2019 IEEE International Conference on Service-Oriented System Engineering (SOSE)

Comparison between Chunk-based and Layer-based Container Image Storage Approaches: an Empirical Study

Yan Li, Bo An, Junming Ma, Donggang Cao

Key Lab of High-Confidence Software Technology (Peking University), Ministry of Education, Beijing, China
$\{yan.l, anbo, mjm520, caodg\}$ @$pku.edu.cn$

***Abstract*—Container management frameworks, such as Docker, pack applications and their complex software environment in self-contained images, which facilitates application deployment, distribution, and sharing. Currently, container images are organized in a multi-layer file-system manner. For example, Docker uses AUFS to organize its container image storage. However, when a new version is created, all the changes are stored in a new layer via a Copy-On-Write mechanism, which can trigger a heavy burden for the network and storage. There are also other image storage approaches such as chunk-based technology. In this paper, we suggest that the container image storage system should not use one storage technology for all images, but should choose different technology for concrete images according to the image's distinct features. We design some experiments to compare the layer-based and chuck-based technology. And based on the analysis of the collected data, this paper gives some principles for the selection of image management techniques and discusses some interesting potential research topics.**

***Index Terms*—Docker, image, storage, casync, chunk**

I. Introduction

In the cloud computing era, the emergence of container technology has changed the process of software developing. Instead of configuring the environment from scratch, developers just need to pull the project image and start a container based on it. When they finish their developing, a new version of the image will be saved and published such that the service can be easily deployed [1]. Therefore, as the iteration of developing goes on, the image will be updated again and again. Currently, container images are organized in a multi-layer file-system manner. For instance, Docker [2] uses AUFS [3], a union file-system, to organize its image storage. When a container is created, a new layer of AUFS will be created and all the changes the user makes will be stored in this new layer.

However, because of the Copy-On-Write mechanism of AUFS, even a byte-size change can create a new copy of the changed file in the new layer. This can bring a heavy burden for the disk and network, especially in the environment of Joint Cloud [4]–[6] where inter-cloud data transfer is very cost-sensitive. Taking the scenario shown in Figure 1 as an example, the container image is organized as a four-layer AUFS system. Except for the top green layer, those three red layers are all immutable. So when a change happens on the red layers, the whole changed file must be copied to the green layer first. Even though the change can be just one byte, a whole new *1G.dat* file will be created, which seem to be unnecessary waste to the storage.

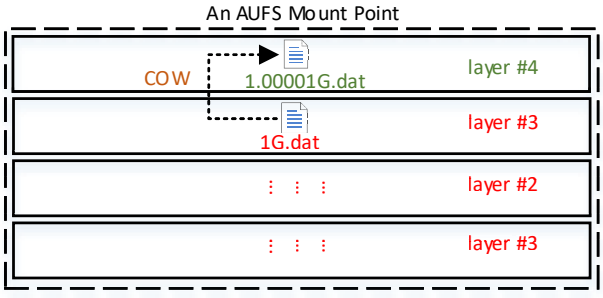


Fig. 1. AUFS Copy-On-Write Mechanism.

Recently there are also some works trying to use other container image storage techniques. Alban et al. [7] integrated *casync*, a chunk-based content synchronization tool, with *rkt* [8]. But their work was not applied in production environment because its performance was not stable. Sometimes the *casync*-integrated *rkt* has a better performance than the native layer-based solution but sometimes it is the opposite, which means each of these two approaches has its own scenes. So in order to make the image storage system more efficient, it is necessary to understand why one can behave better than another in a concrete situation.

In this paper, we argue that different storage approaches should be chosen for different container images. Specifically, layer-based storage approach may just be suitable for partial images while under other circumstances an alternative approach, such as chunk-based technique, might be a better choice. We make a statistical analysis and perform some experiments. Finally, we give some general guides on what container image storage approach should be used in what situation.

