Theoretical part
(objectives 1 to 3)

1. Difference between the main virtualisation hosts (VM et CT)

First we can explain what are Virtual Machine and a Container.

The Virtual Machine (VM) is an operating system (OS) or application environment that is installed on a software which imitates dedicated hardware: mouse, memory, CPU, or network cards. The user has the same experience on the virtual machine as they would have on dedicated hardware[1].

The container is a standard unit of software that regroups code and all its dependencies. Containers include everything to run an application: code, runtime, system tools, system libraries and settings. They always run the same, regardless of the infrastructure, they isolate software from their environment and ensure that they work well when they are moved from one server environment to another[2].

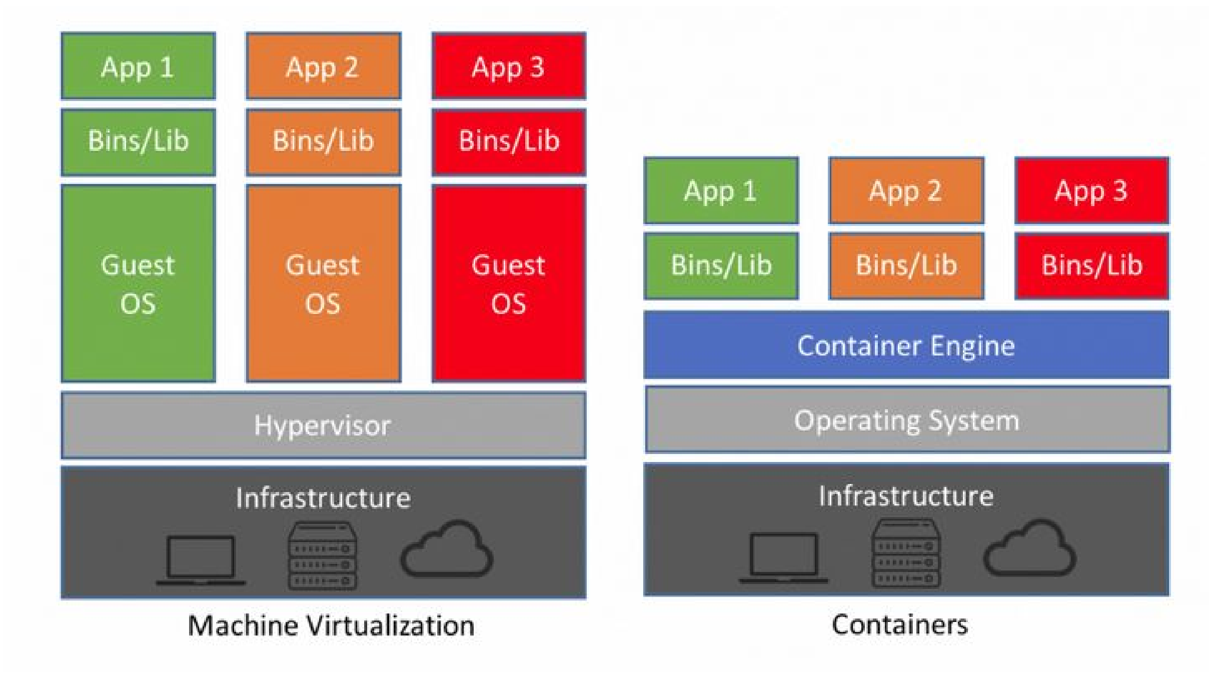


Figure 1: Comparison between Virtual Machine and Containers

One of the main difference between VM and Containers is that for VM, the hypervisor virtualizes the hardware of the host so that the virtual OS appears to have its own processor, memory and other hardware resources. The containers run multiple virtual instance of the

[1] Virtual Machine: https://searchservervirtualization.techtarget.com/definition/virtual-machine
[2] Container: https://www.docker.com/resources/what-container

same OS but over a single hardware (the one of the host). This means according to the virtualization cost, VM needs a lot of resources for the memory size and the CPU.

From an application developer's view, VM has its drawbacks. In matter of facts, they add overhead in memory, storage footprint and CPU usage. As it turns out, this issue adds complexity to all the stages of a software development lifecycle, this ranges from development and test to the production, and includes disaster recovery. So, containers can be created much faster than VM instances[3].

