Lab 1 : Introduction to Cloud hypervisors

Goalard Rémi / Payet Alexis

Link to access the lab : tiny.cc/TP_CloudComputing

Objective of the lab : Acquire knowledge about the concept and techniques of virtualization. Discover various types of virtualization technologies, such as VM and containers, and test functionalities provided by Cloud providers as IaaS (Infrastructures as a Service).

Part 1 : theoretical knowledge

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

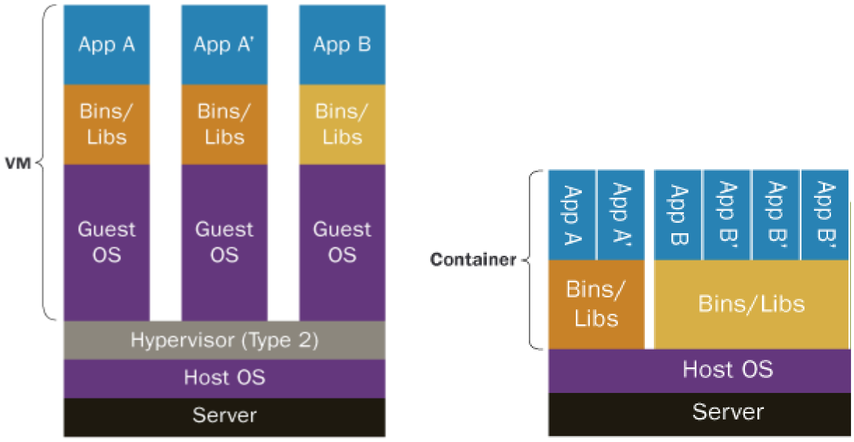


Figure 1 : VM vs CT

The two figures above represent the main virtualisation technologies : Virtual Machines on the left and Containers on the right.

Before elaborating on those figures, it's important to explain the concept of "virtualisation" : the process of creating one or several virtual versions of a piece of computer equipment or software.

The main difference between the two figures, i.e between VM and containers technologies is based on the fact that the first one virtualizes the entire machine (from the hardware), while the second one only virtualizes the software layers above the OS. It means that each VM created has its own guest OS, while all the containers share the same OS : the one from the host machine.
Furthermore, the VM is based on an Hypervisor (hosted in this case), which separates and distributes the resources of the host machine (RAM, processor, etc.) to all the created VM's, and attributes them a guest OS.

Comparison table of the two virtualizations technologies :

	Software developer		Infrastructure administrator	
Technology	VM	CT	VM	CT
Virtualisation cost	- Virtualize everything (OS included) => heavier. It also virtualize the hardware components; => more cost	- Only virtualize above the OS => lighter	- Virtualize everything (OS included) => heavier but they can emulate any OS they want - More performant with big data flow	- Only virtualize above the OS => lighter
CPU - memory - network use for an App	- For a given app, all the hardware is virtualized so it cost much more CPU and memory use => less apps can run	- Only virtualize the libraries => lighter => can run much more apps on any container	- For a given app, all the hardware is virtualized so it cost much more CPU and memory use	- Only virtualize the libraries => lighter => can run much more apps on any container
Security for the application	- Each VM has its own OS => no link between the apps of various VM (Isolation) => interesting as an application developer	- All containers share the same OS (host one) => any breach on an app could lead to the corruption of another app through the shared OS	- Each VM has its own OS => no link between the apps of various VM => good isolation	- All containers share the same OS (host one) => app are not really secured => not a real deal breaker as an infrastructure administrator
Performance (response time)	- Slower access to physical resources for the app, because of more 'layers' (Hypervisor + 2 OS)	- Faster access to physical resources for the app, only the host OS to go through	- Slower access to physical resources for the app, but not an app developer	- Faster access to physical resources for the apps, but still not an app dev
Tooling for continuous integration support	- Big and heavy VM images => need lot of time and resources - Easier backup	- Lot of microservices => very useful tools as software developer - Modular and high portability	- Big and heavy VM images => go with powerful softwares useful as infrastructure administrator - More configuration possibilities	- Lot of microservices => very hard, complexe and long to administrate

Finally, it seems that VM are way more dedicated to infrastructure administration, while Containers are well designed for application development. Although they present a lot of benefits in the case of app development, it's important to unlight the fact that containers present some risks in terms of application security.

2. **Similarities and differences between the CT types**

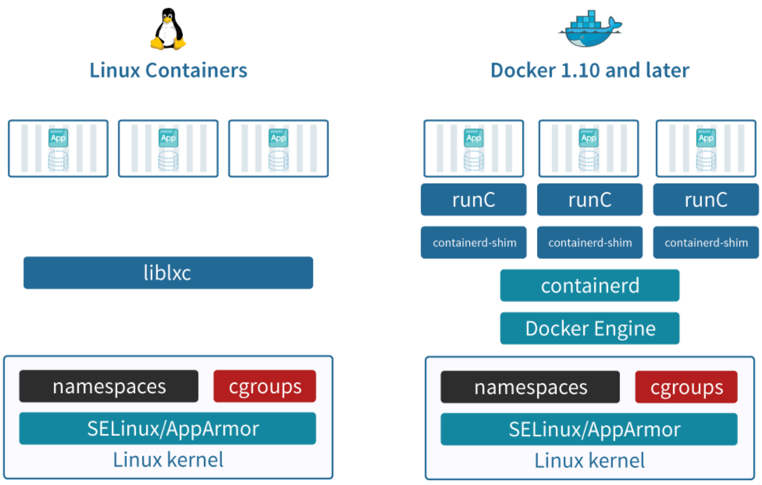


Figure 2 : Linux Lxc vs Docker

There are various container solutions available on the market, with LCX Linux and Docker as two of the most used. Now we know how containers virtualize machines, it's interesting to compare the difference between some CT technologies. In this case, we will focus on LCX Linux and Docker.

From the figure above we can see some difference between both technologies : while on LCX all the running containers have to share the same library 'liblxc', on Docker each instance of a container has access to its own libraries. This implicates more isolation between the apps running on each container, which is possible thanks to the "Docker Engine" module.

Comparison table of the two container solutions :

Critère	Linux LCX	Docker
Application isolation	- All CT share the same library => less application isolation	- Each CT has its own library => app don't share anything outside their CT => More app isolation
Containerization level	- Containerization at the OS level. Each container runs a separate Linux operating system with its own kernel, user space, and system resources.	- Containerization from the application level. Each container shares the host operating system kernel and runs its own isolated user space
Portability	- LCX runs a standard OS => applications easy to carry from one CT to another on the same machine - Not easy to carry to an other solution like Docker - Might need some OS configurations to be carried from one machine to another (not very good for cloud operations)	- High portability to other environments / clouds without code modifications because of it's isolation at the application level - Can be carried to any OS solution if it has Docker installed on it
Security	- Bad app isolation => security breaches - Depend on Linux security => powerful tools and protocols	- Containerization from UserSpace level => better app isolation than on LCX - Run as root => sensible to malwares
Scalability	- Light and fast to run containers - CT images heavier than Docker ones => not so good for scalability	- More scalable than LCX because 'Docker compose' allows multi-container applications - Ready to use containers, no need to configure the OS
Tooling	- Some tools existing, but more minimalistic than Docker one	- Large ecosystem of tooling, including solutions to carry containers from a cloud solution to another, or solution for multi-container applications - Enables the use of 'version' such as git does - Lot of tools for CT management => useful

3. **Similarities and differences between type 1 & type 2 hypervisor's architecture**

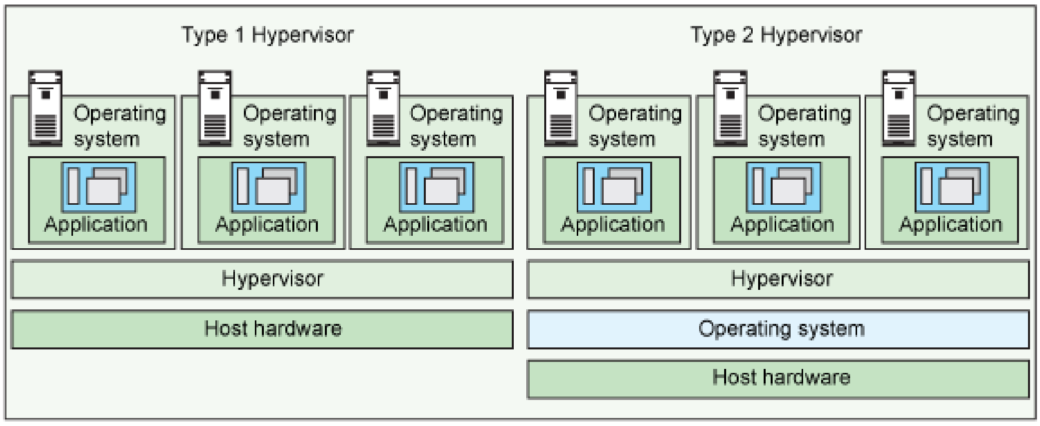


Figure 3 : Bare metal vs hosted hypervisors

Reminder : an hypervisor is a software used to create and manage VMs on a physical machine. It allows one host computer to support multiple guest VMs by virtually sharing its resources, such as memory and processing.

