

**CLOUD COMPUTING : Adaptability and Autonomic
Management**

GEI 2020 - 2021

Introduction to Cloud Hypervisors

BERTA Pauline

BERRADA Ghali

CONCEICAO NUNES Joao

SISS – Promo 54

Table of Contents

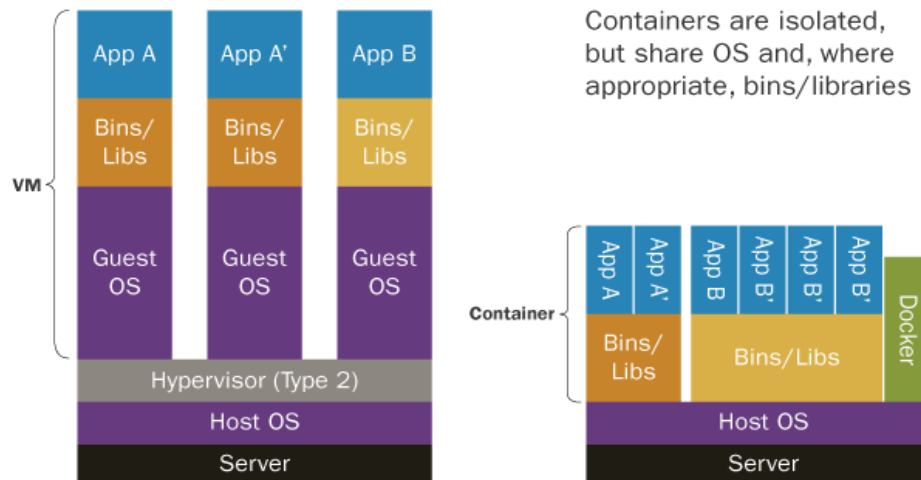
I – Theoretical Part	3
Similarities and differences between the main virtualisation hosts (VM et CT)	3
Similarities and differences between the existing CT types	5
Similarities and differences between Type 1 & Type 2 of hypervisor's architectures	6
II – Practical part	7
Creating a VirtualBox VM (in NAT mode), and setting up the network to enable two-way communication with the outside	7
Clone the VM	10
Docker containers provisioning	10
CT creation and configuration on OpenStack	13
Snapshot, restore and resize a VM	18
Openstack client installation	18
Web-2-tier application topology and specification	19
Deploy the Calculator application on OpenStack	20
Conclusion	21

I – Theoretical Part

Similarities and differences between the main virtualisation hosts (VM et CT)

When we work with cloud computing technologies we have a large panel of different solutions available. For this Lab we will take a closer look to VM and CT technologies.

Containers vs. VMs



The **main difference** is the fact that with a VM we can manage different OS, named “guest OS” controlled by the Hyper Supervisor that runs on the “Host OS”. This means the VM gives the ability to switch from OS to OS without having to reboot the computer, and use different apps with every VM, or the same app but cloned on each VM. Those apps will evolve totally independently. On the other hand CT, is like a normal single boot, only one OS is running but can run different apps, isolated from each other.

Let's compare the two technologies with the board below:

Comparison factor	Virtual Machine	Container
Virtualization cost	The physical hardware from the host machine is virtualized for each VM, and creates the virtual resources used by each VM, exclusively.	The physical hardware isn't virtualized and reserved to a specific CT. All the hardware is available and used to run the different configured CT.
Usage of CPU	The CPU usage is divided between the different VM running.	No division needed.
Security	The fact everything is divided provides a clean	There isn't any guest OS, so everything that is done

	separation on the software and increases the security level. The host OS is separated from the guest OS, and every file on the host OS isn't reachable from the guest OS.	on the host OS will be permanent. The separation is only between containers.
Performance	The CPU and memory available are divided between each VM, so the overall performance decreases. The response time is higher.	All the CPU and memory is available for each CT. The response time is lower.
Continuous integration support	Not available	Available and commonly used in the development world.

We can see that these two technologies have very different ways of working and each one has its own advantages and disadvantages, so let's compare them with two different points of view.

From the **server administrator's** point of view :

The server admin usually wants to ensure that the system runs properly and so security is one of his priorities. The VM is the only technology allowing to separate everything and ensure that if there is a problem somewhere, it will not spread to the entire system.

But the VM technology can be used with CT's because if the server admin wants to add more speed to the system he can organize the server by implementing different VMs with different CTs inside. This will combine the security and speed aspect by separating every CT group inside each VM.

From the **developer's** point of view :

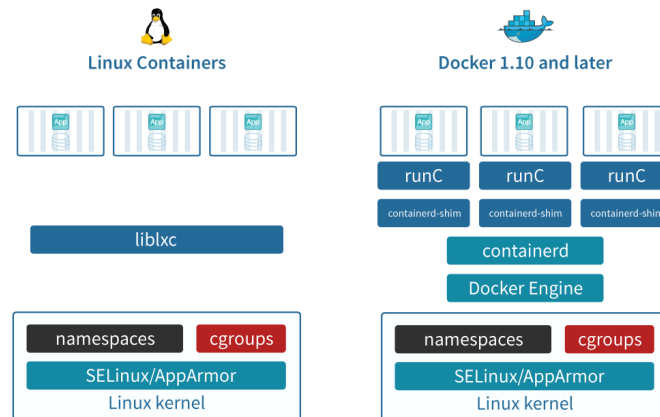
The VM allows the developer to run/develop on different OS using the same machine and without having to reboot at each time. But the fact of using a VM, divides the memory and performance available, making the overall performance less important. So, it depends if the developer is looking for practicality or performance. If performance is what he wants, the CT technology is the right solution but it will not allow him to work on different OS's at the same time.

