. 1. Response delay of a delay-sensitive application request.

3. Cluster edge cloud resource scheduling strategy

For multi-cluster edge clouds, there are two major technical points that need attention: one is resource allocation, and the other is load balancing. This chapter will focus on the multi-cluster edge cloud framework, and respectively introduce the scheduling strategies of delay-sensitive applications and delay-insensitive applications in the framework to effectively implement resource allocation and solve load balancing problems.

3.1. Problem description and parameter definition

The characteristic of delay-sensitive applications is that it must respond to user requests within a specified time. The request process of a delay-sensitive application is shown in Fig. 1. Among them, t_1 is the user sending request time, t_2 is the application processing request time, and t_3 is the edge cloud return result to the user time. For each request, the edge application has to occupy a certain amount of system resources for processing. The calculation time t_2 can be estimated using Eq. (1). In the formula, $size$ is the amount of calculation required for the request, and $mips_v$ is the computing power obtained by the application from the edge cloud through the virtualization technology.

$$t_2 = \frac{size}{mips_v} \tag{1}$$

It can be seen that when the delay-sensitive application request strength increases, that is, when the number of requests per unit time increases, the required calculation size increases. However, because the application has limited computing power mips, the application's processing request time will increase, resulting in delay-sensitive applications not receiving a response within a specified time.

To cope with this change in request intensity, there are generally two strategies to increase the computing power $mips_v$ of the application: one is vertical expansion, and the other is horizontal expansion. Vertical expansion achieves resource expansion by increasing the application resource allocation size, that is, increasing the size of $mips_v$ obtained by the application in virtualization formula (1) through virtualization technology. Horizontal scaling expands applications by replicating applications and increasing the number of application copies, and then using random, polling, or consistent hashing to direct requests to different application copies. For a single application copy, horizontal expansion is equivalent to reducing the size in formula (1), thereby reducing the response delay of the request.

Because the vertical expansion changes the size of the application's resource quota, it may happen that the application's resource quota is too large and the actual total physical node resources are insufficient. In particular, in an edge environment, the resource capacity of a node is limited, and it is more likely that the node cannot complete resource allocation. In addition, after vertical expansion, the application often needs to be restarted, which will cause the application to be interrupted in a short time.

his paper chooses to use horizontal expansion to solve the resource scheduling problem of delay-sensitive applications. As shown in Eq. (2), in horizontal expansion, the overall computing power of the application is directly proportional to the number of copies owned by the application. Because the overall resources of each edge cluster are limited, a certain number of copies must be generated at different times according to the actual amount of resources required by the application to meet the resource requirements of the application at the current time.

$$mips = n \times mips_v \tag{2}$$

In summary, the resource scheduling problem of delay-sensitive applications can be seen as solving the problem of how many copies are generated for an application at any time and under what conditions.

The characteristic of the delay-insensitive application is that it is not strictly required to respond to the user's request within a specified time, but it must be deployed in the edge cloud for data preprocessing such as data filtering and desensitization. Under the condition of limited resources, edge clusters should first meet the horizontal expansion requirements of delay-sensitive applications in the cluster, and then meet the resource requests of delay-insensitive applications as much as possible.

Due to the spatio-temporal differences in the load of edge cloud clusters, in a multi-cluster edge cloud framework based on delay-sensitive characteristics, the coordination layer senses the load differences of different clusters to schedule delay-insensitive applications to clusters with low resource usage to achieve load balancing between clusters. The scheduling problem of delay-insensitive applications can be subdivided into the following sub-problems:

(1) The trigger timing of application scheduling. That is, when the coordination layer performs scheduling. In other word, what conditions the cluster meets will trigger the scheduling of delay-insensitive applications in the cluster.

(2) To-be-scheduled application selection mechanism. That is, after the edge cloud cluster triggers scheduling, what conditions are used to select applications to be scheduled.

(3) That is, after selecting the application to be scheduled, which criteria is used to schedule the application to which cluster.

(4) Destination node selection mechanism. That is, which strategy is used to schedule the application to which node of the cluster.

The (4)th problem can be determined through the collaboration layer, or it can be handed over to the container orchestration engine of the edge cluster. Considering that the edge cloud cluster is a highly autonomous system and when there are a large number of edge clusters connected, the global-based solution complexity is high, the normal operation of the cluster is also more disruptive. Therefore, this paper uses the container orchestration engine of the edge cloud cluster to solve it by itself.

In summary, the scheduling problem of delay-insensitive applications can be seen as solving the problem of scheduling applications to which clusters at any time and under what conditions.

The main purpose of this research is to describe the relationship and model of application resource scheduling under the multi-cluster edge cloud framework, and to define the parameters involved in the model.

Definition 1. This article takes the collaboration layer as the basic unit. We assume that a certain collaboration layer has access to a total of I edge clusters, denoted as $\Gamma = \{F_1, F_2, \ldots, F_I\}$. Among them, the edge node set in the ith edge cluster is $F_i = \{F_{i,1}, F_{i,2}, \ldots, F_{i,j}\}$.

Definition 2. The application set in the collaboration layer is $A = A^s \cup A^u \cup A^p$. Among them, A^s is a delay-sensitive application set, A^u is a delay-insensitive application set, and A^p is a system application set necessary for each edge cloud cluster to operate. The set of three applications is represented as:

$$\begin{aligned} A^s &= \left\{A_1^s, A_2^s, \ldots, A_N^s\right\} \\ A^u &= \left\{A_1^u, A_2^u, \ldots, A_M^u\right\} \\ A^p &= \left\{A_{F_1}^p, A_{F_2}^p, \ldots, A_{F_l}^p\right\} \end{aligned} \tag{3}$$

Delay-sensitive applications are only scheduled within the cluster and do not involve cross-clustering. Therefore, the same delay-sensitive applications in different clusters can be considered as irrelevant applications, and Definition 2 does not lose generality. For any delay-sensitive application, if it is assumed that there are K copies, then it is expressed as:

$$A_n^s = \left\{c_{n,1}, c_{n,2}, \ldots, c_{n,K}\right\} \tag{4}$$

Definition 3. There are four commonly used resources for edge nodes: CPU, RAM, storage, and network. This article only considers the two dimensions of resources that are most relevant to computing: CPU and RAM. The resource dimension in the system is defined as:

$$D = (cpu, ram) \tag{5}$$

The resources that a single node can provide are:

$$C_{f_{i,j}}^D = \left(C_{f_{i,j}}^{cpu}, C_{f_{i,j}}^{ram}\right) \tag{6}$$

The resources that the cluster can provide are:

$$C_{f_i}^D = \left(C_{f_i}^{cpu}, C_{f_i}^{ram}\right) \tag{7}$$

Definition 4. $\rho_{c_{n,k}}^{f_{i,j}} = 1$ represents the kth copy of delay-sensitive application n placed on the jth node in cluster i. $\rho_{A_m^u}^{f_{i,j}} = 1$ represents that the delay-insensitive application m is placed on the jth node in cluster i. $\rho_{A_m^u}^{C} = 1$ represents the time-insensitive application m is deployed in the cloud center when the edge clusters are busy.

Therefore, for delay-sensitive applications, the following formula exists:

$$\sum_{j \in J} \rho_{c_{n,k}}^{f_{i,j}} = 1 \tag{8}$$

For delay-insensitive applications, the following formula exists:

$$\rho_{A_m^u}^{C} + \sum_{i \in I} \sum_{j \in J} \rho_{A_m^u}^{f_{i,j}} = 1 \tag{9}$$

3.2. Scheduling strategies for delay-sensitive applications

In a multi-cluster edge cloud environment, as delay-sensitive application requests continue to change, the resources allocated to delay-sensitive applications within the cluster also change. When the request intensity changes dynamically, the number of application copies is automatically adjusted through horizontal expansion. This strategy is also called dynamic scaling.

