Journal of Systems Engineering and Electronics
Vol. 34, No. 4, August 2023, pp.955 – 965

Hound: a parallel image distribution system for cluster based on Docker

LIU Zijie[1,2], LI Junjiang[1,2], CHEN Can[2,3], and ZHANG Dengyin[1,2,*]

1. School of Internet of Things, Nanjing University of Posts and Telecommunications, Nanjing 210003, China; 2. Jiangsu Key Laboratory of Broadband Wireless Communication and Internet of Things, Nanjing University of Posts and Telecommunications, Nanjing 210003, China; 3. College of Telecommunications & Information Engineering, Nanjing University of Posts and Telecommunications, Nanjing 210003, China

Abstract: Current applications, consisting of multiple replicas, are packaged into lightweight containers with their execution dependencies. Considering the dominant impact of distribution efficiency of gigantic images on container startup (e.g., distributed deep learning application), the image "warm-up" technique which prefetches images of these replicas to destination nodes in the cluster is proposed. However, the current image "warm-up" technique solely focuses on identical image distribution, which fails to take effect when distributing different images to destination nodes. To address this problem, this paper proposes Hound, a simple but efficient cluster image distribution system based on Docker. To support diverse image distribution requests of cluster nodes, Hound additionally adopts node-level parallelism (i.e., downloading images to destination nodes in parallel) to further improve the efficiency of image distribution. The experimental results demonstrate Hound outperforms Docker, kubernetes container runtime interface (CRI-O), and Docker-compose in terms of image distribution performance when cluster nodes request different images. Moreover, the high scalability of Hound is evaluated in the scenario of ten nodes.

Keywords: container image, image distribution, parallelism, containerization.

DOI: 10.23919/JSEE.2023.000105

1. Introduction

With the advancement of lightweight container-based virtualization technologies like Docker [1–3], developers can package application which consists of multiple replicas into the container with their execution dependencies [4–6]. The identical image of these replicas requires to be fetched from the image registry, e.g., Docker Hub (https://hub.docker.com/) or the private registry, to destination nodes during the container startup progress. However, the distribution process of gigantic images (e.g., distributed deep learning application [7–9]) is time-consuming which will delay the container startup and further prolong the task makespan. Thus, the image "warm-up" technique which prefetches images to destination nodes is proposed instead of downloading images during the container startup process.

The current image "warm-up" technique focuses on identical image distribution. However, it fails to take effect when different images are requested by cluster nodes. For example, Docker presents Docker-compose (https://docs.docker.com/compose/) to provide multi-image parallel downloading on a single node. It can be easily migrated to the whole cluster by initiating the Docker command simultaneously.

To address this problem, we propose a simple but efficient cluster image distribution system based on Docker, named Hound. Our project is open source on GitHub at https://github.com/NJUPT- ISL/Hound. Our contributions are summarized as follows:

(i) We propose a novel image distribution mechanism, consisting of node classification and node-level parallelism (i.e., downloading images to destination nodes in parallel), to accelerate container startup.

(ii) We propose a simple but efficient cluster image distribution system based on Docker, named Hound. Hound adopts master-worker architecture which consists of Hound master and Hound workers to achieve parallel image distribution in the cluster view.

(iii) The experimental results demonstrate that Hound outperforms Docker, CRI-O (https://github.com/cri-o/cri-o),

---

Manuscript received October 15, 2021.
*Corresponding author.
This work was supported by the National Natural Science Foundation of China (61872423), Industry Prospective Primary Research & Development Plan of Jiangsu Province (BE2017111), the Scientific Research Foundation of the Higher Education Institutions of Jiangsu Province (19KJA180006), and the Postgraduate Research & Practice Innovation Program of Jiangsu Province (KYCX20_0764).

and Docker-compose in terms of image distribution performance when nodes in the cluster request different images. Moreover, the high scalability of Hound is evaluated in the scenario of ten Hound workers.

The remainder of this paper is organized as follows: Section 2 discusses the related works about efficient image distribution strategies. Section 3 describes the image distribution mechanism of Hound. Section 4 introduces the system design of Hound. In Section 5, we evaluate the image distribution performance and scalability of Hound, followed by discussions about our future work. The conclusions are drawn in Section 6.

2. Related works

A few efforts have been devoted to exploring efficient image distribution strategies in recent years. We divide them into three categories: (i) pruning image size based on various deduplication techniques (Subsection 2.1); (ii) modifying the original image distribution mechanism (Subsection 2.2); (iii) reducing the network bandwidth of the data center (Subsection 2.3).

2.1 Deduplication techniques

The evaluation in [10] shows that only 3% of the files are unique over 167 TB of uncompressed Docker Hub images, which demonstrates deduplication techniques have a great potential to reduce image size. With the assistance of various deduplication techniques, the amount of data transmitted between the registry and destination nodes is dramatically reduced, which further improves the efficiency of image distribution. Lee et al. proposed a deduplication system based on the peer-to-peer protocol for virtual machine (VM) images [11], which deduplicates images according to their similarity, to improve the performance of VM image distribution. Du et al. [12] presented a rapid Docker container deployment system based on shared network storage called Cider by minimizing the data transferred during the process of image download to accelerate the deployment process. Zhang et al. proposed a block-level deduplication system called block-level deduplication (BED) for container images [13], which deduplicates images in the registry according to the fingerprint list and downloads image blocks that are unavailable locally, to accelerate the image downloading process. However, the above methods achieve image deduplication by hash calculation and comparison in the fingerprint list, which is a time-consuming operation. To reduce the computation overhead during the deduplication process, Zhao et al. proposed a deduplication file system called Liquid for VM images, which only performs deduplication operations on modified image blocks [14]. Moreover, Saharan et al. proposed a VM deduplication system called QuickDedup which divides images into different block classes and only performs deduplication operations in each class to reduce the deduplication time [15].