There are actually two kinds of hypervisor that can run on a physical machine :

- Type 1 : "Bare metal"
  - Runs directly on the host hardware => faster access to the resources than type 2 => faster and perform better
  - Isolated from the attack-prone operating system => more secure (all VMs have their own OS)
- Type 2 : "Hosted"
  - Runs like a software on the host operating system (OS)
  - Share the OS on which is build the hypervisor => security breach for the VMs
  - More layers to pass to access to the hardware => slower, less performant and less scalable than bare metal hypervisors.

Identification of the hypervisors type of two VMs technologies :

- VirtualBox : Type 2 or "hosted"
- OpenStack : is based on 'Nova' that supports various hypervisors. Most used is KVM (type2/hosted) but Xen or Hyper-V can be used (type1/bare metal).

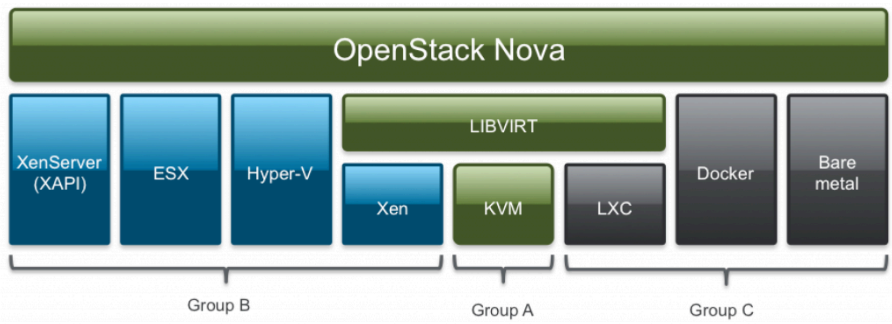


Figure 4 : OpenStack Nova hypervisors supportability

Part 2 : practical part

Objective : In this part, we will focus on the creation of our first VM/CT using VirtualBox and Docker softwares. This will lead us to discover various features offered by those softwares, such as the creation of images, but also a better understanding of the connectivity of virtualized entities.

4. **Creating a VirtualBox VM (NAT mode) & setting up the network to enable 2-way communication with the outside**

In this part, we use VirtualBox and its type 2 hypervisor (hosted) to create a VM using Linux OS (**Annexe 1**), in NAT mode :

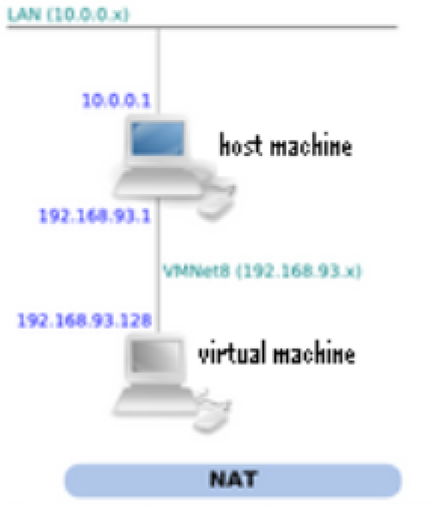


Figure 5 : NAT mode networking

Now that our VM is running, we want to do some connectivity tests, i.e looking to see if the VM is able to communicate with the host machine, and others on the same network.

First, we need our VM and host machine IP addresses :

Machines	Host	VM
IP address	10.1.5.28	10.0.2.15

Both machines have private IP addresses.

Now we will try the VM connectivity, using the “ping“ command in various configurations :

- From the VM to the host (**Annexe 2**)
- From the host to the VM (**Annexe 3**)
- From another computer to the VM (**Annexe 4**)

Configuration	VM => host	Host => VM	Other host => VM
Ping test success	YES	NO	NO

Why is the VM able to communicate with the outside, while no other machine can communicate with it?
As we can see in the figure below, when a VM is launched via VirtualBox on a host machine, the software provides an IP address to the VM, but this address is not a real IP : we could create 1000 VM on the same host with the same IP address.
It’s actually the VirtualBox software that receives ‘ping’ messages from its VMs and forwards them to the host (via the NAT) or other machine outside. But VirtualBox does not allow you to send a message directly to one of the VMs from the outside, because the IP addresses that are assigned to the VMs are not routable.

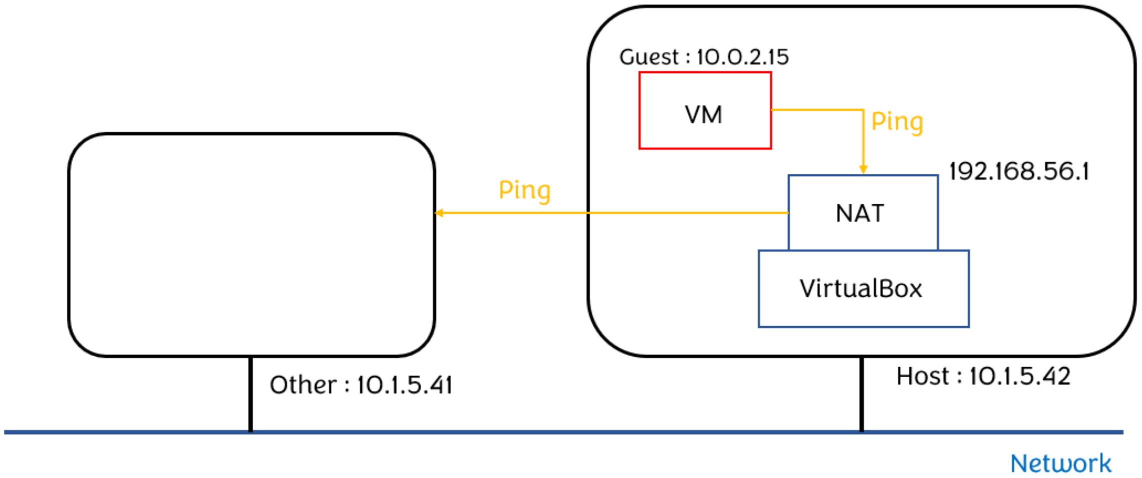


Figure 6 : Communication VM => outside through VirtualBox’s NAT

In order to solve this problem, we can implement a “Port Forwarding” technique (**Annexes 5 - 7**) : for a dedicated application (i.e a dedicated port on the VM), VirtualBox will look at all the messages arriving on a specific port of the host machine, and forward them to the associated port on the VM.
Here, we wanted to access the port 22 of the VM, the one responsible for the ‘ssh’ application.

