FID: A Faster Image Distribution System For Docker Platform

Wang Kangjin[1,2], Yang Yong[1,2]
[1]School of Software and Microelectronics, Peking University
[2]PKU-Tencent Joint Innovative Lab
Beijing, China
{wangkangjin,yang.yong}@pku.edu.cn

Li Ying[*,2]
National Engineering Center of Software Engineering,
Peking University
[2]PKU-Tencent Joint Innovative Lab
Beijing, China
li.ying@pku.edu.cn

Luo Hanmei, Ma Lin
TEG Group, Tencent
Beijing, China
{mavis,luffyma}@tencent.com

***Abstract*—Docker has been widely adopted in enterprise-level container environment. As an important part of Docker-based container ecosystems, Docker Registry provides the service of storing, distributing and managing Docker images, which are crucial to run Docker containers. In large-scale container platforms, deploying a large-scale application is prone to overburdening Docker Registry with flooding network traffic, and this situation may even cause failures of image services. In this paper, we present a new P2P-based large-scale image distribution system called FID (Faster Image Distribution), which is able to serve a larger container cluster by taking full advantage of the bandwidth in not only Docker Registry but also other nodes in the cluster. We implemented and validated FID on the enterprise-level cluster. The experiment results show, that compared to the native image distribution method, FID reduces at least 97% of network traffic for Docker Registry, furthermore, reduces 83.50% of distribution time on average when distributing images among 200 nodes, and particularly, reduce up to 91.35% of distribution time for the 500M image of Hadoop.**

Keywords—Docker; Docker Registry; Peer-to-Peer Network; Distributed Container System

I. INTRODUCTION

As a kernel-level virtualization technology, Docker [1] provides an isolated environment for every container to run apps on the same machine. Developers can easily customize and package the runtime environment of their apps in a set of files, which is called Docker Image. Once a Docker Image is created, users can run the image on any host machine with Docker Engine. The feature of "Build once and run anywhere" greatly simplifies the development, delivery and deployment of applications. Nowadays, more and more companies built their own container-based platforms with Docker. Google is a powerful supporter of Docker. The container management platform developed by Google, called Kubernetes [2], is an important part of the Docker ecosystem. Tencent, the biggest internet company in China, built a container management platform called Gaia [17] which manages over 8000 physical machines, and there are over 150 million containers running on it every day. IaaS provider, such as Tencent Cloud, Ali Cloud and Baidu Cloud also provide their own container-based services. In the future, Docker and Docker-based platforms will play an important role in the IT industry.

In Docker system, which provides the service of storing, distributing, and managing Docker images, Docker Registry [3] is an important component. A new deployment of Docker container comprises (1) Pulling the published image from the Docker Registry, and (2) Starting a container based on the image. Pulling images from Docker Registry is solution of distributing native images and it's a common and important operation that 76% of the time spent on a new deployment will be spent on the pulling [16]. In order to speed it up, administrators usually deploy a private Docker Registry in local clusters. Although it works in small cases, the Docker Registry host will no doubt become a bottleneck for a large-scale container deployment. In our practice of Gaia, when we distribute one container onto 3,000 hosts simultaneously, with the native Docker Registry, the time of deployment is unacceptable. With the depletion of bandwidth, image services provided by the Docker Registry becomes unavailable. Deploying Two or more instances of Docker Registry may alleviate this problem to some degree but new problems such as image synchronization and load balancing will be induced. Therefore, a faster and elegant method to distribute Docker image for large-scale container deployment is in great need.

Considering the similarity between downloading a file from a server and pulling an image from a Docker Registry, we proposed a new way to accelerate image distribution for large-scale container deployment by introducing P2P network to make full use of the bandwidth of all the nodes to share the burden of Docker Registry and increase the speed of deployment. We implemented the P2P-based large-scale image distribution system called FID based on the open-source project Docker Distribution [7] and conducted experiments on a 200-nodes cluster to validate our solution. The experiment result shows, that FID can

*corresponding author.

provide 91.35% of distribution time reduction at most when distributing a 500M-size Docker image on 200 nodes in parallel. And FID can reduce at least 97% network traffic for Docker Registry. To the best of our knowledge, FID is the first enterprise-level P2P Registry system.

The rest of this paper is organized as follows. In Section II we discuss our motivation and design goal. In Section III we list related works. We describe our new solution in detail in Section IV. In Section V we elaborate our system design for FID. In Section VI we present our experiment design and results. Section VII summarizes our work and gives an insight into our future work.

II. Motivation & Goal

Gaia is an internal resource management and container orchestration platform, which provides a platform for various kinds of applications, such as online statistics business and message pushing business. The scale of these applications ranges from 10 to 100+ nodes. With more and more business being deployed on Gaia and the scale of application becoming larger and larger, it challenges the large-scale container deployment capability of the current image distribution solution. For example, the scale of Spark [10] offline analysis job could be very large that thousands of images pulling requests arrive at Registry simultaneously. It is foreseeable that large-scale container deployment will be a frequent and challengeable task. We conclude three features that large-scale container deployment task has.

