Efficient Service Handoff Across Edge Servers via Docker Container Migration

Lele Ma
College of William and Mary
Williamsburg, VA
lma03@email.wm.edu

Shanhe Yi
College of William and Mary
Williamsburg, VA
syi@cs.wm.edu

Qun Li
College of William and Mary
Williamsburg, VA
liqun@cs.wm.edu

ABSTRACT

Supporting smooth movement of mobile clients is important when offloading services on an edge computing platform. Interruption-free client mobility demands seamless migration of the offloading service to nearby edge servers. However, fast migration of offloading services across edge servers in a WAN environment poses significant challenges to the handoff service design. In this paper, we present a novel service handoff system which seamlessly migrates offloading services to the nearest edge server, while the mobile client is moving. Service handoff is achieved via container migration. We identify an important performance problem during Docker container migration. Based on our systematic study of container layer management and image stacking, we propose a migration method which leverages the layered storage system to reduce file system synchronization overhead, without dependence on the distributed file system. We implement a prototype system and conduct experiments using real world product applications. Evaluation results reveal that compared to state-of-the-art service handoff systems designed for edge computing platforms, our system reduces the total duration of service handoff time by 80%(56%) with network bandwidth 5Mbps(20Mbps).

CCS CONCEPTS

• **Networks** → **Cloud computing**; *Wide area networks*; *Cyber-physical networks*; *Mobile networks*; • **Software and its engineering** → **Software infrastructure**; **Virtual machines**; **Cloud computing**; *File systems management*; • **Computing methodologies** → Virtual reality;

KEYWORDS

Docker Migration, Edge Computing, Offloading Services, Union File System

ACM Reference format:
Lele Ma, Shanhe Yi, and Qun Li. 2017. Efficient Service Handoff Across Edge Servers via Docker Container Migration. In *Proceedings of SEC '17, San Jose, CA, USA, October 2017,* 13 pages.
https://doi.org/10.1145/3132211.3134460

SEC '17, October 2017, San Jose, CA, USA

ACM ISBN 978-1-4503-5087-7/17/10…$15.00
https://doi.org/10.1145/3132211.3134460

1 INTRODUCTION

Edge computing has attracted lots of attention both from industry and academia in recent years [5, 14–16, 26, 32–34, 36–38]. By placing resource-rich nodes in close proximity to mobile or Internet of Things (IoT) devices, edge computing offers more responsive services, along with higher scalability and availability than traditional cloud platforms [26, 32]. To take advantage of nearby resources, computation offloading techniques have been playing an important role [1, 9, 20, 22].

In edge computing environments, offloading computation to the nearest edge server is key to cutting network latency, and improving the user experience. However, when a mobile device moves away from its current offloading server, network latency will increase, significantly deteriorating offloading service performance. Ideally, when the user moves, the edge server offloading service should adapt, and move to the nearest edge server in order to maintain highly responsive service. Therefore, migration of the offloading service from the current edge server to an edge server nearer to the user is an important activity in the edge computing infrastructure.

There are several approaches to migrating offloading services. VM handoff [12] has been proposed to accelerate service handoff across offloading edge servers. It divided VM images into two stacked overlays based on Virtual Machine (VM) synthesis [33] techniques. The result is that the mobile device only needs to transfer the VM top application overlay to the target server instead of the whole VM image volume. However, considering that the total transferred size is usually on order of tens or hundreds of megabytes, total handoff time is still relatively long for latency sensitive mobile applications. For example, migrating OpenFace [1], a face recognition application for wearable devices, will cost up to 247 seconds on a 5Mbps wide area network (WAN), barely meeting the requirements of a responsive user experience. Furthermore, VM image overlays are hard to maintain, and not widely available due to limited support and deployment in the real world.

In contrast, the wide deployment of Docker platforms raises the possibility of high speed offloading service handoff. As a container engine, Docker [19] has gained increasing popularity in industrial cloud platforms. It serves as a composing engine for Linux containers, where an application runs in an isolated environment based on OS-level virtualization. Docker's storage driver employs layered images inside containers, enabling fast packaging and shipping of any application as a container. Many container platforms, such as OpenVZ[24], LXC[28], and Docker [2, 10], either completely, or partially support container migration, but none of them are suitable for the edge computing environment. Migration within the official release of OpenVZ [10] eliminates file system transfer using distributed storage. However, due to the geographically distributed

nature of edge nodes, and heterogeneity of edge network devices, it is hard to implement distributed storage over WAN that can meet the latency requirements of edge applications. LXC migration [28] and Docker migration [2, 10] are based on CRIU[8], but need to transfer the whole container file system during the migration, resulting in inefficiency and high network overhead.

In exploring an efficient container migration tailored to edge computing, we focus on reducing the file system transfer size, a technique complementary to previous efforts. Docker's layered storage supports copy-on-write (COW), where all writable application data is encapsulated into one thin layer on top of base image layers. Thus, we only need to transfer the thin top writable layer from each container during migration. Additionally, the Docker infrastructure relies upon widely available public cloud storage to distribute container images (such as Docker Hub[17] and many other self-hosted image hubs). Therefore, application container base image layers can be downloaded from public cloud storage sources before each migration. We argue that taking advantage of the Docker container layered storage has the potential to reduce file system transfer size. To the best of our knowledge, there is no container migration tool that takes efficiently leverages the layered file system.

In this paper, we propose to build an efficient service handoff system based on Docker container migration. The system leverages the Docker container's layered file system to support high speed migration of offloading services within the edge computing environment. There are several challenges to overcome:

First, we need to understand the details of Docker container image layer management. There is little previous work investigating image management inside Docker. In order to provide a systematic birds-eye-view of Docker storage management, we investigate and summarize the layered images based on AUFS storage driver in 3.1.

Second, we need to take advantage of Docker's layered images to speed up migration of containers. Within AUFS storage, we found that Docker creates a new SHA256 number as local identification for each image layer downloaded from the cloud. As a result, if two Docker hosts download the same image layer from the same storage repository, these layers will have different reference identification numbers. This technique was originally a safety mechanism to avoid image overlapping across Docker hosts[18]. However, when we migrate a container from one Docker host to another, we must recognize that those image layers with different local identification numbers are actually the same content downloaded from the same storage location. This is necessary to avoid transferring redundant image layers during the container migration.

Third, we need to reduce data transferred during migration by recognizing that this data includes both the file system as well as checkpointed binary memory images. Although the total data size is reduced dramatically by leveraging Docker's layered storage, we still need to reduce the memory image size in order to further reduce total transfer time.

Lastly, we need to find a way to stringently maintain or reduce user-experienced latency during container migration across different edge servers. User-experienced latency could be sustained, and shorter, than the actual migration time through a well designed migration process strategy. Ideally, our goal is a seamless service handoff wherein users cannot notice that their offloading edge server has been changed.

To mitigate these challenges, we propose a framework that enables high speed offloading service handoff based on Docker container migration. By only encapsulating and transferring the thin writable *container layer* and its incremental runtime status, we reduce the total handoff time significantly. We make the following contributions in this paper:

- We have investigated the current status of techniques for Docker container migration. We evaluated the performance of state-of-the-art work, and find that no current work has leveraged the layered storage to improve migration performance.
- We have analyzed storage management of Docker based on the AUFS storage driver. Based on our analysis, we propose a novel method which leverages layered storage to speed up container migration.
- We have designed a framework that enables efficient handoff of offloading services across edge servers via high speed migration of containers by leveraging the layered storage on Docker platforms. This framework ensures low end-to-end latency of resource-intensive offloading services while maintaining the high mobility of clients.
- We have implemented a prototype of our system and conducted extensive evaluations. We have measured container handoff times on real world product applications. Our evaluation shows that the speed up ratio using our framework is 80%(56%) under 5Mbps(20Mbps) with 50ms network latency.

2 MOTIVATIONS

In this section, we seek to answer the following questions: Why do edge applications need offloading of computation? Why is service handoff needed in edge computing? Why is migration needed for service handoff? Why do we seek to perform service handoff via container migration?

2.1 Emerging Applications in Edge Computing Call For Computation Offloading

With the rapid development of edge computing, many researchers have constructed applications to take advantage of the edge computing platform.

One such example is Augmented Reality (AR). AR applications on mobile devices overlay augmented reality content onto objects viewed with device cameras. Edge servers can provide local object tracking, and local AR content caching [15, 16, 26, 33]. The Gabriel platform [13] was proposed within the context of wearable cognitive assistance applications using a Glass-like wearable device, such as Lego Assistant, Drawing Assistant, or Ping-pong Assistant. Those applications need to react to user actions in real time, following predefined guidance, or guidance from crowd-sourced videos.

