Docker-Based Full Life-Cycle Experimental Cloud Platform Design and Implementation

Huilin Song[1(✉)] and Qilong Li[2]

[1] School of International Economics and Trade, Jiangxi University of Finance and Economics, Nanchang 330013, China
songhuilin@jxufe.edu.cn

[2] Department of Computer Science and Technology, Tongji University, ShangHai 201804, China

Abstract. The university laboratory that adopts the virtual machine method has problems such as the large maintenance workload of the experimental environment, complex configuration, fixed experimental time, and the inability to save the process. We propose a lightweight experimental environment based on Docker. Monitoring service, high availability session management, and resource protection mechanism of Docker runtime are designed. The full life cycle service runs after the Docker container is created and provides a management interface for the upper cloud platform. The prototype system of the cloud experimental platform is realized, which simplifies the construction and management of the cloud experimental platform in colleges and universities.

Keywords: Docker · Full life-cycle · Cloud Platform

1 Introduction

The traditional laboratory in colleges and universities has the defects of large infrastructure investment and high equipment loss rate. Due to the fixed location of the experiment, it is difficult for the instructor to grasp the experimental progress in the limited time, let alone to carry out the expansion of teaching. At present, most of the experimental platforms in colleges and universities are based on virtual machine technology, not only resource utilization but also can not carry out the full life-cycle management.

With the rapid development of computer networks [1–4], the emergence of cloud computing provides a new idea for the experimental platform of colleges and universities [5–7]. By reintegrating and encapsulating resources such as computer hardware, software, and computing capability, and providing them to students in the form of services, this new computing mode can be more flexible and make full use of computing resources. Based on lightweight virtualization technology, such as Docker [8]. Docker directly shares the kernel and hardware of the host, so the performance gap between the application running in the Docker container and the application running on the host is almost negligible. Unlike virtual machines, containers do not require hardware virtualization or run a complete operating system. This benefits higher resource utilization, a faster application running, and more efficient data storage.

Y. Cao and X. Shao (Eds.): MONAMI 2022, LNICST 474, pp. 185–195, 2023.
https://doi.org/10.1007/978-3-031-32443-7_13

In general, the cloud platform for experiments based on Docker should meet the following points: (1) Ability to deploy and distribute lab environments. (2) Can manage the full life cycle of containers. (3) Have a certain experimental preservation ability. However, the virtualization scheme provided by native Docker cannot meet the above requirements. Based on the above requirements and deficiencies, we first redesigned the Docker container from three aspects: monitoring service, high availability session management, and resource protection mechanism, providing a management interface for the upper cloud platform and realizing full life cycle management. Then the prototype system of the cloud experimental platform is designed and implemented, which simplifies the construction and management of the cloud platform.

2 Related Work

None of the three scheduling algorithms built into Docker's native resource scheduling management tool Swarm can give full play to the overall performance of Docker clusters, and many studies have improved Docker container scheduling strategies [9–11]. Kaewkasi et al. [12] proposed an algorithm based on ACO(Ant Colony Optimization, ACO), and experimental results show that ACO placed workload is about 15% better than greedy algorithm workload on the same host configuration. McDaniel et al. [13]. Proposed a two-layer approach to extend Docker and Docker Swarm so that both can monitor and control the I/O of Docker containers and improve resource utilization without being affected by competition. Zhang et al. [14]. Proposed an effective adaptive scheduler by modeling the scheduling problem as integer linear programming, which achieved significant cost savings. Wu et al. [15]. Developed an availability guarantee buffer layer priority scheduler, which could use the local buffer layer on nodes in Docker Swarm to reduce network traffic and speed up the start of service-related tasks. To ensure maximum utilization of system resources, idle containers are closed by recycling resources. To realize this process, it is necessary to monitor and manage the state of the container and to be able to save and restore the memory state of the container. Jimenez et al. implemented CoMA [16], a container monitoring agent for OS-level virtualization platforms.