1) High concurrency: When deploying a container based application in the cluster, a large number of identical images pulling requests will be sent to the Docker Registry at the same time.

2) Simple network structure: Usually, a container platform is deployed in a private network which has high speed and low latency. Every host can use a private IPv4 address to communicate to other hosts. Owning to this simple network structure, we do not need to handle the situation that a host is hidden behind a NAT network.

3) Stable P2P network: When performing large-scale container deployment, the networks topology is quite stable, which means the nodes won't frequently join in or quit from the network.

As described earlier, based on our practice of Gaia, with the rapidly growing scale of cluster and application, the limited bandwidth of Docker Registry will be the bottleneck when distributing container images to thousands of nodes simultaneously. Although we can deploy more Docker Registry instances, it cannot totally solve the problem and the bandwidth of the shared backend storage will be a new bottleneck. Therefore, a new solution to distributing Docker Images is in great need. And here are three goals.

1) Reduce the distribution time of large-scale container deployment.

Large-scale container deployment generally means lots of nodes pulling the same image from the Docker Registry at the same time. Deployment time is an important metric, which consists of two parts: *distribution time* and *start-up time*. Usually, starting a container will only take one to few seconds due to the lightweight feature of Docker container. Therefore, how to shorten the distribution time is the key to increase the speed of large scale container deployment. We define *distribution time* as follows: the mean time of all the target nodes pulling the image. The reason we don't use the pulling time of the last node to finish pulling is to eliminate the effects of abnormal pulling due to hardware or software failures. For example, we used devicemapper to create and store images. Sometimes bugs of devicemapper will cause the pulling image process become much slower than it should be. The bugs of devicemapper sometime even hang the pulling process.

2) Reduce network traffic of Docker Registry

When running a large-scale deployment job in a cluster, the network burden of Docker Registry will be heavy, which leads to increasing response latency of other requests. If the network pressure of Docker Registry was significantly alleviated for large scale image distribution, the serving capability of Registry could be greatly improved and it can continuously serve the cluster even the scale of cluster still growing.

3) Avoid invading the existing Docker Engine source code

In order to adjust to internal business processing requirements, the original Docker Engine is modified in our existing system. Several features have been added to Docker Engine, such as disk I/O limit and hot-restart (added before Docker-1.10). The customized Docker Engine has been installed in thousands of nodes. If the existing Docker Engine source code was intruded, it will increase the operation and maintenance costs. Besides, we want to provide a general solution which is not limited to a specific Docker Engine version and doesn't require users to modify their Docker Engine. Thus a well-designed and non-intrusive solution is necessary and important.

III. Related Work

In this section, we will review major related works of distributing Docker image with the P2P network.

A. VMware Harbor

Harbor [8] is a Registry server which stores and distributes Docker images. Harbor extends the open source Docker Distribution by adding the functionalities commonly required by an enterprise, such as security, authentication and management. Harbor will integrate P2P functions in the future and provided a solution named DID [20]. In DID, they use BT as the P2P distribution implementation. And they have done some experiments with their prototype system.

They implemented a prototype system and validated their P2P solution in a 10 physical machines cluster. Each of the physical machines has 10 virtual machines on it. The experiment shows that with the growing of image size and cluster scale, P2P distribution solution can definitely outperform the native Docker pulling method. There is a "balance point" between Docker and P2P, which is up to the image size and cluster scale. The Docker pulling method has obvious advantage before the "balance point". But once image size or cluster scale grows beyond the "balance point", P2P solution will show its significant performance improvement compared to native Docker pulling. In their experiments, using DID to distribute images will get 80% speed-up than original Docker pulling.

In DID, Controller has to download the whole image from Docker Registry with the native Docker pulling method first, which will increase the image distribution time. P2P distribution can only start after Controller fetches the whole image and starts to be the first seeder to provide the original image resource in the P2P network.

B. *Quay*

Quay [21] is a commercial product that provides Docker Registry service with continuous integration, private image repository, user authentication/authorization, team management and other features for enterprises. Quay also supports the image format of rkt [14]. With the open source Quay client [9], users are able to pull Docker images via BitTorrent.

Similar to DID, Quay doesn't provide P2P service in its Docker Registry. Therefore, a node has to download the whole image first, while other nodes are waiting or pulling the image from Docker Registry, which will decrease the speed and light weight advantage of P2P. Once the first download process is done and this node becomes the first seeder, all other nodes in the P2P network are able to download Docker images to local disk via BitTorrent. Finally, images are loaded into Docker Engine by calling Docker "load".

C. *Docket*

Docket is a Docker Registry that adopts BitTorrent to distribute Docker images to a large number of machines. Docket generates a torrent file for the whole Docker image. Then BT client can use the torrent file to download a tarball file of a Docker image. Then Docket calls "docker load" interface to load the tarball into Docker Engine. Downloading the whole image with one torrent file makes Docket unable to reuse the layer that already exists on local host, which causes a waste of network resources.

DID, Quay and Docket are making torrent for the whole Docker image. In this way, BT client needs to download the entire image tarball although some of the layers already exist. It is not necessary to download the existing layers. In our solution, we have done some optimization in this problem.

