An Optimal Image Storage Strategy for Container-Based Edge Computing in Smart Factory

Luxiu Yin, Juan Luo, *Member, IEEE*, and Keqin Li, *Fellow, IEEE*

***Abstract*—Edge computing provides efficient and low-latency computing services for Internet of Things applications. Container virtualization technology is widely used as an indispensable key technology in edge computing. However, the creation of the container requires reading the corresponding image file. If the image file is not stored locally, it will take a lot of time to download, which increases the user's extremely high service delay. Aiming at decreasing the download time of image files, we develop a two-stage optimization storage strategy of image files to decrease its download time based on edge computing. This strategy optimizes the image file placement in the initialization stage and the runtime stage, respectively. In the initialization stage, we propose a pseudo-polynomial time algorithm to filter all image files and select the image file combination, which best meets the capacity of the edge node for placement. In the runtime stage, we continue to optimize the local image repository based on the historical access records of the edge node. This operation can reduce the number of downloads of image files, thereby further reducing the user's service delay. In addition, we created a real data set according to the service requirements and the structure of image files on the smart factory and the access records on the DockerHub. A large number of experiments are carried out based on the data set. Experimental results show that the two-stage optimization storage strategy can greatly reduce the download time of image files, thus reducing the service delay of edge nodes and improving the service quality of edge nodes.**

***Index Terms*—Container, Docker, edge computing, resource management, virtualization.**

I. Introduction

A. Background

ACCORDING to the Cisco Internet Business Solutions Group, 50 billion devices will be connected in 2025. With the rapid development of Internet of Things (IoT), global data center IP traffic will reach 15.3 ZB [1]. According to the Cisco Global Cloud Index, 45% of the data need to be stored, processed, and analyzed at the edge of the network by 2025. Faced with such an ocean of data, the centralized computing model of the cloud data center can no longer meet the requirement of delay-sensitive applications.

To address this issue, Cisco proposed the concept of fog computing in 2012. In addition, the European Telecommunications Standards Institute proposed the basic framework for mobile-edge computing (MEC) in 2016 [2]. These computational models have the same features, which are based on cloud computing and put part of computing resources to the edge of the network, thereby reducing data transmission delay. In this article, we will use edge computing to express these similar concepts. Edge computing is similar to cloud computing, which requires virtualization technology to implement service requests for multiple users. However, unlike cloud computing, edge computing needs to serve more users and more delay-sensitive services, which proposes higher demands on virtualization technology. The current virtualization technologies are mainly divided into two types, namely, virtual machine technology and lightweight container technology. Compared with virtual machines, containers are more suitable for edge computing [3]. Sollfrank et al. [4] evaluated the performance of Docker containers in industrial automat and claimed that the container virtualization technology can meet the soft real-time requirements of industrial automation. Containers are characterized by fast bootup speed, dynamic resource quota adjustment, and low system overhead. Containers are slightly weaker than virtual machines, but the gap is only approximately 20 μs [5], the detail shows in Table I.

TABLE I
Container Versus Virtual Machine

		VMs	Containers
Performance	CPU	Close native	Close native
	Memory	Close native	Close native
	I/O	Large overhead	Close native
	Network	About 65μs	About 70μs
	Boot-up	Few seconds	Few milliseconds
Deployment	Dependencies	Strongly	Weakly
	Environment	Need to configure	Self-contain
	Live Migration	Supported	Not fully supported
Security	Isolation	Fully isolation	User isolation

B. Motivation

Currently, cloud service providers, such as Amazon and Alibaba Cloud, provide containers as a basic service to users.

Manuscript received 1 August 2022; revised 28 October 2022; accepted 9 December 2022. Date of publication 14 December 2022; date of current version 7 April 2023. This work was supported in part by the National Natural Science Foundation of China under Grant 61972140 and Grant 62002109. *(Corresponding author: Juan Luo.)*

Luxiu Yin and Juan Luo are with the College of Computer Science and Electronic Engineering, Hunan University, Changsha 410012, Hunan, China (e-mail: yinluxiu@hnu.edu.com; juanluo@hnu.edu.cn).

Keqin Li is with the Department of Computer Science, State University of New York, New Paltz, NY 12561 USA (e-mail: lik@newpaltz.edu).

Digital Object Identifier 10.1109/JIOT.2022.3229206

2327-4662 © 2022 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission.
See https://www.ieee.org/publications/rights/index.html for more information.

Authorized licensed use limited to: HUNAN UNIVERSITY. Downloaded on April 07,2023 at 13:52:09 UTC from IEEE Xplore. Restrictions apply.

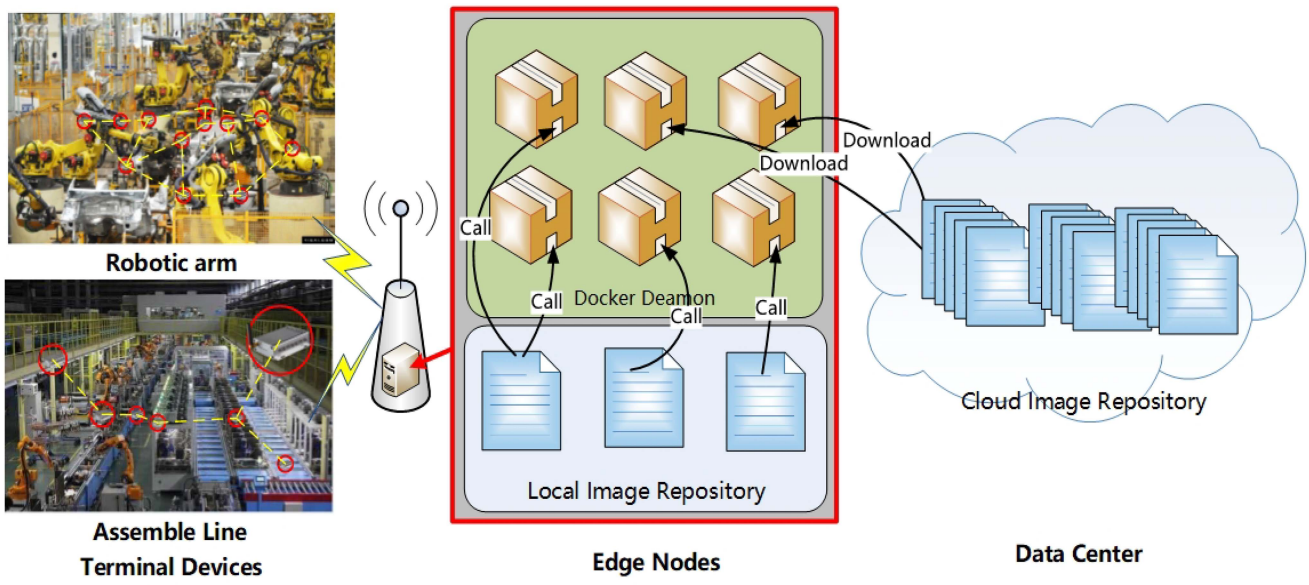

Fig. 1. Smart factory scenario.

A growing number of open-source projects are focused on containers. Docker is currently the most popular application container engine. Therefore, we use Docker as an example in this article. Docker containers and virtual machines have essential differences.

First, the isolation mechanism of Docker containers is different from that of virtual machines. Containers implement user isolation through Cgroups rather than the hypervisor used by virtual machines. Therefore, the allocated resource quota of a container can be modified when it is running, which means that containers can dynamically adjust the resource quota according to the current resource loading.