The separation provided by the VM means that every software running on the guest OS only has the right to write on the guest OS files and not the host OS files. That

keeps every important file safe, and gives the developer a secure environment to work.

Similarities and differences between the existing CT types

Now that we have compared the VM and CT technology we will take a closer look to the different CT services.



For Linux Container, **LXC** :

Linux LXC is an OS level virtualization process, multiple isolated Linux containers can be run at the same time on a single control host. This CT technology is based on the concept of Linux control groups, the cgroups. LXC provides a virtual environment which offers isolation between each control group, which provides applications with complete isolation of resources, including CPU, memory and I/O accesses. The LXC behaves like a VM process, and it means that the containerization level of LXC is the OS level.

Each LXC container can be used as a sandbox, which gives the developer an easier development environment, capable of holding errors.

For **Docker containers** :

Docker is based on the Linux containers, it is used to add higher level capabilities, to build single application containers. This CT technology builds containers using read only layers of the file system and allows applications to be developed and deployed more quickly, using fewer resources. The containerization level of the Docker technology is at the application level.

Because docker containers come with multiple services like integration tools, it makes them an optimized and easier solution to applications for the cloud. The main advantage of Docker is that it is an engine for containers that packs all the applications, along with the dependencies. That means that docker is a portable solution.

But because each container is managed by the same OS, an attack on that OS can damage the entire container's network.

Similarities and differences between Type 1 & Type 2 of hypervisor's architectures

The next step in our theoretical analysis is to compare the two types of Hypervisors architecture.

The **Type1 hypervisor**, is basically the only layer of software directly installed on top of the physical server. The particularity is that Type 1 hypervisors are OS themselves so they are often more performant and stable because they don't run inside another OS. Because of that, the type 1 hypervisor can only run virtual machines because its OS only can be used to do that. Every other application has to be run by the virtual OS (guest OS).

In terms of functionalities offered, the Type 1 offers some simple ones. You can create virtual instances, change the date, the IP address, etc. But you can also allocate more hardware resources than you actually have. The hypervisor will dynamically adjust the resources used according to the resources available at the time.

The **Type 2 hypervisor**, called Hosted, has a very different architecture because they run inside an OS. Basically the hypervisor has one software layer between it and the hardware, where the Type 1 was directly on the hardware layer. The Type 2 hypervisor is usually used on a small number of servers, and it's convenient because there is no management console needed on another machine to set it up and manage every VM. All can be directly done on the server where the hypervisor is installed. It's exactly the type of hypervisor used on our computer when we use Virtualbox for example. Our OS (Windows, Linux etc) is the one running the Type 2 hypervisor for the VM. The problem with this type 2, is that the resources allocated are not dynamically managed by the hypervisor. If we allocate 8GB of ram, the VM will take 8GB of RAM. That's one of the main differences with Type 1. But type 2 remains convenient for testing because it's simple to use the same physical machine with multiple instances and so to execute the code on different instances.

II – Practical part

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

In this part we'll use VirtualBox hypervisor in NAT mode to connect a VM to the network.

If we check the connectivity from the VM to the outside using :

ping facebook.com

With this command we are trying to ping an exterior server by the name of facebook.com and we can see that the connection is successful. This works because the connection is made possible by the VM's NAT, from the VM to the exterior.

```
arm-ada@armada-VirtualBox: /etc$ ping facebook.com
PING facebook.com (157.240.212.35) 56(84) bytes of data.
64 bytes from edge-star-mini-shv-01-lis1.facebook.com (157.240.212.35): icmp_seq=1 ttl=55 time=147 ms
64 bytes from edge-star-mini-shv-01-lis1.facebook.com (157.240.212.35): icmp_seq=2 ttl=55 time=151 ms
64 bytes from edge-star-mini-shv-01-lis1.facebook.com (157.240.212.35): icmp_seq=3 ttl=55 time=149 ms
64 bytes from edge-star-mini-shv-01-lis1.facebook.com (157.240.212.35): icmp_seq=4 ttl=55 time=167 ms
64 bytes from edge-star-mini-shv-01-lis1.facebook.com (157.240.212.35): icmp_seq=5 ttl=55 time=166 ms
^X64 bytes from edge-star-mini-shv-01-lis1.facebook.com (157.240.212.35): icmp_seq=6 ttl=55 time=150 ms
c64 bytes from edge-star-mini-shv-01-lis1.facebook.com (157.240.212.35): icmp_seq=7 ttl=55 time=148 ms
^C
--- facebook.com ping statistics ---
7 packets transmitted, 7 received, 0% packet loss, time 6014ms
rtt min/avg/max/mdev = 147.388/154.427/167.278/17.007 ms
```

If we check the communication from a host from another computer to our VM, the communication does not work. The physical host and our VM must be on the same server to communicate.