P2P has been introduced to accelerate the dispatching of virtual machine image by previous works. VDN [2] shares VM image chunks during distribution. And VDN has an excellent image distribution performance under heavy traffic. Considering that a Docker image is made up from multi-layers, distributing Docker images in chunk level may break the layer structure among the layers. So VDN is not suitable for Docker image distribution. VMtorrent [27] also introduces P2P in VM distribution and gains a performance improvement.

IV. METHOD

A Docker image consists of a series of layers. Each layer is a read-only file system. A layer contains a set of files or folders. A *Docker image* in *Docker Registry* is stored as two main types of static files: *Manifest* and *Blobs*. *Manifest* describes the meta information about a *Docker image*. It indicates the layer that the image has and the *Blob* each layer refers to. Blob is a compressed file of the layer. Every layer has one corresponding *Blob*. The relationship and difference of Docker image stored in *Docker Engine* and *Docker Registry* are shown in Figure 1.

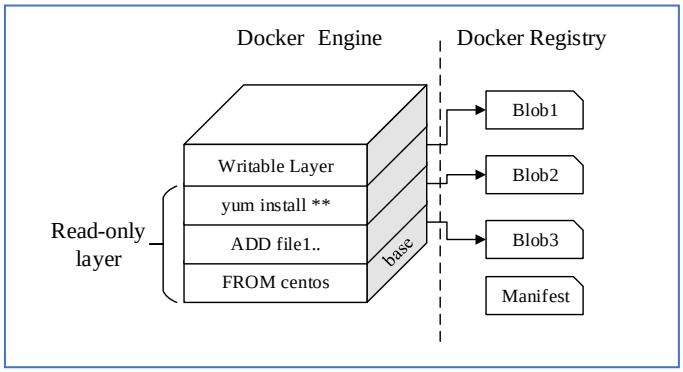

Fig. 1. The structure of a *Docker image* and the files corresponding to it in *Docker Registry*.

Figure 2 shows the process of pulling a *Docker image*. It can be summarized as follows.

1) *Get the Manifest file of the image*
2) *Find layers that are not existing*
3) *Download Blobs if the corresponding layers do not exist in the local Docker Engine*
4) *Decompress Blobs and import them into Docker Engine*

Usually, *Manifest* is a small text file. Therefore, only the way of how the blob is downloaded should be optimized. If we want to use P2P image distribution, we can change the step (3) to P2P downloading. Considering the usage scenario (3) we mentioned in Section II, the internal network topology is more stable compared to the Internet. In the Internet, peers may join in or quit from a P2P network in anytime [24]. And the object in our distribution is static files. So we can use BitTorrent for image P2P distribution. For downloading blobs with

BitTorrent, every blob needs a torrent file corresponding to it. And Docker Registry should be the torrent maker and initial seeder. The time to make a torrent file is the blob uploading finished. Blob data and torrent files are stored on the backend storage. So the Docker Registry needs to provide a torrent fetching interface and every BitTorrent client can fetch a torrent file via this interface.

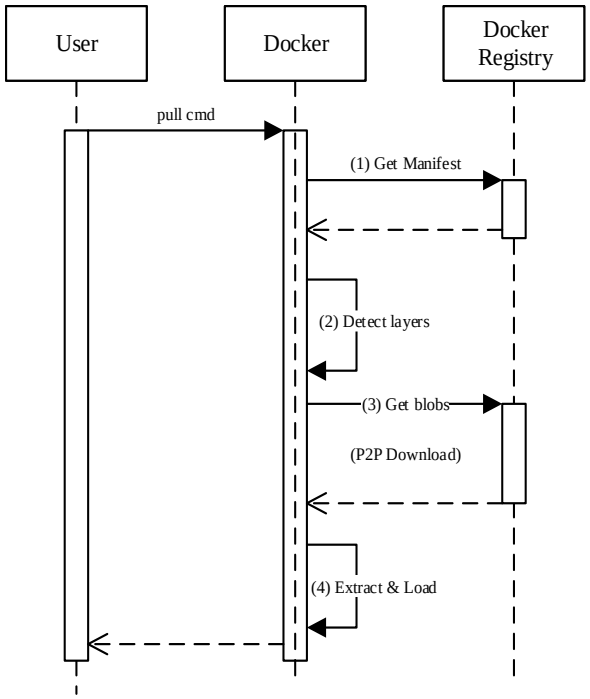

Fig. 2. The sequence diagram of pulling a *Docker image*. It can be divided into four steps: getting Manifest, finding the not existing layers, downloading *Blobs* of the layers, and extracting and importing them into *Docker Engine*.

Another way to get a Docker image is *docker load*. It loads an image from a compressed tar archive or STDIN. The contents of the tar archive are blobs and layers configurations. If we want to import an image by *docker load*, a tar archive of this image is needed. *Registry* provides interfaces for getting manifest and blob via HTTP. We can get materials that the tar archive required by calling *Registry*'s interfaces. The way that integrates BitTorrent in the above process is by using BitTorrent in blob downloading.