Second, containers are strongly associated with image files. An image file contains the required file system structure and contents for the corresponding container to boot up. The creation of a container must depend on their image file. If no corresponding image file exists, then Docker Daemon (a program for managing Docker containers) cannot be created, whereas virtual machines can be created without an image file.

Owing to these differences, the service process of edge computing based on containers (ECBCs) is also different from that of edge computing based on virtual machines. Virtual machines require higher overhead compared to containers. It greatly reduces the performance of the server. And in practice, smart factory application services require a specific execution environment. Although virtual machines can be replicated by virtual machine snapshots for the execution environment, but the image files of containers occupy less storage space than the environment replication of virtual machines, which makes containers more suitable for deployment on the edge nodes.

Fig. 1 shows the service process of the ECBC in smart factory scenarios. For reducing energy consumption and increasing computing efficiency, some computing and storage tasks need to be offloaded to powerful external computing devices, that is, edge nodes. After receiving the task requests of these devices, edge nodes will create isolated operating environments according to the requests on the basis of security and efficiency considerations for ensuring parallel execution. As mentioned before, the creation of containers must depend on its image file. Therefore, if the corresponding image files are not stored locally, then the edge node needs to download the image file from the remote cloud image repository (CIR) or DockerHub. The process of downloading image files causes considerable delay. As a result, edge nodes will require several storage resources for storing image files.

A large-scale analysis of the image repository was carried out, and a total of 355 319 image files were analyzed, with a total size of 167 TB [6]. The author found that only 3% of the files in the image repository are unique, whereas other files are redundant file copies. The storage of image files is related to the storage structure of image files. Container's image files use another union file system (AUFS), which is a union FS that combines different directories into one directory to make a virtual file system. Two techniques are used by AUFS in file management, which are stack-level management and copy-on-write. Write-time replication uses sharing and replication techniques, and only one copy is kept for the same image file. All operations access this file. When an operation needs to modify or add the file, the operating system will copy this part of the data to a new place and then modify or add it. By comparison, other operations still access the original file, which saves the storage space of images and speeds up system startup time.

However, although the AUFS can help edge nodes to save some storage space, but DockerHub [7], which is the most popular registry, currently storing almost 4 million public image repositories comprising approximately 20 million layers. Even though edge nodes have stronger computing and storage capabilities relative to IoT devices, their computing and storage capacity is not comparable to that of the data center, which is not possible to store all image files on the

Authorized licensed use limited to: HUNAN UNIVERSITY. Downloaded on April 07,2023 at 13:52:09 UTC from IEEE Xplore. Restrictions apply.

edge node. Therefore, how to store image files of containers, so as to reduce users image download time as much as possible, is a key problem in ECBC. Owing to the complexity of the storage structure of image files, this problem is not a simple combinatorial optimization problem. The traditional combination optimization method is aimed at a single object, and no association exists between individuals. When considering how to optimize the placement of image files, the storage of image files also needs to consider the nesting relationship of the image files to maximize the utility of storage resources of edge nodes.

C. Contributions

This article aims to improve the resource utilization of edge nodes in smart factory by solving the storage problem of image files. The main contributions are listed as follows.

1) An image file usually needs to be backtracked up multiple levels to calculate its size. By decoupling the nested relationship between image nodes, we propose a novel computing model for image file storage in edge computing and verify the correctness of the computing model in a real scene. The computing model can simplify the calculation process of image file size.
2) Considering the limited storage capacity of edge nodes, we propose a two-stage optimization strategy of image files for edge nodes. First, in the initialization stage, we proposed a minimum transmission volume algorithm, which is a pseudo-polynomial time algorithm to resolve the optimal placement problem of image files at edge nodes, the algorithm makes full use of the nested structure of image files and reduces the downloaded amount of image files. Second, in the runtime stage, a runtime update mechanism of LIR is proposed. Edge nodes can adjust image files in LIR according to the number of downloads, further optimize the file combination of the image repository, and reduce the download time of image files.
3) We have created test data of image files and service request according to the real smart factory scenario. We selected the most commonly used 37 image files from DockerHub and analyzed the structure of them by Microbagger. Simulation experiments based on the data sets show that the two-stage optimization strategy can reduce transmission volume and download time by 20%. We publicize the data set for research and objective performance evaluation.

The remainder of this article is organized as follows. Section II summarizes the related work. II Section III introduces our system model. Section IV presents the optimal image storage strategy. Section V discusses the conducted experiments, the results of which verify the effectiveness of our proposal. Finally, Section VI concludes our work.

II. Related Work

There are many existing work on resource management and task scheduling of edge computing. Zeng et al. [8] considered a software-defined embedded system based on fog computing. They addressed task scheduling and resource management problems in fog computing. Mao et al. [9] developed online joint radio and computing resource management algorithms for multiuser MEC systems with the goal of minimizing long-term average weighting and power consumption of mobile devices and MEC servers, subject to task buffer stability constraints. In addition, Mao et al. [10] studied a green MEC system with energy harvesting equipment, and an effective computational offloading strategy was designed. A low-complexity online algorithm, which was called dynamic computing unloading algorithm based on Lyapunov optimization, was proposed. Xiao et al. [11] proposed a framework called CAME to enhance the computing power of the system, and they designed workload scheduling to balance the tradeoff between system delay and cost. Xiao and Krunz [12] proposed a new collaboration strategy called offload forwarding. In this strategy, each fog node did not always rely on the cloud data center to handle the unprocessed workload; it can also handle some or all of the unprocessed workload.

For the Docker container, a service placement algorithm based on the location of the image file was proposed to maximize the number of user services in [13]. Given that the image file can be shared by multiple services, it can be placed according to the location of the image file when the service was placed, thereby reducing download time and improving storage efficiency. However, the article did not consider that the optimized placement of the image file has been dynamically updated. To address the problem of task scheduling and resource management for smart manufacturing, a container-based task scheduling algorithm was proposed to reduce service delay in [14]. In addition, the authors proposed a resource redistribution algorithm that can dynamically update the resource quota of the current task according to the task number of the current node and improve the node utilization rate. However, this article was aimed at the scheduling algorithm of a single edge node and did not consider the task scheduling of multinode cooperation.

The container as a highly efficient and lightweight virtualization technology has been studied by a large number of scholars [15], [16], [17]. For the resource management of containers, Bhimani et al. [15] developed a new docker controller for scheduling containers of different types of applications. The controller determined the best batch of containers running at the same time to minimize the total execution time and maximize the utilization of resources. However, this article did not consider the characteristics of container image files on edge nodes.

Xie et al. [18] proposed a hybrid model by combination ARIMA and triple exponential smoothing. It can accurately predict the linear and nonlinear relationships in the container resource load series. In order to handle the dynamic docker container resource load, the weight values of two single models in the hybrid model were selected according to the sum of squares of their prediction errors over a period of time. The author also designed and implemented a real-time prediction system, which included collecting, storing, and predicting docker container resource load data, meanwhile scheduling

Authorized licensed use limited to: HUNAN UNIVERSITY. Downloaded on April 07,2023 at 13:52:09 UTC from IEEE Xplore. Restrictions apply.

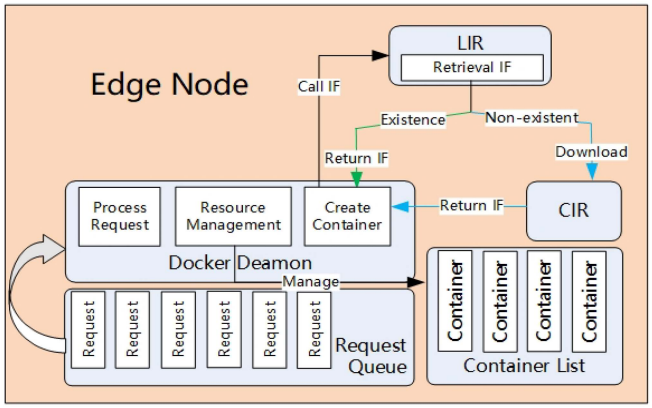

Fig. 2. System model.