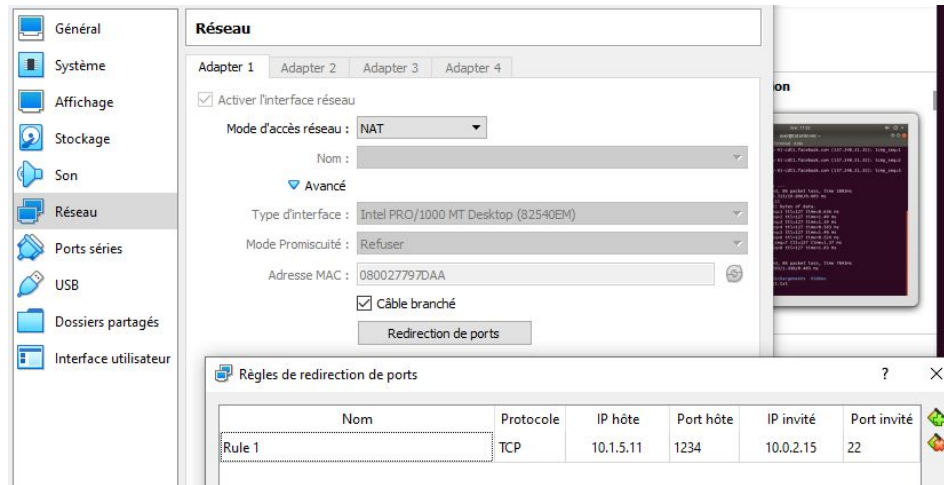
If we check the communication from our Host to our VM we can see that it does not work. The problem is that when the communication is set to NAT, the VM does not have its own IP address within the actual network, it only has the LAN address on the host. To solve this problem, the network adapter on the VM has to be set to a bridged connection.

So if we have to summarize the results, we can see that the communication is only possible from the VM to the exterior, because of the VM's NAT. But every time we try to communicate from the exterior to the VM, that's not possible because the VM does not have its own IP address on the network. And if we have multiple VM, they will all have the same IP address, the host IP address.

To solve this problem we have to connect to the host using ssh (port 1234 by default) and then using ssh again, connect to the VM's port (port 22 by default) where we want to send the message. This solution still uses the NAT network configuration

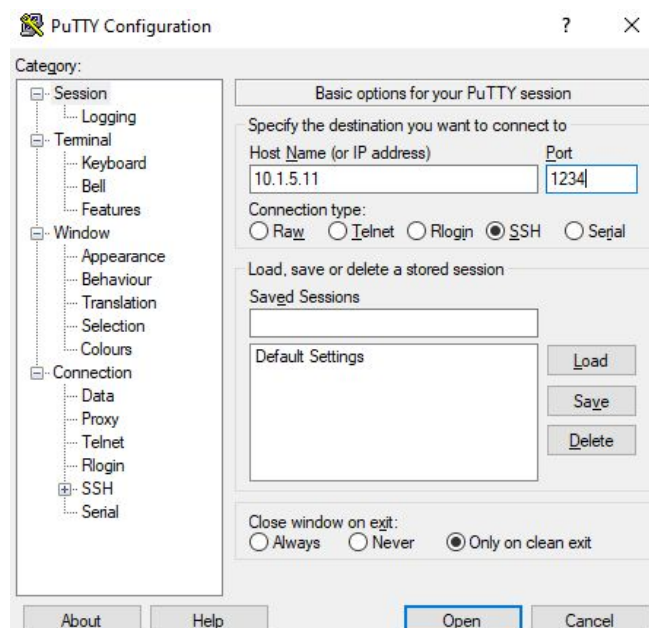
and makes it possible to communicate with the VM from the outside without having its own IP address on the network.

To establish the link we have to configure the VM as follows :



After the port forwarding configuration on the VM's configuration panel, the SSH communication has to be set. We used Putty to establish a communication from the Host to the hosted VM.

We configured as follows :



Then, we use our credential to start the connection, and here is the result :

```
login as: user
user@10.1.5.11's password:
Welcome to Ubuntu 18.04.3 LTS (GNU/Linux 4.15.0-65-generic x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:    https://landscape.canonical.com
 * Support:       https://ubuntu.com/advantage

System information as of Tue Oct  6 11:14:08 CEST 2020

System load:  0.08               Processes:            174
Usage of /:   21.2% of 17.59GB   Users logged in:     1
Memory usage: 75%               IP address for enp0s3: 10.0.2.15
Swap usage:   50%

 * Kubernetes 1.19 is out! Get it in one command with:

    sudo snap install microk8s --channel=1.19 --classic

https://microk8s.io/ has docs and details.

 * Canonical Livepatch is available for installation.
- Reduce system reboots and improve kernel security. Activate at:
  https://ubuntu.com/livepatch

321 paquets peuvent être mis à jour.
251 mises à jour de sécurité.

Nouvelle version « 20.04.1 LTS » disponible.
Lancer « do-release-upgrade » pour mettre à niveau vers celle-ci.

Last login: Fri Oct  4 01:38:33 2019 from 10.0.2.2
user@tutorial-vm:~$
```

