Chinese Journal of Electronics
Vol.32, No.2, Mar. 2023

LBA-EC: Load Balancing Algorithm Based on Weighted Bipartite Graph for Edge Computing

SHAO Sisi[1,2], LIU Shangdong[1,2,3,4], LI Kui[1,2], YOU Shuai[1,2], QIU Huajie[1,2], YAO Xiaoliang[1,2], and JI Yimu[1,2,3,4]

(1. *School of Computer Science, Nanjing University of Posts and Telecommunications, Nanjing 210023, China*)

(2. *Institue of High Performance Computing and Bigdata, Nanjing University of Posts and Telecommunications, Nanjing 210003, China*)

(3. *Nanjing Center of HPC China, Nanjing 210003, China*)

(4. *Jiangsu HPC and Intelligent Processing Engineer Research Center, Nanjing 210003, China*)

Abstract — Compared with cloud computing environment, edge computing has many choices of service providers due to different deployment environments. The flexibility of edge computing makes the environment more complex. The current edge computing architecture has the problems of scattered computing resources and limited resources of single computing node. When the edge node carries too many task requests, the makespan of the task will be delayed. We propose a load balancing algorithm based on weighted bipartite graph for edge computing (LBA-EC), which makes full use of network edge resources, reduces user delay, and improves user service experience. The algorithm is divided into two phases for task scheduling. In the first phase, the tasks are matched to different edge servers. In the second phase, the tasks are optimally allocated to different containers in the edge server to execute according to the two indicators of energy consumption and completion time. The simulations and experimental results show that our algorithm can effectively map all tasks to available resources with a shorter completion time.

Key words — Edge computing, Weighted bipartite graph, Load balancing.

I. Introduction

With the continuous development of the Internet of things, the number of terminal devices such as smart phones and smart glasses continues to increase. The growth rate of data is far faster than that of network bandwidth. Although the integration of intelligent Internet of things and cloud computing supports many applications, it is inefficient to move all tasks to cloud computing in some cases [1]. For example, when the bandwidth is limited and the response time is strict, uploading data to the cloud for processing may lead to a longer response time and occupy a large portion of bandwidth. In this case, edge computing [2], [3] came into being. Edge computing allows data processing at the edge of the network and computing near the data source to avoid unnecessary data movement [4]. Compared with the traditional cloud computing mode, edge computing can better support mobile computing and Internet of things applications.

Edge computing not only has computing and storage capacity but also has the advantages of high bandwidth and low latency [5], which can significantly improve the real-time experience and satisfaction of users. However, the resources of edge computing nodes are scattered, and the computing performance is limited. Facing the applications that require high computing resources (i.e., multimedia processing, social network and natural language processing), edge computing is still limited by resources [6]. When the edge server carries too many task requests, the response time of some tasks will be delayed, even longer than that of calling tasks to

Manuscript Received Feb. 3, 2021; Accepted Dec. 4, 2021. This work was supported by the National Key R&D Program of China (2020YFB2104000, 2020YFB2104002), Natural Science Foundation of the Jiangsu Province (Higher Education Institutions) (BK20170900, 19KJB520046, 20KJA520001), Innovative and Entrepreneurial Talents Projects of Jiangsu Province, Jiangsu Planned Projects for Postdoctoral Research Funds (2019K024), Six Talent Peak Projects in Jiangsu Province (JY02), Postgraduate Research and Practice Innovation Program of Jiangsu Province (KYCX19 0921, KYCX19 0906), Open Research Project of Zhejiang Lab (2021KF0AB05), and NUPT DingShan Scholar Project and NUPTSF (NY219132).

 DOI:10.23919/cje.2021.00.289

the remote cloud server. It can be seen that task load balancing algorithm and computing offload strategy play an important role in edge computing, which determines the computing efficiency and quality.

As mentioned above, we design a framework of task load balancing for edge computing, and propose a task load balancing algorithm based on weighted bipartite graph (LBA-EC). The algorithm is divided into two phases: load balancing between edge nodes and load balancing between containers. First, the task is mapped to different edge nodes. Second, the tasks are optimally allocated to different containers in the edge server to execute according to the two indicators of energy consumption and completion time. The algorithm makes full use of network edge resources, reduces user delay, and improves user service experience. The main contributions of this work are as follows.

1) We design a load balancing framework for edge computing, which is composed of edge device layer, edge layer and cloud layer. When an edge device sends a task request, it can select a nearby edge data center to process the task.

2) The phase of load balancing between edge nodes is responsible for matching tasks from edge devices to different edge data centers. In this phase, combined with the queuing theory model, the task scheduling problem is modeled as a dynamic weighted bipartite graph maximum perfect matching problem. Then the task is matched to different edge data centers by finding the maximum perfect match of the current bipartite graph.

3) The phase of load balancing between containers is responsible for allocating the tasks in the edge data center to a specific container for execution. In this phase, we describe quantitatively the makespan and the energy consumption of the edge nodes. Then, the particle swarm optimization algorithm is used to find the optimal value of the objective optimization problem and decide which specific Docker container the task is allocated to.

The rest of this paper is structured as follows. Section II presents existing related works. Section III gives the framework of task load balancing for edge computing. Section IV proposes a two-stage load balancing algorithm based on bipartite graph. Section V verifies the effectiveness of the algorithm through experiments. Section VI summarizes the research work of this paper and points out our future work.

II. Related Work

Compared with the powerful cloud server, the resource of edge server is limited. When the edge server carries too many task requests, the makespan of some tasks is extended, even exceeding the response time of calling services on the remote cloud server [7]. How to effectively solve the problem of task load balancing in edge computing environment has become a research hotspot. By assigning tasks to different edge data centers for processing through task scheduling, the edge data center and cloud computing resources can be effectively utilized, the completion time of applications can be reduced, and the quality of computing experience can be improved. The fundamental purpose of task load balancing is to effectively support real-time data. It is necessary to schedule various tasks optimally to meet various SLA parameters, such as response time, power consumption, delay, and cost [8]. Many studies show that the task load balancing problem belongs to NP-hard optimization problem. Many heuristic algorithms are used to solve the multi-objective optimization problem, such as particle swarm optimization (PSO) [9], ant colony algorithm (ACO) [10], genetic algorithm (GA), simulated annealing algorithm (SA) [11], etc. For the task assignment problem in edge computing environment, researchers provide different heuristic and meta-heuristic algorithms. Among the heuristic algorithms, particle swarm optimization is the most widely used. Zhan *et al.* [12] describe the resource scheduling problem and its solutions, using several natural heuristic evolutionary methods (i.e., ant colony algorithm, genetic algorithm, particle swarm optimization). Kaur *et al.* [8] use Docker containers instead of traditional virtual machines (VMs) to manage task scheduling and load balancing in a virtualized environment, and propose a container-based edge service management system, which reduced bandwidth utilization and energy consumption. Yang *et al.* [13] develop different service load optimization algorithms and service allocation algorithms for different service providers. Guo *et al.* [14] propose a PSO-based task scheduling optimization method to minimize the cost. Wan *et al.* [15] propose an energy aware task scheduling method based on fog computing to solve the energy consumption problem of task clusters in the manufacturing industry, which can achieve load balancing in the multi-task and multi-objective environments.