Then, containers allow the developers to upgrade the application in a much easier way than for a VM, they also allow to follow all the different versions, and compare them. Containers also isolate the processes of one container with another one and the underlying infrastructure as well. Thus any upgrade or changes in one container do not affect another container.

From an application developer's view, container is an immature technology, it is relatively new in the market, this means that the number of resources available is limited along the developers.

Moreover, one of the main problem with the containers is the security. Containers have access to some critical namespaces, and a lot of important Linux kernel or Windows subsystems are outside the container. So, if an user or application has a superuser privileges within the container, the underlying operating system could, in theory, be cracked. Container should not be run with root or administrator privileges. That is why, during the configuration of the container, you need to pay attention at a lot of potential gate for hackers and the level of access the container. For example, you can set the file system in read-only, like this, the container processes are forced to write only to container-specific file[4].

On the other hand, VM have also their own security defaults.

From an infrastructure administrator point of view, because all the containers share a common operating system, this reduce the management overhead and only a single operating system needs to deal with the bug fixes, patches and so on[5]. However, if the number of containers running an application is too high, the complexity increase dramatically, and the management of too many containers can be challenging for the operating system.

- virtualization cost, taking into consideration memory size and CPU,

[3] Development phase: https://dzone.com/articles/container-technologies-overview
[4] Security problem: https://www.itworld.com/article/2915530/virtualization/containers-vs-virtual-machines-how-to-tell-which-is-the-right-choice-for-your-enterprise.html
[5] Management of Container: https://blog.netapp.com/blogs/containers-vs-vms/

- Usage of CPU, memory and network for a given application,
- **Security for the application, regarding access right and resources sharing,**
- **Performances regarding response time,**
- Tooling for the continuous integration support

https://pubs.vmware.com/vsphere-4-esx-vcenter/index.jsp?topic=/com.vmware.vsphere.server_configclassic.doc_40/esx_server_config/security_for_esx_systems/c_security_and_virtual_machines.html

This approach also severely limits the portability of applications between public clouds, private clouds, and traditional data centers.

2. Differences between the existing CT types

Linux LxC:

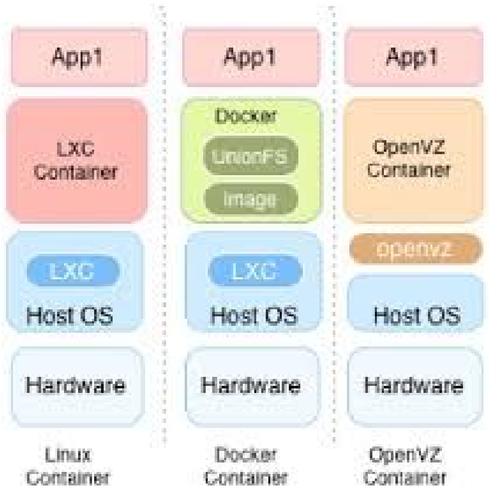
- OS-level virtualization
- Each container is assign a unique process ID. Each container can run a single process.
- Containers get a private IP address and virtual ethernet interface connecting to a linux bridge on the host.
- Allows creation and running of multiple isolated Linux Virtual Environment (VE) to run applications.
- Allows limitation and prioritization of resources like CPU or memory.
- Tooling: API, continuous integration.

Docker:

- Relies on the host operating system.
- Each container is assign a unique process ID.
- Containers get a private IP address and virtual ethernet interface connecting to a linux bridge on the host.
- Cannot run multiple processes in a single container.
- Packages application and all its dependencies in a virtual container.
- Tooling: API, continuous integration.

OpenVZ:

- OS-level virtualization
- Each container has its own PID.
- Uses net namespace, has its own virtual network device and its own IP address.
- Allows creation and running of multiple isolated Linux Virtual Environment (VE) to run applications.
- Requires a special kernel to be installed.
- Access to physical hardware from inside a container is disabled by default (good security feature but need to configure manually if you want to use an hardware component).
- Tooling: API.