The above deduplication techniques make attempt to improve image distribution efficiency by reducing the amount of transmitted data, which could be combined with Hound for further acceleration.

2.2 Modifications on the original distribution mechanism

These methods aim to complete the container startup process in advance by designing a custom image distribution mechanism. To speed up the startup of the container, Harter et al. [16] presented a brand-new Docker storage driver called Slacker which lazily retrievals image data that are required during the container startup progress. To accelerate container startup, Thalheim et al. proposed a lightweight container called CNTR which divides images into fat and thin images [6]. The thin image requires to be retrieved during the startup of CNTR, and the fat image is downloaded on demand. Civolani et al. proposed a container deployment system called FogDocker for the fog computing environment [17], which only fetches image data that is required during startup and then downloads remain data on demand, to speed up the container deployment. To reduce the network load during the image distribution process, Zheng et al. proposed a sharing-enabled file system called Wharf for Docker images which splits images into global and local states and shares image data with the global state in the cluster [18]. Chen et al. proposed a container deployment strategy that extracts different files between images by constructing a direct acyclic graph (DAG) model and transmits them to the destination to accelerate the container deployment [19]. To accelerate the container deployment, Ahmed et al. analyzed the container deployment process in the fog computing environment and presented several optimizations to make sufficient use of computing resources [20].

The above methods mainly focus on designing efficient image distribution mechanisms, which could be combined with Hound for further acceleration.

2.3 Network bandwidth concerned

These methods make full use of network bandwidth to improve image distribution efficiency. Peng et al. proposed an efficient cross-image distribution system called VM image distribution network (VDN) for VM based on the peer-to-peer network [21], which divides VM image into several chunks to save the available network bandwidth of the data center. To accelerate large-scale VM provision, Zhang et al. proposed a VM image distribution strategy named VMThunder which combines the lazy download mechanism with the peer-to-peer network [22]. Liang et al. proposed an image distribution system

called hybrid Docker image distribution (HDID) system for Docker images [23], which downloads image chunks from the Docker registry or other hosts that possess the required blocks according to the chunk size. Wang et al. presented a large-scale image distribution system based on the peer-to-peer network called faster image distribution (FID) to accelerate image deployment [24]. Nathan et al. proposed a co-operative management system called CoMICon which provides a distributed registry for sharing different Docker image layers in the cluster to speed up container deployment [25]. Unfortunately, the above sharing-enabled methods fail to set up private images. Moreover, the mechanism of sharing may affect the security of the entire system. More specifically, a Trojan image exists on all nodes once the underlying distributed storage system is invaded, which causes irreparable damage.

Although these methods make full use of network bandwidth to improve image distribution efficiency, however, performance degradation could occur when requested images are absent in cluster nodes.

3. Hound image distribution mechanism

The current image “warm-up” technique focuses on identical image distribution which fails to take effect when different images are requested. We illustrate in Fig. 1 that Docker-compose only accelerates image distribution when identical images are requested. However, it fails to support diverse image distribution demands in the cluster view. Consequently, there is a lack of a general image distribution strategy for image “warm up” in the cluster view. In Fig. 1(a), three images (i.e., Image A, B, and C) are required by each node before container startup. The identical image distribution request can be initiated simultaneously by Docker-compose using parallel flag. However, in Fig. 1(b), different images are required by each node, which causes Docker-compose to initiate particular image distribution request in order. Thus, it fails to take effect when different images are requested. To address this problem, this paper proposes Hound, a simple but efficient cluster image distribution system based on Docker.