Effective load balancing is one of the most common methods to reduce service delay in edge computing, to avoid overload or waste of resources of any edge server. In order to shorten the task completion time as much as possible, Zeng *et al.* [16] comprehensively consider the load balance between edge devices and edge servers, the placement of task images, and the balance of I/O interrupt requests between storage servers, and propose a solution with high computational efficiency. Souza *et al.* [17] propose a strategy for service alloca-

tion problem as a multidimensional knapsack problem. They apply their model to the integration of fog and cloud computing to optimize latency, load, and energy consumption. Mishra *et al.* [18] propose a service allocation framework based on meta-heuristics to adapt to the heterogeneity of resources in the fog computing environment by analyzing the equipment energy consumption and the maximum task completion time. Liu *et al.* [19] balance the load according to the queuing time, execution time and transmission time of tasks, and propose a one-dimensional search algorithm to find the optimal task scheduling strategy. Li *et al.* [20] propose a two-stage algorithm for computing resource scheduling in edge layer based on task delay constraint and threshold strategy.

Some researchers have solved the load balancing problem by modeling the relationship between computing nodes and tasks as a weighted bipartite graph. Li *et al.* [21] propose a task scheduling method based on cache location for intensive data tasks in edge computing environment. First, the cache is placed, and then the data location is used as the weight to establish a weighted bipartite graph to solve the task scheduling problem. Wang *et al.* [22] propose a dynamic task scheduling algorithm based on weighted bipartite graph considering the dynamic characteristics of tasks and computing nodes, which transforms the scheduling problem into the maximum weighted bipartite graph matching problem. Zhang *et al.* [23] propose a service adaptation method for cloud end dynamic integration. In this method, the adaptation problem between cloud services and end services is modeled as a bipartite graph matching problem to ensure the minimum global average request response time of end service instances in the adaptation process.

As mentioned above, many researchers have established different objective functions to reduce various SLA parameters of services (i.e., response time, service availability, delay and cost). Each method plays an important role in a specific edge computing scene. To solve the problem of limited resources of edge computing nodes, we propose a two-phase load balancing algorithm based on weighted bipartite graph, which makes full use of network edge resources, reduces the makespan of users, and improves the user service experience.

III. Load Balancing Framework for Edge Computing

Our proposed load balancing architecture for edge computing consists of three layers, namely edge device layer, edge layer and cloud layer, as shown in Fig.1. The edge device layer (including different smart devices) sends the tasks to the nearby edge data center. The edge data center consists of edge servers. Inside the edge server is a cluster based on Docker container. Compared with the traditional virtual machine cluster, Docker container has the characteristics of less resource consumption, fast startup, high deployment efficiency

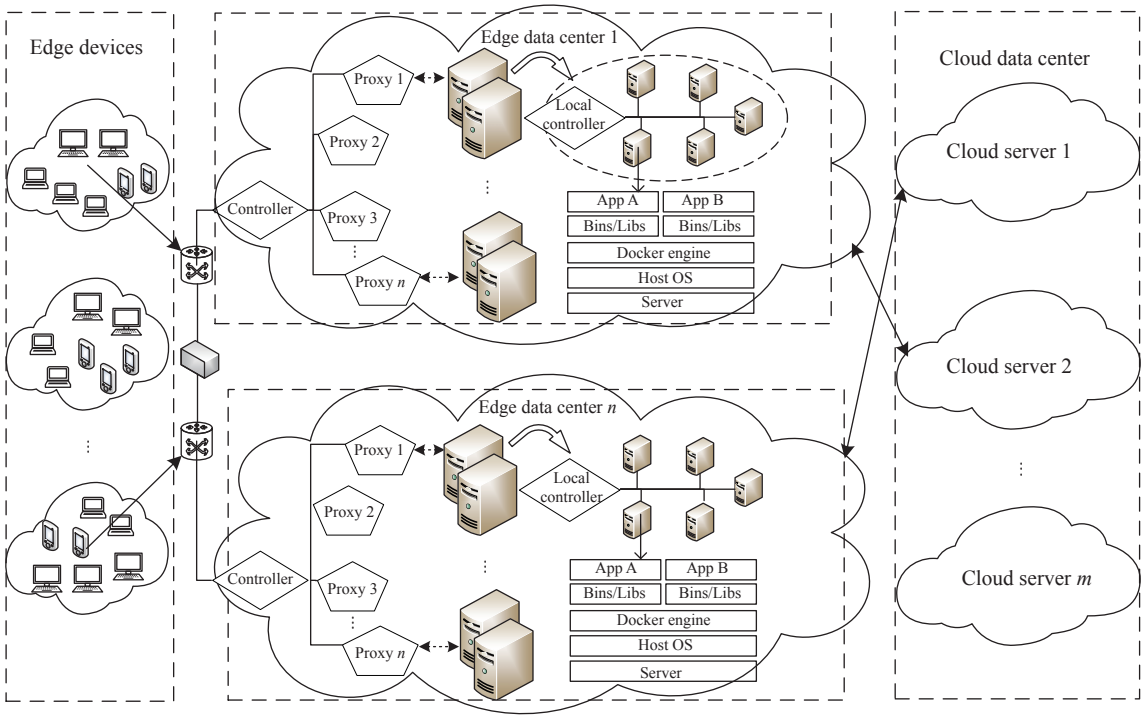


Fig. 1. Load balancing architecture for edge computing.

and good scalability. Container can better ensure the reliability and effectiveness of the flexible supply of cluster resources, so that applications can run almost anywhere in the same way. After receiving the task request, the edge data center selects the appropriate Docker container for processing through the internal scheduling algorithm, and finally returns the processing result to the edge device. The cloud server can provide all the services.

According to the above system architecture, we have the following assumptions.

1) The edge data center can completely cover the entire network service area of the edge device, so when the edge device sends the task request, it can select the nearby edge data center to process the task.

2) For any two edge data centers, network paths are connecting them. For each edge data center, there is a network path to connect it with the cloud data center.

3) For each task request, if the resources in the edge data center are insufficient to process the task, the task will be sent to the cloud data for processing.

4) When the Docker in the edge server receives a task, it can only process one task at a time. If new tasks enter the container at this moment, they need to wait for the previous task to complete. Cloud data center is powerful, so it can perform multiple tasks at the same time.

IV. Load Balancing Algorithm Based on Weighted Bipartite Graph

The proposed load balancing algorithm is based on the network architecture designed by Section III, which is divided into two phases: load balancing between edge nodes and load balancing between containers. Among them, the load balancing between edge nodes is responsible for matching tasks to a specific edge server. The load balancing between containers is responsible for allocating tasks to a specific container. Suppose there are c edge data centers, which contain m edge servers; In addition, there are n tasks waiting to be processed.

