

Container Evaluation - HPSI Assignment #2

José Donato, donato@student.dei.uc.pt, 2016225043

Abstract—Virtual Machines are defined as “software computers”. They can run on top of two types of hypervisors. Type 1 is the most common in production environments, where a dedicated operating system (ESXi, for example) is installed on top of a physical machine normally called “bare metal”. The second type (type 2) operates on top of a common operating system (for example Ubuntu or Windows) and occurs when a user installs a tool like VMware Player or VirtualBox. This last type is more limited because this softwares normally run on a non-dedicated machine for the purpose (against the type one that uses a dedicated physical machine). On the other hand, when talking about microservices, containers can be really useful since they abstract even more of the system than Virtual Machines and have far less overhead. Containers’ goal is to package an application and all the underlying dependencies together. This container package can be run anywhere where a container engine is installed (can run on top of a type 1 or type 2 virtual machine if it has a container engine installed) [1]. This report aims to provide a comparison between these two types of virtualization (containers and virtual machines) in terms of performance.

Index Terms—VMware, VMware ESXi, Virtual Machine, Container, Docker, Linux Containers, Benchmarking, Comparison, CentOS, Dockerfile, Hypervisor, Virtualization

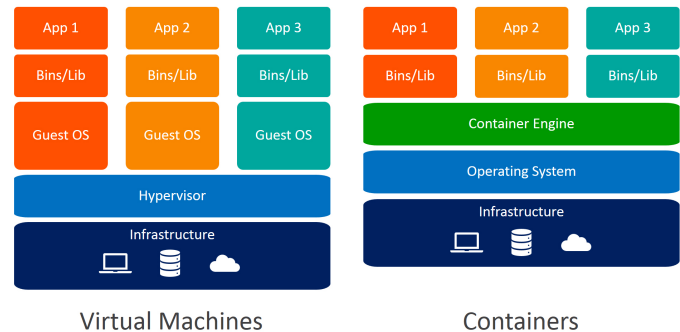
I. INTRODUCTION

This report is divided into two parts. First I start by explaining the state of art on Container evolution and current developments as well as its orchestration. Next, in a more practical way, I do a series of tests to compare both Virtual Machines and Containers always explaining what I will evaluate (I/O performance in terms of networking, mass storage access, memory and CPU overhead).

II. CONTAINER EVOLUTION

A. Introduction

As said in the abstract, Virtual Machines are only possible on machines with hypervisors (the software between host OS and Virtual Machine operating system).



So, people arrived to a conclusion. The next logical step to take in virtualization was to get a quick way to create a isolated area environments on top of a operating system. That is why Containers appeared [2].

Containers can provide virtualization at both application and system level and has some of the following features:

- complete OS environment sandboxed (or isolated)
- packaging of applications with isolation
- portable and lightweight environment that can run anywhere
- maximize resource utilization in data centers
- since it is fast to build a container, we can easily have multiple development environments (test, production, etc)

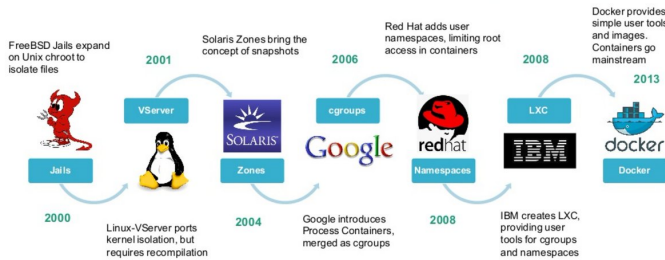
The containers can bundle isolated applications and their dependencies that can run anywhere. We have many other advantages when using containers:

- they are lightweight compared to virtual machines. This means that more containers can run per host machine.
- starting a container is almost instant
- it grants us the capacity to fully utilize the computing resources

Before linux kernel officially introduced the possibility to create containers natively or docker was a reality lot of work and research had to be done. In the next subsection, a timeline is made from the roots of the containerization to the current times [3].