Currently, many cloud vendors and open source container orchestration engines provide a rule-based responsive dynamic scaling method, such as target tracking extensions in AWS and horizontal application extension strategies in Kubernetes. This scaling method is often responsive. First, you need to set the target value of the resource, and then track and monitor the actual value of the resource. Then, by dynamically changing the number of copies of the resource, the two tend to be consistent and the dynamic scaling of the resource is achieved.

Among the responsive dynamic scaling strategies, the horizontal scaling strategy in Kubernetes is most widely used. In Kubernetes,

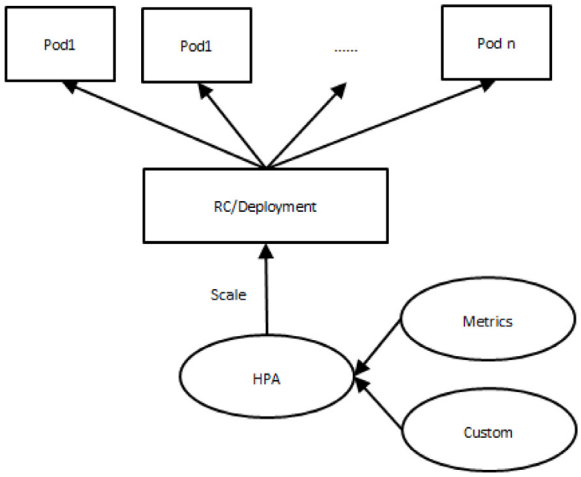

Fig. 2. HPA architecture diagram in Kubemetes.

HPA (Horizontal Pod Autoscaler) objects can be defined to start elastic scaling services for application resources. As shown in Fig. 2, each Pod represents a copy of the application, and RC/Deployment represents an application object. The HPA object regularly monitors the application's resource usage from Metrics Server or custom metrics. Moreover, according to the comparison of the target value of the current cycle resource with the actual value, it is judged whether it is necessary to increase or decrease the pod copy of the RC/Deployment object to realize the dynamic scaling of the resource.

The five basic parameters used by the HPA dynamic scaling strategy are shown in Table 1. Among them, the scaling indicator can be a resource indicator provided by Metrics serVer, or a custom indicator and an external indicator. The more commonly used resource indicators are CPU resource usage rate and RAM resource usage rate. By monitoring the resource usage of the TargetObiect replica set, the HPA autoscaling object is compared with a preset resource index TargetObiect to calculate the target replica number.

The following takes CPU resource usage as an example to introduce the specific algorithm steps of HPA implementation.

Step 1: The HPA object is created, TargetMertics (Target CPU resource usage), MaxReplicas, MinReplicas are set, and the monitoring period of the HPA controller is specified by -horizontal-pod-autoscaler-sync-period

Step 2: In each monitoring cycle, the HPA controller queries the Metircs server to obtain the CPU resource usage of each application copy. The CPU resource usage is an absolute value. The common unit is m, which represents one thousandth of a CPU.

Step 3: The formula (10) is used to calculate the application resource usage rate. In the formula, $metricValue_i$ is the ith copy resource usage of the application, n is the copy amount of the application, and requestValue is the resource demand of a single copy.

$$curMetricValue \frac{\sum_i^n metricValue_i}{n \times requestValue} \tag{10}$$

Step 4: By formula (11), the expected number of application replica sets is calculated.

$$Replicas = ceil\left[curReplicas \times \left(\frac{curMetricValue}{desiredMetricValue}\right)\right] \tag{11}$$

Step 5: The number of target copies $Replicas$ is compared with the maximum number of copies $MaxReplicas$ and the minimum number of copies $MinReplicas$. If $Replicas$ exceeds $MaxReplicas$, then $Replicas$ is set to $MaxReplicas$, and if $Replicas$ is less than $MinReplicas$, $Replicas$ is set to $MinReplicas$.

$$Replicas = \begin{cases} MinReplicas, Replicas \le MinReplicas \\ Replicas, MinReplicas < Replicas < MaxReplicas \\ MaxReplicas, Replicas \ge MinReplicas \end{cases} \tag{12}$$

In addition, in order to prevent frequent dynamic fluctuations of the evaluated metrics, the HPA policy sets the interval between two reductions to be not less than 5 min, and the interval between two expansions to be not less than 3 min.

It can be seen that the dynamic scaling strategy based on responsiveness is to determine the number of copies in the next cycle according to the resource usage of the previous monitoring cycle. This strategy has a certain lag in actual application. In addition, when the application is dynamically expanded, it takes a certain time to deploy the application copy to download and initialize the image, which undoubtedly brings time burden to the expansion. In edge environments, especially for delay-sensitive applications, responsive dynamic scaling is never feasible. Therefore, a new strategy needs to be designed, and the replica capacity should be expanded in advance before the request traffic increases to avoid serious consequences caused by the lag of the reactive strategy.

The dynamic scaling strategy of delay-sensitive applications should follow the following two principles: The first is to make advance predictions as much as possible during the expansion to reduce the time burden of application initialization. The second is to use hysteresis shrinkage as much as possible when shrinking. It is better to waste some resources and ensure the resource requirements of delay-sensitive applications. Active dynamic scaling of delay-sensitive applications is implemented based on prediction algorithms, and the number of copies is dynamically adjusted based on the prediction results.

The prediction algorithm can estimate the amount of application resource usage at the next moment in advance by modeling historical data information. Although the application can be expanded in advance, it will also be reduced in advance, affecting the quality of service of the application. Therefore, this paper proposes a combination model based on gray model and weighted movement model to solve the problems of capacity expansion and capacity reduction respectively. The two models are introduced below.

(1) Gray prediction model

The gray prediction model is a classic prediction model and can obtain more accurate prediction results even when the amount of known data is small. The gray model has been widely used in many technical fields such as architecture, energy, computers, etc. In the gray model, $GM(1,1)$ is the most common one. The general steps of the model are as follows:

Step 1: The n sampling values at time g are periodically taken out, and they are respectively recorded as $x^{(0)}(1), x^{(0)}(2), \ldots, x^{(0)}(n)$ and the following formula:

$$x^{(0)}(1) = \left[x^{(0)}(1), x^{(0)}(2), \ldots, x^{(0)}(n)\right] \tag{13}$$

Step 2: The step ratio of the sampling sequence is calculated by formula (14). If all the step ratios of the series fall within the interval $\left(e^{\frac{-2}{n+1}}, e^{\frac{2}{n+1}}\right)$, the gray model can be used for modeling. Otherwise, the sampling sequence needs to be translated by formula (15).

$$\lambda(k) = \frac{x^{(0)}(k-1)}{x^{(0)}(k)} \tag{14}$$

$$x^{(0)}(k) = x^{(0)}(k) + c, k = 1, 2 \ldots, n \tag{15}$$

Step 3: We assume that $x^{(1)}(i)$ is an accumulation sum of $x^{(0)}(i)$, and use an accumulation sum of $x^{(0)}(i)$ to eliminate the randomness caused by data changes, that is,

$$\begin{aligned} &x^{(1)}(1) = x^{(0)}(1) \\ &x^{(1)}(2) = x^{(0)}(1) + x^{(0)}(2) \\ &x^{(1)}(3) = x^{(0)}(1) + x^{(0)}(2) + x^{(1)}(3) \\ &\cdots \\ &x^{(1)}(n) = x^{(1)}(1) + x^{(1)}(2) + \cdots + x^{(1)}(n) \end{aligned} \tag{16}$$

Then:

$$x^{(1)} = \left[x^{(1)}(1), x^{(1)}(2), \ldots, x^{(1)}(n)\right] \tag{17}$$

Table 1
Resource occupation of delay-insensitive applications in Clusterl.

