

# Towards Docker Live Migration

Master of Science in Technology Thesis  
University of Turku  
Department of Future Technologies  
Faculty of Mathematics and Natural Science  
May 2018  
Anik Barua

Supervisor:  
Farhoud Hosseinpour

Co-Supervisor:  
Juha Plosila

The originality of this thesis has been checked in accordance with the University of Turku quality assurance system using the Turnitin Originality Check service.

UNIVERSITY OF TURKU  
Department of Future Technologies

ANIK BARUA: Towards Docker Live migration

Master of Science in Technology Thesis, 48 p., 0 app. p.  
Faculty of Mathematics and Natural Science  
May 2018

The objective of this thesis was to find out how we can do live migration of a running process using a lightweight container management system known as “Docker”. We will introduce the concept of virtualization with details structure as well as the use of Docker. Live migration is experimental feature of Docker. There are some certain conditions to set up the system environment; our aim is to discuss it in details. We have used 64 bit UBUNTU 16.04.4 LTS as the host OS and docker version 17.09.1~ce is used with experimental feature enabled. We have showed how we have to prepare source and destination environment and all the steps to successfully execute live migration. This live migration requires a time which is called as migration time, transfer time or delay time. We have also calculated this transfer time for different type of source and destination. After that, we have compared and analyzed the data which shows that for a same migration process that transfer time can be different. Also, a graph based on the experimental data shows that the tarball size used for the migration increases proportionately with the time.

Keywords: Docker, Virtualization, Live Migration, Container, Transfer Time

## Acknowledgement

I would like to cordially thank my supervisor Farhoud Hosseinpour for giving me the opportunity to work under his supervision. Without his direct and clear guidelines, it would not have been possible to complete this thesis. I am very grateful to him for his continuous support and direction all the long way. Whenever I got stuck somewhere, he explained me the whole scenario several times to the exact point. His regular communication always kept me in track and it helped me greatly to keep motivated along all the way.

Beside my supervisor, I want to give special thanks to my co-supervisor Juha Plosila. He is such a nice person to be very supportive from the very beginning.

Most importantly, I would like to convey my gratitude to my parents and lovely family members. They always keep inspiring me from the long way home so that I can complete master's degree from University of Turku.

Finally, completing this thesis required something more than academic matters. I have my friends living in Turku from different countries. They always inspired me to complete the thesis. Their support greatly helped me to carry on this thesis in a calm manner.

# Contents

Abstract

Acknowledgement

Contents

List of Figures

List of Tables

Abbreviations

Chapter 01: Introduction .....	1
1.1 Problem Statement .....	1
1.2 Scope and Thesis Methodology .....	2
1.3 Thesis Outcome .....	3
1.4 Thesis Structure .....	3
Chapter 02: Background and Literature Review .....	4
2.1 Docker .....	4
2.2 Docker components .....	5
2.2.1 Docker Images .....	6
2.2.2 Docker client and server .....	6
2.2.3 Registries .....	6
2.2.4 Docker Containers .....	8
2.3 The concept of Virtualization .....	9
2.4 Virtual Machine (VM) vs. Docker Container .....	11
2.5 Benefits of using Docker .....	13

2.6 CRIU .....	14
2.7 Live Migration.....	14
2.8 Challenges of Live Migration.....	16
2.9 Fog and Edge Computing.....	17
2.10 Related Work.....	18
2.11 Summary .....	20
Chapter 03: Methodology and System Setup.....	21
3.1 System setup for Live Migration:.....	21
3.2 Preparing the servers .....	22
3.3 System setup for calculating migration time .....	25
3.4 Summary .....	29
Chapter 04: Implementation.....	30
4.1 Experiment of live Migration .....	31
4.2 Operations in Source Node.....	31
4.3 Operations in Destination Server .....	36
4.4 Calculating transfer time .....	38
4.5 Comparison of the delay time .....	39
4.6 Future Work .....	43
4.7 Summary .....	45
Chapter 05: Conclusion.....	46
References .....	47

# List of Figures

Figure 2.1: Docker Architecture .....	5
Figure 2.2: Top Technologies on Docker .....	8
Figure 2.3: Docker mission .....	9
Figure 2.4: Virtual Machine structure .....	10
Figure 2.5: Virtual Machine .....	11
Figure 2.6: Docker container and Virtual Machine .....	12
Figure 2.7: CRIU .....	14
Figure 2.8: Fog Computing System (Distributed) .....	15
Figure 2.9: Transfer time of live migration .....	16
Figure 2.10: Fog Computing .....	17
Figure 3.1: Enabling experimental feature .....	23
Figure 3.2: Downgrading Docker version .....	24
Figure 3.3: Installing CRIU .....	24
Figure 3.4: Docker version .....	25
Figure 3.5: Live migration .....	26
Figure 3.6: System configuration .....	27
Figure 3.7: Installing hardinfo .....	28
Figure 3.8: Checking hardware information .....	28
Figure 3.9: Checking CPU information .....	29
Figure 4.1: Live migration concept .....	30
Figure 4.2: Checkpoint and Restore .....	31
Figure 4.3: Running “busybox” image .....	32
Figure 4.4: Output of the process .....	32
Figure 4.5: Checkpoint created .....	33
Figure 4.6: Output after the checkpoint .....	34
Figure 4.7: Directory path .....	35
Figure 4.8: Checking tarball .....	35
Figure 4.9: Transferring the process .....	36

Figure 4.10: Untar the received process.....	36
Figure 4.11: Checkpoint checking .....	37
Figure 4.12: New container created .....	37
Figure 4.13: Resuming container .....	38
Figure 4.14: Migration time .....	39
Figure 4.15: Migration using different communication technology .....	43
Figure 4.16: Migration in a sample edge network systems.....	44

# List of Tables

Table 1.1: Most used docker images .....	7
Table 4.1: Comparison of migration time .....	40



# Abbreviations

IoT	: Internet of Things
OS	: Operating System
LTS	: Long Term Support
CRIU	: Checkpoint/Restore In Userspace
API	: Application Program Interface
LXC	: Linux Containers
VMs	: Virtual Machines
GBs	: Gigabytes
AWS	: Amazon Web Services
TCP/IP	: Transmission Control Protocol / Internet Protocol
CPU	: Central Processing Unit

# Chapter 01: Introduction

Internet of Things (IoT) has become a very influential concept in the modern technology systems. In general, IoT is a network of several physical objects which communicate with each other by using network connectivity. Some of the wide application areas of IoT are – smart home, smart agriculture, smart cities, smart transportation, healthcare system, large scale industries and so on. The use of IoT devices is growing exponentially and in number it will be more than 20 billion within 2020 [01]. Within 2030 more than 50 billion smart devices is predicted to be connected with each other. The more IoT devices mean that the more use of different sensors, network systems and data processing subsystem. Here, Fog computing is a very efficient platform for the data processing at the edge of the network. For a large scale IoT system where lots of sensors, network system and complex data processing subsystem are requires, we can use “Docker” to make to whole system more efficient. Docker is a tool that is used to run applications in an isolated environment. It is small, isolated and executable software that has everything included to run. It will make the application process easier and faster by using containers which consists of all the necessary libraries and dependencies.

There are several advantages of using docker in IoT applications. First of all, it will make the deployment system easier and faster. We can also make the system scalable, intelligent, energy efficient and secure. Inside docker, every app uses its own container; therefore, it doesn't overlap with any other application. Also, as docker does not need Hypervisor and Guest OS as virtual machine which make the container lightweight.

## 1.1 Problem Statement

Although there are lots of benefits of using docker, one of the most important features is that it supports live migration. It means we can run a process into machine and then transfer the running process into another machine. One challenge is that, live migration is an experimental feature of Docker. Therefore, it can be done only in Linux Operating System with few fixed versions of docker. Unfortunately, the latest version of docker does not

support this experimental feature. Moreover, as Docker is quite a new technology hence access to the exact resources with proper documentation about live migration was very challenging.

Therefore, in this thesis, our first task is to introduce the lightweight container management system called Docker. We will broadly discuss about the structure and the components of Docker. Moreover, the concept of virtualization, advantages and limitations will also be taken into consideration. After that, we will experiment the live migration. A workflow and documentation will be created as well. Being able to set up the test environment correctly will be a very important issue here and therefore, all the necessary conditions and mechanism will be explored in this thesis. Later on, the challenges of Docker will be discussed.

In the second experimental part of this thesis, our aim is to calculate the delay time (also known as file transfer time) that requires to migrate the process from one machine to another machine. The same migration will be done between machine to machine, server to server and machine to server. Among them, one will be working as source machine and other one will be destination machine. We will analyze the delay time for the live migration and will come up with a conclusion about the best way for migration.

## 1.2 Scope and Thesis Methodology

The thesis is based on the best container platform in the recent years – Docker. We can deploy and run applications using docker container very efficiently and easily. The experiment has been done only in Linux operating system with the experimental feature enabled. 64 bit UBUNTU 16.04.4 LTS has been used as the Linux host and the docker version 17.09.1-ce has been used to experiment the live migration. “Windows” or “macOS” is has not been taken into account as the scope of this thesis.

The thesis consists of a detailed review of the container system and virtualization techniques. The structures, advantage and limitations has been discussed. It also introduces the CRIU and the concept of live migration. Then the proof of concept has been done with a practical experiment. System environment setup and requirements has been explored

precisely. Finally, the migration time has been calculated and compared for different system environment.

### 1.3 Thesis Outcome

In this thesis, we have found how we can migrate a running process from one machine to another machine using Docker. For example, using container system we will run a process, transfer to the destination and resume the process again in the destination machine. This live migration can be used in the fog computing system to a great extent and it can make the system very efficient. We have also found that although this is a live migration, it requires a migration time that varies from 0.1 second to 0.5 second for a simple container migration application. For a same application this migration time can vary depending on the system environment of host and destination.

### 1.4 Thesis Structure

In the chapter 2, docker container and virtualization has been discussed and compared. The concept of CRIU and live migration has been explored in details. Also, the benefits and limitation of these systems have been discussed besides the background studies. In the chapter 3, the methodology to do the live migration experiment has been presented. There are some conditions to set up the system as required, which has been discussed. In chapter 4, we have showed how have done the experiment with step by step as well as the docker commands. The migration time has also been calculated and compared for different experimental environment. The challenges to do the experiment and the future works have also been explored. Finally, in chapter 5, the whole thesis wok has been summarized.

## Chapter 02: Background and Literature Review

### 2.1 Docker

Docker is a very well-recognized container management system with a huge community group. A recent survey says that, almost 93 percent participant willing to use docker. It is relatively easier to use Docker than any other system and it has a public “docker hub” that contains more almost 450K container images. It is possible to upload or download a container image very easily within a few seconds. It also has an API to easily manage containers.

Docker is one of the best and practical uses of Linux containers system. Here, Linux works as the host OS [02]. The idea here is that, each container will work as an individual application. We can think about a scenario that one application is based on multiple components; in this case multiple containers will be used for that single application. A docker container is consists of several layers in the operating system. If we need to change or add a new component, it put on to the previously existed layers.

Docker is an engine that is used to automate the deployment of application. It is lightweight and open-source. It is easier and faster to run a code efficiently from testing to production. One of the major targets of docker is to lower the complexity from development time to test and run an application. It gained huge popularity in the recent years.

It is possible to dockerize and run an application within a minute. This is often compared with virtualization techniques. However, the major difference between container and virtualization techniques is that container only pack up the application and the necessary dependencies while virtualization pack the whole Operating system [03].

Docker is adjunct to the LXC which is the short form of Linux Container system. It is lightweight and it allows us to create a virtualization in the Operating System. It enables us to deploy and maintain any software applications very rapidly and easily into the servers.

This docker container allows programmers to pack up all the system libraries and dependencies together with the application container. It means the application can be run in any OS platform without any binding [01].

The recent version of docker is based on “libcontainer” although it is still possible to use LXC based system. Docker uses the host kernel like other containers; therefore, a separate guest operating system is not required.