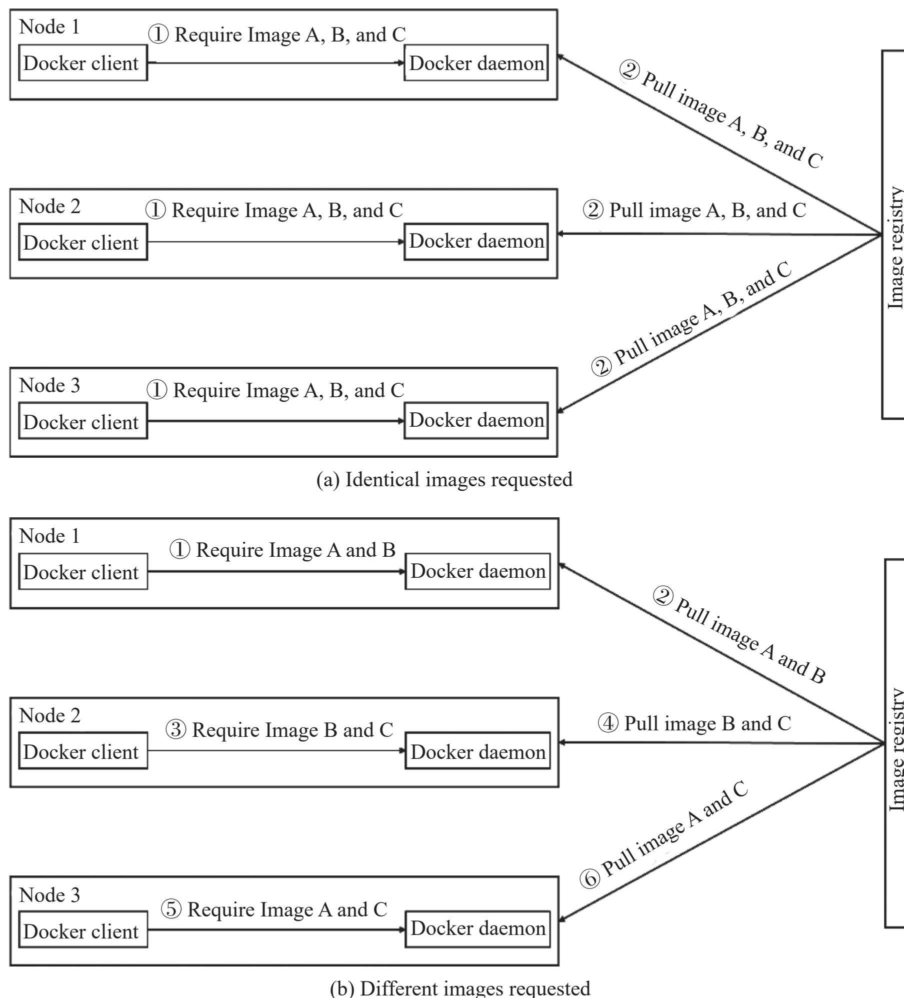

Fig. 1 Image distribution mechanism of Docker-compose in the cluster view

In this section, we introduce the image distribution mechanism of Hound which consists of two parts: (i) node classification based on requested images (Subsection 3.1), and (ii) node-level parallelism which downloads images in parallel in the cluster view (Subsection 3.2).

3.1 Node classification

Different from the existing image distribution strategies (e.g., Docker-compose) which cause performance degradation when different images are requested by cluster nodes, we propose a simple but efficient mechanism to classify cluster nodes with diverse image distribution requests into several groups, each of which requests identical images. Specifically, nodes with identical image demand are classified into a node group. Moreover, nodes within the node group possess the same node label which contains the requested image name.

For example, Table 1 shows the image demands requested by a cluster which consists of five nodes. Subsequently, Table 2 shows the classification result based on the image demands of each node. We can observe from Table 2 that the correspondence between images and requested nodes is established by the node classification mechanism. In Subsection 3.3, we combine the node classification mechanism with the node-level parallelism to implement the image distribution mechanism of Hound, which efficiently improves the image distribution efficiency in the cluster view.

Table 1 Image demands of the sample cluster which consists of five nodes

Node name	Requested image
Node 1	Image A, Image B, Image C
Node 2	Image B, Image D
Node 3	Image A, Image D
Node 4	Image B, Image C, Image D
Node 5	Image A, Image B, Image C, Image D

Table 2 Node classification result of the sample cluster

Node group (request image)	Node included	Node label
Group 1 (Image A)	Node 1, Node 3, Node 5	ImageName: Image A
Group 2 (Image B)	Node 1, Node 2, Node 4, Node 5	ImageName: Image B
Group 3 (Image C)	Node 1, Node 4, Node 5	ImageName: Image C
Group 4 (Image D)	Node 2, Node 3, Node 4, Node 5	ImageName: Image D

3.2 Node-level parallelism

We illustrate in Fig. 1 that only partial parallelism is achieved by migrating Docker-compose to cluster image distribution. To achieve complete parallelism for further improving image distribution efficiency, we propose a novel node-level parallelism which downloads images in parallel in the cluster view by means of multi-thread. Specifically, each thread corresponds to the image download request which is initiated to one of the destination nodes. We demonstrate in Subsection 5.2.1 that the node-level parallelism significantly improves the image distribution efficiency in the cluster view.

3.3 Overall mechanism

Algorithm 1 shows the image distribution mechanism of Hound which combines node classification and node-level parallelism. The number of enabled threads and image distribution information (i.e., requested images and nodes) are served as inputs. Hound first classifies nodes which request the identical image into a node group (Line 4−9). Then, Hound obtains the number of image distribution requests in all node groups (Line 11). To avoid unnecessary waste of computing resources, the number of enabled threads is automatically adjusted to the number of image distribution requests when the latter is less than the former (Line 13−14). Hound enables multiple threads to download images in parallel in the cluster view, each of which corresponds to the image download request initiated to one of the nodes in the node list (Line 16−22). Note that the upper limit number of enabled threads is equal to the value of workers which can be tuned flexibly according to user requests. A smaller value of workers refers to relatively weak parallelism which leads to performance degradation of image distribution. A large value of workers is not preferable either, because it will cause excessive resource consumption. Thus, a moderate value of workers is essential in our implementation.