However, the experimental platform has two special scenarios: (1) Different services are sensitive to different resource types. (2) A large number of concurrent requests may occur in a short time. Previous studies have not discussed this, so it is worth studying how to design corresponding algorithms for different scenarios and propose a multi-algorithm collaborative scheduling strategy to meet scheduling requirements in different scenarios by using different algorithms in turn to process requests.

3 Full Life-Cycle Management

3.1 Monitor Service

To obtain monitoring information on each node, it is necessary to require nodes to be able to transmit data to each other. Therefore, we adopt a centralized architecture, and the nodes in the cluster are divided into master nodes and slave nodes. The overall

monitor service is divided into a monitor agent, a monitor center, and the monitoring data processing logic on the server-side. Since the system uses the monitoring data in the way of active requests, and the demand of the request is real-time data, there is no need to store the data. The data that the monitoring service needs to collect is mainly divided into two categories: container-oriented monitor and host-oriented monitoring.

3.1.1 Monitor Agent

The monitor agent is deployed on each node in the cluster. As a data collector, the monitor agent collects various performance data of the node host, keeps monitoring status, and responds to data requests from the server. The structures used to record performance data are shown in Table 1.

Table 1. Structure for recording host information

Variable	Type	JSON
CoreNum	int	json:"core_number"
CpuRatio	float64	json:"cpu_ratio"
MemCap	float64	json:"memory_capacity"
MemUsage	float64	json:"memory_usage"
MemRatio	float64	json:"memory_ratio"
NetIO	float64	json:"net_io"
BlockIO	float64	json:"block_io"

The structure records basic host performance parameters: number of CPU cores, CPU usage, total memory, memory usage, network, and disk I/O throughput. In addition to basic configuration data, such as the number of CPU cores and memory capacity, other performance data must be collected and calculated in real-time. The CPU usage is calculated as follows:

$$CPU\,(N_i) = \left(\frac{us + sy}{us + sy + id}\right) \times 100\% \tag{1}$$

where, *us*, *sy,* and *id* respectively indicate the kernel state, user state, and idle id in Linux. The memory usage is read by the values of MemTotal and MemAvailable in */proc/meminfo*. Then the following formula is calculated:

$$Mem(N_i) = \frac{MemTotal - MemAvailable}{MemTotal} \times 100\% \tag{2}$$

The network load of the system in a certain period can be expressed as formula 3:

$$Net(N_i) = \frac{(ReceByte2 - ReceByte1) + (TransByte2 - TransByte1)}{(T2 - T1) \times Throughput} \tag{3}$$

3.1.2 Monitor Center

The main responsibility of the monitoring center is to collect the global image information of the cluster, the basic information of docker containers and the operation performance data of containers. The structure design for recording performance data is shown in Tables 2 and 3. The image structure records the basic parameters of the Docker image: image repository, tag, and ID.

Table 2. Image structure

Variable	Type	JSON
Repository	string	json:"image_repository"
ID	float64	json:"image_id"
Tag	float64	json:"tag"

In the container structure, the ID, Name, CPU, and memory capacity are all basic information after a container is generated. BlkIO indicates the average I/O speed of disk data, and NetIO indicates the average I/O speed of network data flows.

Table 3. Container structure

Variable	Type	JSON
ID	string	json:"container_id"
Name	string	json:"container_name"
CpuCap	float64	json:"cpu_capacity"
MemCap	float64	json:"memory_capacity"
Host	string	json:"host_name"
Cpu	float64	json:"cpu_usage"
Mem	float64	json:"memory_usage"
BlkIO	float64	json:"block_io"
NetIO	float64	json:"net_io"

3.2 Session Services

We use port mapping to expose the host computer to external users for access. However, Docker itself does not provide the function of managing ports. If the port number is not specified when creating containers, Docker will randomly allocate host ports, which is not conducive to management in a multi-host cluster environment. To this end, we designed a highly available session management service based on an *etcd* cluster.

