ISSN (Online) 2394-2320

International Journal of Engineering Research in Computer Science and Engineering (IJERCSE)
Vol 3, Issue 11, November 2016

Efficient way of Migrating Docker Images

[1]Shiva Kumar Pentyala, [2]Goutam Sanyal
[1][2] National Institute of Technology - Durgapur
[1]shivakumar.pentyala@gmail.com [2] nitgsanyal@gmail.com

***Abstract:* -- Docker is a tool designed to make it easier to create, deploy, and run applications by using containers. Containers allow a developer to package up an application with all of the parts it needs, such as libraries and other dependencies, and ship it all out as one package. Docker containers wrap a piece of software in a complete filesystem that contains everything needed to run: code, runtime, system tools and system libraries – anything that can be installed on a server. This guarantees that the software will always run the same, regardless of its environment. Migration of OS instances across distinct physical hosts is a useful tool for administration of data centers and clusters. The main purpose of migrating is load balancing. It also provides for management, maintenance and considerable reduction in energy consumed. In the process of migration, while the OSes run, we can achieve high performances with minimal service failures. In this paper we will suggest an efficient approach to minimize the energy and time required for container migration.**

***Keywords:*-- Docker, Migration of OS instances, load balancing.**

I. INTRODUCTION

Docker is a bit like a virtual machine. But unlike a virtual machine, rather than creating a whole virtual operating system, Docker allows applications to use the same Linux kernel as the system that they're running on and only requires applications be shipped with things not already running on the host computer. This gives a significant performance boost and reduces the size of the application.
The key difference between containers and VMs is that while the hypervisor abstracts an entire device, containers just abstract the operating system kernel. Containers are the products of operating system virtualization. They provide a lightweight virtual environment that groups and isolates a set of processes and resources such as memory, CPU, disk, etc., from the host and any other containers. The isolation guarantees that any processes inside the container cannot see any processes or resources outside the containers. Building of Docker Containers use Copy on Write strategy (CoW). Any RUN commands you specify in the Dockerfile creates a new layer for the container. In the end when you run your container, Docker combines these layers and runs your containers. Layering helps Docker to reduce duplication and increases the re-use. This is very helpful when you want to create different containers for your components. You can start with a base image that is common for all the components and then just add layers that are specific to your component. Layering also helps when you want to rollback your changes as you can simply switch to the old layers, and there is almost no overhead involved in doing so. When you create a new container, you add a new, thin, writable layer on top of the underlying stack. This layer is often called the "container layer". All changes made to the running container - such as writing new files, modifying existing files, and deleting files - are written to this thin writable container layer. The diagram(Figure 1)below shows a container based on the Ubuntu 15.04 image.

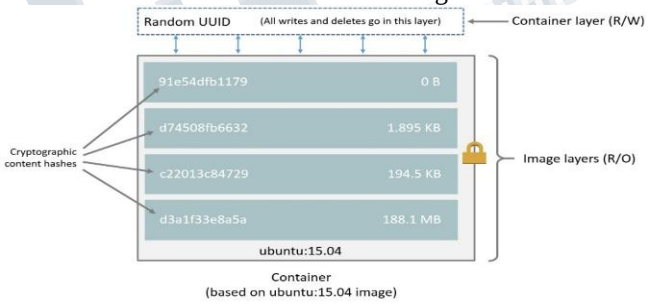


Fig. 1 Container based on the Ubuntu 15.04 image

II. RELATED WORK

All image and container layers exist inside the Docker host's local storage area and are managed by the storage driver. On Linux-based Docker hosts this is usually located under /var/lib/docker/. The Docker client reports on image layers when instructed to pull and push images with docker pull and docker push. The command below pulls the ubuntu:15.04 Docker image from Docker Hub.
$ docker pull ubuntu:15.04
15.04: Pulling from library/Ubuntu
1ba8ac955b97: Pull complete f157c4e5ede7: Pull complete 0b7e98f84c4c: Pull complete a3ed95caeb02: Pull complete Digest:
sha256:5e279a9df07990286cce22e1b0f5b0490629ca6d187698746ae5e28e604a640e Status: Downloaded newer image for ubuntu:15.04

ISSN (Online) 2394-2320

International Journal of Engineering Research in Computer Science and Engineering (IJERCSE)
Vol 3, Issue 11, November 2016

From the output, you'll see that the command actually pulls 4 image layers. Each of the above lines lists an image layer and its UUID or cryptographic hash. The combination of these four layers makes up the ubuntu:15.04 Docker image. Each of these layers is stored in its own directory inside the Docker host's local storage. If we make changes to the Dockerfile and build the new image then below diagram shows the shared layers and newly formed layers. changed-ubuntu image does not have its own copies of every layer. As can be seen in the Fig. 2 below, the new image is sharing its four underlying layers with the ubuntu:15.04 image.