B. Timeline

Container Evolution Timeline		
Year	Technology	OS supported
1982	Chroot	Unix-like
2000	Jails	FreeBSD
2000	Virtuozzo	Linux/Windows
2001	Linux VServer	Linux/Windows
2004	Solaris Container	Sun Solaris/Open Solaris
2005	OpenVZ	Linux
2006	Process Containers	Linux
2008	LXC	Linux
2011	Warden	Linux
2013	LMCTFY	Linux
2013	Docker	Linux/MacOS/Windows
2016	Security in Containers	-
2017	Kubernetes	Linux



In the following subsection, I explain the most important technologies that got us to the point where we are now with containers [4].

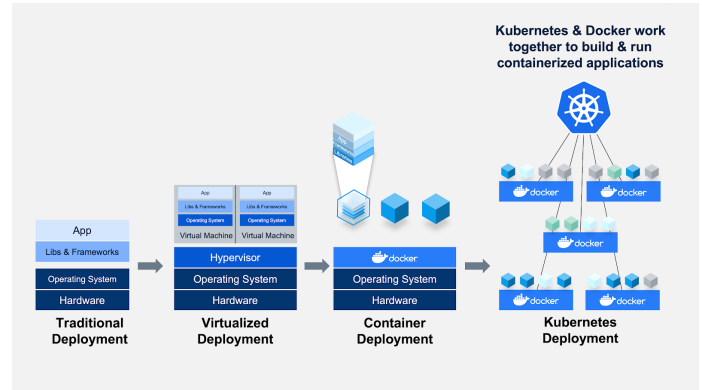
C. Explanation

- 1) **Chroot:** filesystem isolation by switching root directory for a process and their children to another new location in the filesystem.
- 2) **FreeBSD Jails:** provides the ability to partition the operating system environment, maintaining the simplicity of the UNIX "root" model. The requests are limited to the jail, allowing sys admin to delegate management capabilities for each virtual machine environment. With FreeBSD jails we can create multiple native virtual environments and it is based on the creation of a different directory, hostname and a network address for the specific virtual machine (can be accessed via ssh on this address), i.e., we can partition a FreeBSD computer into several independent smaller systems (jails) each one with an IP address. The jails provide a separation between services.
- 3) **Linux VServer:** similar to freebsd jails. Partition resources (file systems, network addresses and

memory) on a computer. This type of virtualization could be achieved by patching linux kernel.

- 4) **Solaris Containers:** is the combination of system resource controls and has isolation provided by zones (completed isolation virtual servers on one host machine). Zones have a low overhead on CPU and memory (this grants some advantages over virtual machines on some cases). They can cap the resource pool or the compute capacity to a certain limit.
- 5) **Open VZ:** uses a patched linux kernel (like Linux VServer) with capabilities of virtualization, isolation, resource management and checkpointing (not officially provided from linux kernel).
- 6) **Process Containers:** in 2006, Google designed process containers with the goal of limiting, accounting and isolating resource usage (cpu, memory, disk i/o, network) of a collection of processes. They were after renamed to control groups (cgroups) and merged with linux kernel (this is one of the features that made lxc possible, talked below).
- 7) **LXC (Linux Container):** most complete implementation of linux container manager. Works on a linux kernel without the need of patches. Key features were added to the linux kernel making possible container virtualization as we see in the first versions of tools such as docker. Manage to provide a contained/isolated area in the host machine. This isolated area is similar to virtual machine but with containers we don't need a hypervisor. The three key features enabled containers were:
 - **Control Groups:** solution presented by google to a problem: a running process when request for resources that are unavailable gets deleted by system. With control groups, resources can be controlled and accounted for based on. Process groups aggregate tasks and their future children into hierarchical groups.
 - **Namespaces:** abstraction to a global system resource that appear to the processes. Used to implement containers. Provide the isolation between container and host system.
 - **Filesystem:** disk image, provides the root filesystem for the container. Set of files mounted at root on any linux-based machine. Note that the container shares the same kernel as host machine. The size can be reduce by

simply contain the application and share the filesystem with the host machine. Using Copy On Write (COW), only the pointers are saved and only one disk image can be shared between multiple containers (allowing reduced sizes).