First, the load balancing phase between edge nodes is carried out. After the user publishes the task request, the task will be matched to the appropriate edge server. In this process, the task matching problem is modeled as a dynamic weighted bipartite graph maximum perfect matching problem. Combined with the queuing theory of multi-server waiting system ($M/M/m/\infty$), the tasks in the queue are modeled. The average waiting time of the task is minimized, and the optimization algorithm is used to speed up the convergence speed. The weight between task and edge server is obtained by task scheduling priority (sp), task resource similarity (Sim) and data transmission cost (Cost). The maximum perfect matching algorithm, i.e., Kuhn-Munkres (KM) algorithm in a bipartite graph is optimized to minimize the average response time and energy consumption.

Second, the load balancing phase between containers is performed to determine which Docker container the task will be assigned to process. In this process, we will first quantitatively describe the makespan and energy consumption to keep a balance between them. The problem is defined as a bi-objective minimization problem. Then, the feasible solution space of the problem is determined according to the particle swarm algorithm for fast search. At this point, the Docker container allocation process is over. After getting the specific allocation result, the cluster schedules the task and allocates the task to a specific container for processing. Finally, the cluster returns the processed result to the edge device.

1. Load balancing between edge nodes

In the phase of load balancing between edge nodes, the task scheduling problem is modeled as a dynamic weighted bipartite graph maximum perfect matching problem, and the task in the task queue is modeled with queuing theory model. The load balancing algorithm between edge nodes is shown in Fig.2. This section mainly describes the construction of task matching model based on dynamic weighted bipartite graph, the construction of task queue based on $M/M/m/\infty$, the calculation of weight between task and edge server, and the maximum weight perfect matching of bipartite graph. The task matching model based on the dynamic weighted bipartite graph is dynamic growth, so the matching vertex in graph G is deleted after the task is matched, which means that the vertex is not considered in the next matching. The virtual node is added to represent the adapted proxy vertex, and the weight between task vertex and virtual node is 0. The construction of task queue based on $M/M/m/\infty$ is to get the optimal task sorting queue into bipartite graph modeling, to reduce the average waiting time of tasks. The weight between task and edge server is obtained by (sp), (Sim) and (Cost).

1) Task matching model based on dynamic weighted bipartite graph

We assume that the task set is $T = \{t_\lambda, |\lambda = 1, 2, \ldots, n\}$ and the edge server set is $P = \{p_\gamma, |\gamma = 1, 2, \ldots, m\}$, where $n \geq m$. We model the matching between tasks and edge servers as a dynamic weighted bipartite graph problem. The vertex of one end of bipartite graph is task set T, and the other end is edge server set P, so weighted bipartite graph $G = \langle V(G), E(G)\rangle$, where $V(G) = T \cup P$, $E(G) = \{e_{ij}, |e_{ij} = (t_\lambda, p_\gamma), t_\lambda \in T, p_\gamma \in P\}$. To get the best match, weight is ad-

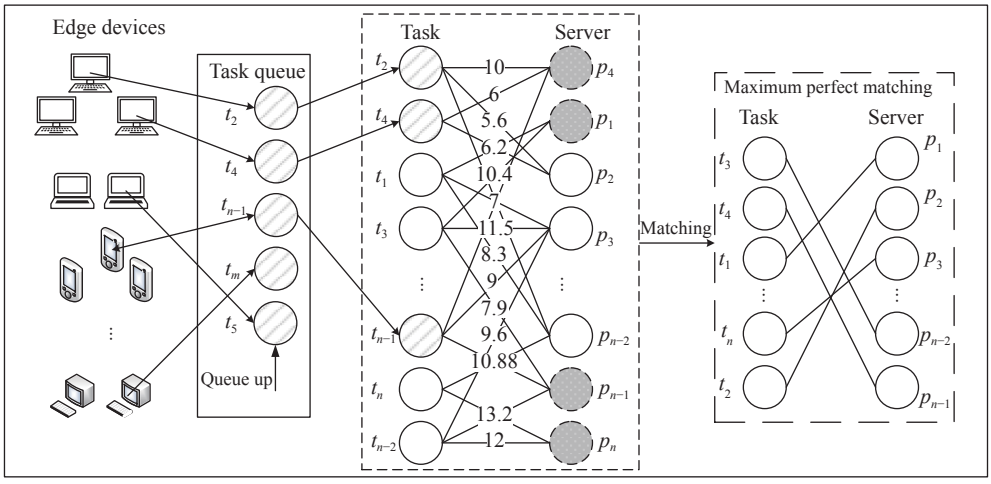


Fig. 2. Load balancing algorithm between edge nodes.

ded between the vertices of bipartite graph. The weight of each vertex is determined by the priority value of task scheduling *sp*, the similarity between task and resource *Sim*, and the cost of data transmission Cost. The specific calculation method of weight is shown in Section IV.1.3). After that, we use the maximum perfect matching algorithm (KM) to get the maximum matching. The number of edge servers can be predicted in advance. However, tasks arrive dynamically, so the corresponding weighted bipartite graph G grows dynamically. In order to get the maximum perfect match of bipartite graph, we need to satisfy that the number of servers in the graph is equal to the number of tasks. After the task gets the server matching, it will delete the matching vertex in graph G, which means that the vertex will not be considered in the next matching, and the virtual server node will be added to represent the adapted proxy vertex. The weight between task vertex and virtual server node is 0.

2) Task queue modeling based on $M/M/m/\infty$ queuing theory

For tasks, its generation follows a negative exponential distribution with mean value λ, and the processing time of each task follows a negative exponential distribution with parameter μ. At time t, the probability of system state n is shown by $P_n(t)$. $P(t)$ is the probability that the system state is n in steady state. $N(T)$ is the number of tasks arriving in the period time $[0, t)$. Suppose there are m edge servers, $d = n - |T|$ represents the maximum number of tasks that can be held in the task set, $d_{\max} = n$. n is the average number of tasks matching the edge server in the task set, and $p_n\ (n = 0, 1, 2, \ldots)$ is the distribution of the stationary state of the queuing model. For any state n, the average number of times to enter the state in unit time and the average number of times to leave the state in unit time should be equal, which is called "birth and death process." Therefore, according to the birth and death process, the steady state of the queue in any state is as follows:

$$\begin{aligned}
&0 \quad \mu_1 p_1 = \lambda_0 p_0 \\
&1 \quad \lambda_0 p_0 + \mu_2 p_2 = (\lambda_1 + \mu_1) p_1 \\
&2 \quad \lambda_1 p_1 + \mu_3 p_3 = (\lambda_2 + \mu_2) p_2 \\
&\qquad \vdots \\
&n \quad \lambda_{n-1} p_{n-1} + \mu_{n+1} p_{n+1} = (\lambda_n + \mu_n) p_n
\end{aligned} \tag{1}$$