V. Implementation

In order to achieve the goals listed in Section II. We carefully designed the system architecture. Figure 3 shows the architecture of FID. Several instances of P2P Registry are deployed in the system. For keeping the data consistency between P2P Registries, a shared distributed storage is adopted. FID Agent is a new component that is responsible for downloading Docker images. BT trackers keep peers' information. Every peer can find each other via trackers. The details of every component are described below.

A. P2P Docker Registry

Based on Docker Registry [7], We developed P2P Docker Registry that supports P2P image downloading. There are three main changes.

1) Add BitTorrent client module in Registry runtime.

We added a BitTorrent client in Registry runtime. It is the core modification in P2P Registry. But the BT client in P2P Registry has a feature that is different with other FID Agent's. That is BT client in P2P Registry only transferring data to other peers without receiving data. Because P2P Registry has the complete data of the blobs.

2) Generate torrent file at the end of Blob uploading

When Docker Engine pushes an image to Docker Registry, Docker Engine exports every layer of this image as compressed *Blob* files and then uploads them via Blob uploading interface provided by Registry. The best time for torrent file generation is at the end of the Blob uploading. So we modified the implement of Blob uploading and added a torrent generation function. After torrent file is generated, the torrent file should be stored on the backend storage.

3) Added an interface of getting torrent files

In this interface, other clients can get torrent files of layers via http requests. Clients need to provide the Blob ID of a layer in the http request. The response of this interface is the content of the torrent file. But P2P Registry needs to be the initial seeding peer in the BitTorrent distribution. Otherwise no one can get the data even though they hold the torrent file.

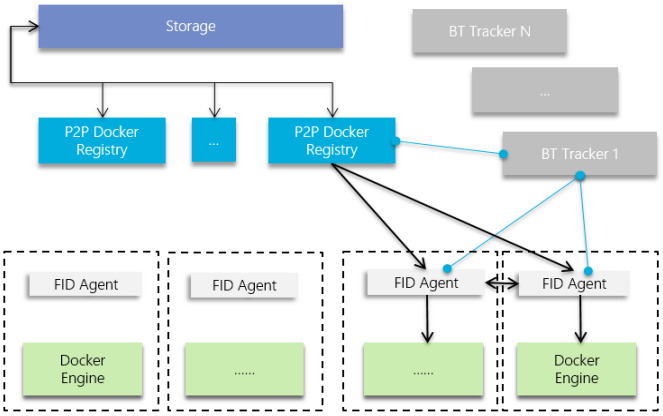

Fig. 3. Architecture of FID. Boxes with dotted line represent a Docker Host. While the black lines indicate the data flow of Docker images. Blue lines mean the communication between BitTorrent and trackers.

To be the initial seeding peer, P2P Registry should get the Blob from the backend storage and store the Blob in BT client's work directory. Then BitTorrent client gets the corresponding of this blob and downloads file specified in the torrent file. Because the file already exists in BT client

work directory, BT client only needs to announce to trackers that it has all pieces of this resource. Then other peers can find P2P Docker Registry by getting peers interface provided by *trackers*.

B. FID Agent

In order to avoid modifying the already existing source code of Docker Engine, we developed a component called FID Agent to handle BT downloading. FID Agent has two different working modes. The first one is called *load mode*. In this mode, FID Agent downloads Blobs and package them into Docker image's tar archive. Then the agent calls Docker's *load* interface to load this tar archive [23]. DID and Docket have similar implements. Another mode is called *proxy mode.* In this mode, the FID Agent is running like a lightweight HTTP proxy for the Docker Engine. FID Agent intercepts Docker's Blobs downloading requests, then downloads Blobs via BitTorrent and returns the data of Blobs to *Docker Engine*. In FID Agent, random peer selection policy is adopted to avoid that the P2P Registry is more inclined to be chosen as peer, which could reduce P2P Registry's network overload.

The differences between *load mode* and *proxy mode* are elaborated below.

1) Load mode

While studying the interfaces of Docker Engine, we found that there is no interface that can import a layer of an image separately. And the only way to import a Docker image is to call "docker load" interface. So in this mode, FID Agent downloads the layers that is not existing in the Docker daemon and packages them into a tar file. The pseudo-code for FID running in *load mode* is shown in Algorithm 1.

Algorithm 1 Pseudocode of Load Mode

```
loop
  request = getImagePullRequestFromUser()
  manifest = getManifestFromRegistry()
  layers = getLayersFromManifest(manifest)
  wantedBlobIDs = filterOutExistingLayers(layers)
  for all blobID in wantedBlobIDs do
    torrentFile = getTorrentFromRegistry(blobID)
    downloadTorrent(torrentFile)
  end for
  tar = package downloaded blobs
  load tar into Docker
end loop
```

In *load mode*, FID Agent provides a restful API to users. Users can call this API with the image name to get the image they want. After receiving the pulling request, FID Agent will call the P2P Registry's restful API to get the Manifest for specific image. Next, FID Agent analyzes the Manifest and gets the Blob IDs belong to the image. In this step, the FID Agent checks whether layers already exist on Docker host. The existing layer won't be downloaded. After this step, a list of Blob IDs that need to be downloaded will be produced. Then the FID Agent calls the "Get Torrent" interface of P2P Registry for every Blob. And the FID Agent puts torrent files that just got to the BT client to start the downloading process. BT client is running in the FID Agent. BT client creates download task for every torrent file. When all the download tasks are finished, FID Agent package the Blobs and configuration files according to image tar archive's organization. Finally, "docker load" interface provided by Docker client library can load this tar archive into Docker Engine.