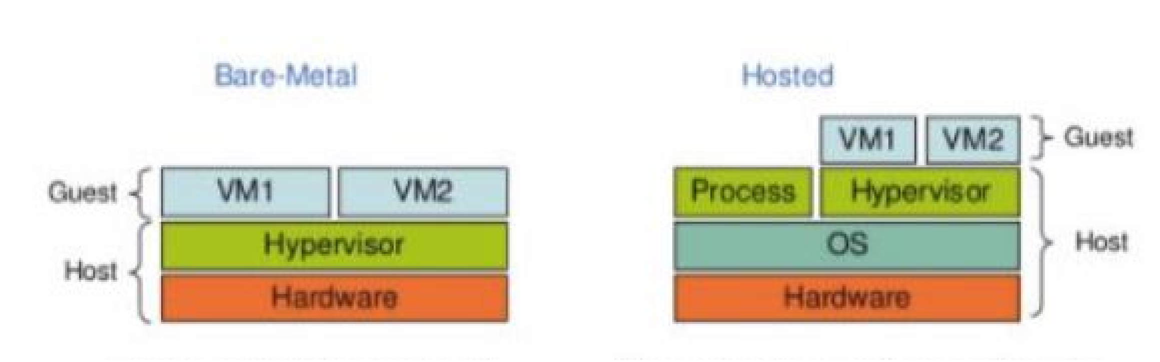


Comparison between LxC, Docker and OpenVZ

3. Differences between Type 1 & Type 2 for hypervisors' architectures

There is two types of hypervisors' architectures:

1. **Bare-metal (or Hardware level)** in which the hypervisor run directly on the system hardware. It controls the hardware and manage guest operating systems.
2. **Hosted (or OS level)** in which the hypervisor run on a host operating system like other process. VirtualBox uses this type of hypervisor architecture as well as Proxmox which uses KVM (*Kernel-based Virtual Machine*) hypervisor.



Practical part
(objectives 4 to 7)

1 - Creating a VirtualBox VM (in NAT mode), and setting up the network to enable two-way communication with the outside

After creating the VM and applying the given configurations in the subject we obtain the ip address of the VM, which is: **10.0.2.15**
The IP address of the host system is: **10.1.5.46**

Second part: Testing connectivity

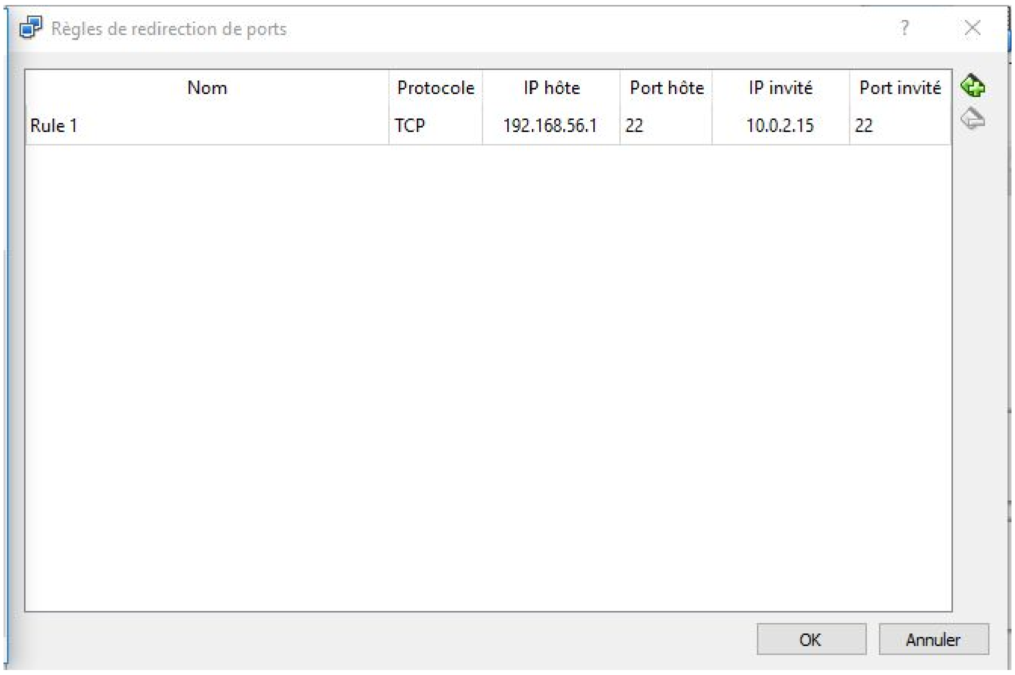
- **Check the connectivity from the VM to the outside. What do you observe?**
  We check the connectivity with the outside by making a *ping 8.8.8.8*. This one works ! This shows that the VM can reach the outside. In fact, the hypervisor (VirtualBox) act as a NAT, allowing our ping to transit to the LAN (outside).