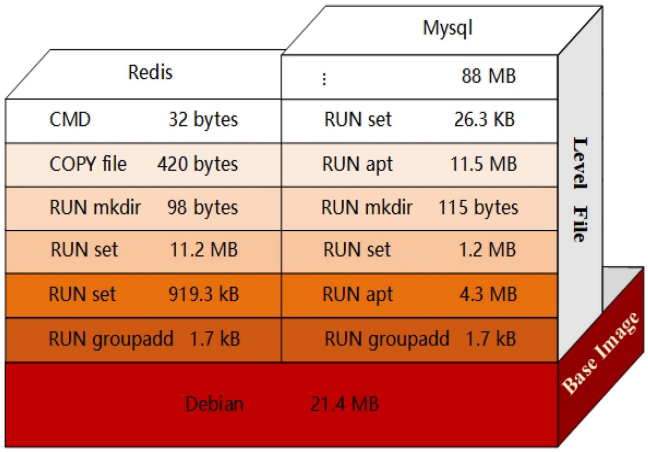

Fig. 3. Image file structures of Redis and MySQL.

and optimizing the usage of CPU and memory resources based on the predicted value.

Abdullah et al. [19] presented a new method to automatically allocate the best CPU resources to the container based on deep learning. The author utilized the diminishing returns to determine the optimal number of CPU pins of the container, thus maximizing the number of concurrent jobs and obtaining maximum performance. However, the author did not consider the impact of image files on containers.

Despite several studies on container resource management, knowledge on image placement and service migration is insufficient. Thus, this article proposes an optimized placement strategy for the placement of image files in Docker containers in edge computing. The strategy improves edge node utilization and reduces service delays between terminal devices and edge nodes.

III. System Model

Fig. 2 shows the service model of a single edge node. Multiple IoT devices send service requests to the edge node, and the edge node creates a corresponding container after accepting a request. The Docker, which is an application container engine, runs in the edge node. Docker Daemon is a process of Docker container, which receives requests from a client on the server and is responsible for container creation and resource management. When Docker Daemon accepts a request, it retrieves the LIR first. If the corresponding image file is not found in the LIR, then the edge node needs to access the CIR to download the image file. The CIR is similar to the Gits code repository for storing and managing image files uploaded by developers. Currently, Docker's official repository is DockerHub [7] at present, which stores more than 1 000 000 of image files. In our system model, there are two image sets, the CIR I^C and the LIR I^L, respectively. LIR is a subset of CIR, that is, $I^L \subseteq I^C$.

A. Models of Image and LIR

In Dockers image storage mechanism, each image file consists of multiple image layers. The layers are stacked from the bottom to the top to form the root file system of the container. Dockers storage driver is used to manage these image layers and provides a single file system to the outside. The size of a single image file is the cumulative sum of the file size of each layer. Other images can be created based on an image, and their own files can be added. Fig. 3 shows the level file structure of Redis and MySQL. The total size of Redis is 33.6 MB. The Debian, which is bottom-level, is the base image of Redis and MySQL. The total size of Debian is 21.4 MB.

A single image file consists of three parts: 1) *bootfs*; 2) *base_image*; and 3) *level_file*. The size of an image file S_i can be calculated as

$$S_i = bootfs + base\_image_i + level\_file_i \tag{1}$$

where *bootfs* is the lowest boot file system of the Docker image, including the bootloader and the operating system kernel. Given that *bootfs* takes up a small amount of space, it can be ignored in the calculation. The *base_image* is the base image of the image i. An image can only have one base image but it can be used as a base image by multiple images. The *level_file* denotes the level files of the image i. The level files of images cannot be stored separately, and it must be stored as a complete image file. When calculating the storage space of an image file, we use the base image and the file size other than itself to calculate the size of the image file. This approach simplifies the complexity of the model. The size S_i of the ith image can be calculated as

$$S_i = \begin{cases} L_i, & \text{if the img}_i \text{ without } base\_image \\ S_j + L_i, & \text{if the img}_i \text{ based on img}_j \end{cases} \tag{2}$$

where L_i represents the size of the level files of the image i. We can use the command "docker history ⟨ImageID⟩" to view the content and corresponding size of each layer of an image.

The LIR is used to store image files. Owing to the stack layer management of AUFS, accumulating the storage space of all image files is not a simple task when calculating the size of the LIR. We test the LIR of the Docker container. The test images in the LIR contain several base images and application images, and some test images are created by ourselves. The test images are composed of the five image files generated by Ubuntu 16.04 and Ubuntu 14.04 and combined with a test.zip

Authorized licensed use limited to: HUNAN UNIVERSITY. Downloaded on April 07,2023 at 13:52:09 UTC from IEEE Xplore. Restrictions apply.

TABLE II
INSTANCES OF IMAGE FILES

Image Name	Content	File Size	Actual Space Used
Ubuntu 14.04	Ubuntu 14.04	188 MB	188 MB
Ubuntu 16.04	Ubuntu 16.04	84 MB	84 MB
Test	Ubuntu 14.04 + /test.zip	398 MB	210 MB
Test2	Ubuntu 16.04 + /test.zip	294 MB	210 MB
Test3	Ubuntu 16.04 + /test.zip + /test.zip	294 MB	0 MB
Test4	Ubuntu 16.04 + /test.zip + /yin/test.zip	504 MB	210 MB
Test5	Ubuntu 14.04 + /test.zip + /yin/test.zip	608 MB	210 MB
Test6	Ubuntu 14.04 + /yin/test.zip	398 MB	210 MB

file. The image names are Test, Test 2, Test 3, Test 4, Test 5, and Test 6. The size of test.zip is 210 MB. Table II shows the structure and storage space of each image file.

As shown in Table II, regardless of whether the base image is local or not, the size of a single image file in the image repository is actually the size of its layer file. A single image file in the LIR exists as a layer file. The nested structure of these layer files is stored in an XML file. Docker reads the XML file and then forms the entire image when creating the container. When calculating the storage space of an image file, the storage space is usually the layer file of the image file. However, using the layer file to calculate the storage space of the image repository is difficult to associate with the image. Therefore, we build the image file into a tree structure, and the value of each node is the actual occupied space of the image file. Note that S_i in (2) only indicates the size of the image file and cannot represent the actual occupied storage space of the image file i. The image repository of the edge node is a forest that is composed of multiple trees. When calculating the storage space of the image repository, it iterates the size of all nodes.

The LIR consists of N image files, denoted by $I^L = \{\text{img}_1, \text{img}_2, \text{img}_3, \ldots, \text{img}_n\}$. We denote the storage space of LIR of the edge node by C. The storage space of the image repository in the edge node can be calculated as

$$C = \sum_{i=1}^{N} S_i - \sum_{i=1}^{N} Ch_i \times S_i = \sum_{i=1}^{N} L_i \tag{3}$$

where Ch_i represents the number of all child nodes of the node, including the child nodes of the child nodes.

B. Transmission Model

As shown in Fig. 2, when a user needs the service from an edge node, it first sends a request to the edge node. The content of the request includes the service data, delay constraints, and the image file needed by this service. After receiving the request, the edge node retrieves its LIR. If the requested image file is stored locally, then it immediately creates a container. If the requested image file is not stored locally, the edge node needs to download the image file from the CIR. When downloading the image file, the request needs to be suspended to wait for the completion of the download. After the image file is downloaded, the container is created by Docker Daemon. The storage space of the LIR is allocated by the edge node according to its own storage resources.