Application name	CPU quota	RAM quota
U1	400 mm	100 MB
U2	200 mm	200 MB
U3	100 mm	400 MB

Step 4: The differential equation is established using the first-order accumulation sequence and Eq. (18). We assume that g is the required parameter, where a is called the development gray number and b is called the endogenous control gray number.

$$\frac{dx^{(1)}}{dt} + ax^{(1)} = b \tag{18}$$

Step 5: We assume that $Y = \left[x^{(0)}(2) + x^{(0)}(3) + \cdots + x^{(0)}(n)\right]^T$ and T represent transposes, and the matrix of B is represented by formula (19). Then, the parameters are solved by formula (20).

$$B = \begin{bmatrix} -\frac{1}{2}\left[x^{(1)}(2) + x^{(1)}(1)\right] & 1 \\ -\frac{1}{2}\left[x^{(1)}(3) + x^{(1)}(2)\right] & 1 \\ \cdots & \cdots \\ -\frac{1}{2}\left[x^{(1)}(n) + x^{(1)}(n-1)\right] & 1 \end{bmatrix} \tag{19}$$

$$\hat{u} = \begin{bmatrix} a \\ b \end{bmatrix} = \left(B^T B\right)^{-1} B^T Y \tag{20}$$

Step 5: By solving the first order ordinary differential equation of formula (18), formula (21) can be obtained:

$$x^{(1)}(k+1) = \left[x^{(0)}(1) - \frac{b}{a}\right] \times e^{-ak} + \frac{b}{a}, k = 1, 2, \ldots, n \tag{21}$$

Step 6: The predicted value obtained by reducing the result of one accumulation is:

$$\begin{aligned} &x^{(0)}(k+1) = x^{(1)}(k+1) - x^{(1)}(k) = (1 - e^{a})\left[x^{(0)}(1) - \tfrac{b}{a}\right] e^{-ak} \\ &k = 1, 2, \ldots, n \end{aligned} \tag{22}$$

At this time, $x^{(0)}(k+1)$ is the predicted value at time $k+1$ obtained by the gray prediction algorithm.

(2) Weighted moving average prediction model

The weighted moving average forecasting model is a commonly used forecasting model that reflects recent trends in data. The model predicts the recent weighted average of the data by assigning different weights to the recent data and assigning higher weight to the more recent data based on the time node of the data. Compared with the gray prediction model, the algorithm of the weighted moving average prediction model is simpler. For the historical data sequence in Eq. (13), the weighted moving average prediction model directly calculates the predicted value at the next moment through Eq. (23). It can be seen that the weighted moving average model can effectively reflect the recent average of the time series.

$$\begin{aligned} &x^{(0)}(k+1) = \beta_1 x^{(0)}(1) + \beta_2 x^{(0)}(2) + \cdots + \beta_k x^{(0)}(k), \\ &\beta_1 + \beta_2 + \cdots + \beta_k = 1 \end{aligned} \tag{23}$$

In response to the needs of delay-sensitive applications, this paper proposes an Gray and Moving for Horizontal Pod Autoscaling strategy, which is dominated by the gray model and supplemented by the moving weighted average model, referred to as GMHPA.

The GMHPA strategy establishes a load queue for the application as historical data. In each monitoring cycle, a gray model is used to predict the load usage of the application in the next cycle. The prediction results are compared with the collection results of this cycle. When the predicted amount is greater than the current collected amount, the load is considered to have an upward trend, and the predicted amount is used as the input of the HPA strategy. When the predicted amount is smaller than the current collected amount for several times in a row,

the load is considered to have a downward trend, and the predicted amount is used as the input of the HPA strategy. Otherwise, a moving weighted average prediction is performed on the historical data, and this result is selected as the input of the HPA strategy. Obviously, when the queue length of the moving weighted average model is 1, and when the capacity reduction is adopted, the GMHPA strategy degenerates into the same responsive capacity reduction as the HPA strategy. The following uses the strategy that the CPU is used as a scaling index as an example to introduce the specific workflow of the GMHPA strategy.

Step 1: The parameters of the delay-sensitive application A are initialized, the resource monitoring period, and the threshold of the number of shrinkage judgments are set. At the same time, the gray queue GQ and the weighted moving average queue MQ of the application A are established.

Step 2: The monitoring module collects the CPU resource usage rate of all copies of application A every time T and records it as $A(t)$, and updates $A(t)$ into the GQ and MQ queues.

Step 3: It is judged whether the GQ queue can meet the requirements of the gray model. If the requirements are met, the algorithm executes step 4, if the requirements are not met, the algorithm executes step 7.

Step 4: The gray model is called to calculate the resource usage $PA(t+1)$ of application A in the next cycle, and determine whether $PA(t+1) > A(t) \parallel n > th$ is satisfied.

Step 5: The number n of load forecast drops is reset, $PA(t+1)$ is used as the total resource usage, and the HPA strategy is called to calculate the number of copies in the next cycle of the application. Then, the algorithm jumps to step 8.

Step 6: The number of times the load forecast drops is increased by one.

Step 7: The weighted moving average model is called to predict the resource usage $MA(t+1)$ of application A in the next cycle, and uses $MA(t+1)$ as the total resource usage of application A in the next cycle, and calls the HPA strategy to calculate the number of copies. Then, the algorithm jumps to step 8.

Step 8: According to the number of target copies, it is judged whether to trigger dynamic scaling.

4. Delay-insensitive application scheduling strategy

Delay-insensitive applications have lower priority in the cluster than delay-sensitive applications. When the cluster has limited resources, it will release the resources occupied by delay-insensitive applications, prioritize the resource requirements of delay-sensitive applications, and schedule delay-insensitive applications into other clusters. Delay-insensitive application scheduling is divided into three sub-steps: trigger scheduling, application selection, and cluster selection. This process is called Delay Insensitive Cross-Cluster Scheduling, DICC.

Delay-sensitive applications are generally requesting near the edge clusters, while delay-insensitive applications are requests from multiple edge clusters. If the edge cluster is now busy, the algorithm needs to move delay-insensitive applications out of the cluster. If the edge cluster is not busy, the algorithm can move delay-insensitive applications into the cluster.

Then, the scheduling of delay-insensitive applications must first solve the problem of how to determine whether the cluster is busy and classify the cluster. Since the application in the cluster involves moving in and out, this paper proposes a cluster classification method based on double threshold.

First, two thresholds γ_h and γ_l are specified, $0 < \gamma_l < \gamma_h < 1$. The total resource of the group can be calculated by formula (24):

$$C_{F_i}^D = \sum_{j\in J} C_{F_{i,j}}^D = \left(\sum_{j\in J} C_{f_{i,j}}^{cpu}, \sum_{j\in J} C_{f_{i,j}}^{ram}\right) \tag{24}$$

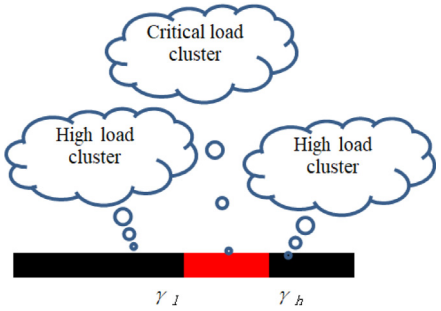

Fig. 3. Edge cluster classification.

