INNOVATIVE SMART SYSTEMS

INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES TOULOUSE

CLOUD COMPUTING AND VIRTUALIZATION

2024-2025

Noël Jumin
Clément Gauché

1. Introduction

This lab, conducted in binome, aimed to deepen our understanding of cloud computing and virtualization technologies. Throughout the lab sessions, we explored key concepts related to the use of hypervisors, virtualization hosts (virtual machines and containers), and network configuration. The objectives focused on testing the features offered by cloud service providers at the infrastructure level, using tools such as VirtualBox and OpenStack. These practical exercises enabled us to apply theoretical knowledge to real-world scenarios by provisioning VMs and containers, managing network connections. Our main goal was to understand how modern IT environments leverage virtualization to ensure efficient resource management, security, and scalability.

Noël Jumin
Clément Gauché

2. Theoretical part

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

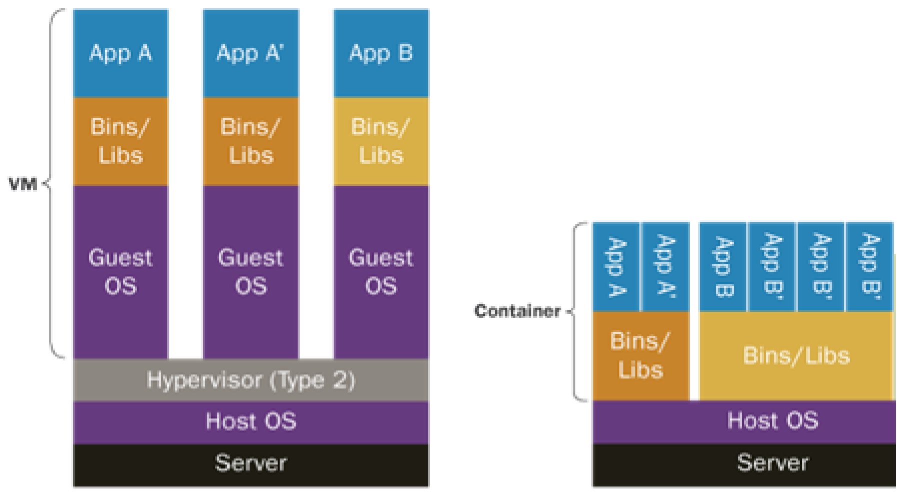


Figure 1: VM vs CT

The two figures represent the Virtual Machines and Containers concepts, which are the current main virtualization technologies.
The main difference between these 2 figures is that one is virtualizing the whole machine (Os, Bins/Libs, App), and the other is only virtualizing the softwares layers above the OS. A second difference is the presence of an hypervisor in the first figure which divides the hardwares ressources of the host machine.

	Developer		Administrator	
Technology	VM	CT	VM	CT
Virtualization cost	It virtualizes a complete OS for each machine. → Heavy cost	Virtualize only a small part of the OS and above. It shares the host operating system's. → Light cost	Virtualize the whole machine. It has a heavier cost but can emulate different OS simultaneously.	Virtualize only a small part of the OS and above.

Usage of CPU, memory and network	Every OS needs to be configured and it borrows the hardware components during the execution.	The application uses a lot of the CPU, Memory, etc… only if needed.	It reserves hardware components so the application works exactly as it was developed.	Hardware is not guaranteed so if the application needs 2 cores, it is possible that they will not be available at every time.
Security	There is a real separation between each OS. There is no communication between them.	There is no real separation between the machines because the OS is shared.	There is a real separation between each OS. There is no communication between them. So no risk of problems on the original OS.	There is no real separation between the machines because the OS is shared. Risk of damages on the OS.
Performance	Slower to access ressources cause it needs to pass through the hypervisor.	Faster to access resources because it only needs to go through the OS.		
Tooling for the continuous integration	Need to create a whole new OS each time we want to create a new machine	Updating or upgrading the OS is done once, and can then be deployed at scale.	Great portability with images.. The only problem is that they are heavy.	Great portability between each application because they are working on the same OS.

2.2. Similarities and differences between the existing CT types

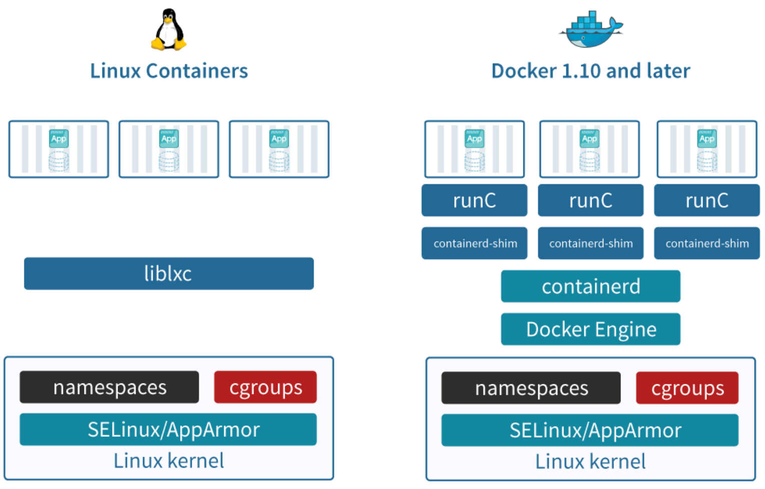


Figure 2: Linux Lxc vs Docker

We have determined this difference by using this source:

Technology	Linux Lxc	Docker
Application isolation	Every container shares the same liblxc library that contains runtime for example.	Each container has its own runtime and library so the app is more isolated.
Containerization level	The containerization is done on the kernel level If the host kernel has a problem, it will not necessarily affect the kernel of the different containers because they have their own Linux OS.	The host OS is shared which means that a problem on this OS can affect the different containers. Each container only runs their own user space.
Tooling	The tools are really near from the linux ones like systemd, but there are no special tools for it like docker has.	Prebuilt images available on DockerHub. A lot of plugins are also available with extended functionality. There is also a large community and every problem can be resolved easily with the different forums existing.

Portability	It can only work on a Linux kernel.	High portability on every machine such as MacOS, Windows, etc… It only needs to have docker on it to run.
Performance	Slow to launch cause it needs to launch the Kernel of each container. Fast to run cause it runs like a normal Linux application.	Fast to launch cause it only needs the host OS to start. Fast to run (little slower than Lcx) on Linux kernel because it doesn't need to pass through the docker engine. It is more slower on other kernels because it needs to pass through the docker engine.
Security	The host kernel is separated so the containers can not access it.	Bigger target due to his popularity. The containers also run on the host kernel so one container can possibly access it.

2.3. Similarities and differences between Type 1 & Type 2 of hypervisors' architectures

Hypervisors, or virtual machine monitors (VMMs), enable the creation and management of virtual machines (VMs). They are categorized in two types: Type 1 (bare-metal) and Type 2 (hosted).

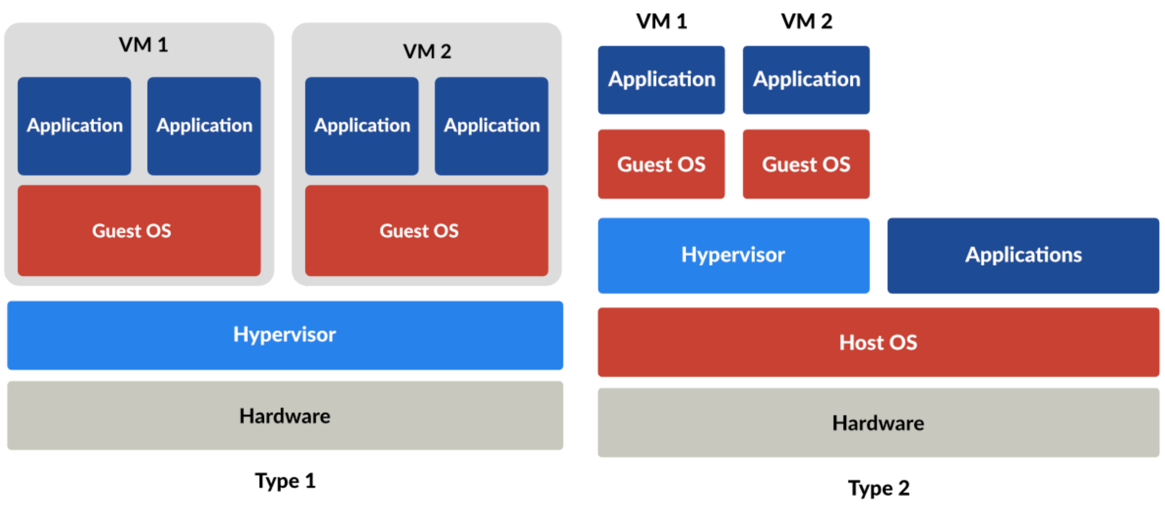


Figure 3: Bare metal and hosted supervisors

	Type 1 Hypervisors	Type 2 (Hosted)
Architecture	Runs directly on the host's hardware.	Runs on top of a host operating system.
Performance	Offers better performance due to direct access to hardware.	Can have higher latency and overhead due to the additional OS layer. Less scalable
VM management / tools	Includes built-in management tools and interfaces for VM management.	Relies on the host OS's management capabilities or installed tools
Security	Isolated from any operating system likely to be compromised → Each VM has its own OS.	More vulnerable to attacks through the host OS, which can expose the hypervisor.
Example of VMs technologies	Microsoft Hyper-V, Xen	VirtualBox, VMware, Parallels Desktop.

The open-source cloud computing platform OpenStack, is using the … hypervisors.

3. Practical part

In this section, we will focus on creating our first virtual machine (VM) or container (CT) using VirtualBox and Docker. This will enable us to explore various features offered by these tools, including image creation, as well as gaining a better understanding of the connectivity of virtualized entities.

3.1. Virtual machine

3.1.1. Creation and configuration of a VM

In this part the creation of the Linux OS VM is done on VirtualBox, a type 2 hypervisor.

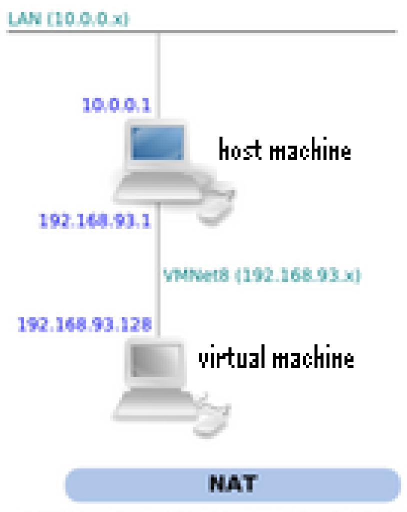


Figure 4: Linux VM network

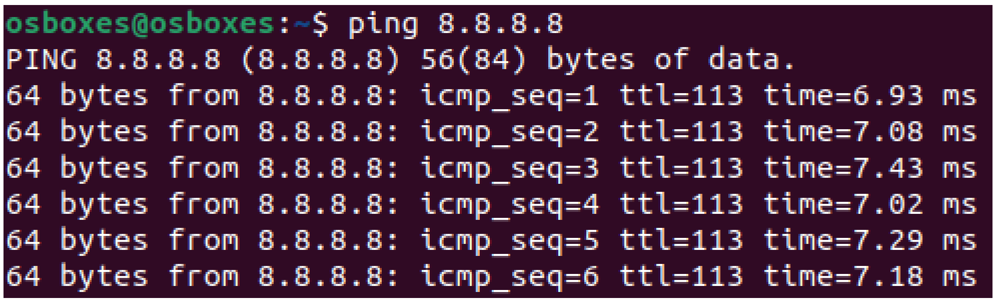
3.1.2. Testing the VM connectivity

VirtualBox hypervisor (Type 2) in NAT mode. By using the *ipconfig* and *ifconfig* commands, we are able to know the different addresses available on our computer.