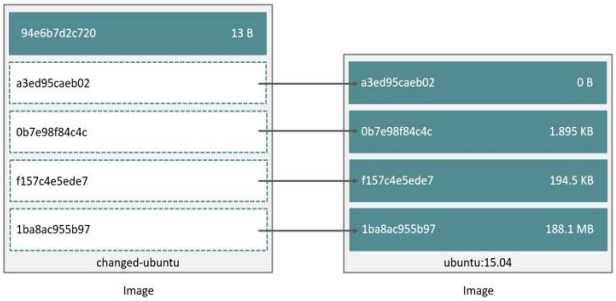
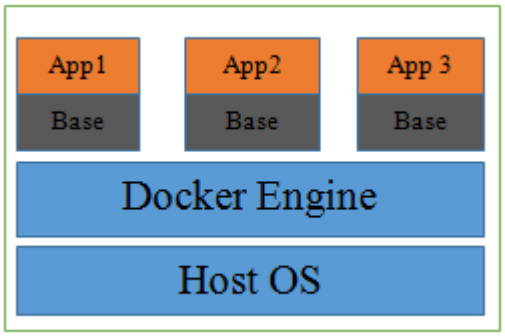


Fig. 2 Comparision of new image with old ubuntu:15.04 image

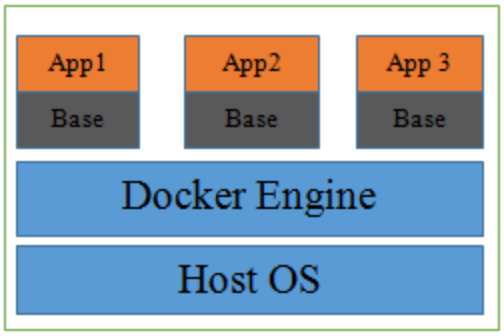
As you can see, the 94e6b7d2c720layer is only consuming 12 Bytes of disk space. This means that the changed-ubuntu image we just created is only consuming an additional 12 Bytes of disk space on the Docker host - all layers below the 94e6b7d2c720 layer already exist on the Docker host and are shared by other images. This sharing of image layers is what makes Docker images and containers so space efficient.

Images or containers can be migrated from one machine to another using docker-save, where the image is converted in to a .tar file and using docker-load in the second machine we can retrieve the image from the tar file. Docker-save wipes all the parent layers and creates a tar file. By this approach if we have multiple applications with common layer then this layer would be present in all the tar files, thereby increasing the filesize that is migrated from one system to another. So this approach is not efficient if we want to migrate the containers with common layers to another system in our local network. For example if we have three applications which use a common base layer then the migration of these applications on to an another system using docker-save followed by docker-load involves the transfer of common base layer three times.

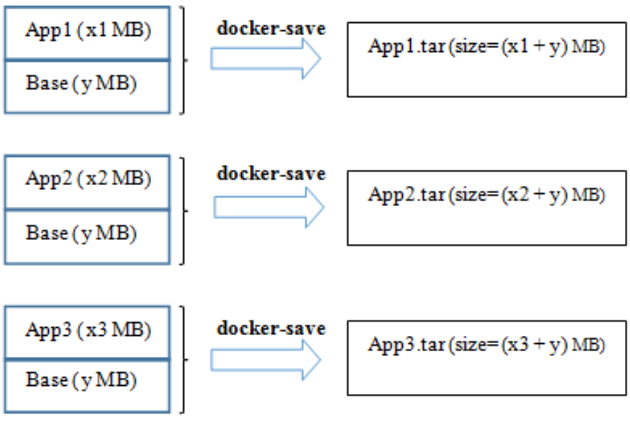
Machine - 1



Machine – 2



Generally to replicate an application we use docker save in M1 and docker load in M2. Docker save creates a tar file of an image (including base image). If we consider 3 applications of sizes x1 MB , x2 MB, x3 MB respectively and a common base layer of size y MB then the migration of these applications to another system involves an additional transfer of 2y MB.