OpenFace[1] is a real-time mobile facial recognition application that offers high accuracy along with low training and prediction times. The mobile client sends captured pictures from the camera to a nearby server. The server is running a facial recognition service that sends symbolic feedback to the mobile client in real time.

More edge applications can be found in [32, 36, 37]. All of these edge applications need to offload intensive computations (e.g., machine learning, computer vision, signal processing, etc.) to the edge server in order to achieve real time response. However, there we face several challenges before proceeding to deploy computation offloading services in the edge computing architecture.

2.2 Effective Edge Offloading Needs *Service Handoff*

As has been mentioned, most edge applications can gain benefits by offloading heavy computations from mobile clients to nearby edge servers. However, responsive real time services largely rely upon relatively short network distances between the mobile client and the edge server. When the mobile client moves farther away , benefits from offloading performance will be diminished dramatically. Therefore, effective edge offloading needs to support mobile services and users.

In the centralized cloud infrastructure, mobility of clients can be well supported since the client is supposed to connect to the centralized cloud server through the long latency WAN internet. However, in the edge computing infrastructure, mobile devices connect to nearby edge servers to benefit from high bandwidth and low latency connections. Therefore, when the mobile device moves farther away from its edge server, it might suffer from higher latency, or even become out of service.

In order to be continuously served by a nearby edge server, the offloading computation should be moved to a new nearby edge server from the previous server. We regard this process as a *service handoff* from the current edge server to a nearer edge server. This is analagous to the *seamless handoff* or *handover* mechanism in cellular networks, wherein the moving client connects to the nearest available base station, maintaining connectivity to the cellular network with minimal interruption. Therefore, one of the primary requirements for edge computing is to enable *service handoff* across edge servers, so that a mobile client is always served by nearby edge servers with high bandwidth and low latency.

2.3 Seamless Service Handoff via VM Migration

There exists one key difference between the cellular network handoff and and edge server handoff. In cellular networks, changing a base station for a client is as simple as rebuilding a wireless connection. Most run-time service states are not stored on the base station but are saved on the client, or on the remote server instead. Therefore, after re-connection, the run-time state can be seamlessly resumed through the new connection.

In the edge infrastructure, mobile devices use edge servers to offload resource-hungry or computation-intensive computations. This means that the edge server needs to hold the states of all resource intensive workloads. When offloading services handoff from one edge server to another, just rebuilding the connection is certainly not enough. Instead, we need to transfer all the runtime states of offloaded workloads from the current edge server to the nearer edge server.

One possible solution is to use virtual machine (VM) live migration [6] to migrate a VM from one edge server to another in order to seamlessly transfer the offloading workloads. However, this approach has already been shown to be not suitable for edge computing environments in [12]. First, live migration and service handoff are optimized according to different performance metrics. While live migration aims to reduce *downtime* of the VM, service handoff aims to reduce the *total time* from the time when handoff request is issued to the completion time of the migration. This is well discussed in [12]. Second, live migration is originally designed for high performance data centers with high bandwidth networks. However, this is not possible for edge servers which are deployed over the WAN. Furthermore, live migration relies on network-based storage sharing so only run-time memory state is transferred and not storage data. Apparently, network-based storage sharing across the edge computing infrastructure is not feasible due to its widely distributed nature and low WAN bandwidth between edge servers.

In order to enable handoff across edge computing servers, much research has focused on VM migration [12, 33]. However, the total handoff time was still several minutes on a WAN network. For example, it was shown it requires 245.5 seconds to migrate a running OpenFace instance under 5Mbps bandwidth (50ms latency) network in [12].

One of the reasons for the long latency of handoff is the large transfer size during the VM migration. Although VM synthesis could reduce the image size by splitting images into multiple layers, and only transferring the application-specific layer, the total transferred size is still in the magnitude of tens, or even hundreds of megabytes. The application layer is encapsulated with the whole application, including both the static binary programs and runtime memory data. We think this is an unnecessary cost.

On the other hand, the deployment of the VM synthesis system is challenging for the legacy system. In order to enable VM synthesis, the VM hypervisor needs to be patched to track dirty memory at runtime. Also, storage of VM images must be adapted to Linux FUSE interfaces in order to track file system changes inside running VMs. Those two changes are hard to deploy in practice since they change the behavior of legacy platform hypervisors and file systems, along with adding lots of performance overhead.

2.4 Why We Need Migration of Docker Container

Docker is a composing engine for Linux containers, an OS-level virtualization technique, isolating applications in the same Linux kernel based on namespace and cgroup management. Docker enables layered storage inside containers. Each Docker image references a list of read-only storage layers that represent filesystem differences. Layers are stacked hierarchically and union mounted as a container's root filesystem [18]. The layered storage allows fast packaging and shipping of any application as a lightweight container based upon sharing of common layers.

These layered images have the potential for fast migration of containers by avoiding transferrance of common image layers between two migration nodes. With container images (like in DockerHub) located in cloud storage, all the container images are available through the centralized image server. Therefore, before migration starts, an edge server has the opportunity to download the system and application images as the container base image stack. During

Figure 1: OpenFace Container's Image Layer Stack. Container's ID is *9ec79a095ef4db1fc5edc5e4059f5a10*. The stack list is stored in file *./aufs/layers/febfb1642ebeb25857bf2a9c-558bf695*. On the top is the writable (R/W) layer – *container layer*, and all the readonly (RO) layers are called *base image layers*.

the migration, we only need to transfer the run-time memory states and the thin *container layer* on top of the Docker image stack.

Apparently, the migration of Docker containers allows smaller transfer sizes than the virtual machine based approaches we have introduced above. The layered storage in Docker infrastructure enables an opportunity for service handoff based on container migration. By reducing the transfer size as much as possible, we can provide a nearly seamless offloading service handoff across the adjacent edge servers on a WAN network.

However, as of this writing, there is no current tool to leverage Docker's layered images to migrate containers efficiently. In this paper, we propose our work to reduce transfer size during container migration by leveraging layered storage.

3 PRELIMINARIES

In this section, we discuss the background of our work and dive into the inner details of Docker's layered storage system for containers. We will also identify the parts of the file system required to be transferred during migration.

Docker is becoming more and more popular and widely adopted in the industrial world, however, the technical details of layered storage management is still not well-documented. To the best of the author's knowledge, we are the first to examine the technical details of the Docker layered storage system, and leverage it to speed up the migration of Docker containers. Our work does not require any change to the Docker software stack.

3.1 Docker Layered Storage Management

As we mentioned above, Docker uses layered storage for its containers. Each Docker image references a list of read-only layers that represent file system differences. Layers are stacked on top of each other to form a base for a container's root filesystem [18].

3.1.1 *Container Layer and Base Image Layers.*

When a new container is created, a new, thin, writable storage layer is created on top of the underlying read-only stack of image layers. The new layer on the top is called *container layer*. All changes made to the container – such as creation, modification, or deletion of any file – are written to this *container layer*[18].

For example, Figure 1 shows the stacked image layers for OpenFace application. The dashed box on the top is the container layer of OpenFace. All the underlying layers are *base image layers*. To resolve the access request for a file name, the storage driver will search the file name from top layer towards the bottom layer, the first copy of the file will be returned for accessing, regardless the any other copies with same file name in the underlying layers.

3.1.2 *Image Layer ID Mapping.*

Since Docker 1.10, all the image and the layers in it are addressed by secure content SHA256 hash IDs. This addressable image design is supposed to improve security by avoiding name collisions, and at the same time, to maintain data integrity after pull, push, load, and save operations. It also enables better sharing of layers by allowing many images to freely share their layers locally even if they didn't come from the same build [18].

Given that there is no existing effort dived into how those addressable images worked by those SHA256 hash, we have investigated into the source code of Docker and its storage drivers. We find that there is an image layer ID mapping relationship which is not well documented yet: if the same image is downloaded from the same build on the cloud, Docker will mapping the layers' original layer IDs to a new secure content hash, called *cache ID*. Every image layer's original ID will be replaced with its cache ID. From then on, Docker daemon will address the image layer by this cache ID when it creates, starts, stops, or checkpoints/restores a container.

3.1.3 *ID Matching Between Docker Host.*

The mapping of layer IDs exposes a challenge to address those layers when we migrate a container between different hosts. As we have found, the image layer on the cloud will have different cache IDs when downloaded to different Docker host. Therefore, if we want to share the common image layers between different Docker hosts, we need to resolve the image layers ID mapping problem according different Docker host.

To address this problem, on each Docker host, we rebuild the mapping between its cache IDs to their original layer IDs by querying the Docker image database. Thus, we could match the image layers by its original IDs instead of the cache IDs exposed to the local Docker daemons. More details can be found in section 4.3.