2) Proxy mode

There is a configurable option for Docker Engine called "http_proxy". This configuration specifies a proxy server for Docker Engine. Every http request sent from Docker Engine will be forwarded to proxy server first. Based on this feature, we developed *proxy mode* for FID Agent. The pseudo-code for FID running in *proxy mode* is shown in Algorithm 2.

In *proxy mode*, FID Agent intercepts all the http requests from Docker Engine. But FID Agent only processes the Blob requests and simply forwards other requests. FID Agent extracts Blob ID from requests. Then it calls the "Get Torrent" interface of P2P Docker Registry to get the torrent belonging to this Blob. And BT client does the same thing as *load mode* does. At the beginning of the downloading task, FID Agent needs to get a reader of the downloading file. Then FID Agent can read the data from this reader and write the data to Blob request continually.

The *proxy mode* doesn't need to handle a lot of problems, such as Docker version adapting, layer detecting, and images loading. In *proxy mode*, FID Agent just focuses on Blob downloading.

Obviously, *proxy mode* is more light weight than *load mode*. We conduct some experiments to compare the performance about the two modes in the next section.

Algorithm 2 Pseudocode of Proxy Mode

```
loop
  request = CaptureRequestFromDocker()
  if isGetBlob(request) then
    blobID = getBlobID(request)
    torrentFile = getTorrentFromRegistry(blobID)
    fileReader = downloadTorrent(torrentFile)
    wirte fileReader to request
  else
    forward request
  end if
end loop
```

C. Docker Engine

In our architecture, FID Agents are responsible for downloading images. P2P downloading is integrated in FID Agent, so the P2P logic is decoupled from Docker. In our design, the source code of Docker Engine has not been modified.

In *load mode*, FID Agent uses "docker load" command to import a tar archive into Docker Engine. Nothing is needed in this situation but running Docker Engine.

In *proxy mode*, we should configure the proxy address for Docker Engine. Restarting the docker daemon is required to make the configuration effected. Starting with Docker 1.12, Docker added a new feature named "live restore". Users can configure the Docker Engine so that containers remain running even if the Docker Engine is terminated. Before Docker 1.12, restarting Docker Engine will cause all the running containers to be killed. The version of Docker we used supports "live restore". So modifying the configure file of Docker and restarting Docker Engine does not affect the running container-based applications.

D. BT Tracker

BT Tracker is a core component in BitTorrent network. Tracker assists the communication between peers by using BitTorrent protocol. Once the tracker is down, the BitTorrent network will not work. For high availability of the service, we added multiple trackers in our system design. And these trackers' address will be written to torrent files when torrent files are generated. During the BitTorrent download, every BitTorrent client will announce to all trackers the resources it had. Even though some trackers were stopped, BT client still can get another peers' information from running trackers.

E. Storage

There are serval types of storage to support, such as file system, swift [15], gcs (Google Cloud Storage) [25]. In cross-IDC scenario, the distrusted file system Ceph [11] is recommended. Its multi-cluster synchronization feature can help synchronizing data between several IDCs, which is important for large-scale container deployment cross multi-IDCs. As known, bandwidth between IDCs is far less than inner-IDC. Multiple Ceph clusters can be deployed in multiple IDCs with several Registry in different IDCs. So Docker Engine can pull images from nearest Registry. It will reduce the traffic between multi IDC and reduce the time of distributing images.

Docker Registry can be easily configured and connected to Ceph via Swift interface without modifying any source code. [19]

VI. Experiment

In order to verify that our design and implements meet the design goals, we conducted experiments to evaluate FID. First we compared the performance of *load mode*, *proxy mode* and original *docker pull* in a single node. And then, we presented the performance of FID in large scale deployment. Furthermore, we analyzed the network traffic during P2P image distribution.