Entity	IP Address	Infos
PC	10.1.5.147	It is an INSA address. It is addressable from the INSA network.
Virtual machine	10.0.2.15	This is neither a public or an INSA address. It's not addressable from anywhere.

From our first assumption, none of these addresses are on the same network so technically the VM is not routable and she cannot communicate with the exterior too.

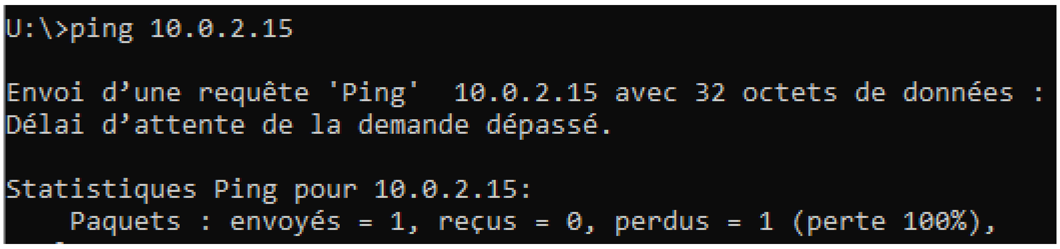
When we ping Google from the VM to show the possibility of connection to the outside, it works:



```
osboxes@osboxes:~$ ping 8.8.8.8
PING 8.8.8.8 (8.8.8.8) 56(84) bytes of data.
64 bytes from 8.8.8.8: icmp_seq=1 ttl=113 time=6.93 ms
64 bytes from 8.8.8.8: icmp_seq=2 ttl=113 time=7.08 ms
64 bytes from 8.8.8.8: icmp_seq=3 ttl=113 time=7.43 ms
64 bytes from 8.8.8.8: icmp_seq=4 ttl=113 time=7.02 ms
64 bytes from 8.8.8.8: icmp_seq=5 ttl=113 time=7.29 ms
64 bytes from 8.8.8.8: icmp_seq=6 ttl=113 time=7.18 ms
```

Figure 5: Snapshot of VM ping to Google

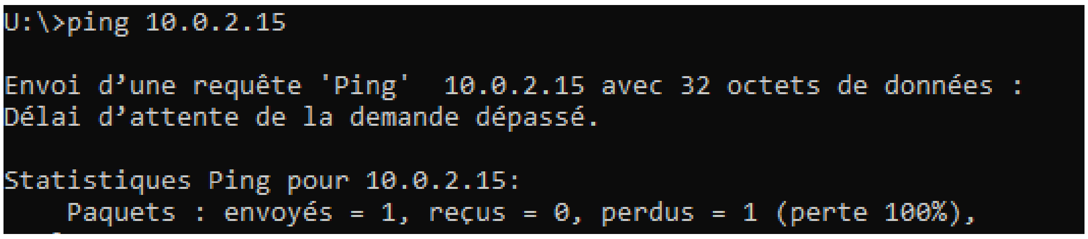
But when we test pinging the VM from the host, it doesn't work:



```
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.

Statistiques Ping pour 10.0.2.15:
    Paquets : envoyés = 1, reçus = 0, perdus = 1 (perte 100%),
```

Figure 6: Snapshot from host pinging the VM

Same problem; when pinging a neighbor host to the VM:



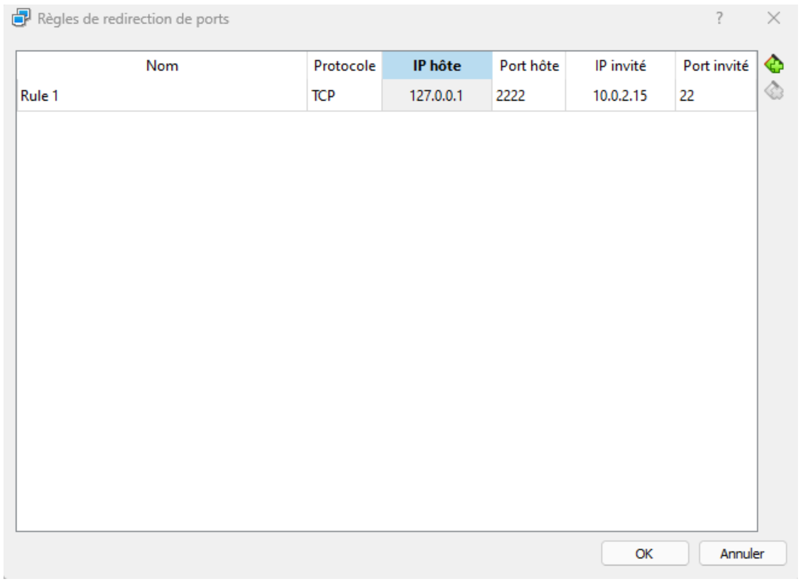
```
U:\>ping 10.0.2.15

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

Statistiques Ping pour 10.0.2.15:
    Paquets : envoyés = 1, reçus = 0, perdus = 1 (perte 100%),
```

Figure 7: Snapshot from neighbor host pinging the VM

So why does it work on one side but not the other one ?
There is a virtual router on the VM that works like a NAT. When we send a ping from the VM to a machine outside the local network, a public address (the host's IP address) is given by the virtual router so that it can communicate with the outside.

The VM address is not routable/reachable if access is initiated from the outside. To be able to address this private address, we use the host address on a specific port. We configure a port redirection on the virtual LAN to be able to redirect the flow to the virtual machine's IP address.

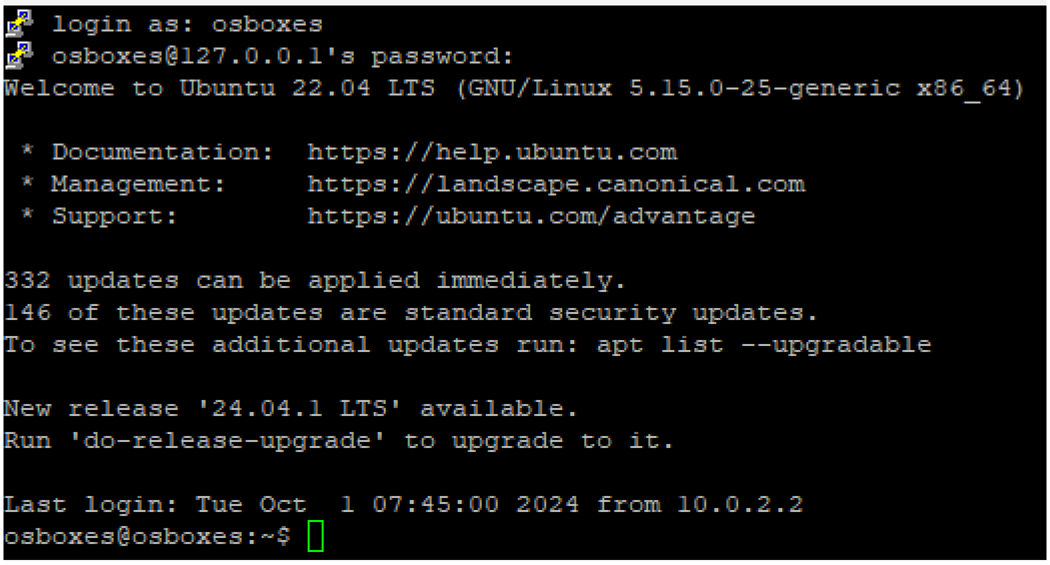
The host address currently in place is the loopback address. This means that only the host will be able to connect to the VM.



Règles de redirection de ports

Nom	Protocole	IP hôte	Port hôte	IP invité	Port invité
Rule 1	TCP	127.0.0.1	2222	10.0.2.15	22

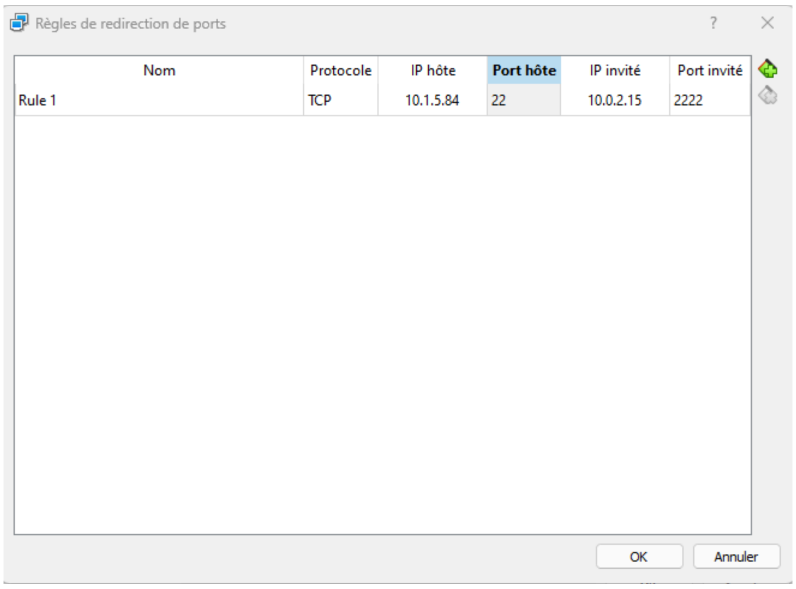
Figure 8: Snapshot of the configuration of the port forwarding



```
login as: osboxes
osboxes@127.0.0.1's password:
Welcome to Ubuntu 22.04 LTS (GNU/Linux 5.15.0-25-generic x86_64)

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

332 updates can be applied immediately.
146 of these updates are standard security updates.
To see these additional updates run: apt list --upgradable

New release '24.04.1 LTS' available.
Run 'do-release-upgrade' to upgrade to it.

Last login: Tue Oct  1 07:45:00 2024 from 10.0.2.2
osboxes@osboxes:~$
```

Figure 9: Snapshot of a SSH connexion from host to the VM using loopback

The VM is not accessible from a neighbor PC, so we are now using the host pc address as the host address and the VM address as the invited address.

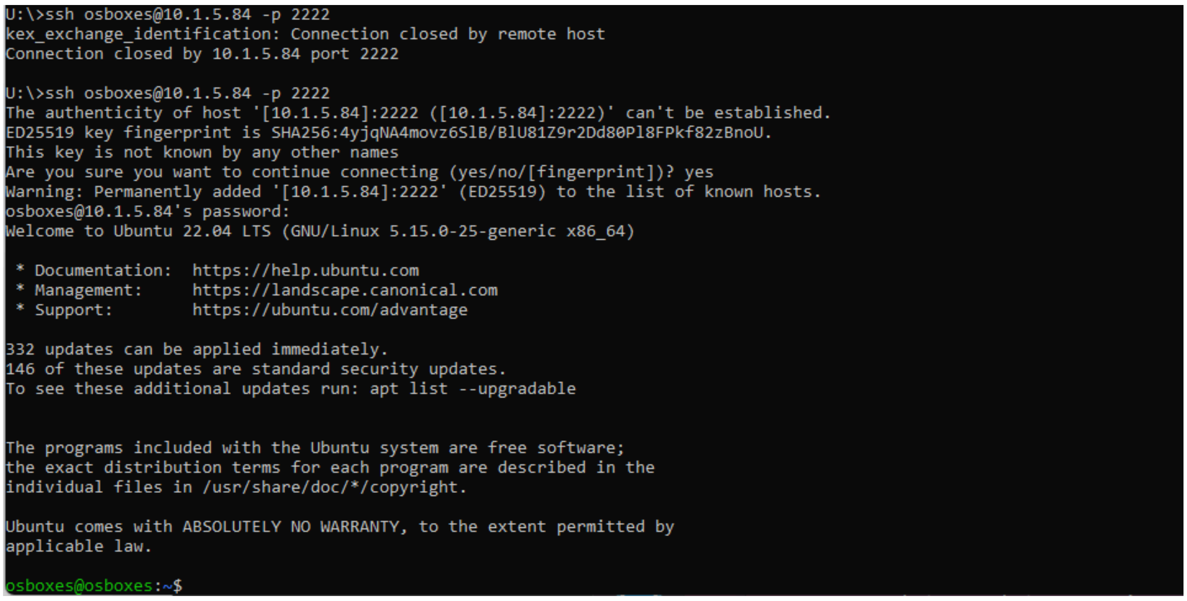


Règles de redirection de ports

Nom	Protocole	IP hôte	Port hôte	IP invité	Port invité
Rule 1	TCP	10.1.5.84	22	10.0.2.15	2222