The used resource usage in the cluster is:

$$\begin{aligned} R_{F_i}^D &= \sum_{n\in N} R_{A_n^s}^D + \sum_{m\in M} R_{A_m^u}^D + R_{A_{F_i}^p}^D \\ &= \sum_{n\in N}\sum_{k\in K} R_{c_{n,k}}^D + \sum_{m\in M} \rho_{A_m^u}^{F_i} R_{A_m^u}^D + R_{A_{F_i}^p}^D \end{aligned} \tag{25}$$

That is,

$$R_{F_i}^{cpu} = \sum_{n\in N}\sum_{k\in K} R_{c_{n,k}}^{cpu} + \sum_{m\in M} \rho_{A_m^u}^{F_i} R_{A_m^u}^{cpu} + R_{A_{F_i}^p}^{cpu} \tag{26}$$

$$R_{F_i}^{ram} = \sum_{n\in N}\sum_{k\in K} R_{c_{n,k}}^{ram} + \sum_{m\in M} \rho_{A_m^u}^{F_i} R_{A_m^u}^{ram} + R_{A_{F_i}^p}^{ram} \tag{27}$$

The resource utilization rate of cluster i can be calculated by using the total resource amount and the used resource amount:

$$\theta_{F_i}^D = \frac{U_{F_i}^D}{C_{F_i}^D} = \frac{C_{F_i}^D - R_{F_i}^D}{C_{F_i}^D}, 0 < \theta_{F_i}^D < 1 \tag{28}$$

That is,

$$\theta_{F_i}^{cpu} = \frac{U_{F_i}^{cpu}}{C_{F_i}^{Dcpu}} = \frac{C_{F_i}^{cpu} - R_{F_i}^{cpu}}{C_{F_i}^{cpu}}, 0 < \theta_{F_i}^{cpu} < 1 \tag{29}$$

$$\theta_{F_i}^{ram} = \frac{U_{F_i}^{ram}}{C_{F_i}^{ram}} = \frac{C_{F_i}^{ram} - R_{F_i}^{ram}}{C_{F_i}^{ram}}, 0 < \theta_{F_i}^{ram} < 1 \tag{30}$$

As shown in Fig. 3, clusters are divided into three categories based on the relationship between resource usage and thresholds: low-load clusters, critical-load clusters, and high-load clusters. The low-load cluster satisfies $\left(\theta_{F_i}^{cpu} < \gamma_l\right) \cap \left(\theta_{F_i}^{ram} < \gamma_l\right)$. That is, the load of the CPU and RAM that can be moved into the cluster is not high, and the delay-insensitive applications can be accepted. High load clusters satisfy $\left(\theta_{F_i}^{cpu} < \gamma_h\right) \cup \left(\theta_{F_i}^{ram} < \gamma_h\right)$. As long as the computing resources of any dimension in the cluster exceed the threshold range, the cluster is considered to be a high-load cluster, and a scheduling mechanism is triggered to recover the resources occupied by individual delay-insensitive applications in the cluster and transfer them to the low-load cluster.

The remaining clusters are critical load clusters, which means that the resource usage of any dimension in the cluster is between two thresholds, which may be CPU resource usage, RAM resource usage, or both. Although this type of cluster will not migrate applications from the cluster, it will no longer accept high-load cluster migration application requests.

In summary, the clusters connected to the collaboration layer are classified, and the monitoring module is used to sense the resource load status of each cluster. When any cluster is detected to be under high load, delay-insensitive application scheduling is triggered.

The basic principle of the scheduling mechanism is that one trigger corresponds to only one application. That is, when a high load is detected in each monitoring cycle, only one of the delay-insensitive applications is selected for scheduling. After that, through periodic and

continuous monitoring, it is finally guaranteed that the cluster is no longer in a high load state.

When the collaboration layer monitors that a cluster is under high load, it can be divided into single-resource overload and dual-resource overload according to the types of resources that trigger scheduling. According to the type of triggering scheduling, different strategies are adopted to select the applications to be scheduled.

For the single resource overload type, the scheduling module directly selects the delay-insensitive application that has the most resources of the type in the current cluster as the scheduling object. For the dual resource overload type, it means that both the CPU and RAM of the cluster exceed the threshold. The scheduling module first selects the application with the highest CPU resource occupation and RAM resource occupation in the cluster. If the two applications are the same application, the application is selected. If the two applications are not the same application, the scheduling priority scores for the two applications are respectively adopted by formula (31), and the higher W score is selected as the application to be scheduled.

$$W = w_1 \times \frac{R_{A^w}^{cpu}}{C_{F_i}^{cpu}} + w_2 \times \frac{R_{A^w}^{ram}}{C_{F_i}^{ram}} \tag{31}$$

In the formula, $w_1+w_2 = 1$, respectively represent the weights of the CPU and ram of the cluster, and A^w is the delay-insensitive application to be selected.

After the scheduling module selects the application to be scheduled, it starts to select the appropriate cluster for the application and deploys the application to the cluster. From the correspondence between delay-insensitive applications and cluster nodes in the formula (9), we can see that each application is either deployed in the central cloud or in a node in the edge cluster. Only when all clusters are unable to accept application migration, delay-insensitive applications will be deployed to the central cloud.

When there is a low-load cluster, the scheduling module first adds all the low-load clusters to the schedulable set, denoted as Ω. In order to avoid cluster state turbulence after applying scheduling, the scheduling module first pre-selects the cluster. After that, the scheduling module traverses Ω and calculates the resource utilization rate after the application is scheduled to the cluster by using (32) for each cluster of Ω. According to the criteria for cluster classification, the cluster of $\theta_{F_i}^{D^t} > \gamma_h$ is removed from Ω.

$$\theta_{F_i}^{D^t} = \theta_{F_i}^{D} + \frac{R_{A^w}^{D}}{C_{F_i}^{D}} \tag{32}$$

We assume that $\widetilde{R}_{F_i}^{D}$ represents the computing resources available in the cluster:

$$\widetilde{R}_{F_i}^{D} = \left(\gamma_h - \theta_{F_i}^{D}\right) \times R_{F_i}^{D} \tag{33}$$

That is.

$$\widetilde{R}_{F_i}^{cpu} = \left(\gamma_h - \theta_{F_i}^{cpu}\right) \times R_{F_i}^{cpu} \tag{34}$$

$$\widetilde{R}_{F_i}^{ram} = \left(\gamma_h - \theta_{F_i}^{ram}\right) \times R_{F_i}^{ram} \tag{35}$$

According to the preselected result Ω, the computing resources provided by the cluster and the usage rates of the two types of computing resources are respectively clustered by formula (36) for scoring.

$$S(j) = \alpha_1 \times \frac{\widetilde{R}_{F_i}^{cpu}}{\sum_{F_i \in \Omega} \widetilde{R}_{F_i}^{cpu}} + \alpha_2 \times \frac{\widetilde{R}_{F_i}^{ram}}{\sum_{F_i \in \Omega} \widetilde{R}_{F_i}^{ram}} + \alpha_3 \cdot \frac{1}{\left|\theta_{F_j}^{cpu^t} - \theta_{F_j}^{ram^t}\right|} \tag{36}$$

Where $\alpha_1 + \alpha_2 + \alpha_3 = 1$ represents their respective weights. Among them, the first two terms of formula (36) are used to ensure the load balance of each cluster, that is, the more resources a cluster can provide, the more likely it is to be selected. The last item is used to ensure that the selected cluster is migrated to the application as much as possible, and the two resource utilization rates remain balanced.

