TITLE:

Efficient Container Image Updating in Low-bandwidth Networks with Delta Encoding

AUTHOR(S):

Matsumoto, Naoki; Kotani, Daisuke; Okabe, Yasuo

CITATION:

Matsumoto, Naoki ...[et al]. Efficient Container Image Updating in Low-bandwidth Networks with Delta Encoding. 2023 IEEE International Conference on Cloud Engineering (IC2E) 2023: 1-10

ISSUE DATE:

2023

URL:

http://hdl.handle.net/2433/286295

RIGHT:

© 2023 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.; This is not the published version. Please cite only the published version. この論文は出版社版でありません。引用の際には出版社版をご確認ご利用ください。

京都大学学術情報リポジトリ
KURENAI 紅
Kyoto University Research Information Repository

京都大学 KYOTO UNIVERSITY
A Self-archived copy in
Kyoto University Research Information Repository
https://repository.kulib.kyoto-u.ac.jp
京都大学学術情報リポジトリ KURENAI 紅 Kyoto University Research Information Repository

Efficient Container Image Updating in Low-bandwidth Networks with Delta Encoding

Naoki Matsumoto
Kyoto University
Kyoto, Japan
mt2.naoki@inet.media.kyoto-u.ac.jp

Daisuke Kotani
Kyoto University
Kyoto, Japan
kotani@media.kyoto-u.ac.jp

Yasuo Okabe
Kyoto University
Kyoto, Japan
okabe@i.kyoto-u.ac.jp

***Abstract*—Containers are the technology for Linux to isolate execution environments. By distributing a container image, which is a collection of files contained in the container, users can use an execution environment that includes the necessary files and libraries. However, container images are tens to hundreds of megabytes in size and require many network resources to be transferred. Especially in low-bandwidth network environments like edge computing, frequent image updating can be difficult and affect other services' communication. In this paper, we propose a method to reduce the data size required for image updates using delta encoding. We use delta encoding to reduce data size and finish updating quickly, but generating and applying deltas is a time-consuming operation. Our method proposes DeltaMerging which enables faster delta generation by merging existing deltas, and Di3FS which applies deltas lazily. The proposed method reduces the data size required to update container images from 5 to 40% of that of existing methods. Also, the time required to generate and apply deltas is greatly reduced with DeltaMerging and Di3FS. Furthermore, the performance degradation of the application in the container was almost negligible.**

***Index Terms*—Container, Delta encoding, Edge computing**

I. Introduction

Containers are a technology that provides an execution environment isolated from the host. The independent root file system allows the container to run applications without modifying the host's root file system. In addition, users can distribute an environment with necessary libraries and applications installed in advance. It makes it easy to run applications in a wide variety of environments. The root file system which contains applications and libraries is called a container image. Users can distribute container images through registries that host container images.

Pull is an operation to download the container image and extract its root file system. If there is no container image locally or if the container image is updated, container hosts need to pull the container image. Container images can sometimes have a size ranging from tens to hundreds of megabytes, even if they are optimized. Therefore, pulling new images every time consumes a lot of network resources. Also, the time to pull increases as the size of the image increases. It is a challenge in container use. Serverless computing[1] requires containers to start fast not to degrade service quality. In edge and fog computing, where computing and network resources are severely constrained, pulling an image can take several minutes[2]. The increase in container image size prevents containers from various utilization.

To improve the pull, lazy-pulling[3][4][5][6][7][8] has been proposed. Lazy-pulling starts the container without pulling the entire container image. In these methods, the minimum number of files required to start the container is preferentially downloaded, and containers are launched. However, because these methods eventually pull the entire container image, the transferred data size is not reduced. Another method proposes reducing the transferred data size in container image updating. Starlight[9] utilizes local images and sends only updated files. This method transfers the entire updated files. Thus, even if only a tiny portion of the file is updated, the entire file must be transferred in this approach. In environments with limited network bandwidth, such as WANs[10] and cellular networks[11], it is necessary to reduce the size of transferred data as much as possible, and there is room to reduce it.

In this paper, we propose a method to reduce the data size and time required for updating container images with binary delta encoding. Delta encoding is a technique for reducing the size of data for updates. Binary delta encoding is a kind of delta encoding designed for binary files. It extracts a delta from files or contents. To apply binary delta encoding in container image updating, we need to consider the characteristics of container images. In delta encoding, generating and applying deltas for them take longer than simply compressing and decompressing them. Container images contain a large number of files. Thus, using delta encoding in that shape leads to increase time to pull in clients. Also, in edge computing and IoT contexts, containers will be deployed on both servers with rich computing resources and devices with poor computing resources, such as Raspberry Pi. We consider both environments and propose **DeltaMerging** and **Di3FS** to solve these problems. DeltaMerging speeds up the generation of client-tailored deltas by merging already generated deltas. Di3FS applies the deltas when reading updated files so that clients can use the updated container image without the time-consuming decompressing and applying all deltas.

We confirmed proposed method can reduce the size of the update bundle from 5 to 40% of that of file-based deltas[9]. We also performed evaluations not only in a server environment but also in a Linux board environment as IoT devices. The result shows that the performance degradation of the application

京都大学 KYOTO UNIVERSITY
A Self-archived copy in
Kyoto University Research Information Repository
https://repository.kulib.kyoto-u.ac.jp
京都大学学術情報リポジトリ KURENAI 紅 Kyoto University Research Information Repository

TABLE I
COMPARISON BETWEEN APPLICATIONS WITH DELTA ENCODING

	Content type	Granurarity of update	Protocol	When applying deltas
Web[12][13]	a few files	a file	batch	immediately
Firmware[14] or Application[15] Updating	a bundle with a few files	a bundle	batch	immediately
File Sync[16][17] Remote File System[18]	many files	a file	sequential	immediately
Container Image Updating	a bundle with many files	a bundle	batch	**lazily allowed**

in the container caused by the proposed method is limited and acceptable.

II. RELATED WORKS

A. Reduction in Data Size and Time Required for Pull

Several improvements have been proposed to reduce the size or time required to update container images. The reduction can be divided into two categories: reduction of the container image size and reduction by improving the pull itself.

One is to reduce the size of the container image itself. Some approaches to minimizing the size of the container image itself have been proposed[19][20][21]. They eliminate files that are no longer needed in the layer or are duplicated between layers. Skourtis et al.[19] formulated the problem of image size reduction based on the trend of files included in a layer and proposed a method to merge similar layers between container images. Lu et al.[20] proposed a method to detect operations that cause image bloat in the layer structure, such as deleting downloaded files in image builds. However, Starlight[9] pointed out that improving efficiency in a layer-based structure is difficult.

The other is an improvement to the pull itself. More precisely, it improves the operation of downloading and decompressing the tar.gz compressed container image. The approach is divided into three categories: not pulling container images, partially and lazily pulling container images, and using a dedicated system for updating container images.

Slacker[3], Cider[4], Wharf[5], and CernVM-FS[22] mount container images on shared storage or distributed filesystems. However, since these methods do not have container images on the host, they require a stable network connection and shared storage. Such approaches are not suitable for remote hosts in terms of availability. P2P-based methods[23][24][25][26][27] have also been proposed to take advantage of the fact that each host has its container image. These are not effective if network resources are limited. Because these approaches consume many network resources, the network becomes the bottleneck.