Figure 10: Snapshot of the updated configuration of the port forwarding

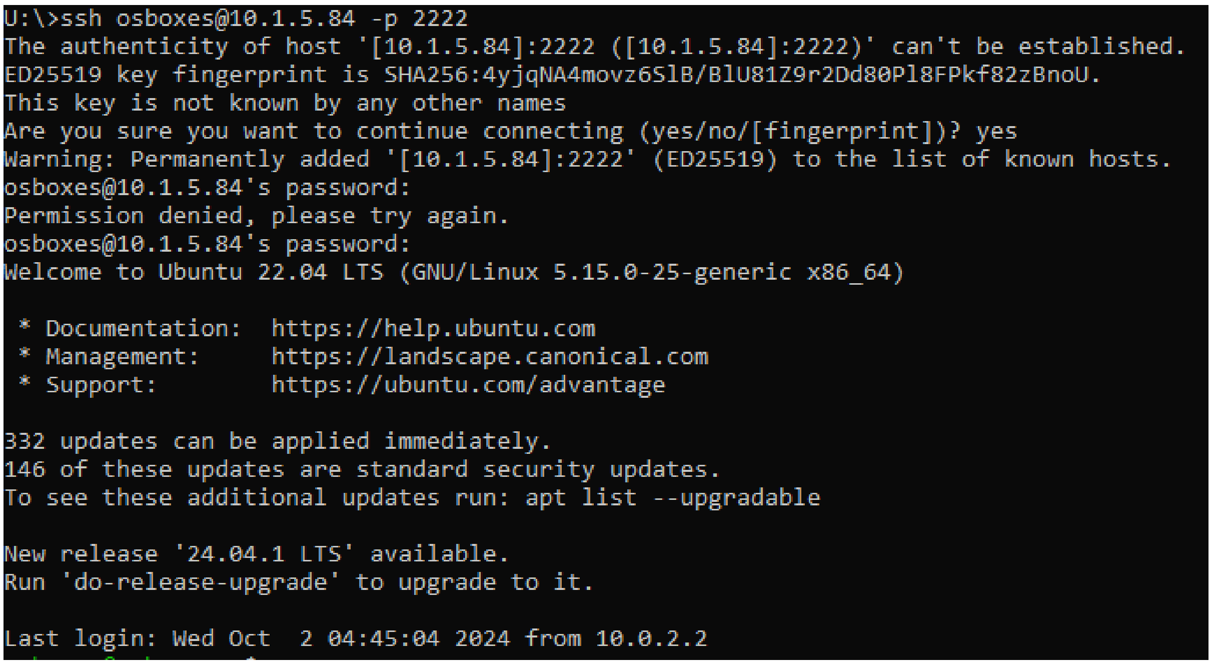
From the Host PC, it is now possible to connect to the VM using ssh:



```
U:\>ssh osboxes@10.1.5.84 -p 2222
kex_exchange_identification: Connection closed by remote host
Connection closed by 10.1.5.84 port 2222

U:\>ssh osboxes@10.1.5.84 -p 2222
The authenticity of host '[10.1.5.84]:2222 ([10.1.5.84]:2222)' can't be established.
ED25519 key fingerprint is SHA256:4yjqNA4movz6SlB/BlU81Z9r2Dd80Pl8FPkf82zBnoU.
This key is not known by any other names
Are you sure you want to continue connecting (yes/no/[fingerprint])? yes
Warning: Permanently added '[10.1.5.84]:2222' (ED25519) to the list of known hosts.
osboxes@10.1.5.84's password:
Welcome to Ubuntu 22.04 LTS (GNU/Linux 5.15.0-25-generic x86_64)

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

332 updates can be applied immediately.
146 of these updates are standard security updates.
To see these additional updates run: apt list --upgradable


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

Ubuntu comes with ABSOLUTELY NO WARRANTY, to the extent permitted by
applicable law.

osboxes@osboxes:~$
```

Figure 11: Snapshot of a SSH connexion from host to the VM

It’s now possible to do the same from any INSA address:



```
U:\>ssh osboxes@10.1.5.84 -p 2222
The authenticity of host '[10.1.5.84]:2222 ([10.1.5.84]:2222)' can't be established.
ED25519 key fingerprint is SHA256:4yjqNA4movz6SlB/BlU81Z9r2Dd80Pl8FPkf82zBnoU.
This key is not known by any other names
Are you sure you want to continue connecting (yes/no/[fingerprint])? yes
Warning: Permanently added '[10.1.5.84]:2222' (ED25519) to the list of known hosts.
osboxes@10.1.5.84's password:
Permission denied, please try again.
osboxes@10.1.5.84's password:
Welcome to Ubuntu 22.04 LTS (GNU/Linux 5.15.0-25-generic x86_64)

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

332 updates can be applied immediately.
146 of these updates are standard security updates.
To see these additional updates run: apt list --upgradable

New release '24.04.1 LTS' available.
Run 'do-release-upgrade' to upgrade to it.

Last login: Wed Oct  2 04:45:04 2024 from 10.0.2.2
```

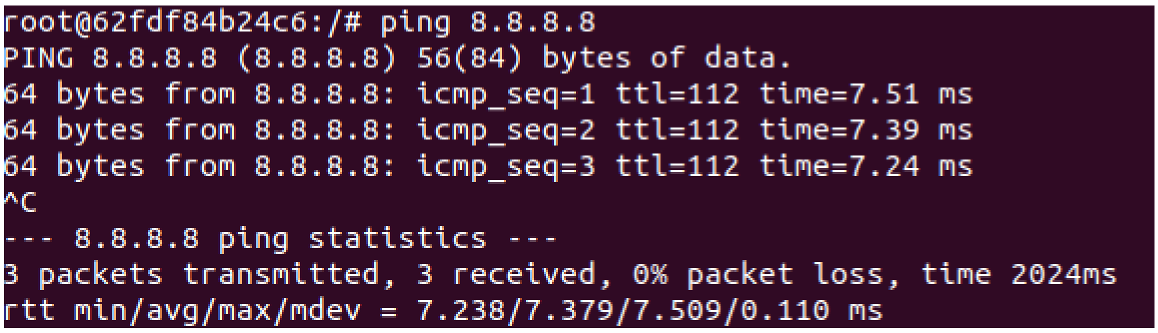
Figure 12: Snapshot of a SSH connexion from neighbor host to the VM

Our VM is now correctly configured to communicate using ssh with the exterior, we can do a copy of the disk to be able to reuse it in the future using this command:
"C:\Program Files\Oracle\VirtualBox\VBoxManage.exe" clonemedium
"U:\Windows\Bureau\VirtualBoxVM\Ubuntu 22.04 (64bit).vdi"
"U:\Windows\Bureau\VirtualBoxVM\Ubuntu 22.04 (64bit) copy.vdi"

3.2. Docker

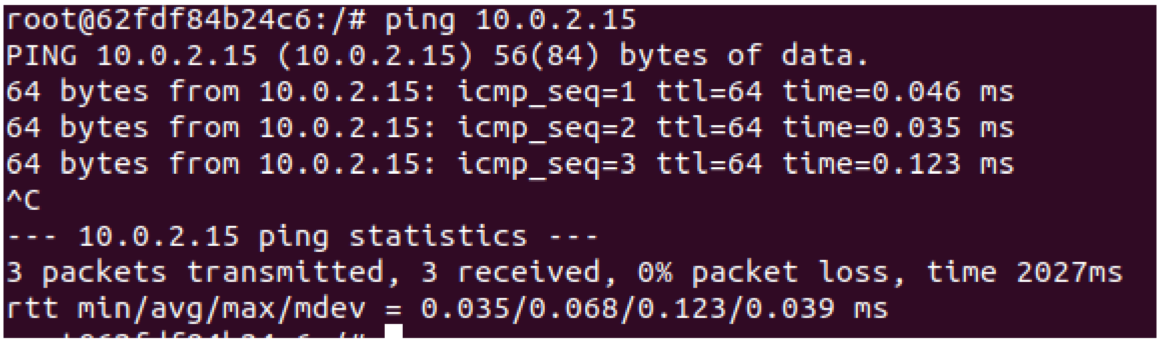
In this part, we will configure a docker with an Ubuntu image to understand how containers work. For practical reasons, this Docker container will be deployed on the VM we have created in the last part.

After configuring it, an IP address was given to this container which is: 172.17.0.2. With this address we've been able to see the same problems as the VM, it was pinging the exterior but the exterior could not ping the container:



```
root@62fdf84b24c6:/# ping 8.8.8.8
PING 8.8.8.8 (8.8.8.8) 56(84) bytes of data.
64 bytes from 8.8.8.8: icmp_seq=1 ttl=112 time=7.51 ms
64 bytes from 8.8.8.8: icmp_seq=2 ttl=112 time=7.39 ms
64 bytes from 8.8.8.8: icmp_seq=3 ttl=112 time=7.24 ms
^C
--- 8.8.8.8 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2024ms
rtt min/avg/max/mdev = 7.238/7.379/7.509/0.110 ms
```

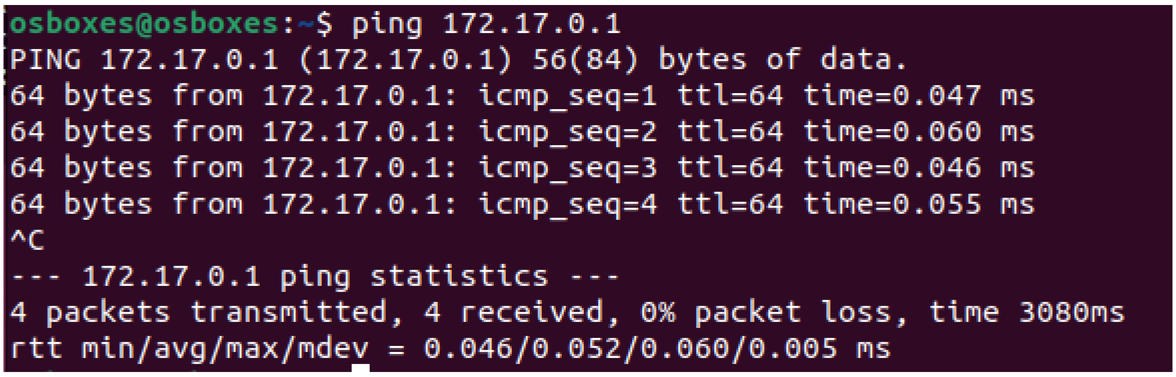
Figure 13: Snapshot of Google ping from the container



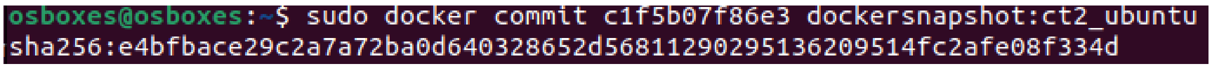
```
root@62fdf84b24c6:/# ping 10.0.2.15
PING 10.0.2.15 (10.0.2.15) 56(84) bytes of data.
64 bytes from 10.0.2.15: icmp_seq=1 ttl=64 time=0.046 ms
64 bytes from 10.0.2.15: icmp_seq=2 ttl=64 time=0.035 ms
64 bytes from 10.0.2.15: icmp_seq=3 ttl=64 time=0.123 ms
^C
--- 10.0.2.15 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2027ms
rtt min/avg/max/mdev = 0.035/0.068/0.123/0.039 ms
```