ISSN (Online) 2394-2320

International Journal of Engineering Research in Computer Science and Engineering (IJERCSE)
Vol 3, Issue 11, November 2016

Fig. 3 Overview of tar file sizes for different applications

As shown in the Fig. 3, docker-save creates a tar file irrespective of the common layers. There by leading to an additional transfer of 2y MB, which consumes a lot of time and energy.

III. PROPOSED APPROACH

We can overcome the above mentioned problem by untaring and removing the common layers that are preset on both the machines and taring them back. This method significantly decreases the size of the tar file to be migrated. This approach is illustrated as below.

1. Untar the file and remove the base layer and tar it back on Machine-1:

Using 'docker history <image name>' command you can identify the base image UUID, which helps in removing this base layer from the untared file. After the removal of this base layer tar the files again. Generally, base layer size will be greater than the remaining layers of that image. So, concentrating primarily on base image would give us the noticeable results.

2. Untaring the file and adding the base and tar it back on Machine-2:

Untar the file that was migrated from machine-1 and add the base layer that was already available on Machine-2. Tar it back after adding the required base layer. For simple scenario we considered that they share only base layer as common. In future the same steps can be followed for other common layers than base layer. Using this approach we can decrease the size of the file that is to be migrated to a considerable amount. This approach will be highly beneficial for the users who are migrating the files in a network without using Ethernet. Users who are using file transfer protocols like SCP, FTP, RSync would be highly benefited using our proposed approach.

IV. EXPERIMENT AND PERFORMANCE EVALUATION

A. Time and Size Related:

The experiment was performed between two identical machines(M1 and M2), and each machine runs Raspbian Jessie and Docker installed on them were used to compare the time and energy consumed in migration of Images from machine M1 to M2. The Machines were equipped with Raspberry Pi-3 as hardware. We use OLSR protocol to form a wireless ad hoc network. We performed the experiment using three different containers for serving applications named Below in TABLE I. Apache, Nginx, MySQL. These three containers used base images of the resin/rpi-raspbian, armbuild/Ubuntu and armbuild/debian respectively. Three Applications were present in machine M1 with following specifications –

Table I

Applications and Corresponding Base Image, Size

App	Image Name	Size (MB)	Base Image Name	Size (MB)
App1	Apache:latest	194.1	resin/rpi-raspbian	117.2
App2	Nginx:latest	179.5	armbuild/Ubuntu	163.2
App3	Mysql:latest	256.6	armbuild/debain	139.3

We migrated the Images in two ways, one using our method and other using docker-save method and we observed time and energy consumed in migration in both the methods and compared them. The below graph depicts the time taken for each process of the migration of different applications from M1 to M2.We used network protocol called SCP to transfer files between the two machines.

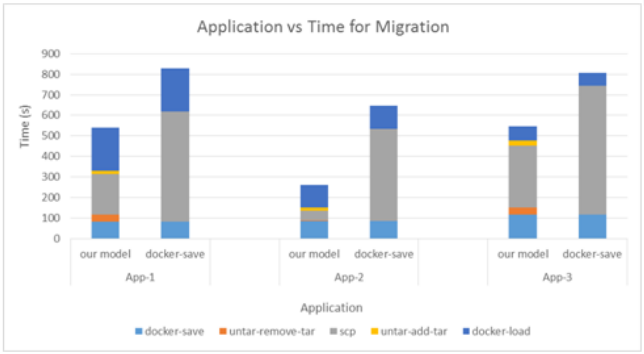
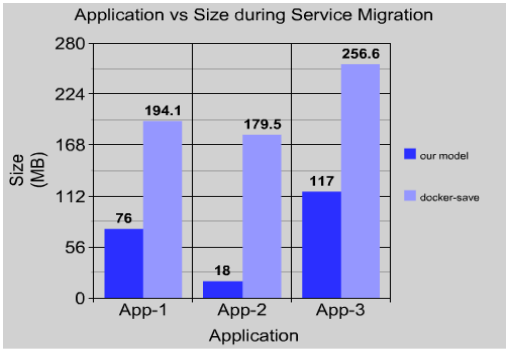


Fig. 4 Application vs Time for Migration

As shown in the Fig. 4 the time required for Service Migration is significantly less for our method than the docker-save method. The size of the files that are migrated was also significantly decreased in our method. Fig. 5 show the sizes of different tar files of the applications that are to be migrated from M1 to M2 for our method and docker-save method.