## 2.2 Docker components

Docker is based on 4 major components (Figure 2.1). They are –

1. Docker Images
2. Docker client and server
3. Docker Containers
4. Registries

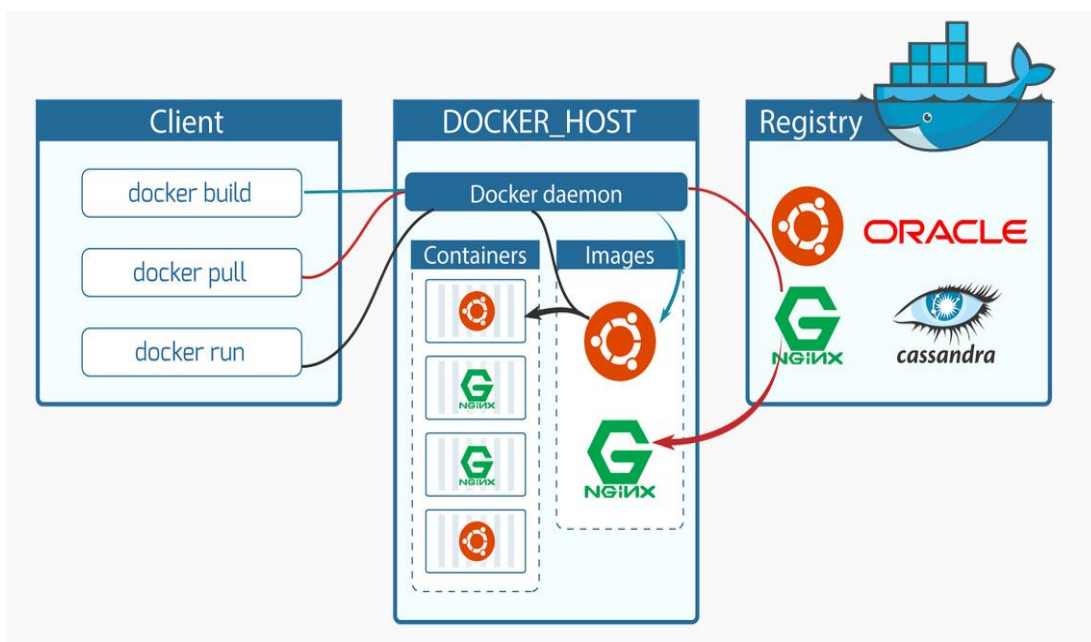


Figure 2.1: Docker Architecture (Saral Saxena, 2017)

### 2.2.1 Docker Images

Docker Images are referred as the building blocks of the whole Docker world [03]. Docker images carry necessary instructions that are used to build docker containers. We can also call it as the source code of a container. It uses Union file systems and executes several commands one after another. For example, it adds a file and runs a command to open a port. In the Docker hub, there are lots of built-in images that we can use instantly by a few lines of commands. We can also build our own docker image and run it.

### 2.2.2 Docker client and server

Docker application follows a client-server approach. Docker client connects with the Docker server (also known as Docker Daemon). It is possible to run the client and Docker daemon in the same server [03].

### 2.2.3 Registries

Docker registries are used as the storage place of the docker images. It can be divided into two parts – public and private. The public registry is used to store and share the images created by us or any organization. We have to register in the DockerHub account to be able to use it. We can find almost 400K public images in the DockerHub [04]. Some of the most popular images of official repositories are – alpine, nginx, httpd, redis, busybox, Ubuntu, mongo, mysql and hello-world. All of these images listed below (Table 1.1) have been pulled for more than 10 million times with just one line of pull command [05].

Image	Description	Command	Pulls
alpine	A 5 MB Alpine based Docker image	<i>docker pull alpine</i>	10M+
nginx	Official build of Nginx	<i>docker pull nginx</i>	10M+
httpd	The Apache HTTP Server Project	<i>docker pull httpd</i>	10M+
ubuntu	Linux operating system based on free software	<i>docker pull ubuntu</i>	10M+
mongo	MongoDB databases provide high availability and easy scalability	<i>docker pull mongo</i>	10M+
mysql	MySQL is a very popular open-source relational database management system (RDBMS)	<i>docker pull mysql</i>	10M+
hello-world	An example of basic Dockerization	<i>docker pull hello-world</i>	10M+
php	PHP scripting language used for web development	<i>docker pull php</i>	10M+
wordpress	Content management system based on plugins, widgets, and themes	<i>docker pull wordpress</i>	10M+
python	An open-source and object-oriented, programming language	<i>docker pull python</i>	10M+

Table 1.1: Most used docker images



The following figure 2.2 shows the most popular technologies on Docker.

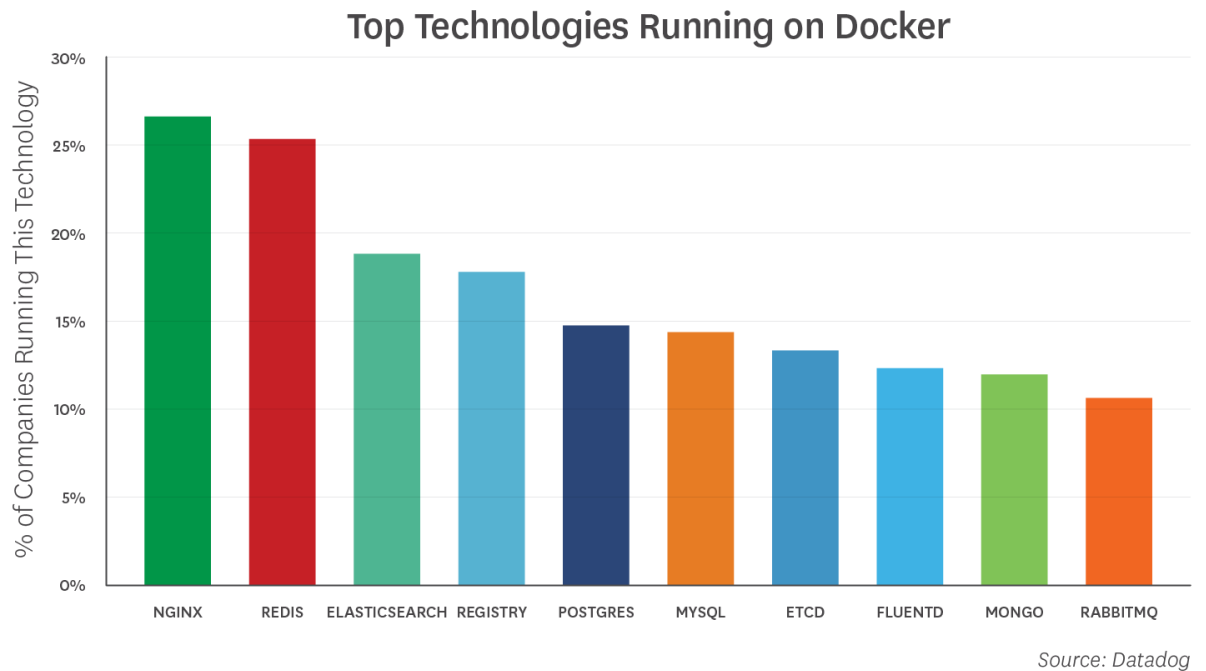


Figure 2.2: Top Technologies on Docker

#### 2.2.4 Docker Containers

Containers are one of the most important parts of Docker system. It can be defined as an executable environment that contains an image. It can execute multiple processes as well. Docker container should incorporate only small-scale and logical segment of an application [06]. Here, every container contains a small application. If we have a fog computing system where we have several nodes, we can keep a container application in every node. This will ensure that every node is responsible to accomplish a small part of the total task.

The concept of Docker has been taken from a real life scenario, which is transporting goods using shipping system. We use it to transport different products from one place to another place. Here we will just replace the “goods” by using “software”. That means, we can send software to any place we want using Docker. It is also not necessary to check the element inside the container. It can be anything like a simple software program, a database system or even a server. The server may be web server or application server. So, whatever it is

inside the container, the process of execution is same. As the summary, container can be considered as a small and isolated virtual environment that carries all the needed dependencies to run an application [07].

We can also create and ship a docker application to anywhere. For example, an application has been made in a personal computer, then uploaded into the DockerHub using registries and it can be pulled and used from anywhere to any server or host. We can also do it in reverse way which made the Docker system portable and very efficient.

In overall, Docker follows 3 simple steps – Build, ship and run (Figure 2.3).

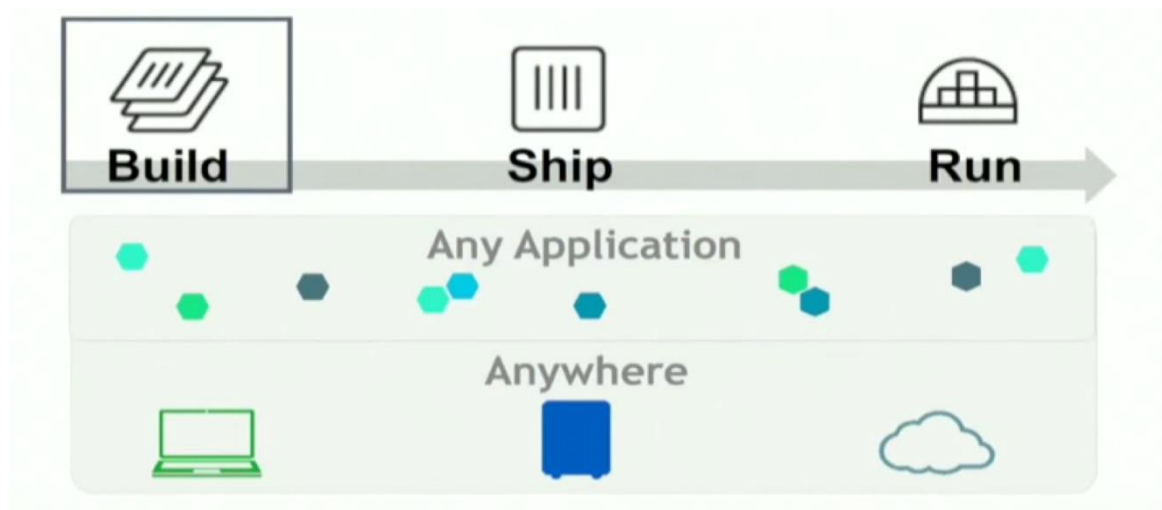


Figure 2.3: Docker mission (Source: neo4j)

## 2.3 The concept of Virtualization

The primary concept of Virtualization started in 1960, it means almost seven decades old concept. In that time, it was used to efficiently distribute the machine resources among applications [08]. Later the concept becomes recognized as “Hardware virtualization”. It will create a virtual machine that will work like an independent computer system. It will have its own Operating system as well. For example, we can use a Linux OS under a Windows OS. Here, Windows OS is the host machine and Linux OS is the guest machine. One host machine can support multiple guest machines (Figure 2.4).

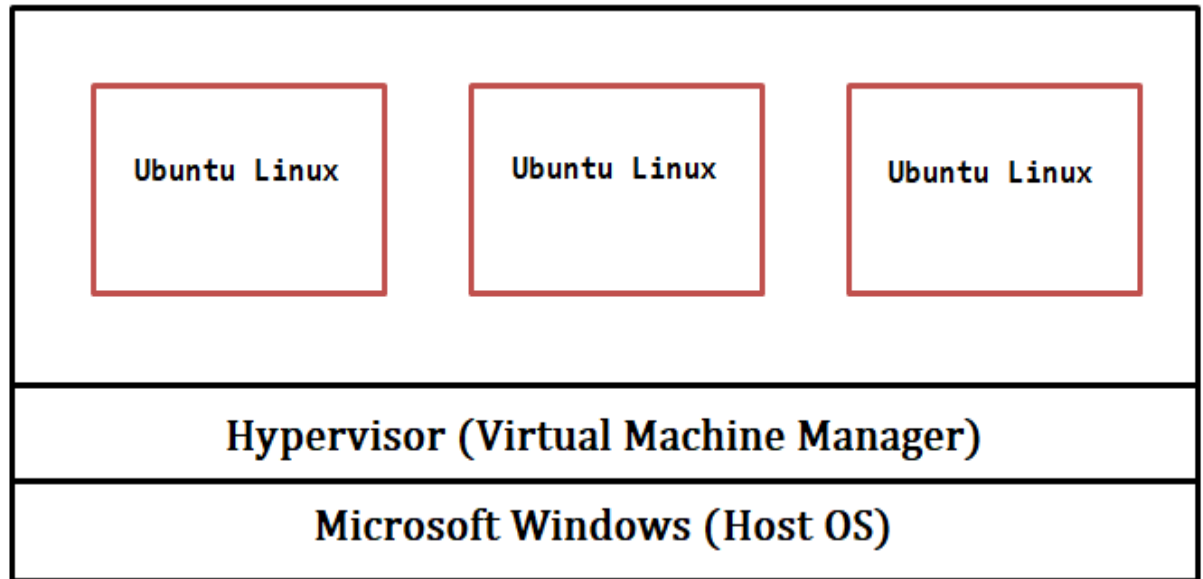


Figure 2.4: Virtual Machine structure

The features of virtualization enable server companies to provide their customers dedicated virtual machines as needed. Even for this thesis experiment of live migration, we will use two virtual machines from a renowned cloud service provider named “Digital Ocean”. Two major advantages of virtualization are as follows:

1. We can use multiple virtual machines using only one hardware infrastructure instead of setting different hardware for each virtual machine. Therefore, it greatly reduces the system setup cost. It also makes the system more efficient because the host OS can optimize the use of each virtual machine.
2. Another good point of virtualization is that, it can make each virtual machine isolated and unique as required. Every virtual machine can have its own operating system, memories, applications and necessary dependencies. It may happen that some applications cannot be executed in the same host operating system and requires a dedicated OS. In that case, instead of spending money to setup another physical machine, we can just use a virtual machine on the same hardware platform. It will make the hardware and software usages more energy efficient and productive.