Figure 14: Snapshot of ping the VM from Docker

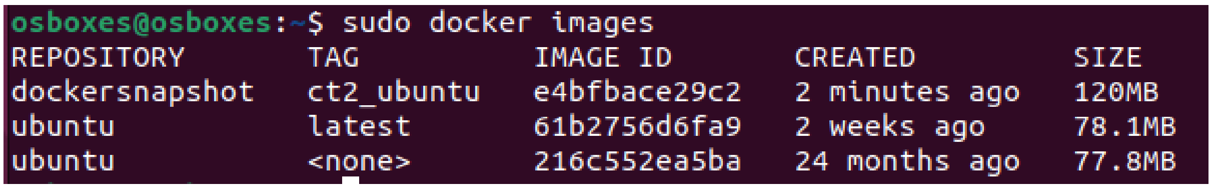
However, it is possible to ping our container from our VM because Docker automatically deploys a bridge between the machine deploying the container and the container:


```
osboxes@osboxes:~$ 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=0.047 ms
64 bytes from 172.17.0.1: icmp_seq=2 ttl=64 time=0.060 ms
64 bytes from 172.17.0.1: icmp_seq=3 ttl=64 time=0.046 ms
64 bytes from 172.17.0.1: icmp_seq=4 ttl=64 time=0.055 ms
^C
--- 172.17.0.1 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3080ms
rtt min/avg/max/mdev = 0.046/0.052/0.060/0.005 ms
```
Figure 15: Snapshot of ping the Docker from VM

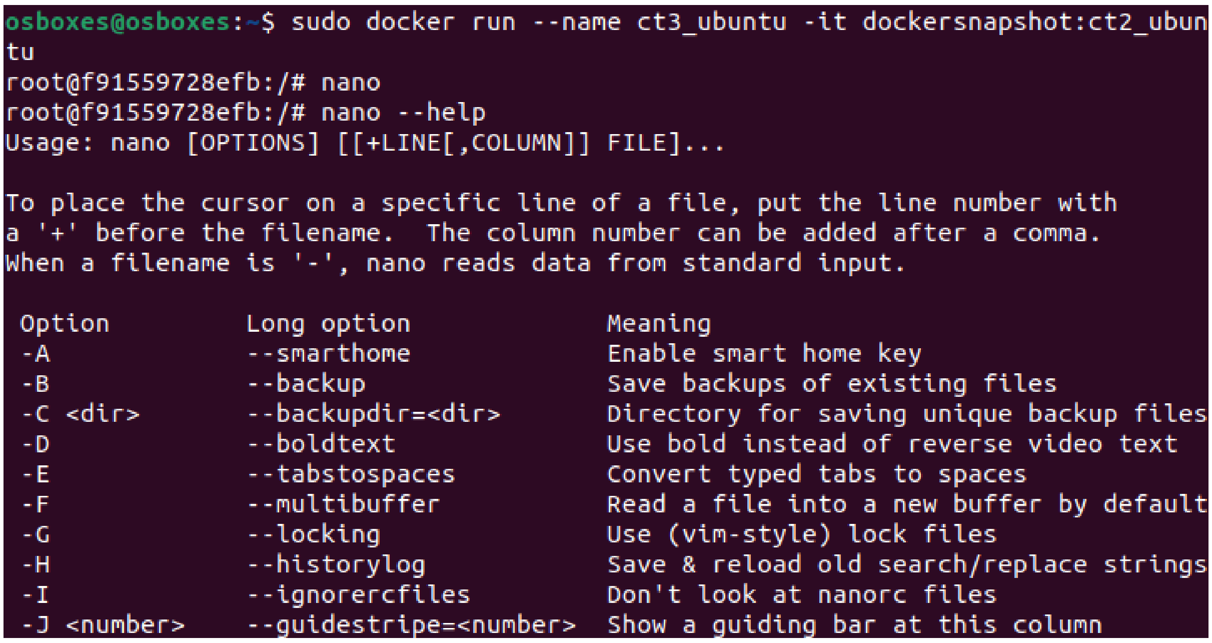
We can now if we want, commit our container image on Docker HUB to be able to reuse it on any other future container:


```
osboxes@osboxes:~$ sudo docker commit c1f5b07f86e3 dockersnapshot:ct2_ubuntu
sha256:e4bfbace29c2a7a72ba0d640328652d568112902951362095l4fc2afe08f334d
```
Figure 16: Snapshot of a commit on Docker HUB

As we can see, we can now see our image “ct2_ubuntu”:


```
osboxes@osboxes:~$ sudo docker images
REPOSITORY       TAG          IMAGE ID       CREATED          SIZE
dockersnapshot   ct2_ubuntu   e4bfbace29c2   2 minutes ago    120MB
ubuntu           latest       61b2756d6fa9   2 weeks ago      78.1MB
ubuntu           <none>       216c552ea5ba   24 months ago    77.8MB
```
Figure 17: Snapshot of docker images

This image contains a usable nano installation so we can save time of configuration:



```
osboxes@osboxes:~$ sudo docker run --name ct3_ubuntu -it dockersnapshot:ct2_ubun
tu
root@f91559728efb:/# nano
root@f91559728efb:/# nano --help
Usage: nano [OPTIONS] [[+LINE[,COLUMN]] FILE]...

To place the cursor on a specific line of a file, put the line number with
a '+' before the filename.  The column number can be added after a comma.
When a filename is '-', nano reads data from standard input.

 Option         Long option             Meaning
 -A             --smarthome             Enable smart home key
 -B             --backup                Save backups of existing files
 -C <dir>       --backupdir=<dir>       Directory for saving unique backup files
 -D             --boldtext              Use bold instead of reverse video text
 -E             --tabstospaces          Convert typed tabs to spaces
 -F             --multibuffer           Read a file into a new buffer by default
 -G             --locking               Use (vim-style) lock files
 -H             --historylog            Save & reload old search/replace strings
 -I             --ignorercfiles         Don't look at nanorc files
 -J <number>    --guidestripe=<number>  Show a guiding bar at this column
```