In this article, we conducted an extensive experiment to build a container image transmission model. Ubuntu 14.04 was not stored in the LIR at the beginning. When the user needs to create the test container in which its image file is based on Ubuntu 14.04, the LIR needs to download the basic image of Ubuntu 14.04. After the download is complete, Docker Daemon only starts to create the container. The transmission size of a single image file is not equal to the size of the image file. We assume that a service request corresponds to an image file, then the size of the image file to be downloaded for a request is the transmission size. The transmission size of image i can be calculated by the following:

$$\widetilde{S}_i = \begin{cases} 0, & \text{if the } \textit{base\_image} \text{ and } \textit{level\_files} \text{ are exist in the repository;} \\ L_i, & \text{if the } \textit{base\_image} \text{ is exist in the repository} \\ \widetilde{S}_j, & \text{if } \textit{the level\_files} \text{ are exist in the repository} \\ \widetilde{S}_j + L_i, & \text{if the } \textit{level\_files} \text{ and } \textit{base\_image are} \text{ not exist in the repository.} \end{cases} \tag{4}$$

Correspondingly, the transmission time of a service request can be calculated as follows:

$$T(i) = \frac{\widetilde{S}_i}{BW} \tag{5}$$

where BW represents the bandwidth allocated by the edge node. Generally, the bandwidth between the edge node and CIR is constant, therefore minimizing the transmission time is to minimize the transmission size of an image file.

IV. PROBLEM FORMULATION AND ALGORITHMS

Containers are different from virtual machines. Virtual machines can be created directly without operating system images, whereas Docker containers must be created on the basis of image files. If no image files exist, then Docker Daemon cannot create containers. Edge nodes have limited storage resources which cannot store all the image files. This limitation will cause IoT devices to wait for the image to download when accessing the local image, which will cause service delay. Thus, the purpose of this article is to maximize the utilization of storage resources at the edge node and optimize the image storage solution. To reduce the transmission delay of an image file, we consider reducing service completion time and improving the quality of service. We divide the optimization storage problem of an image file into two stages for optimization. The first stage is the initial placement stage of image files. The second stage is the dynamic update stage at run time.

A. Initial Placement Stage

In the initial placement stage, the optimal image file placement combination is selected according to the storage capacity of the edge node, thereby reducing the download time of the image file. Generally, we hope that many image files can be

Authorized licensed use limited to: HUNAN UNIVERSITY. Downloaded on April 07,2023 at 13:52:09 UTC from IEEE Xplore. Restrictions apply.

stored in the LIR as much as possible, which can reduce download time substantially. By selecting the optimal combination, the most commonly used image will be stored on the edge node, which can effectively reduce the number of image file downloads, thereby reducing the average download time in the entire process. However, owing to the absence of a download record in the initial placement stage, the frequency of use of an image file cannot be obtained. Therefore, in this stage, we assume that the access probability of all image files is the same. Therefore, the higher the number of child nodes, the higher the probability that the image is downloaded. If the image file is downloaded in the LIR, then its parent node must also be downloaded.

The download probability $P(i)$ of the image i can be calculated as

$$P(i) = \frac{Ch_i + 1}{\sum_{i \in I^C} Ch_i + 1}. \tag{6}$$

However, when selecting image file placement, the edge node cannot always choose the basic image to store, it needs to take the size of the node itself and the number of child nodes into account comprehensively. Therefore, we define a scalar quantity called the expectation of transmission volume of a service request. The expectation of transmission volume of a service request represents the expectation amount of data that needs to be transferred from the CIR to the LIR when the edge node completes a request. The expectation of transmission volume $\mathbb{E}(R)$ of a service request R can be calculated by the following:

$$\mathbb{E}(R) = \sum_{i \in I^C} P(i) \times \widetilde{S}_i. \tag{7}$$

Owing to the limited storage space of the node, the edge node may be overwritten by the newly downloaded image file. Therefore, the worst case needs to be considered in the initial placement stage, which means that the image file will be redownloaded each time when the image file is requested. As shown in (4), the smaller the transmission size, the shorter the average user waiting time.

In the initial placement stage, the optimization goal is to minimize the expectation of transmission volume of service requests. Let x_i be a binary variable to indicate whether the ith image file is selected and put into the LIR, that is

$$x_i = \begin{cases} 0, & \text{if } \text{img}_i \in I^L \\ 1, & \text{if } \text{img}_i \notin I^L. \end{cases} \tag{8}$$

The goal function can be expressed as follows:

$$\begin{aligned} P1: \min \ & \mathbb{E}(R) \\ \text{s.t.} \ & \sum_{i \in I^C} x_i \times L_i \le C. \end{aligned} \tag{9}$$

We can see from the optimization objective function that $P(i)$ is the download probability of each image in the CIR, which is generally a fixed value. $\widetilde{S}_i$ is the placement combination of image files. This problem is a bin packing problem, which means that it is an NP-hard problem. To find the optimal placement combination of the image file, we propose an algorithm called image placement based on transmission volume, which is a pseudo-polynomial time algorithm to solve the problem. Given the nested structure between the image files in the container image placement problem, when placed in an image file, the operation will affect the value of other image files which is based on this image file. Moreover, it causes the image placement problem without optimal substructure.

To provide the optimal substructure characteristics for this problem, we define the transmission function of an image file. The nested structure of the image files is decoupled so as to achieve the purpose of decoupling between image files. The transmission function can be expressed as

$$\text{Tran}(i) = (Ch_i + 1) \times L_i. \tag{10}$$

The transmission function takes into account the impact of all child nodes. With $Tran(i)$, whether the child node is placed will not affect the value of its parent node. Therefore, we can minimize the objective function by optimizing $Tran(i)$. The optimization objective function of this problem can be transformed into the following:

$$\begin{aligned} P2: \min \ & \sum_{i \in I^C} x_i \times \text{Tran}(i) \\ \text{s.t.} \ & \sum_{i \in I^L} x_i \times L_i \le C. \end{aligned} \tag{11}$$

To find the optimal substructure, let $\text{Tran}(i)$ be the transmission volume of the ith image. When the node storage capacity remains j, the all transmission volume corresponding to the best combination of the first i images is $V(i, j)$. At the beginning of the calculation, $V(i, j)$ is the total transmission volume when all images are stored. At this point, two cases need to be considered.

1) The remaining capacity of the LIR is smaller than the image and cannot be stored. The total transmission volume at this time is the same as the transmission volume of the first $i - 1$, that is

$$V(i, j) = V(i - 1, j). \tag{12}$$

2) The capacity to store the image is not enough, but it does not necessarily reach the current optimal value when installed. Thus, the best strategy between storage and nonstorage must be selected, that is

$$V(i, j) = \min\{V(i - 1, j), V(i - 1, j - L_i) - \text{Tran}(i)\} \tag{13}$$

where $V(i-1, j)$ indicates that the transmission volume of image i is not stored, $V(i-1, j-L_i)-\text{Tran}(i)$ indicates that the ith image is stored, the LIR capacity is reduced by L_i, but the transmission volume is reduced by $\text{Tran}(i)$. Therefore, its recursive expression is as follows:

$$V(i, j) = \min\{V(i - 1, j), V(i - 1, j - L_i - \text{Tran}(i))\}. \tag{14}$$

The pseudocode of the placement algorithm in the initialization stage is shown as follows. Obviously, the time complexity of the initialization stage placement algorithm is $O(imageList.size \times C)$.