Algorithm 1 Hound image distribution mechanism

Input: The number of threads permitted to be enabled workers, requested image list imageList, node list nodeList

Output: None

1 nodeGroup[][] = Null
2 $i = 0$
3 # Node classification
4 **for** image in imageList **do**
5   **for** node in nodeList **do**
6   **for** nodeImage in node.requestImages() **do**
7   **if** image == nodeImage **then**
8   add(nodeGroup[image], node)

```
9            break;
10   # Obtain the number of images requested in all node groups
11   pieces = len(nodeGroup)
12   # Determine the number of enabled threads
13   if pieces < workers then
14       workers = pieces
15   # Enable threads to download images
16   While i < workers do
17       object = pop(nodeGroup)
18       if object == Null then
19           break;
20       createThread(i, imagePull(object))
21       if i == workers − 1 && nodeGroup != Null then
22           i = 0
```

4. Hound design

We propose a cluster image distribution system called Hound based on Docker. Fig. 2 shows the master-worker architecture of Hound which consists of a Hound master and several Hound workers. The Hound master which plays the role of distributing images among all Hound workers is composed of two parts: (i) a node information database called NIDB which records the information of all Hound workers and (ii) an image manager called IM that forwards image distribution operations to Hound workers. The Hound worker responsible for performing image distribution operations is composed of two parts: (i) an image controller agent called ICA which coordinates with the Docker engine to complete the image distribution and (ii) a custom middleware for identity authentication during communication between the Hound master and Hound workers.

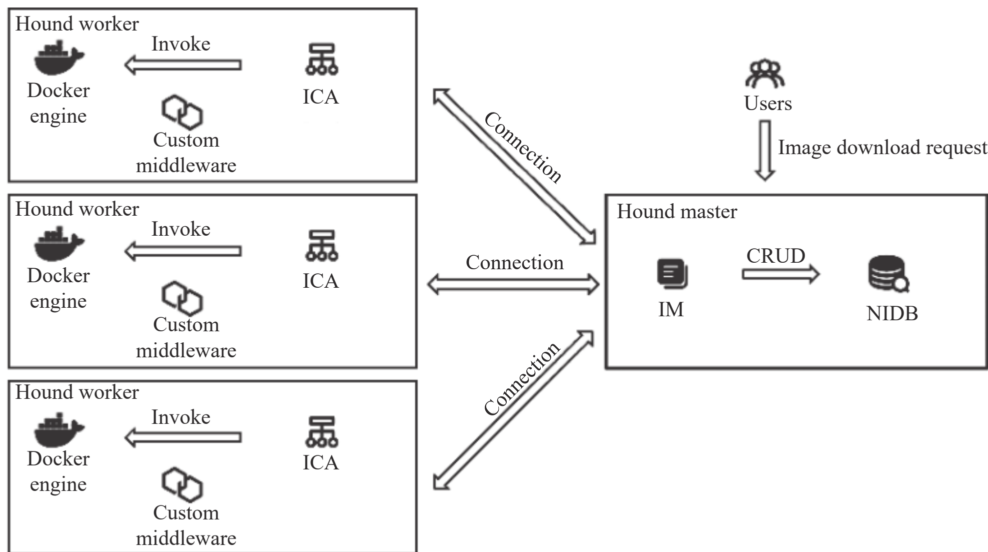

Fig. 2 Overview of Hound architecture

4.1 NIDB

The information of all Hound workers is stored in NIDB, which is shown in Table 3. Note that all node labels can be retrieved by IM to form the node group which requires performing the identical image distribution operation. To ensure secure communication between the Hound master and Hound workers, the token information is stored in NIDB for identity authentication which we will be mentioned in Subsection 4.4. In addition, execution logs of Hound are also stored in the NIDB for DevOps engineers to troubleshoot.

Table 3 Structure of information stored in NIDB

Label information	Node information	Token information
LabelName	NodeName	TokenContent
NodeList	KernelVersion	GenerateTime
–	OperatingSystem	UpdateTime
–	DockerVersion	–

4.2 IM

IM is a web server which captures image distribution requests sent by users. Nodes and their requested

images are first parsed by IM, followed by node classification to form the node group. Thus, the Hound master can forward image distribution requests to Hound workers in all node groups in parallel. To discover node failure in time, IM receives the heartbeat information from all Hound workers. The state of each Hound worker is updated periodically by overwriting the last heartbeat information stored in NIDB with the received information.

4.3 ICA

ICA is a web server running on each Hound worker, which captures image distribution requests forwarded by the Hound master. Note that the image list in the request is parsed so that ICA can download images in parallel. The role of ICA is to perform image distribution operations according to the image list and send the heartbeat information to the Hound master periodically to inform its health state.

4.4 Custom middleware

The custom middleware is designed to ensure secure communication of Hound by verifying the consistency of the token carried in the image distribution request and the one which the Hound worker possesses. The token is encapsulated by the SHA256 algorithm according to the hostname and timestamp. Note that the token of each Hound worker is updated periodically by attaching a refresh flag to the request. The updated token will be sent to the Hound master for further communication. To prevent the Hound master from failing to receive the token, an environment variable called Hound_key is presented. The latest token will be sent to the Hound master if the correct Hound_key is included in the request.

4.5 Hound workflow

Fig. 3 shows the workflow of Hound which is divided into four steps and each step corresponds to a black bounding box.