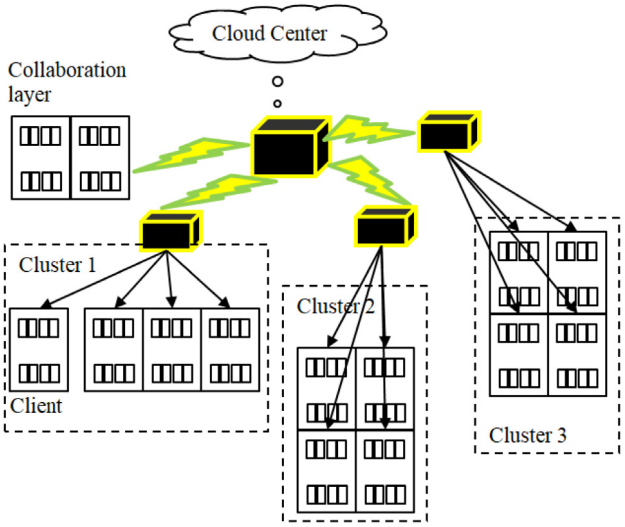

Fig. 4. Network topology of the test environment.

5. Algorithm simulation tests

System testing is mainly divided into the effectiveness and performance analysis of the scheduling strategy of delay-sensitive applications and the effectiveness and performance analysis of the scheduling strategy of delay-insensitive applications. The system test hardware environment includes a PC, a DELLPowerEdge 820 server, two DELLPowerEdge R730 servers, and a Tencent cloud server. On the server, a KVM virtual machine is created and a Kubernetes cluster is set up using the virtual machine. The virtual machine uses a bridge to connect to the LAN. The master node in the cluster is responsible for managing the edge cluster, and provides APISerVer and kubeproxy services, and does not participate in the workload distribution. In addition, a Windows virtual machine is created in the server to which the Clusterl cluster belongs as a client to send requests. The operating system of the client virtual machine is Winl0, and a 4-core CPU and 8G memory are allocated. The test environment network topology is shown in Fig. 4.

For the delay-sensitive application resource scheduling test, the experiment uses pressure access to the real system, and observes the change in the number of copies of the application under the GMHPA and HPA policies and the comparison of the application response time. Because the dynamic scaling of delay-sensitive applications is based on single-factor triggering, the design chose CPU resources that are convenient for changes to conduct experiments. Among them, the CPU resource quota unit is m, which represents 1/1000 of a CPU core.

The experiment first prepared a computationally intensive Dokcer containerization application. Moreover, each time the request is made, the app performs a pi calculation. Then, the application was deployed in Cluster with GMHPA strategy and HPA strategy for comparative experiments. Through multi-threaded concurrent access, the steady, steady rise, rapid rise, rapid decline, and steady decline of the request volume are simulated, and the actual load of the application, the predicted load, the number of copies, and the response delay change are observed.

The actual load and forecast results are shown in Fig. 5. At the beginning, the application is in a low-load and stable state. The load starts to increase slowly in the 7–15 min. After 5 min of stabilization, the load starts to accelerate and rise in 20–23 min. After 5 min of stabilization, the load begins to accelerate in the 28–30 min. After 5 min of stabilization, it slowly decreased from the 35th minute. It can be seen that during the entire load change process, the GMHPA strategy can roughly fit the actual load curve, but at the 28th minute when the load increases from a rapid to a stable corner, the GMHPA strategy's

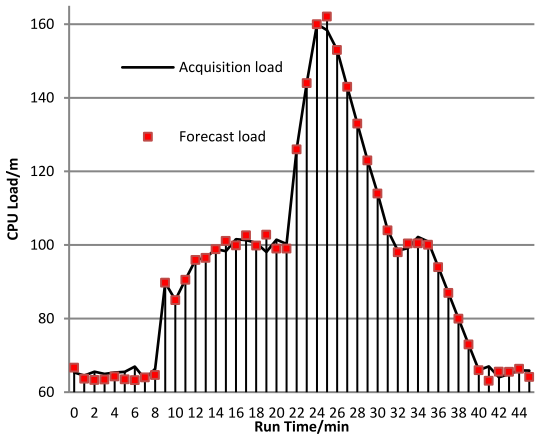

Fig. 5. Collection load and predicted load.

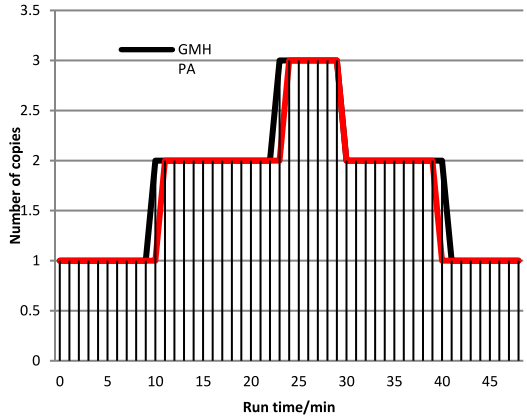

Fig. 6. The situation of replica changes.

prediction algorithm has some errors. However, for the case where the predicted value is larger than the actual value, the delay-sensitive application is a saturated resource supply. Even if the application expansion is triggered, the response delay of the application will not be affected. At the same time, due to the setting of the number of shrinkage judgments and the introduction of a moving weighted average model, the predicted value did not decrease significantly during the period when the load rapidly dropped to a steady change in the 31st minute. Moreover, during the entire load drop period, the predicted value of GMHPA is slightly larger than the actual value, which just meets the delay recovery requirements of delay-sensitive applications.

Fig. 6 shows the changes in the replicas applied under the GMHPA policy. When the application was initially initialized, there was only one copy instance. As the load continues to increase, the application triggers the first expansion in the 9th minute and the application triggers the second expansion in the 22nd minute. During the two capacity expansions, the GMHPA mechanism can effectively predict the load increase. Compared with the existing HPA mechanism, the capacity can be expanded one cycle in advance. In the load reduction phase, because the GMHPA mechanism uses a saturated resource supply when the capacity is reduced, the gray model result is selected when the gray predicted load is lower than the current load, otherwise the moving weighted average model is selected. Therefore, in the 39th minute, the GMHPA mechanism is delayed for a period of time compared with the HPA mechanism to reduce the size, and the delayed resource recovery task is better completed.

At the same time, experiments recorded response times for both strategies. Because the response time is greatly affected by network

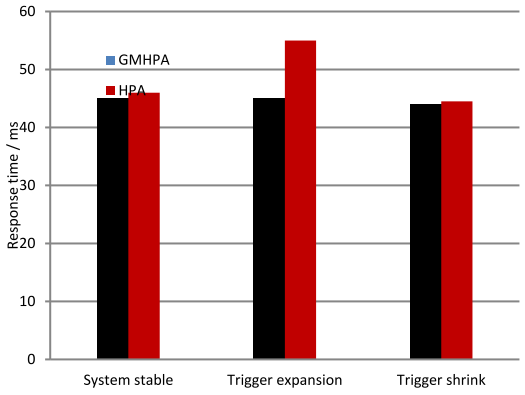

Fig. 7. Comparison of response time between GMHPA strategy and HPA strategy.

Table 2
Resource quotas for delay-sensitive application replicas in different groups.

Group	Application type	CPU quota	RAM quota
A	Sensitive	60 m	100 M
B	Sensitive	50 m	120 M
C	Sensitive	60 m	120 M

fluctuations, for a more intuitive comparison of test results, the test is divided into three phases: the system is stable, the capacity expansion is triggered, and the capacity reduction is triggered. Among them, the triggering expansion/reduction capacity includes the triggering time and its adjacent time, and the rest of the time is the system stability time. As shown in Fig. 7, the response times of the two strategies are basically the same when the system is stable and triggering the capacity reduction. When the capacity expansion is triggered, the average response time of the HPA strategy is increased by about 11 ms compared to the GMHPA strategy. The main reason is that the HPA strategy is based on the responsive nature, and the capacity is expanded when the load is sensed, and the application takes a certain amount of time to initialize during the expansion, which results in an untimely supply of resources required by the application and an increase in average response time. However, the GMHPA strategy can effectively use the prediction algorithm to achieve the capacity expansion goal in advance, and better meet the resource requirements of delay-sensitive applications.