It can be seen from the above steady-state equation that

$$\begin{aligned}
&0 \quad p_1 = \frac{\lambda_0 p_0}{\mu_1} \\
&1 \quad p_2 = \frac{\lambda_1 \lambda_0}{\mu_1 \mu_2} p_0 \\
&2 \quad p_3 = \frac{\lambda_2 \lambda_1 \lambda_0}{\mu_1 \mu_2 \mu_3} p_0 \\
&\qquad \vdots \\
&n \quad p_{n+1} = \frac{\lambda_n \lambda_{n-1} \ldots \lambda_0}{\mu_1 \mu_2 \ldots \mu_{n+1}} p_0
\end{aligned} \tag{2}$$

Let $C_n = \frac{\lambda_{n-1} \lambda_{n-2} \ldots \lambda_0}{\mu_1 \mu_2 \ldots \mu_n}$, $n = 1, 2, 3, \ldots$, then

$$p_n = C_n p_0,\ n = 1, 2, 3, \ldots \tag{3}$$

According to the requirement of probability distribution: $\sum_{n=0}^{\infty} p_n = 1$, so $[1 + \sum_{n=1}^{\infty} C_n] p_0 = 1$, then

$$p_0 = \frac{1}{1 + \sum_{n=1}^{\infty} C_n} \tag{4}$$

For m server nodes, record the distribution $p_n = P\{N = n\}\ (n = 0, 1, 2, \ldots, k)$ in stationary state,

$$\lambda_n = \lambda,\ n = 0, 1, 2, \ldots, m \tag{5}$$

$$\mu_n = \begin{cases} n\mu, & n = 0, 1, 2, \ldots, m \\ m\mu, & n = m, m+1, \ldots \end{cases} \tag{6}$$

Let $\rho = \frac{\lambda}{\mu}$, because $\rho_m = \frac{\rho}{m} = \frac{\lambda}{m\mu}$, so when $\rho_m < 1$, from (3) and (4), we can see that

$$C_n = \begin{cases} \dfrac{(\lambda/\mu)^n}{n!}, & n = 1, 2, \ldots, m \\ \dfrac{(\lambda/\mu)^n}{m! m^{n-m}}, & n \geq m \end{cases} \tag{7}$$

It can be deduced that

$$p_n = \begin{cases} \dfrac{\rho^n}{n!} p_0, & n = 1, 2, \ldots, m \\ \dfrac{\rho^n}{m! m^{n-m}} p_0, & n \geq m \end{cases} \tag{8}$$

where

$$p_0 = \left[\sum_{n=0}^{m-1} \frac{\rho^n}{n!} + \frac{\rho^n}{m!(1-\rho_m)} \right]^{-1} \tag{9}$$

In the stationary state, if $n < m$, the new task can directly match the dynamic bipartite graph and the proxy node without waiting, as shown in (10).

$$P_{t_i} = 1 \tag{10}$$

Otherwise, if $n \geq m$, the new task needs to wait after entering the queuing system, as shown in (11).

$$P_{t_i} = \sum_{n=s}^{\infty} p_n = \frac{\rho^m}{m!(1-\rho_m)} p_0 \tag{11}$$

Finally, we can get the average queue length L_m, average queue length L_m, average processing time W_m, and the average waiting time of the task W_q, as shown in (12)–(15).

$$L_m = \mathrm{L}_q + \rho \tag{12}$$

$$L_q = \frac{p_0 \rho^m \rho_m}{m!(1-\rho_m)^2} = \frac{P_{t_i} \rho_m}{1-\rho_m} \tag{13}$$

$$W_m = \frac{L_m}{\lambda} \tag{14}$$

$$W_q = \frac{L_q}{\lambda_{\mathrm{e}}} = W_m - \frac{1}{\mu} \tag{15}$$

Generally, in the queuing system, the task is queued according to the first come first served, but this will cause the phenomenon of a long queue. To optimize the average waiting time and reduce the length of the task queue, we use PSO algorithm to manage the task queue. Firstly, the average waiting time of the task is calculated, then the optimal value returned by PSO is the correct order of the task in the queue. The PSO algorithm is used to get the optimal queue sequence to minimize task congestion. Therefore, our goal is to minimize the average waiting time, as shown in (16).

$$\text{ObjectiveFunction} = \min\{W_q\} \tag{16}$$

In the PSO algorithm, the fitness function used to calculate each particle solution is as follows:

$$\text{Fitness} = \min\left\{ \frac{1}{n} \sum_{i=1}^{n} WT_{t_i} \right\} \tag{17}$$

where WT_{t_i} is the average waiting time of the t_i task and n is the number of tasks in the queue. When the PSO algorithm returns the optimal value, we reorder the task queue according to the value. When there is a new task to enter the queue, it will first determine whether the dynamic bipartite graph has completed a maximum perfect match. If it is finished, the saturated vertices in the bipartite graph will be deleted, and the top task in the queue will be selected to insert into the bipartite graph. Otherwise, the task will continue to wait in the queue. Therefore, the optimized queue can maintain a transient stable state in the process of bipartite graph matching.

3) Weight calculation between task and edge server

In the process of modeling the task matching problem as the dynamic weighted bipartite graph maximum matching problem, the definition of weight is very important. It determines the advantages and disadvantages of the task allocation algorithm and whether it is effective or not. Here, the weight between task and edge server is calculated by sp, Sim, and Cost.

Task scheduling priority value (sp) Task scheduling priority value sp is determined by the task priority value tp_i (where i is the ith task), so $sp_i = tp_i$. tp_i is defined by the user when transferring tasks, $tp_i = \{b_1, b_2, b_3 | b_1 < b_2 < b_3\}$.

Similarity between task and resource (Sim) A task is defined as a row vector $t_\lambda = [tc_\lambda, tm_\lambda]$, which tc_λ represents the CPU requirement of the task t_λ and tm_λ represents the memory requirement of task t_λ. The resource of edge server is defined as row vector $p_\gamma = [pc_\gamma, tm_\gamma]$, which pc_γ represents the CPU of edge server and tm_γ represents the memory of edge server. So the similarity between tasks and resources is as follows:

$$Sim(t_\lambda, p_\gamma) = \frac{t_\lambda \cdot {p_\gamma}^{\mathrm{T}}}{|t_\lambda||p_\gamma|} \tag{18}$$

Data transmission cost (Cost) The data transmission cost is determined by the network bandwidth and the distance between the task and the edge server, so the data transmission cost is as follows:

$$\mathrm{Cost}_{\lambda,\gamma} = \frac{L_\lambda}{\mathrm{Band}} dis \tag{19}$$

where L_λ represents the size of the task, Band represents the network bandwidth, and dis represents the distance between the task and the edge server.