## 2.4 Virtual Machine (VM) vs. Docker Container

The concept of Virtual machines (VMs) and Dockers seems similar but there are some major differences. If we consider a physical computer machine, then Virtual machine (VM) is the abstract version of it. One of the very important features of VMs is that if we convert one server to multiple servers using VM. Everything is happening here just within a single physical machine and it is done by the hypervisor.

Every VM is consists of a full version of Operating System, several application programs and other necessary dependencies. Altogether it takes several Gigabytes of memory spaces, more precisely about seven to ten Gigabytes (GBs). It takes from 2 to 3 minutes to start Virtual machines (VMs) depending on the system configuration and memory space. The following figure 2.5 shows a complete picture what we had discussed right now. It shows that within one host hardware and operating system, there may be multiple Virtual machines by using different Guest OS.

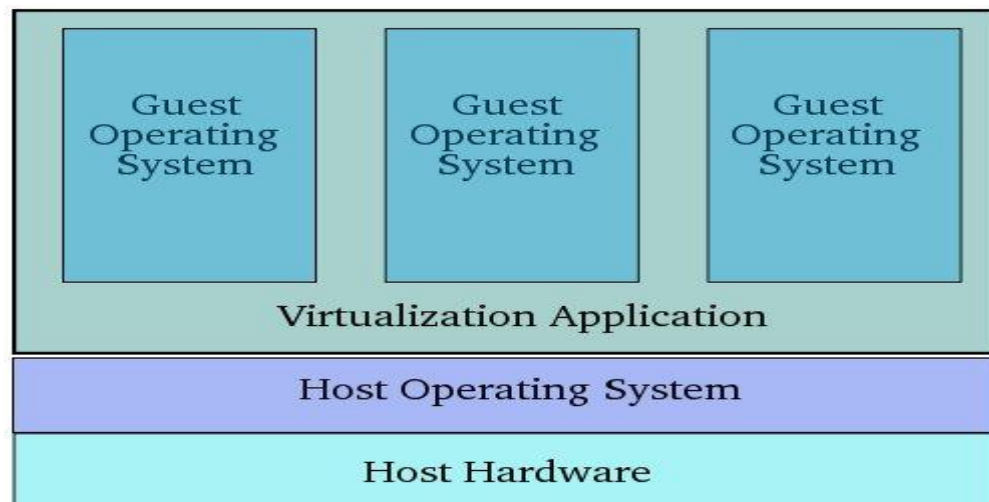


Figure 2.5: Virtual Machine

In overall, Virtual machines (VMs) are the abstract version of an Operating system whereas containers are the abstract version of executable applications. It also includes necessary dependencies and packages. It usually takes a couple of Megabytes and can execute in a few seconds, which means that Containers are way faster and lightweight than Virtual machines (VMs). Containers do not have any OS for their own but it uses the host OS.

Containers are well recognized model of micro-service. It is supported by “Kubernetes” which is a standard framework. Docker is one of the best and practical uses of Linux containers system. In addition to that, we can use more number of containers than Virtual machines (VMs) for a same hardware configuration. If we use the container in a mobile software system and face any problem, we can fix or reinstall that particular container instead of updating whole software system. It make the system very efficient and all these features made container systems much popular than Virtual machines (VMs). Distributed system can be managed efficiently using container.

As we know that, docker is a container based system and there are already a large number of containers being used by the users, therefore, it has also became important to be able to manage or organize the containers efficiently. As a solution of this, “Flocker” is being used to manage data and container. Here is the structural diagram of a Virtual Machine (VM) and Docker Container (Figure 2.6). It clearly shows the structural differences between Virtual Machine (VM) and Docker Container system.

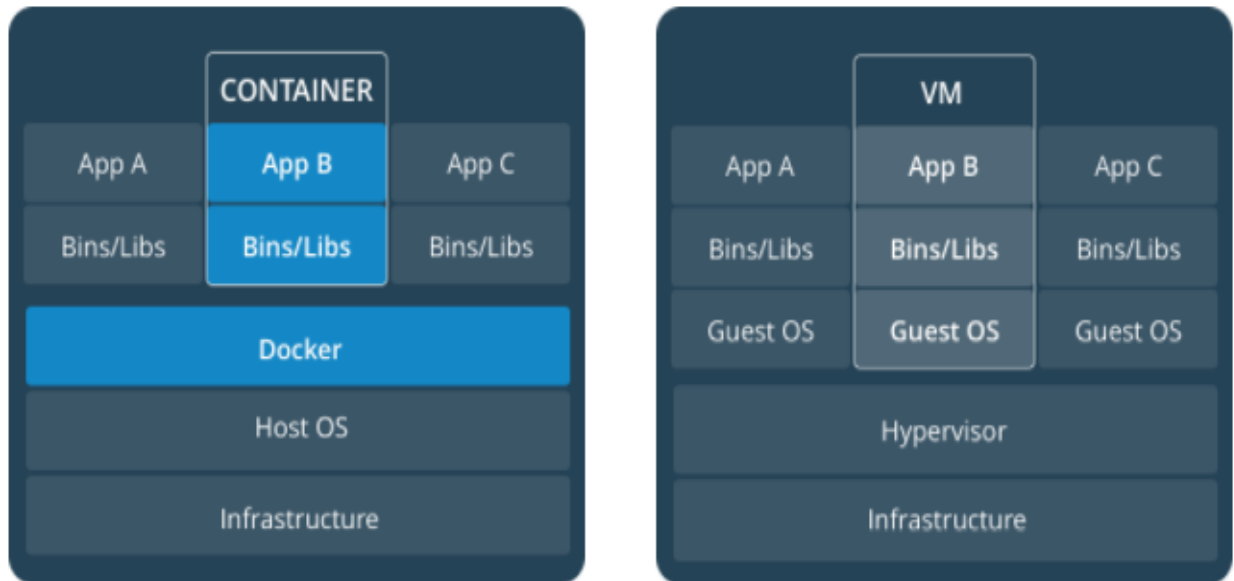


Figure 2.6: Docker container and Virtual Machine (Source: Docker)

## 2.5 Benefits of using Docker

Running applications in containers gained huge popularity in the IT industry in the recent years. It makes the whole process very faster by simple creating the application, deploy and run. The best part is that we can include all the necessary dependencies and plugins in the same container, which means everything is there as a package to run an application. Some of the major benefits of using Docker are as follows-

1. Docker makes the application deployment process very fast using the shortest runtime and smaller application size.
2. All the dependencies are packed into a single container which can be run into any platform or operating system. So, it does not have any compatibility issue.
3. Version control is very easier in Docker. From example, if we want to upgrade an application, it only collects the new dependencies or changes. That mean, it does not download the whole system again except the necessary changes. So, it makes version control incredibly faster by reducing redundancy. Alternatively, if we need to restore the previous version, we can do it within couple of minutes.
4. We can share containers in public using DockerHub. If now, it is also possible to have a private repository.
5. The size of docker images is compact. For example, one of the most popular images in DockerHub is “nginx:alpine” which is less than 5MB in size. So, it is very fast to deploy and run the application.
6. Using Docker make the application development procedure very. We can also build and test the application repeatedly. The developers can also examine, detect and fix bug rapidly.
7. Docker supports all the major cloud platforms like Amazon Web Services (AWS) or Compute Platform of Google. We can execute container operation Compute Engine (Google), Elastic Compute (EC2 from Amazon) or to any VirtualBox [09]. We can switch between these platforms for any containers that make the whole system portable.
8. “Multi-stage build” is one of the new and very exciting features which are available from Docker version 17.05. Docker image is consists with several layers and all the layers carry some instructions. Multi-stage build feature can efficiently optimize the Dockerfile and also easier to maintain.

## 2.6 CRIU

The term CRIU stands for “Checkpoint Restore In Userspace”. CRIU is a CR (Checkpoint and Restart) tool. It was a project of OpenVZ. This is an experimental feature of Docker. We can stop a running process and can continue again by using CRIU. This stop is done by saving the memory state which creates a file and transferred into another machine and environment [10]. And then we start the saved file again so that we can start again from the point it exactly stop (Figure 2.7). We can even customize or slow down the startup time of a container. This is still an experimental feature, so if we want to use that we have to enable the feature first which is disable by default.

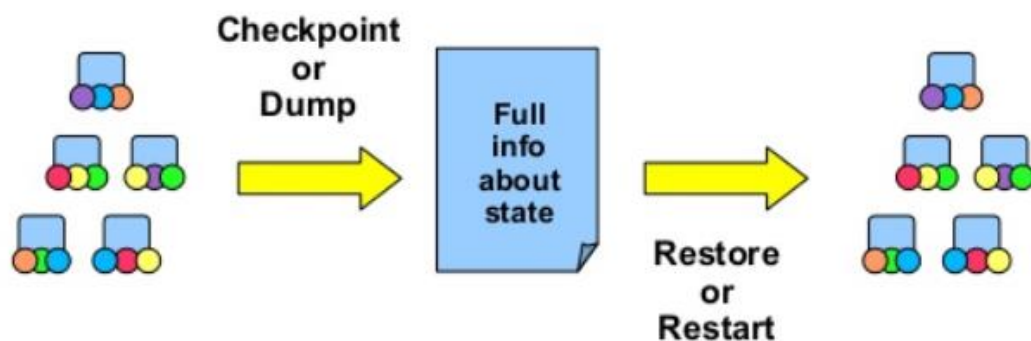


Figure 2.7: CRIU

## 2.7 Live Migration

Docker Live Migration is a very exciting feature of Docker. Theoretically, live migration means moving a running process to one machine to another machine without any interruption. The migration of applications can be done between two or more physical machines. Live migration greatly helps to efficiently manage a system.

The migration done between two machines ensures proper load balancing. We can consider a scenario of fog computing system that consists of several machines which run different process. Here, some of the machines may become much overloaded because of multiple processes running into a single machine.

It will increase the device temperature and power consumption and specially will decrease the performance which is not good for the device [12].

On the other hand, some of the machines may remain idle or having a very less amount of CPU utilization. Therefore, the concept is to distribute the working load equally using live migration. If a machine becomes overloaded then it will transfer the running process to another machine which is comparably freer to run applications.