By increasing the delay-sensitive application load of a single cluster in a multi-cluster environment, the experimental test of delay-insensitive application resource scheduling is used to observe the scheduling process of the DICCS strategy and analyze the load change of the cluster before and after scheduling. The experiment first deployed three delay-insensitive applications in Clusterl. The fixed CPU was allocated through the resources.1imits.cpu and -cpus parameters in the Kubernetes application configuration file, and the fixed RAM was allocated through the resources.1imitS.meory and -mem.total parameters. The three application CPU and RAM resource quotas are shown in Table 1.

Then, the same method is used to deploy applications in the cluster to initialize the cluster load so that they all meet the low-load cluster. Then, a delay-sensitive application is deployed in Clusterl, and the system monitoring module is used to detect the usage of CluSter resources, and a new copy is generated for the application every monitoring cycle until ClusterSterl becomes a high-load cluster. During the test, three sets of control experiments were set up to simulate different resource requirements for delay-sensitive applications. The delay-sensitive application resource quotas used in the three groups of experiments are shown in Table 2.

The parameters used in the DICCS strategy during the test are shown in Table 3.

Table 3
Parameter values of the DlCCS strategy.

Parameter	Value
Monitoring period T	1 min
Critical threshold γl	0.6
High load threshold γh	0.7
Apply weights $\omega 1$, $\omega 2$	0.5, 0.5
Cluster weights $\alpha 1$, $\alpha 2$, $\alpha 3$	0.4, 0.4, 0.2

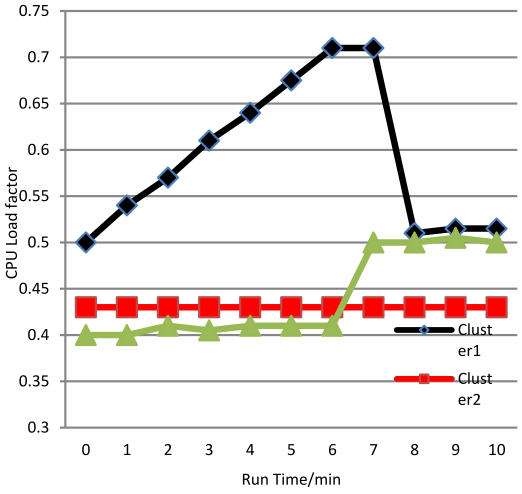

Fig. 8. Change graph of the CPU load rate of the experimental cluster in group A.

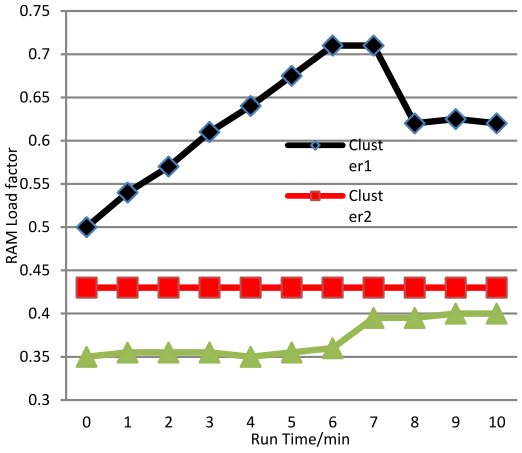

Fig. 9. Change graph of RAM load rate of the experimental cluster in group A.

The experimental test results of group A are shown in Figs. 8 and 9. From the first minute, Clusterl continuously increases the amount of delay-sensitive application copies. In the sixth minute, the CPU resource load in Clusterl exceeds the high load threshold of nine, triggering scheduling. According to the principle of single resource factor trigger, U1, the time-insensitive application with the highest CPU usage, is selected as the application to be scheduled, and Cluster2 and Cluster3 are scored to obtain a Cluster2 score of 2.39 and a Cluster3 score of 2.41. Therefore, C1uster3 is selected as the destination cluster, and scheduling instructions are sent to it to try to deploy U1 on Cluster3. In the 7th minute, the monitoring module found that the application was successfully deployed in Cluster3, and U1 resources in Clusterl were recovered. In the 8th minute, it was observed that U1 application resources in the Clusterl cluster were recovered, and Clusterl became a low-load cluster.

The experimental results of group B are shown in Figs. 10 and 11. The results in the first 5 min are the same as those in the group

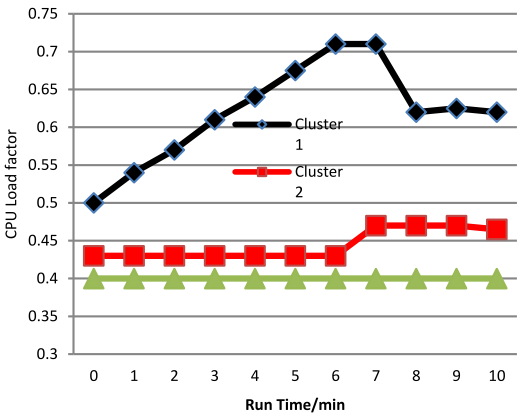

Fig. 10. Change graph of CPU load rate of the experimental cluster in group B.

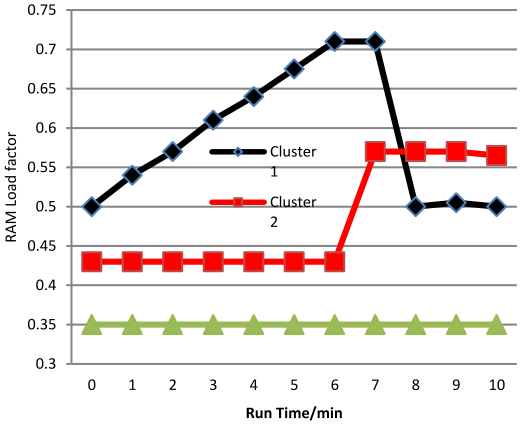

Fig. 11. Change graph of CPU load rate of the experimental cluster in group B.

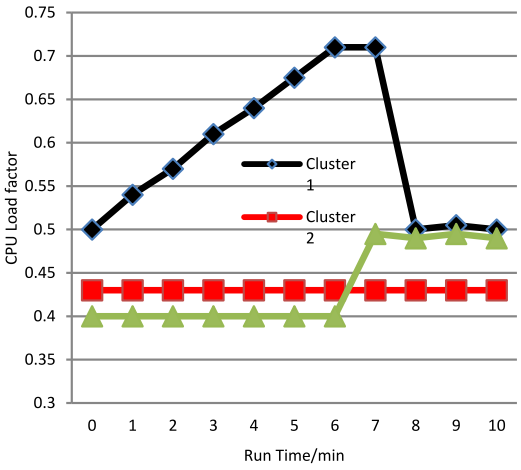

Fig. 12. Change graph of CPU load rate of the experimental cluster in group C.

A experiment. In the 6th minute, the load rate of RAM resources in Clusterl exceeds the high load threshold y, which triggers scheduling. Through DICCS strategy calculation, U3 is selected to be dispatched to Cluster2, and the adjusted Clusterl cluster becomes a low-load cluster again.