This paper is organized as follow: In Section I we introduce the basic idea of this paper. In Section II we discuss the background of Docker image management and some chuck-based techniques. In Section III we present our evaluation

978-1-7281-1442-2/19/$31.00 ©2019 IEEE
DOI 10.1109/SOSE.2019.00034

method. In Section IV we describe our experiments and analysis. In Section V we discuss some other topics beyond this paper. Section VI concludes this paper.

II. Background and Related Work

Image storage management methods have been concerned since the container technology became popular in software development and cloud computing fields. To well support iterating developing processes, container images are usually organized in a version-difference-aware manner, which means it supports developers switch the image version on demand like some version control toolkits. But unlike *git* [9], there always exists massive binaries in a container image. So instead of some text-based differences detection methods like *git diff* and *vimdiff* [10], traditional container technologies usually choose to control an image's version on a coarse-grained basis [11]. Recently some works also focus on some fine-grained container image version control methods such as chuck-based technique. In the next two subsections, these two different kinds of methods will be discussed in detail.

A. Traditional Container Image Management

Traditional container management frameworks, such as Docker, organize its image system as a layered file-system. Every Docker container is created based on a container image. An image contains all necessary files to run the container and consists of multiple read-only layers. A layer is a set of files which represents a part of the container file-system tree [12]. Layers are stacked on top of each other and joined to expose a single mount-point via a union file system such AUFS or OverlayFS. An image typically consists of several layers with sizes ranging from several KB to hundreds of MB.

The layered image structure supports layer-sharing across different containers. Docker identifies layers by hashing over their contents [12]. When multiple containers are created from images which contain identical layers, Docker does not copy those layers. Instead, the same layer is mounted into those different containers

Once a the image is fetched, the Docker daemon will create the container.First it unions all the read-only layers at a single mount-point and then creates a *writable* layer on top of it. Any changes made to the containers file system are stored in the top layer using a *copy-on-write*(COW) mechanism. It means that before changing a file from a read-only layer, it is copied to the top layer first and all changes are made to that copy as is presented before in Figure 1. Once the container exits, the writable layer is typically discarded unless a *docker commit* command is requested to create a new version of the image.

This whole layer-based image management method can well support inter-container image content sharing and new-version image publication. Nevertheless, the coarse-grained approach can be wasting sometimes. Except for the scenario shown in Figure 1, there are many other cases that file-grained COW will bring intolerable overhead. For instance,

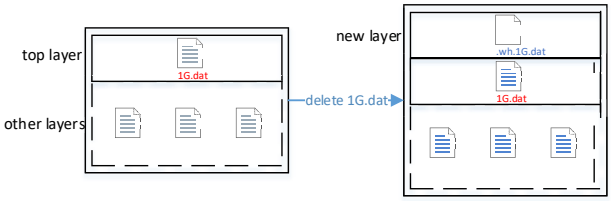


Fig. 2. A New Image is just a Partial Copy of the Old One.

when the newer version image deletes some files compared to the older version as is shown in Figure 2. In this case, the file *1G.dat* is not removed from the file-system actually. In fact, AUFS just simply creates a delete mark file called *.wh.1G.dat* to tell the file-system that this file should not display in the mount-point. If this new image is published to the registry and another host would create a container based on it, the totally irrelevant file *1G.dat* will be pulled from the registry, which will bring unnecessary overhead for the network and storage [13].

B. Fine-grained Container Image Management

Fine-grained container image management techniques mean that control an image's version with a much smaller management unit than a whole file. Chunk-based image version control is one of them. There are many popular chunk-based content delivery techniques. We will take *casync* [14] as an example next.

casync is a content-addressable data synchronization tool. It takes inspiration from the popular *rsync* file synchronization tool as well as the popular *git* revision control system. It combines the idea of the *rsync* algorithm with the idea of *git*-style content-addressable file systems, and creates a new system for efficiently storing and delivering file system images, optimized for high-frequency update cycles over the Internet. [15]

The whole *casync* work-flow of building chunks based on a block device or a file tree is shown as Figure 3. It can be summarized as three steps:

- First, a block device or a file tree will be serialized to a single stream, which is reserved to be segmented in the next step.
- Second, the whole stream is partitioned to several chunks. The size of each chunk can be variable.
- Last, indexes are generated by hashing each chunk, from which the original file tree can be recovered. Meanwhile, each chunk is compressed to a smaller chunk and all the chunks are nested together as the *chunk-store*.