In a fog computing system, we can consider these machines or devices as nodes (Figure 2.8). In a complex a fog computing system, there will be multiple lightweight nodes and usually the operating power is less. Therefore, load balancing is very crucial there to ensure energy efficient smooth operation. It can be done by live migration. A typical structure of a distributed fog computing system is as follow-

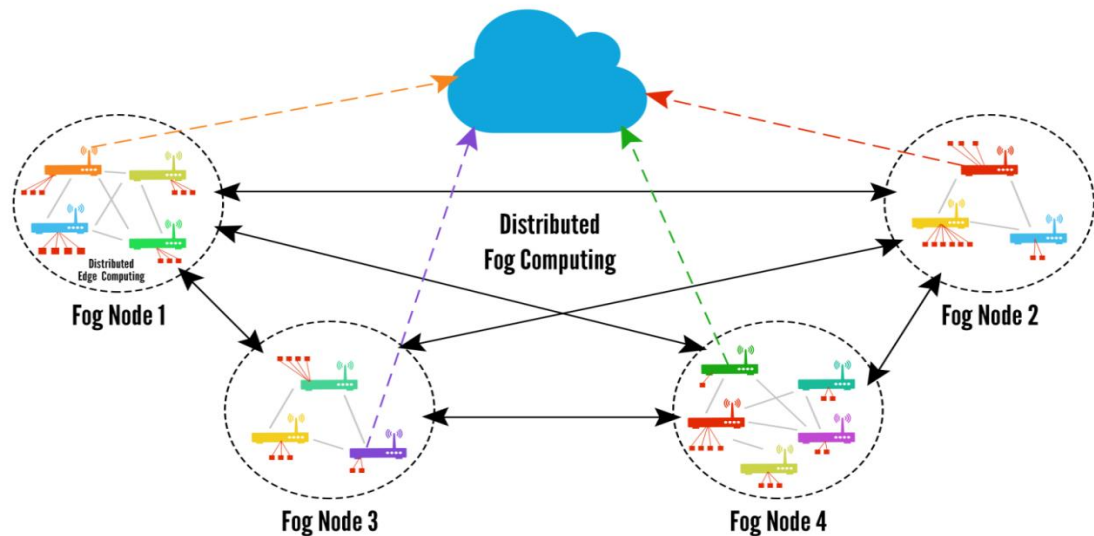


Figure 2.8: Fog Computing System (Distributed)

Due to the increased capacity of cloud architecture, we can transfer our application in different Virtual Machines (VM) or servers as needed. These VMs may be situated in different geographical locations. However, we need to ensure that the migration should be smooth enough that it does not affect the running application. The delay time should be very small so that it seems it has been done without disconnecting the running application. It can be done using CRIU or Docker Checkpoint/Restore [13].



We can divide into three steps.

1. Create checkpoint in the source machine
2. Copy and send the file to the destination
3. Restore the file again in the destination

In the case of Live Migration, there should not be any downtime while transferring the file from one server to another. Although there will be some delay, but must need to very smaller so that it can be ignored. If there are multiple servers, then we can use any scheduling system that will take care of the load balancing by itself (Figure 2.9).

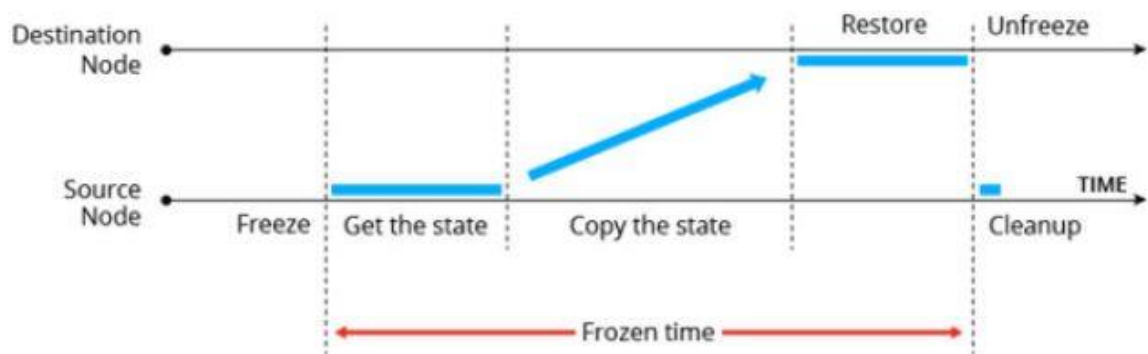


Figure 2.9: Transfer time of live migration [13]

## 2.8 Challenges of Live Migration

Although there are several advantages of using Live Migration, however, it has few challenges too. Some of them are as below –

1. As Live Migration needs to stop the process for some time, therefore, it may have some drop in the performance. In the most of the cases, for example, web application – it is acceptable to have a small frozen time. However, some critical real time applications are very sensitive to performance degradation, so it will be challenging to use Live Migration for those applications.
2. In the recent years, we have seen an increased popularity of cloud computing. We also have huge amount of data in the physical storages or in virtual machine which we typically call as Big Data.

If we need to use Live Migration in the case of Big Data, then it is a challenging task due to the network latency to transfer the huge dataset [10].

3. Some Docker container may use native API of a particular cloud system. So, if we need to use Live Migration to another cloud, it might be difficult to implement.
4. Live migration of a process in Docker is still an experimental feature. Only few versions of Linux Operating System and Docker support the feature. This is also a bottleneck of Live Migration.
5. If we consider the security perspective, docker is mainly dependent of the host Operating system [14], therefore, if there any kind of attack or security problem in the host Operating system, then it will have an impact on the docker system as well.

## 2.9 Fog and Edge Computing

Fog computing is a virtualized computing platform that enables to compute, contain and communicate between end devices and cloud servers. One of the main reasons of introducing Fog Computing was to eliminate the high latency of Cloud Computing [15]. This fog computing platform is also virtualized like cloud computing platform. This is very efficient to use in the case of small scale applications using IoT. A fog computing architecture can be divided into three parts – Cloud network, Fog computing and end devices. It is often called as “Edge network”, however, there is a small difference between the two that is, in the Fog computing the end devices do not have the processing capability while in the case of edge computing, the end nodes have the processing capability (Figure 2.10).

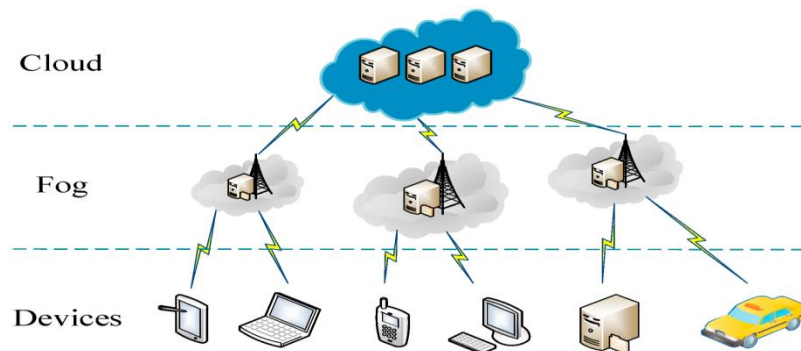


Figure 2.10: Fog Computing [16]

Some of the main advantages of Fog Computing are-

1. Lower latency
2. Wireless access
3. Support many nodes
4. Geographically distributed
5. Multiple sensor network
6. Have the ability to move
7. Diversity of the nodes
8. Regular interoperability
9. Very useful for real-time programs
10. High efficiency

## 2.10 Related Work

Chenying et al. proposed a solution for live migration. They have used Logging and Replay method. The aim was to decrease the delay of live migration.

An approach called “Pre-copy” is one of the most popular methods used for this experiment. The idea is that to make a copy the memory state first. Then it will send the state to the destination and resume it again in that destination machine. If needed, it is also possible to keep doing the process again and again, so that it can be migrated several times. If there is a process that needs to be migrated several times, then the iterative process can be used. One of the bottlenecks of this method is that, it can affect the performance that requires continuous iteration and having heavy workload. Some of the real time processes may be very sensitive to the memory and longer delay. This “Pre-copy” approach can be a bit challenging for those applications. The “Pre-copy” approach appeared to be very efficient for almost all the applications, therefore, it gained more popularity. Moreover, it was also challenging to find out another solution that can replace this method. Therefore, researchers focused on how they can use this method but with an improved performance. So, one solution was to use smaller amount of data than before. It will make the transfer faster with lesser delay time. Therefore, they compressed the files before transferring so that the size of the files is reduced [11].

Shripad Nadgowda et al. worked with a live migration system called as “just in time migration” [14]. They tried to make this migration process more efficient by minimizing downtime. A container system is a combination of file system that we can extract in the second machine or directory as needed. When we execute a container, all the dependencies are also instantiated. Here, container migration includes everything which can be divided into three parts:

1. Migration of the memory state
2. Migration of the local file system
3. Migration of the network file system

To transfer the file system, a lazy replicator has been used in the background. It keeps track of all the data inputs and deletes the replicate copies. While transferring the state, the lazy replications do not use any application downtime or has limited downtime. There experiment shows the checkpoint time is less than 2 seconds for 1 gigabyte of data. This model ensures that the process has been continued again just in time [14].

Samu Toimela et al. researched on the containerization techniques on cloud applications [17]. Virtualized networking systems have become very popular for the applications based on fog and cloud computing system. This research shows how Linux containers can make the applications faster to be executed. It is recommended to use small sized applications in containers. The main focus of this research is to reduce the overhead with increased performance in the case of fog computing systems. In the complex systems, the number of data traffic is quite high with complicated communication pattern. In this case, if we use container based application such as docker, it will reduce the overhead and complexity with increased performance. The experimental result shows, that docker container can start up a process in a very short time. Moreover, the live migration can make the whole system more organized and energy-efficient.

Yahya Al-Dhuraibi et al. have worked with a flexible migration system called “ELASTICDOCKER”. The concept of elasticity has been taken from the features of cloud computing. The main idea of “ELASTICDOCKER” is that when a machine will be very busy of executing applications and this it is having limited resources, then it will transfer the application in another machine to balance the workload. This transfer is done by live

migration. It can distribute the workload evenly. CRIU is used to execute the migration to keep the workload in a minimum level. The experiment has been done in 7vCPUs with Centos 7.2 OS and Graylog application to check the resources used. The experimental result shows that the “ELASTICDOCKER” is more stable than the docker with increased performance of almost 80%. As the summary, it was said that “ELASTICDOCKER” or live migration using docker can provide better and automated resource utilization [18].

Corentin Dupon et al. have presented a platform that for edge computing using container migration in Linux [19]. They have proposed 2 types of migration – vertical and horizontal. The horizontal migration is the similar concept of live migration using Docker which has been tested here in the context on health care data processing. In the experiment part, OpenStack and Kubernetes have been used as the Virtual Machine and container manager consequently. The communication has been done by Bluetooth Low Energy (BLE). The primary processing of the data has been done by a docker image to reduce latency. The aim of this research was to show that migration of the data processing system can be very useful for the Edge Computing.

## 2.11 Summary

Docker is the most popular container based Linux system that has a very large number of public images. It considers every container as an application. For the larger application, it consists of multiple containers. Docker system is based on images, client and server, containers and registries. Docker container system is often compared with virtual machines. However, the major difference is that virtual machines (VMs) are the abstract version of an operating system (OS) whereas containers are the abstract version of executable applications. Virtual machine uses hypervisor where docker uses host operating system. Docker make the application very fast, compact and easily maintainable. Live migration is a tremendous feature of docker which can be used very effectively in the end nodes of fog computing system. Live migration has very small delay times which do not have any impact on the performance usually, however, in some real time application we have to carefully consider the migration time as well.

## Chapter 03: Methodology and System Setup

To experiment live migration of a process, I will use docker as the container manager. There are several container managers but docker is the most accepted in the recent years and it is very easy to use. In today's IT industry, more than 3.5 million applications are run by docker. This container manager is very efficient to manage applications physically distributed. Later, Kubernetes is being used to efficiently organize docker container.

In this part, we will discuss about how we are going to make the system environment ready so that we can experiment live migration. This is an experimental feature of docker that makes the process difficult to execute. There are some conditions we need to follow to be able to set up the system environment accurately as required.

First of all, Docker experimental feature is not enabled by default, which means we have make it enabled. Secondly, the experimental feature is not supported by any version of Docker after version 17.09. We will use docker version 17.09.1~ce. Finally, as the host operating system we will use 64 bit UBUNTU 16.04.4 LTS. All the steps of system setup are as follows-

1. Set up 64 bit UBUNTU 16.04.4 LTS in both servers or machines
2. Install Docker, it will install the latest version by default
3. Downgrade to the docker version 17.09.1~ce
4. Enable docker experimental feature
5. Install CRIU in the docker

### 3.1 System setup for Live Migration:

One of the important parts to experiment Live Migration is to ensure the environment is set up correctly. We will primarily experiment this live migration between two servers. The concept is that we will print some integer numbers in the incremental way in the source, then we will use checkpoint to stop the process and rest of the process will be executed in the destination server.