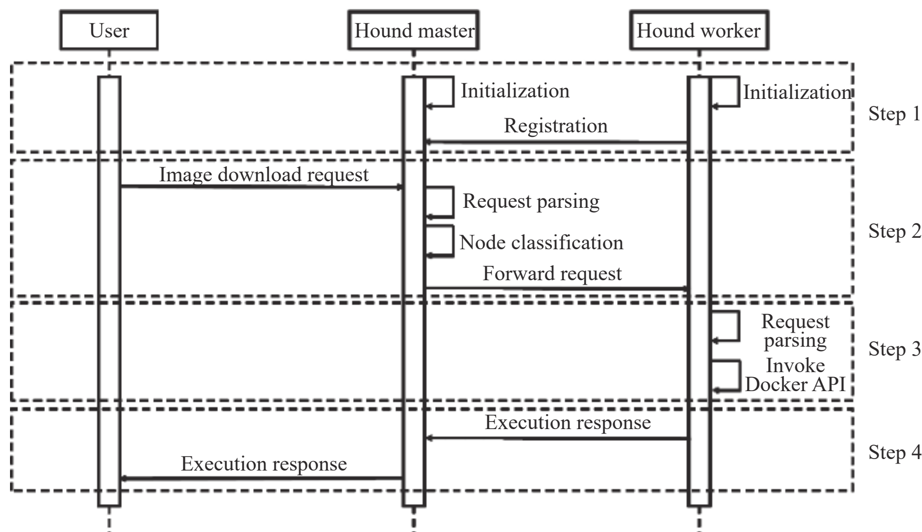

Fig. 3 Workflow of Hound

Step 1 Establish Hound cluster.

Both the Hound master and Hound workers require to perform initialization first multi-thread. The initialization for the Hound master is mainly about establishing a connection between the IM and the NIDB. As for Hound workers, the initialization includes token generation and the ICA startup. After initialization, the Hound worker sends an application programming interface (API) request to the Hound master for applying to join the cluster. After the worker is successfully joined, the token is sent to the Hound master for further communication. The heartbeat information is sent to the Hound master periodically (every five minutes in our implementation) to inform its health state.

Step 2 Request parsing and node classification.

After the cluster is established, users can initiate an image distribution request to the Hound master. Nodes and their required images are parsed by IM. Then, the Hound master classifies nodes into multiple node groups according to the requested images. After that, the Hound master asynchronously forwards the image distribution requests to Hound workers in all node groups by adopting multi-thread.

Step 3 Execute image distribution operations.

Note that the token in the request is verified by the custom middleware and the request will be discarded if the token in the request is inconsistent with the one which the Hound worker possesses. After that, the request is parsed by ICA to obtain the image list. Images in the image list are downloaded in parallel by invoking Docker API.

Step 4 Return execution result.

The execution result will be sent to the Hound master after the image distribution operation is completed. The Hound master stores the execution result to the NIDB, followed by informing the user that the request is completed.

5. Evaluation

In this section, we first briefly describe the experimental setup. Furthermore, we compare Hound with the native Docker, CRI-O, and Docker-compose in terms of cluster image distribution performance, followed by the demonstration of Hound scalability.

5.1 Experimental setup

In our experiments, Hound is deployed on a cluster with 11 servers. Each server is equipped with four Intel(R) Xeon(R) Silver 4114 CPU @ 2.20 GHz (40-cores CPU), four 16 GB DDR4 memory (64 GB of RAM), and four WD Black 1TB hard-disks. Each server possesses exclusive gigabit network bandwidth. We select one of them as the Hound master, and the others serve as Hound workers. All the experiments of Hound are performed with Docker engine 19.03 while the image distribution requests are sent to the Hound master through Postman 7.25.3. We repeat each experiment three times to obtain the average results in all experiments for ensuring the validity of the results.

We have selected several container images from Docker Hub, which are shown in Table 4. The selected images cover multiple aspects such as deep learning framework, web server, programming language, compiler, operating system, etc. Images are downloaded from the private registry we established instead of Docker Hub in all experiments. Considering the image list in the request contains more than one image, different image combinations are required to be involved in the experiments. We define the request number to denote different image combinations involved in the request, which is shown in Table 5.

Table 4 Information about the selected images

Content	Image size
Tomcat	140 MB, 329 MB
Jetty	225 MB
Django	267 MB
Gcc	409 MB
Rust	543 MB
Ros	768 MB
Silverpeas	845 MB
Nuxeo	910 MB
MXnet	1.34 GB
Tensorflow	1.52 GB, 2.25 GB
Pytorch	3.39 GB, 4.2 GB

Table 5 Correlation between the request number and the image combination

Request number	Image combination
1	140 MB + 225 MB
2	267 MB + 329 MB
3	409 MB + 543 MB
4	768 MB + 845 MB
5	910 MB + 1.34 GB
6	1.52 GB + 3.39 GB
7	2.25 GB + 4.2 GB
8	140 MB + 267 MB + 409 MB
9	768 MB + 845 MB + 910 MB
10	1.34 GB + 1.52 GB + 3.39 GB

5.2 Experimental results

5.2.1 Image distribution performance

The image distribution performance is evaluated by the time consumed from the creation of the image distribution request to the download completion. We compare Hound with the native Docker 19.03, CRI-O 1.17.0, and Docker-compose 1.25.5. Note that Docker and CRI-O implement image download function by using docker pull command and crictl pull command (https://github.com/containerd/cri/blob/master/docs/crictl.md), respectively. While the method for Docker-compose to download images is to use the docker-compose pull command with the --parallel flag (https://docs.docker.com/compose/reference/pull/), the required YAML configuration file is set in advance.