Chunk-based content management system like *casync* also obeys the principle of Copy-On-Write like AUFS. But when a change happens, only those chunks whose hash values are different from the former version will be copied instead of

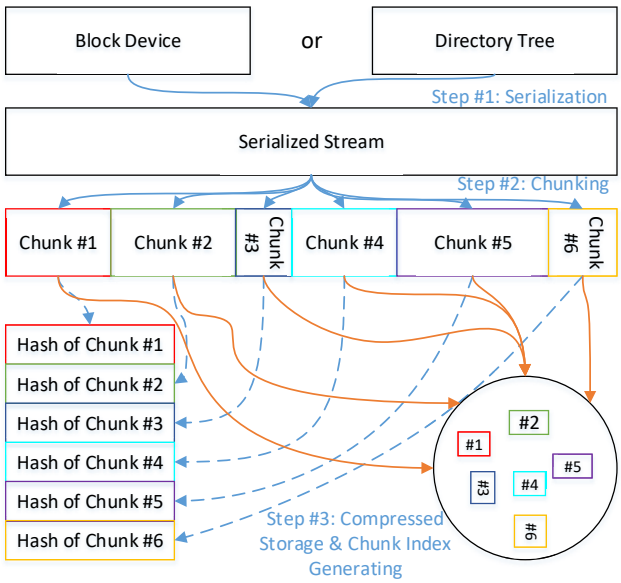


Fig. 3. Casync Workflow

the whole file. So in this situation, the storage increment will be:

$$storage\ increment = \sum_{changed} chunk\ size \qquad (1)$$

Equation (1) shows that the storage increment caused by a modification is actually determined by two factors — number of the changed chunks and the size of each changed chunk. And the increasing of each factor can affect the total storage increment. However, the number of changed chunks is positively related to the total chunk number while the size of each chunk is negatively related to it. If the total chunk number is *1*, there will always be one changed chunk but the chunk size can be the same as the whole serialized stream. If the chunk size is *1 byte*, the changed-chunk number will be huge and the index file will be big as well. So there is a trade-off between these two factors.

III. Evaluation Method

In order to give some guides on the selection of container image storage technologies, we design a reliable method to dig into the different storage-related performances of these two image management approaches.

Instead of building a prototype of a chunk-based container distribution system, we design a statistical simulating method for comparing the storage consumption of these two approaches. Since we can not collect the original metrics without a real chunk-based system, our idea was to reverse the chunk-based image retrieving process to get the corresponding data. In our design, we choose *docker pull* as the traditional image management method and *casync* as the chunk-based image management method. As for a list of images and their several versions, we collect the metrics using Algorithm 1. The detailed steps of Algorithm 1 is as follow:

1) Make sure the Docker image layer directory(e.g. */var/lib/docker/aufs/diff*) is clean.
2) Pull a version of this image.
3) In Fucntion *CalDockerCost*, pack all the new layers(for the first version, all the layers are new; for the next versions, only the layers downloaded in this iteration is considered to be new) to compressed files and sum their sizes up as the *docker_pull_cost*.
4) In Function *CalCasyncCost*, save the version of this image to a compressed file by executing *docker save* and use *casync* to create chunks based on it. Since *casync* create a new copy of only the changed chunks, the storage space increment of the chunk-store directory is the storage cost, which is marked as *casync_pull_cost*.
5) Go to the second step, pull a new version until traversing the entire version list. And if it is the last version of this image, clear the Docker image layer directory and start the next image until it is the last image in the list.