3.2.1 *Etcd* Storage Directory

Etcd's cluster awareness can quickly respond to the addition and removal of nodes. Can provide high availability for the cluster, so that the cluster can cope with node failure and expansion problems. The directory structure of the storage system node is shown in Fig. 1. The cluster contains multiple node information and builds a Swarm, so the node information format is a string composed of IP and the port monitored by the Swarm, and the key and value are the same, and these key-value pairs store the Swarm cluster information required by the Swarm manager.

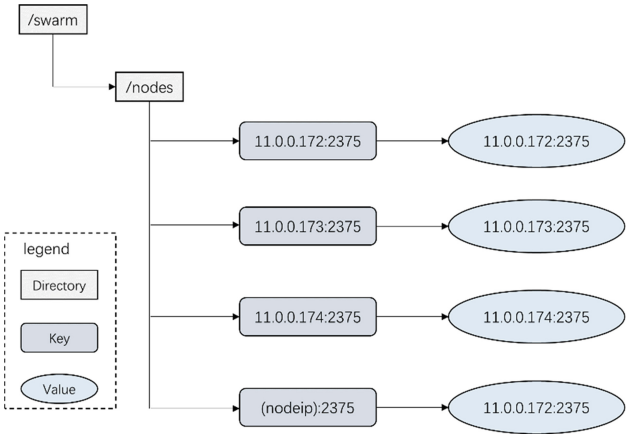

Fig. 1. Information directory structure of etcd cluster nodes

The container information used by users, including port information, container state, container access links, and so on, is maintained by the *etcd* instead of database storage, which improves the portability of the system. As shown in Fig. 2, container information for the entire cluster is stored in the *user* directory of the *etcd*. The *user* directory is a subdirectory named in the format of the username. To effectively utilize system resources, each user can use only one container at a time. Therefore, there is a one-to-one relationship between users and containers. Each subdirectory contains five key-value pairs that record details of containers created by that user. IP indicates the IP address of the host running the container. Port records the cluster port corresponding to the container. ConID records the container id. Status records the status of the container, including None, created, and saved. Url records the URL that accesses the container. The above data is subject to change frequently and can be kept up to date using *etcd's* key-value store.

3.2.2 Port List

The session structure for recording the port session is shown in Table 4. The variables in the session structure correspond to the port storage of *etcd*.

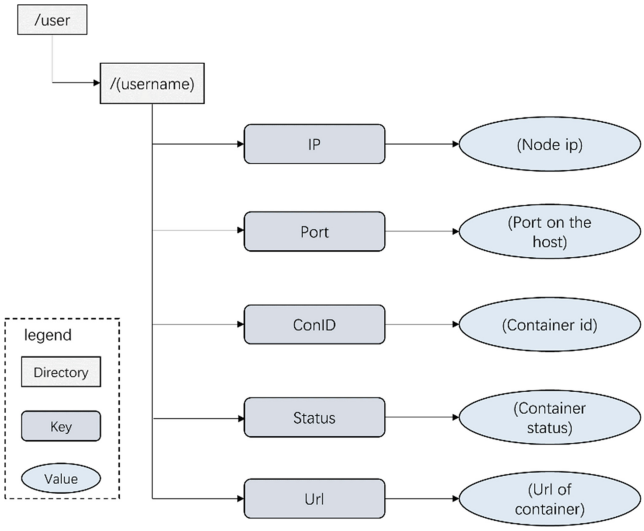

Fig. 2. The *etcd* container information directory structure

Table 4. Session structure

Variable	Type	JSON
IP	string	json:"host_ip"
Port	string	json:"container_port"
ConID	string	json:"container_id"
Status	string	json:"container_status"
Url	string	json:"container_url"

3.3 Resource Conservation

Although the resource utilization of the Docker container is much more efficient than that of the virtual machine, it is difficult to determine the demand for system resources of the application that needs to run for a long time when it is started in the Docker container, and some applications are transient. If the quota is allocated according to the maximum possible demand, when the application demand is in a slow period, Most of the resources allocated cannot be used efficiently. We designed a resource protection mechanism to obtain the container operating status information with the assistance of the monitoring system. The main functions are as follows: (1) If the container resource is tight, the resource quota is dynamically increased. (2) If the container is not in use for a long time, save the container on-site and recycle the container resources.