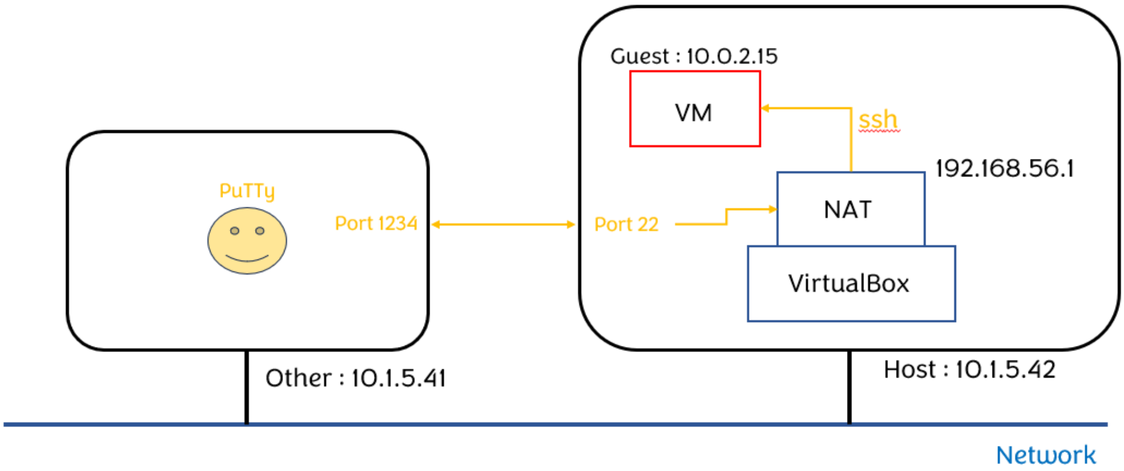


Figure 7 : Port 22 "port forwarding" => Enables 'ssh' on the VM

Last, we will create an image of our VM (**Annexe 8**), and create a new one from the settings of the first one (**Annexe 9**).

5. Docker containers provisioning

After creating a VM, it's now the time to create containers using Docker. To do this, we will run Docker on the VirtualBox VM, because Docker runs as root, and we can't do it directly on INSA's computers.

First we install Docker Engine on the VM (**Annexes 10-14**). Then we create a Docker container instance (**Annexe 16**) using an ubuntu image (**Annexe 15**).

With our first Docker container created, we want to do some connectivity tests :

Machine	Host	Docker Container (Annexe 18)
IP address	10.1.5.28	172.17.0.2

Configuration	CT => Host	Host => CT	Other host => CT
Ping test success	YES	YES	YES

On Docker we are not working in NAT mode anymore, but using **Bridge mode** instead. Docker creates a VLAN on which its containers are deployed, and provides them IP addresses + DNS function : it leads to bidirectional communication between the VM and Docker's containers.

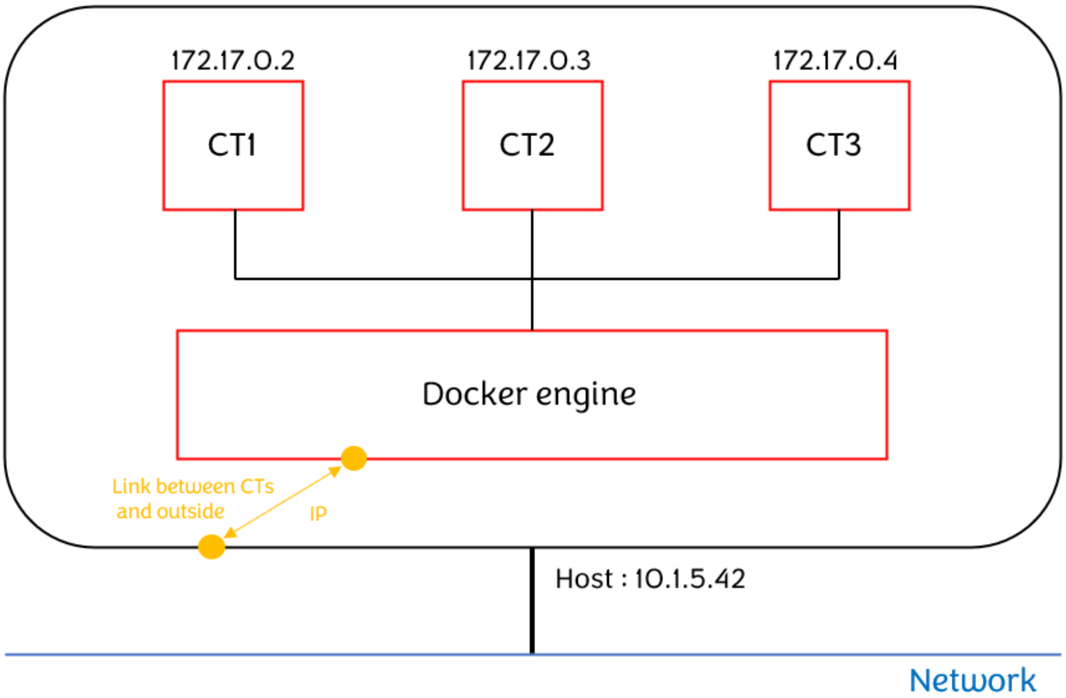


Figure 8 : Communication containers ⇔ outside through Docker's bridge mode

Like for VM, containers can be created from images, in order to access them with softwares and data already prepared to use. We will create a new container instance (**Annexe 19**) with a package on it (**Annexe 20**), and use two different ways to make an image of the instance :

- Using Docker commands to make a snapshot (**Annexe 21**) (**Annexe 22**)
- Using a Dockerfile containing all the informations about the data & software to start the container with (**Annexe 23**)

In both cases, the package 'nano' installed on the container which was taken as an image is still available on the container created from the image => that's the purpose of the image.

Part 3 : OpenStack

Objective : In this part we will focus on the use of OpenStack, a set of frameworks used to create and manage a VM infrastructure => cloud management. We will see how to create a VM and various operations existing to manage those VM

6. Using OpenStack to create and manage VM's and their networking connectivity

Before trying to manage a cloud environment, we have to learn how to create a VM on an OpenStack environment. We try to create a VM on OpenStack console using the following steps :

- VM name
- "No volume"
- Image : "ubuntu4CLV"
- Flavor : "small2"

We end up encountering an error (**Annexe 24**) because OpenStack tries to connect the VM to the internet, but we are not allowed to use a public address from the INSA network to do it. To solve this issue, we create a private network linked to the public network with a router (**gateway**), and create our VM on this new private network.

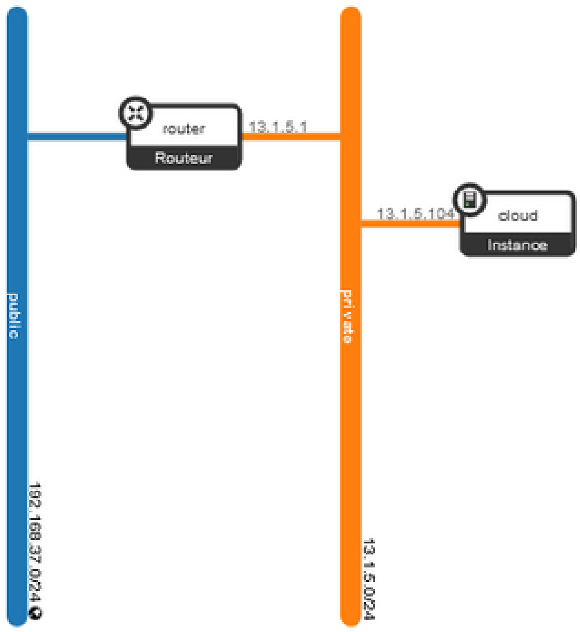


Figure 9 : First VM network topology (figure from another group ⇔ different addresses for our network/VM)

Now we have created a VM, we want to do some connectivity tests. By default, OpenStack blocks network traffic on the virtualized private network, so we need to modify/create specific **security rules** on the console (**Annexe 25**).
We add two rules to allow the required traffic :

- ICMP (ingress) => to receive ping
- SSH

Once the security rules are set up, we can do the connectivity tests :

Configuration	VM => Host	Host => VM	VM => outside
Ping test success	YES	NO	YES

Pinging the host or the outside world from the VM works (**Annexe 26**), but the VM can't be pinged (**Annexe 27**) even if the security rule allows it. It is due to the fact that our router (gateway) is set up for NAT, so it is possible to access the public network from the VM but not the other side (like lab 1).
To solve this issue, we create a floating IP address for the VM, so the host machine and others can use 'ping' or 'SSH' commands on the VM.