An approach called lazy-pulling exploits a container-specific characteristic to reduce pulling time. When starting a container, downloading and decompressing the container image accounts for 76% of the startup time, while the files needed to start the container account for 6.4% of the total image pulling gtime[3]. Lazy-pulling downloads files that are necessary to start the container preferentially. eStargz[8], FogDocker[6], and Gotanda et al.'s approach[7] employ lazy-pulling. These methods record essential files for container startup during container execution and optimize downloading with the record. eStargz creates and distributes container images using stargz[28]. stargz is a gz-compatible compression format and enables files to be read without decompressing entirely. It allows lazy-pulling container images to be used as conventional ones.

Starlight[9] is a method for achieving fast updates and startups, even in environments with limited network resources, by transferring only the updated files. This method compares container images on the local and remote. When a large file is updated, the entire file must be downloaded even if the change is small, resulting in an inflated update bundle size. Such an approach is undesirable in environments where network bandwidth is limited, such as a WAN[10], or on mobile networks[11]. To provide more efficient and fast updating, we must reduce the size of deltas. Also, Starlight generates deltas on demand according to the container image on the client. Such a design increases the load on the server side. This is true of binary delta encoding. In return for the smaller deltas, it increases the server's load more, so we need to consider a way to suppress it regarding when and how to generate deltas.

B. Content Distribution Using Delta Encoding

Table I compares applications with delta encoding and container image updating. These approaches improved the process of generating deltas to achieve a better compression ratio. WebExpress[12] and Banga et al.[13] have proposed extracting deltas for each URL-tagged content. Delta encoding is also used in file synchronization, such as rsync[16] and NetSync[17]. Servers calculate hash values for each chunk of the file and send them to clients. The client sends the unmatched chunks in the file to the server as deltas, allowing files to be synchronized with the minimum data transfer. LBFS[18] is a file system that uses delta encoding to utilize the client's local caches. It enables fast file operations even in low-bandwidth network environments.

These approaches in the Web and firmware or application updating handle a few files or a single bundle containing a few files. Thus the time to generate deltas is short or not considered. Different from them, container images have various files, such as executable binaries and compressed files. In container image updating, servers should generate deltas for many files fast to reduce updating time. An approach is needed

京都大学 KYOTO UNIVERSITY
A Self-archived copy in
Kyoto University Research Information Repository
https://repository.kulib.kyoto-u.ac.jp
京都大学学術情報リポジトリ KURENAI 紅 Kyoto University Research Information Repository

to generate deltas tailored to each client fastly. Additionally, container image updating has a unique characteristic: there is no need to apply all deltas when the updated images are used, and applying deltas lazily is allowed. This characteristic is useful for faster updating.

In terms of a protocol, file synchronization and remote file system methods use a sequential transfer protocol that uses a hash to detect different chunks between the client and server and transfers the deltas sequentially. The protocol is designed to work with non-versioned files. However, Chen et al.[9] have pointed out that such sequential protocol can cause performance degradation when provisioning container images. Also, in container image updating, each image is versioned. We can generate deltas in advance and bundle them into a single file. Therefore, the protocol in container image updating should be a batch transfer protocol that transfers all deltas as a single bundle.

III. Proposed Method

A. Requirements for Updating Container Images

There are many use cases for delta encoding, such as web and file synchronization. These applications are unsuitable for container image updating as described in Section II. Additionally, the following unique requirements exist when using binary delta encoding in container image updating.

1) Fast client-tailored delta generation
2) Fast applying deltas to large numbers of files
3) Considering both rich servers and poor IoT devices

The first point stems from the fact that the version of the old image held by the client varies from client to client. Servers need to provide deltas considering each client's old image version. Generating deltas takes time and it leads to increase server loads and updating time. In addition, as the size or number of files increases, the resources to generate deltas increase. Therefore, there is a need to make delta generation itself faster. The proposed method reduces the required time and load to provide deltas by merging deltas with **DeltaMerging**. The detail is described in Section III-E.

For the second point, applying the deltas at high speed is necessary to let the updated image available as quickly as possible. Typical container images are distributed as compressed files, so they need to be decompressed. From the third point, in clients with poor storage performance, the decompression of the downloaded image itself takes much time[2]. Therefore, the downloaded deltas must be available without decompression, and applying deltas must be fast. In the proposed method, this is achieved by two approaches. One is the delta format which enables files to be read without decompressing the entire image. The other is **Di3FS(DiFF File System)** which applies deltas lazily. Each approach is described in Section III-C and Section III-F.

B. Overview of the Proposed Method

Fig. 1 shows an overview of our proposed method. Deltas between two container images are bundled into a single file called *delta bundle*. A delta bundle consists of deltas and metadata. Its detail is described in Section III-C. A bundle with a delta bundle and metadata needed to launch containers is called *update bundle*. An update bundle is generated based on one or more delta bundles, and the client performs update operations with the update bundle.

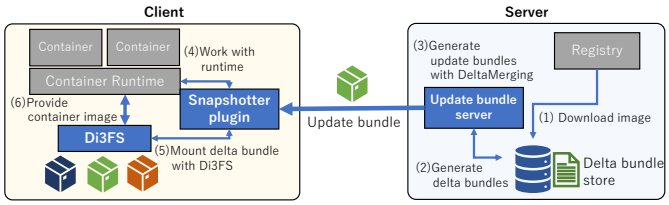

Fig. 1. The overview of the proposed method

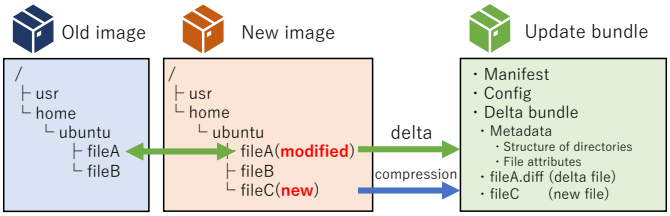

Fig. 2. The overview of delta bundle generation

The proposed method places components on the client and server sides for transferring update bundles.

The server consists of an update bundle server and a delta bundle store. The update bundle server generates and delivers update bundles in response to requests from clients, taking into account the version of the old images in the clients. The delta bundle store is a storage that stores delta bundles. According to the updating bundle generation strategy in Section III-D, delta bundles are generated in advance and stored in the delta bundle store. When a client requests an update bundle, a server generates the update bundle from delta bundles using DeltaMerging.

The client consists of the snapshotter plugin, which works with a container runtime and update bundle server, and Di3FS, which provides the files for containers with update bundles. Snapshotter plugin presents updated container images as a snapshot, and a container runtime can use the container images through Di3FS. Di3FS is a file system that provides new images based on an update bundle and is designed to provide updated container images without decompressing the update bundles. The new image is mounted using Di3FS with the provided update bundle, and the container can be launched.