By running Algorithm 1, we collected the needed storage cost metrics of *docker pull* and *casync*. The detailed experiments will be presented in Section IV

IV. Experiments

We chose *50* popular Docker images on *docker hub* [16] and randomly selected *10* versions of each image. Then we ran Algorithm 1 to collect the metrics we needed. Based on the metrics dataset, we performed two kinds of experiments which are respectively general analysis and typical case study.

A. Data Preprocessing

The metrics dataset is in fact a *50x10* dimension matrix. However the image size can range from several KB to hundreds of MB, so we performed normalization on the data set. In order to let the first pulled version of each image be the same, we simply normalize the data according to (2).

$$data_(image_i, v_j) = \frac{raw\_data_(image_i, v_j)}{raw\_data_(image_i, v_0)} \qquad (2)$$

In fact, during the data preprocessing we observed that the $raw\_data_(v_0)$ of *docker pull* and *casync* is very close, which also proves that our design of retrieving metrics is reliable.

B. General Analysis

In the first part of our experiments, we analyze the overall data to observe some distribution characteristics. Based on the normalized dataset described above, we calculated the difference data between *docker pull* and *casync* by subtracting *docker_pull_cost* from *casync_pull_cost*. The miner the difference data is, the cost-saving effect of *casync* is more significant. Figure 4 shows a CDF based on the difference data. The green, blue and red lines respectively represent

Algorithm 1 Collect Storage Cost Metrics

Input: $ImageList$ Image and version list
Output: $Data$ Performance of *docker pull* and *casync*

function EXEC($command$)
  return execute $command$
end function

function CALDOCKERCOST($NewLayers$)
  $Size \leftarrow 0$
  for $Layer \in NewLayers$ **do**
    $Tar \leftarrow$ EXEC($pack\ layer$)
    $CurrSize \leftarrow$ EXEC($get\ size\ of\ Tar$)
    $Size \leftarrow Size + CurrSize$
  end for
  return $Size$
end function

function CALCASYNCCOST($Image, Version$)
  $OldSize \leftarrow$ EXEC($get\ size\ of\ chunk\ store\ dir$)
  EXEC($docker\ save\ image : version$)
  EXEC($build\ chunks\ of\ this\ version$)
  $NewSize \leftarrow$ EXEC($get\ size\ of\ chunk\ store\ dir$)
  return $NewSize - OldSize$
end function

$Data \leftarrow \{\}$
for $I \in ImageList$ **do**
  clean the image directory
  $Scaned \leftarrow \phi$
  $ImageData \leftarrow \{\}$
  for $V \in Image$ **do**
    EXEC($docker\ pull\ I : V$)
    $All \leftarrow$ EXEC($get\ all\ layers'\ name$)
    $New \leftarrow All - Scanned$
    $DockerCost \leftarrow$ CALDOCKERCOST(New)
    $CasyncCost \leftarrow$ CALCASYNCCOST(I, V)
    $ImageData\,[docker] \leftarrow DockerCost$
    $ImageData\,[casync] \leftarrow CasyncCost$
  end for
  $Data\,[I] \leftarrow ImageData$
end for

the maximum, minimum and average cost saving in all *50* image samples.

We made the accumulated ratio be fixed to *50%* first. The green line shows that the max-cost-saving is about *13.20%*. This indicates that quite a number of images benefit from *casync* a lot during the pulling of at least one version. The blue line shows that the average-cost-saving is about *0.58%*. This reflects that over a half of images benefit from *casync* if taking a general consideration. The orange line shows that the min-cost-saving is about *-1.83%*. At first the orange line goes really slowly, in fact horizontally, which means that in most images, there at least exists one version that *casync* cost more storage than *docker pull*.