So the weight $w_{\lambda,\gamma}$ is

$$w_{\lambda,\gamma} = a_1 \cdot \frac{Sim(t_\lambda, p_\gamma)}{\overline{Sim(t_\lambda, p_\gamma)}} - a_2 \cdot \frac{sp_\lambda}{\overline{sp_\lambda}} - a_3 \cdot \frac{\mathrm{Cost}_{\lambda,\gamma}}{\overline{\mathrm{Cost}_{\lambda,\gamma}}} \quad (20)$$

where a_1, a_2 and a_3 are the weight coefficients of the three influence factors, and $\overline{Sim(t_\lambda, p_\gamma)}$, $\overline{sp}$ and $\overline{\mathrm{Cost}_{\lambda,\gamma}}$ are defined as follows:

$$\begin{cases} \overline{Sim(t_\lambda, p_\gamma)} = \dfrac{\sum \lambda \sum \gamma Sim(t_\lambda, p_\gamma)}{\lambda \cdot \gamma} \\ \overline{sp} = \dfrac{\sum \lambda \sum \gamma sp_\lambda}{\lambda \cdot \gamma} \\ \overline{\mathrm{Cost}_{\lambda,\gamma}} = \dfrac{\sum \lambda \sum \gamma \mathrm{Cost}_{\lambda,\gamma}}{\lambda \cdot \gamma} \end{cases} \quad (21)$$

4) Maximum weight perfect matching of bipartite graph

In our algorithm, we use KM algorithm [23] to get the maximum weight perfect match of bipartite graph and get the best allocation method after getting the weight according to Section IV.1.3). The main steps of the algorithm are as follows:

a) Select the initial feasible fixed-point label l, determine G_l, and select a pair set M in G_l.

b) If all the vertices in T are paired by M, then stop, and M is the most weighted perfect pair set of G; Otherwise, take the unpaired vertex u of G_l, let $S = \{u\}$, $Q = \emptyset$.

c) If $N_{G_l}(S) \supset Q$, go to step d); If $N_{G_l}(S) = Q$, take $\partial_1 = \min_{x \in S, y \in Q}\{l(x) + l(y) - w(xy)\}$, in which $\bar{l}(v) = \begin{cases} l(v) - \partial_1, v \in S \\ l(v) + \partial_1, v \in Q \\ l(v), \text{otherwise} \end{cases}$, $l = \bar{l}$, $G_l = G_{\bar{l}}$, and $N_{G_l}(S)$ is the set of adjacent vertices of S in G_l.

Algorithm 1 Load balancing between edge nodes

Input: M_s is a server instance, N_s is a task instance, k is the node queue that enters the task instance.

Output: $map(M_s, N_s) = \{ms \to ns, ms \in M_s, ns \in N_s\}$.

Begin

$Q' \leftarrow PSO(Q)$ // Optimize queue sequence;

if $(Q \neq \theta)$

Establish a bipartite diagram $G(M_s \cup k, E)$ for M_s and k according to Section IV.1.1);

calculate $\omega_{\lambda,\gamma}$ according to (20);

initialize matching $M \subseteq E(G)$

if ($\exists$ $ms \in M_s$ is saturated and satisfies (10))

$\{WMW = M;\}$

else

Get the unsaturated point of M, $Y = \{y, y \in M_s\}$, $T = Q$;

Initializing the feasible vertex label L in Y according (11);

Obtain the equal subgraph E_L of G;

$\partial_L - \min\{L(x) + L(y) - w(xy) | x \in X, y \in (K - T)\}$;

Modify the feasible label of each vertex as $\bar{l}(v) = \begin{cases} l(v) - \partial_L, v \in S \\ l(v) + \partial_L, v \in T; \\ l(v), \text{otherwise} \end{cases}$

Let $L = L', E_L = E_{L'}$, give a match M' of E_L again;

$P = (y_k)$ is the augmented path of M' in E_L;

then

$M^* = M' \oplus P$ // Execute the Hungarian algorithm until the M^* of E_L is found;

$WMW = M^*$;

$\overline{M_s} = \{ms = e.ms,\ e \in WMW\}$;

$\bar{k} = \{k = e.k, e \in WMW\}$;

$\overline{N_s} = \{ns = k.sid, k \in \bar{k} \& e \in NS\}$;

$map(M_s, N_s) = \{ms \to ns, ms \in M_s, ns \in N_s\}$

$M_s = M_s - \overline{M_s} + \{m'_1, m'_2, m'_3, \ldots, m'_c\}$

$k = k - \bar{k} + \{k'_1, k'_2, k'_3, \ldots, k'_c\}$, $\{k'_1, k'_2, k'_3, \ldots, k'_c\} \in Q$

//Update the N_s and k sets so that $|M_s| = |k|$;

$map(M_s, N_s)$

End

So far, the maximum weight perfect matching of bipartite graph is obtained, that is, the specific matching scheme of task mapping to edge server is obtained. Algorithm 1 implements the load balancing phase between edge nodes of the algorithm. The time complexity of the KM algorithm is $O(n^3)$. When a virtual server node is inserted into G, the improved KM algorithm is only called once, and the waiting time of the key node is W_q. Therefore, the time complexity of the entire process is $O(W_q n^3)$, which is $O(n^3)$.

2. Load balancing between containers

Docker container has the characteristics of less resource consumption, fast startup speed, high deployment efficiency and good scalability. It can also ensure the reliability and effectiveness of the flexible supply of cluster resources, so that applications can run in the same way almost anywhere. Compared with virtual machine (VM), container is a relatively lightweight virtualization instance [24]. In a server, there can be many containers to manage task scheduling and load balancing in a virtualized environment. Therefore, in terms of reducing the bandwidth utilization and energy consumption, their scheduling and migration are more suitable than traditional VMs. As mentioned above, the edge server we use is a cluster based on Docker container.

In this section, we will discuss the load balancing phase between containers, which is mainly divided into the modeling of bi-objective minimization problem and the process of container cluster allocation. The bi-ob-

jective minimization problem is modeled by quantitatively describing the makespan and the energy consumption to obtain the objective function. In the process of container cluster allocation and scheduling, a particle swarm optimization algorithm is used to determine the feasible solution space of the above modeling problem. The particle swarm algorithm performs a fast search to obtain the approximate optimal value. Finally, the container cluster schedules the tasks according to the results, assigns the tasks to a specific container for processing, and returns the processed results to the edge devices.

1) Modeling of bi-objective optimization problem

In the process of load balancing of edge nodes, tasks are matched to each edge server. Then the algorithm executes the task allocation process of the container cluster to determine which specific Docker container to assign the task to for processing. Suppose a container set is $C = \{c_\theta, |\theta = 1, 2, \ldots, s\}$. In this process, the system first describes the problem quantitatively according to the makespan and the energy consumption, and defines the problem as a bi-objective minimization problem.