Fig. 1 shows that the proposed method sends all deltas at once. As mentioned in Section II, the sequential transfer protocol used in file synchronization and remote file systems is unsuitable for updating container images. The proposed method uses a batch transfer protocol. The update bundle server generates an update bundle according to the client's old image version side and then transfers the entire of it to the client side.

京都大学 KYOTO UNIVERSITY
A Self-archived copy in
Kyoto University Research Information Repository
https://repository.kulib.kyoto-u.ac.jp
京都大学学術情報リポジトリ KURENAI 紅 Kyoto University Research Information Repository

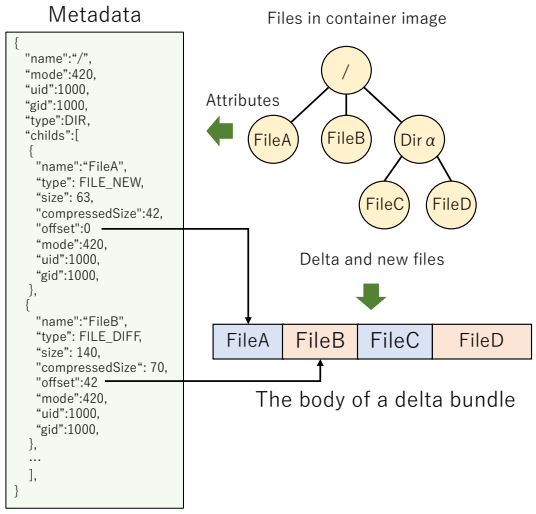

Fig. 3. The structure of a delta bundle

TABLE II
TYPES OF ENTRY IN METADATA

Type	Meaning	Content
FILE_NEW	Newly created file	Attributes and compressed file
FILE_SAME	Nothing modified file	Attributes
FILE_DIFF	Something modified file	Attributes and delta file
DIR	Directory	Attributes with its child
SYMLINK	Symlink	Attributes with real path

C. Delta Bundle Generation

Fig. 2 shows an overview of delta bundle generation from container images. The proposed method does not use the layer structure. The layers in the image are combined into a single layer and images are checked its updated files on a file-by-file basis.

The files in the old image are compared to those in the new image. The deltas between files obtained by bsdiff[29] are called *delta file*. If the file is newly created, the entire file is compressed and copied. The newly created file is called *new file*. A delta bundle has only files that exist in a new container image, and the deleted files are ignored. When generating a delta bundle, the directory structure of the files in the new image and file attribute information such as the name, size, permissions, and owner of each file are retained as metadata. As described in Section III-A, delta bundles must be available without decompressing them. In the proposed method (Fig. 3), the following two structures are employed, which are also used in existing methods[8][9]. First, the delta bundle's metadata retains attributes and offsets of the files. Second, the delta bundle has the metadata at the beginning, and the delta and new files are concatenated at the end. This design enables delta bundles to be a single file and access the contained delta and new files without decompressing the bundle.

The file types included in a delta bundle are listed in Table II. The metadata consists of these 5 file types and contains attributes for files provided by the delta bundle.

TABLE III
OPERATIONS FOR $Merge$ DELTA BUNDLES

Lower file	Upper file	Operation
Not existing SYMLINK FILE_SAME	Any file types	Copy the upper file
Any file types	SYMLINK FILE_NEW	Copy the upper file
Any file types	FILE_SAME	Copy the lower file
DIR	DIR	Recusively merge
Not DIR	DIR	Copy the upper file
FILE_NEW	FILE_DIFF	Apply upper diff to lower file and copy as FILE_NEW
FILE_DIFF	FILE_DIFF	**DeltaMerging**

D. Update Bundle Generation by Merging Delta Bundles

The proposed method generates update bundles that match the client's old image version by merging multiple delta bundles. One characteristic of updating using binary delta encoding is that the delta files require the old files. This characteristic is problematic for methods that generate the delta files in advance. Assuming that $\Delta_{(V_i,V_j)}$ is the delta bundle from $Image_{V_i}$ to $Image_{V_j}$. If the $Image_{V_1}$ exists locally in a client and updating to $Image_{V_3}$, there are three options.

The first is that a server transfers the delta bundles both $\Delta_{(V_1,V_2)}$ and $\Delta_{(V_2,V3)}$. However, the total size of the delta bundles can exceed the size of the new container image[30].

The second is to generate a new delta bundle $\Delta_{(V_1,V_3)}$ from $Image_{V_1}$ to $Image_{V_3}$. If the number of target versions increases, generating delta bundles when requested or generating all delta bundles in advance is not realistic in terms of the amount of delta generation.

The third method, which is employed in the proposed method, generates delta bundles $\Delta_{(V_i,V_{i+1})}$ in advance and generates update bundles by merging them. If the delta bundle between version V_i and version V_j is $\Delta_{(V_i,V_j)}$, the update bundle $U_{(V_i,V_j)}$ is formulated as follows.

$$\Delta_{(V_i,V_j)} = MERGE(i,j)$$
$$= \begin{cases} Merge(MERGE(i,j-1), \Delta_{(V_{j-1},V_j)}) \ (j > i+2) \\ Merge(\Delta_{(V_i,V_{i+1})}, \Delta_{(V_{i+1},V_{i+2})}) \ (j = i+2) \end{cases}$$
$$U_{(V_i,V_j)} = Pack(\Delta_{(V_i,V_j)})$$

In $Merge$, delta bundles are merged into a single delta bundle. The proposed method uses DeltaMerging to perform fast merging of delta bundles. In $Pack$, the metadata required to handle the update at container runtime is assigned to the delta bundle. The compressed metadata is combined at the beginning of the delta bundle, and the combined data of metadata and delta bundle is the update bundle.

Table III shows the operations for $Merge$ delta bundles. Files in an older delta bundle are called *lower files* and files in a newer delta bundle are called *upper files*. $Merge$ is performed on upper files. Each upper file is processed according to Table III and the result files are stored as the merged delta bundle. To merge FILE_DIFF files, DeltaMerging is used. The detail is described in Section III-E.

京都大学 KYOTO UNIVERSITY
A Self-archived copy in
Kyoto University Research Information Repository
https://repository.kulib.kyoto-u.ac.jp
京都大学学術情報リポジトリ KURENAI 紅 Kyoto University Research Information Repository

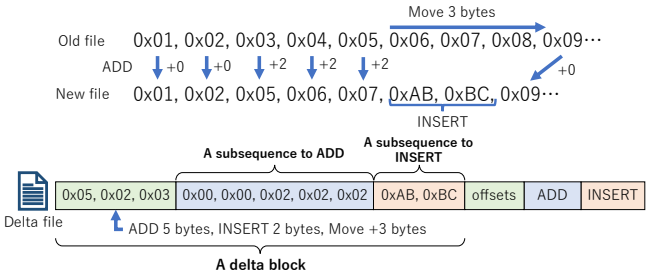

Fig. 4. Format of a delta file in bsdiff

E. Merging Delta Files with DeltaMerging

DeltaMerging is an algorithm designed for bsdiff to merge delta files into a single delta file like an algorithm[31] for VCDIFF. By merging existing delta files, DeltaMerging performs a delta file generation at high speed without regenerating delta files. Let $\delta^{FileA}_{(V_i,V_{i+1})}$ be the delta file for file $FileA$ contained in the delta bundle $\Delta_{(V_i,V_{i+1})}$. DeltaMerging is formulated as follows.

$$\delta^{FileA}_{(V_i,V_{i+2})} = DeltaMerging(\delta^{FileA}_{(V_i,V_{i+1})}, \delta^{FileA}_{(V_{i+1},V_{i+2})})$$

DeltaMerging merges two delta files. When three or more delta files need to merge, DeltaMerging is applied multiple times.