Authorized licensed use limited to: HUNAN UNIVERSITY. Downloaded on April 07,2023 at 13:52:09 UTC from IEEE Xplore. Restrictions apply.

Algorithm 1 Initialization Stage Placement Algorithm

Input: *imageList*, C;
Output: *optImageList*;

```
Initialize totalTransw[], v[], V[][];
Calculate the total transmission volume by using Eq. (3);
Set totalTrans ← the value calculated by Eq. (3);
for k = C to 0 do
    V[0][k] ← totalTrans;
end for
for i = 0 to imageList.size do
    V[i][0] ← totalTrans;
end for
for k = C to 0 do
    for i = 1 to imageList.size do
        if k ≤ w[i] then
            V[i][k] ← V[i − 1][k];
        else
            V[i][k] ← min{V[i − 1][k], V[i − 1][k − w[i] − v[i])};
        end if
    end for
end for

Use V[][] to find the best image combination optImageList;
Return optImageList;
```

B. Runtime Optimization Stage

Owing to the diversity of users served by edge nodes, for popular application services, the use frequency of the image is bound to be higher. As a result, the placement frequency of image files in the node image repository will also change. The edge node needs to periodically analyze the access to the previous period and update the image repository in real time, thereby further reducing the download time of the image file.

In the runtime stage, the edge node periodically updates the image file of the LIR according to the user request received by the node and stores the commonly used image files locally. For the optimization of the LIR in the runtime stage, designing and updating its trigger mechanism first is necessary. In the runtime stage, the edge node will be able to obtain the request habits of the user of the node so that the usage of the image file can be analyzed. Therefore, the edge node needs to calculate the download volume of the current period after completing a certain number of service requests, which is

$$\text{down}_r = \sum_{i=1}^{I'} \tilde{S}_i \times \left(f_i + \sum_{f_k \in CH_i} f_k \right) \tag{15}$$

where I' represents the image collection that has been downloaded and is not stored locally. r is the current round number, the number of child nodes of image i, and CH_i represents the collection of child nodes of image i, f_i represents the number of downloads of image i in this round. Given that the base image is also downloaded when the image is downloaded, counting the access times of all its child nodes is necessary when calculating the transmission volume.

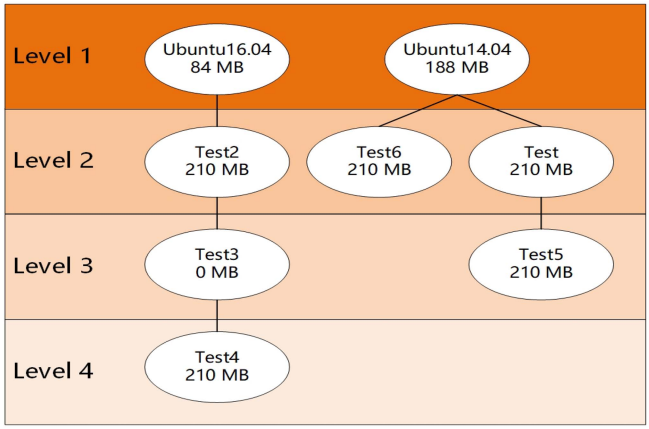

Fig. 4. Forest of the image repository.

After obtaining the download volume of the current round, the edge node compares the download volume of the current round with the download volume of the previous round. If the download volume of the previous round is less than the download volume of the current round, that is $\text{down}_{r-1} < \text{down}_r$, then the LIR update is performed; otherwise no update is made. When performing the update operation, the LIR is updated while the edge node is providing services. If the LIR needs to download a large number of image files, then it will cause an excessive network burden. Thus, the LIR needs to consider the download volume caused by updating.

After the update mechanism is triggered, the LIR needs to select the deleted image and the newly added image file. When updating the image file in the LIR, the LIR not only saves the image file blindly with the highest download number; it also needs to consider the cost of saving the image. Fig. 4 shows the Ubuntu 16.04 subtree as an example. Assume that the number of downloads of node 1 is 2, the number of downloads of node 2 is 10, the number of downloads of node 3 is 3, and the number of downloads of node 4 is 5. The capacity of nodes is 250 MB, if storage node 1, the download volume is $10\times210+0\times3+5\times210 = 3150$, if storage node 2, the download volume is $(2+10+3+5)\times84+3\times0+5\times210 = 2730$. Obviously, node 2 should be selected. The download volume is less than the download volume by stored node 1. Therefore, for the image file replacement strategy at runtime, comprehensively considering the number of downloads and the amount of downloads are necessary to reduce the number of users download time.

When choosing to update the image, first, we calculate the download volume D_i^r of each image file in I' in the current round, that is, the download volume that may be reduced when storing the image

$$D_i^r = \tilde{S}_i \times \left(f_i + \sum_{f_k \in CH_i} f_k \right) \forall \text{i} \in I'. \tag{16}$$

When selecting the deleted image file, for minimizing the impact on other images after deleting the image, only the layer

Authorized licensed use limited to: HUNAN UNIVERSITY. Downloaded on April 07,2023 at 13:52:09 UTC from IEEE Xplore. Restrictions apply.

Algorithm 2 Runtime Update Algorithm

Input: *downedImageList*;
Output: *optImageList*;

Calculate the transfer volume of each image file in this round of request;
Generate two sets, one sets is a sets of images stored locally *imgLocal* and the other is a sets of images that have been downloaded but not stored locally *imgCloud*;
Initialize $optImageList = \{\}, il \leftarrow imgLocal.size, ic \leftarrow imgCloud.size$
Sort the two sets from large to small;
for $i = 0$ to il **do**
    for $j = ic$ to 0 **do**
        Replace the first image that is not stored locally with the last image collection locally;
        if The transmission volume after replacement is smaller than that of the previous round and can be put in **then**
            Update the transmission volume of the last round to the one after replacement;
        else
            Break;
        end if
    end for
end for
Output the LIR of the next round *optImageList*;
Return *optImageList*;

file of the image will be deleted, and the basic image will not be deleted. As for the image stored locally, calculating the number of downloads that may increase after deletion is necessary, that is

$$D_j^r = L_j \times \left(f_j + \sum_{f_k \in CH_i} f_k \right) \ \forall j \in I. \tag{17}$$

Given that only the layer file of the image file is deleted, the download times of other image files based on the image need to be counted. When choosing to replace, we also need to consider the replacement income. The replacement income refers to the reduction ratio of download volume to reduced storage capacity

$$\text{revenue}_{i,j}^r = \frac{D_i^r - D_j^r}{\widetilde{S}_i - L_j} \ \forall \text{i} \in I' \ \forall j \in I. \tag{18}$$

This means how many units of download volume will be reduced for each additional unit of storage capacity.

V. Simulation Experiments

In this section, we use the Java development simulation environment to test the proposed image file placement strategy. The data set used in the test comes from DockerHub, which is currently the world's largest image file library. This article makes a statistical analysis of the most commonly used image files in DockerHub, collects the 37 most commonly used image files, and uses MicroBadger [20] to analyze the

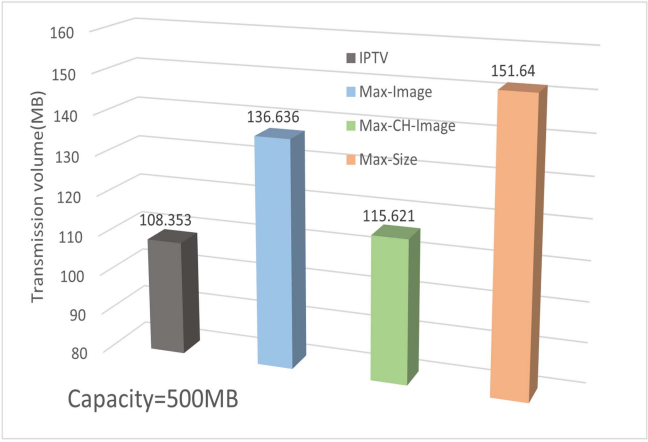

Fig. 5. Transmission volume during initialization with 500 MB of Data 1.