IP type	VM IP	Floating IP
IP address	172.16.10.81	172.17.0.2

Connectivity test using the floating IP address (**Annexe 28-29**) :

Configuration	VM => Host (ping)	Host => VM (ping)	Host => VM (SSH)
Connectivity success	YES	YES	YES

7. **Test management operations available for VM on OpenStack framework**

To explore the operations given by OpenStack to manage VM we will try 4 different manipulations :

- **Resize VM while running** => not possible because of INSA's restrictions. It is possible on our own but it is a dangerous operation (cost a lot).
- **Resize VM while shut down** => Same answer as previous.
- **VM's snapshot** => the created image is using a different software/hardware than the original VM. Also the image is a bit heavier than its original (**Annexe 30**).
- **Restore VM from backup (image)** => not possible to modify an existing VM to make it look like an image, but you can use the image when creating a new VM.

8. **Using OpenStack VMs to deploy a calculator web service**

With the knowledge on how to create an manage VM, we now can use them to deploy a web service : we will create various VM and deploy on each of them a web service corresponding to one calculator application : [+ , - , * , /]

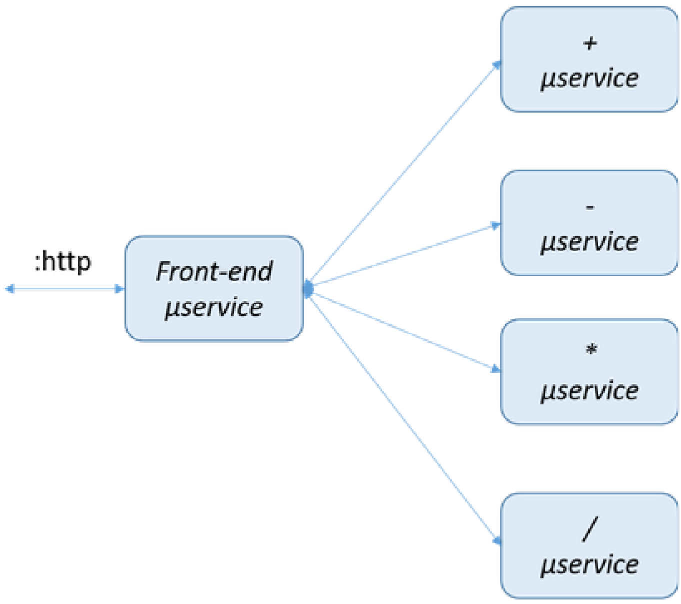


Figure 10 : Web Service calculator deployment using OpenStack

To do that, we will use the OpenStack client instead of the web console. We can't install the client on INSA's machine, so we use a VM from VirtualBox to be our 'host machine'. We install the client on it (**Annexe 31 - 33**).
Now the client is ready, we will install some useful packages on the VMs (**Annexe 34**).

We create one VM for each microservice with the previous packages on it (**Annexe 35**) and add HTTP rules (**Annexe 36**) + floating IP address (**Annexe 37**), so calculator services can be accessed from outside machines.

We run each of the services on the corresponding VM, and open a terminal on the local host to check if the Web Service is correctly deployed (**Annexe 38 - 39**) => the output is the intended one.

9. Automating of VM management using OpenStack API

We created a very little application, but it was already boring and fastidious to configure and launch all the microservices by hand. To avoid this repetitive task, we can use a Docker client such as NodeJS to automatically deploy the nodes of the service.

First step, we create a Dockerfile to automatically configure each VM when created (**Annexe 40**). Each web service particular operation (URL of the microservice) is given as an argument.

To deploy the calculator web service, the next 2 steps have to be followed :

- Build each VM from an image (**Annexe 41**)
- Run image (**Annexe 42**)

Once the service is deployed, we check if it's working, and yes it is (**Annexe 38 - 39**).

Part 4 : Targeted network topology

Objective : in the last part, we will once again deploy our service following some client requirements. Indeed we have to host the VMs on 2 IP sub-networks according to the following schema :

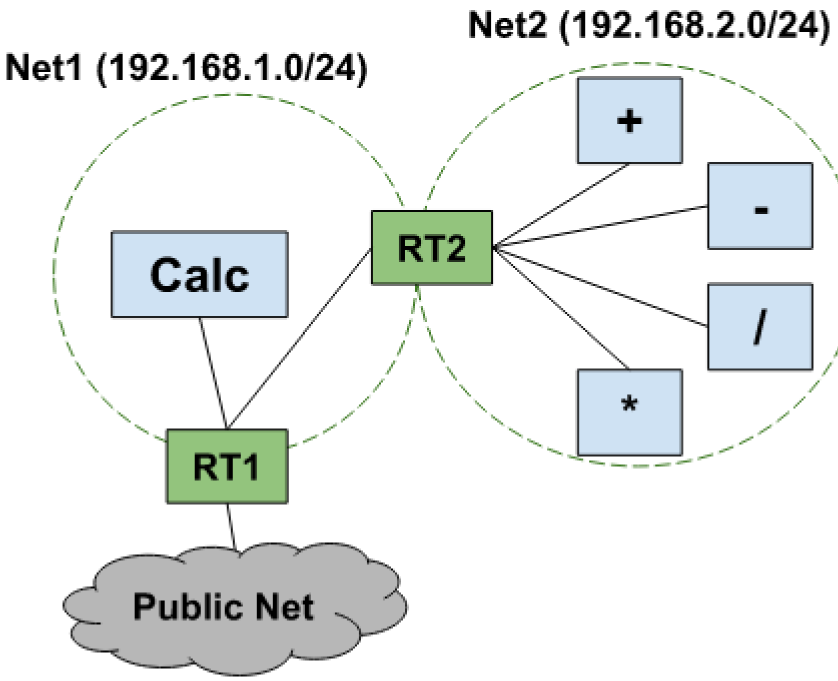


Figure 11 : Targeted network topology

We use a script to create the wanted network / subnetworks, web services and routers (gateways) between the subnetworks and the public net (**Annexe 43**).

Lab 2 : Orchestrating services in hybrid cloud/edge environment

Objective of the lab : the objective of this second lab is to understand the management of services in an hybrid cloud environment => link between cloud and edge computing.

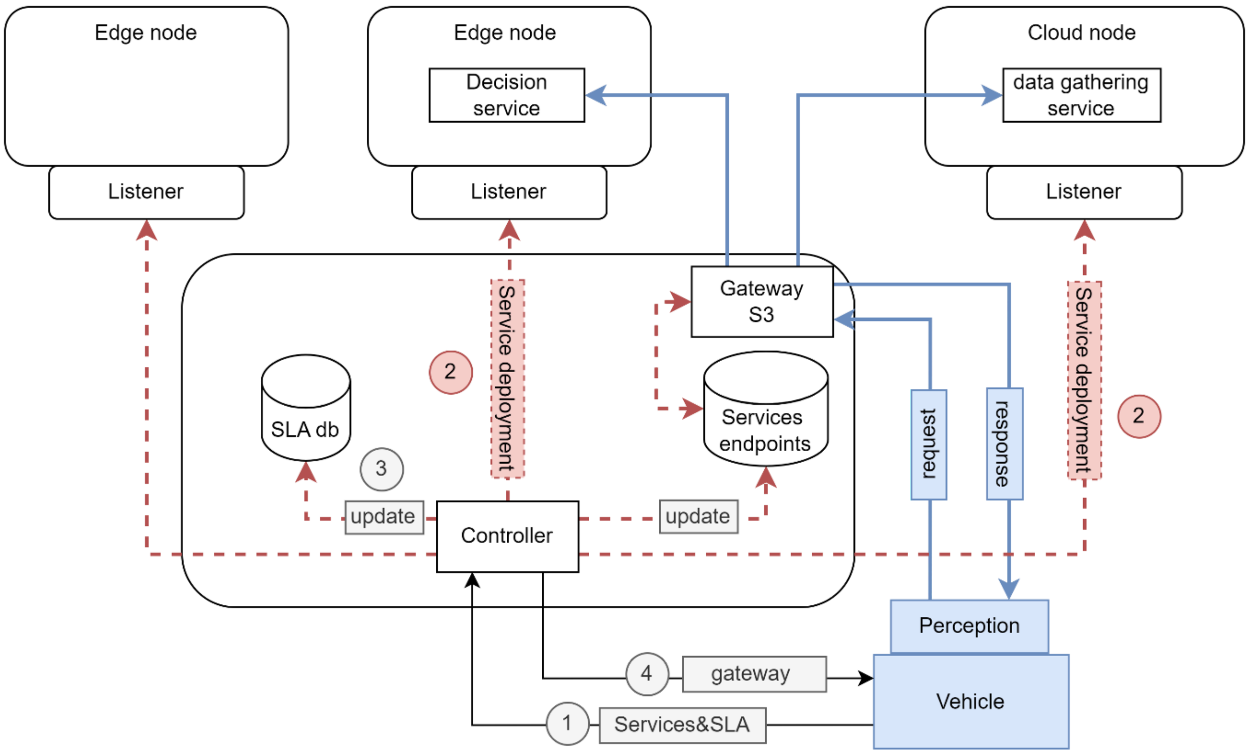


Figure 12 : Hybrid cloud/edge environment

We first begin by creating and configuring the master and 2 workers :