- 8) **Warden:** uses LXC and can isolate environments on any OS with a daemon and an API for container management running. Include the features of LXC (cgroups, namespaces and the process life cycle). It can manage a collection of containers across multiple hosts.
- 9) **Let Me Contain That For You (LMCTFY):** open-source version of Google container stack. It provided linux application containers. Ended after google contributing the core of LMCTFY to libcontainer (used in docker).
- 10) **Docker:** this tool made containers popularity explode exponentially. Similar to warden, in the beginning, Docker used LXC and replaced it later with their own library *libcontainer*. They offered an entire ecosystem for containers easy to manage. This tool provides a tool that makes easy to interact with linux containers and is being championed by many as the new standard in cloud software management [2].
- 11) **Security:** in 2016, with the high adoption of containers, vulnerabilities also expanded. This led to an increase importance of security in containers. The goal was to build secure containers from the beginning.
- 12) **Kubernetes:** in 2017, open-source project from google appeared called kubernetes. First project that received a donation from Cloud Native Computing Foundation (CNCF) and is now supported by them. It is a container orchestrator and allows complex containerized applications [5] to scale easily. We can call this tool the vSphere (or ESXi) of the containers' world. It allows companies to easily manage the containers and, consequently, their applications (comparing to virtual machines, vSphere or ESXi manage the Virtual Machines needed in a data center).

This figure explains well the difference between a traditional deployment, a virtualized deployment, container deployment and a kubernetes deployment [6]. Traditional deployments were used when there were no virtualized environments, then with virtualized data centers the virtualized deployments appeared with different virtual machines on a physical server (VMware ESXi or vSphere for example). With the birth and stabilization of containers, the process of developing and deploying applications turn out much easier but when that application gains high usage and popularity, it becomes a problem. That is where Container Orchestration tools such as Kubernetes comes to rescue. Kubernetes makes the process of deploying and scaling up complex containerized applications easy, managing multiple containers for you.

In the beginning of containers, people always thought Containers and Virtual Machines were rivals. Now, people think the same about docker and kubernetes. There are situations that ones are better and another situations that they are not recommended, in the majority of the times they are not competing against each other and can be used simultaneously. Kubernetes can use docker containers in the same way we can use containers inside virtual machines.

III. BENCHMARKINGS

A. Introduction

In this second section of the report, a series of benchmarking to measure the performance between Virtual Machines and Containers was done.

B. Tools

To measure container-level indicators the tool provided by docker: `docker stats` (<https://docs.docker.com/engine/reference/commandline/stats/>) was used (to measure CPU and Memory). Also, the VMware ESXi monitor was used to measure the machine that runs the container (for disk and network I/O). To measure the

virtual machine without the container only the VMware ESXi interface statistics was used.


C. Test Environment

All the tests were performed on the server running VMware ESXi 6.7 provided in the course. For consistency, the container and virtual machines will be using the same operating system, CentOS (virtual machines use CentOS 8 and container CentOS 7 because the version 8 wasn't stable).

Two machines were used:
Machines to perform the tests

Name in ESXi	OS	CPU (number)	Memory (GB)	Disk (GB)	Linux Containers
jdonato	Cent OS 8	2	4	40	No
jdonato-client	Cent OS 8	2	4	40	Yes

As we can see in the previous table, both machines are provisioned with exactly the same resources. The following image shows the resources each machine has in VMware ESXi (only one image because they are both the same):

Hardware Configuration	
CPU	2 vCPUs
Memory	4 GB
Hard disk 1	40 GB
USB controller	USB 2.0
Network adapter 1	VM Network (Connected)
Network adapter 2	testjdonato (Connected)
Network adapter 3	FRONTEND (Connected)
Video card	4 MB
CD/DVD drive 1	ISO [local_DS] ISO/CentOS-8-x86_64-1905-boot.iso 
Others	Additional Hardware

The only difference between the two machines is that one has linux containers running (with the help of docker) and the other doesn't. Both performed the same operations and ran the same tools in order to compare them. Because they have not been updated since their creation, I started by updating them:

```
$ sudo yum update
```

After the machines have updated, I installed docker-ce in jdonato-client machine following this tutorial: <https://linuxide.com/linux-how-to/how-to-install-docker-on-centos/> and created a