Figure 18: Snapshot of new container based on the last container image

3.3. OpenStack

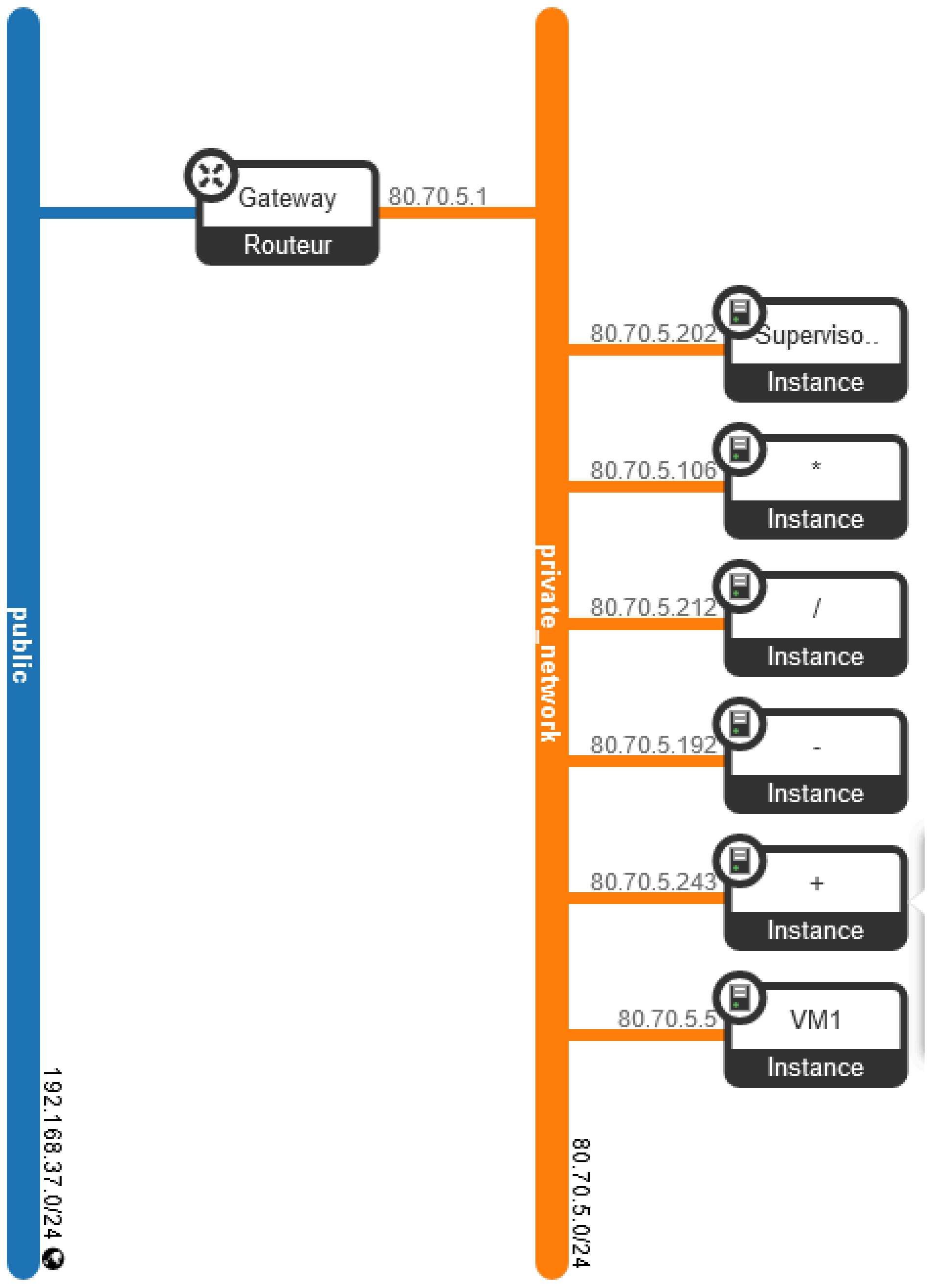


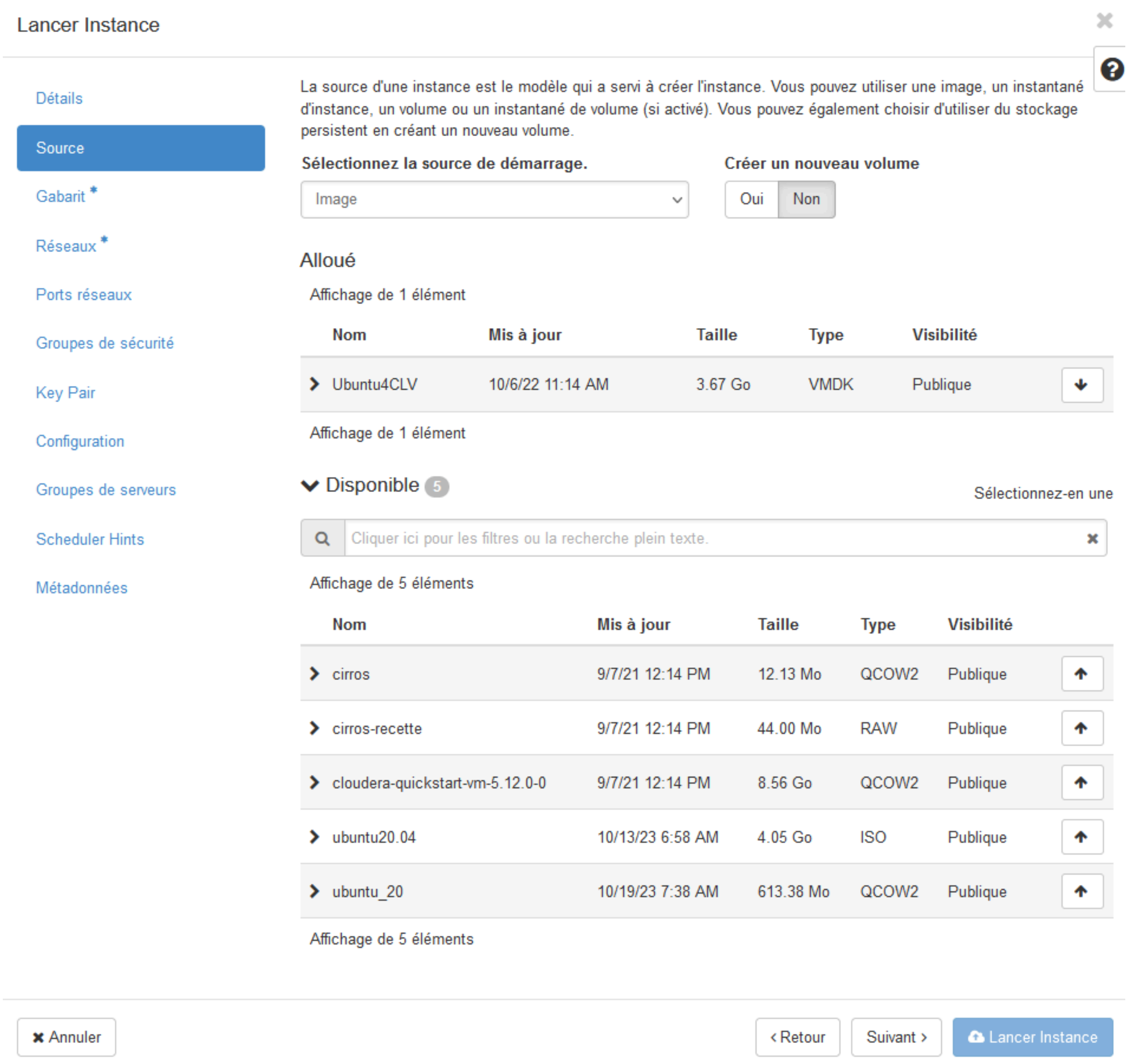
Figure 19: OpenStack network configuration

In this part we will create a network of containers, each one running a service that will be used for computation. There will be different arithmetic operations and the supervisor will call the operation needed for the operation requested by the client.

3.3.1. Creation of the containers using OpenStack

With OpenStack, it is really simple to create a container just in 3 steps:

1. Choose an existing image for your container:



Lancer Instance

La source d'une instance est le modèle qui a servi à créer l'instance. Vous pouvez utiliser une image, un instantané d'instance, un volume ou un instantané de volume (si activé). Vous pouvez également choisir d'utiliser du stockage persistent en créant un nouveau volume.

Sélectionnez la source de démarrage. Image

Créer un nouveau volume: Oui [✓ Non]

Alloué

Affichage de 1 élément

Nom	Mis à jour	Taille	Type	Visibilité
Ubuntu4CLV	10/6/22 11:14 AM	3.67 Go	VMDK	Publique

Affichage de 1 élément

Disponible 5 — Sélectionnez-en une

Cliquer ici pour les filtres ou la recherche plein texte.

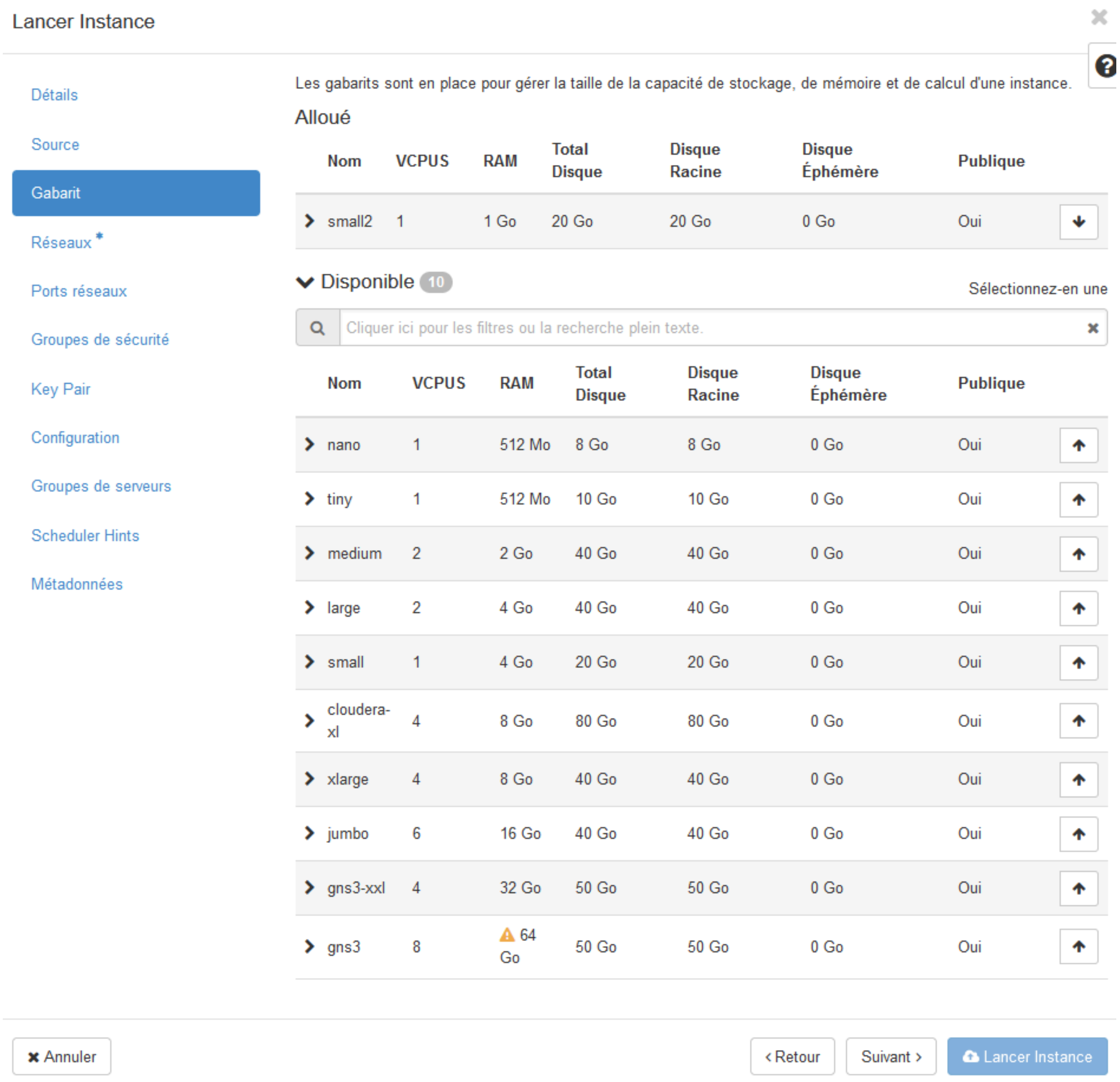
Affichage de 5 éléments

Nom	Mis à jour	Taille	Type	Visibilité
cirros	9/7/21 12:14 PM	12.13 Mo	QCOW2	Publique
cirros-recette	9/7/21 12:14 PM	44.00 Mo	RAW	Publique
cloudera-quickstart-vm-5.12.0-0	9/7/21 12:14 PM	8.56 Go	QCOW2	Publique
ubuntu20.04	10/13/23 6:58 AM	4.05 Go	ISO	Publique
ubuntu_20	10/19/23 7:38 AM	613.38 Mo	QCOW2	Publique

Affichage de 5 éléments

Figure 20: Snapshot of the image container configuration

2. Set a size for your container's storage:



Les gabarits sont en place pour gérer la taille de la capacité de stockage, de mémoire et de calcul d'une instance.

Alloué

Nom	VCPUS	RAM	Total Disque	Disque Racine	Disque Éphémère	Publique
small2	1	1 Go	20 Go	20 Go	0 Go	Oui

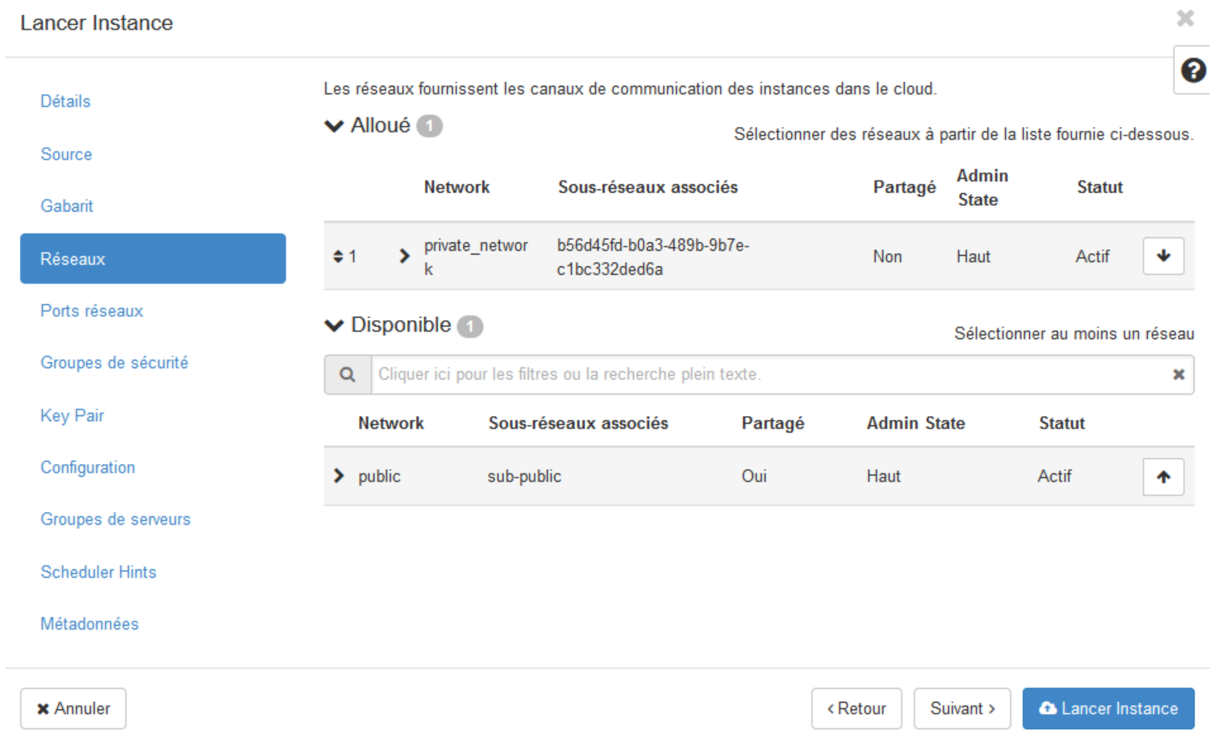
Disponible 10 — Sélectionnez-en une

Cliquer ici pour les filtres ou la recherche plein texte.

Nom	VCPUS	RAM	Total Disque	Disque Racine	Disque Éphémère	Publique
nano	1	512 Mo	8 Go	8 Go	0 Go	Oui
tiny	1	512 Mo	10 Go	10 Go	0 Go	Oui
medium	2	2 Go	40 Go	40 Go	0 Go	Oui
large	2	4 Go	40 Go	40 Go	0 Go	Oui
small	1	4 Go	20 Go	20 Go	0 Go	Oui
cloudera-xl	4	8 Go	80 Go	80 Go	0 Go	Oui
xlarge	4	8 Go	40 Go	40 Go	0 Go	Oui
jumbo	6	16 Go	40 Go	40 Go	0 Go	Oui
gns3-xxl	4	32 Go	50 Go	50 Go	0 Go	Oui
gns3	8	⚠ 64 Go	50 Go	50 Go	0 Go	Oui

Figure 21: Snapshot of the size container configuration

3. You can now link your container to a network existing:



Lancer Instance

Les réseaux fournissent les canaux de communication des instances dans le cloud.