We first illustrate the effectiveness of Hound when all

nodes request to download the identical image combinations, which is shown in Fig. 4. The x axis represents the request numbers of different image combinations, and the y axis represents the time required for image distribution. We can observe that Hound can improve the image distribution performance in the range of 31% to 53% compared to the native Docker and CRI-O. Moreover, the margin between Hound and these two schemes becomes larger when the image size grows. This is because images are downloaded sequentially by both Docker and CRI-O, while images are downloaded in parallel by Hound owing to the multi-thread technique. Moreover, Hound achieves performance comparable to Docker-compose because both of them achieve parallel image download through the multi-thread technique and invoking Docker API in the background.

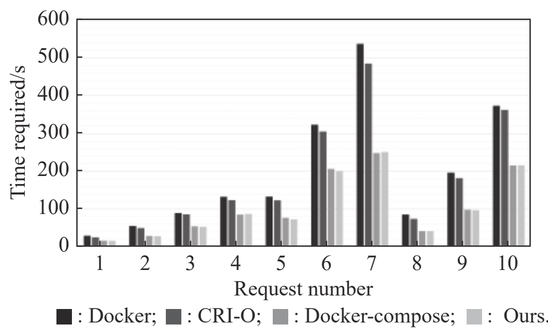

Fig. 4 Comparison of image distribution performance between Docker, CRI-O, Docker-compose, and Hound when all nodes request the identical image combination

Then, we further consider the image distribution performance of Hound and Docker-compose when nodes request different image combinations, which is shown in Fig. 5. Three cases (i.e., Case A, Case B, and Case C) are selected to represent requests of nodes for different image combinations, which is shown in Table 6. Compared with Docker-compose, we can observe Hound reduces image distribution time by 11.7%, 20.3%, and 31.6% in three cases, respectively. This is because Docker-compose has to initiate a particular image distribution request when different images are requested, which is relatively time-consuming. For example, given the node which requests the 2nd image combination and another node which requests the 5th image combination, Docker-compose requires to initiate different image distribution requests to these nodes. While Hound classifies nodes with the identical requested image combination into a node group and forwards requests to each node in all node groups in parallel by the multi-thread technique. Consequently, Hound outperforms Docker-compose in terms of image distribution performance when nodes in the cluster request different image combinations.

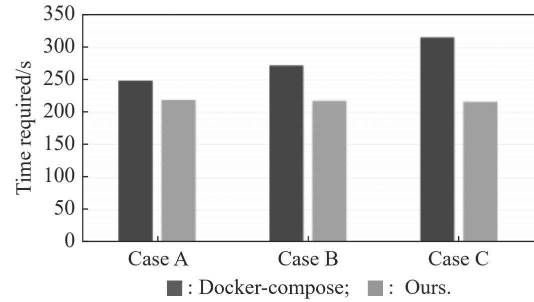

Fig. 5 Comparison of image distribution performance between Docker-compose and Hound when nodes request different image combinations

Table 6 Three selected cases of node request for different image combinations

Request number	1	2	3	4	5	6	7	8	9	10
Case A	0	3	0	0	3	0	0	0	0	4
Case B	2	0	0	2	0	2	0	0	2	2
Case C	1	1	1	1	1	1	1	1	1	1

5.2.2 Hound scalability

We demonstrate the scalability of Hound in terms of the image distribution performance, resource consumption, and response time as the number of Hound workers grows from 1 to 10.

Fig. 6 shows the image distribution performance of different numbers of Hound workers. Although the number of Hound workers grows from 1 to 10, we can observe that the margin of the required time for downloading these image combinations is negligible (the maximum margin is within 5 s). The result demonstrates that Hound achieves high scalability in terms of image distribution.

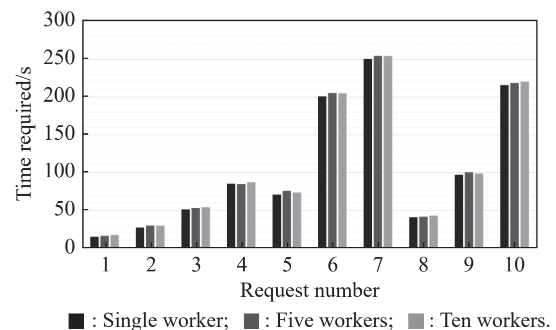

Fig. 6 Image distribution performance of different numbers of Hound workers

Then, the resource consumption of Hound, which includes average load, CPU usage, disk write speed, and network bandwidth, is evaluated as the number of Hound workers grows. We adopt several testing tools to monitor the consumption of these resources during the image dis-

tribution process:

Average load: We monitor the average load by watching the /proc/loadavg at a 1-s granularity.

CPU usage: We monitor the CPU utilization with the top command which monitors the CPU utilization by the process in real-time at a 1-s granularity.

Disk write rate: We monitor the disk write speed with the iotop command which monitors the number of bytes written to the disk by the process at a 1-s granularity. The average value is shown in the result.