Fig. 4 shows the format of the delta file generated by bsdiff. In bsdiff, the delta is handled by three operations: adding (ADD), inserting (INSERT) subsequences to the old file, and moving the delta applying offset in the file. We refer to this set of operations as a delta block. The subsequence to be added or inserted is calculated so that the size of the delta file becomes as small as possible. In DeltaMerging, when an older delta file (lowerFile) is merged with a newer delta file (upperFile), the lowerFile is considered as the source file of the upperFile. The upperFile is applied to the lowerFile.

When merging, the delta blocks are converted into the structures shown below, and the entire delta file is treated as an array of delta blocks.

```
type DeltaBlock = struct {
  oldPos      int64
  newPos      int64
  addBytes    []byte
  insertBytes []byte
}
```

`oldPos` is the absolute offset in the old file where the subsequence is ADDed, and `newPos` is the absolute offset where the result of ADD is written to the new file. `addBytes` and `insertBytes` are the subsequences for ADD and INSERT.

DeltaMerging is performed after converting lowerFile and upperFile to arrays of DeltaBlock lowerBlocks and upperBlocks, respectively. DeltaMerging merges all delta blocks in upperBlocks with delta blocks in lowerBlocks. Algorithm 1 describes an algorithm to merge an upper block with lower blocks. Fig. 5 shows the example of DeltaMerging. Algorithm 1 takes a delta block in upperBlocks as upper. cur is the current offset for merge starting from 0. FIND_BLOCK gets the delta block lower that satisfies lower.newPos < upper.oldPos+cur < lower.insertEnd from lowerBlocks. Then, lower and upper are merged according to the rule described in Table IV. The merged delta block(mergeBlock) is stored in an array(merged). After merging all delta blocks

Algorithm 1 Merge delta blocks in DeltaMerging

Require: upper: a delta block to be merged
Require: lowerBlocks: an array of delta blocks for lowerFile
Ensure: merged: a merged delta block

```
function MERGEBLOCK(upper, lowerBlocks)
    merged ← NewArrayOfDeltaBlock() // Array for merged DeltaBlocks
    cur ← 0 // Current offset for merge
    mergeBlock ← NewDeltaBlock()
    // State is one of NotAdded, Added, Inserted and manages the state of mergeBlock.
    // NotAdded: mergeBlock is not initialized and nothing merged into it.
    // Added: Something merged as ADD into mergeBlock.
    // Inserted: Something merged as INSERT into mergeBlock.
    // Merged subsequence as ADD cannot be inserted into mergeBlock with state Inserted.
    // If the state is Inserted, mergeBlock is complete DeltaBlock and appended to merged.
    state ← NotAdded
    while cur is in the upper do
        lower ← FIND_BLOCK(lowerBlocks, upper.oldPos+cur)
        if state is Added then
            merged.append(mergeBlock)
            state ← NotAdded
        end if
        while cur is in the upper and the lower do
            if state is NotAdded then
                mergeBlock ← NewDeltaBlock()
            end if
            if lower is ADD at cur then
                if upper is ADD at cur then
                    if state is Inserted then
                        merged.append(mergeBlock)
                        mergeBlock ← NewDeltaBlock()
                    end if
                    mergeBlock ← Merge ADD regions as ADD
                    state ← Added
                else
                    mergeBlock ← Merge ADD and INSERT regions as INSERT
                    state ← Inserted
                end if
            else
                if upper is ADD at cur then
                    mergeBlock ← Merge INSERT and ADD regions as INSERT
                else
                    mergeBlock ← Merge INSERT and INSERT regions as INSERT
                end if
                state ← Inserted
            end if
            cur ← cur + merged length
        end while
    end while
    if state is not NotAdded then
        merged.append(mergeBlock)
    end if
    return merged
end function
```

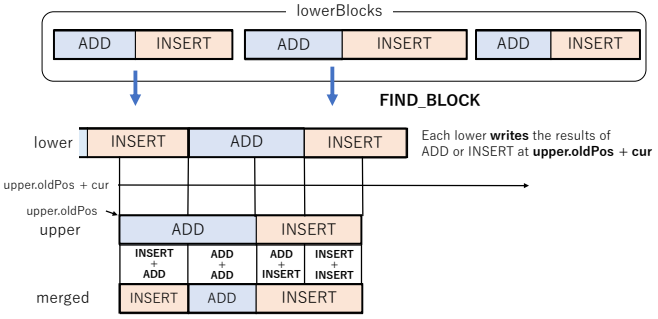

Fig. 5. The example of DeltaMerging

A Self-archived copy in
Kyoto University Research Information Repository
https://repository.kulib.kyoto-u.ac.jp

TABLE IV
OPERATIONS TO MERGE DELTA BLOCKS

lower	upper	Operation
ADD	ADD	Adding each subsequence and storing it as ADD
ADD	INSERT	Storing INSERT of upper
INSERT	ADD	Adding each subsequence and storing it as INSERT
INSERT	INSERT	Storing INSERT of upper

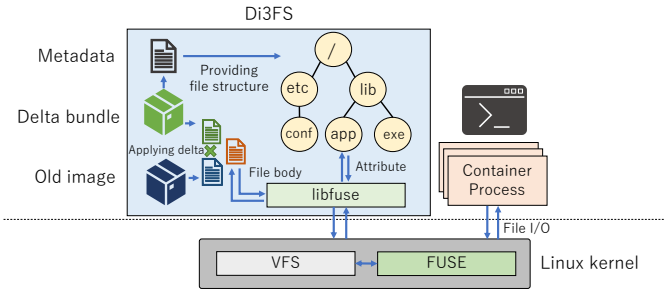

Fig. 6. The architecture of Di3FS

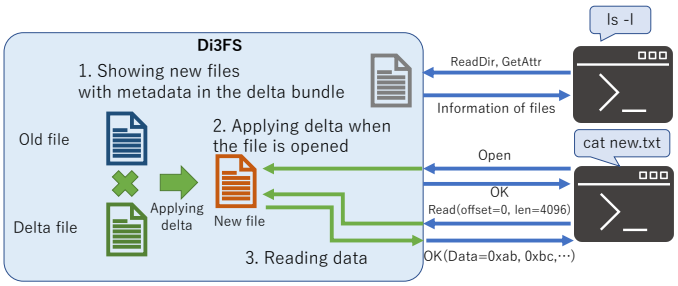

Fig. 7. Reading file with Di3FS

in upperBlocks, delta blocks in merged are converted into a merged delta file.