The workflows of whole processes are as follow:

1. Source and Destination environment setup
2. Start and run a process in the source
3. Pause the process using checkpoint
4. Make a tarball to compress the files
5. Send it compressed files to the destination
6. Untar or uncompress the files in the destination
7. Resume the process again in the destination

## 3.2 Preparing the servers

For experimenting live migration, we need two servers. We will use 64x UBUNTU 16.04.4 LTS in both the servers. LTS version of Ubuntu is used to experiment in the server. Live migration is an experimental feature of docker; therefore, all the updated versions of Operating Systems do not support to experiment this. That is why, instead of using the updated versions we have used some certain versions to make it work accurately.

To migrate process to another destination, we have to use “checkpoint”, which can be done using CRIU. At present, Docker for Windows or Mac, do not support “CRIU” binary, we must for Linux, more particularly Ubuntu. It will work as the host of Docker.

At first we have set up 64 bit UBUNTU 16.04.4 LTS version in either the servers or machines. In the large scale industrial application in docker, servers are being used. We can also do the same if we need to experiment the migration in the computers or in the end nodes for fog computing. This UBUNTU 16.04.4 LTS will work as the host for docker applications. The version is called “Xenial Xerus”. Migration in Docker cannot be done in Windows operating system or macOS, so we will experiment it in Linux only.

After we set up UBUNTU 16.04.4 LTS in both end (source and destination), we have to set up docker using terminal. All the things we are doing here, needs to be done in both the servers. We use “sudo” in the command if we want to run in the admin mode and it allows

us to access all the needed files and directories. If we are already in admin mood or using as a “root” user, then we do not need to use “sudo” in the command line. If we need to execute commands in the admin mode, we can use following command line 1 just once so that we do not need to use “sudo” all the times. It allows executing commands as the admin mode.

1. # sudo usermod -aG docker \$USER

We will set up docker by using following official script with the experimental feature enabled (Figure 3.1). We just need to run this in the command terminal where docker is installed. We can execute either one by one line or all at the same time, both ways will work.

```
export CHANNEL=stable
curl -fsSL https://get.docker.com/ | sh
## Add Docker daemon configuration
cat <<EOF | sudo tee /etc/docker/daemon.json
{
    "icc": false,
    "disable-legacy-registry": true,
    "userland-proxy": false,
    "live-restore": true,
    "experimental": true,
    "debug": true
}
EOF
## Start docker service
sudo systemctl enable docker
sudo systemctl restart docker
```

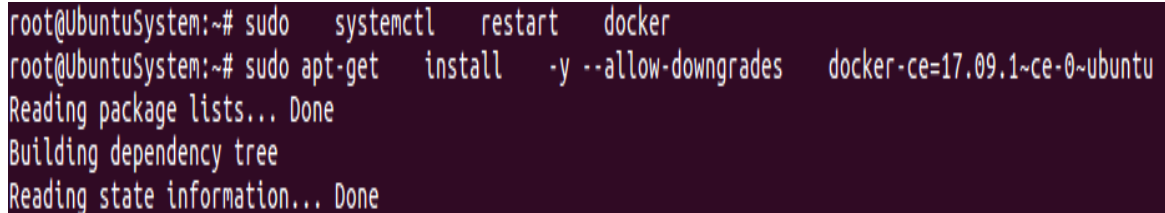
Figure 3.1: Enabling experimental feature

This will install the latest version of Docker which is currently 18.03.0-ce. However, live migration is still an experimental feature, therefore, a certain version of docker support this feature. The experimental feature is not supported by any version of Docker after 17.09. So, we have to install Docker 17.09 by downgrading the updated version.



We will use the following command number 2 to downgrade the version of docker (Figure 3.2).

2. `# sudo apt-get install -y --allow-downgrades docker-ce=17.09.1~ce-0~ubuntu`

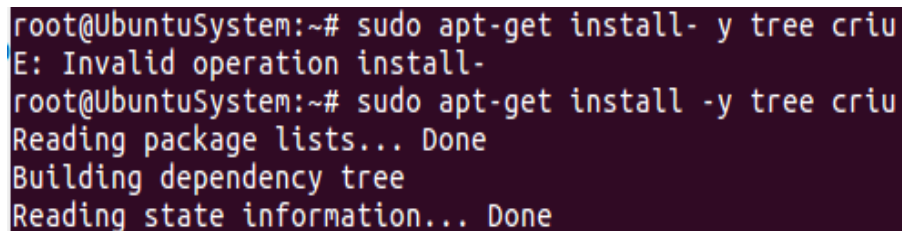


```
root@UbuntuSystem:~# sudo systemctl restart docker
root@UbuntuSystem:~# sudo apt-get install -y --allow-downgrades docker-ce=17.09.1~ce-0~ubuntu
Reading package lists... Done
Building dependency tree
Reading state information... Done
```

Figure 3.2: Downgrading Docker version

Then we will install CRIU so that we can use checkpoint for this experiment. CRIU is the feature that enables us to stop and resume a process again. We can resume the process in the same machine or in the different machine which we call live migration. The feature of CRIU can be used in Docker only in experimental mode. In this phase of the experiment we will install CRIU in docker using command 3 (Figure 3.3).

3. `# sudo apt-get install -y tree criu`



```
root@UbuntuSystem:~# sudo apt-get install -y tree criu
E: Invalid operation install-
root@UbuntuSystem:~# sudo apt-get install -y tree criu
Reading package lists... Done
Building dependency tree
Reading state information... Done
```

Figure 3.3: Installing CRIU

After both of our servers are ready, we have to check the version. This is very important to check that we have installed the correct version of Docker to be able to do live migration. While experimenting, I had problem for experimenting because I tried with other versions or with the latest version. Therefore, as a part of system setup we will check the docker

version. As we mentioned earlier, we will use docker version 17.09.1-ce. We will use the following command number 4 in the terminal. It will show the version of docker, Operating System architecture as well as the experimental feature is enabled (Figure 3.4).

#### 4. # docker version

```
root@UbuntuServer2:~# docker version
Client:
 Version:      17.09.1-ce
 API version:  1.32
 Go version:   go1.8.3
 Git commit:   19e2cf6
 Built:        Thu Dec  7 22:24:23 2017
 OS/Arch:      linux/amd64

Server:
 Version:      17.09.1-ce
 API version:  1.32 (minimum version 1.12)
 Go version:   go1.8.3
 Git commit:   19e2cf6
 Built:        Thu Dec  7 22:23:00 2017
 OS/Arch:      linux/amd64
 Experimental: true
```

Figure 3.4: Docker version

From the output we can make sure that, we have the correct version of docker 17.09.1-ce. It also shows the system configuration in details and also the experimental feature is enabled. It means both of our machines or servers are ready now for experimenting live migration.

### 3.3 System setup for calculating migration time

The concept of live migration is to transfer a running process from one machine to another machine. We stop the running process into a state using checkpoint and then we transfer the state to another machine. Then we resume it again to the destination machine. Practically, the migration is done so fast that the user cannot understand the process has been migration to another server. In most of the cases, it's seems that everything is running smoothly although there is a live migration behind the scene. Apparently it seems that everything is live without any interruption so it is also called as live migration (Figure 3.5).

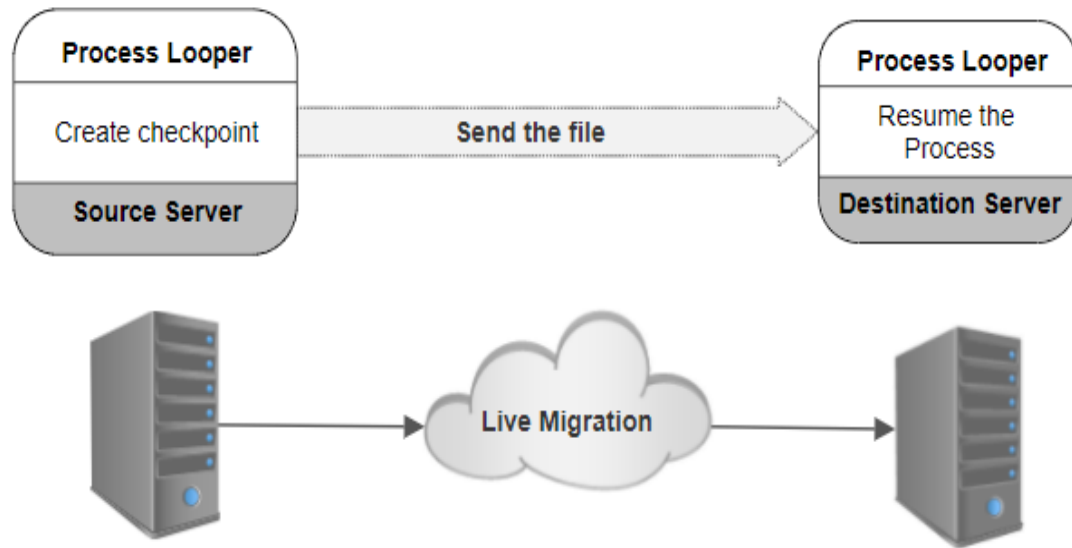


Figure 3.5: Live migration

The figure shows an overview of the live migration. We can consider, we have a busybox process called “Looper”. It executes a number in every one second such as 1, 2, 3, 4, 5 and so on. For the live migration, we create checkpoint after a period of time. In Linux, the file is called as tarball which is similar to the zip file for Windows operating System. When we send the file from source to destination, it definitely needs some time interval. The time needed to transfer the file to the destination is called as delay time or migration time. Therefore, our target is to find the time it exactly took to transfer the file.

However, the fact is that even the migration time is too small to ignore, still there must be a small delay time to transfer the process. Our aim is to calculate the delay time. For this experiment, we will use two 64 bit UBUNTU 16.04.4 LTS servers, one of them is source and another is destination server. The CPU version is Intel(R) Xeon(R) CPU E5-2630L v2 @ 2.40GHz.

We can check the detailed configuration by following command no 5 (Figure 3.6).

5. # lscpu

```

root@UbuntuServer2:~# lscpu
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:             Little Endian
CPU(s):                 1
On-line CPU(s) list:   0
Thread(s) per core:    1
Core(s) per socket:    1
Socket(s):              1
NUMA node(s):          1
Vendor ID:              GenuineIntel
CPU family:             6
Model:                  62
Model name:             Intel(R) Xeon(R) CPU E5-2630L v2 @ 2.40GHz
Stepping:               4
CPU MHz:                2399.998
BogoMIPS:               4799.99
Virtualization:         VT-x
Hypervisor vendor:      KVM
Virtualization type:    full
L1d cache:              32K
L1i cache:              32K
L2 cache:               256K
L3 cache:               15360K
NUMA node0 CPU(s):     0

```

Figure 3.6: System configuration

When we will do the experiment in a fog computing system, where a number of end nodes or small embedded system will be available, we may need to check the hardware information as well. At first, we need to install and then we can check it by the following commands (Figure 3.8).

This will show very detailed information about the configuration; here I have just presented a part of it. Figure 3.7 shows the command to install “hardinfo” using command no 6. It generates a dependency tree.

6. # apt install hardinfo

```
root@UbuntuServer2:~# apt install hardinfo
Reading package lists... Done
Building dependency tree
```

Figure 3.7: Installing hardinfo

After we have installed “hardinfo”, we can check the detailed hardware information. Figure 3.8 shows the summary of the hardware information. It shows the processor, operating system and memory using command no 7.

7. # hardinfo

```
Summary
-----

-Computer-
Processor           : Intel(R) Xeon(R) CPU E5-2630L v2 @ 2.40GHz
Memory              : 1016MB (380MB used)
Operating System     : Ubuntu 16.04.4 LTS
User Name            : root (root)
Date/Time            : Thu May  3 09:20:02 2018
-Display-
Resolution           : 0x0 pixels
OpenGL Renderer      : Unknown
X11 Vendor           : (null)
-Multimedia-
-Input Devices-
Power Button
AT Translated Set 2 keyboard
VirtualPS/2 VMware VMMouse
VirtualPS/2 VMware VMMouse
```

Figure 3.8: Checking hardware information

We can also get more information about processors from the following command no 8 (Figure 3.9) [18]. For my thesis, these information in not necessary, however, if there is a fog computing system with multiple nodes and having real time application, it will be necessary to know the hardware configuration to make the system more efficient.