3.3.1 Dynamic Increases

The monitoring center designed in Sect. 3.1 can collect the basic information and performance data of Docker containers in the global scope of the cluster. The resource usage of a container can be determined based on the CPU and memory usage. If the resource usage of a container is high, it indicates that the container needs more resources to ensure normal running. In this case, the monitoring center triggers the resource protection mechanism to increase the resource quota for the container. In the case of wasted resources, the system does not do much because the container does not have a good prediction of whether it will need its allocated resources in the future.

When a container resource is scarce and the usage of a certain resource (memory or CPU) reaches 90%, the system increases the allocated resource quota by 20% to ensure the normal running of the container. For example, if the memory limit of container A is 200M and the monitoring center detects that the memory usage of container A is 92%, the resource protection mechanism is triggered and the docker update command is executed to reset the memory limit of container A to $(200 + 200 \times 20\%)$ M.

3.3.2 Storage of Containers

The monitoring center collects global container information every 15 min. When BlkIO and NetIO are both 0, it means that the user has not used the container for at least 15 min. Trigger the resource protection mechanism to save the container. When saving a container, you need to take into account what is being edited in the saving container. The storage algorithm of the container is shown in Table 5.

Table 5. EclipseSave.sh

```
1.  #!/bin/bash
2.  WID=`xdotool search --
    name "Eclipse Platform" | head -1`
3.  if [ -n "$WID" ]
4.  then
5.  xdotool windowactivate $WID
6.  xdotool windowfocus $WID
7.  xdotool key ctrl+S
8.  xdotool key --window $WID Return
9.  xdotool windowkill $WID
10. else
11. echo "eclipse is not running"
12. fi
```

4 Cloud Platform Implementation

4.1 System Structure

Based on the full life-cycle management mechanism, we designed the overall architecture of the system, as shown in Fig. 3. When a user initiates a request through the front-end page, the management center will route to different execution modules according to the corresponding API of the request, and these execution modules will complete the user's request. System users have two roles: teacher and student. As an administrator, teachers can view the basic information of Docker images and Docker containers and have certain management rights, such as setting experimental images. The operation of students is mainly the creation, saving, recovery, and destruction of the experimental environment (Docker container).

Specifically, the request to create the container is distributed to the scheduling module, which is responsible for scheduling the container in the cluster. When the container is created, the session management module allocates ports and records connection information for remote sessions. The monitoring module will constantly detect the health status of the container, and provide the host load information and container status information for the scheduling module and the runtime module. The runtime module is responsible for saving and restoring the container runtime. The data module is responsible for allocating storage directories to containers.