The experimental results of group C are shown in Figs. 12 and 13. The results in the first 5 min are similar to those in Groups A and B. In the 6th minute, the CPU and RAM load rates in Clusterl exceed the high load threshold y, triggering scheduling. According to the dual

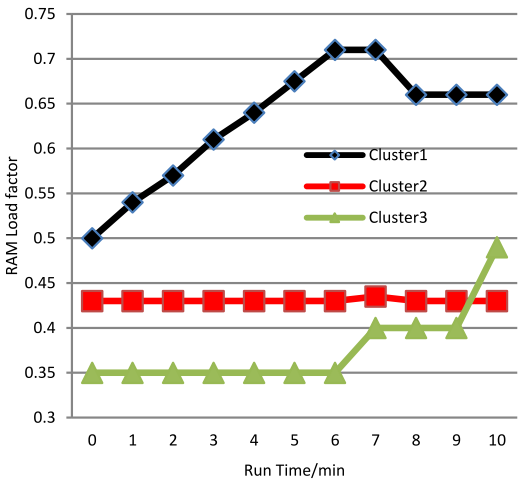

Fig. 13. Change graph of the RAM load rate of the experimental cluster in group C.

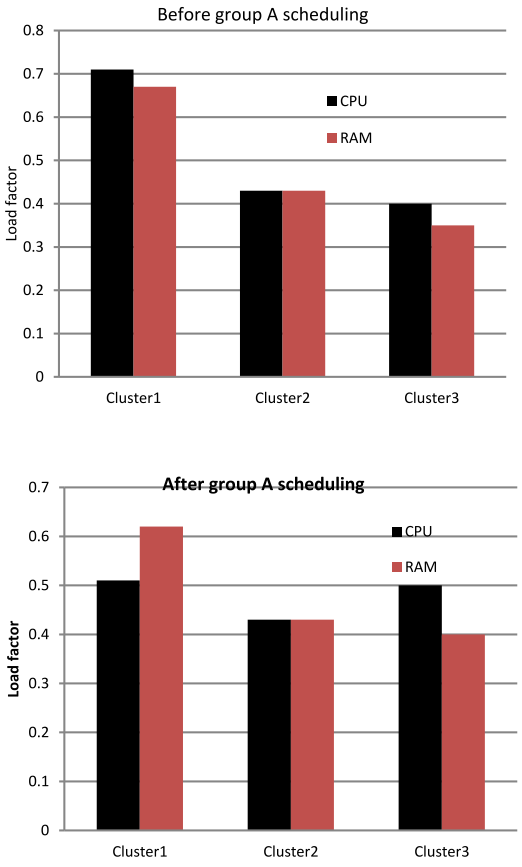

Fig. 14. Comparison of resource load ratio before and after experimental scheduling in group A.

resource factor trigger principle, the applications with the highest CPU and RAM occupancy U1 and U2 are selected and scored. U1 score is 0.15 and U2 score is 0.037. Therefore, U1 application is selected as the application to be scheduled. Then, by scoring Cluster2 and Cluster3, it was found that the scores of the two clusters were the same as those of the group A experiment. The reason is that scoring the cluster depends only on the status of the destination cluster and the resource quota of the application to be scheduled, and the status of the application to be scheduled and the destination cluster selected by the two sets of experiments are the same. Therefore, the scoring results are the same.

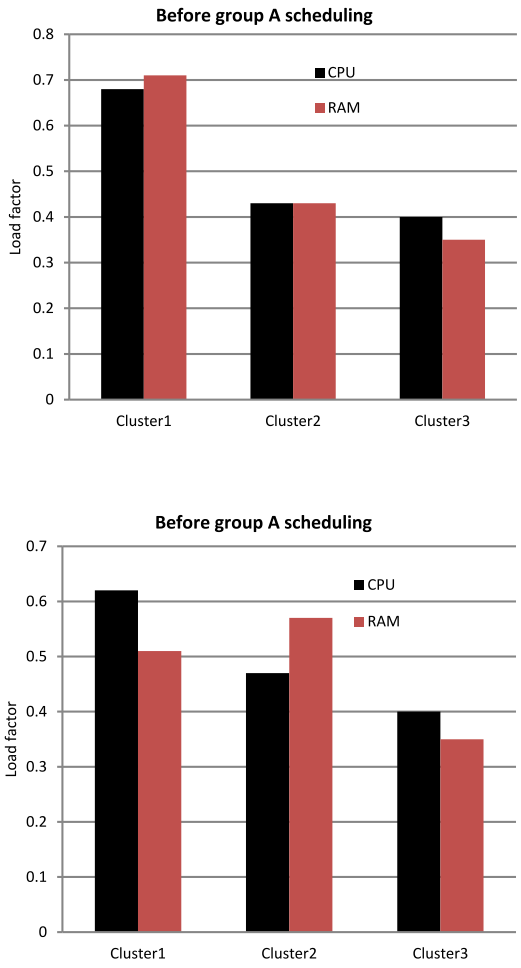

Fig. 15. Comparison of resource load ratio before and after experimental scheduling in group B.

It can be known from three sets of experiments that the proposed DICCS strategy can effectively select the corresponding application to be dispatched to the destination cluster to implement the cross-cluster scheduling function of the time-insensitive application according to the triggering factors of application scheduling.

During the experiment, the load utilization rate of each cluster before and after application scheduling is statistically shown in Figs. 14, 15, and 16. By quantifying the variance of the cluster load utilization, it was found that the variance of the CPU and RAM load rates of Group A was reduced from 0.029 and 0.0285 to 0.0017 and 0.0147, respectively. Moreover, the variances of the CPU and RAM load rates of Group B were reduced from 0.00225 and O.0355 to 0.0134 and 0.0126. In addition, the variances of CPu and RAM load rates in group c were reduced from 0.0029 and 0.0005 to 0.0017 and 0.02, respectively. After each cluster is scheduled, the load is more balanced than before the scheduling. However, because the D1CCS strategy only selects the application with the highest resource occupation rate for scheduling, some applications with high two types of resource occupation but not the highest can only wait until the next monitoring cycle for scheduling. However, in general, the DICCS strategy proposed in this paper can effectively solve the constraints of edge cluster resources, schedule delay-insensitive applications into other low-load clusters and achieve load balancing between clusters.

6. Conclusion

In order to improve the effective operation of the piano teaching system of the Internet of Things, this study analyzes the resource

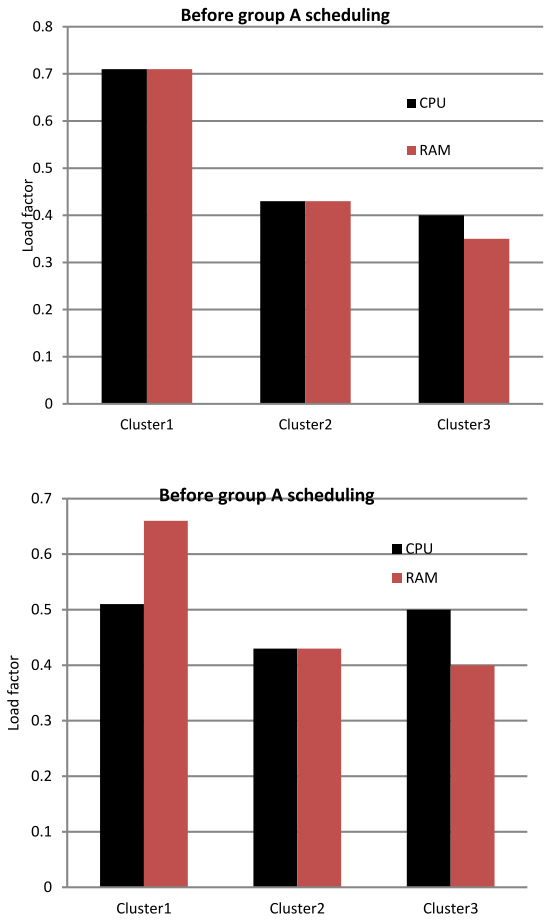

Fig. 16. Comparison of resource load ratio before and after experimental scheduling in group C.

scheduling of delay-sensitive applications and proposes a multi-cluster edge cloud framework based on application-sensitive delay differences. The framework distinguishes application categories based on domain names and performs multi-cluster management through a master–slave model. On this basis, this paper studies the framework's deployment architecture, workflow, resource scheduling strategy and related functional modules in detail, and proves its rationality and feasibility of implementation through tests in a real physical environment.