- **Check the connectivity from your neighbour's host to your hosted VM. What do you observe?**
  It is impossible for our neighbour's host to reach our VM. This is because our VM is on it's own (virtual) network which is unreachable from the outside.

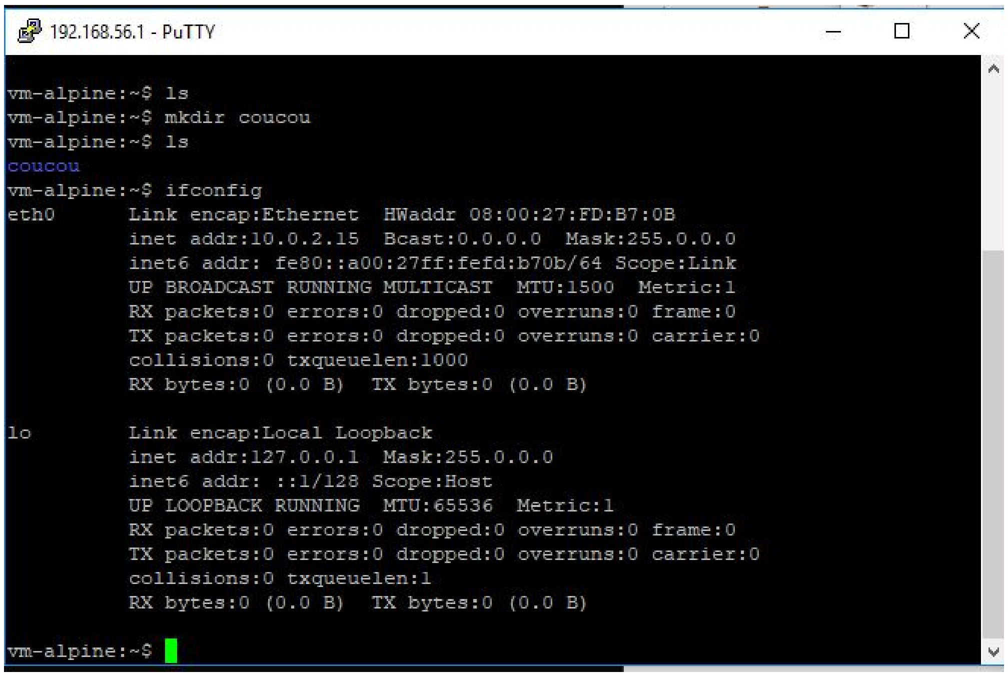
- **Check the connectivity from your host to the hosted VM. What do you observe?**
  If we make *ping 10.0.2.15* this doesn't work. This is for the same reason as the previous question.
  If we make *ping 192.168.56.1* which is the IP address of "VirtualBox Host-Only Network", of course it works because this is the address of the virtual network created by VirtualBox for our VM.

Third part: Set up the "missing" connectivity

In this part we are going to add a connectivity from the outside to our VM. To do that, we use the *port-forwarding* method to add the *SSH* (port 22) connectivity.
Once the configuration is done, we try with *Putty*, on the host machine, to connect by *SSH* to our VM.
As shown in the following pictures, the *SSH* works:



Nom	Protocole	IP hôte	Port hôte	IP invité	Port invité
Rule 1	TCP	192.168.56.1	22	10.0.2.15	22



```
vm-alpine:~$ ls
vm-alpine:~$ mkdir coucou
vm-alpine:~$ ls
coucou
vm-alpine:~$ ifconfig
eth0      Link encap:Ethernet  HWaddr 08:00:27:FD:B7:0B
          inet addr:10.0.2.15  Bcast:0.0.0.0  Mask:255.0.0.0
          inet6 addr: fe80::a00:27ff:fefd:b70b/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)

vm-alpine:~$
```

2 - Create a Proxmox CT (in Bridge and DHCP mode), Manually snapshot and restore this CT, Manually migrate this CT from the host server to another

Second part: Connectivity test