Dockerfile with CentOS image. The Dockerfile is simple and can be seen below:

```
# our base image
FROM centos:7
# run updates
# install wget to get necessary
# tools
# clean the packages
RUN yum -y update && \
yum -y install wget && \
yum clean all
```

The docker image can be built by running:

```
$ docker build --network host -t CentOS .
```

And can be later started with:

```
# --network host to get access to internet
# from the host, i.e., container in the same
# network as host
$ docker run -it --network host CentOS
```

When we build the docker image with this dockerfile, we can see start the comparisons. The virtual machines are currently with around 5GB occupied. The linux container with a Cent OS image is fewer than 300MB:

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
cent	latest	ble8df7018eb	13 hours ago	290MB

We begin to see clearly the differences between the two systems by highlighting how lightweight containers can be relative to a virtual machine. When comparing these two values (300MB and 5GB), we can see that we could have around 16 containers with this size (300MB) and still wouldn't be the same size ($16 \times 0.3 = 4.8\text{GB}$) as a virtual machine with almost no software installed besides the CentOS operating system (5GB).

After the container is running, we have access to a bash shell of the container with the latest 7 version of Cent OS (I tried running the version 8 to match with the Virtual Machine OS but, as I said before, due to some instability of Cent OS docker images it was not possible). With the shell we can run commands just like in the terminal of the virtual machine with CentOS.

Since docker have great tools to gather statistics from the containers running I used this tools (docker-stats) to perform the tests. If I run the following command from the host machine running docker:

```
$ docker stats <container_id>
```

I get the following output:

CONTAINER ID	NAME	CPU %	MEM USAGE / LIMIT	MEM %	NET I/O	BLOCK I/O
0b9c1d5-dca	cilly_stallman	0.00%	10.97MB / 3.63GB	0.30%	0B / 0B	0B / 0B

As we can see, it provides some important statistics about the container: CPU, memory usage, Disk and Network I/O, etc. To save this statistics to a file for further analysis, I used the following command:

```
$ while true; do docker stats \
--no-stream \
>> stats_docker.txt; done
```

This way, until i stop the command (with control+c or similar), the output of docker stats will be appended to "stats_docker.txt".

Additionally, network and disk I/O statistics were gathered directly from the VMware ESXi since the container is directly connected to the host network and the disk I/O from docker stats was not accurate.

The same tests that were run on the container were run also on the virtual machine without the container, after all it is the purpose of this assignment, to compare both situations. To measure the indicators in the virtual machine, as I said before, I used the VMware monitor tool inside ESXi. Like the docker stats, it also reports indicators good enough for this kind of comparison.

D. Tests Performed

The list below enumerates the tests that were performed on both the container and the virtual machine:

- 1) Download of a big file (2.6GB) to compare CPU/Memory/Disk/Network usage
- 2) Run "stress" (<https://www.cyberciti.biz/faq/stress-test-linux-unix-server-with-stress-ng/>) to generate workload for Unix systems, it can impose a configurable amount of CPU, memory, I/O and disk stress on the system. To install it is as simple as running in both container and virtual machine the following command:

```
$ sudo yum install stress
```

In the following subsections, both tests will be explained and the results exposed.

E. Downloading File

In this first test, a file with 2.6 GB was downloaded with the command:

```
$ wget https://cdimage.kali.org/ \
kali-2019.4/kali-linux-2019.4 \
-amd64.iso
```

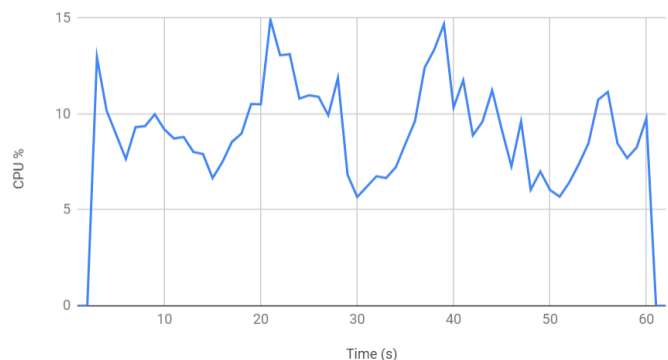
When downloading a big file, we can compare CPU, Memory, Network and Disk I/O usage in both systems (container and virtual machine). All the results are in the following spreadsheet <https://docs.google.com/spreadsheets/d/1zeIbgFRCnKQQv1VKIAVhzKcfZFbAHGjya8sLDk5vD7U/edit?usp=sharing>.