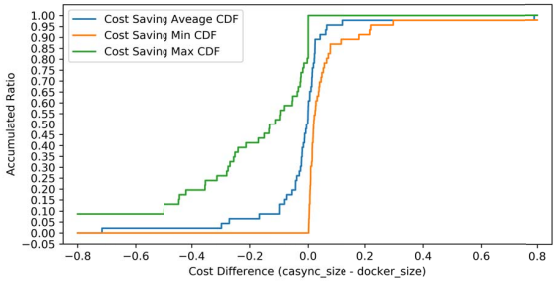


Fig. 4. Storage Cost Saving Distribution

We argue that *20%* of storage cost saving can be considered very significant. Especially when the image is very large, several GB for example, *20%* storage cost saving can prevent hundreds of MB irrelevant content from being pulled. So we fix the cost difference to *-0.2*. The green line shows that about *41.30%* images get a *20%* storage cost saving when pulling at least one version. In fact, if we put the fixed *x-point* to *0.00%*, the green line shows that *100%* of the images at least have a version that *casync* cost less storage. The blue line tells that about *6.52%* images get a *20%* storage cost saving on average. As is analyzed above, all of the orange line's data point is bigger than *0.00%* at the *x-axis*.

We can conclude the CDF as follow:

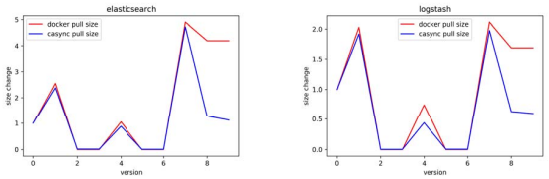
- About a half of the images can benefit from *casync* on an average basis.
- All images can benefit from *casync* during the pulling of at least one version, over *40%* of whom can get a *20%* cost saving.
- All images have at least one version that *docker pull* cost less storage space than *casync*.

So none of these two approaches has a better performance on a general basis. In fact, there are some patterns we could conclude that which one is better than another when given a typical image. We will discuss this in the next subsection.

C. Typical Cases Study

In this subsection we selected some representative image samples to dig into. Among these samples, some are suitable for *docker pull* while the others are suitable for *casync*.

- **Casync costs less because of the variation on large compressed files.** Figure 5 shows the cost comparison

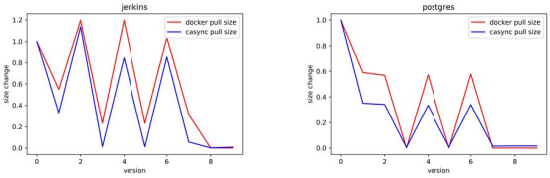


(a) elasticsearch. (b) logstash.

Fig. 5. Cost Comparison #1.

of *elasticsearch* and *logstash*(in fact *kibana* is also quite similar to them). The average cost saving of both of them are over *20%*. Taking *logstash* as an example, we tracked the layer variation during the pulling of these *10* versions. We found that those large margins between *casync* and *docker pull* are caused by changes on some big compressed files. For example, since a *java* project is usually packed to a *jar* file, even though the changes could be a small modification in a single *java* file, AUFS will take the whole *jar* as a new one. However in *casync* only the chunks related to the changed *java* files will be reconstructed. Other *java*-based images may have the same behaviour as the ELK family.

- **Duplicated files in two layers make *docker pull* less efficient.** Figure 6 shows the cost comparison

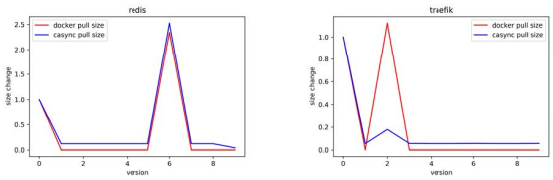


(a) jenkins. (b) postgres.

Fig. 6. Cost Comparison #2.

of *jenkins* and *postgres*. The average cost saving of both of them are over *10%*. Unlike the case of large *jar* files, too many duplicated files between two layers can also cause *docker pull* fetching additional content. Taking *postgres* as an example, we found that there could be many duplicated content between two layers that are in different versions. Ideally, the redundant content between layers should be merged in order to achieve higher performance. But an image may have thousands of versions such that no one can well organize them properly. Besides, during the developing of a fast-iterating project, changes frequently happen on duplicated files and no one will slow down to tidy the image, which will make the project image become larger and larger because of the redundant content. So in many situations, redundant is inevitable because of the massive versions on a public registry or the fast project iterating speed. In this case, *casync* can fix this issue as well.