## 8. #cpuid

```

CPU 0:
  vendor_id = "GenuineIntel"
  version information (1/eax):
    processor type = primary processor (0)
    family         = Intel Pentium Pro/II/III/Celeron/Core/Core 2/Atom, AMD A
    thlon/Duron, Cyrix M2, VIA C3 (6)
    model          = 0xe (14)
    stepping id    = 0x4 (4)
    extended family = 0x0 (0)
    extended model = 0x3 (3)
    (simple synth) = Intel Core i7-4000 / i9-4000 / Xeon E5-1600/E5-2600 v2 (
    Ivy Bridge-EP C1/M1/S1), 22nm
  miscellaneous (1/ebx):
    process local APIC physical ID = 0x0 (0)
    cpu count                       = 0x0 (0)
    CLFLUSH line size              = 0x8 (8)
    brand index                     = 0x0 (0)
  brand id = 0x00 (0): unknown
  feature information (1/edx):
    x87 FPU on chip                = true
    virtual-8086 mode enhancement = true

```

Figure 3.9: Checking CPU information

### 3.4 Summary

This chapter shows the detailed procedure of setting up the system environment for our experiment. Live migration is still an experimental feature, therefore, we have to follow some system requirements more specifically the correct docker and Ubuntu Linux version. As the operating system, we will use 64 bit UBUNTU 16.04.4 LTS in both the servers and machines. Also, we will use docker version 17.09.1~ce. In the docker, we will run the commands in the admin mode. In addition to Linux and docker, we have to install CRIU which enables us to do the live migration. Most importantly, experimental feature is not enabled by default, so we have to make it enabled first. The live migration will need some time for as the transfer time, we will also set up the system so that we can calculate the migration time which is also known as transfer or delay time.

## Chapter 04: Implementation

In this part of the thesis, we will experiment the live migration. In the implementation section, we have two targets. First one is to experiment how we can do the live migration and the second one is to calculate the migration time. In the previous chapter, we have already discussed about how we made our system ready for this experiment. As the host operating system, Ubuntu and the correct version are ready. We have also enabled experimental feature. In the implementation part, my target is to create a container and then run a process in one server and then transfer the process in another server. This will be done by using CRIU which we have set up already in the docker. This CRIU will enable us to stop the process in the first server, send it to the second server and finally, we will resume the process into the second server. Our total workflow is as follows (Figure 4.1).

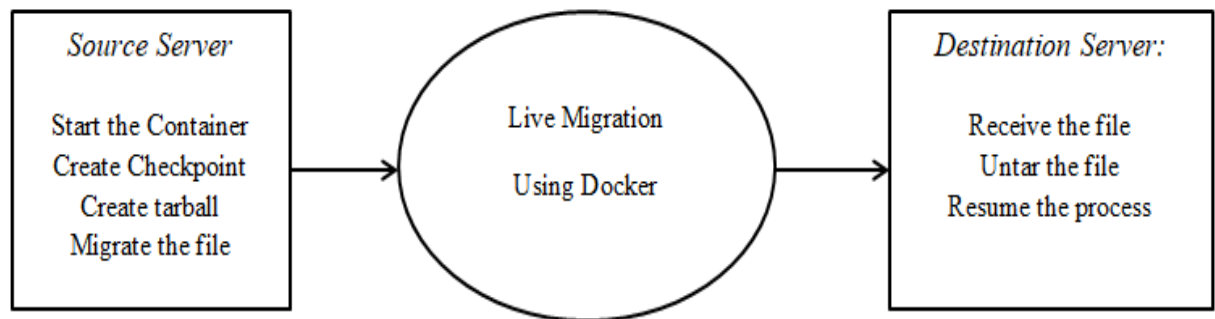


Figure 4.1: Live migration concept

In the second part, we will try to measure the migration time for this experiment. As Live Migration needs to stop the process for some time, therefore, it will have some delay for the migration. In the most of the cases, the frozen or file transfer time is very small. However, some critical real time applications are very sensitive to delay, therefore, it is important to find out the delay time. We will also compare between different communications systems to find out which system takes lesser time for a same migration process.

## 4.1 Experiment of live Migration

Now, we will explain the whole process in details (Figure 4.2). It will be easier for us to understand if we can divide the whole task into three parts, which are –

1. Preparing the servers
2. Operations in Source Server
3. Operations in Destination Server

The whole process of live migration can be divided into three parts. They are –

1. Creating the checkpoint in the source node
2. Create tarball and send it to the destination
3. Resume the checkpoint in the destination server

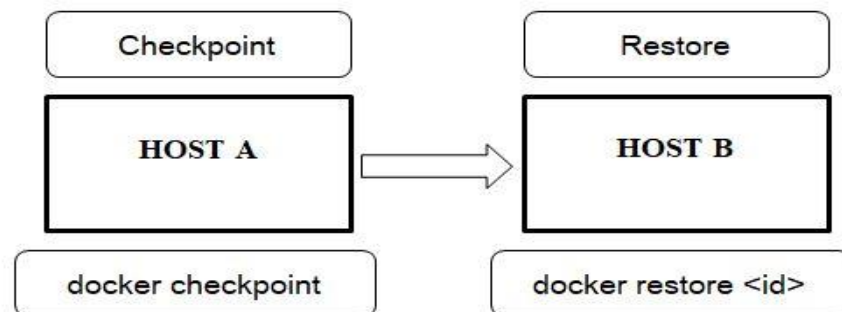


Figure 4.2: Checkpoint and Restore

## 4.2 Operations in Source Node

In the source server, we will create a simple and executable container named “busybox”. This container consumes just from 1 Megabyte to 5 Megabyte disk space. It is very useful where the system memory space is limited, for example edge nodes of fog computing system or small resource constraint embedded systems. We can also use busybox to generate 200 types of small executable operation. Therefore, it is very useful to create an application environment in the small embedded systems or for the end nodes in the fog computing system. Using busybox, we will also create a small and continuously running



application [20]. We will create a container that will print a number in every one second in an increasing order. Here, we have created a container named “looper100” that will keep printing integer numbers in every one second in the increasing order. It is a counter that counts in every second in the increasing order. We have to use following command 1.

1. # docker run -d --name looper100 busybox /bin/sh -c 'i=0; while true; do echo \$i; i=\$(expr \$i + 1); sleep 1; done'

After we run the command, it will create a new container and it will generate a container ID as the following figure 4.3.

```
root@UbuntuServer2:~# docker run -d --name looper100 busybox /bin/sh -c 'i=0; while true; do echo $i; i=$(expr $i + 1); sleep 1; done'
5b6ab68c2d11b2082f9b7c49aff793cfd9e3cea4846a83ec39b45023e2abf302
```

Figure 4.3: Running “busybox” image

As we can see that the container ID has been created, now we need to make sure that the container is running. So, using the command 2, we can see that the container has been created from busybox image and the name of the container is “looper100”. In the command 3, that prints the output of the container. So, we print the value of “looper100” and we can see that numbers are keep printing in every one second (Figure 4.4).

2. # docker ps
3. # docker logs looper100

```
CONTAINER ID    STATUS    IMAGE        PORTS    COMMAND    NAMES    CREATED
975c5d0f98b8    Up 4 seconds    busybox      "/bin/sh -c 'i=0; ..."    looper100    5 second
s ago
root@UbuntuServer2:~# docker logs looper100
0
1
2
3
4
5
6
7
```

Figure 4.4: Output of the process

The number is already started printing which means that we have a running process. Now, we will stop the running process at some point as we wish. We will create checkpoint for the container we just created. We named the checkpoint as “checkpoint100”. This is the exact application of CRIU that we can stop or checkpoint a running process.

Before creating the checkpoint, we will also create a separate directory so that it becomes easier for us to find it out later and we can compress the files from that directory. We will create a “tmp100” directory where we keep the checkpoint file. Also, the checkpoint we created, we named it as “checkpoint100” (Figure 4.5). The commands 4 and 5 are as following:

4. # mkdir -p /tmp100/looper\_checkpoint
5. # docker checkpoint create --checkpoint-dir=/tmp100/looper\_checkpoint looper100 checkpoint100

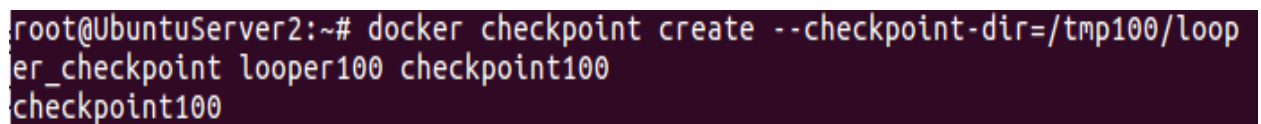
A terminal window with a dark background and light-colored text. The prompt is 'root@UbuntuServer2:~#'. The command entered is 'docker checkpoint create --checkpoint-dir=/tmp100/looper\_checkpoint looper100 checkpoint100'. The output of the command is 'checkpoint100'.

Figure 4.5: Checkpoint created

We can see that the checkpoint has already been created and the name is “checkpoint100” as we created. At this point, we have to check by printing the logs of the container “looper100”. Our target is to make sure that the checkpoint has been created, means the process has been paused. To check this, we will simply print the logs as we done before.

After printing the logs using command 6, we can see that the checkpoint has been created at number 10 and even if we execute it after sometime, the number is still 10. That means, instead of increasing the numbers, it just paused. It means we have successfully created the checkpoint and it will not continue unless we resume it again (Figure 4.6).

6. # docker logs loop100

```
root@UbuntuServer2:~# docker logs loop100
0
1
2
3
4
5
6
7
8
9
10
root@UbuntuServer2:~# docker logs loop100
0
1
2
3
4
5
6
7
8
9
10
```

Figure 4.6: Output after the checkpoint

In the next step, we have to copy the checkpoint directory. Therefore, we have to check the full directory path first where this “checkpoint100” has been created. We can check the detailed directory path now by using “tree”.

It helps to find out the full directory path so that we can copy using the right directory path. From the following figure 4.7, we can see that the “checkpoint100” has been created under checkpoints directory using command 7.

7. # tree -L 4 /tmp100

```
root@UbuntuServer2:~# tree -L 4 /tmp100
/tmp100
├── looper_checkpoint
│   └── 5b6ab68c2d11b2082f9b7c49aff793cfd9e3cea4846a83ec39b45023e2abf302
│       ├── checkpoints
│       └── checkpoint100
```

Figure 4.7: Directory path

After this, we will copy the full directory where the checkpoint has been created. Then will compress the files of that directory. In Linux, it is called as creating the tarball. It is similar to the zip file in windows operating system. Here, we named it as “looper\_cp100.tgz”.

We can check if the file has been created correctly by executing “ls” by command number 8. We can see the “looper\_cp100.tgz” has been created (Figure 4.8).

8. # ls

```
root@UbuntuServer2:~# ls
looper_cp.tgz      looper_cp15.tgz  mm-1.img
looper_cp100.tgz  looper_cp71.tgz  start.sh
```

Figure 4.8: Checking tarball

Finally, we copy the tar file and send it to the destination server. In this case, we have directly transferred the target server. Another way of doing this may be, downloading the file first into the local PC and then upload into the target.

It is also possible to transfer it between servers, machines and nodes. Executing the following commands 9, 10 and 11 we can check that the tarball has been transferred (Figure 4.9).

9. # cp\_dir=`find /tmp100/looper\_checkpoint/<full\_path> -name checkpoints`

10. # sudo tar -czvf looper\_cp100.tgz -C \$cp\_dir .