3.1.4 *Docker's Storage Driver Backend.*

Docker delegates the task of managing the container's layered file system to its pluggable storage driver. The Docker storage driver is designed to provide a single unified view from a stack of image layers. Users can choose from different kinds of storage drivers that is the best for their particular case of usage.

In order to get more details of how those addressable images work, we investigate the source code of Docker system along with one of its most popular storage driver, AUFS. For other storage drivers like Btrfs, Device Mapper, overlay, and ZFS, they implement the management of image layers and the *container layer* in their own ways, but our framework could also be extended to those drivers.

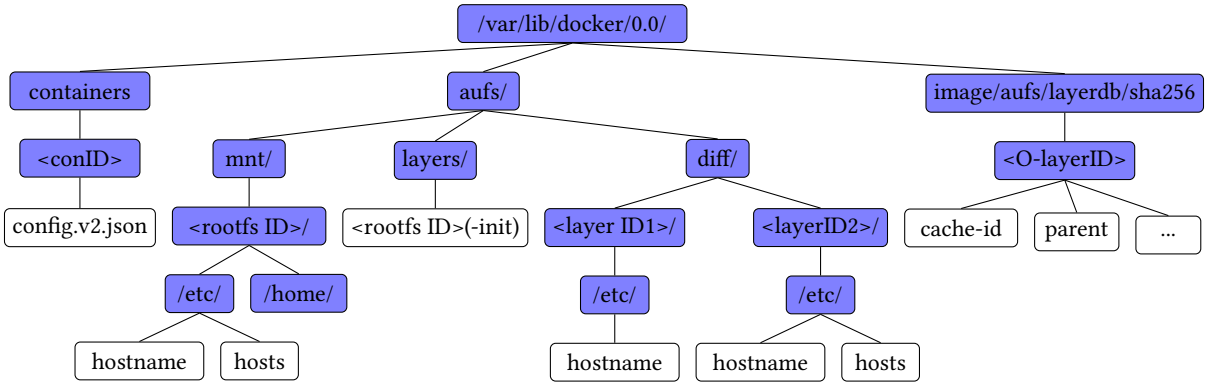


Figure 2: Docker Layered File System Structure Based on AUFS Storage Driver

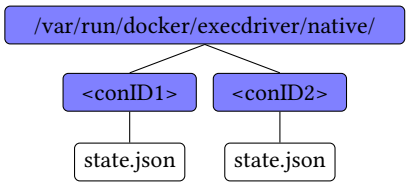


Figure 3: Runtime Data for Containers[1]

Due to limited time and space, we only conduct experiments on AUFS. The following section will discuss the inner details about our findings inside the Docker's AUFS storage driver.

3.2 AUFS Storage: A Case Study

The default storage driver on Docker is Advanced multi-layered Unification FileSystem (AUFS). Therefore, we take this as an example to introduce the layered images management.

AUFS storage driver implements Docker image layers by a union mount system. Union mount is a way of combining numerous directories into one directory that looks like it contains the contents from all the them [23]. Docker uses union filesystem to merge all image layers together and presents them as one single read-only view at a union mount point. If there are duplicate identities (i.e. file names) in different layers, only the one on the highest layer is accessible.

Figure 2 shows the Docker storage structure based on the AUFS driver. White box stands for a file and blue box stands for a directory. Since all directories share the same parent path */var/lib/docker/0.0/*, we will use '.' to represent this common directory in the following discussion.

AUFS driver use three main directories to manage image layers: *layers/*, *diff/*, and *mnt/*: directory *layers/* contains the metadata of how image layers are stacked together; directory *diff/* stores the content data for each layers; directory *mnt/* contains the mount point of the root file system for each the container.

3.2.1 *Container's Image Layer Stack List.*

We know that each Docker image contains several image layers. Those image layers are addressed by their SHA256 content hash IDs. Each Docker image has a list of layer IDs in the order of how they stacked from top to bottom. There are two files, *./aufs/layers/<rootfs ID>-init* and *./aufs/layers/<rootfs ID>*, both of which store a list of layer IDs. The former stores IDs of all initial image layers when the container is created. The latter stores IDs of the newly created layers in addition to the initial layers. In Figure 2, we use *./aufs/layers/<rootfs ID>(-init)* as the notion of the two files.

For example, for the container *OpenFace* with rootfs ID of *febfb16-42ebeb25857bf2a9c558bf695* [2] , it's initial layer stack is stored in the file *./aufs/ layers/febfb1642ebeb25857bf2a9c558bf695-init*. It contains all layers in the downloaded container image *bamos/openface*[3]. These layers will be read-only throughtout the whole life cycle of the container. Once a new layer is created, i.e. the *container layer*, the layer ID will be listed on the top line of the file *./aufs/ layers/febfb1642ebeb25857bf2a9c558bf695*.

When the Docker daemon starts or restores a container, it will refer to those two files to get a list of all underlying Docker image layer IDs and the container layer ID. Then it will resolve those addressable IDs and union mount all those layer stacks together in the specific order. After this, the container will get the full view of its root file system under the root mount point. We find that this file behaves like an important handler for all the union file systems for the container. If this file is missing, one container will not be able to union mount the root file system correctly.

3.2.2 *Image Layer Content Directory.*

AUFS manages the content of image layers in the directory of *./aufs/diff/*. The directory *./aufs/diff/<layer ID>/* stores all the files inside the specific layer identified by its *<layer ID>*. This can be either a readonly image layers or layers newly created for a container. If *<layer ID>* is the same as *<rootfs ID>* of one container, then this directory is where the content of *container layer* stores, i.e. all the file system changes of the container will be stored under this directory.

3.2.3 *Unified Mount Point.*

The directory *./aufs/mnt/<rootfs ID>/* is the mount point of the container's root file system. All image layers are union mounted to

[1]This is for Docker 1.10-dev, the latest Docker (version 17.04.0-ce, build 4845c56) has the runtime data directory changed to */var/run/docker/libcontainerd/containerd/*.
[2]SHA256 ID has 64 hexadecimal characters, here we truncate it to 32 hexadecimal characters in order to save space.
[3]https://hub.docker.com/r/bamos/openface/.

this folder and provide a single file system view for the container. For example, as shown in Figure 2, when a container is created based on a Linux image, its mount point will contain the root directory contents like */usr/, /home/, /boot/, etc.* . All those directories are mounted from its underlying layered storage, including both the read only image layers downloaded from the registry and the newly created layers for the container. Since this directory is a mount point for a running container's file system, it will be only available when the container is running. If the container stops running, all the image layers will be unmounted from this mount point. So it will become an empty directory.

Here, the name of the root file system directory, *<rootfs ID>*, is the same as the name of the *container layer* for this container.

3.2.4 Layer ID Mapping.

Until now, the layer IDs we have discussed above are just local SHA256 IDs, or the so called cache IDs, which are generated dynamically when each image layer is downloaded by *'docker pull'* command. From then on, Docker daemon will address the image layer use the cache ID instead of its original layer ID (noted as O-layerID in this paper).

We find the Docker storage system maintains a mapping relationship between the original layer IDs and its cache IDs. All the cached IDs of image layers are stored in the */image/aufs/layerdb/sha256* directory. For example, the file *./image/aufs/layerdb/sha256/<O-layerID>/cache-id* shown in Figure 2 stores the cache ID of the image with original ID <O-layerID>. For example, if a hash ID *fac86d61dfe33f821e8d0e7660473381* is stored in the file of *./image/aufs/ layerdb/ sha256/ 6384c447ddd6cd859f9be3b53f8b015c/cache-id*, this means there is an image layer with an original ID of *6384c-447ddd6cd859f9be3b53f8b015c* and it's cache ID is *fac86d61dfe33f-821e8d0e7660473381*.

3.2.5 Container Configuration and Runtime State.

There are several directories that store the configuration files and runtime data. Figure 3 shows the runtime data directories for each containers. For one container with ID of *<conID>*, there will be a JavaScript object notation (JSON) file *state.json* that stores the runtime state of the container. For example, the *init pid* of the containers' processes is identified by key "*init_process_pid*", and the root file system mount point path can be found via key "*rootfs*". There are also some runtime cgroup and namespace meta data, etc..

Along with the runtime data directory, there is another directory named */var/lib/docker/0.0/containers/<conID>* that contains the configuration files for each container. The directory is shown in Figure 2. We use *<conID>* as a notation of the container's hash ID. For example, from the file of *config.v2.json*, we can find the container's creation time, the command that was run once the container was created, etc..

3.3 Docker Container Migration in Practice

Although there is no official migration tool for Docker containers yet, many enthusiastic developers have constructed some customized tools for certain versions of Docker platforms. These tools have demonstrated the feasibility of Docker container migration. For example, the CRIU project [8] supports migration of Docker-1.9.0-dev, and project [2] extends it to support Docker 1.10-dev.