- **Empty pulling causes extra overhead for *casync*** Figure 7 shows the cost comparison of *redis* and *traefik*. The red lines in these two samples both have some *zero* points(in *y-axis*), which indicates that *docker pull* does not cost any storage space at all. In fact we looked over the red lines of all the samples to find that each had at least one *zero* point. During the pulling of *zero*-point-version, all the layers needed has been pulled in the former versions so no new layer is needed, which means the layers this version needs is a subset of the already existing layers. This usually happens when rolling back a project form a newer version to an older



(a) redis. (b) traefik.

Fig. 7. Cost Comparison #3.

version. In this case, though *casync* may have generated the chunks based on the combination of other subsets, there still could be a few new chunks because of the variation of the files' meta-data or the structure of the file tree.

- ***casync* does not work well for changes on executable/lib files**. What should also be addressed in Figure 7 is that *redis* does not benefit much from *casync* in all the versions. We compared the file tree of those *redis* versions and found that the main changes happen on some compiled lib files or executable files. Unlike other binary files such as *tar*, the changes of executable/lib files are usually irregular, which means that though the changes of the source files may be small, the compiled files can have a totally different binary stream. In this case, *casync* has to reconstruct all the changed chunks, which may not cost less storage space than *docker pull*.

From these case studies, we can conclude some principles for the selection between layer-based and chunk-based technology:

1) Some files are **variation-sensitive**, which means there may not be a fixed simple pattern to follow their changes. For example, as is discussed above, even though the change in the source code file may just be a byte, the compiled executable file can be completely different. If the changes between two versions mainly happen on this kind of files, both layer-based and chunk-based approaches need to pull the complete content of the changed file. But the chunk-based solution also have some extra overhead in rebuilding the original image file tree structure. So in this situation, layer-based technology is a better solution for the image storage system.
2) Some files have a **simple incremental pattern** when changes happen. When changes between two version mainly happen on this kind of files, such as *tar* files mentioned above, chunk-based technology can just pull the change-relevant chunks, making the overhead of storage space as small.
3) When developing a **fast-iterating** project, the project image can always be under maintenance. The layer-based image storage system will make the project image prohibitively large because of the duplicated files in different versions. In this scenario, the chunk-based solution can reduce the cost of storage space

and free developers from tidying the project image.

4) When **rolling back** a project, usually no new content need to be re-pulled, which makes layer-based image storage solution a better alternative.

V. Discussions

A. Other Overhead Consideration

Except for the storage cost when pulling a new version of an image, there are many other overhead factors to consider. Though *casync* may cost less storage and network traffic, it will bring other overhead at the same time. For example, after pulling the new chunks and the index file, it has to rebuild the original image file tree by executing the reverse process of Figure 3, which can be time-consuming sometimes. Also the real network traffic should be monitored gracefully as well [11], [17]. So in order to use it in the production environment, there must be a trade-off among these factors.

B. Layer-grained Chunk Generating

Currently, we build the chunks and index file based on a whole image generated by *docker save*. In fact, the difference between two layers can be easier to capture by *casync* than two entire images. And by building chunks layer by layer, the small overhead described in IV-C can be eliminated as well.

C. Intelligent Image Characteristics Detection

For practical use, the image's file-system characteristics must be well detected to choose the corresponding image management method adaptively. Perhaps some machine learning or deep learning methods can help to achieve this.

D. Other Image Management Approaches

This paper just discusses one alternative image management method. There are many other popular image delivery and management approaches [18], [19] such as p2p [20], etc. So more work needs to be done in order to find what approach should be used in what scenario [21].