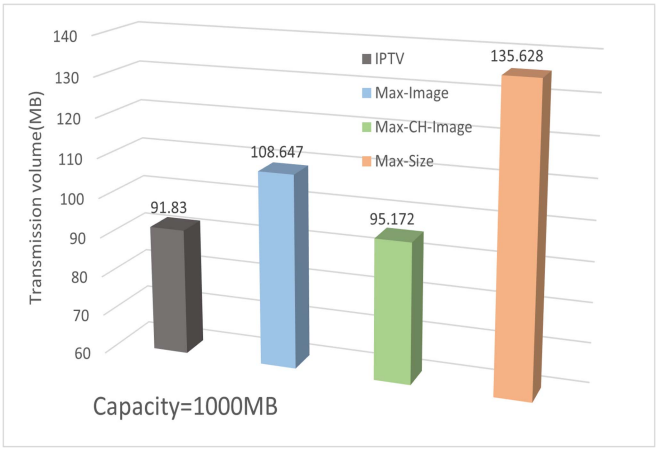

Fig. 6. Transmission volume during initialization with 1000 MB of Data 1.

image file hierarchy. Among them, 12 basic images and a maximum of four layers of embedded images set of structures are selected. In addition, this article uses two sets of user request data sets, including the generation of 1100 user requests. Data 1 allocated 1100 requests to an average of 37 images, whereas Data 2 allocated 1100 requests according to the real user data on DockerHub.

A. Initial Placement Stage

This article uses the image file transmission volume and the number of image file downloads to judge the performance of different placement strategies. In the initialization placement stage, the comparison of placement strategies is as follows.

1) *Max-Image:* Place as many images as possible.
2) *Max-CH-Image:* Select the images with the most child nodes to place.
3) *Max-Size:* Select the images with the largest storage space to place.

We first use data set Data 1 to compare the image transfer volume when the nodes LIR capacity is 500 and 1000 MB. Fig. 5 shows the experimental result of the LIR with a capacity of 500 MB, and Fig. 6 shows the experimental result of the LIR with a capacity of 1000 MB. As shown in the figure, whether the capacity is 500 or 1000 MB, the algorithm

Authorized licensed use limited to: HUNAN UNIVERSITY. Downloaded on April 07,2023 at 13:52:09 UTC from IEEE Xplore. Restrictions apply.

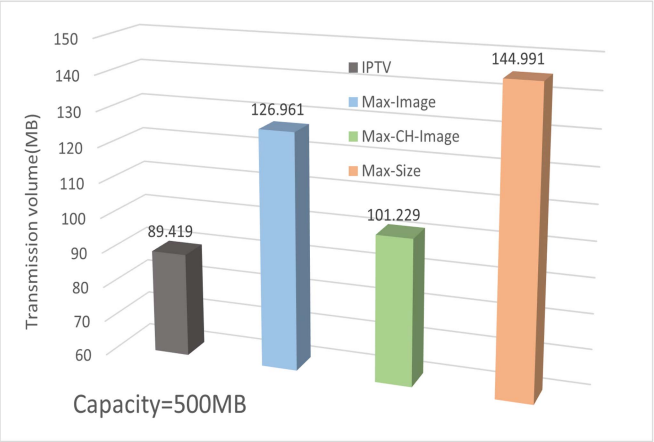

Fig. 7. Transmission volume during initialization with 500 MB of Data 2.

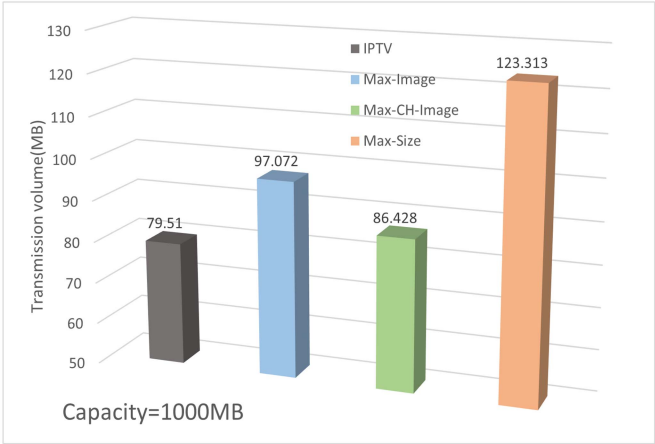

Fig. 8. Transmission volume during initialization with 1000 MB of Data 2.

proposed in this article can significantly reduce the transmission volume compared with other algorithms. Although the Max-Image strategy selects the most image file placement, it can reduce the download probability of the image file. However, most of the stored images are images with less resource overhead, and those images that take up more resources need to be downloaded. As a result, for some of the images, the base image is downloaded more times, leading to increased transmission volume.

The Max-CH-Image strategy has less transmission than Max-Image and Max-Size, because Max-CH-Image considers the nested structure of the image file and stores the basic image file with a large download probability locally to avoid too many repeated downloads. Thus, its transmission volume is better than that of Max-Image and Max-Size. Moreover, the nested structure is important for the placement of the image file, and the selection of the image file needs to be combined with its basic image for screening. However, the Max-CH-Image strategy does not consider that when the occupied space of the basic image is small, the transmission volume will not be increased too much. As a result, the transmission volume is greater than the strategy proposed in our strategy.

The Max-Size strategy has the largest transmission volume, because the Max-Size strategy only stores images with the largest resource consumption locally, that is, it does not consider that the download probability of the base image is greater than that of the nonbase image, nor does it consider how to reduce the number of downloads. This feature leads to an excessively high throughput.

Compared with the other three strategies, the proposed strategy has the lowest transmission volume. This is because the strategy proposed in this article looks for the optimal combination of transmission volume and fully considers the impact of the nested structure of the image file. This feature provides several advantages. First, the repeated downloading caused by not storing the basic image is avoided. Second, the strategy fully considers how high the basic image with less resource consumption will not affect the transmission volume.

Similarly, we use data set Data 2 to compare the amount of the transmission when the LIR capacity of the node is 500 and 1000 MB. The results are shown in Figs. 7 and 8. A comparison of the results of Data 1 and Data 2 shows that when the LIR capacity is 500 MB, the gaps between the strategies proposed in this article, Max-Image, Max-CH-Image, and Max-Size are 18.934, 9.675, 14.392, and 6.649 MB, respectively. When the LIR capacity is 1000 MB, the gaps between the strategies proposed in this article, Max-Image, Max-CH-Image, and Max-Size are 12.32, 11.575, 8.744, and 12.315 MB, respectively. After analysis, the transmission volume of Data 2 is smaller than the transmission volume of Data 1, because Data 2 is smaller than Data 1 in the total transmission volume. However, the gap between Data 1 and Data 2 is not large, because the optimal image placement combination is selected on the premise that the access probability of all image files is the same in the initialization stage. Moreover, whether the number of image requests is evenly distributed has little effect on the experimental results.