[1zeIbgFRCnKQQv1VKIAVhzKcfZFbAHGjya8sLDk5vD7U/edit?usp=sharing](https://docs.google.com/spreadsheets/d/1zeIbgFRCnKQQv1VKIAVhzKcfZFbAHGjya8sLDk5vD7U/edit?usp=sharing).

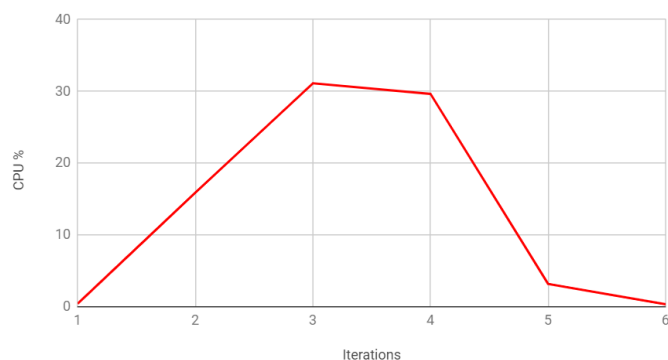
The following images and tables are the results for the downloading file test. For each type of usage, I present two images containing the utilization in both container and virtual machine and a table that contains the average, minimum and maximum values of each one.

1. CPU

CPU utilization % in Container



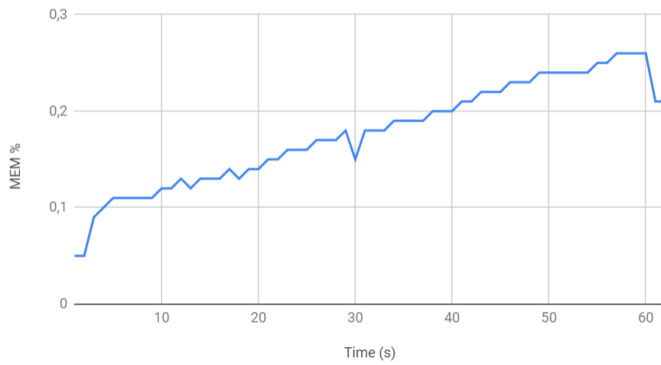
CPU utilization % in VM



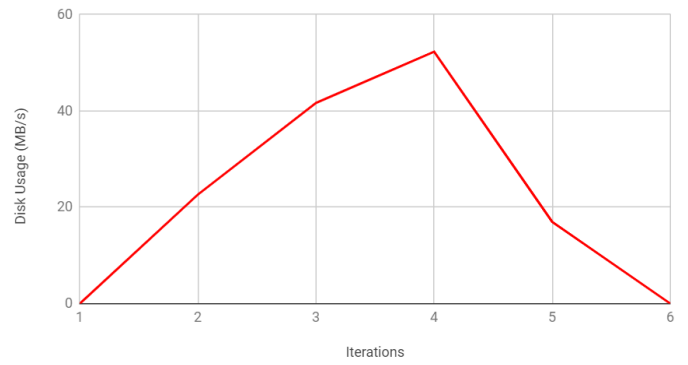
CPU (%) Container vs Virtual Machine		
/	Container	VM
AVG	8,68	13,43
MIN	0	0,35
MAX	14,91	31,12