App	Total time	Down time	FS Size	Total Size
Busybox	7.54 *s*	3.49 *s*	140 KB	290KB
OpenFace	26.19 *s*	5.02 *s*	2.0 GB	2.17GB

Table 1: Docker Container Migration Time (between two VMs on the same host machine, bandwidth 600Mbps, latency 0.4ms)

App	Total time	Down time	FS Size	Total Size
busybox	133.11*s*	9*s*	140 KB	290KB
openface	~ 3200*s*	153.82*s*	2.0G	2.17G

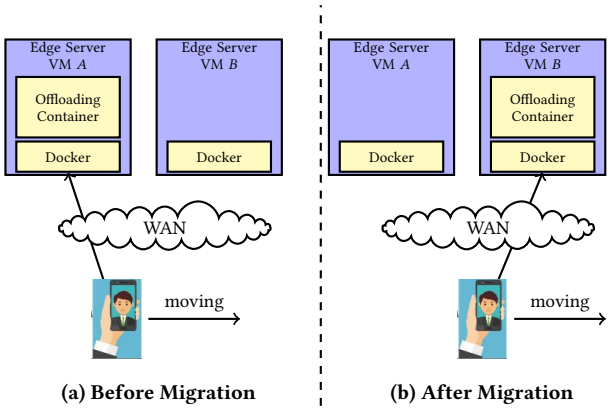
Table 2: Docker Container Migration Time (between two different hosts through Wireless LAN, bandwidth 15Mbps, latency 5.4ms)

However, both methods simply transfer all the files located under the container's root file system from source server to the target server, where the files are actually composed from all the image layers of the migrated container. The methods just ignore the underlying union mounting of the storage layers. This behavior would cause severe problems from several aspects:

(1) It will corrupt the layered file system inside the container after restoration on the target server. The tools just transfer the whole file system into one directory on the destination server and mount it as root directory for the container. After restoration on the target host, the container no longer maintains its layered image stacks as on the source node.
(2) It substantially reduces the efficiency and robustness of migration. It will synchronize the whole file system of the Docker container using Linux *rsync* command while the container is still running. First, running *rsync* command on a whole file system would be pretty slow due to the large amount of files. Second, this can result in file contention when the container process and *rsync* process on the source node attempt to access a same file. Contention will cause synchronization error which can result in migration error or failure.

To verify our claim, we have tested this tool with experiments to migrate containers through different network connections. Our experiments use one simple container, Busybox, and an application, OpenFace, for edge server offloading. Busybox is a stripped-down Unix tool in a single executable file. This results in a tiny file system inside the container. OpenFace[1] is an application that dispatches images from mobile devices to the edge server, then executes the facial recognition algorithm on server, and finally sends back a text string with the name of the person. The container has a huge file system, approximately 2 gigabytes.

Table 1 indicates migration could be done within 10 seconds for Busybox, and within 30 seconds for OpenFace. The two nodes are two virtual machines on the same physical host. The network between two virtual hosts has a 1Gbps bandwidth with latency of 0.5 milliseconds which transfers 2.17 GB data within a short time.



(a) Before Migration **(b) After Migration**

Figure 4: Offloading Serivce Handoff: Before and After Migration of Offloading Container.

Due to the dominance of wireless connections in edge computing networks, we further test their containers migrating between two physical hosts. We used the same wireless LAN network with bandwidth of 15Mbps and latency of 4ms. Table 2 shows that the migration of the Busybox container takes 133.11 seconds with transferred size as small as 152 kilobytes. As for migration of OpenFace, it needs to transfer more than 2 Gigabytes data and costs about 3200 seconds.

As we have stated, this poor performance is caused by transferring the whole file system including all the stacked image layers of the container. This is even worse than a mature VM migration. Migration of VMs could avoid transferring a portion of the file system by sharing the base VM images [12], which will finish the migration within several minutes.

Therefore, we need a new tool to properly migrate the Docker containers, avoiding unnecessary transmission of common image layer stacks. This new tool should leverage the layered file systems to transfer the *container layer* only.

4 EFFICIENT MIGRATION OF DOCKER CONTAINERS

In this section, we introduce the design of our service hand-off framework based on Docker container migration. First, we provide a simple usage scenario, give an overview of system architecture and the algorithm of service handoff. Second, in sections 4.2 and 4.3, we discuss our methodology of storage synchronization based on Docker image layer shared between two different Docker hosts. Finally, in sections 4.4, 4.5, and 4.6, we show how to further accelerate the migration speed through memory difference transfer, file compression, pipelined and parallel processing during the migration of Docker containers.

4.1 System Overview

Figure 4 shows an exemplar usage scenario of offloading service hand-off based on Docker container migration. In this example (OpenFace[1]), the mobile client achieves real-time face recognition by offloading workloads to an edge server. The mobile client continuously reads images from the camera and sends them to the edge server. The edge server runs the facial recognition application in a container, processes the image via deep neural networks algorithm, and finally sends the recognition result back to the client.

All containers are running inside Virtual Machines (for example, VM *A*, VM *B* in Figure 4). This allows users to scale up the deployment more easily and control the isolation among the applications in different levels.

Before migration, each mobile client offloads computations to its nearest edge server **A**, where all the computations are processed inside a Docker container. In this paper, we simply call the container which computes the offloading workloads on the server side, the *offloading container*. When the mobile client moves beyond the reach of server **A** and reaches the service area of edge server **B**, its offloading computation shall be migrated from server **A** to server **B**. This is done via migration of its offloading container, where all the runtime memory states as well as associated storage data should be synchronized to the target server **B**.

Figure 5 shows the design details of this architecture as well as the migration algorithm in multiple processing stages described below:

- **S1 Synchronize Base Image Layers** Once a container is created, it starts to offer offloading service to the client. We dynamically predict the possible target edge servers that are nearest to the client. Then we request those target servers to start synchronizing the base image layers for that container.
- **S2 Pre-dump Container.** Before we received the migration request, we will try to dump a snapshot of the container runtime memory and send to the possible target edge servers predicted in the previous stage **S1**. The container is still running during this stage.
- **S3 Migration Request Received on Source Server**. Once the migration request is initiated, the request will be sent to the source server, which we regard as the start point of the service handoff.
- **S4 Checkpoint and Stop Container.** Upon receiving the migration request, the source edge server will checkpoint and stop the container in order to send all of the latest run-time states to the target edge server. From this point the container will stop running on the source server and wait to be restored on the target server.
- **S5 Container Layer Synchronization** After checkpointing the container, the file system will not be changed. In this step, we will send the container layer contents to the target server. At the same time, all the checkpointed runtime states and configuration files, such as *state.json, config.v2.json, etc.* are also transferred to the target server.
- **S6 Docker Daemon Reload** After we send all the runtime states and configuration files to the target node, Docker daemon on the target node still cannot recognize the container immediately. This is because those runtime states and configuration files are created by another Docker daemon on the source node. The daemon on target node doesn't have those configurations loaded into the runtime database. We must reload the Docker daemon in order to reload those runtime state and configuration files just received from the source node.