B. Runtime Stage

This section mainly compares the experimental results at the runtime stage. Given that the Max-Image, Max-CH-Image, and Max-Size strategies only select the image file to be placed during the initialization stage, the performance of the runtime stage is not affected. Therefore, no comparison is made at runtime, and the optimal placement strategy proposed in this article is used by default. This article takes the LIR capacity of 500 MB as an example and checks whether an update is required every 100 requests in the runtime stage.

Figs. 9 and 10 are the transmission volume and download times, respectively, generated by Data 1 in the runtime stage after each round of update of the requested data set. First, we compare the amount of transmission per round in Fig. 9. In the experimental results of Data 1, in the first round and the second round, the transmission volume of the two is the same, that is, 9.5 and 10.4 MB, respectively. In the next round, in rounds 3, 9, and 10, the update mechanism is slightly higher than the transfer volume by approximately 100–300 kB without the update mechanism. In the 4th, 5th, 6th, 7th, 8th, and 11th rounds, the update mechanism is lower than the transfer

Authorized licensed use limited to: HUNAN UNIVERSITY. Downloaded on April 07,2023 at 13:52:09 UTC from IEEE Xplore. Restrictions apply.

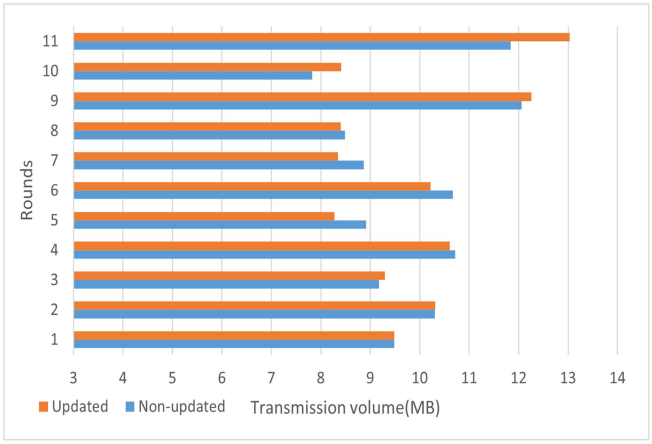

Fig. 9. Each round of Data 1 transmission volume during runtime.

Fig. 11. Each round of Data 2 transmission volume during runtime.

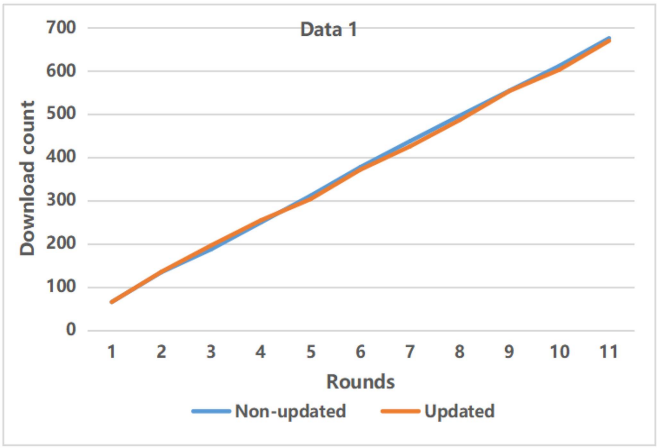

Fig. 10. Number of downloads per round of Data 1 at runtime.

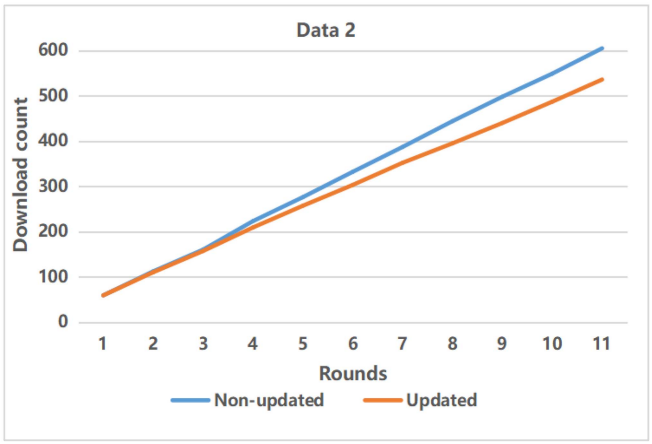

Fig. 12. Number of downloads per round of Data 2 at runtime.

volume by approximately 100–500 kB without the update mechanism.

Fig. 10 shows the total number of downloads per round in the runtime stage. According to the analysis of the results, the update mechanism can only slightly reduce the transmission volume and the number of downloads for the evenly distributed data set. This is due to the fact that in the placement algorithm in the initialization stage, the transmission volume of the image file is calculated according to the consistent access probability of all images. Therefore, in the runtime stage, when the number of requests for each image is the same, the update mechanism cannot bring about a large change. Thus, compared with the nonupdate mechanism, the transmission volume and the number of downloads are not much different.

The previous paragraph compares the transmission volume and download times of the evenly distributed data set Data 1 in the runtime stage. The experimental results show that the effect of the update mechanism is not significant when the number of requests for each image is the same. However, the experimental results of Data 2 show that the effect of the update mechanism is due to the nonupdate mechanism. Fig. 11 shows the transmission volume of Data 2 in each round. In the first round, the update mechanism is not triggered; thus, the transmission volume of the update mechanism and that of the nonupdate mechanism are 7.8 MB. In the second round, the difference between the two is approximately 400 kB. In the third round, the gap between the update mechanism and the nonupdate mechanism has gradually increased, and the maximum gap is approximately 1200 kB.

Fig. 12 shows the total number of downloads for each round of data set Data 2 during the runtime stage. According to the result analysis, when using DockerHubs request data set, the transmission volume difference in the previous update round is not large. However, in the later rounds, the update mechanism can greatly reduce the required transmission volume and download times with continuous optimization. This is because when the users request reaches a certain amount, the update mechanism can well identify the user habits of the node. Storing frequently used image files in LIR can avoid many unnecessary downloads efficiently. Edge nodes without the update mechanism increase the average transmission volume by approximately 20% per round. The result of this experiment proves that the dynamic optimization strategy proposed in the runtime stage can effectively reduce the transmission volume of image files.

VI. Conclusion

This article deeply analyzes the container image file storage mechanism and constructs the image file storage model

Authorized licensed use limited to: HUNAN UNIVERSITY. Downloaded on April 07,2023 at 13:52:09 UTC from IEEE Xplore. Restrictions apply.

on the basis of edge computing. According to the hierarchical structure of the image file, a two-stage storage strategy of the image file is designed. In the initialization stage, the optimal image file combination that meets the storage space of the node is selected. In the runtime stage, the file combination of the local image repository is constantly adjusted. The simulation results show that the two-stage storage strategy of image file proposed in this article can make efficient use of the storage space of nodes and improve the utilization of storage resources. In the runtime stage, the update mechanism can continuously optimize the image storage strategy according to the service characteristics of different nodes and user habits, so as to reduce the download delay of image files.

However, we only optimize the image file storage for a single edge node in this article, which has some limitations. It is usually stored collaboratively by multiple nodes in real scenarios, and this approach can better improve the storage efficiency of image files. In the future, we will optimize the collaborative storage of multiple edge nodes using multiintelligent reinforcement learning methods. Moreover, we will further optimize the image file storage updated strategy in the runtime stage.

REFERENCES

[1] *Cisco Global Cloud Index: Forecast and Methodology, 2020–2025*, Cisco, San Jose, CA, USA, 2020.