Alloué 1 — Sélectionner des réseaux à partir de la liste fournie ci-dessous.

	Network	Sous-réseaux associés	Partagé	Admin State	Statut
1	private_network	b56d45fd-b0a3-489b-9b7e-c1bc332ded6a	Non	Haut	Actif

Disponible 1 — Sélectionner au moins un réseau

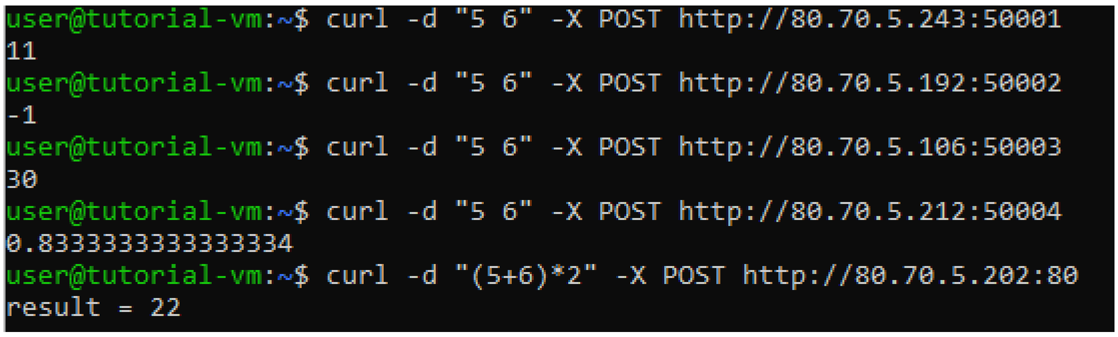
Cliquer ici pour les filtres ou la recherche plein texte.

Network	Sous-réseaux associés	Partagé	Admin State	Statut
public	sub-public	Oui	Haut	Actif

Figure 22: Snapshot of the network container configuration

3.3.2. Configuring a calculator

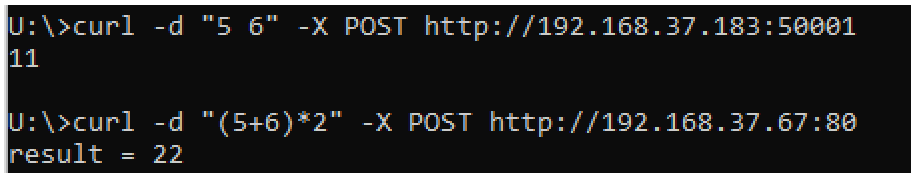
After that, you can open your containers with OpenStack and configure each one according to your needs. We can now test if we can use our arithmetic operation firstly from our client in the same network (VM1):



```
user@tutorial-vm:~$ curl -d "5 6" -X POST http://80.70.5.243:50001
11
user@tutorial-vm:~$ curl -d "5 6" -X POST http://80.70.5.192:50002
-1
user@tutorial-vm:~$ curl -d "5 6" -X POST http://80.70.5.106:50003
30
user@tutorial-vm:~$ curl -d "5 6" -X POST http://80.70.5.212:50004
0.8333333333333334
user@tutorial-vm:~$ curl -d "(5+6)*2" -X POST http://80.70.5.202:80
result = 22
```

Figure 23: Snapshot of the services called from a client on the same network

We can also, by associating a public address with the desired machine, call the desired service from anywhere:



```
U:\>curl -d "5 6" -X POST http://192.168.37.183:50001
11

U:\>curl -d "(5+6)*2" -X POST http://192.168.37.67:80
result = 22
```

Figure 24: Snapshot of the services called from a client outside the network

3.3.3. Secure the network

For security reasons we will create a private one to be sure that the client (public) can only access the calculator service and not the others.

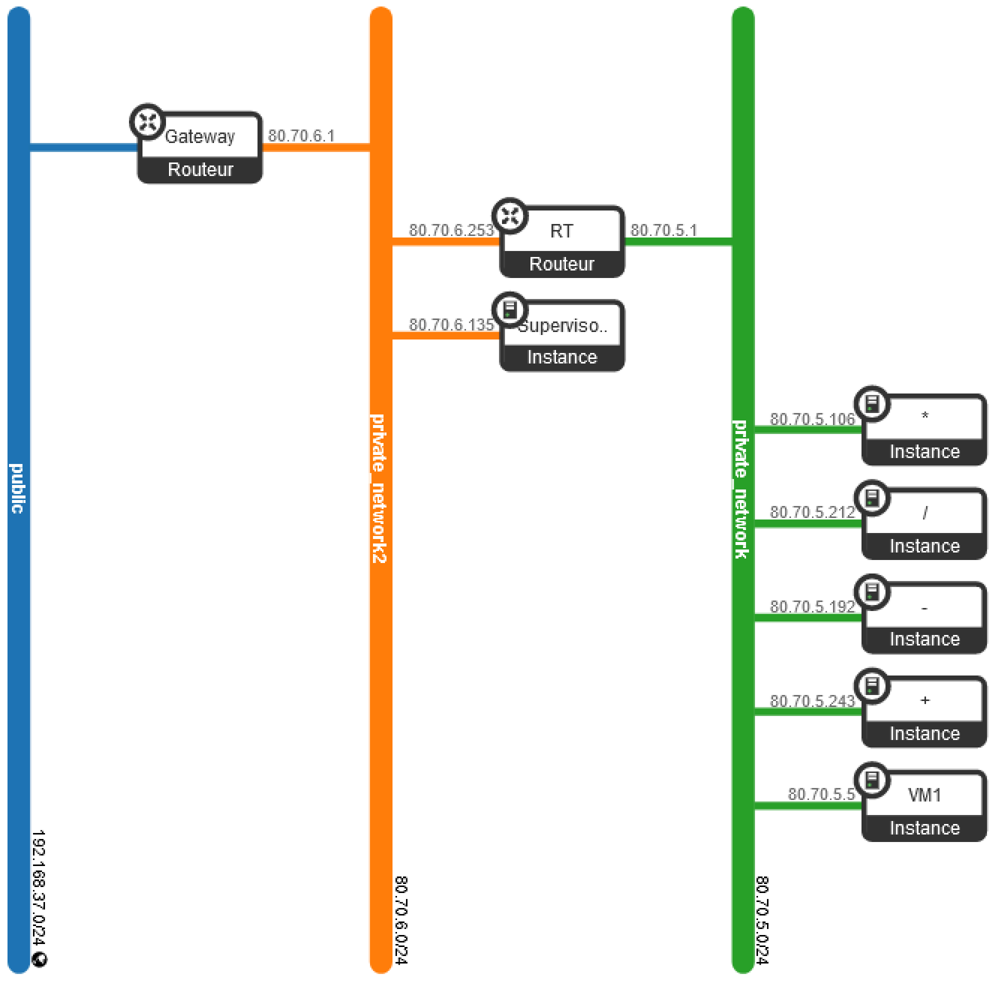
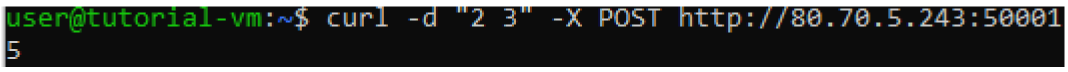


Figure 25: OpenStack network more secure configuration

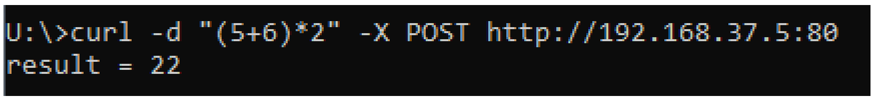
By adding a simple router and a route between the private network2, which contains the calculator service, and the private network, which contains the arithmetic operations.
We can now call the arithmetic operations from the supervisor:



```
user@tutorial-vm:~$ curl -d "2 3" -X POST http://80.70.5.243:50001
5
```

Figure 26: Snapshot of the arithmetic operation services from the supervisor

With the public address we linked to our supervisor in the last section and the route we created between the private network and the supervisor, we can now call the calculator service but no longer the arithmetic operations:



```
U:\>curl -d "(5+6)*2" -X POST http://192.168.37.5:80
result = 22
```

Figure 27: Snapshot of the calculator service from outside the network

3.4. Edge computing

With advancements in virtualization and computing paradigms such as cloud computing, service providers now offer robust platforms for hosting and accessing services and applications from anywhere, at any time. These platforms can meet diverse computing needs, but for services requiring real-time interaction, cloud infrastructure must be closer to end-users. Edge computing addresses this need, allowing platforms to support real-time services that operate under strict delay requirements.

We will focus on an autonomous car example. The car has four services:

- **Navigation**: Controls the acceleration, direction, and braking of the vehicle.
- **DataCycling**: Processes the data from perception information.
- **Perception**: Gather information from the car's sensors.
- **Decision**: Makes decisions and sends commands to the navigation (brake, accelerate, turn, etc.).

The last service (Decision) can be virtualized in the cloud to reduce the car's processing load. However, the car needs the information in real time, as if the service were running locally. To maintain low latency, the service will be moved from one point of deployment (PoD) to another to keep latency as low as possible.

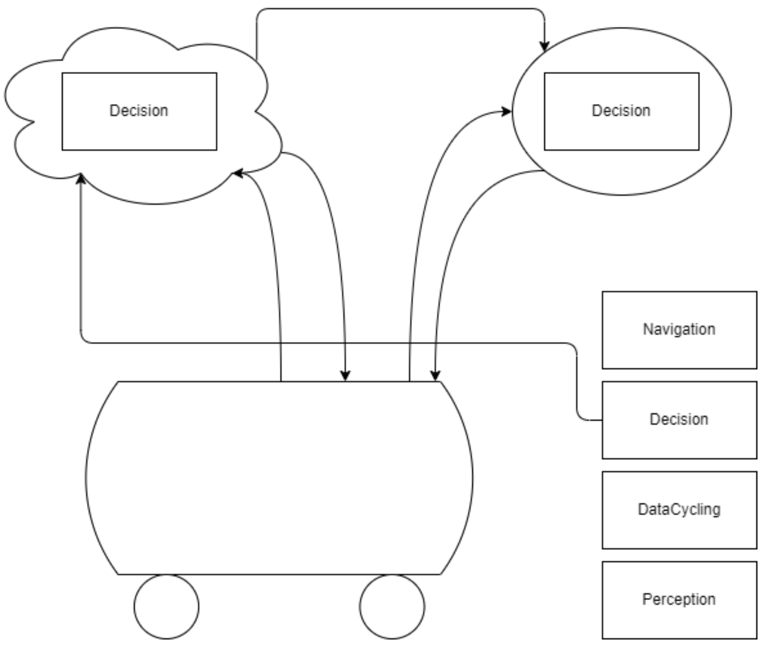


Figure 28: Schematic that represents our case of study

To test this, we first need to set up our network. The network is composed of:

- **Two worker nodes**: A Docker container will be deployed on one of them, simulating the service running in edge computing.
- **A master node**: Based on the latency to access the service, the master node will migrate the service to a Docker container running on the other worker node.