F. Di3FS

Di3FS (Fig. 6) is a file system for making a new image available with delta bundles by applying deltas lazily. In binary delta encoding, an operation to apply the deltas is required. The time required to apply deltas increases as the number or size of files increases. This tendency is particularly noticeable in environments with low storage performance. Di3FS provides the new image by applying the deltas lazily to reduce the time to apply deltas.

The flow from mounting the file system to reading files with Di3FS is shown in Fig. 7. Di3FS builds a tree of files and directories based on the delta bundle's metadata and outputs them as a file system. At the mounting, the file system only needs the attribute information of files and directories and does not need the file's content, so there is no need to apply the deltas. When a file is opened, the handler to apply deltas is called. The corresponding delta file is read from the delta bundle and applied, and the applied results are stored in memory. Later, the handler to read the file is called when the file is read. Based on the offset and length to read, the delta-applied data held in memory is read and returned.

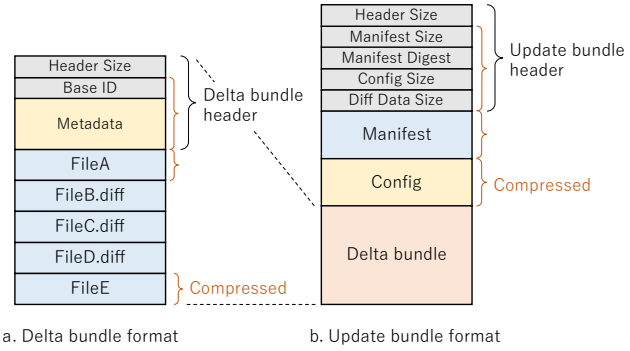

Fig. 8. Delta and update bundle format

Di3FS does not handle writing data and modifying file attribute operations. To store modified files, a writable layer is stacked over Di3FS. This layer stores written files and file attribute modification. This function is provided by a snapshotter plugin described in Section IV-B.

IV. IMPLEMENTATION

The implementation uses go-bsdiff[32], a bsdiff library for golang. The container runtime is containerd[33]. The implementation is available at https://github.com/naoki9911/d4c.

A. Delta and Update Bundle Format

Fig. 8.a shows delta and update bundle format. The delta bundle is designed to provide access to the included delta and new files while eliminating time-consuming image decompression operations. A delta bundle consists of metadata and file bodies. Metadata is a tree with directories as nodes and files and symlinks as leaves, encoded in JSON format. The file body section combines the deltas generated by bsdiff as delta files. Newly created files are compressed with zstd and combined into a delta bundle as new files.

The format of an update bundle is shown in Fig. 8.b. The update data includes the compressed Manifest, Config, and a delta bundle used for the update. Manifest is a file that describes the layers required to distribute a container image. The proposed method generates a Manifest with a single layer representing a delta bundle and appends it to the updated bundle. The Config specifies the environment variables and commands required to launch the container.

B. Snapshotter Plugin

containerd can handle not only the OCI container image but also any file system via snapshotter plugins as a *snapshot*. A snapshot is a bundle of files used for containers.

To provide snapshots, containerd requires a snapshotter plugin and appropriate settings. A snapshotter plugin manages snapshots and file systems which provide files in container images. The plugin provides only the functions for the snapshot, so a component to register images as snapshots is required. This component is implemented as a command line (CLI) tool. The CLI tool registers and configures snapshots to containerd with some metadata and notices the snapshotter plugin to

京都大学 KYOTO UNIVERSITY
A Self-archived copy in
Kyoto University Research Information Repository
https://repository.kulib.kyoto-u.ac.jp
京都大学学術情報リポジトリ KURENAI 紅 Kyoto University Research Information Repository

TABLE V
CONTAINER IMAGES USED FOR EVALUATION

Image	Version(Size)
postgres	13.1(109.44MB), 13.2(109.51MB), 13.3(109.62MB)
mysql	8.0.29(125.68MB), 8.0.30(127.69MB), 8.0.31(153.15MB)
redis	7.0.5(40.43MB), 7.0.6(40.44MB), 7.0.7(40.44MB)
nginx	1.23.1(54.14MB), 1.23.2(54.19MB), 1.23.3(54.25MB)
apache	2.4.52(53.80MB), 2.4.53(53.82MB), 2.4.54(54.47MB)

TABLE VI
SERVER ENVIRONMENT(VIRTUAL MACHINE)

CPU	AMD EPYC 7452(8 core)
Memory	32GB
Storage	SSD 200GB Sequential read 2GB/s Sequential write 700MB/s
OS	Ubuntu 22.04 (kernel 5.15.0-58-generic)

TABLE VII
LINUX BOARD ENVIRONMENT(RASPBERRY PI 4)

CPU	Broadcom BCM2835(4 core)
Memory	8GB
Storage	microSD 64GB Sequential read 40MB/s Sequential write 20MB/s
OS	Ubuntu 20.04 (kernel 5.4.0-1078-raspi)

mount file systems. In the proposed method, a snapshotter plugin mounts delta bundles with Di3FS, and the CLI tool operates container image updating and registers delta bundles as snapshots.

V. EVALUATION

A. Environments for Evaluation

Table V shows the container images used in the evaluation. These images are published at DockerHub[1]. Tags assigned to images are treated as versions.

Two evaluation environments are assumed as a client: Server Environment(Table VI) with rich computing resources and Linux board Environment(Table VII) for IoT or edge computing. The server is also used for the update bundle server. Virtual Machines(VMs) on a physical server are used as the server. An update bundle server and a client run on the same physical server as VMs. Each VM is connected to a virtual network provided by Linux. As the Linux board, Raspberry Pi 4 is used. An update bundle server and Raspberry Pi 4 are connected with physical 1Gbps Ethernet. Throughput and latency are controlled with Linux's traffic control (tc).

B. Delta Bundle Generation

We evaluated the time required to generate the delta bundles and their size. The same environment as the Server Environment is used to generate delta bundles. Starlight[9] compares updated files file-by-file and transfers the entire updated file. We denote this delta generation as *file-by-file delta* and implemented it in our approach by always generating FILE_NEW as deltas instead of FILE_DIFF.