We can see the connection is established between the host and the hosted-VM. If we create a text file using Putty, we can see it will be created on the VM, as it's shown below:

```
user@tutorial-vm:~$ touch test.txt
user@tutorial-vm:~$
user@tutorial-vm:~$ ls
Bureau      Images      Musique     Téléchargements  Vidéos
Documents   Modèles     Public      test.txt

user@tutorial-vm:~$ ls
Bureau      Images      Musique     Téléchargements  Vidéos
Documents   Modèles     Public      test.txt
```

Clone the VM

In order to clone the VM we cannot just make a simple copy of it because each copy would have the same identifier. Each VM has to have its own identifier in the hypervisor namespace, otherwise VirtualBox will report an error.

So to clone the VM, we placed ourselves on the following file, containing VBoxManage:

C:\Program Files\Oracle\VirtualBox\VBoxManage.exe in Windows

Next, the following command has to be entered :

```
C:\Program Files\Oracle\VirtualBox>VBoxManage clonemedium "C:\Users\berta\Downloads\disk.vmdk" "C:\Users\berta\Downloads\disk-copy.vmdk"
0%...10%...20%...30%...40%...50%...60%...70%...80%...90%...100%
Clone medium created in format 'VMDK'. UUID: 21cc6481-3126-48a8-9ab3-1832478db0fd
```

And that's it, the VM is cloned, and each VM has its own identifier.

Docker containers provisioning

We start by creating a Docker container, CT1 inside the VM. To do so we use the following linux command :

```
user@tutorial-vm:~$ sudo docker start -i ct1
root@2b818b7eaf89:/# ls
bin    dev    home  lib32  libx32  mnt    proc  run    srv    tmp    var
boot  etc    lib   lib64  media   opt    root /sbin  sys    usr
```

In this command line the *-i* is used to attach the STDIN container. If we want to stop the docker container we'll have to use the following command, *sudo docker stop ct1*. Now that the Docker container is created we can check the connectivity.

To start we need the IP address : IP docker ct1 : 172.17.0.1

```
user@tutorial-vm:~$ ifconfig
docker0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
    inet 172.17.0.1  netmask 255.255.0.0  broadcast 172.17.255.255
    inet6 fe80::42:efff:fe34:cc4b  prefixlen 64  scopeid 0x20<link>
    ether 02:42:ef:34:cc:4b  txqueuelen 0  (Ethernet)
    RX packets 0  bytes 0 (0.0 B)
    RX errors 0  dropped 0  overruns 0  frame 0
    TX packets 15  bytes 1186 (1.1 KB)
    TX errors 0  dropped 0  overruns 0  carrier 0  collisions 0
```

Then we have to test the connection using the *ping* command, but before doing so, the command has to be installed on the docker container:

```
root@2b818b7eaf89:/# apt-get -y update && apt-get -y install net-tools iputils-ping
```

Once every option is installed we proceed with the test :

1- Ping from the docker containers to the exterior. Here we took facebook server as an example:

```
root@2b818b7eaf89:/# ping facebook.com
PING facebook.com (179.60.192.36) 56(84) bytes of data.
64 bytes from edge-star-mini-shv-01-cdg2.facebook.com (179.60.192.36): icmp_seq=
1 ttl=48 time=19.4 ms
64 bytes from edge-star-mini-shv-01-cdg2.facebook.com (179.60.192.36): icmp_seq=
2 ttl=48 time=20.4 ms
64 bytes from edge-star-mini-shv-01-cdg2.facebook.com (179.60.192.36): icmp_seq=
3 ttl=48 time=19.9 ms
```

2- Ping from the docker container to the VM :

```
root@2b818b7eaf89:/# ping 10.0.2.15
PING 10.0.2.15 (10.0.2.15) 56(84) bytes of data.
64 bytes from 10.0.2.15: icmp_seq=1 ttl=64 time=0.037 ms
64 bytes from 10.0.2.15: icmp_seq=2 ttl=64 time=0.081 ms
64 bytes from 10.0.2.15: icmp_seq=3 ttl=64 time=0.082 ms
64 bytes from 10.0.2.15: icmp_seq=4 ttl=64 time=0.081 ms
```

3- Ping from the VM to the docker container :

```
user@tutorial-vm:~$ ping 172.17.0.1
PING 172.17.0.1 (172.17.0.1) 56(84) bytes of data.
64 bytes from 172.17.0.1: icmp_seq=1 ttl=64 time=0.025 ms
64 bytes from 172.17.0.1: icmp_seq=2 ttl=64 time=0.069 ms
64 bytes from 172.17.0.1: icmp_seq=3 ttl=64 time=0.348 ms
```

After the test we can conclude that the docker container has no connectivity issue. We saw that the container is able to communicate with the VM and the VM with the container, and more important, the docker is able to communicate with an exterior server.