Affichage de 3 éléments

	Instance Name	Image Name	IP Address	Flavor	Key Pair	Status	Availability Zone	Task	Power State	Age	Actions
☐	worker_node_2	ubuntu_20	172.16.10.140, 192.168.37.65	small2	-	Active	nova	Aucun	En fonctionnement	2 minutes	Créer un instantané
☐	worker_node_1	ubuntu_20	172.16.10.176, 192.168.37.50	small2	-	Active	nova	Aucun	En fonctionnement	3 minutes	Créer un instantané
☐	master-node	ubuntu_20	172.16.10.77, 192.168.37.37	medium	-	Active	nova	Aucun	En fonctionnement	5 minutes	Créer un instantané

View of the 3 nodes :

```
user@master:~$ kubectl get nodes -o wide
NAME      STATUS   ROLES           AGE   VERSION   INTERNAL-IP     EXTERNAL-IP   OS-IMAGE             KERNEL-VERSION      CONTAINER-RUNTIME
master    Ready    control-plane   12m   v1.27.6   172.16.10.77    <none>        Ubuntu 20.04.6 LTS   5.4.0-163-generic   containerd://1.7.2
worker1   Ready    <none>          10m   v1.27.6   172.16.10.176   <none>        Ubuntu 20.04.6 LTS   5.4.0-163-generic   containerd://1.7.2
worker2   Ready    <none>          10m   v1.27.6   172.16.10.140   <none>        Ubuntu 20.04.6 LTS   5.4.0-163-generic   containerd://1.7.2
user@master:~$
```

The following figure corresponds to the topology we want to implement between our master and the 2 workers.

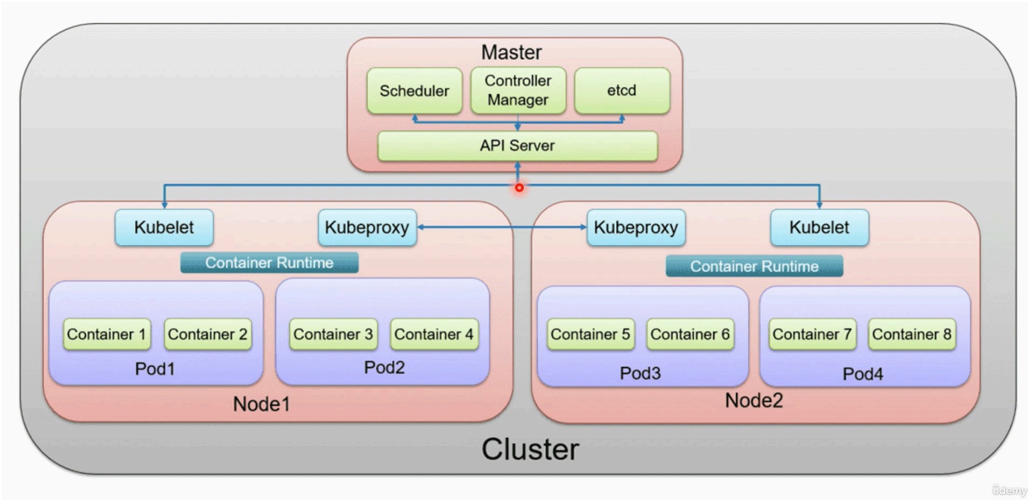


Figure 13 : Master - Workers relationship

We run the command $ kubectl apply -f ./ClusterIP and see that we now have 3 pods with a 'running' status related to worker1

```
NAME                           READY   STATUS    RESTARTS   AGE   IP               NODE      NOMINATED NODE   READINESS GATES
fastapi-app-5476944694-js8ss   1/1     Running   0          10m   172.18.235.129   worker1   <none>           <none>
fastapi-app-5476944694-v4xtf   1/1     Running   0          10m   172.18.235.131   worker1   <none>           <none>
fastapi-app-5476944694-wxnc4   1/1     Running   0          10m   172.18.235.130   worker1   <none>           <none>
```

```
root@master:~/ClusterIP#  kubectl get services -o wide
NAME                          TYPE        CLUSTER-IP     EXTERNAL-IP   PORT(S)   AGE   SELECTOR
fastapi-app-clusterip-service ClusterIP   10.96.218.56   <none>        80/TCP    11m   app=fastapi-app
kubernetes                    ClusterIP   10.96.0.1      <none>        443/TCP   16m   <none>
root@master:~/ClusterIP# _
```

Using the command kubectl describe services to have more informations about the 'node' (Port 5000/TCP) :

```
root@master:~/ClusterIP# kubectl describe services
fastapi-app-clusterip-service
Name:              fastapi-app-clusterip-service
Namespace:         default
Labels:            <none>
Annotations:       <none>
Selector:          app=fastapi-app
Type:              ClusterIP
IP Family Policy:   SingleStack
IP Families:       IPv4
```

```
IP:                 10.96.218.56
IPs:                10.96.218.56
Port:               <unset>  80/TCP
TargetPort:         5000/TCP
Endpoints:
172.18.235.129:5000,172.18.235.130:5000,172.18.235.131:5000
Session Affinity:  None
Events:             <none>
```

Finally we can curl the service using the node port (5000) on worker's 1 IP address where the service is running => ok

```
root@master:~/ClusterIP# curl 172.18.235.129:5000
{"hello":"world"}
```

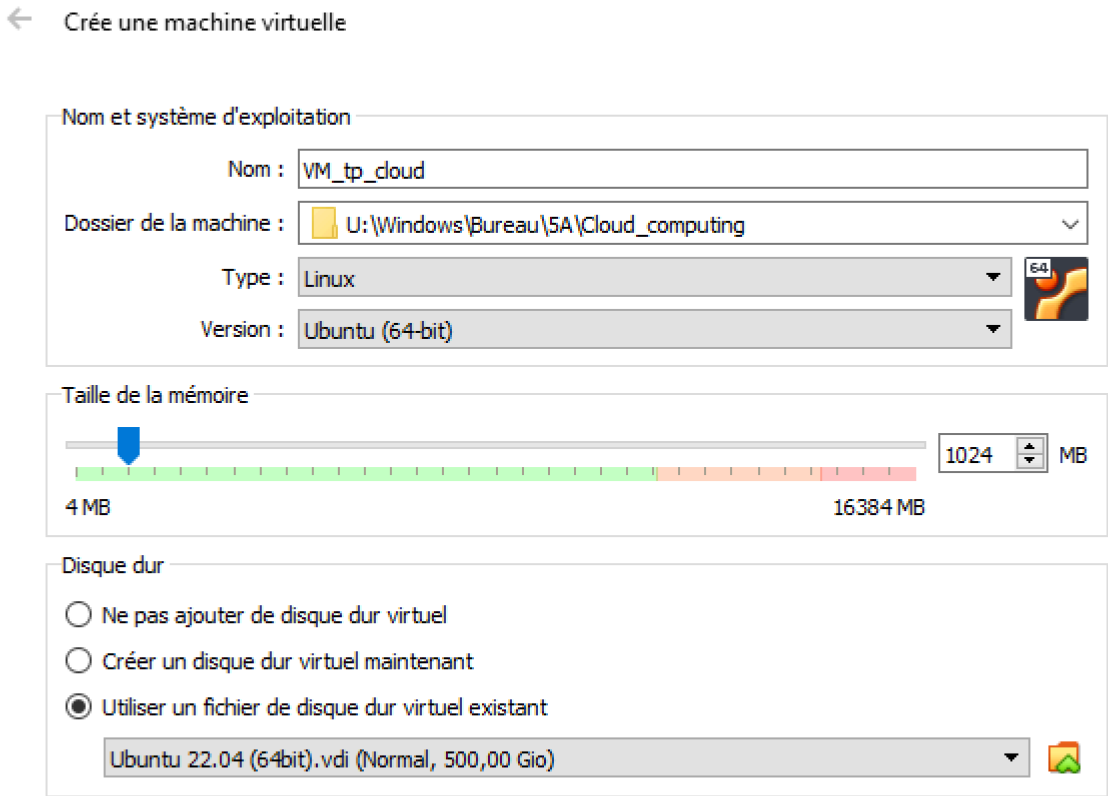
Node Port :

```
root@master:~# kubectl describe services fastapi-app-service
Name:                     fastapi-app-service
Namespace:                default
Labels:                   <none>
Annotations:              <none>
Selector:                 app=fastapi-app
Type:                     NodePort
IP Family Policy:         SingleStack
IP Families:              IPv4
IP:                       10.107.149.150
IPs:                      10.107.149.150
Port:                     <unset>  80/TCP
TargetPort:               5000/TCP
NodePort:                 <unset>  30584/TCP
Endpoints:
172.18.235.132:5000,172.18.235.133:5000,172.18.235.134:5000
Session Affinity:         None
External Traffic Policy:  Cluster
Events:                   <none>
```

```
ports:
      - protocol: TCP
      port: 80
      targetPort: 5000
```