11. # scp looper\_cp100.tgz root@178.62.73.93:./

```
root@UbuntuServer2:~# scp looper_cp100.tgz root@178.62.73.93:./
root@178.62.73.93's password:
looper_cp100.tgz                                100%   32KB   32.4KB/s   00:00
```

Figure 4.9: Transferring the process

### 4.3 Operations in Destination Server

We have already transferred the file into the destination server. The rest of the command number 12, 13, 14 will be executed in the destination server. Our target is to resume the operation here. Here, we can check the docker version which shows 17.09.1-ce has been installed here. Then, we create a directory where we will keep the tar file. After that, we Untar or uncompress the “looper\_cp100.tgz” into that directory (Figure 4.10).

12. # docker -v

13. # mkdir -p /tmp100/looper\_checkpoint

14. # sudo tar -zxvf looper\_cp100.tgz -C /tmp100/looper\_checkpoint

```
root@UbuntuServer3:~# docker -v
Docker version 17.09.1-ce, build 19e2cf6
root@UbuntuServer3:~# mkdir -p /tmp100/looper_checkpoint
root@UbuntuServer3:~# sudo tar -zxvf looper_cp100.tgz -C /tmp100/looper_checkpoint
./
./checkpoint100/
./checkpoint100/fs-26.img
./checkpoint100/mountpoints-12.img
./checkpoint100/seccomp.img
./checkpoint100/criu.work/
./checkpoint100/criu.work/stats-dump
```

Figure 4.10: Untar the received process

The previous image shows that we have recovered or uncompressed all the files from “looper\_cp100.tgz”. We can check the directory by using tree which also shows the “checkpoint100” has been copied here by command number 15 ((Figure 4.11).

15. # tree -L 2/tmp100

```
root@UbuntuServer3:~# tree -L 2 /tmp100
/tmp100
├── looper_checkpoint
│   └── checkpoint100
└── 2 directories, 0 files
```

Figure 4.11: Checkpoint checking

Now we will generate the same container first so that we can start from the checkpoint created by command number 16. We have created the same container but with the different name “looper100-clone”. It also generates a counter that counts integer number in every second (Figure 4.12).

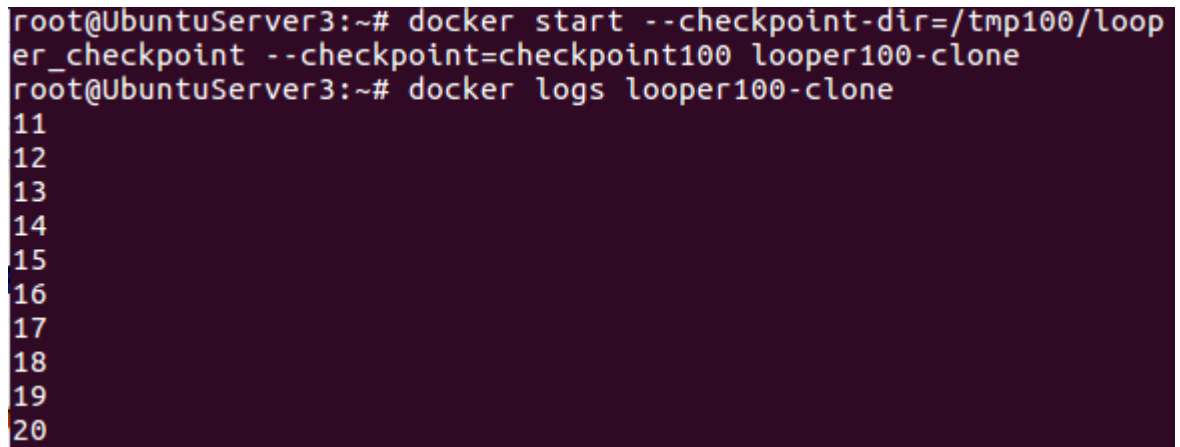
16. # docker create --name looper100-clone busybox /bin/sh -c 'i=0; while true; do echo \$i; i=\$(expr \$i + 1); sleep 1; done'

```
root@UbuntuServer3:~# docker create --name looper100-clone busybox /bin/sh
-c 'i=0; while true; do echo $i; i=$(expr $i + 1); sleep 1; done'
354a64b43f72656c713e63ea9899a6d47c26b1f037acb91c4cdad3282265eac8
```

Figure 4.12: New container created

Now we will start the container from the checkpoint. As it will resume the process, it should start printing from 11. After printing the logs, we can check that the numbers has been start counting from 11 means the process has been resumed from where we paused. We will use command 17 and 18 what shows the output in Figure 4.13.

17. # docker start --checkpoint-dir=/tmp100/looper\_checkpoint --  
checkpoint=checkpoint100 looper100-clone
18. # docker logs looper100-clone



```
root@UbuntuServer3:~# docker start --checkpoint-dir=/tmp100/loop
er_checkpoint --checkpoint=checkpoint100 looper100-clone
root@UbuntuServer3:~# docker logs looper100-clone
11
12
13
14
15
16
17
18
19
20
```

Figure 4.13: Resuming container

The whole experiment shows, we started and paused a process in a machine, then resumed it in the second machine. That means we have successfully migrated the process which is called Live Migration in the Docker technology.

## 4.4 Calculating transfer time

After we have successfully experimented live migration, we will start our second task of this thesis. Our mission is to find out the time it took to migrate the process. In the case of “looper\_cp100.tgz” we can see that the file size is 33KB which took 0.4 second to migrate from host server to the destination server.

The command number 19 will execute a detailed report; however, we have just showed the important part in the following figure 4.14.

19. # scp -Cv looper\_cp100.tgz <destination>:./

```

Sending file modes: C0644 33926 loopier_cp100.tgz
loopier_cp100.tgz      0%   0   0.0KB/s  --:-- ETASink: C0644 33926 loopier_cp100.tgz
loopier_cp100.tgz      100% 33KB 33.1KB/s  00:00
debug1: client_input_channel_req: channel 0 rtype exit-status reply 0
debug1: channel 0: free: client-session, nchannels 1
debug1: fd 0 clearing O_NONBLOCK
debug1: fd 1 clearing O_NONBLOCK
Transferred: sent 35780, received 2572 bytes, in 0.4 seconds
Bytes per second: sent 96481.5, received 6935.5
debug1: Exit status 0
debug1: compress outgoing: raw data 34543, compressed 33929, factor 0.98
debug1: compress incoming: raw data 33929, compressed 34543, factor 1.02

```

Figure 4.14: Migration time

We also need to check the tarball file details to experiment the relation between the checkpoint creation time and tarball size. We can check the tarball size by using “stat” before the file name by following command 20 (Figure 4.15).

20. # stat loopier\_cp100.tgz

```

anik@anikVB:~$ stat loopier_cp100.tgz
  File: 'loopier_cp100.tgz'
  Size: 33926          Blocks: 72          IO Block: 4096   regular file
Device: 801h/2049d    Inode: 263504       Links: 1
Access: (0644/-rw-r--r--)  Uid: (  0/   root)   Gid: (  0/   root)
Access: 2018-05-04 01:49:29.186604650 +0300
Modify: 2018-05-04 01:49:03.102320105 +0300
Change: 2018-05-04 01:49:03.102320105 +0300

```

Figure 4.15: Checking tarball file details

## 4.5 Comparison of the delay time

In this part of the experiment, our mission is to measure the migration time and observe the result. In the table 4.1, I have showed the name of the created tarball, host and destination environment, tarball size in kilobytes, sent and received data and most importantly the required time for the live migration.



Tarball Name	Source	Destination	File Size (Kilo Bytes)	Data Sent from Source (Bytes)	Acknowledgement Received at Source (Bytes)	Time (second)
looper_cp.tgz	Server 1	Server 2	21	23508	2608	0.2
looper_cp.tgz	Server 2	Server 1	21	23508	2608	0.2
looper_cp.tgz	Machine	Server 2	21	23560	2572	0.5
looper_cp.tgz	Machine	Server 1	21	23560	2572	0.5
looper_cp.tgz	Server 2	Machine	21	23440	2572	0.1
looper_cp.tgz	Server 1	Machine	21	23440	2572	0.1
looper_cp100.tgz	Server 1	Server 2	32	34924	2572	0.2
looper_cp100.tgz	Server 2	Server 1	32	34924	2572	0.2
looper_cp100.tgz	Machine	Server 2	33	35780	2572	0.5
looper_cp100.tgz	Machine	Server 1	33	35780	2572	0.5
looper_cp100.tgz	Server 2	Machine	33	35780	2572	0.1
looper_cp100.tgz	Server 1	Machine	33	35780	2572	0.1

Table 4.1: Comparison of migration time

By analyzing the data of the previous table 4.1, we can make some decisions about the behavior of the migration time. What I have found for this experiment is that –

1. For a same live migration process, the transfer time or delay can vary based on the host to destination connection. For example, for the simple migration application of busybox image, the migration time varied from 0.1 second to 0.5 second.

This is because we have tested the same migration for different source and destination environment (server to server, machine to server and server to machine). The system configuration was different for different environment in case of available memory, CPU capability, processing power and communication medium. Therefore, the system configuration has a direct impact on the time required for the live migration.

2. Migrating from server to machine was the fastest for this small process which is 0.1 second while the reverse was slowest with the value of 0.5 second. And, the migration time between servers was 0.2 second. This different migration time for a same process shows how the system configuration can affect the transfer. There are 2 main reasons for this.

Firstly, in our experiment, the configurations of servers were much better than the local machine. Therefore, when the migration has been done from server, it was faster than the migration done from local machine.

Secondly, servers are more secured usually; therefore, some protocols are required to transfer a process into such as executing command as administrator and login with correct password. Therefore, when we have migrated from server to machine, it took just 0.1 second because no security checkup was required in the machine. However, when we migrated to the server, it has to validate the login credentials first and only then it allowed migrating to the server with a longer transfer time of 0.5 second. Server to server migration also needs to validate, therefore it took 0.2 second while server to machine was 0.1 second as no validation required.

3. For a tarball of same size, the amount of sent data and received acknowledgement bytes in the source node are almost same regardless the host and destination environment. When we migrate a process from the source by secure copy, it just

shows how much data has been sent and in return, it receives some acknowledgement (also known as communication overhead) that the process has been transferred. For this reason, the number of sent data (23508 bytes) is larger than the number of acknowledgement received (2608 bytes).

4. The tarball size then sent data and received acknowledgement data and the migration time is exactly same for the migration in any direction whatever is machine or server.
5. The tarball size increases with the time we have taken more to create checkpoint. In this case, we have created checkpoint in several times and observed the tarball file size. When we have created checkpoint at the 10<sup>th</sup> second, the file size was 21KB. At the 21<sup>th</sup> second, the file size was 33KB. Finally, at 100<sup>th</sup> second the tarball size was 101 bytes. It shows that the tarball size increased linearly with time. This is because, in every second, the busybox image used to generate a number in the increasing order. At the 10<sup>th</sup> second there was 10 numbers and at the 100<sup>th</sup> second there were 100 numbers. Therefore, more numbers have increased the tarball size consistently (Figure 4.16).

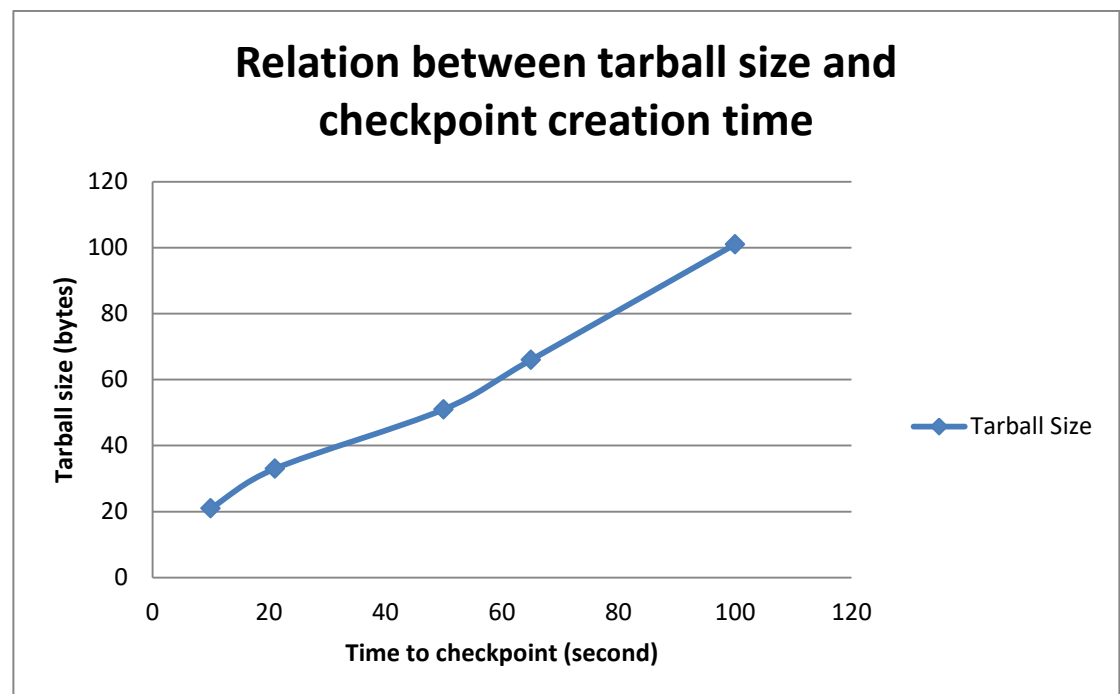


Figure 4.16: Increased file size with time

## 4.6 Future Work

This thesis is based on two major experiments. Firstly, it experiments the process of live migration and secondly, it observes the migration time. However, it can be expanded to some new experiments.