[1]https://hub.docker.com/

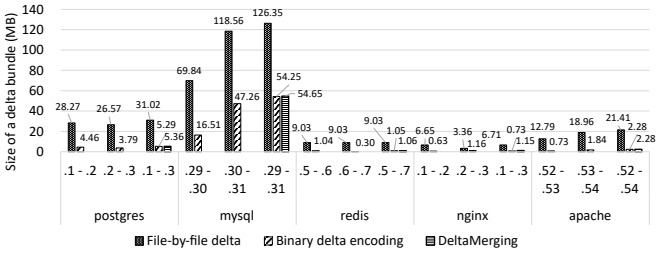

Fig. 9. Comparison of delta bundle size

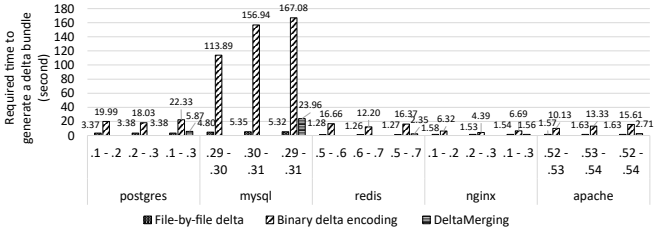

Fig. 10. Comparison of time required to generate delta bundles

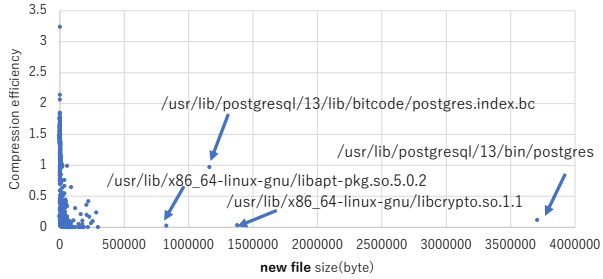

Fig. 11. Reduction efficiency per file

Fig. 9 compares the size of delta bundles between file-by-file delta and the proposed method. The proposed method reduces the size to about 5%-40% compared to the file-by-file delta.

Fig. 10 shows the time required to generate delta bundles. In most cases, the generation with file-by-file delta is completed in less than 5 seconds. On the other hand, the proposed method using binary delta encoding takes more than 10 seconds to generate delta bundles. In particular, for the MySQL image, it takes almost 180 seconds to generate its delta bundle. As for merging delta bundles with DeltaMerging, Fig. 10 shows that it is completed in about 10-20% of the time compared to the case of delta generation. In addition, as shown in Fig. 9, the size of the merged delta bundles is almost the same as that of the generated delta bundles.

Fig. 11 shows the efficiency of the proposed method between postgres 13.1 and 13.2. Compression efficiency is the ratio of delta file size over new file size. If the ratio is greater than 1, file-by-file delta is more efficient, and the proposed method is more efficient if the ratio is less than 1. Fig. 11 shows that executable binaries and shared libraries, such as postgres and libcrypto.so.1.1, have a significant

京都大学 KYOTO UNIVERSITY
A Self-archived copy in
Kyoto University Research Information Repository
https://repository.kulib.kyoto-u.ac.jp
京都大学学術情報リポジトリ KURENAI 紅 Kyoto University Research Information Repository

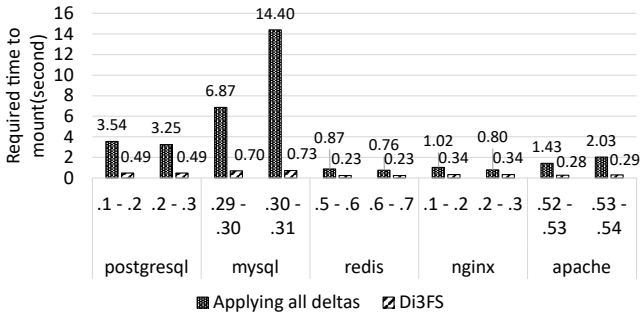

Fig. 12. Time required to apply an update bundle in Server Environment

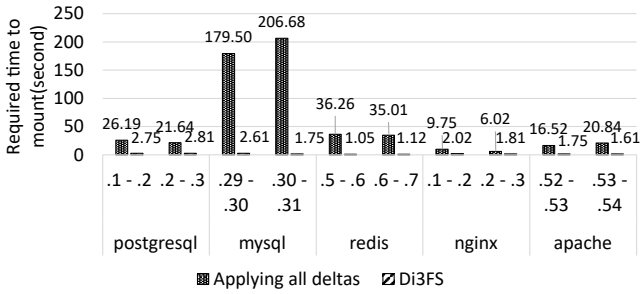

Fig. 13. Time required to apply an update bundle in Linux board Environment

reduction. On the other hand, there is almost no reduction for postgres.index.bc. The *.bc is LLVM bit-code encoded in bitstream. This is because bsdiff is designed to extract deltas for byte-encoded files and cannot extract changes efficiently for such bit-encoded files.

C. Applying Delta Bundles

We compared the time between fully applying delta bundles to the old container image and mounting delta bundles with Di3FS. The same delta bundles used for updating time evaluation are used in this evaluation. As shown in Fig. 12 and Fig. 13, applying delta bundles to an old container image takes more time depending on the size of the delta bundle, while the Di3FS takes at most 3 seconds to make the updated image available. Because Server Environment has a fast CPU and storage, applying the entire delta bundle takes about 10 seconds. However, since Linux board Environment has a slow CPU and storage, applying delta bundles takes more than 160 seconds, which is longer than the time required to transfer the update bundle. By using Di3FS, it is possible to mount the updated image in 3 seconds. Therefore, Di3FS is very useful for fast container image updating.

D. Time Required for Updating

We measured the time required to update a container image. The time is from when the container image update begins to when the downloaded update bundle is mounted. Results are averages of 10 times measurements We assumed Mobile Networks like 4G and 5G as a network between a server and a client. This network is a low-bandwidth network with 50Mbps[11] bandwidth and 40ms[34] delay (RTT).

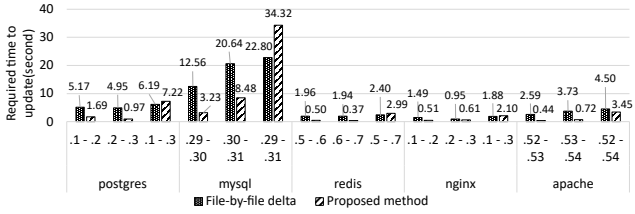

Fig. 14. Time required to update in Linux board Environment

TABLE VIII
TIME REQUIRED TO CHECK FILES IN POSTGRES 13.2

Methods	Time required (sec)
Di3FS	6.019
Native	0.234

TABLE IX
TPC-B BENCHMARK COMPARISON IN POSTGRES 13.2

Methods	Latency Average(ms)	Transactions Per Second
Di3FS	15.748	634.997
Native	15.747	635.037

Fig. 14 shows the results for Linux board Environment. Horizontal axes in the graph represent old and new image versions to be updated. For the update (V_i, V_{i+1}), the delta bundles are generated in advance. When the update bundle $U_{(V_i,V_{i+2})}$ is requested, $\Delta(V_i, V_{i+1}), \Delta(V_{i+1}, V_{i+2})$ are merged on the update bundle server. The result shows that the update time is reduced up to 20-40% of that of the file-by-file delta. In addition, the update time was reduced by up to 12 seconds. In the case of merging delta bundles $U_{(V_i,V_{i+2}}$, due to the merging overhead, the update time is a little worse than the file-by-file delta. As shown in Fig. 10, generating deltas with DeltaMerging is slower than file-by-file delta generation. This is the reason why the proposed method's updating time is worse than that of file-by-file. We also evaluated them in Server Environment and, it shows almost the same trend as the results in the Linux board Environment.

E. Impact on Application Performance

We evaluated the impact of the proposed method on file access performance using the diff command. It reads and compares files. We also evaluated the application performance. We used a benchmarking tool for database software pgbench. Each evaluation was performed in Server Environment.