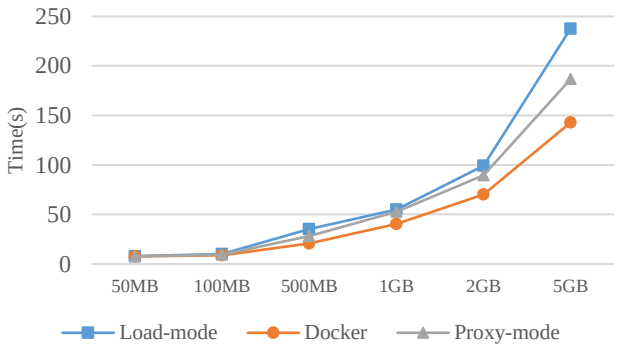

Fig. 4. Compare the performance of three ways when distributing Docker images

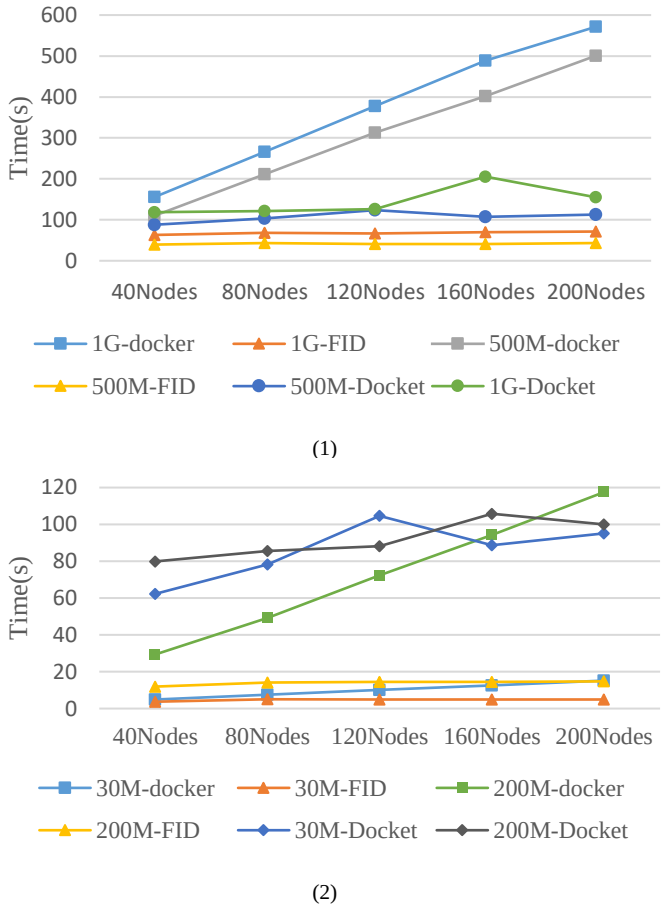

Fig. 5. The contrast of Docker, Docket and FID when distributing images of (1) 500M and 1GB, (2) 30MB and 200MB.

A. Distribution time of load mode and proxy mode

In our implements, we provided a FID Agent that could run in two modes. As we know, a P2P protocol brings extra communication in a network. So comparing the performance differences between P2P distribution and Docker native distribution is needed. And it is necessary to know the performance about *load mode* and *proxy mode* to make a choice between these two modes. The experiment is executed on a single node. Figure 4 shows the performance

of three solutions of distributing images. The x axis represents the size of the Docker image in experiment. And the y axis represents the time of distributing this image. We can see that in three ways, the distribution time an image grows as the image size grows. And native *docker pull* (named Docker in Figure 4) has the best performance. *Load mode* has the highest slope. *Proxy mode* is in the middle.

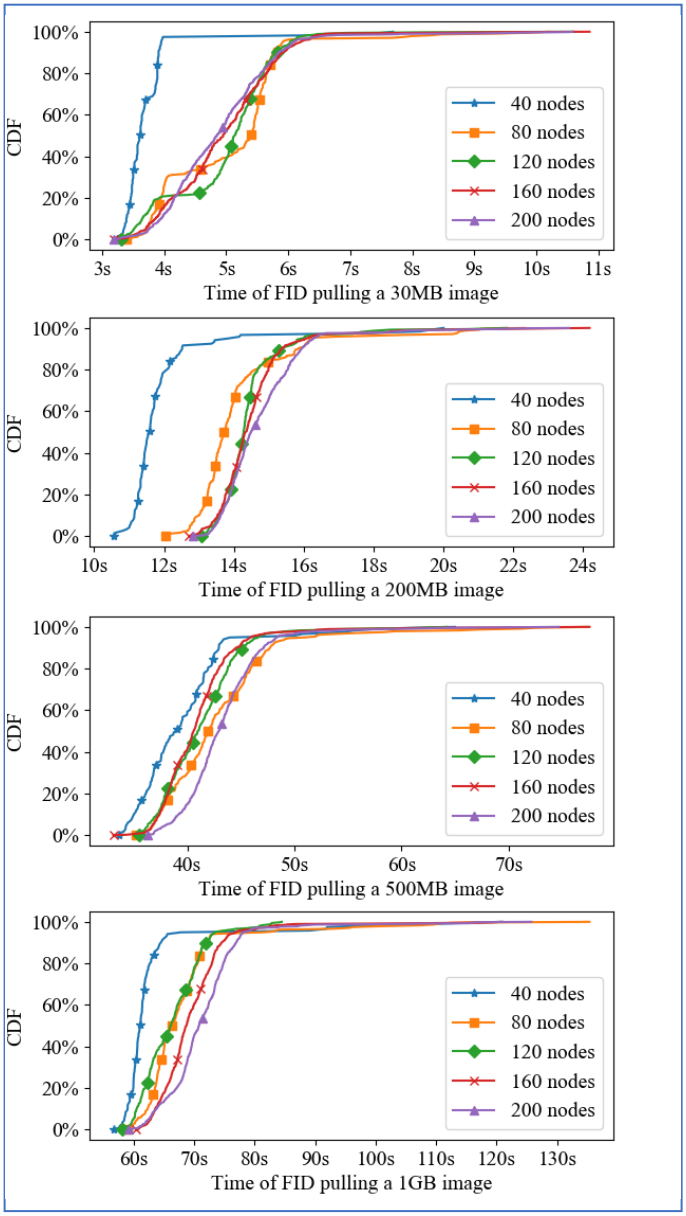

Fig. 6. The time of FID pulling an image in a distribution job, as cumulative distribution functions (CDFs)