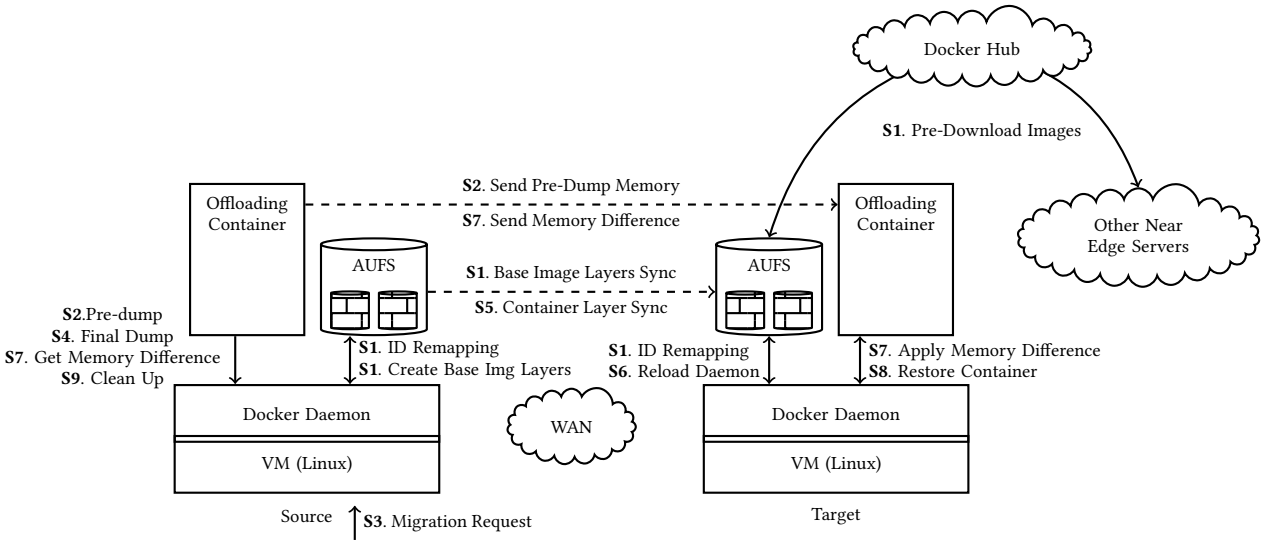


Figure 5: System Architecture for Offloading Service Hand-off

- **S7 Get, Send, and Apply Memory Difference.** After we get checkpointed images from the final dump of the container, we then compare this final dump memory to the pre-dumped memory in stage **S2**. We then could get the memory differences and send only these differences to the target server.
- **S8 Restore Container.** After all transfer finish, we restore the container on the target host with all runtime status retrieved. At the same time, the target host will go to stage **S1** to prepare the next iteration of service handoff in the future. Now the migration has finished and the user starts to be served from the target edge server.
- **S9 Clean Up Source Node.** Finally, on the source node, we need to clean up the footprints of the offloading service. To do this, we just simply remove the container, then all its related runtime footprints will be gone. We might need to carefully choose the right time to clean up, in case the user moves back sometime in the future. If this is the case, it will be better if we keep the old footprints extant to avoid some transmission overhead.

4.2 Strategy to Synchronize Storage Layers

As has been mentioned, Docker's storage driver supports layered images whereby each layer represents the summary of file system differences. A running container's layered storage is composed of one writable container layer and several read only base image layers.

The thin writable container layer stores all the files created or modified by the newly created container from the base image. As long as the container is running, this layer is subject to change. So we postpone the synchronization of the container layer until the container is stopped.

All the base image layers are read only inside containers. Once an image layer is created, it will not be changed during the whole life cycle of all the containers running on top of it. We synchronize those base image layers as early as possible.

There are two kinds of base image layers. First, in most cases, the base image layers for the container are downloaded by **docker pull** command from the centralized image registry, like Docker Hub. All those images can be shared by downloading from the same registry. Second, the container itself can also create its own image layers by saving the current container layer as one read-only image layer.

For these two kinds of base images, we employ different synchronization strategies. More specifically, we download the common image layers between two Docker hosts from the centralized image registry and transfer only the different image layers between two Docker hosts.

By downloading the common image layer from the registry, we reduce the traffic between two edge servers. Furthermore, given that the download could start as soon as the container is created on the source server, the download time could be considered ahead of migration start and therefore amortized.

Finally, for the base image layers created locally by the container, we transfer each such image layer as the image layer is created, regardless if the migration has started or not.

4.3 Layer ID Remapping

As discussed in section 3.1.3 and 3.2.4. The downloaded images from the common registry have different cache IDs exposed to each Docker host. In order to share these common images across different Docker hosts, we need to match these image layers based upon the original IDs instead of the cache IDs.

In order to do this, we first remap the cache IDs to the original IDs and compare the original IDs on two different Docker hosts. If the two image layers on two hosts share the same original IDs, we infer that they are exactly the same image layers.

Then, for the matched original layer IDs on both Docker hosts, we remap the original IDs to the local cache IDs on the target host. Now we have the new cache IDs on the target Docker host. Then

we update the migrated container with the new cache IDs on the target Docker host.

By doing so, the common image layers on the migrated container will be reset with the new cache IDs that are addressable to the Docker host on the target server. When we restore the container in the future, the file system will be mounted correctly from the shared image layers on the target server.

For the original IDs that don't match on the two hosts, we regard them as new image layers and add them to a waiting list to transfer in step **S5**.

4.4 Pre-Dump & Dirty Memory Synchronization

In order to reduce the total memory image size during hand-off, we checkpoint the container and dump a snapshot of container memory in stage **S2**. This could happen as soon as the container is created, or as required, we could dump the memory when the most frequently used binary programs in the application are loaded into the memory. This snapshot of memory will be served as the base memory image for the migration.

After the base memory image is dumped, it is transferred immediately to the server. We assume that the transfer will be finished before the hand-off starts, which is reasonable since we can send the base memory image as soon as the container starts. After the container starts, and before the hand-off starts, the download of application Docker images are also started on the near edge servers. Hence, we process those two steps in parallel to reduce the total time span. This is further discussed in section 4.6. Thus, upon hand-off start, we have the base memory image of the container already on the target server.

Once the migration request is received, we checkpoint and stop the container. After getting the checkpointed memory, we do a diff operation on it from the base memory image we dumped in stage **S2**. In this way, we only need to transfer the difference of the dump memory to the target node.

4.5 Data Transfer

During container migration, we mainly have 4 types of data needing to be transferred: the layer stacks information, the file system of thin writable *container layer*, the meta data files of the container, and the snapshot of container memory and memory difference. Some of the data is in the form of string messages, such as layer stack information. Some data are in plain text files, such as most contents and configuration files. Memory snapshots and memory differences are all binary image files. According to the feature of the files, we design different strategies to transfer the data.

The layer stacks information is sent via UNIX RPC API implementation in [10]. This is based on a socket connection created by python scripts. This information consists of a list of SHA256 ID strings, so its quite efficiently sent as a socket message. It does not merit applying compression because the overhead of compression outweighs the benefits for those short strings.

As for other data, including the container writable layer, meta data files, the dump memory images, and image differences, we use bzip2 to compress and send via authorized ssh connection.

4.6 Parallel & Pipelined Processing

With the help of parallel and pipelined processing, we could further improve the efficiency of the whole process, and reduce the total migration time.

First, starting a container will trigger two events to run in parallel: a) on the potential target servers near the source node, downloading images from centralized registry, and b) on the source node, pre-dumping/sending base memory images to the potential target servers. Those two processes could be run at the same time in order to reduce the total time of stage **S1** and **S2**.

Second, daemon reload in stage **S6** is required on the target host, it could be triggered immediately after **S5**. It could be paralleled with stage **S7**, when the source server is sending the memory difference to the target host. Stage **S5** cannot be paralleled with **S6**, because daemon reload on the target host requires the configuration data files sent in stage **S5**.

Third, in stage **S5**, we use compression to send all files in the *container layer* over an authorized ssh connection between the source and target host. The compression and transfer of the *container layer* can be pipelined using Linux pipes.

Lastly, in stage **S7**, after we get the final memory snapshot, we will need to identify memory differences by comparing the base memory images with the images in the new snapshot, then we send the differences to the target and patch the differences to the base memory image on the target host. This whole process could also be piplined using Linux pipes.

4.7 Security Isolation

Finally, it is critical to minimize security risks to the offloading services running on the edge servers. Isolation between different services could provide a certain level of security. Our framework provides an isolated running environment for the offloading service via two layers of the system virtualization hierarchy. Different services can be isolated by running inside different *Linux containers*, and different containers are allowed to be further isolated by running in different *virtual machines*.

More thorough security solutions need to be designed before this framework can be deployed to the real world. These solutions include, but are not limited to efficient run-time monitoring, secure system updating, etc.. But here we focus on the mobility of services from the performance side, and leave most security work to the future.

5 EVALUATION

In this section, we evaluate the performance of our system under different workloads and different network conditions. Specifically, we evaluate offloading with our service handoff system by measuring the reduction of the overall hand-off time and transferred data size.

5.1 Set-Up and Benchmark Workloads

Our testbed is built on a desktop server, which is equipped with Intel Core i3-6100 Processor (3.70GHz, 2 cores, 4 threads), and 16GB DDR4 memory. Two virtual machines are running with 2 vcpus and 4GB memory each. A laptop acts as a client with an Intel Core i5-2450M CPU (2.5GHz, 2 cores, 4 threads) and 4GB DDR3 memory.

The hosts and virtual machines are running Ubuntu 16.04 LTS as the operating system. Docker version is 1.10-dev, which is built with experimental feature enabled.

In our experiment, we set up migration scenarios using emulated bandwidth on two virtual machines each running a Docker instance. Docker containers are migrated from the Docker host on source VM to the Docker host on another VM. Two virtual machines run on the same physical server.

We use Linux Traffic Control (tc[3]) tool to control network traffic. In order to test our system under WANs , we emulated low bandwidth ranging from 5Mbps to 45Mbps. According to the average bandwidth observed on the Internet[29], we set latency at a fixed 50ms to emulate the WAN environment for edge computing.