We measured the time to compare files with diff command between the Di3FS mounted directory and the native file system's directory. The target container image is postgres 13.2, and the old image is postgres 13.1. As shown in Table VIII, we observed that Di3FS took 25 times longer than the native file system. This is because Di3FS needs to apply deltas when files are read for the first time.

We measured the impact of Di3FS on the performance of applications using pgbench. We compared the performance

京都大学 KYOTO UNIVERSITY
A Self-archived copy in
Kyoto University Research Information Repository
https://repository.kulib.kyoto-u.ac.jp
京都大学学術情報リポジトリ KURENAI 紅 Kyoto University Research Information Repository

of applications launched using Di3FS and those launched from the native file system using the official postgres 13.2 container image. pgbench is a benchmarking software for the database software PostgreSQL and can benchmark with loads that simulate real applications. The parameters of pgbench are the number of concurrent connections is 10, the measurement time is 120 seconds, and the other is default values. The results in Table IX show that the benchmark results do not indicate any significant performance degradation.

This result shows that Di3FS does not cause performance degradation to PostgreSQL. pgbench issues a series of queries which is specified as TPC-B[35]. The test says that "it is characterized by significant disk input and outputs, moderate system and application execution time. and transaction integrity." In other words, TPC-B consumes much disk IO and CPU resources to process queries and transactions. From the results of pgbench and the characteristics of TPC-B, applications with significant disk IO and CPU resource consumption will not be affected by Di3FS.

VI. Discussion

A. Accelerated Delta Bundle Generation with DeltaMerging

In order to speed up generating a delta bundle that matches the version of the old image, we propose DeltaMerging. We have confirmed that the delta bundle generation with DeltaMerging is faster than that with generating deltas in Fig. 10. In addition, the size of the delta bundle merged with DeltaMerging is almost the same as that of generated delta bundles. This shows that generating client-specific delta bundles at high speed is possible without increasing their size. This indicates that the update bundle generation strategy of the proposed method is realistic. However, as shown in Fig. 10, it takes 23.96 seconds, or 2.3 MB/s (18 Mbps), to retrieve 54.65 MB of delta bundle. When a delta bundle is transferred over a 4G network where the average throughput is 53Mbps[11], merging with DeltaMerging may not fully utilize the link capacity. Also, if the network is fast enough, the time required for transfer decreases, and merging time with DeltaMerging becomes dominant. In most cases that require merging deltas shown in Fig. 14, the proposed method is slower than file-by-file delta. We need to improve the process of merging delta bundles. Because DeltaMerging can be parallelized for each file in the delta bundle, it is possible to speed up the process by utilizing the computing resources on the server side.

Our prototype server implementation provides only simple delta management. A server receives the client's information and provides requested delta bundles. The current server implementation does not have a cache mechanism and merges delta bundles with DeltaMerging every time. In a sense, the evaluations in Fig. 10 and Fig. 14 are the worst case in our proposed method. Once the required delta is generated and cached, a server can serve the same delta without merging. To provide more efficient and faster container updating, we will work on more sophisticated delta bundle management.

Also, we will address the more efficient update in "version jumping" (e.g. update from V_1 to V_5). The current implementation simply merges bundles recursively as described in Section III-D. Denoting version difference as n (e.g. n is 4 from V_1 to V_5), generating deltas takes $O(n)$ times merging. When the delta with "version jumping" is requested, the time to provide deltas will increase linearly according to n. We have a plan to reduce merging costs with a divide-and-conquer merging strategy. It will reduce the merging costs to $O(log n)$.

These server-side improvements including faster merging, delta bundle management, and merging strategy are future work.

B. Breakdown of Delta Bundle Size Reduction

The reduction rates shown in Fig. 11 indicate significant reductions for the executable binaries and shared libraries, which are target of bsdiff. However, the delta size for LLVM IR bit code (.bc) was not reduced. This is because bsdiff used to obtain deltas processes data in bytes and cannot handle bit-wise deltas. Thus, the reduction effect was insignificant or counterproductive for bit-wise data and compressed files. For such files for which bsdiff is not suited, it is necessary to use a different delta encoding algorithm or to convert the file to a format suitable for bsdiff before extracting the deltas. Since container images contain various types of files, the size of update bundles can be further reduced by incorporating a framework that can handle multiple delta encoding algorithms into the proposed method and using them appropriately, depending on the file type.

C. Overhead with Di3FS

Di3FS, which reduces the overhead of applying deltas, confirmed that container images can be used with applying deltas lazily. Specifically, in environments with low storage performance, while it took nearly 3 minutes to apply all the deltas, Di3FS was able to mount the updated image in about 2 seconds. Di3FS plays an essential role in speeding up container image updates.

Although significant performance degradation was observed when reading many files from the updated container image, a benchmark targeting PostgreSQL confirmed little performance degradation. As described in Section V-E, the benchmark is designed to consume much disk IO and CPU resources to process queries and transactions. It means that applications that do not read many files from the container image are unaffected by Di3FS, even if they have significant disk IO and CPU resource consumption. In our proposed method, files in delta bundles are provided by Di3FS as a read-only layer. Applications in containers create files, and the files are stored in a temporal read-writable layer provided by a native file system. This is why the performance degradation is not discovered in the benchmark using pgbench.

However, as the number of containers using the same image increases, file read requests from those containers go through Di3FS. The delta-applied data is retained in memory, so the deltas are only applied once for the first time in Di3FS. Containers with the same images have almost the same tendency to read[3], so it is expected that Di3FS will not cause

京都大学 KYOTO UNIVERSITY
A Self-archived copy in
Kyoto University Research Information Repository
https://repository.kulib.kyoto-u.ac.jp
京都大学学術情報リポジトリ KURENAI 紅 Kyoto University Research Information Repository

more performance degradation as the number of containers increases. FUSE, which passes file-related operations from the Linux kernel to Di3FS, processes all file operations. It can cause performance degradation as the number of containers increases. This problem is not caused by Di3FS but by FUSE. Therefore, if FUSE causes performance problems or degradation, it can be replaced with other implementations.

VII. Conclusion

In this paper, we propose a method to reduce the amount of data transferred for container image updating using binary delta encoding. We show that the proposed method reduces the size of the update bundle, which bundles the deltas for the update, to about 5-40% of the size of the existing method[9]. The time required to update container images was also reduced. When updating container images using binary delta encoding, it is necessary to quickly generate the update bundle for each client. Clients need to make the updated image available as soon as possible. To address these issues, we proposed DeltaMerging and Di3FS, respectively. DeltaMerging makes it faster to generate a delta bundle by merging multiple delta bundles than generating delta bundles with binary delta encoding. Di3FS provides updated images applying deltas lazily. It eliminates the time-consuming operations of decompressing the delta bundle and applying deltas. We confirmed that the performance degradation of container applications by using the proposed method is limited and acceptable.

Further improvements to provide more sophisticated delta bundle management and reduce data size and updating time, including DeltaMerging and lazy-pulling, are our future tasks.

Acknowledgement

This work was supported by JSPS KAKENHI Grant Number 23KJ1329.