Annexes :

Annexe 1 : VM creation with wanted parameters



Annexe 2 : ping VM => host

```
osboxes@osboxes:~/Desktop$ ping 10.1.5.28
PING 10.1.5.28 (10.1.5.28) 56(84) bytes of data.
64 bytes from 10.1.5.28: icmp_seq=1 ttl=127 time=0.725 ms
```

Annexe 3 : Ping host => VM

```
U:\>ping 10.0.2.15

Envoi d’une requête 'Ping'  10.0.2.15 avec 32 octets de données :
Délai d’attente de la demande dépassé.
```

Annexe 4 :
ADD

Annexe 5 : ssh server installation on the VM

```
osboxes@osboxes:~/Desktop$ sudo apt-get install openssh-server
```

Annexe 6 : Port forwarding

Nom	Protocole	IP hôte	Port hôte	IP invité	Port invité
ssh	TCP	10.1.5.231	1234	10.0.2.15	22

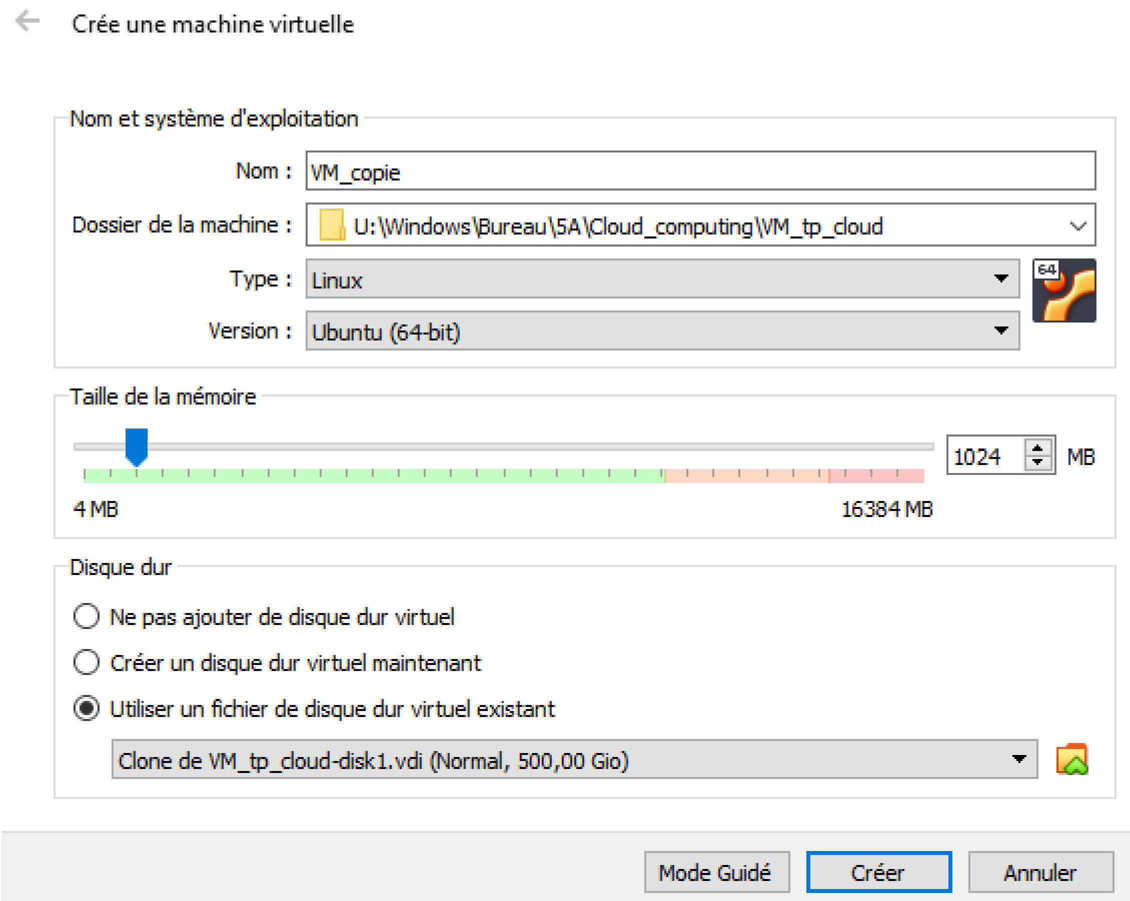
Annexe 7 : ssh sur port forward

```
U:\>ssh osboxes@192.168.56.1 -p 2222
The authenticity of host '[192.168.56.1]:2222 ([192.168
ECDSA key fingerprint is SHA256:vtbafypDgdxPvKVEahVqyal
Are you sure you want to continue connecting (yes/no/[f
Warning: Permanently added '[192.168.56.1]:2222' (ECDSA
osboxes@192.168.56.1's password:
Welcome to Ubuntu 22.04 LTS (GNU/Linux 5.15.0-25-generi

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

529 updates can be applied immediately.
300 of these updates are standard security updates.
To see these additional updates run: apt list --upgrada


The programs included with the Ubuntu system are free s
the exact distribution terms for each program are descr
individual files in /usr/share/doc/*/copyright.
```

Annexe 8 : VirtualBox VM image

Clone de VM_tp_cloud-disk1.vdi	10/10/2023 18:44	Virtual Disk Image	13 359 104 ...