The first one is to find the best communication technology. There are several technologies we can use to experiment live migration between the machines. We can transfer the file by Bluetooth, Wi-Fi, LAN, WLAN, Zigbee or by TCP/IP. Every technology has its own file transfer speed and delay time. So, we can experiment the delay or migration time with each technology individually. This figure of 4.17 will give a clear idea about which communication protocol will be the best to use to implement in the fog computing system.

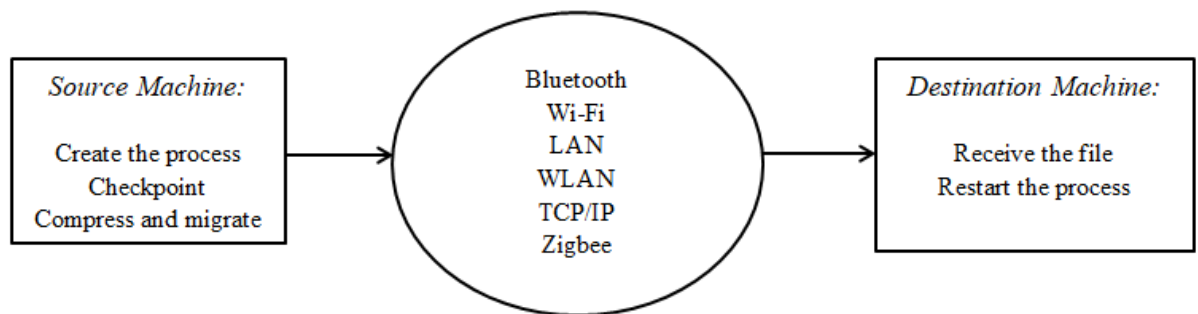


Figure 4.17: Migration using different communication technology

The second idea is to design a smart and automated migration system. The live migration is done between two machines, we migrate the process from host machine to destination machine. One of the important reasons of this live migration is to ensure proper load-balancing. In a Fog computing system, there might be multiple machines or nodes working on different process at the same time. So, it might happen that one node become very busy in terms of utilization on the other hand the other node is almost free. Too much workload on a single machine will degrade the performance and will increase the processing time.

To overcome that, the idea is to transfer the running process from first node to the second node, so that that the load can be balanced equally. We can just discuss about a sample scenario of figure 4.18.

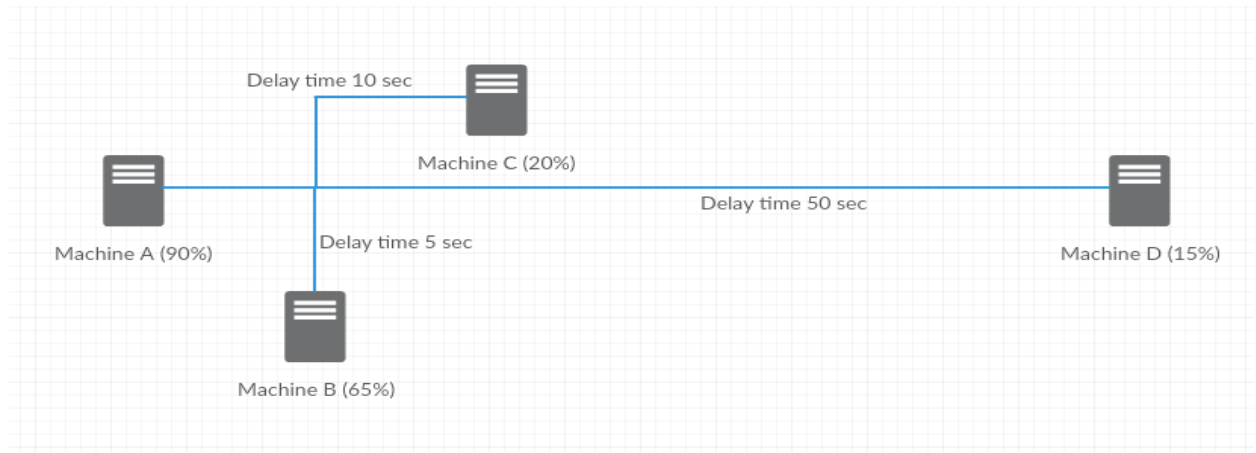


Figure 4.18: Migration in a sample edge network systems

In this scenario, we have considered system, where we have 4 edge network, we named it Machine A, Machine B, Machine C and Machine D where the CPU usage is 90%, 65%, 20% and 15%. Here, Machine A has being highly utilized, so it can transfer the process to the other machine that has less usage by live migration. Machine A can migrate the process either in Machine C or in Machine D.

Every machine is located in different location and therefore, it will have some different delay time for the live migration. Here, although Machine D has least CPU utilization but it will have a higher delay time to migrate. Therefore, Machine C will be the optimal solution to migrate as it has lesser CPU utilization as well as lesser delay time.

In this scenario, we have considered two most important factors for live migration – CPU utilization and delay time. For a better solution we can also consider distance between all the nodes, current workload and pending processes of every node, memory usages, device configuration, processing capability and temperature. All this things will help to build a very smart automated system for live migration in the edge network.

## 4.7 Summary

In this chapter, we have experimented live migration, then calculated the migration time and made a comparison for different scenarios. The whole process of live migration can be divided into three major parts – creating checkpoint in the source, send the tarball to the destination and resume the process in the destination. As a process, we have used busybox image to create a running process of printing numbers that increase the value by 1 in every one second. Then we have calculated the migration time for this process and which was almost 0.3 second in average. Interestingly, for the same process, the migration time varied between 0.1 second to 0.5 second depending on the host and destination environment. Therefore, when we will use this live migration in the real time application of a fog computing system, we need to find out which one take less delay time to increase the efficiency of the application. We have also analyzed the relationship between tarball size and time, which shows that tarball size increases proportionately with time.

## Chapter 05: Conclusion

Docker is a very popular and growing lightweight container technology. At the same time, more than 30 billion IoT devices or end networks will be connected within 2018. As this small scale applications required a lower latency time, using docker will be very smart to ensure maximum efficiency. Docker has a very exciting feature of live migration. Our plan was to introduce that the live migration can be used in the edge network. Every docker container holds an application. For the large complex application, multiple containers are used. Docker is also a virtualization technology; however, Docker replaces the “hypervisor” that is used in the virtual machine. Docker can run applications very fast with lower latency. It is very easy to upgrade and version control. One of the most important features of docker is that it supports live migration. It means transferring a running process from one node to another node. This live migration is done by CRIU which stands for Checkpoint/Restore In Userspace.

Live migration is still an experimental feature of Docker. However, the biggest challenge was to know it is actually works. Docker is a comparatively new technology and as live migration is an experimental feature; the proper resources were very limited to know more in details. It has also some conditions for the experiment set up. Therefore, my target was to research and know all the conditions and to successfully experiment the live migration. I have been able to do that and I have presented all the steps in details with proper workflow. Live migration has some challenges too. Every live migration requires a transfer to delay time, in most of the applications the delay time is small and it does not affect the running application. However, for real time system, the delay time can be sensitive to the application performance.

In the second part of the implementing, my mission was to calculate the migration time. So the simple busybox container, the migration time was from 0.1 second to 0.5 second. It proves that, the migration time can vary depending of the source and destination environment. Also, the proportionate relationship between tarball size and time has been found in according to the experimental data.

## References

1. C. Pahl and B. Lee, “Containers and Clusters for Edge Cloud Architectures -- A Technology Review,” *2015 3rd International Conference on Future Internet of Things and Cloud*, pp. 379–386, 2015.
2. D. Mulfari, M. Fazio, A. Celesti, M. Villari, and A. Puliafito, “Design of an IoT Cloud System for Container Virtualization on Smart Objects,” *Communications in Computer and Information Science Advances in Service-Oriented and Cloud Computing*, pp. 33–47, 2016.
3. J. Turnbull, *The Docker Book*, vol. v1.0.7 (8f1618c). 2014.
4. Currie, “How Many Public Images are there on Docker Hub? – Microscaling Systems–Medium,” *Medium*, 06-Oct-2016. [Online]. Available: <https://medium.com/microscaling-systems/how-many-public-images-are-there-on-docker-hub-bcdd2f7d6100>. [Accessed: 17-May-2018].
5. *hub.docker.com*. [Online]. Available: <https://hub.docker.com/explore/>. [Accessed: 17-May-2018].
6. OpenVZ Follow. “CRIU: Time and Space Travel Service for Linux Applications.” *LinkedIn SlideShare*, 21 May 2015, [www.slideshare.net/openvz/criu-texaslinuxfest2014140614182445phpapp01](http://www.slideshare.net/openvz/criu-texaslinuxfest2014140614182445phpapp01).
7. Morabito, Roberto. “A Performance Evaluation of Container Technologies on Internet of Things Devices.” *2016 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, 2016.
8. Giannakopoulos, Ioannis, et al. “Isolation in Docker through Layer Encryption.” *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, 2017.
9. Novoseltseva, Ekaterina. “Top 10 Benefits of Docker – Dzone DevOps.” *Dzone.com*, 27 Apr. 2017, [dzone.com/articles/top-10-benefits-of-using-docker](http://dzone.com/articles/top-10-benefits-of-using-docker).
10. Bhakti Bohara, *Moving Target Defense Using Live Migration of Docker Containers*. Arizona State University, August 2017.



11. Chenying Yu, and Fei Huan. *Live Migration of Docker Containers through Logging and Replay*. 3rd International Conference on Mechatronics and Industrial Informatics, Jan. 2015.
12. "Live Migration." *CRIU*, [criu.org/Live\\_migration](http://criu.org/Live_migration).
13. "Containers Live Migration: Behind the Scenes." *InfoQ*, [www.infoq.com/articles/container-live-migration](http://www.infoq.com/articles/container-live-migration).
14. Nadgowda, Shripad, et al. "Voyager: Complete Container State Migration." *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, 2017.
15. Bonomi, Flavio, et al. "Fog Computing and Its Role in the Internet of Things." *Proceedings of the First Edition of the MCC Workshop on Mobile Cloud Computing - MCC 12*, 2012.
16. Fan, Kai, et al. "A Secure and Verifiable Outsourced Access Control Scheme in Fog-Cloud Computing." *Sensors*, vol. 17, no. 7, 2017, p. 1695.
17. Samu Toimela, Containerization of telco cloud applications, 2017
18. Al-Dhuraibi, Yahya, et al. "Autonomic Vertical Elasticity of Docker Containers with ELASTICDOCKER." *2017 IEEE 10th International Conference on Cloud Computing (CLOUD)*, 2017.
19. Dupont, Corentin, et al. "Edge Computing in IoT Context: Horizontal and Vertical Linux Container Migration." *2017 Global Internet of Things Summit (GloTS)*, 2017.
20. "Supported Tags and Respective Dockerfile Links." *Hub.docker.com*, [hub.docker.com/\\_/busybox/](http://hub.docker.com/_/busybox/).