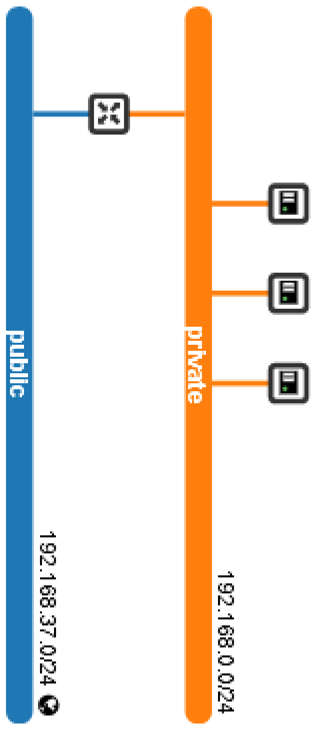
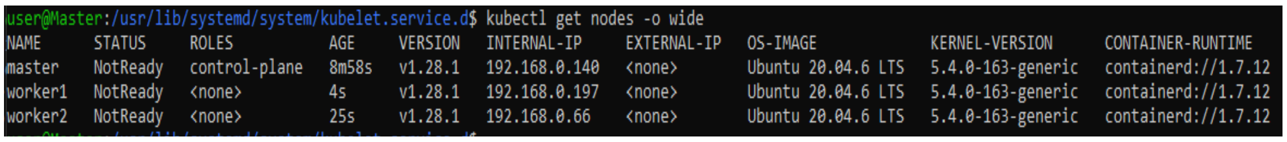


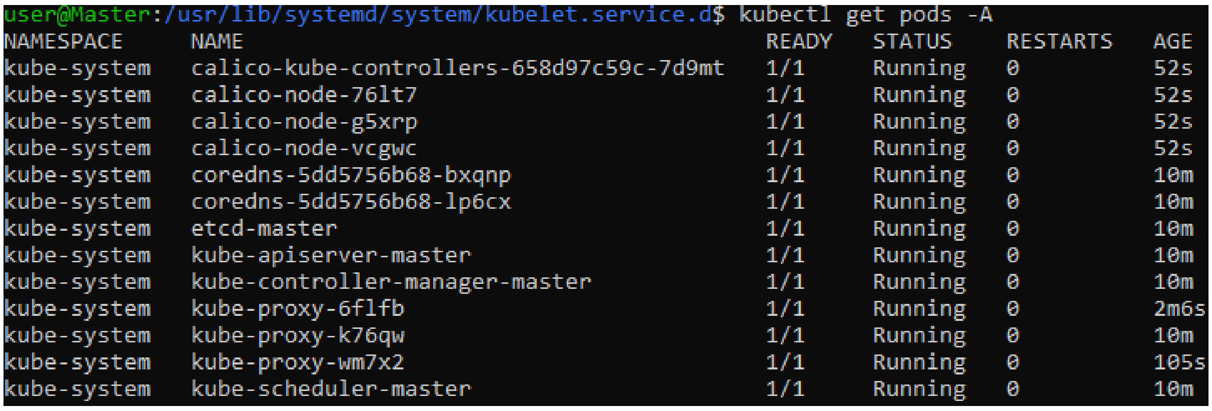
Figure 29: OpenStack network to simulate our case of study

The master node will manage our cluster, which consists of two worker nodes that will operate under its control.



```
user@Master:/usr/lib/systemd/system/kubelet.service.d$ kubectl get nodes -o wide
NAME      STATUS     ROLES           AGE     VERSION   INTERNAL-IP     EXTERNAL-IP   OS-IMAGE             KERNEL-VERSION      CONTAINER-RUNTIME
master    NotReady   control-plane   8m58s   v1.28.1   192.168.0.140   <none>        Ubuntu 20.04.6 LTS   5.4.0-163-generic   containerd://1.7.12
worker1   NotReady   <none>          4s      v1.28.1   192.168.0.197   <none>        Ubuntu 20.04.6 LTS   5.4.0-163-generic   containerd://1.7.12
worker2   NotReady   <none>          25s     v1.28.1   192.168.0.66    <none>        Ubuntu 20.04.6 LTS   5.4.0-163-generic   containerd://1.7.12
```

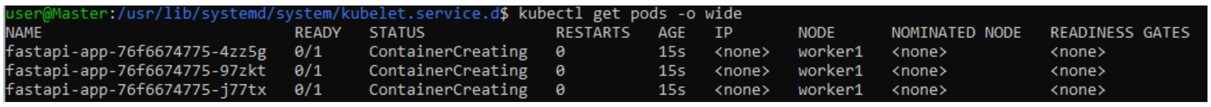
Figure 30: Snapshot of the list of all the nodes



```
user@Master:/usr/lib/systemd/system/kubelet.service.d$ kubectl get pods -A
NAMESPACE     NAME                                       READY   STATUS    RESTARTS   AGE
kube-system   calico-kube-controllers-658d97c59c-7d9mt   1/1     Running   0          52s
kube-system   calico-node-76lt7                          1/1     Running   0          52s
kube-system   calico-node-g5xrp                          1/1     Running   0          52s
kube-system   calico-node-vcgwc                          1/1     Running   0          52s
kube-system   coredns-5dd5756b68-bxqnp                   1/1     Running   0          10m
kube-system   coredns-5dd5756b68-lp6cx                   1/1     Running   0          10m
kube-system   etcd-master                                1/1     Running   0          10m
kube-system   kube-apiserver-master                      1/1     Running   0          10m
kube-system   kube-controller-manager-master             1/1     Running   0          10m
kube-system   kube-proxy-6flfb                           1/1     Running   0          2m6s
kube-system   kube-proxy-k76qw                           1/1     Running   0          10m
kube-system   kube-proxy-wm7x2                           1/1     Running   0          105s
kube-system   kube-scheduler-master                      1/1     Running   0          10m
```

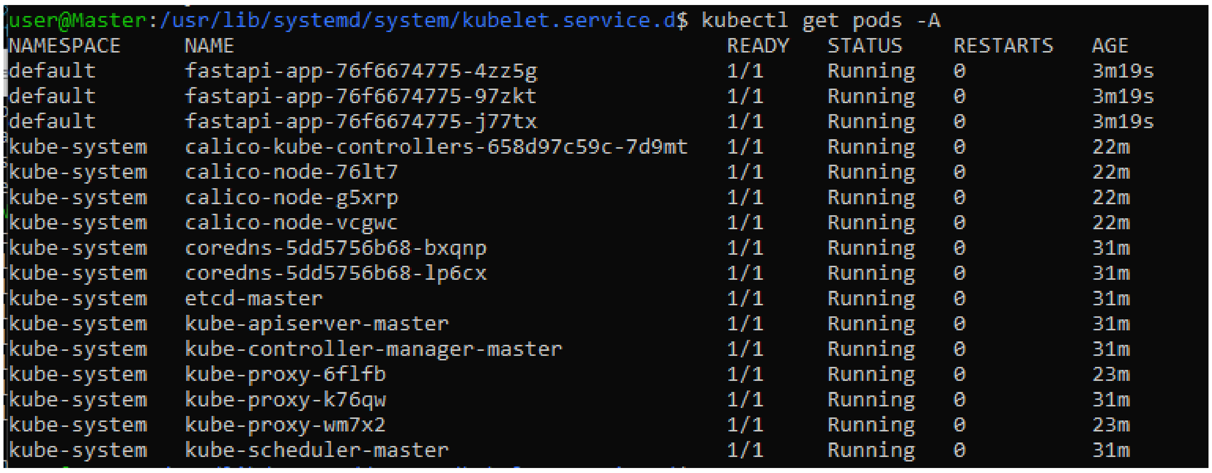
Figure 31: Snapshot of the PoDs running on our cluster

As you can see on the screen below, we have created services that run on our worker node 1:



```
user@Master:/usr/lib/systemd/system/kubelet.service.d$ kubectl get pods -o wide
NAME                           READY   STATUS              RESTARTS   AGE   IP       NODE      NOMINATED NODE   READINESS GATES
fastapi-app-76f6674775-4zz5g   0/1     ContainerCreating   0          15s   <none>   worker1   <none>           <none>
fastapi-app-76f6674775-97zkt   0/1     ContainerCreating   0          15s   <none>   worker1   <none>           <none>
fastapi-app-76f6674775-j77tx   0/1     ContainerCreating   0          15s   <none>   worker1   <none>           <none>
```

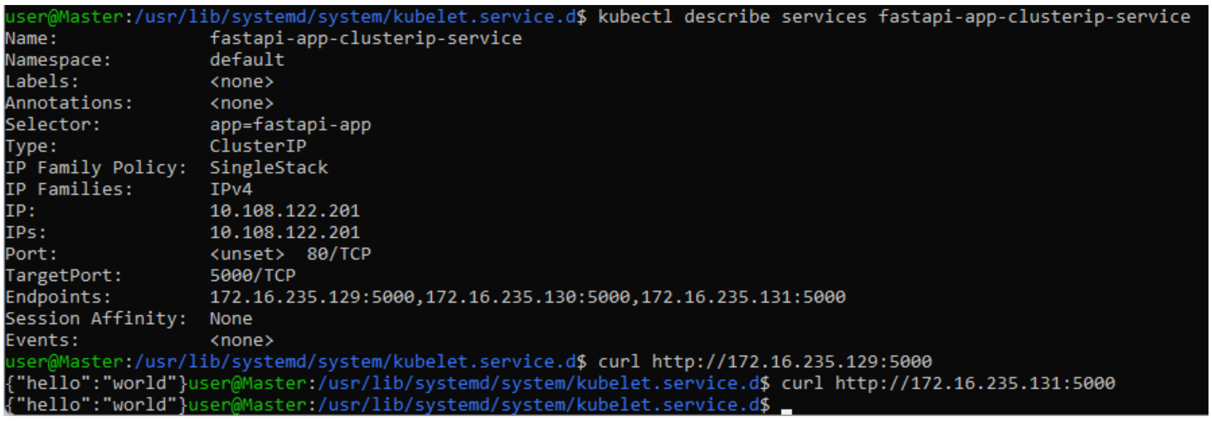
Figure 32: Snapshot of the PoDs just created



```
user@Master:/usr/lib/systemd/system/kubelet.service.d$ kubectl get pods -A
NAMESPACE     NAME                                       READY   STATUS    RESTARTS   AGE
default       fastapi-app-76f6674775-4zz5g               1/1     Running   0          3m19s
default       fastapi-app-76f6674775-97zkt               1/1     Running   0          3m19s
default       fastapi-app-76f6674775-j77tx               1/1     Running   0          3m19s
kube-system   calico-kube-controllers-658d97c59c-7d9mt   1/1     Running   0          22m
kube-system   calico-node-76lt7                          1/1     Running   0          22m
kube-system   calico-node-g5xrp                          1/1     Running   0          22m
kube-system   calico-node-vcgwc                          1/1     Running   0          22m
kube-system   coredns-5dd5756b68-bxqnp                   1/1     Running   0          31m
kube-system   coredns-5dd5756b68-lp6cx                   1/1     Running   0          31m
kube-system   etcd-master                                1/1     Running   0          31m
kube-system   kube-apiserver-master                      1/1     Running   0          31m
kube-system   kube-controller-manager-master             1/1     Running   0          31m
kube-system   kube-proxy-6flfb                           1/1     Running   0          23m
kube-system   kube-proxy-k76qw                           1/1     Running   0          31m
kube-system   kube-proxy-wm7x2                           1/1     Running   0          23m
kube-system   kube-scheduler-master                      1/1     Running   0          31m
```

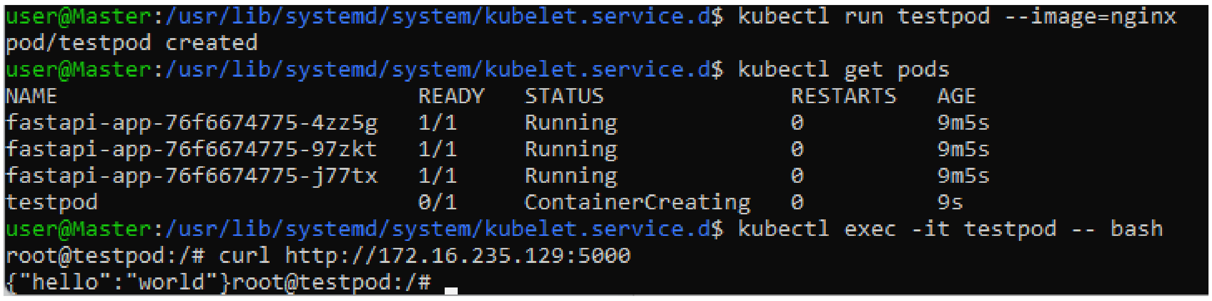
Figure 33: Snapshot of the new PoDs running on our cluster