Since edge computing environments can also be adapted to the LAN network such as in a university, smart homes, or other community, they can usually have a high bandwidth as well as low network latency. Therefore, we also tested several higher bandwidths, ranging from 50Mpbs to 500Mpbs. The latency is set to 6ms which is an average latency from the author's university LAN.

For the offloading workloads, we use Busybox as a simple workload to show the functionality of the system as well as the non-avoidable system overhead when processing container migration. In order to show the performance of offloading service handoff regarding real world applications, we chose OpenFace to provides a sample workload.

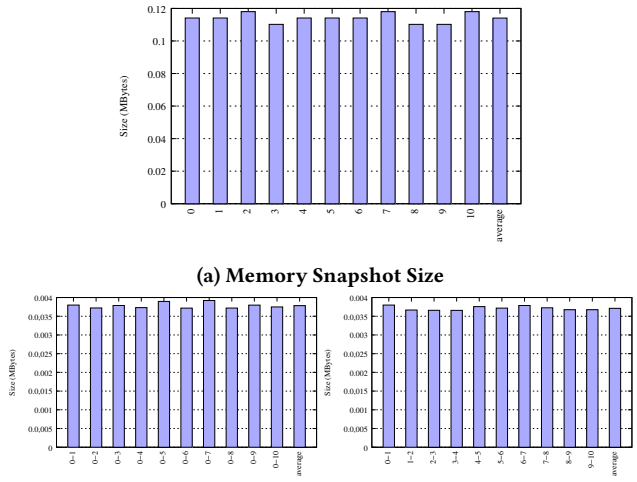
5.2 Dirty Memory & Experimental Findings

6 and Figure 7 give an overview of the dump memory image sizes while running the container of Busybox and OpenFace. The data is collected from 11 dumps of the running container of Busybox and OpenFace, labeled from dump 0 to 10. Memory is dumped every 10 seconds. Container continues to run during the first 10 dumps, and stops after 11th dump. From the figure we can see the memory difference (or dirty memory) is much smaller than the original memory dump. The migration could be sped up by only transferring the memory difference.

Furthermore, Figure 6b ~ 6c and 7b ~ 7c show us the sizes of memory differences between adjacent dumps as well as the original dump. We find a feature of memory access patterns for Busybox and OpenFace application. Although their memory is continuously changing, the changes reside in a specified area: a 4KB area for Busybox and 25MB area for OpenFace. In this case, iterative transferring of memory difference to the target server will be waste of effort. Although this will not happen on every application, it is valuable to be considered as a special case for further research regarding the migration of containers by iterative transfer. One way to utilize those kinds of memory access patterns is by not iteratively transferring every memory difference. Instead, transfer only the last iterative difference to achieve the same end result.

5.3 Evaluation of Pipeline Performance

In order to verify the effectiveness of pipelined processing, we implemented the pipeline processing of two time consuming steps: *imgDiff* and *imgSend*, where *imgDiff* is to get memory difference and *imgSend* is to send memory difference to the target during migration. Figure 8 and Figure 9 reports the timing benefits we could achieve by using pipelined processing. From the figure, we can see that, without pipelined processing, most time costs are by getting and sending the memory difference. After applying pipelined processing, we could save 5 ~ 8 seconds for OpenFace migration. Busybox also saves a certain amount of time with pipelined processing.

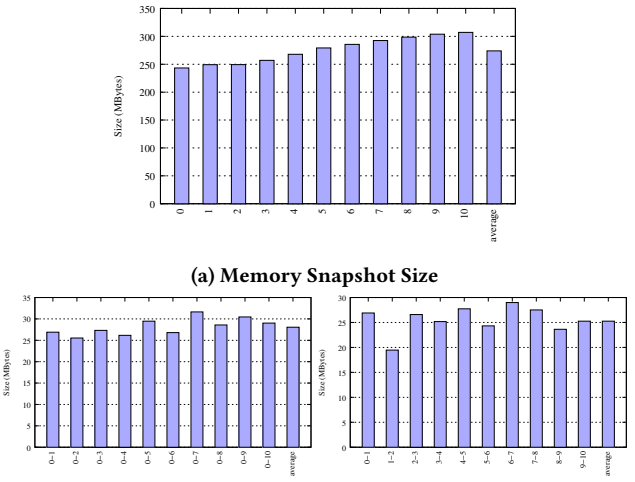


(a) Memory Snapshot Size

(b) Dirty Memory, original

(c) Dirty Memory, adjacent

Figure 6: Dirty Memory Size Analysis for Busybox. Figure 6a shows checkpointed memory size for total 11 dumps. Figure 6b shows dirty memory size beween each of dump 1 10 and the original dump 0. Figure 6c shows dirty memory size between two adjacent dumps.



(a) Memory Snapshot Size

(b) Dirty Memory, original

(c) Dirty Memory, adjacent

Figure 7: Dirty Memory Size Analysis for OpenFace. Figure 7a shows checkpointed memory size for total 11 dumps. Figure 7b shows dirty memory size beween each of dump 1 10 and the original dump 0. Figure 7c shows dirty memory size between two adjacent dumps.

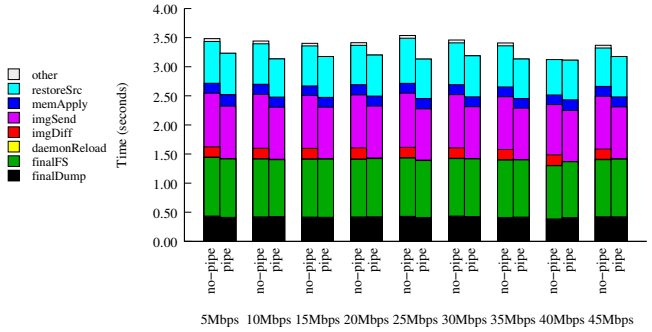


Figure 8: Busybox: Time of Container Migration Stages with and without Pipelined Processing.

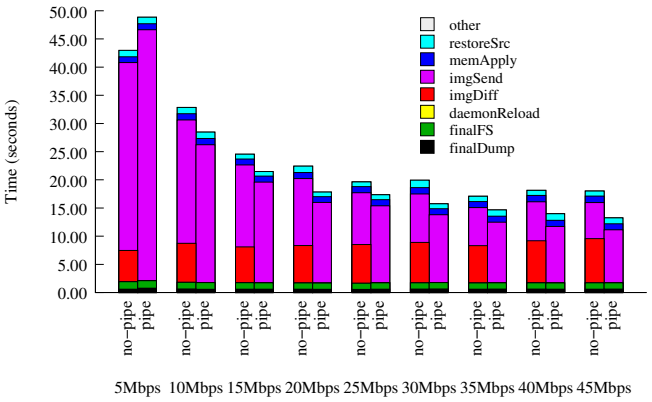


Figure 9: OpenFace: Time of Container Migration Stages with and without Pipelined Processing

5.4 Overall Performance and Comparison with State-of-the-Art

We evaluate the total *handoff time* and *transferred size* during offloading service handoff under different bandwidths and network latencies.

Table 3 shows an overview of the performance of our system under different network bandwidth conditions.

Handoff time is from the time the source host receives a migration request until the offloading container is successfully restored on the target host. The *Down time* is from the time when the container is stopped on the source server to the time when the container is restored on the target server. *Handoff time* and *Down time* are very similar since we immediately checkpoint and stop the container once we get the migration request.

Pre-Transfer Size is the transferred size before *handoff* starts, i.e. from stage **S1** until stage **S3** . *Final-Transfer Size* is the transferred size during the handoff, i.e. from stage **S3** until the end of final stage **S8**.

Figure 10 and Figure 11 shows the performance under difference network latencies of 50*ms* and 6*ms* for Busybox and OpenFace. It shows a tiny difference when facing different latencies. This means our system is suitable for a wide range of network latencies.