2. Memory

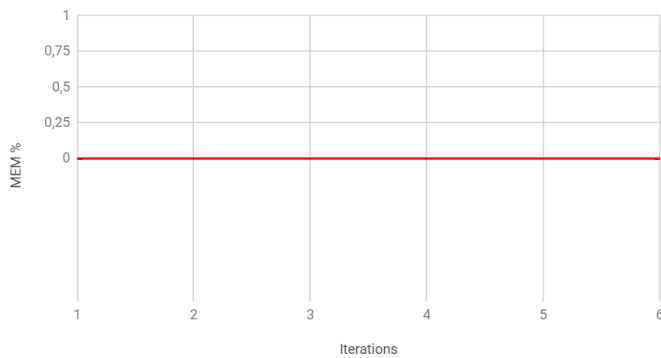
Memory utilization % in Container



DISK IO MB/s in VM



Memory utilization % in VM

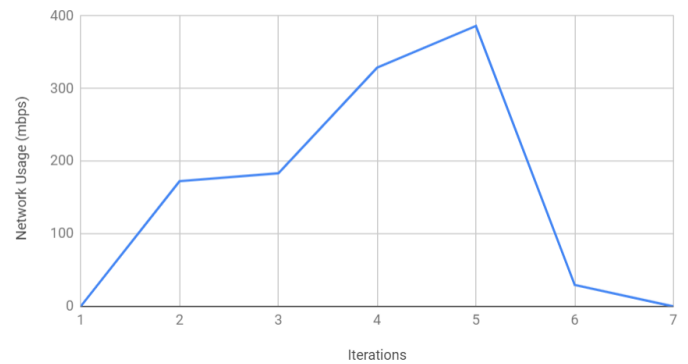


DISK IO (MB/s) Container vs Virtual Machine

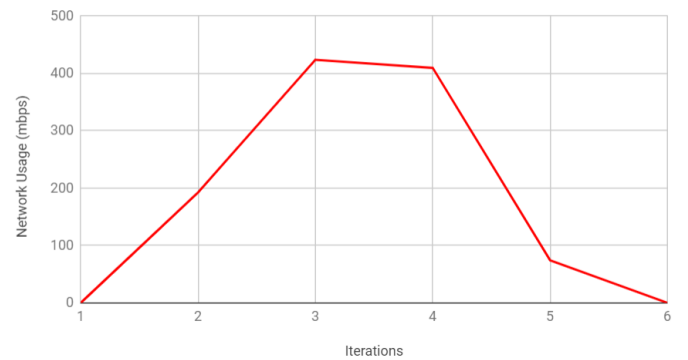
/	Container	VM
AVG	19,60	22,26
MIN	0,01	0
MAX	47,88	52,29

4. Network IO

NETWORK IO mbps in Container

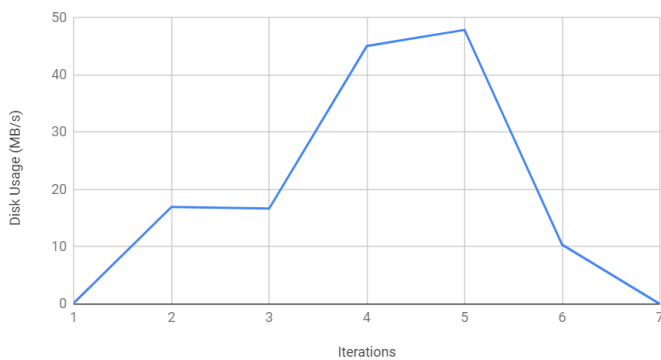


NETWORK IO mbps in VM



3. Disk IO

DISK IO MB/s in Container



NETWORK IO (mbps) Container vs Virtual Machine

/	Container	VM
AVG	157,21	183,33
MIN	0	0
MAX	386,28	423,7

F. Stress Test

After the first, another test was performed. In this one, I tried to do a stress test using the unix tool named "stress" to measure CPU, Memory and Disk IO.