The makespan of a task represents the time required in the whole process from task input to result output, so the makespan is mainly determined by task calculation time, task waiting time and task transmission time. The calculation time CT_λ of the task is determined by the size of the task and the processing speed, as shown in (22).

$$CT_\lambda = \frac{L_\lambda}{V_\lambda} \tag{22}$$

where L_λ is the size of the task (in millions of instructions (MI)) and V_λ is the processing speed of the task. The task waiting time Wait_λ is the average waiting time WT_{t_i} of the task matching phase with the edge server and the service delay Delay_λ of the Docker container allocation phase, as shown in the (23).

$$\text{Wait}_\lambda = WT_{t_i} + \text{Delay}_\lambda \tag{23}$$

Task transmission time T_λ is determined by task size L_λ and network bandwidth, as follows:

$$T_\lambda = \frac{L_\lambda}{\text{Band}} \tag{24}$$

So the makespan is

$$\text{Makespan} = CT_\lambda + \text{Wait}_\lambda + T_\lambda \tag{25}$$

Energy consumption represents the energy consumed by the edge server. The edge server has two states: idle state and active state. It is assumed that the energy consumption of an idle state is 60% of that of an active state. When container c_θ is active, the energy consumed is β_θ (Joules/MI); When container c_θ is idle, the energy consumed is α_θ (Joules/MI). Here, $\beta_\theta = 10^{-8} \times (MIPS_\theta)^2$, $\alpha_\theta = 0.6 \times \beta_\theta$ Joules/MI. The task size is L_λ, and the data processing speed is V_λ, so the task calculation time is $CT_\lambda = \frac{L_\lambda}{V_\lambda}$, and each task will correspond to a calculation time, so we put it into the ETC matrix, and the matrix elements are represented by $ETC_{\lambda,\theta}$. As shown in (26).

$$ETC_{\lambda,\theta} = CT_\lambda = \frac{L_\lambda}{V_\lambda} \tag{26}$$

Set $\chi_{\lambda\theta}$ to indicate whether task t_λ is executed in container c_θ. If $\chi_{\lambda\theta}=1$, task t_λ is executed in container c_θ; If $\chi_{\lambda\theta}=0$, it means that task t_λ is not executed in container c_θ. So the total service time of c_θ is

$$ET_\theta = \sum_{\lambda=1}^{n} \chi_{\lambda\theta} \times ETC_{\lambda,\theta} \tag{27}$$

So the energy consumption of container c_θ is

$$E(c_\theta) = [ET_\theta \times \beta_\theta + (\text{Makespan} - ET_\theta)] \times \text{MIPS}_\theta \tag{28}$$

The total energy consumption is

$$E = \sum_{\theta=i}^{s} E(c_\theta) \tag{29}$$

Therefore, the objective function of task allocation optimization among containers is as follows:

$$\text{Min} : \Phi_{\text{Load}} = \lambda_1 \cdot \text{Makespan} + \lambda_2 \cdot E \tag{30}$$

where λ_1 and λ_2 are the weight coefficients between Makespan and E. Here, we set $\lambda_1 = \lambda_2 = 0.5$.

2) Container cluster allocation process

After the objective function is obtained, the feasible solution space of the problem is determined according to the PSO algorithm. The PSO algorithm will find the optimal value of the dual-objective optimization problem to complete the final task assignment. PSO algorithm is simple to implement, high efficiency, and few parameters. Especially the algorithm with natural real number coding characteristics is more suitable for real-time optimization problems. The notations and definitions are shown in Table 1.

According to the basic formula of PSO algorithm, we can get the following results:

$$V_i^{k+1} = \omega(k) V_i^k + c_1 r_1 (P_{\text{best},i}^k - X_i^k) + c_2 r_2 (P_{\text{global},i}^k - X_i^k) \tag{31}$$

$$X_i^{k+1} = X_i^k + V_i^{k+1} \tag{32}$$

Table 1. Notations and definitions of particle swarm optimization

Notartions	Definitions
c_θ	Number of containers
m	Number of particles
X_i	The position of particle i
V_i	The velocity of particle i
ω	Inertia weight $\omega \in [0,1]$
η	Random number $\eta \in [k, k_{\max}]$
c_1, c_2	Acceleration coefficient equal to 2
r_1, r_2	Pseudo random number $r_1, r_2 \in [0,1]$
$P_{\text{best},i}$	The best place in history for particle i
$P_{\text{global},i}$	The best position for all particles

In (31), ω is an important parameter in PSO algorithm. It balances the global or local search ability of particles. Setting a higher ω will promote global search, and a lower ω will promote fast read local search. Therefore, to avoid premature convergence of particles, we use the exponential decreasing weight formula [25] to improve the global search ability and search accuracy of particle swarm optimization algorithm. The exponential decreasing weight formula is shown in (33). When the maximum number of iterations or minimum load balancing is achieved, the optimal particle position can be determined.

$$\omega(k) = (\omega_{\max} - \omega_{\min})\exp\left[-\frac{k^2}{\eta k_{\max}}\right] + \omega_{\min} \tag{33}$$

At this point, the Docker container allocation process is finished. After getting the specific allocation result, the cluster schedules the task to a specific container for processing, and returns the processed result to the edge device. The container cluster is responsible for optimally assigning tasks to different containers with minimum energy consumption and makespan. Algorithm 2 implements the load balancing phase between the containers of the algorithm. The input of the algorithm has n task requests and θ Docker containers. Therefore, the time complexity of Algorithm 2 is $O(\theta \times n)$.

Algorithm 2 Load balancing between containers

Input: λ is the number of iterations, ξ is the minimum balance difference, ETC matrix
Output:Allocation result of services to Dockers.
Begin
  initialization // initialized particle;
  for $i \leftarrow 1$ to m
    for $j \leftarrow 1$ to n
      $X_{i,j}^0 = \theta_{\Gamma_i}(m_j)$
    end for
  end for
  do
  for $i \leftarrow 1$ to m //update the position of particle;
    //achieve the previous best position of particle;
    for $j \leftarrow 0$ to n
      //Φ_{Load} is calculated according to the public (30);
      if($\Phi_{\text{Load}}\left(X_i^j\right) \le \Phi_{\text{Load}}(P_{\text{best},i})$)then
        // update the best position for particle;
        $P_{\text{best},i} = X_i^j$;
      end if
    end for
  if ($\Phi_{\text{Load}}(P_{\text{global}}) \le \Phi_{\text{Load}}(P_{\text{best},i})$) then
    $P_{\text{global}} = P_{\text{best},i}$ //achieve the best neighbor particle
  end if
  V_i^{k+1}
  $X_i^{k+1} = X_i^k + V_i^{k+1}$
  $k++$
  //the terminal condition
while ($\Phi_{\text{Load}}(X_{\text{best}}) \le \xi$ or $k \ge \lambda$)
end while

V. Experimental and Performance Evaluation

1. Experimental environment

1) Edge server

Each edge data center uses four edge servers to build a container cluster. In this cluster, there are one master node and three slave nodes. The specific configuration information of each server node is shown in Table 2.

Table 2. Calculate node configuration information

CPU	Intel E5-2680 @2.80 GHz
Memory	4 GB
Disk	6300 GB
OS	CentOS 6.5
JVM version	Java 1.8.0
Docker version	Docker 1.5

2) Network environment

The edge servers in the edge data center are connected through 56 Gbps high-speed switch, and the edge servers and external devices are connected through the Gigabit Ethernet switch of TP-Link.