Bandwidth (Mbps)		Handoff Time(s)	Down Time(s)	Pre-Transfer Size (MB)	Final-Transfer Size (MB)
Busybox	5	3.2 (7.3%)	2.8 (7.9%)	0.01 (0.2%)	0.03 (0.3%)
	10	3.1 (1.8%)	2.7 (1.6%)	0.01 (0.2%)	0.03 (0.6%)
	15	3.2 (1.4%)	2.8 (1.6%)	0.01 (0.5%)	0.03 (0.9%)
	20	3.2 (1.6%)	2.8 (1.8%)	0.01 (0.3%)	0.03 (0.4%)
	25	3.1 (1.6%)	2.7 (1.8%)	0.01 (0.2%)	0.03 (0.9%)
	30	3.2 (1.4%)	2.8 (1.2%)	0.01 (0.3%)	0.03 (0.5%)
	35	3.1 (3.5%)	2.7 (3.3%)	0.01 (0.3%)	0.03 (0.6%)
	40	3.1 (3.4%)	2.7 (3.5%)	0.01 (0.2%)	0.03 (0.5%)
	45	3.2 (1.9%)	2.7 (1.8%)	0.01 (0.2%)	0.03 (0.8%)
	50	3.2 (1.7%)	2.7 (1.6%)	0.01 (0.2%)	0.03 (2.7%)
	100	3.2 (1.6%)	2.7 (1.4%)	0.01 (0.3%)	0.03 (0.4%)
	200	3.1 (1.8%)	2.7 (1.8%)	0.01 (0.1%)	0.03 (0.5%)
	500	3.2 (2.0%)	2.8 (2.2%)	0.01 (0.2%)	0.03 (0.4%)
OpenFace	5	48.9 (12.6%)	48.1 (12.7%)	115.2 (6.1%)	22.6 (13.0%)
	10	28.5 (6.9%)	27.9 (7.0%)	119.4 (3.5%)	22.2 (10.9%)
	15	21.5 (9.1%)	20.9 (9.4%)	116.0 (7.3%)	22.1 (11.1%)
	20	17.8 (8.6%)	17.3 (8.9%)	116.0 (6.9%)	21.2 (12.0%)
	25	17.4 (11.5%)	16.8 (12.0%)	114.3 (7.6%)	23.7 (14.8%)
	30	15.8 (7.5%)	15.1 (7.4%)	119.3 (2.5%)	22.7 (9.3%)
	35	14.7 (13.6%)	14.0 (14.3%)	116.8 (5.9%)	22.2 (15.6%)
	40	14.0 (7.3%)	13.4 (7.6%)	112.5 (8.1%)	23.0 (8.8%)
	45	13.3 (8.6%)	12.6 (9.1%)	111.9 (9.1%)	22.6 (11.7%)
	50	13.4 (10.7%)	12.8 (11.1%)	115.2 (5.3%)	23.2 (5.3%)
	100	10.7 (9.6%)	10.1 (10.1%)	117.2 (2.4%)	21.6 (10.8%)
	200	10.2 (12.9%)	9.6 (13.5%)	116.8 (2.4%)	20.6 (17.6%)
	500	10.9 (5.6%)	10.3 (5.9%)	117.4 (1.5%)	23.0 (3.9%)

Table 3: Overall System Performance. Average of 10 runs and relative standard deviations (RSDs, in parentheses) are reported.

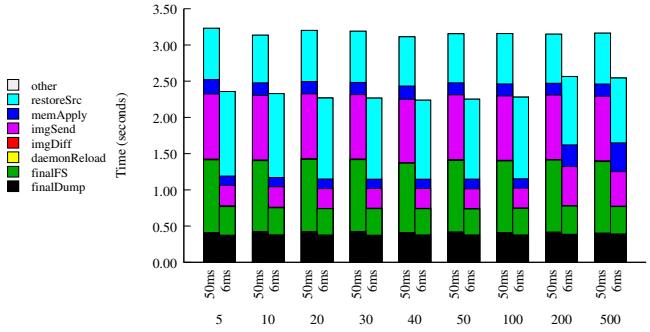


Figure 10: Busybox: Comparison of Migration Time Under Latency of 50ms and 6ms.

From Table 3 and Figure 10 , we can see the simple Busybox container can be migrated very quickly regardless the network bandwidth and latency.

From Table 3 and Figure 11 , we can see the OpenFace offloading container can be migrated within 49 seconds under the lowest bandwidth 5Mbps with 50 ms latency. Compared to the work in [12], which has a handoff time of 247 seconds under the same network conditions. For 25Mbps and 50 ms latency, our system achieves 17.4 seconds while [12] needs 39 seconds. The relative standard deviations in Table 3 shows the robustness of our experimental result.

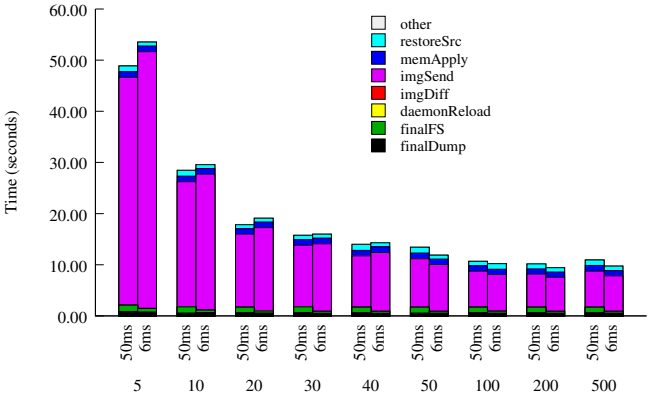


Figure 11: OpenFace: Comparison of Migration Time Under Latency of 50ms and 6ms.

Therefore, our system could reduce the total handoff time by 56% ~ 80% compared to the state-of-the-art work of VM handoff [12] on edge computing platforms.

6 RELATED WORK

In this section, we discuss related work in the areas of dynamic VM synthesis, VM handoff, and container migration.

6.1 Dynamic VM Synthesis and VM Hand-off

Techniques based on virtual machine migration have been proposed in [12] [33] to accelerate the service handoff across edge servers.

In order to enable high speed service handoff based on virtual machine migration techniques, Satyanarayanan *et al.* in [33] proposed VM synthesis to divide huge VM images into a base VM image and a relatively small overlay image for one specific application. Before migration starts, the server is assumed to have a base image, so that the transition of this base image is avoided. During migration, only the application overlay image is transferred from the source to the target server.

Based on the work of VM synthesis, VM handoff across Cloudlet servers (alias of edge servers) was proposed in [12]. VM handoff allows the mobile client to seamlessly transfer the runtime status of its VM from one edge sever to a nearer edge server. Besides VM synthesis, it uses dirty tracking for both file system and main memory to reduce the transferred size during handoff. Additionally, it also uses compression and pipelined processing to reduce transfer time, and algorithms to adapt to different network conditions. Finally, it highly reduced the transfer size and migration time under WAN environments.

While significant efforts have been invested into VM synthesis and handoff, the performance cannot meet the requirements of offloading services in edge computing environments, given the mobility of clients, dominance of wireless access, geographical distribution of edge nodes, and real-time interaction requirement. Instead, our work is along the lines of container, a lightweight OS-level virtualization technique, with focus on the migration of containers to reduce the total handoff time.

6.2 Containers Migration

Containers provide operating system level virtualization by running a group of processes in isolated environments. It is supported by the kernel features of namespace and cgroups (control groups)[31]. Namespaces are used to provide a private view of system resources for processes [27], while cgroups are used to restrict the quantity of resources a group of processes can access[30].

Container runtime is a tool that provides an easy-to-use API for managing containers by abstracting the low-level technical details of namespaces and cgroups. Although creating a container by crafting cgroup and namepaces step by step is possible [4], managing containers by container runtime is much easier. Such tools include LXC/LXD[21] , runC[11], rkt[7], OpenVZ[24], Docker[19], etc.. Different container runtime has different scenerios of usage. For example, LXC/LXD only cares about full system containers and doesn't care about what kind of application runs inside the container, while Docker aims to encapsulate a specific application in the container.

Migration of containers becomes possible when CRIU (Checkpoint/Restore In Userspace)[8] supports the checkpoint/restore functionality for Linux. Now CRIU supports the checkpoint and restore of containers for OpenVZ, LXC, and Docker.

Based on CRIU, OpenVZ now supports migration of containers. The implementation can be found from the CRIU community's open source project, P.Haul (process Hauler) [10]. It is claimed that migration could be done within 5 seconds[35]. However, OpenVZ uses a distributed storage system [25], where all files are shared across a high bandwidth network. This means that during container migration, it only needs to migrate the checkpointed memory images and no file transfer needs to be done. However, due to the limited WAN bandwidth for edge servers, it is not possible to deploy distributed storage. Therefore, migration of OpenVZ containers is not suitable for service handoff on edge computing platforms.

For LXC containers, the migration is implemented in Qiu's thesis work [28], which is also based on the CRIU project. However, LXC regards containers as a whole system container, and there is no layered storage for containers. Therefore, during container migration, all of the file system for that container must be migrated together with all memory status. The total transfer size will be comparable with migrating a whole virtual machine(VM). So, this is still not feasible for an edge computing environment.

For Docker containers, there is a sample migration helper in P.Haul's source code, which supports only an older version of docker-1.9.0-dev. It is also extended by Ross Boucher to support docker-1.10-dev. However, as we have discussed in section 3.3, we find this implementation transmits the entire file system of that container, regardless of the layered storage of the Docker platform. This makes the migration unsatisfactorily slow across the edges of the WAN. Therefore, in this paper, we dive into the inner technical details of Docker storage and investigate leveraging the layered storage of Docker platform during migration. Finally we avoid the transfer of unnecessary file system data by sharing container image layers between different Docker hosts.