If we proceed the tests, we can create a second Docker container, using the following command :

```
user@tutorial-vm:~$ sudo docker run --name ct2 -p 2223:22 -it ubuntu
root@a590c4ad5ac1:/# apt-get -y update && apt install nano
```

After the creation of the second container we can take an image of the CT2 container, in order to have all the parameters and configuration saved on this image.

We can do that by using the following command line, with the containers ID :

```
user@tutorial-vm:~$ sudo docker commit a590c4ad5ac1 photo:01
sha256:7b9ace88b8a122935801bb65e189bhd305f1530fdcf77e0471cd9ae00f8f0287
user@tutorial-vm:~$ sudo docker ps
[sudo] Mot de passe de user :
CONTAINER ID   IMAGE     COMMAND                  CREATED        STATUS
PORTS          NAMES
a590c4ad5ac1   ubuntu   "/bin/bash"             About a minute ago Up About a minute
0.0.0.0:2223->22/tcp   ct2
2b818b7eaf89   ubuntu   "/bin/bash"             21 hours ago   Up 25 minutes
ct1
```

Docker ID: a590c4ad5ac1, obtained using the following command *sudo docker ps*.

Image Name : 01, using the *photo:* parameter.

After the image is created we can stop and erase the last container, with the following command:

```
user@tutorial-vm:~$ sudo docker stop ct2
ct2
user@tutorial-vm:~$ sudo docker rm a590c4ad5ac1
a590c4ad5ac1
```

Now we can use the snapshot previously created using the CT2 container to create a CT3 container. But before that, let's list the different snapshots available on the VM :

```
user@tutorial-vm:~$ sudo docker images
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
photo	01	7b9ace88b8a1	2 minutes ago	97.9MB
<none>	<none>	a1486bfe515f	3 minutes ago	97.9MB
ubuntu	latest	9140108b62dc	11 days ago	72.9MB

To create the CT3 container using the CT2 snapshot :

```
user@tutorial-vm:~$ sudo docker run --name ct3 -it photo:01
```

If we try to launch *nano* software, we observe that everything is just like it was on the CT2 container, and that is logical because we created the third container using the CT2 snapshot.

Every configuration parameter and software initially present on the CT2 container, is also present on the CT3 container.

After checking the specifications of the CT3 container, created using the CT2 snapshot, we can also check how to create a persistent image of a container. To do so, we used the following commands:

```
user@tutorial-vm:~$ touch myDocker.dockerfile
user@tutorial-vm:~$ ls
```

Bureau	Images	Musique	Public	test.txt
Documents	Modèles	myDocker.dockerfile	Téléchargements	Vidéos

This means we are creating a *dockerfile* inside the VM, with the following information:

FROM ubuntu

RUN apt update -y

RUN apt install -y nano

CMD ["/bin/bash"]

And finally we built a docker image on another format and saved it inside the *dockerfile*.

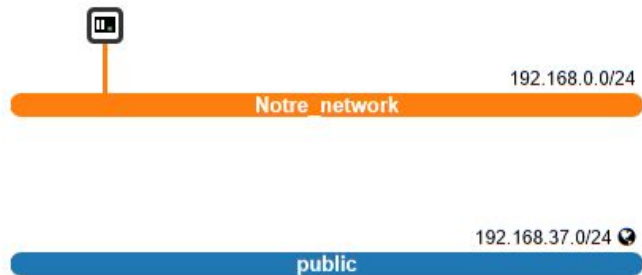
Operate proper VMs provisioning with OpenStack and manage the networking connectivity.

CT creation and configuration on OpenStack

Using Openstack we can create a virtual machine online. To do so, we connected to Openstack using the following link : [OpenStack Web interface](#) and used the following credentials.

INSAT
login
pwd

Before creating the VM online we had to create a network, otherwise we would have created the VM on Openstack's public network (only admin users can do so).



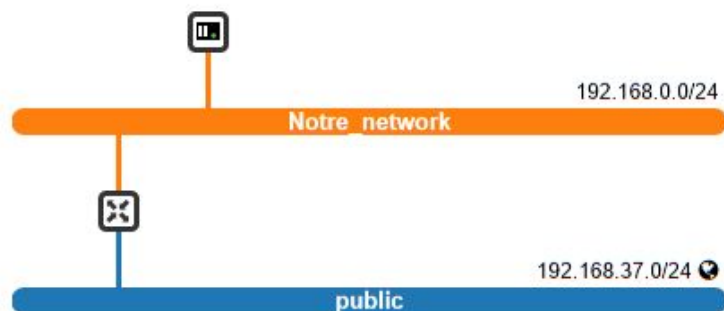
Our own network has the following IP address : 192.168.0.0/24

Inside our network, we created our new VM, with the following information,

Name : VM
Image Name : alpine-mode
Flavor : small
Network : Notre_network

Instance Name	Image Name	IP Address	Flavor	Key Pair	Status	Availability Zone
VM	alpine-node	192.168.0.147	small	-	Active	nova

After the VM is created, the internet access has to be set. To configure the internet access we'll have to connect our network to a Router connecting our network to the public network.



- Create Router (since the public network), and click on the link created.
- Add an Interface (make the connection between our network and the router)

Once the connection is set, we connected to the VM using the following credentials :

Login : root

Password : root

Inside the VM terminal we changed the keyboard to AZERTY:

```
alpine-node:~# setup-keymap fr fr-us
* WARNING: you are stopping a boot service
* Caching service dependencies ...
* Setting keymap ...
```

Then we create another VM, this time with Ubuntu4CLV, because it is easier then alpine_mode for the ssh connection. This VM doesn't have a predefined size because we don't need memory space for the ssh connections, we'll just analyse the communications way of working.



Instance Name	Image Name	IP Address	Flavor	Key Pair	Status	Availability Zone
VM_Sans_Volume	ubuntu4CLV	192.168.0.238	small	-	Active	nova

VM_Sans_Volume : 192.168.0.238

In order to change the keyboard from QWERTY to AZERTY :

```
user@tutorial-vm:~$ setxkbmap us
```

Then we ping from the VM to the host, to verify if it's working, and has shown below, it is.

```
user@tutorial-vm:~$ ping 10.1.1.67
PING 10.1.1.67 (10.1.1.67) 56(84) bytes of data.
64 bytes from 10.1.1.67: icmp_seq=1 ttl=126 time=2.56 ms
64 bytes from 10.1.1.67: icmp_seq=2 ttl=126 time=1.33 ms
64 bytes from 10.1.1.67: icmp_seq=3 ttl=126 time=1.26 ms
```

When we created a private network on OpenStack, it basically created the equivalent to a sub-network of the INSA's network, because our host is on INSA's network, that's why the connection works.

Then, if we try to ping from the host to the VM, it does not work, as shown below :

```
U:\>ping 192.168.0.238

Envoi d'une requête 'Ping' 192.168.0.238 avec 32 octets de données :
Délai d'attente de la demande dépassé.
Délai d'attente de la demande dépassé.
Délai d'attente de la demande dépassé.
Délai d'attente de la demande dépassé.

Statistiques Ping pour 192.168.0.238:
    Paquets : envoyés = 4, reçus = 0, perdus = 4 (perte 100%),
```

The communication does not work because Openstack gives our private-network, a fake IP address, and the VM cannot be easily recognized on the network.

In order to solve this issue we have to create a link between our private network IP address and the fake IP address given to the VM. By doing so, every message intended for our VM will be relayed by our private network to the VM (Same concept that what we did earlier with the VirtualBox VM).

We have to establish a link between the IP address on the public network of our private network and the fake IP address of the VM. This way, the messages received by our private network will arrive at the VM. To do so, we'll use floating IP addresses.

We create the floating address and associated it to our VM :

IP Address	Description	DNS Name	DNS Domain	Mapped Fixed IP Address
192.168.37.96				VM_Sans_Volume 192.168.0.238

We ping the VM from the host using the floating IP address 192.168.37.96, and it works:

```
U:\>ping 192.168.37.96

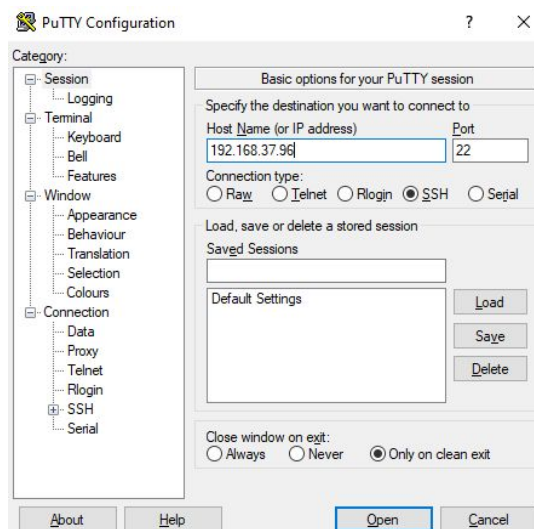
Envoi d'une requête 'Ping' 192.168.37.96 avec 32 octets de données :
Réponse de 192.168.37.96 : octets=32 temps=2 ms TTL=62
Réponse de 192.168.37.96 : octets=32 temps=1 ms TTL=62
Réponse de 192.168.37.96 : octets=32 temps=1 ms TTL=62
Réponse de 192.168.37.96 : octets=32 temps=1 ms TTL=62

Statistiques Ping pour 192.168.37.96:
    Paquets : envoyés = 4, reçus = 4, perdus = 0 (perte 0%),
Durée approximative des boucles en millisecondes :
    Minimum = 1ms, Maximum = 2ms, Moyenne = 1ms
```

Then, it's time to check the SSH communication from the host to the VM with putty command. By default our VMs port is 22, and it has the IP address on INSAs network (192.168.37.96). In the other part of this practical exercise we had to use the 1234 port with VirtualBox because we used the port forwarding technique and our VM did not have the fake IP address.

From the terminal :

```
U:\>putty
```



Connection credentials :

Login : user

Password : user

```
login as: user
user@192.168.37.96's password:
Welcome to Ubuntu 18.04.3 LTS (GNU/Linux 4.15.0-65-generic x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:    https://landscape.canonical.com
 * Support:       https://ubuntu.com/advantage

System information as of Fri Oct  9 13:44:47 CEST 2020

System load:  0.17               Processes:    165
Usage of /:   21.5% of 17.59GB   Users logged in:  1
Memory usage: 21%               IP address for ens3: 192.168.0.238
Swap usage:   0%

 * Kubernetes 1.19 is out! Get it in one command with:

    sudo snap install microk8s --channel=1.19 --classic

https://microk8s.io/ has docs and details.