Network bandwidth: We monitor the traffic to the Docker daemon (excluding other unrelated processes) on the Ethernet interface using NetHogs tool at a 1-s granularity.

The comparison result is shown in Table 7. Our experiment is implemented in the case of downloading the 10th image combination. It can be observed that the average load, disk write rate, and network bandwidth maintain a relatively stable state as the number of workers increases during the image distribution process. Thus, the result demonstrates that Hound achieves high scalability in terms of resource consumption.

Table 7 Resource consumption of different numbers of Hound workers (request number: 10)

Scenario	Resource				
	Average load(idle)	Average load(busy)	CPU usage/%	Disk write rate/(MB/s)	Network bandwidth/(MB/s)
Single worker	0.14	3.98	339.8	63.47	90.98
Five workers	0.14	3.81	342.4	61.83	90.68
Ten workers	0.14	3.72	327.2	60.58	90.27

Finally, we evaluate the response time of different numbers of Hound workers, which is shown in Table 8. Likewise, our experiment is implemented in the case of downloading the image combination which corresponds to the request number 10. Note that the response time of the Hound master is denoted as the time elapsed from the request initiation to the receipt of the reply from all Hound workers. The response time of the Hound worker refers to the time elapsed from the receipt of the forwarding request to the start of the image download. We can observe the response time of the Hound master increases but within an acceptable range (from 12 ms to 21 ms) as the number of Hound workers grows. This is because the Hound master requires to forward the image distribution request to more Hound workers in parallel and keep waiting until all Hound workers reply to it. The same trend is seen in the response time of the Hound worker due to the increasing transmission overhead of requests. Consequently, the result demonstrates that Hound achieves high scalability in terms of response time.

Table 8 Response time of different numbers of Hound workers (request number: 10)

Scenario	Master response time/ms	Worker response time/μs
Single worker	12.68	183.87
Five workers	17.37	245.24
Ten workers	21.05	268.18

5.3 Discussions

Note that the maximum number of Hound workers in our experiment is 10, which is relatively small compared to the current cluster size of the production environment. Thus, the scalability of Hound in the scenario of a large-scale cluster remains to be evaluated, which is served as our future work.

In Subsection 5.2, we can observe that the response time increases as the number of Hound workers grows because of the request forwarding overhead. Based on this observation, we infer that the only Hound master will become the bottleneck in the scenario of a large-scale cluster. Meanwhile, the only registry will also become the bottleneck in the scenario of a large-scale cluster because of the image download overhead. Thus, multiple Hound masters and registries require to be deployed in the scenario of a large-scale cluster. In the future, we will focus on designing the load balancing scheme of Hound master for efficient request forwarding and the image distribution strategy which selects the optimal registry to fetch images.

Note that Hound attempts to achieve parallel image distribution in the cluster view, which is orthogonal to works dedicated to improving image distribution performance. Thus, these works can be combined with Hound to further improve the image distribution performance. In the future, the combination of single-node image distribution strategy and Hound requires to be further explored.

6. Conclusions

In this paper, we propose a cluster image distribution system based on Docker called Hound which downloads images to the destination nodes in parallel. We observe

that the existing image distribution strategies only take effect when distributing identical images to the destination nodes. To support diverse image distribution demands in the cluster view, we propose a novel image distribution mechanism which consists of node classification and node-level parallelism. Experimental results demonstrate that Hound achieves better image distribution performance compared to Docker, CRI-O, and Docker-compose when nodes in the cluster request different images. Moreover, the high scalability of Hound is also evaluated in the scenario of ten Hound workers.

References

[1] MERKEL D. Docker: lightweight linux containers for consistent development and deployment. Linux Journal, 2014. https://www.linuxjournal.com/content/docker-lightweight-linux-containers-consistent-development-and-deployment.

[2] SMET P, DHOEDT B, SIMOENS P. Docker layer placement for on-demand provisioning of services on edge clouds. IEEE Trans. on Network and Service Management, 2018, 15(3): 1161–1174.

[3] SINGH S, SINGH N. Containers & Docker: emerging roles & future of Cloud technology. Proc. of the 2nd International Conference on Applied and Theoretical Computing and Communication Technology, 2016: 804−807.

[4] MERELLI I, FORNARI F, TORDINI F, et al. Exploiting Docker containers over Grid computing for a comprehensive study of chromatin conformation in different cell types. Journal of Parallel and Distributed Computing, 2019, 134: 116–127.

[5] PAHL C. Containerization and the paas cloud. IEEE Cloud Computing, 2015, 2(3): 24–31.

[6] THALHEIM J, BHATOTIA P, FONSECA P, et al. CNTR: lightweight {OS} containers. Proc. of the {USENIX} Annual Technical Conference ({USENIX}{ATC} 18), 2018: 199–212.

[7] PENG Y H, BAO Y X, CHEN Y R, et al. Dl2: a deep learning-driven scheduler for deep learning clusters. IEEE Trans. on Parallel and Distributed Systems, 2021, 32(8): 1947–1960.

[8] WU Y D, MA K H, YAN X, et al. Elastic deep learning in multi-tenant GPU clusters. IEEE Trans. on Parallel and Distributed Systems, 2021, 33(1): 144–158.