ISSN (Online) 2394-2320

International Journal of Engineering Research in Computer Science and Engineering (IJERCSE)
Vol 3, Issue 11, November 2016



	our model	docker-save
App-1	76	194.1
App-2	18	179.5
App-3	117	256.6

Fig. 5 Application vs Size during Migration

B. Energy Related:

The two Machines with hardware Raspberry Pi 3 has two wireless interfaces, one of which is for the OLSR Ad-Hoc mesh and the other for acting as the wireless access point. The OLSR Ad-hoc network connects the two Raspberry Pi powered machines. The Pi 3's inbuilt Wi-Fi is configured for OLSR and an Edimax USB Wi-Fi dongle is configured for the access point to which the user connects to access web services. Both the Pi runs Raspbian Jessie and Docker installed in it. Fig. 6 show the experimental setup of the proposed work that measures the current drawn by the raspberry pi at a particular instant. We have used ADS1015 chip for Analog to Digital Signal conversion and it's easy to use this chip with the Raspberry Pi using its I2C communication bus. We used INA 169 chip for current monitoring.

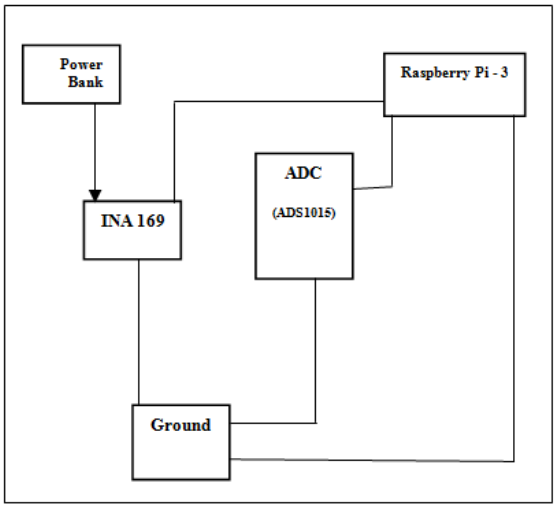
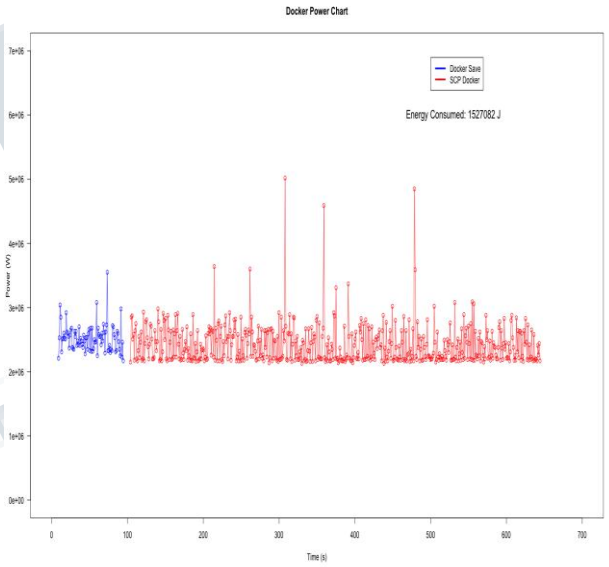


Fig. 6 Experimental setup of the proposed work to measure Current

The raspberry pi is booted up and once it reaches its steady value of current this experiment is performed. We used INA 169 to measure the current drawn by raspberry pi. INA 169 is the high-side, unipolar, current shunt monitor. ADS 1015 is used as Analog to Digital converter, which converts analog signal into digital signal. From the formula P=V.I, we obtain Power as V is constant and equal to 5V.Area under Power-Time graph gives the energy consumed over that time. We observed that docker-save model approximately consumed 1527 KJ whereas our model consumed only 798 KJ of energy. Therefore our model saves approximately 729 KJ of energy.