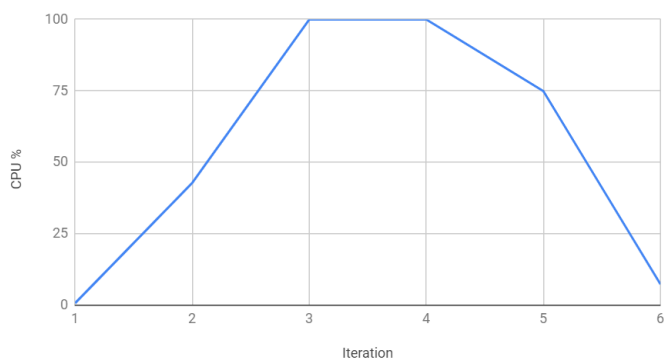
The command used to perform the tests was:

```
$ stress --cpu 8 --io 4 --vm 2 \
--vm-bytes 128M --hdd 4 --timeout 60s
```

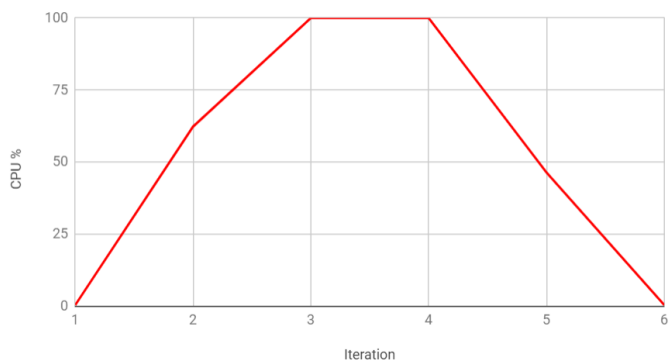
Below there are the results from this test:

1. CPU

CPU utilization % in Container



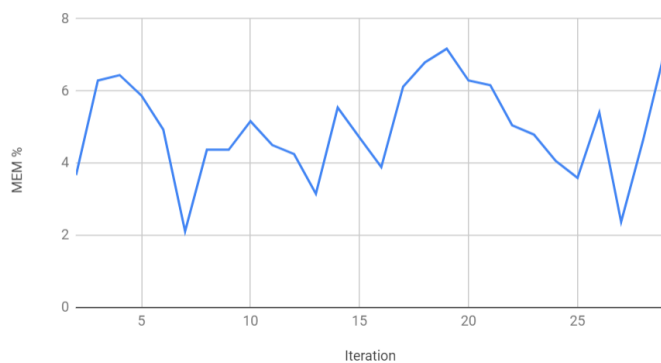
CPU utilization % in VM



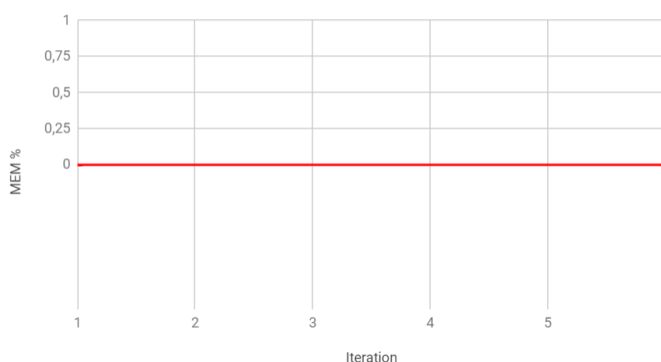
CPU (%) Container vs Virtual Machine		
/	Container	VM
AVG	54,23	51,61
MIN	0,68	0,45
MAX	100	100