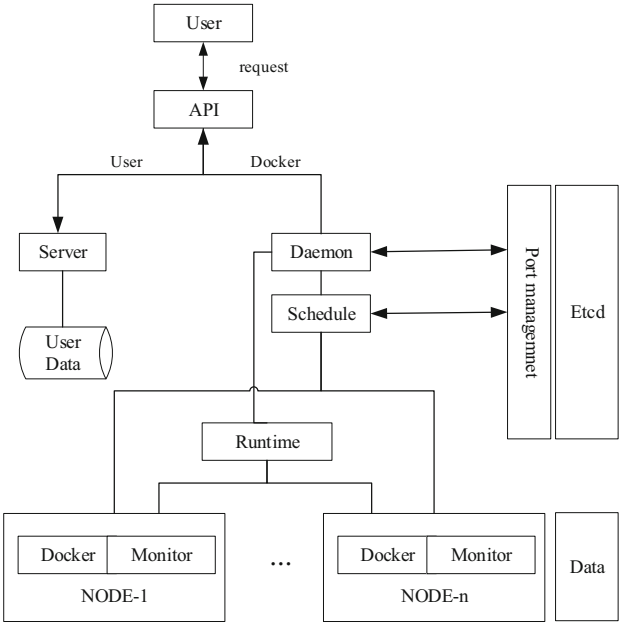

Fig. 3. Overall architecture of the system

4.2 Module Page

The experimental platform is B/S mode. Students create, save, restore, and destroy the web page. Teachers can view containers and set up experimental mirrors, including the status information of hosts in the cluster and the list of remotely connected ports (Fig. 4).

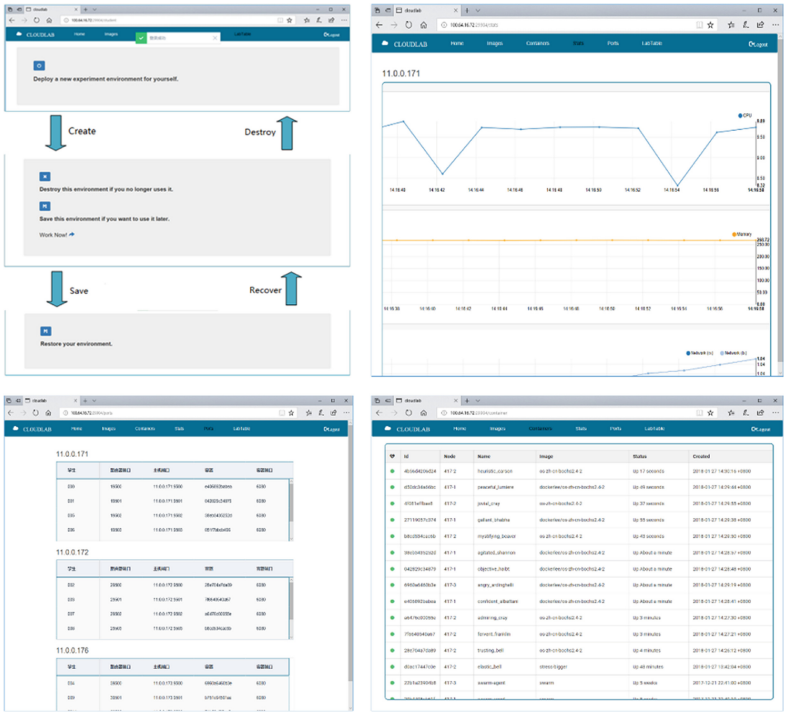

Fig. 4. Interface corresponding to experimental environment operation

5 Conclusion

In this paper, the full life-cycle management system of the cloud experimental platform is designed and implemented. First, the monitoring service is designed and implemented, the docker monitoring tool is analyzed and the monitoring agent and monitoring center are introduced in detail. Secondly, by analyzing the forwarding mechanism of the Docker network, the port management storage structure of *etcd* is designed, and a port list is maintained for each node in the cluster. Thirdly, the corresponding dynamic increment and container saving operations are described in two cases of resource shortage and container idle respectively. Fourthly, the prototype system of the cloud experimental platform is designed. In the future, we will study the customized Settings of different experimental environments combined with Docker images.

Funding. This study was supported by the Science and Technology Research Project of Jiangxi Provincial Department of Education (No. GJJ200318 and GJJ210520).