7 CONCLUSION

We propose a framework that enhances service handoff across edge offloading servers by leveraging the layered storage system of Docker containers to improve migration performance. Our system enables the edge computing platform to continuously provide offloading services with low end-to-end latency while supporting high client mobility. By leveraging the layered file system of Docker containers, we eliminate unnecessary transfers of a redundant and significant portion of the application file system. By transferring the base memory image ahead of the handoff, and transferring only the incremental memory difference when migration starts, we further reduce total transfer size. Finally, our system shows that the hand-off time is reduced by 56% ∼ 80% compared to the state-of-the-art VM handoff for the edge computing platform.

REFERENCES

[1] Brandon Amos, Bartosz Ludwiczuk, and Mahadev Satyanarayanan. 2016. *OpenFace: A general-purpose face recognition library with mobile applications.* Technical Report. CMU-CS-16-118, CMU School of Computer Science.

[2] Ross Boucher. 2017. Live migration using CRIU. (2017). Retrieved Apr 22, 2017 from https://github.com/boucher/p.haul

[3] Martin A. Brown. 2017. Traffic Control HOWTO. (2017). Retrieved Apr 22, 2017 from http://www.tldp.org/HOWTO/Traffic-Control-HOWTO/

[4] Eric Chiang. 2017. Containers from Scratch. *Online, https://ericchiang.github.io/post/containers-from-scratch/* (2017).

[5] Mung Chiang and Tao Zhang. 2016. Fog and IoT: An overview of research opportunities. *IEEE Internet of Things Journal* 3, 6 (2016), 854–864.

[6] Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. 2005. Live migration of virtual machines. In *Proceedings of the 2nd Conference on Symposium on Networked Systems Design & Implementation-Volume 2*. USENIX Association, 273–286.

[7] CoreOS. 2017. A security-minded, standards-based container engine. (2017). Retrieved Apr 22, 2017 from https://coreos.com/rkt

[8] CRIU. 2017. CRIU. (2017). Retrieved Apr 22, 2017 from https://criu.org/Main_Page

[9] Eduardo Cuervo, Aruna Balasubramanian, Dae-ki Cho, Alec Wolman, Stefan Saroiu, Ranveer Chandra, and Paramvir Bahl. 2010. MAUI: Making Smartphones Last Longer with Code Offload. In *Proceedings of the 8th International Conference on Mobile Systems, Applications, and Services (MobiSys '10)*. ACM, New York, NY, USA, 49–62.

[10] Pavel Emelyanov. 2017. Live migration using CRIU. (2017). Retrieved Apr 22, 2017 from https://github.com/xemul/p.haul

[11] Linux Foundation. 2017. runC. (2017). Retrieved Apr 22, 2017 from https://runc.io/

[12] Kiryong Ha, Yoshihisa Abe, Zhuo Chen, Wenlu Hu, Brandon Amos, Padmanabhan Pillai, and Mahadev Satyanarayanan. 2015. *Adaptive vm handoff across cloudlets.* Technical Report. Technical Report CMU-CS-15-113, CMU School of Computer Science.

[13] Kiryong Ha, Zhuo Chen, Wenlu Hu, Wolfgang Richter, Padmanabhan Pillai, and Mahadev Satyanarayanan. 2014. Towards wearable cognitive assistance. In *Proceedings of the 12th annual international conference on Mobile systems, applications, and services*. ACM, 68–81.

[14] Zijiang Hao and Qun Li. 2016. Edgestore: Integrating edge computing into cloud-based storage systems. In *Edge Computing (SEC), IEEE/ACM Symposium on*. IEEE, 115–116.

[15] Zijiang Hao, Ed Novak, Shanhe Yi, and Qun Li. 2017. Challenges and Software Architecture for Fog Computing. *IEEE Internet Computing* 21, 2 (2017), 44–53.

[16] Yun Chao Hu, Milan Patel, Dario Sabella, Nurit Sprecher, and Valerie Young. 2015. Mobile edge computingâĂŤA key technology towards 5G. *ETSI White Paper* 11 (2015).

[17] Docker Inc. 2017. Docker Hub. (2017). Retrieved Apr 22, 2017 from https://hub.docker.com/

[18] Docker Inc. 2017. Docker Images and Containers. (2017). Retrieved Apr 22, 2017 from https://docs.docker.com/engine/userguide/storagedriver/imagesandcontainers/#images-and-layers

[19] Docker Inc. 2017. What is Docker? (2017). Retrieved Apr 22, 2017 from https://www.docker.com/what-docker

[20] Nicholas D Lane, Sourav Bhattacharya, Petko Georgiev, Claudio Forlivesi, Lei Jiao, Lorena Qendro, and Fahim Kawsar. 2016. Deepx: A software accelerator for low-power deep learning inference on mobile devices. In *Information Processing in Sensor Networks (IPSN), 2016 15th ACM/IEEE International Conference on*. IEEE, 1–12.

[21] Daniel Lezcano. 2017. LXC - Linux Containers. (2017). Retrieved Apr 22, 2017 from https://github.com/lxc/lxc

[22] Peng Liu, Dale Willis, and Suman Banerjee. 2016. ParaDrop: Enabling Lightweight Multi-tenancy at the NetworkâĂŹs Extreme Edge. In *Edge Computing (SEC), IEEE/ACM Symposium on*. IEEE, 1–13.

[23] JR Okajima. 2017. Aufs. (2017). Retrieved Apr 22, 2017 from http://aufs.sourceforge.net/aufs3/man.html

[24] OpenVZ. 2017. OpenVZ Virtuozzo Containers Wiki. (2017). Retrieved Apr 22, 2017 from https://openvz.org/Main_Page

[25] OpenVZ. 2017. Virtuozzo Storage. (2017). Retrieved Apr 22, 2017 from https://openvz.org/Virtuozzo_Storage

[26] Milan Patel, B Naughton, C Chan, N Sprecher, S Abeta, A Neal, and others. 2014. Mobile-edge computing introductory technical white paper. *White Paper, Mobile-edge Computing (MEC) industry initiative* (2014).

[27] Rob Pike, Dave Presotto, Ken Thompson, Howard Trickey, and Phil Winterbottom. 1992. The use of name spaces in Plan 9. In *Proceedings of the 5th workshop on ACM SIGOPS European workshop: Models and paradigms for distributed systems structuring*. ACM, 1–5.

[28] Yuqing Qiu. 2016. *Evaluating and Improving LXC Container Migration between Cloudlets Using Multipath TCP.* Ph.D. Dissertation. Carleton University Ottawa.

[29] Akamai Releases Second Quarter. 2014. State of the Internet Report. *Akamai: http://www. akamai. com/html/about/press/releases/2014/press-093014. html. Accessed* 2 (2014).

[30] Rami Rosen. 2013. Resource management: Linux kernel namespaces and cgroups. *Haifux, May* 186 (2013).

[31] Rami Rosen. 2014. Linux containers and the future cloud. *Linux J* 2014, 240 (2014).

[32] Mahadev Satyanarayanan. 2017. The Emergence of Edge Computing. *Computer* 50, 1 (2017), 30–39.

[33] Mahadev Satyanarayanan, Paramvir Bahl, Ramón Caceres, and Nigel Davies. 2009. The case for vm-based cloudlets in mobile computing. *IEEE pervasive Computing* 8, 4 (2009).

[34] Weisong Shi, Jie Cao, Quan Zhang, Youhuizi Li, and Lanyu Xu. 2016. Edge computing: Vision and challenges. *IEEE Internet of Things Journal* 3, 5 (2016), 637–646.

[35] Andrew Vagin. 2017. FOSDEM 2015 - Live migration for containers is around the corner. *Online, https://archive.fosdem.org/2015/schedule/event/livemigration/* (2017).

[36] Shanhe Yi, Zijiang Hao, Zhengrui Qin, and Qun Li. 2015. Fog computing: Platform and applications. In *Hot Topics in Web Systems and Technologies (HotWeb), 2015 Third IEEE Workshop on*. IEEE, 73–78.

[37] Shanhe Yi, Cheng Li, and Qun Li. 2015. A survey of fog computing: concepts, applications and issues. In *Proceedings of the 2015 Workshop on Mobile Big Data*. ACM, 37–42.

[38] Shanhe Yi, Zhengrui Qin, and Qun Li. 2015. Security and privacy issues of fog computing: A survey. In *International Conference on Wireless Algorithms, Systems, and Applications*. Springer, 685–695.