In order to meet the resource requirements of delay-sensitive applications in the framework and guarantee the quality of service of delay-sensitive applications, this paper proposes an active scaling strategy GMHPA. By combining the gray prediction model and the weighted moving average model, the GMHPA strategy can realize the function of delay-sensitive applications expanding capacity in advance when the request load increases and decreasing capacity in following when the request load decreases. In order to improve the overall cluster resource utilization rate in the framework, for delay-insensitive applications, this paper proposes a cross-cluster scheduling strategy, DICCS. The DICCS strategy can perform delay-insensitive application scheduling during system operation, achieve cooperative scheduling goals of multiple clusters, and make the load between clusters more balanced.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Funded projects

2013 Hunan Philosophy and Social Science Foundation, China Project Name of Project "Ecological Research on Xiangxi Flower Lantern Play in the Perspective of Intangible Culture" Project Number: 13YBB173.

References

[1] J. Pan, J. Mcelhannon, Future edge cloud and edge computing for Internet of Things applications, IEEE Internet Things J. PP (99) (2017) 1.
[2] L. Baresi, Danilo Filgueira Mendonça, M. Garriga, Empowering Low-Latency Applications Through a Serverless Edge Computing Architecture, 2017.
[3] Z. Li, W.M. Wang, G. Liu, et al., Toward open manufacturing: A cross-enterprises knowledge and services exchange framework based on blockchain and edge computing, Ind. Manage. Data Syst. 118 (9) (2018) 303–320.
[4] A. Aral, T. Ovatman, A decentralized replica placement algorithm for edge computing, IEEE Trans. Netw. Serv. Manag. (2018).
[5] S. Shahzadi, M. Iqbal, T. Dagiuklas, et al., Multi-access edge computing: Open issues, challenges and future perspective, J. Cloud Comput. 6 (1) (2017) 30.
[6] D. Xiaoheng, G. Peiyuan, W. Zhiwen, et al., Integrated trust based resource cooperation in edge computing, J. Comput. Res. Dev. (2018).
[7] J. Yang, J. Wen, B. Jiang, et al., Marine depth mapping algorithm based on the edge computing in Internet of things, J. Parallel Distrib. Comput. 114 (2018).
[8] Y. Peng, X. Wang, D. Shen, et al., Design and modeling of survivable network planning for software-defined data center networks in smart city, Int. J. Commun. Syst. 31 (16) (2018).
[9] N. Mohan, J. Kangasharju, Edge-Fog Cloud: A Distributed Cloud for Internet of Things Computations, 2017.
[10] D. Park, S. Kim, Y. An, et al., LiReD: A light-weight real-time fault detection system for edge computing using LSTM recurrent neural networks, Sensors 18 (7) (2018) 2110.
[11] Q. Fan, N. Ansari, Application aware workload allocation for edge computing based IoT, IEEE Internet Things J. 5 (3) (2018) 2146–2153.
[12] N. Abbas, Y. Zhang, A. Taherkordi, et al., Mobile edge computing: A survey, IEEE Internet Things J. (99) (2017) 1.
[13] K. Zhang, S. Leng, Y. He, et al., Mobile edge computing and networking for Green and low-latency Internet of Things, IEEE Commun. Mag. 56 (5) (2018) 39–45.
[14] J. Ren, Y. Guo, D. Zhang, et al., Distributed and efficient object detection in edge computing: Challenges and solutions, IEEE Netw. PP (99) (2018) 1–7.
[15] X. Gong, L. Guo, G. Shen, et al., Virtual network embedding for collaborative edge computing in optical-wireless networks, J. Lightwave Technol. PP (99) (2017) 1.
[16] P. Bellavista, S. Chessa, L. Foschini, et al., Human-enabled edge computing: Exploiting the crowd as a dynamic extension of mobile edge computing, IEEE Commun. Mag. 56 (1) (2018) 145–155.
[17] Y. Mao, J. Zhang, S.H. Song, et al., Stochastic joint radio and computational resource management for multi-user mobile-edge computing systems, IEEE Trans. Wireless Commun. 16 (9) (2017) 5994–6009.
[18] K. Zhang, Y. Mao, S. Leng, et al., Mobile-edge computing for vehicular networks: A promising network paradigm with predictive off-loading, IEEE Veh. Technol. Mag. 12 (2) (2017) 36–44.
[19] X. Yang, Z. Chen, K. Li, et al., Communication-constrained mobile edge computing systems for wireless virtual reality: Scheduling and tradeoff, IEEE Access 6 (2018) 16665–16677.
[20] X. Huang, R. Yu, J. Kang, et al., Distributed reputation management for secure and efficient vehicular edge computing and networks, IEEE Access PP (99) (2017) 1.
[21] X. Xu, J. Liu, X. Tao, Mobile edge computing enhanced adaptive bitrate video delivery with joint cache and radio resource allocation, IEEE Access PP (99) (2017) 1.
[22] R. Morabito, V. Cozzolino, A.Y. Ding, et al., Consolidate IoT edge computing with lightweight virtualization, IEEE Netw. XX (X) (2018) 102–111.
[23] Y. Duan, X. Sun, H. Che, et al., Modeling data, information and knowledge for security protection of hybrid IoT and edge resources, IEEE Access 7 (2019) 1.
[24] Z. Liu, K.K.R. Choo, J. Grossschadl, Securing edge devices in the post-quantum Internet of Things using lattice-based cryptography, IEEE Commun. Mag. 56 (2) (2018) 158–162.
[25] J. Yang, Z. Lu, J. Wu, Smart-toy-edge-computing-oriented data exchange based on blockchain, J. Syst. Archit. 87 (2018).
[26] Raül Muñoz, S. Member, IEEE, et al., Integration of IoT, transport SDN, and edge/cloud computing for dynamic distribution of IoT analytics and efficient use of network resources, J. Lightwave Technol. 36 (7) (2018) 1420–1428.
[27] J. Yuan, X. Li, A reliable and lightweight trust computing mechanism for IoT edge devices based on multi-source feedback information fusion, IEEE Access 6 (2018) 23626–23638.

[28] A.T. Al-Hammouri, Z. Al-Ali, B. Al-Duwairi, ReCAP: A distributed CAPTCHA service at the edge of the network to handle server overload, Trans. Emerg. Telecommun. Technol. 29 (2) (2017) e3187.
[29] M.S. Elbamby, C. Perfecto, C.F. Liu, et al., Wireless edge computing with latency and reliability guarantees, Proc. IEEE 107 (8) (2019).
[30] Z. Qi, H. Yupeng, J. Cun, et al., Edge computing application:real-time anomaly detection algorithm for sensing data, J. Comput. Res. Dev. (2018).
[31] N.N. Dao, D.N. Vu, W. Na, et al., SGCO: Stabilized Green crosshaul orchestration for dense IoT offloading services, IEEE J. Sel. Areas Commun. PP (99) (2018) 1.
[32] Y.D. Lin, Editorial: Fourth quarter 2017 IEEE communications surveys & tutorials, IEEE Commun. Surv. Tutor. 19 (4) (2017) 2018–2025.
[33] W.C. Chang, P.C. Wang, Adaptive replication for mobile edge computing, IEEE J. Sel. Areas Commun. PP (99) (2018) 1.
[34] D. Wang, Y. Peng, X. Ma, et al., Adaptive wireless video streaming based on edge computing: Opportunities and approaches, IEEE Trans. Serv. Comput. PP (99) (2018) 1.