References

1. Sadeeq, M.M., Abdulkareem, N.M., Zeebaree, S.R.M., et al.: IoT and Cloud computing issues, challenges and opportunities: a review. Qubahan Acad. J. **1**(2), 1–7 (2021)
2. Huang, X., Yi, W., Wang, J., Xu, Z.: Hadoop-based medical image storage and access method for examination series. Math. Probl. Eng. **2021**, 1–10 (2021)
3. Cao, Y., Ji, R., Ji, L., Lei, G., Wang, H., Shao, X.: l2-MPTCP: a learning-driven latency-aware multipath transport scheme for industrial internet applications. IEEE Trans. Ind. Inform. **18**, 8456–8466 (2022)
4. Cao, Y., Ji, R., Huang, X., Lei, G., Shao, X., You, I.: Empirical mode decomposition-empowered network traffic anomaly detection for secure multipath TCP communications. Mobile Netw. Appl. **27**, 2254–2263 (2022)
5. Ali, M.B., Wood-Harper, T., Mohamad, M.: Benefits and challenges of cloud computing adoption and usage in higher education: a systematic literature review. Int. J. Enterp. Inform. Syst. **14**(4), 64–77 (2018)
6. Baldassarre, M.T., Caivano, D., Dimauro, G., et al.: Cloud computing for education: a systematic mapping study. IEEE Trans. Educ. **61**(3), 234–244 (2018)
7. Zhang, Z., Min, H.: Analysis on the construction of personalized physical education teaching system based on a cloud computing platform. Wireless Commun. Mobile Comput. **2020**, 1–8 (2020)
8. Merkel, D.: Docker: lightweight linux containers for consistent development and deployment. Linux J. **2014**(239), 2 (2014)
9. Marathe, N., Gandhi, A., Shah, J.M.: Docker swarm and kubernetes in cloud computing environment. In: 2019 3rd International Conference on Trends in Electronics and Informatics (ICOEI). IEEE, pp. 179–184 (2019)
10. Huang, C.H., Lee, C.R.: Enhancing the availability of Docker Swarm using checkpoint-and-restore. In: 2017 14th International Symposium on Pervasive Systems, Algorithms and Networks & 2017 11th International Conference on Frontier of Computer Science and Technology & 2017 Third International Symposium of Creative Computing (ISPAN-FCST-ISCC), pp. 357–362. IEEE (2017)
11. Magableh, B., Almiani, M.: A self healing microservices architecture: a case study in docker swarm cluster. In: Barolli, L., Takizawa, M., Xhafa, F., Enokido, T. (eds.) Advanced Information Networking and Applications: Proceedings of the 33rd International Conference on Advanced Information Networking and Applications (AINA-2019), pp. 846–858. Springer International Publishing, Cham (2020). https://doi.org/10.1007/978-3-030-15032-7_71
12. Kaewkasi, C., Chuenmuneewong, K.: Improvement of container scheduling for docker using ant colony optimization. In: 2017 9th international conference on knowledge and smart technology (KST), pp. 254–259. IEEE (2017)
13. McDaniel, S., Herbein, S., Taufer, M.: A two-tiered approach to I/O quality of service in docker containers. In: 2015 IEEE International Conference on Cluster Computing, pp. 490–491. IEEE (2015)
14. Zhang, D., Yan, B.H., Feng, Z., et al.: Container oriented job scheduling using linear programming model. In: 2017 3rd International Conference on Information Management (ICIM), pp. 174–180. IEEE (2017)
15. Wu, Y., Chen, H.: Abp scheduler: Speeding up service spread in docker swarm. In: 2017 IEEE International Symposium on Parallel and Distributed Processing with Applications

and 2017 IEEE International Conference on Ubiquitous Computing and Communications (ISPA/IUCC), pp. 691–698. IEEE (2017)
16. Jimenez, L.L., Simon, M.G., Schelén, O., et al.: CoMA: Resource monitoring of docker containers. In: International Conference on Cloud Computing and Services Science: 20/05/2015–22/05/2015, vol. 1, pp. 145–154. SCITEPRESS Digital Library (2015)
17. Cérin, C., Menouer, T., Saad, W., et al.: A new docker swarm scheduling strategy. In: 2017 IEEE 7th international symposium on cloud and service computing (SC2), pp. 112–117. IEEE (2017)