Annexe 9 : VM creation from an image



Annexe 10 : package to let 'apt' access package over HTTPS

```
osboxes@osboxes:~/Desktop$ sudo apt install apt-transport-https ca-certificates curl
softwares-properties-common
```

Annexe 11 : add GPG key to access Docker repository

```
osboxes@osboxes:~/Desktop$ curl -fsSL https://download.docker.com/linux/ubuntu/gpg |
sudo gpg --dearmor -o /usr/share/keyrings/docker-archive-keyring.gpg
```

Annexe 12 : Add Docker repository to APT sources

```
osboxes@osboxes:~/Desktop$ echo "deb [arch=$(dpkg --print-architecture) signed-by=/usr/share
/keyrings/docker-archive-keyring.gpg] https://download.docker.com/linux/ubuntu $(lsb_release
 -cs) stable" | sudo tee /etc/apt/sources.list.d/docker.list > /dev/null
```

Annexe 13 : Docker installation

```
osboxes@osboxes:~/Desktop$ sudo apt install docker-ce
```

Annexe 14 : Checking Dockers installation status

```
osboxes@osboxes:~/Desktop$ sudo systemctl status docker
● docker.service - Docker Application Container Engine
     Loaded: loaded (/lib/systemd/system/docker.service; enabled; vendor preset: enabled)
     Active: active (running) since Tue 2023-10-10 13:07:13 EDT; 1min 43s ago
TriggeredBy: ● docker.socket
```

Annexe 15 : getting Ubuntu image for the Docker container

```
osboxes@osboxes:~/Desktop$ sudo docker pull ubuntu
Using default tag: latest
latest: Pulling from library/ubuntu
37aaf24cf781: Pull complete
Digest: sha256:9b8dec3bf938bc80fbe758d856e96fdfab5f56c39d44b0cff351e847bb1b01ea
Status: Downloaded newer image for ubuntu:latest
docker.io/library/ubuntu:latest
```

Annexe 16 : Creating Docker Container from Ubuntu image

```
osboxes@osboxes:~/Desktop$ sudo docker run --name ct1 -it ubuntu
root@df1b4bc3dc71:/#
```

Annexe 17 : Install packages to see connectivity

```
osboxes@osboxes:~/Desktop$ sudo apt-get -y update && apt-get -y install net-tools
iputils-ping
```

Annexe 18 : command to see a container's IP address

```
osboxes@osboxes:~/Desktop$ sudo docker inspect ct1
```

Annexe 19 : Creation of a second container instance

```
osboxes@osboxes:~/Desktop$ sudo docker run --name ct2 -it ubuntu
```

Annexe 20 : installation of ‘nano’ package on a container

```
root@bc584615a72a:/# apt-get -y update && apt install nano
```

Annexe 21 : Container snapshot using Docker command ‘commit’

```
osboxes@osboxes:~/Desktop$ sudo docker image ls
REPOSITORY   TAG         IMAGE ID       CREATED            SIZE
cloud        ct2_copie   698d6c2ba75c   About a minute ago 124MB
```

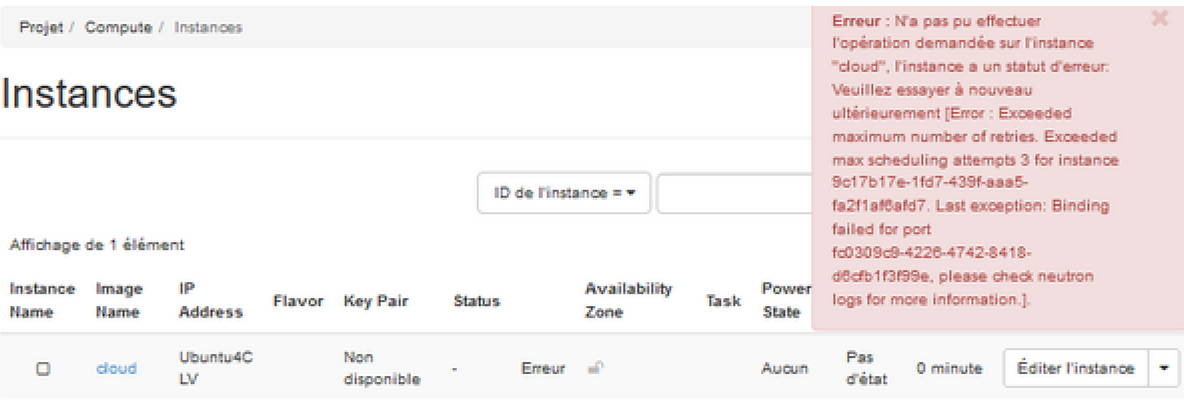
Annexe 22 : Container created from another container image

```
osboxes@osboxes:~/Desktop$ sudo docker run --name ct3 -it cloud:ct2_copie
root@22a5c0e1104d:/#
```

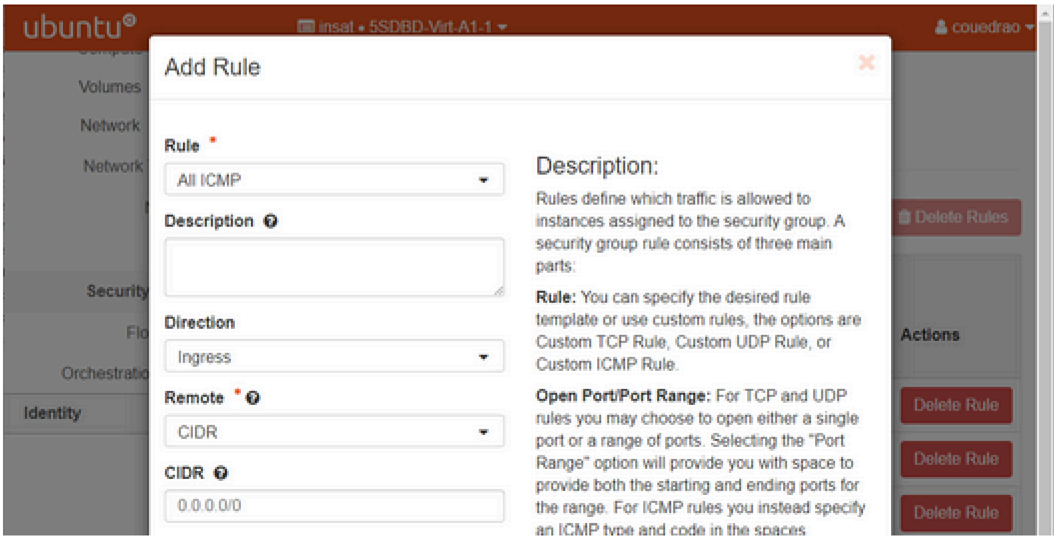
Annexe 23 : Container image using Dockerfile

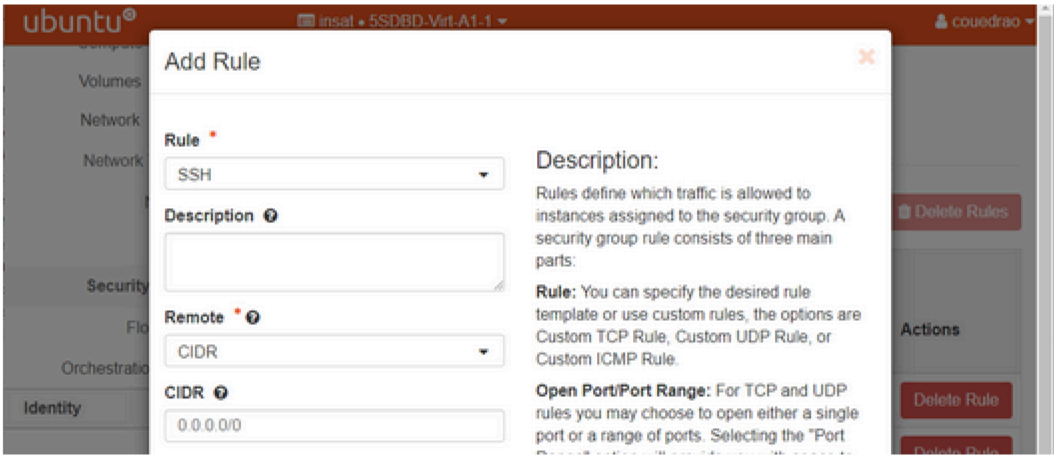
```
osboxes@osboxes:~/Desktop$ sudo docker images
REPOSITORY   TAG          IMAGE ID       CREATED          SIZE
images       dockerfile   b133b594d915   7 seconds ago    124MB
images       version0     07e9d6179e84   13 minutes ago   124MB
ubuntu       latest       3565a89d9e81   2 weeks ago      77.8MB
ubuntu       <none>       216c552ea5ba   12 months ago    77.8MB
```

Annexe 24 : OpenStack error : trying to connect VM to internet through public IP address



Annexe 25 : OpenStack security rules modification (add 'SSH' + 'ICMP')





Annexe 26 : ping VM => host

```
user@tutorial-vm:~$ ping  10.1.5.87
PING 10.1.5.87 (10.1.5.87) 56(84) bytes of data.
64 bytes from 10.1.5.87: icmp_seq=1 ttl=126 time=0.815 ms
64 bytes from 10.1.5.87: icmp_seq=2 ttl=126 time=1.07 ms
64 bytes from 10.1.5.87: icmp_seq=3 ttl=126 time=1.05 m
```