[2] "Mobile-edge computing-introductory technical white paper," ETSI, Sophia Antipolis, France, White Paper, 2012. [Online]. Available: https://portal.etsi.org/Portals/0/TBpages/MEC/Docs/Mobile-edge_Computing_Introductory_Technical_White_Paper_%2018-09-14.pdf

[3] B. I. Ismail et al., "Evaluation of Docker as edge computing platform," in *Proc. IEEE Conf. Open Syst. (ICOS)*, 2016, pp. 130–135.

[4] M. Sollfrank, F. Loch, S. Denteneer, and B. Vogel-Heuser, "Evaluating Docker for lightweight virtualization of distributed and time-sensitive applications in industrial automation," *IEEE Trans. Ind. Informat.*, vol. 17, no. 5, pp. 3566–3576, May 2021.

[5] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio, "An updated performance comparison of virtual machines and Linux containers," in *Proc. IEEE Int. Symp. Perform. Anal. Syst. Softw. (ISPASS)*, 2015, pp. 171–172.

[6] N. Zhao et al., "Large-scale analysis of Docker images and performance implications for container storage systems," *IEEE Trans. Parallel Distrib. Syst.*, vol. 32, no. 4, pp. 918–930, Apr. 2021.

[7] "DockerHub." Accessed: Feb. 10, 2021. [Online]. Available: https://hub.docker.com/

[8] D. Zeng, L. Gu, S. Guo, Z. Cheng, and S. Yu, "Joint optimization of task scheduling and image placement in fog computing supported software-defined embedded system," *IEEE Trans. Comput.*, vol. 65, no. 12, pp. 3702–3712, Dec. 2016.

[9] Y. Mao, J. Zhang, S. H. Song, and K. B. Letaief, "Stochastic joint radio and computational resource management for multi-user mobile-edge computing systems," *IEEE Trans. Wireless Commun.*, vol. 16, no. 9, pp. 5994–6009, Sep. 2017.

[10] Y. Mao, J. Zhang, and K. B. Letaief, "Dynamic computation offloading for mobile-edge computing with energy harvesting devices," *IEEE J. Sel. Areas Commun.*, vol. 34, no. 12, pp. 3590–3605, Dec. 2016.

[11] M. Xiao, S. Zhang, W. Li, P. Zhang, C. Lin, and X. S. Shen, "Cost-efficient workload scheduling in cloud assisted mobile edge computing," in *Proc. IEEE/ACM 25th Int. Symp. Qual. Serv. (IWQoS)*, 2017, pp. 1–10.

[12] Y. Xiao and M. Krunz, "QoE and power efficiency tradeoff for fog computing networks with fog node cooperation," in *Proc. IEEE Conf. Comput. Commun.(INFOCOM)*, 2017, pp. 1–9.

[13] P. Smet, B. Dhoedt, and P. Simoens, "Docker layer placement for on-demand provisioning of services on edge clouds," *IEEE Trans. Netw. Service Manage.*, vol. 15, no. 3, pp. 1161–1174, Sep. 2018.

[14] L. Yin, J. Luo, and H. Luo, "Tasks scheduling and resource allocation in fog computing based on containers for smart manufacturing," *IEEE Trans. Ind. Informat.*, vol. 14, no. 10, pp. 4712–4721, Oct. 2018.

[15] J. Bhimani et al., "Docker container scheduler for I/O intensive applications running on NVMe SSDs," *IEEE Trans. Multi-Scale Comput. Syst.*, vol. 4, no. 3, pp. 313–326, Jul.–Sep. 2018.

[16] C. Pahl, S. Helmer, L. Miori, J. Sanin, and B. Lee, "A container-based edge cloud PaaS architecture based on raspberry Pi clusters," in *Proc. 4th Int. Conf. Future Internet Things Cloud Workshops*, 2016, pp. 117–124.

[17] F. Ramalho and A. Neto, "Virtualization at the network edge: A performance comparison," in *Proc. IEEE 17th Int. Symp. World Wireless Mobile Multimedia Netw. (WoWMoM)*, 2016, pp. 1–6.

[18] Y. Xie et al., "Real-time prediction of Docker container resource load based on a hybrid model of ARIMA and triple exponential smoothing," *IEEE Trans. Cloud Comput.*, vol. 10, no. 2, pp. 1386–1401, Apr.–Jun. 2022.

[19] M. Abdullah, W. Iqbal, F. Bukhari, and A. Erradi, "Diminishing returns and deep learning for adaptive CPU resource allocation of containers," *IEEE Trans. Netw. Service Manag.*, vol. 17, no. 4, pp. 2052–2063, Dec. 2020.

[20] "MicroBadger." Accessed: Apr. 15, 2021. [Online]. Available: https://microbadger.com/

Luxiu Yin received the bachelor's degree in computer science and technology from Changsha University, Changsha, Hunan, China, in 2011, and the master's degree in computer software and theory from Hunan Normal University, Changsha, in 2015. He is currently pursuing the Ph.D. degree with the College of Information Science and Engineering, Hunan University, Changsha.

His research is focused on cloud computing, fog computing, and wireless virtual network.

Juan Luo (Member, IEEE) received the bachelor's degree in electronic engineering from the National University of Defense Technology, Changsha, Hunan, China, in 1997, and the master's and Ph.D. degrees in communication and information system from Wuhan University, Wuhan, Hubei, China, in 2000 and 2005, respectively.

From 2008 to 2009, she was a Visiting Scholar with the University of California at Irvine, Irvine, CA, USA. She is currently a Professor and a Doctoral Supervisor with the College of Computer Science and Electronic Engineering, Hunan University, Changsha. She has published more than 80 papers. Her research interests are focused on the Internet of Things, cloud computing, and middleware.

Prof. Luo is a member of ACM and SIGCOM. She is also a Distinguished Member of CCF.

Keqin Li (Fellow, IEEE) received the bachelor's and master's degrees in computer science from Tsinghua Unversity, Beijing, China, in 1985 and 1987, respectively, and the Ph.D. degree from the University of Houston, Houston, TX, USA, in 1990.

He is a SUNY Distinguished Professor of Computer Science with the State University of New York, New Paltz, NY, USA. He is also a National Distinguished Professor with Hunan University, Changsha, China. He has authored or coauthored over 870 journal articles, book chapters, and refereed conference papers. He holds nearly 70 patents announced or authorized by the Chinese National Intellectual Property Administration. His current research interests include cloud computing, fog computing and mobile-edge computing, energy-efficient computing and communication, embedded systems and cyber–physical systems, heterogeneous computing systems, big data computing, high-performance computing, CPU–GPU hybrid and cooperative computing, computer architectures and systems, computer networking, machine learning, and intelligent and soft computing.

Dr. Li has received several best paper awards. He is among the world's top 5 most influential scientists in parallel and distributed computing in terms of both single-year impact and career-long impact based on a composite indicator of Scopus citation database. He has chaired many international conferences. He is currently an Associate Editor of the *ACM Computing Surveys* and the *CCF Transactions on High Performance Computing*. He has served on the editorial boards of the IEEE TRANSACTIONS ON PARALLEL AND DISTRIBUTED SYSTEMS, the IEEE TRANSACTIONS ON COMPUTERS, the IEEE TRANSACTIONS ON CLOUD COMPUTING, the IEEE TRANSACTIONS ON SERVICES COMPUTING, and the IEEE TRANSACTIONS ON SUSTAINABLE COMPUTING. He is an AAIA Fellow. He is also a member of Academia Europaea (Academician of the Academy of Europe).

Authorized licensed use limited to: HUNAN UNIVERSITY. Downloaded on April 07,2023 at 13:52:09 UTC from IEEE Xplore. Restrictions apply.