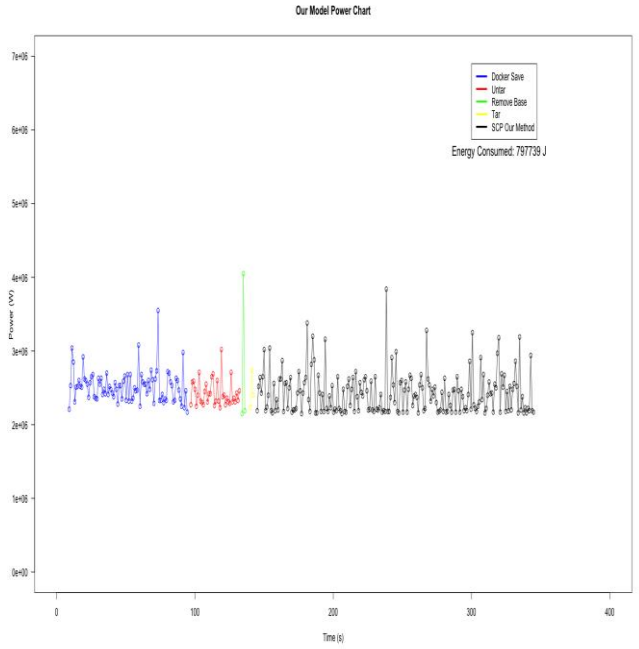
Power vs Time for docker-save method



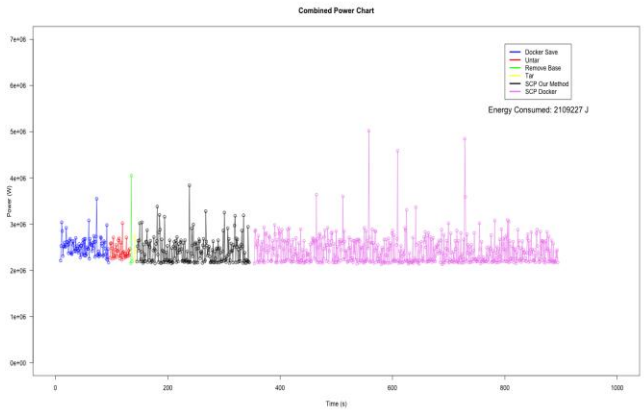
Power vs Time for our method:

ISSN (Online) 2394-2320

International Journal of Engineering Research in Computer Science and Engineering (IJERCSE)
Vol 3, Issue 11, November 2016



Power vs Time for both the methods:



V. CONCLUSION

This study focuses on the techniques of Docker container migration and how to improve it. To improve migration time the amount of space needed to transfer during migration must be minimized. This minimization is achieved with the help of above approach. The Image migration time and energy consumed reduced significantly for different applications by 50% approximately.

Acknowledgement

Deepest gratitude would like to be delivered to all those who have given advices and help on how to make this work possible.

REFERENCES

[1] Chenying Yu1 and Fei Huan2 ," Live Migration of Docker Containers through Logging and Replay," Shanghai Jiao Tong University, China.

[2] Jake Roemer, Mark Groman, Zhengyu Yang, Yufeng Wang, Chiu C. Tan, and Ningfang Mi," Improving Virtual Machine Migration via Deduplication," Department of Computer and Information Sciences, Temple University ‡ Department of Computer Science and Engineering, Lehigh University, Department of Electrical and Computer Engineering, Northeastern University.

[3] N. Kratzke, "A lightweight virtualization cluster reference architecture derived from open source paas platforms," Open Journal of Mobile Computing & Cloud Computing.

[4] Information on https:// https://docs.docker.com/

[5] C. C. Keir, C. Clark, K. Fraser, S. H, J. G. Hansen , E. Jul, C. Limpach, I. Pratt, and A. Warfield, "Live migration of virtual machines," in In Proceedings of the 2nd ACM/USENIX Symposium on Networked Systems Design and Implementation (NSDI, 2005, pp. 273–286.

[6] F. Ma, F. Liu and Z. Liu," Live virtual machine migration based on improved pre-copy approach, "IEEE International Conference on Software Engineering and Service Sciences, IEEE Conference Publications, New York, 2010, pp. 230-233.

[7] H. Jin, L. Deng, S. Wu, X. Shi and X. Pan," Live migration of virtual machines by adaptively compressing memory pages," Future Generation Comp. Syst., 38 (2014) 23-35.

[8] M. Sun and W. Ren, "Improvement on dynamic migration technology of virtual machine based on Xen," International Forum on Strategic Technology, 2 (2013) 124-127.