Annexe 27 : ping host => VM

```
ping 172.16.10.81

Envoi d'une requête 'Ping'  172.16.10.81 avec 32 octets de données :
Délai d'attente de la demande dépassé.
Délai d'attente de la demande dépassé.

Statistiques Ping pour 172.16.10.81:
        Paquets : envoyés = 2, reçus = 0, perdus = 2 (perte 100%),
```

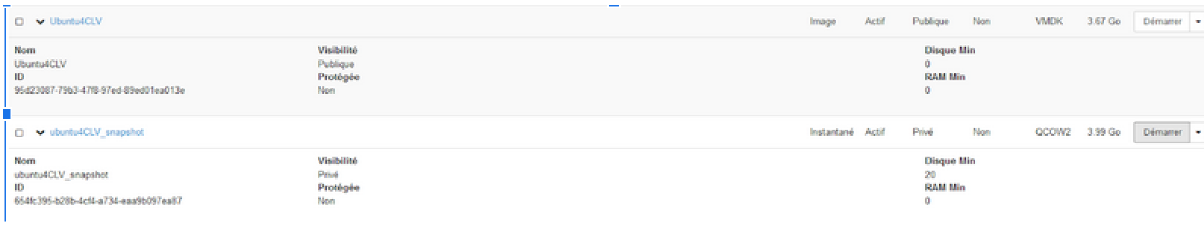
Annexe 28 : ping host => VM's floating IP

```
ping 192.168.37.50

Envoi d'une requête 'Ping'  192.168.37.50 avec 32 octets de données :
Réponse de 192.168.37.50 : octets=32 temps=2 ms TTL=62
Réponse de 192.168.37.50 : octets=32 temps=1 ms TTL=62
Réponse de 192.168.37.50 : octets=32 temps=1 ms TTL=62
Réponse de 192.168.37.50 : octets=32 temps<1ms TTL=62
```

Annexe 29 : SSH host => VM's floating IP

```
ssh user@192.168.37.50
user@192.168.37.50's password:
Welcome to Ubuntu 18.04.3 LTS (GNU/Linux 4.15.0-65-generic x86_64)
```

Annexe 30 : VM snapshot on OpenStack



Annexe 31 : client installation of VirtualBox VM

```
$ sudo apt install python3-openstackclient
```

Annexe 32 : client configuration from OpenStack "RCv3" file

```
$source fichier_rc.sh
```

Annexe 33 : open OpenStack client

```
$ openstack
```

Annexe 34 : package installation to deploy the calculator

```
$ sudo apt install nodejs npm curl
```

Annexe 35 : VM for web services deployment

Instances

ID de l'instance = ▾ | Filtrer | Lancer une instance | Supprimer

Affichage de 5 éléments

Instance Name	Image Name	IP Address	Flavor	Key Pair	Status	Availability Zone	Task	Power State	Age
DivService	alpine-node	172.16.10.45	Non disponible	-	Active	nova	Aucun	En fonctionnement	0 minute
MulService	alpine-node	172.16.10.152	tiny	-	Active	nova	Aucun	En fonctionnement	0 minute
SubService	alpine-node	172.16.10.14	tiny	-	Active	nova	Aucun	En fonctionnement	0 minute
SumService	alpine-node	172.16.10.95	tiny	-	Active	nova	Aucun	En fonctionnement	1 minute

Annexe 36 : HTTP rules to access microservices

Direction	Ether Type	IP Protocol	Port Range	Remote IP Prefix	Remote Security Group	Description	Actions
Sortie	IPv4	Tous	Tous	0.0.0.0/0	-	-	Supprimer une Règle
Sortie	IPv4	ICMP	Tous	0.0.0.0/0	-	-	Supprimer une Règle
Sortie	IPv6	Tous	Tous	::/0	-	-	Supprimer une Règle
Entrée	IPv4	Tous	Tous	-	default	-	Supprimer une Règle
Entrée	IPv4	ICMP	Tous	0.0.0.0/0	-	-	Supprimer une Règle
Entrée	IPv4	TCP	22 (SSH)	0.0.0.0/0	-	-	Supprimer une Règle
Entrée	IPv4	TCP	80 (HTTP)	0.0.0.0/0	-	-	Supprimer une Règle
Entrée	IPv6	Tous	Tous	-	default	-	Supprimer une Règle

Affichage de 8 éléments

Annexe 37 : floating IP address to access microservices

CalculatorService	Ubuntu4CLV	172.16.10.177, 192.168.37.50

Annexe 38 : HTTP request to the web service

```
curl -d "(5+6)*2" -X POST http://192.168.37.50
result = 22
```

Annexe 39 : HTTP response from the web service

```
New request :
(5+6)*2 = 22
```

Annexe 40 : Dockerfile to configure each VM when created

```
FROM ubuntu

VAR SERVICE_URL

RUN apt update
RUN apt install -y wget nodejs npm

RUN mkdir services

WORKDIR /services
RUN npm install sync-request
RUN wget -c ${SERVICE_URL} -O service.js

CMD ["node service.js"]
```

Annexe 41 : Command to build all the the VM with the associated microservice

```
docker build --build-arg
service_url="http://homepages.laas.fr/smedjiah/tmp/CalculatorService.js"
-t service:calculator -f service.dockerfile .

docker build --build-arg
service_url="http://homepages.laas.fr/smedjiah/tmp/SumService.js" -t
service:sum -f service.dockerfile .

docker build --build-arg
service_url="http://homepages.laas.fr/smedjiah/tmp/SubService.js" -t
service:sub -f service.dockerfile .

docker build --build-arg
service_url="http://homepages.laas.fr/smedjiah/tmp/MulService.js" -t
service:mul -f service.dockerfile .

docker build --build-arg
service_url="http://homepages.laas.fr/smedjiah/tmp/DivService.js" -t
service:div -f service.dockerfile .
```

Annexe 42 : Launching web service

```
sudo docker run --name CalculatorService --net calculator-network --ip
172.16.10.12 -it service:test
```

Annexe 43 : Targeted network configuration

```
const { exec } = require('child_process');

async function Exec(command) {
    return new Promise((resolve, reject) => {
       exec(command, (err, stdout, stderr) => {
             if (err) {
                   console.log(err.message);
                   reject();
                   return;
             }
             if (stderr) {
                   console.log(stderr);
                   reject();
                   return;
             }
             console.log(stdout);
             resolve();
```

```
        });
    });
}

async function CreateNetwork(name, ip, gw, pool) {
    await Exec(`openstack network create ${name}`);
    await Exec(`openstack subnet create ${name} --network ${name}
--subnet-range ${ip} --gateway ${gw} --allocation-pool
start=${pool[0]},end=${pool[1]}`);
}

async function CreateRouter(name) {
    await Exec(`openstack router create ${name}`);
}

async function AddRouterToNetwork(router, network) {
    await Exec(`openstack router add subnet ${router} ${network}`);
}

async function CreateNetworks(networks) {
    for (const network of networks) await CreateNetwork(network.name,
network.subnet, network.gw, network.pool);
}

async function CreateInstance(name, image, disk, networks) {
    await Exec(`openstack flavor create todo`)
}

async function CreateRouters(routers) {
    for (const router of routers) {
       await CreateRouter(router.name);
       for (const network of router.networks) await
AddRouterToNetwork(router.name, network.name);
    }
}

(async () => {
    const conf = {
       networks : [
             {
                   name: "test1",
                   subnet: "192.168.1.0/24",
                   gw: "192.168.1.1",
                   pool: ["192.168.1.100", "192.168.1.200"]
             },
       ],
```

```
        routers : [
              {
                    name: "rt_test",
                    networks: [
                          {
                                name: "test1",
                                ip: "192.168.1.1"
                          }
                    ]
              }
        ],
        instances : [
              {
                    name: "instance1",
                    network: [],
                    image: "nom_image",
                    disk: "nom_disk",
              }
        ]
    }

    await CreateNetworks(conf.networks);

    await CreateRouters(conf.routers);
})();
```