3) Dataset

Since there is no standard experimental platform and test data set, we generate the experimental data asynchronously. We assume that the number of incoming task requests per time unit is fixed, and the number of available edge servers per request ranges from 2 to 8. Three positive integers are assigned to each task, which are task priority value, task data size and task location. Each task needs three kinds of CPU or memory resources, and the number of each resource is

randomly generated in $(2, 5)$. Each edge server has three kinds of resources, and the available quantity range of each resource is 5–10. The processing speed of each edge server task is randomly generated from 1–5, and the data transmission speed between each edge server and the cloud is 1.

2. Experiment and analysis

1) Response time

We analyze the effect of different parameters on the task response time. We set two groups of parameters, which are the number of task requests per unit time and the number of edge servers, as shown in Table 3. In each group, only one parameter changes and the other remains unchanged. All experiments were repeated 20 times and the average value was used as the result. In addition, we compare the proposed algorithm with baseline algorithms FCFS, Min-Min and Max-min. The number of iterations is set to 1000.

Table 3. Parameter setting

Setting	1	2
Number of task requests	60–100	60
Number of edge servers	5	5–9

The impact of the number of task requests on response time. We set the experimental parameters according to setting 1 in Table 3, and analyze the impact of different task requests per unit time on the response time of five scheduling algorithms. The results are shown in Table 4. From the table, we can see that with the increase of the number of task requests, the response times of the five scheduling algorithms are increasing. This is because when other parameters are fixed, with the increase of task requests, more tasks are assigned to the edge server, which increases the load of the edge server. As a result, the waiting time of the task increases and the response time increases.

Table 4. The impact of different number of task requests on response time (ms)

Method	60	70	80	90	100
LBA-EC	26.29	30.14	34.98	38.22	43.16
FCFS	42.38	43.31	47.11	54.98	58
Min-Min	41.97	43.42	49.58	58.95	63.75
Max-Min	46.21	46.14	50.97	60.28	64.71

The impact of the number of edge servers on response time. We set the experimental parameters according to setting 2 in Table 3, and analyze the influence of a different number of edge servers on the response time of five scheduling algorithms. The results are shown in Table 5.

As can be seen from Table 5, with the increase of edge servers, the response time is decreasing. This is because, with other parameters unchanged, the increase of edge servers can make more servers process task requests, shorten the waiting time of tasks, and further reduce the response time of tasks.

Table 5. The impact of different number of edge servers on response time (ms)

Method	5	6	7	58	9
LBA-EC	26.29	19.49	17.62	14.90	14.94
FCFS	42.38	41.16	37.26	33.96	25.19
Min-Min	41.97	44.13	39.25	43.79	25.97
Max-Min	46.21	45.13	40.30	35.91	26.57

2) Efficiency evaluation

In this part, we conduct two groups of experiments to test the scalability of our proposed algorithm. Similarly, we analyze the effect of the number of task requests and the number of edge servers on the efficiency of the algorithm.

Figs.3 and 4 show the change of execution time of our algorithm with the number of task requests and the number of edge servers, respectively. Among them, blue represents LBA-EC algorithm and red represents FCFS algorithm. As can be seen from Fig.3, as the number of task requests increases, the execution time of our algorithm also increases. This is because as the number of task requests increases, the time for particles to find the optimal solution becomes longer. However, compared with FCFS algorithm, the execution time of LBA-EC will be longer. There are two main reasons for this. The first is that the algorithm proposed in this paper adds load balancing operations between the proxy and the edge server before the load balancing between the containers. Then the particle swarm algorithm needs to go through multiple iterations to converge to the minimum and complete the scheduling between tasks and virtual machines. As can be seen from Fig.4, with the increase of the number of edge servers, the execution time of LBA-EC increases slightly. This is because the in-

crease of edge servers makes the algorithm schedule for more edge servers in the scheduling process. Since the number of edge servers can not affect the iteration of the algorithm, the execution time of the algorithm will not increase sharply.

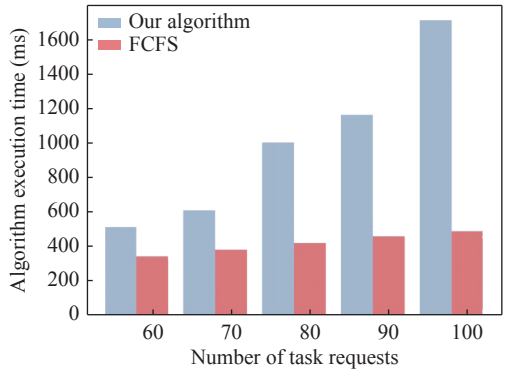


Fig. 3. The impact of task request number on algorithm execution time.

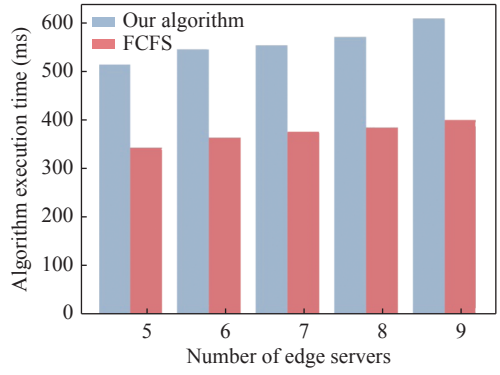


Fig. 4. The impact of edge server number on algorithm execution time.

VI. Conclusions and Future Work

In this paper, we propose a load balancing algorithm based on weighted bipartite graph for edge computing (LBA-EC). The algorithm is divided into two phases: load balancing between edge nodes and load balancing between containers. In the first phase, the tasks are matched to different edge servers. In the second phase, the tasks are optimally allocated to different containers in the edge server according to the two indicators of energy consumption and completion time. It makes full use of network edge resources, reduces user delay, and improves user service experience. We conducted experiments on the algorithm and analyzed the impact of task response time and algorithm execution time under different number of task requests and edge servers. The experimental result show that our algorithm can effectively map all tasks to available resources with a shorter completion time.

Compared with other baseline load balancing algorithms, our proposed load balancing algorithm for edge computing has improved the response time. However, because the algorithm increases the load balancing operation between tasks and edge servers before the load balancing between containers, the algorithm efficiency is lower than other algorithms. In the follow-up study, we can further consider improving the efficiency of the algorithm.

References

[1] H. Wang, J. Gong, Y. Zhuang, *et al.* "Task scheduling for edge computing with health emergency and human behavior consideration in smart homes," in *Proceedings of 2017 IEEE International Conference on Big Data*, Boston, MA, USA, pp.1213–1222, 2017.

[2] W. Shi and S. Dustdar, "The promise of edge computing," *Computer*, vol.49, no.5, pp.78–81, 2016.

[3] C. M. Fernández, M. D. Rodríguez, and B. R. Muo, "An edge computing architecture in the Internet of things," in *Proceedings of 2018 IEEE 21st International Symposium on Real-Time Distributed Computing*, Singapore, pp.99–102, 2018.