In BitTorrent, peer often communicates with trackers and the other peers. Thus BitTorrent will bring extra overhead and it leads to BT-pulling, which costs more time than *docker pull*. Between *load mode* and *proxy mode*, *proxy mode* has a better performance. Because in *proxy mode*, FID Agent does not need to wait for all layers to be downloaded, then packages these layers into a tar archive. And in *proxy mode*, downloading and extracting can be done at the same time. For example, image X has two layers, layer A and layer B. And layer A is bigger than layer B. In *proxy mode*, the pulling time can be described as (1).

$$T_{proxy} = Max(D_A + E_A, D_B + E_B) \quad (1)$$

D_x means the downloading time of layer x, E_x means the extracting time of layer x. So we can see that the downloading of layer A and decompressing of layer B can be parallel. But in *load mode*, the pulling time is described by (2). $Max(D_A, D_B)$ equals to the time of downloading Blobs of image X. $T_{tar}(A, B)$ is the time of creating tarball from Blobs. $T_{load}(Tar_X)$ means the time of loading the tarball Tar_X to Docker Engine. The processes of downloading and loading is serial. So the *proxy mode* achieves better performance. And we run FID in *proxy mode* for large scale deployment.

$$T_{load} = Max(D_A, D_B) + T_{tar}(A, B) + T_{load}(tar_X) \quad (2)$$

B. *Distribution time in large scale deployment*

To evaluate distribution time of one image large scale deployment, we deployed FID on a cluster, which contains 200+ physical nodes. Each node consists 2 CPU, every CPU has 8*2.13Ghz cores, 64GB memory, and 1*300GB disk. The bandwidth between any two nodes is 1 Gbps. P2P Registry is deployed in the cluster and it serves 200 Docker clients. For the backend storage of registry, we used local file system to store the images. Four Docker images with different sizes are used for testing, i.e., “docker registry” (30 MB), “centos” (200MB), “hadoop” (500MB), and “tensorflow” (1GB). In every distribution test, we tested 3 times and calculated the average of distribution time. In the all experiments, we deployed our FID Agent in *proxy mode.*

Figure 5 shows the contrast between original distribution (named “docker” in the figure), FID distribution (named “FID” in the figure) and *Docket* (named “Docket” in the figure). In Figure 5, the x axis means the node number of deployment job and the y axis means the average time of distribution time. We can see that for *docker pull*, the time of distribution images increases as the number of nodes increases. And the growth of time in linear. But for FID, with the increasing number of nodes, distribution time did not increase significantly. The lines of FID are almost horizontal. Compared to *Docket*, FID has the better performance. When distributing a 500MB image (hadoop) in 200 nodes, distribution time using Docker native pulling method costs 500.77s and Docket costs 112.45s on average. But FID only costs 43.30s on average, and 96.5% of pulling task can be completed in 50s (shown in Figure 6). Although VMware

Harbor [8] has not provided their source code of P2P Registry, but their experiments data shows that Harbor distributes 500MB Docker image in 100 nodes costs about 180s. In the same scale, with the FID, the longest pulling time in a distribution job is less than 70s. Table I shows the reduction of distribution time using FID in different scale of distribution task. FID reduced 91.35% distribution time of using Docker native distribution. FID could save 83.50% of distribution time in a 200-scale distribution.

TABLE I. REDUCTION OF DISTRIBUTION TIME USING FID

Image Size	Time reduction of using FID				
	40 nodes	*80 nodes*	*120 nodes*	*160 nodes*	*200 nodes*
30MB	24.31%	33.52%	51.13%	60.65%	67.70%
200MB	59.41%	71.23%	80.05%	84.60%	87.45%
500MB	64.02%	79.68%	86.82%	89.82%	91.35%
1GB	59.47%	74.39%	82.41%	85.76%	87.52%

C. *Network traffic decrease of Docker Registry*

During image distribution, FID Agent could get data from another FID Agents. These data should be obtained from Docker Registry if we did not use FID in image distribution. So the traffic from peers could be considered as the traffic we saved for Docker Registry. We recorded the data traffic between P2P-peers in experiment B. Our statistics suggests that using BitTorrent during Docker images distribution can save a lot of traffic of Docker Registry. Figure 7 shows the traffic distribution in our experiment B. The blue part shows the traffic from FID Agent, the green part indicates the traffic from P2P Registry. Statistics shows that FID Agent can share as least 93% traffic of image distribution.