 * Canonical Livepatch is available for installation.
   - Reduce system reboots and improve kernel security. Activate at:
     https://ubuntu.com/livepatch

318 paquets peuvent être mis à jour.
248 mises à jour de sécurité.

Nouvelle version « 20.04.1 LTS » disponible.
Lancer « do-release-upgrade » pour mettre à niveau vers celle-ci.

Last login: Fri Oct  9 13:41:13 2020 from 192.168.37.96
user@tutorial-vm:~$ touch text.txt
user@tutorial-vm:~$
```

It works, so we can communicate in SSH from the host to the VM.

We can see that from the host, we managed to create the text.txt file on the VM.

```
user@tutorial-vm:~$ ls
Bureau      Images      Musique     Téléchargements  Vidéos
Documents   Modèles     Public      text.txt
```

Snapshot, restore and resize a VM

If we try to resize the VM and reduce its size, this wouldn't work because the data would be lost and so it's protected to prevent that. What we can do is to add more size and so pass it to *medium size*.

<input type="checkbox"/>	Instance Name	Image Name	IP Address	Flavor	Key Pair	Status	Availability Zone
<input type="checkbox"/>	VM_Sans_Volume	ubuntu4CLV	192.168.0.238, 192.168.37.96	medium	-	Active	nova

We can resize the VM either when it is running or not.

“The 2 previous operations highlight the flexibility and the agility that could be provided thanks to the virtualization setting as it was previously explained during the lectures.”

With Openstack we can easily reconfigure the VM, the same is not true with VirtualBox. The only limit is to resize it with an inferior size to its original.

For the rest of this lab, we resized our VM back to small and took a snapshot of the VM.

OpenStack API to automate the operations described in objectives 4, 5, 6 and 7. Implement a Web 2-tier application.

Openstack client installation

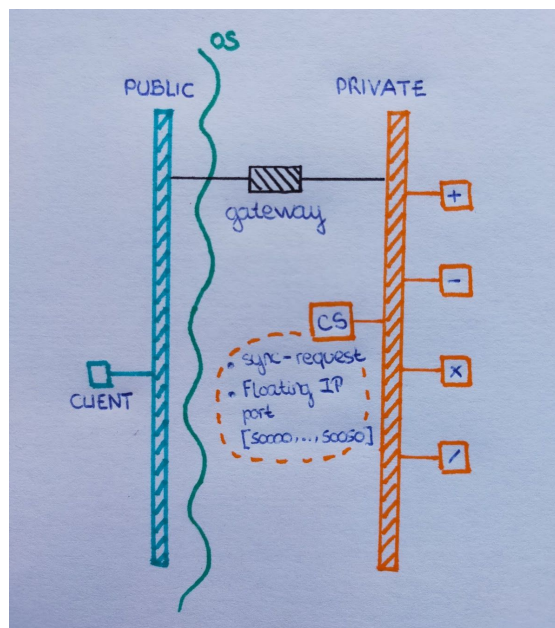
As mentioned on the lab file, we installed the OpenStack Client on the VM, and configured it with the rc.sh file downloaded on OpenStack.

Web-2-tier application topology and specification

In this part, in order to use the various services we have to install NodeJs, Npm and CURL.

The command we used to download the different services was wget <http://homepages...>

So to make this work we have to create a new VM per web service on OpenStack.



The client will communicate with the CS, he will send every calculating operation, and the CS will call the different web services when needed.

Before everything work as it should, we have to change the CS script:

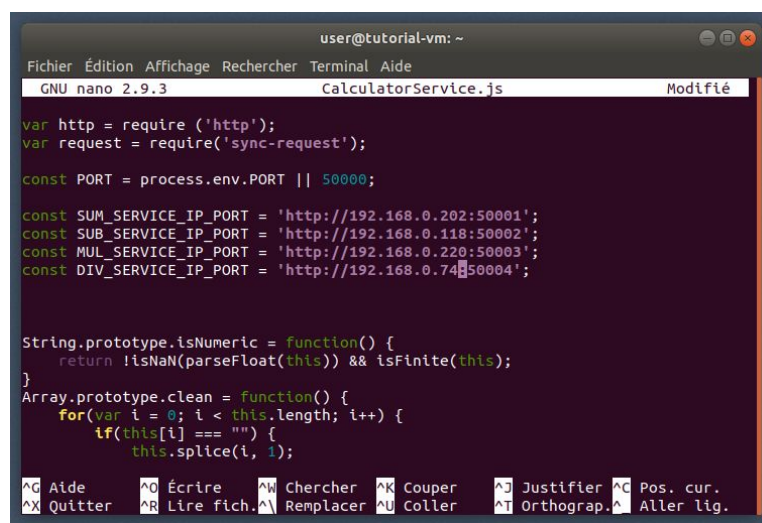
Change every IP address from each VM containing the different web services

Change the port, from 80 to a port between 50 000 and 50 050, because it's the only port accepted by the CSN. If we use other ports, the gateway (controlled by the CSN) will block the communication.

The floating IP address from the CS VM has also to be changed.

In addition to that we have to add a rule in order for the port 50000 to be activated.

After all the changes, the files looks like the image below :