2. Memory

Memory utilization % in Container



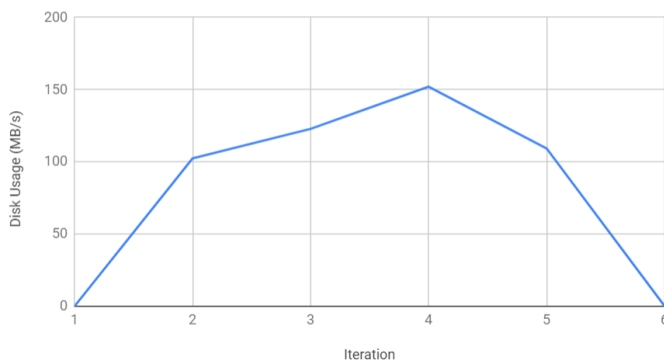
Memory utilization % in VM



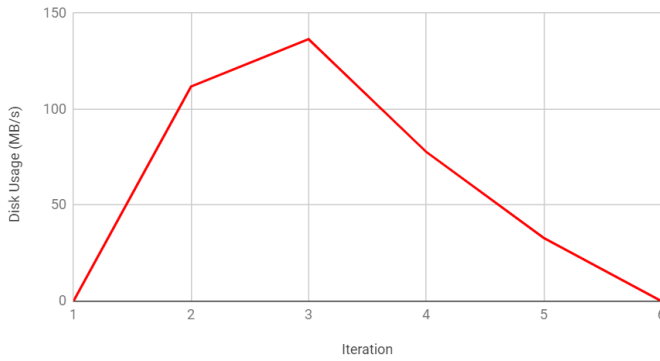
Memory (%) Container vs Virtual Machine		
/	Container	VM
AVG	4,80	0
MIN	0,51	0
MAX	7,17	0

3. Disk IO

DISK IO MB/s in Container



DISK IO MB/s in Container



DISK IO (MB/s) Container vs Virtual Machine		
/	Container	VM
AVG	81,09	58,83
MIN	0	0
MAX	152,05	136,5

IV. DISCUSSION

After analyzing the results I obtained some conclusions.

From a CPU level, we can observe that a full virtual machine has far more overhead than a lightweight container. Therefore, the CPU utilization in Virtual Machines are far higher (can be clearly seen in the first test).

Regarding memory usage, I couldn't come up with good conclusions. In both tests, the container consumed very little memory (at max 5 %). Although, I couldn't compare it to Virtual Machine since VMware ESXi Monitor always showed 0% on memory usage in all tests.

Disk IO results were interesting. Although in the first test Virtual Machine had slightly better results (around 2.5MB/s higher), in the second one the container surpassed the virtual machine, with an average of more 23 MB/s and peaked over 152.05 MB/s while virtual machine only reached 136.5 MB/s. Since the containers introduce no overhead, the performance of disk writes are even better than in a virtual machine. Therefore, when a developer is choosing his virtual environment, if his applications require a high disk usage, containers should be considered as the main option [7].

Finally, as expected, the virtual machine won the network "battle". Since the container was using the host network (i.e., the same network as the virtual machine), it was expected to have the same or lower network IO.

To sum up, in my opinion, the results are similar in both VM and container. From a performance level, container gains some points for being lightweight but

loses another to Virtual Machines that are more robust and powerful in other situations.

V. CONCLUSION

A brief performance comparison between Linux Containers and Virtual Machines was done in this report. As expected, we have seen that Linux Containers are far more lightweight than Virtual Machines (the last ones introduce more overhead than the containers). Because of this they have their advantages. On the other hand, there are situations where Virtual Machines are more recommended.

Therefore, in my opinion, they are not direct opponents as they can often be used together. There are times when it is most recommended to use virtual machines, sometimes when it is advisable to use containers and even situations where they can be used simultaneously. It is up to the developer or system administrator to decide what is best for the situation.

REFERENCES

- [1] David Zomaya. Container vs. Hypervisor: What's the Difference? <https://www.cbttuggets.com/blog/certifications/cloud/container-v-hypervisor-whats-the-difference>.
- [2] Alex Loxham. Docker and Containerisation: An introduction. <https://ember.co.uk/articles/docker-and-containerisation-an-introduction>.
- [3] Senthil Kumaran S. Practical LXC and LXD. <https://www.springer.com/gp/book/9781484230237>.
- [4] Rani Osnat. A Brief History of Containers: From the 1970s to 2017. <https://blog.aquasec.com/a-brief-history-of-containers-from-1970s-chroot-to-docker-2016>.
- [5] Ann Mary Joy. Performance comparison between Linux containers and virtual machines. <https://ieeexplore.ieee.org/document/7164727>.
- [6] Jim Armstrong. Docker and Kubernetes? I Thought You Were Competitors! <https://www.docker.com/blog/top-questions-docker-kubernetes-competitors-or-together/>.
- [7] Ridip De Anish Grover Sumit Maheshwari, Saurabh Deochake. Comparative Study of Virtual Machines and Containers for DevOps Developers. https://www.researchgate.net/publication/327237114_Comparative_Study_of_Virtual_Machines_and_Containers_for_DevOps_Developers.