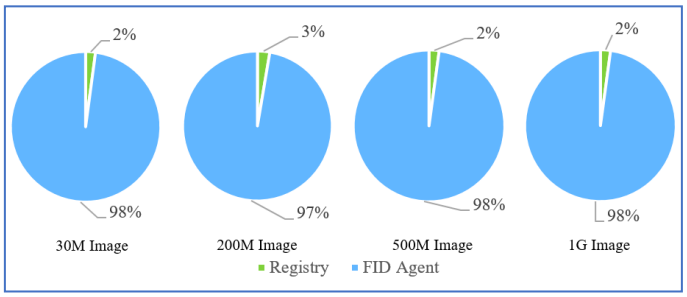

Fig. 7. The comparison of traffic that read from P2P Registry and FID Agent.

VII. CONCLUSION

In this paper, we proposed a system named FID, which can distribute Docker image in P2P network. Then, we analyzed the performance of FID. The experiments show that distributing Docker images in P2P way using FID will get a significant performance improvement., FID can reduce 91.35% distribution time when distributing 500MB image (hadoop) in 200 nodes, and save 93% network traffic of Docker Registry when distributing 200M-size image in the entire experiments.

Future works could be come up with a P2P traffic scheduling between different IDCs. In the current architecture, nodes in different IDCs share the same BitTorrent Trackers, and the peer selection algorithm is randomly choosing peers. We will study and develop the peer selection and traffic schedule solution in P2P algorithm. In practice, we found that extracting and importing time of Docker images also influences on the image distribution time. We will explore the acceleration of image extracting and importing to further reduce the time of distributing images in large scale deployment.

ACKNOWLEDGMENT

This work is supported by Tencent and PKU-Tencent Joint Innovative Lab. Tencent Cloud provided the compute nodes for our experiments. PKU-Finelab also has made a lot of contribution.

REFERENCES

[1] Merkel D. Docker: lightweight linux containers for consistent development and deployment[J]. Linux Journal, 2014, 2014(239): 2.

[2] Burns B, Grant B, Oppenheimer D, et al. Borg, omega, and kubernetes[J]. Communications of the ACM, 2016, 59(5): 50-57.

[3] "Docker Registry" https://docs.docker.com/registry/ .

[4] Schollmeier R. A definition of peer-to-peer networking for the classification of peer-to-peer architectures and applications[C]//Peer-to-Peer Computing, 2001. Proceedings. First International Conference on. IEEE, 2001: 101-102.

[5] Liu Y, Guo Y, Liang C. A survey on peer-to-peer video streaming systems[J]. Peer-to-peer Networking and Applications, 2008, 1(1): 18-28.

[6] Cohen B. Incentives build robustness in BitTorrent[C]//Workshop on Economics of Peer-to-Peer systems. 2003, 6: 68-72.

[7] "Docker Distribution" https://github.com/docker/distribution

[8] "Harbor" https://github.com/vmware/harbor

[9] "quayctl" https://github.com/coreos/quayctl

[10] "Spark" https://spark.apache.org/

[11] Weil S A, Brandt S A, Miller E L, et al. Ceph: A scalable, high-performance distributed file system[C]//Proceedings of the 7th symposium on Operating systems design and implementation. USENIX Association, 2006: 307-320.

[12] Maltzahn C, Molina-Estolano E, Khurana A, et al. Ceph as a scalable alternative to the hadoop distributed file system[J]. login: The USENIX Magazine, 2010, 35: 38-49.

[13] Shvachko K, Kuang H, Radia S, et al. The hadoop distributed file system[C]//Mass storage systems and technologies (MSST), 2010 IEEE 26th symposium on. IEEE, 2010: 1-10.

[14] "rkt: A security-minded, standards-based container engine" https://coreos.com/rkt

[15] "OpenStack Swift" http://swift.openstack.org/

[16] Tyler H. Brandom S., et al., Slacker: Fast Distribution with Lazy Docker Containers, 14th USENIX Conference on File and Storage Technologies (FAST'16), 2016.

[17] "Practice of Docker in Tencent Gaia" http://dockone.io/article/1555

[18] Hausenblas, M.: Docker Registries: The Good, the Bad & the Ugly, http://bit.ly/1OsrnIu

[19] Day, S.: Docker Registry V2 - A New Model for Image Distribution. In: Docker Con 2015. San Francisco (2015)

[20] "Fast Distribution of Docker Images using P2P method" http://dockone.io/article/1162

[21] "Quay Documentation" https://docs.quay.io/

[22] “Xiao, D.: Custom docker registry that allows for lightning fast deploys through” https://github.com/netvarun/docket

[23] “Docker load | Docker Documentation” https://docs.docker.com/engine/reference/commandline/load/

[24] Cohen, B.: Incentives build robustness in bittorrent. In: Workshop on Economics of Peer-to-Peer Systems, vol. 6, pp. 68–72 (2003)

[25] “Google Cloud Storage” https://cloud.google.com/storage

[26] Peng C, Kim M, Zhang Z, et al. VDN: Virtual machine image distribution network for cloud data centers[C]//INFOCOM, 2012 Proceedings IEEE. IEEE, 2012: 181-189.

[27] Reich J, Laadan O, Brosh E, et al. VMTorrent: scalable P2P virtual machine streaming[C]//CoNEXT. 2012, 12: 289-300.