[9] ZHOU Q H, GUO S, QU Z H, et al. Petrel: heterogeneity-aware distributed deep learning via hybrid synchronization. IEEE Trans. on Parallel and Distributed Systems, 2020, 32(5): 1030–1043.

[10] ZHAO N N, TARASOV V, ALBAHAR H, et al. Large-scale analysis of docker images and performance implications for container storage systems. IEEE Trans. on Parallel and Distributed Systems, 2020, 32(4): 918–930.

[11] LEE C, KIM S, KIM E. A deduplication-enabled P2P protocol for VM image distribution. IEICE Transactions on Information and Systems, 2015, 98(5): 1108–1111.

[12] DU L, WO T Y, YANG R Y, et al. Cider: a rapid docker container deployment system through sharing network storage. Proc. of the IEEE 19th International Conference on High Performance Computing and Communications, IEEE 15th International Conference on Smart City, IEEE 3rd International Conference on Data Science and Systems, 2017: 332−339.

[13] ZHANG S Q, WU S, FAN H, et al. BED: a block-level deduplication-based container deployment framework. Proc. of the International Conference on Green, Pervasive, and Cloud Computing, 2020: 504−518.

[14] ZHAO X, ZHANG Y, WU Y W, et al. Liquid: a scalable deduplication file system for virtual machine images. IEEE Trans. on Parallel and Distributed Systems, 2013, 25(5): 1257–1266.

[15] SAHARAN S, SOMANI G, GUPTA G, et al. QuickDedup: efficient VM deduplication in cloud computing environments. Journal of Parallel and Distributed Computing, 2020, 139: 18–31.

[16] HARTER T, SALMON B, LIU R, et al. Slacker: fast distribution with lazy docker containers. Proc. of the 14th {USENIX} Conference on File and Storage Technologies, 2016: 181−195.

[17] CIVOLANI L, PIERRE G, BELLAVISTA P. FogDocker: start container now, fetch image later. Proc. of the 12th IEEE/ACM International Conference on Utility and Cloud Computing, 2019: 51−59.

[18] ZHENG C, RUPPRECHT L, TARASOV V, et al. Wharf: sharing docker images in a distributed file system. Proc. of the ACM Symposium on Cloud Computing, 2018: 174−185.

[19] CHEN J L, LIAQAT D, GABEL M, et al. Poster: an accelerator for fast container-based applications deployment on the edge. Proc. of the IEEE/ACM Symposium on Edge Computing, 2020: 175−177.

[20] AHMED A, PIERRE G. Docker container deployment in fog computing infrastructures. Proc. of the IEEE International Conference on Edge Computing, 2018: 1−8.

[21] PENG C Y, KIM M, ZHANG Z, et al. VDN: virtual machine image distribution network for cloud data centers. 2012 Proceedings IEEE INFOCOM, 2012: 181−189.

[22] ZHANG Z N, LI Z Y, WU K, et al. VMThunder: fast provisioning of large-scale virtual machine clusters. IEEE Trans. on Parallel and Distributed Systems, 2014, 25(12): 3328–3338.

[23] LIANG M Y, SHEN S Q, LI D S, et al. HDID: an efficient hybrid docker image distribution system for datacenters. Proc. of the National Software Application Conference, 2016: 179−194.

[24] WANG K J, YANG Y, LI Y, et al. Fid: a faster image distribution system for docker platform. Proc. of the IEEE 2nd International Workshops on Foundations and Applications of Self* Systems, 2017: 191−198.

[25] NATHAN S, GHOSH R, MUKHERJEE T, et al. Comicon: a co-operative management system for docker container images. Proc. of the IEEE International Conference on Cloud Engineering, 2017: 116−126.

Biographies

LIU Zijie was born in 1996. He received his B.S. degree from Nanjing Institute of Technology, Nanjing, China, in 2014. He is currently pursuing his Ph.D. degree in information network with Nanjing University of Posts and Telecommunications, Nanjing, China. His current research interests include cloud computing and distributed systems.
E-mail: 2019070274@njupt.edu.cn

LI Junjiang was born in 1996. He received his B.S. degree from Tongda College of Nanjing University of Posts and Telecommunications, Yangzhou, China, in 2014. He is currently pursuing his M.S. degree in logistics engineering with Nanjing University of Posts and Telecommunications, Nanjing, China. His current research interests include cloud computing, container orchestration systems, and distributed systems.
E-mail: admin@run-linux.com

CHEN Can was born in 1993. He received his B.S. degree from Nanjing University of Posts and Telecommunications, Nanjing, China, in 2015. He is currently pursuing his Ph.D. degree in signal and information processing in the College of Telecommunications and Information Engineering, Nanjing University of Posts and Telecommunications, Nanjing, China. His research interests include image and video coding, image and video processing, machine learning, and compressive sensing.
E-mail: chencan@njupt.edu.cn

ZHANG Dengyin was born in 1964. He received his B.S., M.S. and Ph.D. degrees from Nanjing University of Posts and Telecommunications, Nanjing, China, in 1986, 1989 and 2004 respectively. He was in Digital Media Lab at Umea University in Sweden as a visiting scholar from 2007 to 2008. He is currently a professor with the School of Internet of Things, Nanjing University of Posts and Telecommunications, Nanjing, China. His research interests include signal and information processing, networking technique, and information security.
E-mail: zhangdy@njupt.edu.cn