```

user@tutorial-vm: ~
Fichier Édition Affichage Rechercher Terminal Aide
GNU nano 2.9.3 CalculatorService.js Modifié

var http = require('http');
var request = require('sync-request');

const PORT = process.env.PORT || 50000;

const SUM_SERVICE_IP_PORT = 'http://192.168.0.202:50001';
const SUB_SERVICE_IP_PORT = 'http://192.168.0.118:50002';
const MUL_SERVICE_IP_PORT = 'http://192.168.0.220:50003';
const DIV_SERVICE_IP_PORT = 'http://192.168.0.74:50004';

String.prototype.isNumeric = function() {
    return !isNaN(parseFloat(this)) && isFinite(this);
}
Array.prototype.clean = function() {
    for(var i = 0; i < this.length; i++) {
        if(this[i] === "") {
            this.splice(i, 1);
        }
    }
}

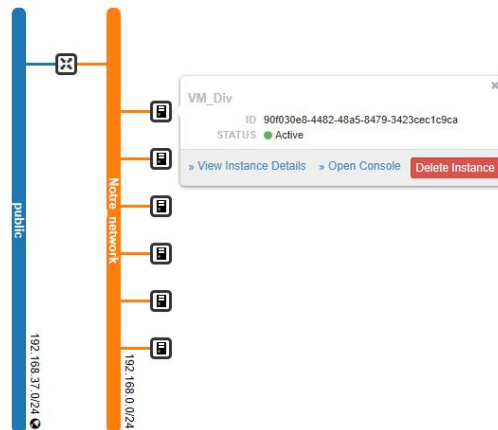
^G Aide      ^O Écrire   ^W Chercher ^K Couper   ^J Justifier ^C Pos. cur.
^X Quitter   ^R Lire fich.^M Remplacer ^U Coller   ^T Orthograp.^_ Aller lig.
  
```

Deploy the Calculator application on OpenStack

After the configuration part, we created an independent VM to each service, as shown below :

<input type="checkbox"/>	Instance Name	Image Name	IP Address	Flavor	Key Pair	Status	Availability Zone
<input type="checkbox"/>	VM_Div	ubuntu4CLV	192.168.0.74	small	-	Active	nova
<input type="checkbox"/>	VM_Mul	ubuntu4CLV	192.168.0.220	small	-	Active	nova
<input type="checkbox"/>	VM_Sub	ubuntu4CLV	192.168.0.118	small	-	Active	nova
<input type="checkbox"/>	VM_Sum	ubuntu4CLV	192.168.0.202	small	-	Active	nova
<input type="checkbox"/>	VM_Sans_Volume	ubuntu4CLV	192.168.0.238, 192.168.37.96	large	-	Active	nova

Each VM, for the different services has to be on the same network if we want our system to work properly. After the creation of each VM, our network looks like the image below :



On the main VM (VM_sans_volume), we have to do a sync-request if we want our VM to be able to communicate with the exterior.

```
user@tutorial-vm:~$ npm install sync-request
extract:readable-stream
```

After every configuration step, we tested the system as shown below:

For the first test, we tried the sum service and we placed ourselves on the different ports to ensure that every information sent is the correct one. As shown below, the test was successful.

```
user@tutorial-vm:~/Téléchargements$ curl -d "2+3" -X POST http://192.168.37.96:50000
result = 5
```

```
user@tutorial-vm:~$ node SumService.js

Listening on port : 50001
New request :
A = 2
B = 3
A + B = 5
```

```
user@tutorial-vm:~$ node CalculatorService.js
Listening on port : 50000
New request :
2+3 = 5
```

The last test we did was the multiplication because we wanted to test another service available.

```
user@tutorial-vm:~/Téléchargements$ curl -d "2*3" -X POST http://192.168.37.96:50000
result = 6
```

For the fourth part of this objective we had to implement the same logic but using docker containers instead of 5 VM.

Because we ran out of time, we couldn't implement this system based on docker containers, but theoretically the system would remain the same. Everything would be managed by a single docker container, the "calculator" one. This container would then ask the other containers to do the math operation depending on which command was sent to it.

For the remaining objectives (10 and 11) we did not have the time to finish them.

Conclusion

With this lab we first started by understanding the various concepts and tools around cloud computing. Then we had the opportunity to put it in use, and see in which case the VM technology is more efficient than the docker containers.

This lab, and the multiple objectives, has allowed us to test different applications for a virtual machine, see when the connection was available and what solution we had to implement in order to solve the connection issues. But has also allowed us to use the docker containers, providing us with another solution, better suited for some applications than the VMs.