References

[1] E. Jonas, J. Schleier-Smith, V. Sreekanti, C.-C. Tsai, A. Khandelwal, Q. Pu, V. Shankar, J. Carreira, K. Krauth, N. Yadwadkar, J. E. Gonzalez, R. A. Popa, I. Stoica, and D. A. Patterson, "Cloud programming simplified: A berkeley view on serverless computing," 2019. [Online]. Available: https://arxiv.org/abs/1902.03383

[2] A. Ahmed and G. Pierre, "Docker container deployment in fog computing infrastructures," in *2018 IEEE International Conference on Edge Computing (EDGE)*, 2018, pp. 1–8.

[3] T. Harter, B. Salmon, R. Liu, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Slacker: Fast distribution with lazy docker containers," in *USENIX FAST 16*, Feb. 2016, pp. 181–195.

[4] L. Du, T. Wo, R. Yang, and C. Hu, "Cider: a rapid docker container deployment system through sharing network storage," in *2017 IEEE HPCC/SmartCity/DSS*, 2017, pp. 332–339.

[5] C. Zheng, L. Rupprecht, V. Tarasov, D. Thain, M. Mohamed, D. Skourtis, A. S. Warke, and D. Hildebrand, "Wharf: Sharing docker images in a distributed file system," in *ACM SoCC '18*, 2018, pp. 174–185.

[6] L. Civolani, G. Pierre, and P. Bellavista, "Fogdocker: Start container now, fetch image later," in *ACM UCC '19*, 2019, pp. 51–59.

[7] S. Gotanda and T. Shinagawa, "Short paper: Highly compatible fast container startup with lazy layer pull," in *2021 IEEE IC2E*, 2021, pp. 53–59.

[8] containerd, "estargz," https://github.com/containerd/stargz-snapshotter (2023/02/06 accessed), 2022.

[9] J. L. Chen, D. Liaqat, M. Gabel, and E. de Lara, "Starlight: Fast container provisioning on the edge and over the WAN," in *USENIX NSDI 22*, Apr. 2022, pp. 35–50.

[10] V. Persico, A. Botta, P. Marchetta, A. Montieri, and A. Pescapé, "On the performance of the wide-area networks interconnecting public-cloud datacenters around the globe," *Computer Networks*, vol. 112, pp. 67–83, 2017.

[11] X. Yang, H. Lin, Z. Li, F. Qian, X. Li, Z. He, X. Wu, X. Wang, Y. Liu, Z. Liao, D. Hu, and T. Xu, "Mobile access bandwidth in practice: Measurement, analysis, and implications," in *ACM SIGCOMM '22*, 2022, pp. 114–128.

[12] B. C. Housel and D. B. Lindquist, "Webexpress: A system for optimizing web browsing in a wireless environment," in *ACM MobiCom '96*, 1996, pp. 108–116. [Online]. Available: https://doi.org/10.1145/236387.236416

[13] G. Banga, F. Douglis, and M. Rabinovich, "Optimistic deltas for www latency reduction," in *USENIX ATEC '97*, 1997, p. 22.

[14] H. Teraoka, F. Nakahara, and K. Kurosawa, "Incremental update method for in-vehicle ecus," *IPSJ transactions. CDS*, vol. 7, no. 2, pp. 41–50, may 2017.

[15] N. Samteladze and K. Christensen, "Delta++: Reducing the size of android application updates," *IEEE Internet Computing*, vol. 18, no. 2, pp. 50–57, 2014.

[16] A. Tridgell and P. Mackerras, "The rsync algorithm," *The Australian National University*, pp. 1–8, 1996.

[17] W. Xia, C. Wei, Z. Li, X. Wang, and X. Zou, "Netsync: A network adaptive and deduplication-inspired delta synchronization approach for cloud storage services," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 10, pp. 2554–2570, 2022.

[18] A. Muthitacharoen, B. Chen, and D. Mazières, "A low-bandwidth network file system," in *ACM SOSP '01*, 2001, pp. 174–187.

[19] D. Skourtis, L. Rupprecht, V. Tarasov, and N. Megiddo, "Carving perfect layers out of docker images," in *USENIX HotCloud 19*, Jul. 2019, pp. 1–8.

[20] Z. Lu, J. Xu, Y. Wu, T. Wang, and T. Huang, "An empirical case study on the temporary file smell in dockerfiles," *IEEE Access*, vol. 7, pp. 63 650–63 659, 2019.

[21] N. Zhao, H. Albahar, S. Abraham, K. Chen, V. Tarasov, D. Skourtis, L. Rupprecht, A. Anwar, and A. R. Butt, "DupHunter: Flexible High-Performance deduplication for docker registries," in *USENIX ATC 20*, Jul. 2020, pp. 769–783.

[22] N. Hardi, J. Blomer, G. Ganis, and R. Popescu, "Making containers lazy with docker and cernvm-fs," in *Journal of Physics: Conference Series*, vol. 1085, no. 3. IOP Publishing, 2018, pp. 1–7.

[23] Linux Foundation, "Dragonfly," https://d7y.io/ (2023/02/06 accessed), 2022.

[24] uber, "kraken," https://github.com/uber/kraken (2023/02/06 accessed), 2022.

[25] W. Kangjin, Y. Yong, L. Ying, L. Hanmei, and M. Lin, "Fid: A faster image distribution system for docker platform," in *2017 IEEE FAS*W*, 2017, pp. 191–198.

[26] A. Verma, L. Pedrosa, M. R. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes, "Large-scale cluster management at Google with Borg," in *EuroSys '15*, Bordeaux, France, 2015, pp. 1–17.

[27] S. Becker, F. Schmidt, and O. Kao, "Edgepier: P2p-based container image distribution in edge computing environments," in *2021 IEEE IPCCC*, 2021, pp. 1–8.

[28] Google, "Crfs: Container registry filesystem," https://github.com/google/crfs (2023/03/26 accessed), 2019.

[29] C. Percival, "Naive differences of executable code," https://www.daemonology.net/papers/bsdiff.pdf (2023/02/06 accessed), pp. 1–3, 08 2003.

[30] A. C. G. Mennucci, "3. debdelta-upgrade service," http://debdelta.debian.net/html/x65.html#no_incremental (2023/04/18 accessed), 2011.

[31] R. Kiyohara, K. Tanaka, and Y. Terashima, "S/w upgrade for on-vehicle information devices," in *2012 IEEE ICCE*, 2012, pp. 19–20.

[32] icedream. (2022) go-bsdiff. https://github.com/icedream/go-bsdiff.

[33] The Linux Foundation. (2023) containerd - An industry-standard container runtime with an emphasis on simplicity, robustness and portability. https://containerd.io/.

[34] B. Varghese, E. de Lara, A. Y. Ding, C.-H. Hong, F. Bonomi, S. Dustdar, P. Harvey, P. Hewkin, W. Shi, M. Thiele, and P. Willis, "Revisiting the arguments for edge computing research," *IEEE Internet Computing*, vol. 25, no. 5, pp. 36–42, 2021.

[35] R. J. Hanson, "TPC-B," https://www.tpc.org/tpcb/ (2023/02/06 accessed), 1990.