[4] G. Li, Y. Yao, J. Wu, *et al.*, "A new load balancing strategy by task allocation in edge computing based on intermediary nodes," *EURASIP Journal on Wireless Communications and Networking*, vol.2020, no.1, pp.1–10, 2020.

[5] W. Liu, Y. C. Huang, W. Du, *et al.*, "Resource-constrained serial task offload strategy in mobile edge computing," *Journal of Software*, vol.31, no.6, pp.1889–1908, 2020.

[6] H. Lu, C. Gu, F. Luo, *et al.*, "Optimization of lightweight task offloading strategy for mobile edge computing based on deep reinforcement learning," *Future Generation Computer Systems*, vol.102, pp.847–861, 2020.

[7] H. Wu, S. Deng, W. Li, *et al.* "Request dispatching for minimizing service response time in edge cloud systems," in *Proceedings of 2018 27th International Conference on Computer Communication and Networks*, Hangzhou, China, pp.1–9, 2018.

[8] K. Kaur, T. Dhand, N. Kumar, *et al.*, "Container-as-a-service at the edge: Trade-off between energy efficiency and service availability at fog nano data centers," *IEEE Wireless Communications*, vol.24, no.3, pp.48–56, 2017.

[9] H. Ishibuchi, T. Yoshida, and T. Murata, "Balance between genetic search and local search in memetic algorithms for multiobjective permutation flowshop scheduling," *IEEE Transactions on Evolutionary Computation*, vol.7, no.2, pp.204–223, 2003.

[10] M. Dorigo, M. Birattari, and T. Stutzle, "Ant colony optimization," *IEEE Computational Intelligence Magazine*, vol.1, no.4, pp.28–39, 2006.

[11] M. Dai, D. Tang, A. Giret, *et al.*, "Energy-efficient scheduling for a flexible flow shop using an improved genetic-simulated annealing algorithm," *Robotics and Computer-Integrated Manufacturing*, vol.29, no.5, pp.418–429, 2013.

[12] Z. H. Zhan, X. F. Liu, Y. J. Gong, *et al.*, "Cloud computing resource scheduling and a survey of its evolutionary approaches," *ACM Computing Surveys (CSUR)*, vol.47, no.4,

pp.1–33, 2015.

[13] L. Yang, J. Cao, G. Liang, *et al.*, "Cost aware service placement and load dispatching in mobile cloud systems," *IEEE Transactions on Computers*, vol.65, no.5, pp.1440–1452, 2015.

[14] L. Guo, S. Zhao, S. Shen, *et al.*, "Task scheduling optimization in cloud computing based on heuristic algorithm," *Journal of Networks*, vol.7, no.3, article no.547, 2012.

[15] J. Wan, B. Chen, S. Wang, *et al.*, "Fog computing for energy-aware load balancing and scheduling in smart factory," *IEEE Transactions on Industrial Informatics*, vol.14, no.10, pp.4548–4556, 2018.

[16] D. Zeng, L. Gu, S. Guo, *et al.*, "Joint optimization of task scheduling and image placement in fog computing supported software-defined embedded system," *IEEE Transactions on Computers*, vol.65, no.12, pp.3702–3712, 2016.

[17] V. B. Souza, X. Masip-Bruin, E. Marín-Tordera, *et al.*, "Towards distributed service allocation in fog-to-cloud (f2c) scenarios," in *Proceedings of 2016 IEEE Global Communications Conference*, Washington, DC, USA, pp.1–6, 2016.

[18] S. K. Mishra, D. Puthal, J. J. P. C. Rodrigues, *et al.*, "Sustainable service allocation using a metaheuristic technique in a fog server for industrial applications," *IEEE Transactions on Industrial Informatics*, vol.14, no.10, pp.4497–4506, 2018.

[19] J. Liu, Y. Mao, J. Zhang, *et al.*, "Delay-optimal computation task scheduling for mobile-edge computing systems," in *Proceedings of 2016 IEEE International Symposium on Information Theory*, Barcelona, Spain, pp.1451–1455, 2016.

[20] X. Li, J. Wan, H. N. Dai, *et al.*, "A hybrid computing solution and resource scheduling strategy for edge computing in smart manufacturing," *IEEE Transactions on Industrial Informatics*, vol.15, no.7, pp.4225–4234, 2019.

[21] C. Li, J. Tang, H. Tang, *et al.*, "Collaborative cache allocation and task scheduling for data-intensive applications in edge computing environment," *Future Generation Computer Systems*, vol.95, pp.249–264, 2019.

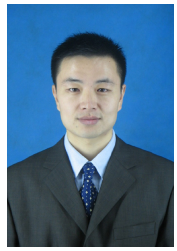
[22] T. Wang, X. Wei, Y. Liang, *et al.*, "Dynamic tasks scheduling based on weighted bi-graph in mobile cloud computing," *Sustainable Computing: Informatics and Systems*, vol.19, pp.214–222, 2018.

[23] S. L. Zhang, C. Liu, Y. B. Han, *et al.*, "DANCE: A service adaptation method for cloud-end dynamic integration," *Chinese Journal of Computers*, vol.43, no.3, pp.423–439, 2020. (in Chinese)

[24] C. Pahl, " Containerization and the paas cloud," *IEEE Cloud Computing*, vol.2, no.3, pp.24–31, 2015.

[25] D. Bernstein, "Containers and cloud: From LXC to Docker to kubernetes," *IEEE Cloud Computing*, vol.1, no.3, pp.81–84, 2014.

SHAO Sisi was born in 1997. She is a Ph.D. candidate at the School of Internet of Things, Nanjing University of Posts and Telecommunications. Her main research interests include cloud computing, edge computing task scheduling and security. (Email: 361571409@qq.com)



LIU Shangdong was born in 1979. He received Ph.D. degree in Southeast University. He is a Lecturer at the School of Computer, Nanjing University of Posts and Telecommunications. His main research interests include network behavior analysis, big data, and AI. (Email: lsd@njupt.edu.cn)

LI Kui was born in 1988. He is a Ph.D. candidate at the School of Internet of Things, Nanjing University of Posts and Telecommunications. His research interests include high performance computing, big data theory and technology. (Email: 825315689@qq.com)

YOU Shuai was born in 1995. He is a Ph.D. candidate at the School of Internet of Things, Nanjing University of Posts and Telecommunications. His research interests include machine learing, CV, and edge computing. (Email:1065858439@qq.com)

QIU Huajie was born in 1997. He is an M.S. candidate at the School of Computer, Nanjing University of Posts and Telecommunications. His research interests include cloud computing, cloud task scheduling, and reinforcement learning. (Email: 1029930034@qq.com)

YAO Xiaoliang was born in 1999. He is an M.S. candidate at the School of Computer, Nanjing University of Posts and Telecommunications. His main research interests include AI. (Email: 1392803263@qq.com)

JI Yimu (corresponding author) was born in 1978. He is a Professor at the School of Computer, Nanjing University of Posts and Telecommunications. His main research interests include the security and applications in cloud computing, bigdata, IoT, and AI. (Email: jiym@njupt.edu.cn)