E. Statistical Analytics Only

Though we designed a reliable statistical method to collect the storage cost metrics during the pulling of Docker images, we did not build a chunk-based image management prototype to compare these two approaches on a practical basis. In the near future, we plan to do more engineering work to get more realistic metrics dataset.

VI. Conclusion

In this paper, we compare two alternatives for managing container images — chunk-based and layer-based technology. We make a statistical analysis to find out which approach is the better choice in some certain situations. The choosing between these two approaches depends on the image's file-system characteristics. For example, when a project is developed using *java* or there are too many duplicated files between layers in two different versions, chunk-based method could be a better image management choice.

For future work, we plan to address the limitations discussed in Section V. In particular, we plan to build a container image management prototype based on layer-grained chunk generating.

Acknowledgment

This work is supported by the National Key R&D Program of China (2016YFB1000105), the Science Fund for Creative Research Groups of China under Grant, No. 61421091.

References

[1] Merkel D. Docker: lightweight linux containers for consistent development and deployment[J]. Linux Journal, 2014, 2014(239): 2.
[2] 2018. Docker. (2018). https://www.docker.com/.
[3] 2018. AUFS. (2018). http://aufs.sourceforge.net. Accessed March 2018.
[4] Wang H, Shi P, Zhang Y. Jointcloud: A cross-cloud cooperation architecture for integrated internet service customization[C]//Distributed Computing Systems (ICDCS), 2017 IEEE 37th International Conference on. IEEE, 2017: 1846-1855.
[5] SHI P, WANG H, ZHENG Z, et al. Collaboration environment for JointCloud computing[J]. SCIENTIA SINICA Informationis, 2017, 47(9): 1129-1148.
[6] Cao D G, An B, Shi P C, et al. Providing virtual cloud for special purposes on demand in jointcloud computing environment[J]. Journal of Computer Science and Technology, 2017, 32(2): 211-218.
[7] ALban Crequy, et al. Exploring container image distribution with casync[C]//FOSDEM 2018.
[8] 2018. rkt - the pod-native container engine. (2018) https://github.com/rkt/rkt/.
[9] 2018. Git. (2018) https://github.com/git/git/.
[10] 2018. Vim. (2018) https://www.vim.org/.
[11] Harter T, Salmon B, Liu R, et al. Slacker: Fast Distribution with Lazy Docker Containers[C]//FAST. 2016, 16: 181-195.
[12] Zheng C, Rupprecht L, Tarasov V, et al. Wharf: Sharing Docker Images in a Distributed File System[C]//Proceedings of the ACM Symposium on Cloud Computing. ACM, 2018: 174-185.
[13] Anwar A, Mohamed M, Tarasov V, et al. Improving docker registry design based on production workload analysis[C]//16th USENIX Conference on File and Storage Technologies. 2018: 265.
[14] 2018. Casync. (2018). https://github.com/systemd/casync/.
[15] 2018. Casync Blog. (2018). http://0pointer.net/blog/casync-a-tool-for-distributing-file-system-images.html.
[16] 2018. Docker Hub. (2018). https://hub.docker.com/.
[17] Kangjin W, Yong Y, Ying L, et al. Fid: A faster image distribution system for docker platform[C]//Foundations and Applications of Self* Systems (FAS* W), 2017 IEEE 2nd International Workshops on. IEEE, 2017: 191-198.
[18] Nathan S, Ghosh R, Mukherjee T, et al. Comicon: A co-operative management system for docker container images[C]//2017 IEEE International Conference on Cloud Engineering (IC2E). IEEE, 2017: 116-126.
[19] 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.
[20] Reich J, Laadan O, Brosh E, et al. VMTorrent: scalable P2P virtual machine streaming[C]//CoNEXT. 2012, 12: 289-300.
[21] Tarasov V, Rupprecht L, Skourtis D, et al. In search of the ideal storage configuration for Docker containers[C]//Foundations and Applications of Self* Systems (FAS* W), 2017 IEEE 2nd International Workshops on. IEEE, 2017: 199-206.