We can communicate with the services running on worker 1 and see that is working well because it is printing “Hello World”:



```
user@Master:/usr/lib/systemd/system/kubelet.service.d$ 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:                10.108.122.201
IPs:               10.108.122.201
Port:              <unset>  80/TCP
TargetPort:        5000/TCP
Endpoints:         172.16.235.129:5000,172.16.235.130:5000,172.16.235.131:5000
Session Affinity:  None
Events:            <none>
user@Master:/usr/lib/systemd/system/kubelet.service.d$ curl http://172.16.235.129:5000
{"hello":"world"}user@Master:/usr/lib/systemd/system/kubelet.service.d$ curl http://172.16.235.131:5000
{"hello":"world"}user@Master:/usr/lib/systemd/system/kubelet.service.d$ _
```

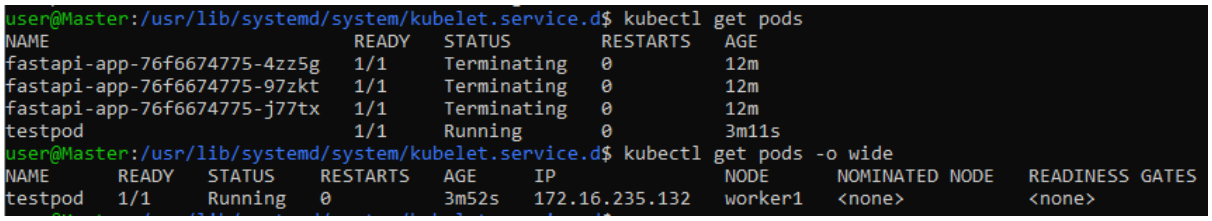
Figure 34: Snapshot of calling services from Master



```
user@Master:/usr/lib/systemd/system/kubelet.service.d$ kubectl run testpod --image=nginx
pod/testpod created
user@Master:/usr/lib/systemd/system/kubelet.service.d$ kubectl get pods
NAME                           READY   STATUS              RESTARTS   AGE
fastapi-app-76f6674775-4zz5g   1/1     Running             0          9m5s
fastapi-app-76f6674775-97zkt   1/1     Running             0          9m5s
fastapi-app-76f6674775-j77tx   1/1     Running             0          9m5s
testpod                        0/1     ContainerCreating   0          9s
user@Master:/usr/lib/systemd/system/kubelet.service.d$ kubectl exec -it testpod -- bash
root@testpod:/# curl http://172.16.235.129:5000
{"hello":"world"}root@testpod:/# _
```

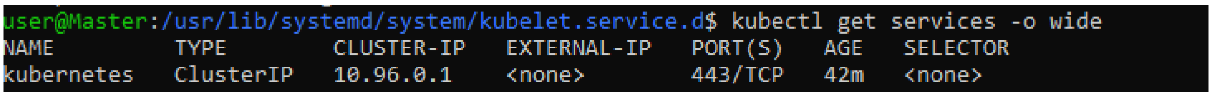
Figure 35: Snapshot of calling services from another PoD

We can also delete the PoDs we have created and only kubernetes still exist:



```
user@Master:/usr/lib/systemd/system/kubelet.service.d$ kubectl get pods
NAME                           READY   STATUS        RESTARTS   AGE
fastapi-app-76f6674775-4zz5g   1/1     Terminating   0          12m
fastapi-app-76f6674775-97zkt   1/1     Terminating   0          12m
fastapi-app-76f6674775-j77tx   1/1     Terminating   0          12m
testpod                        1/1     Running       0          3m11s
user@Master:/usr/lib/systemd/system/kubelet.service.d$ kubectl get pods -o wide
NAME      READY   STATUS    RESTARTS   AGE     IP               NODE      NOMINATED NODE   READINESS GATES
testpod   1/1     Running   0          3m52s   172.16.235.132   worker1   <none>           <none>
```

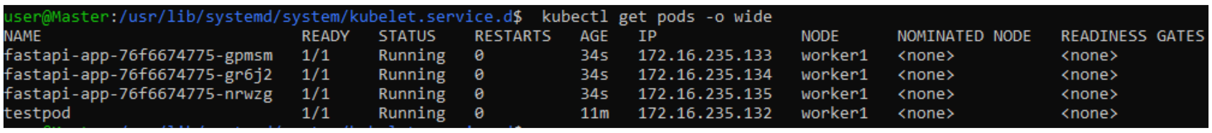
Figure 36: Snapshot of PoDs deleted



```
user@Master:/usr/lib/systemd/system/kubelet.service.d$ kubectl get services -o wide
NAME         TYPE        CLUSTER-IP   EXTERNAL-IP   PORT(S)   AGE   SELECTOR
kubernetes   ClusterIP   10.96.0.1    <none>        443/TCP   42m   <none>
```

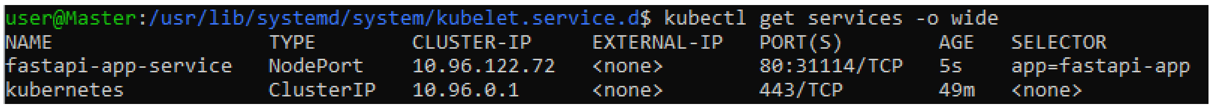
Figure 37: Snapshot of kubernetes being the only one still running

We have now deployed the same service using “kubectl apply -f ./kube_service/NodePort” just after deleting the ones created before. It is a more simple way to deploy the services wanted:



```
user@Master:/usr/lib/systemd/system/kubelet.service.d$  kubectl get pods -o wide
NAME                           READY   STATUS    RESTARTS   AGE   IP               NODE      NOMINATED NODE   READINESS GATES
fastapi-app-76f6674775-gpmsm   1/1     Running   0          34s   172.16.235.133   worker1   <none>           <none>
fastapi-app-76f6674775-gr6j2   1/1     Running   0          34s   172.16.235.134   worker1   <none>           <none>
fastapi-app-76f6674775-nrwzg   1/1     Running   0          34s   172.16.235.135   worker1   <none>           <none>
testpod                        1/1     Running   0          11m   172.16.235.132   worker1   <none>           <none>
```

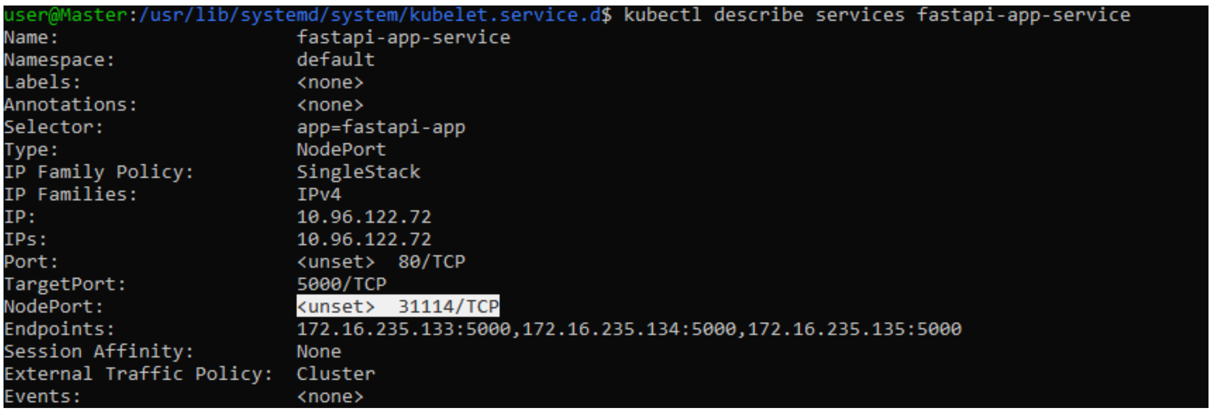
Figure 38: Snapshot of new PoDs just created



```
user@Master:/usr/lib/systemd/system/kubelet.service.d$ kubectl get services -o wide
NAME                  TYPE        CLUSTER-IP     EXTERNAL-IP   PORT(S)        AGE   SELECTOR
fastapi-app-service   NodePort    10.96.122.72   <none>        80:31114/TCP   5s    app=fastapi-app
kubernetes            ClusterIP   10.96.0.1      <none>        443/TCP        49m   <none>
```

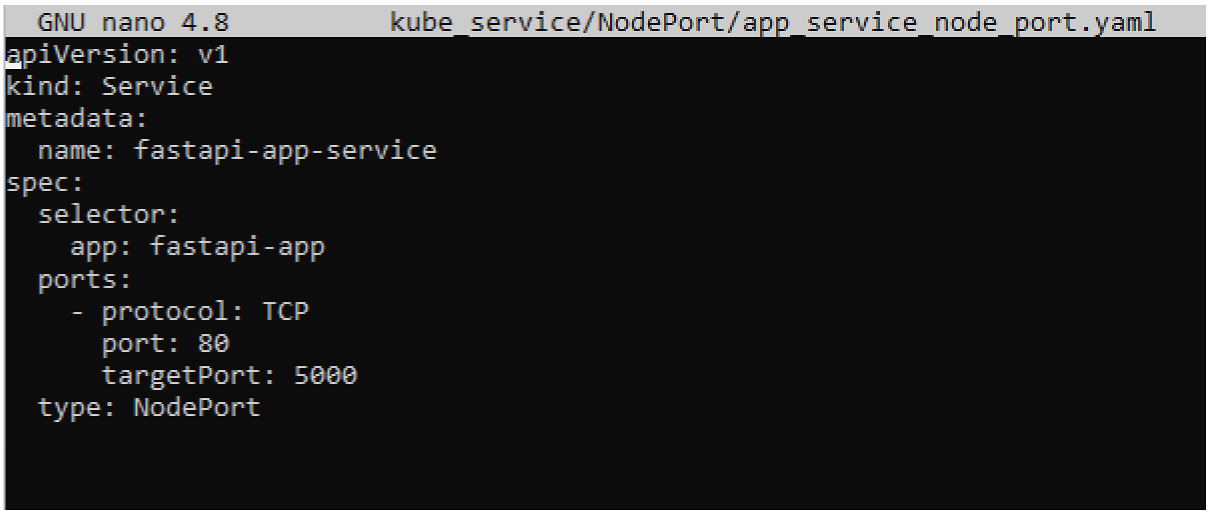
Figure 39: Snapshot of the new PoDs running on our cluster

A port has been defined for our PoD automatically. It is why he does not appear in the yaml file:



```
user@Master:/usr/lib/systemd/system/kubelet.service.d$ 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:                       10.96.122.72
IPs:                      10.96.122.72
Port:                     <unset>  80/TCP
TargetPort:               5000/TCP
NodePort:                 <unset>  31114/TCP
Endpoints:                172.16.235.133:5000,172.16.235.134:5000,172.16.235.135:5000
Session Affinity:         None
External Traffic Policy:  Cluster
Events:                   <none>
```

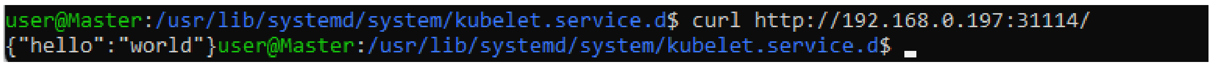
Figure 40: Snapshot of the service description



```
  GNU nano 4.8            kube_service/NodePort/app_service_node_port.yaml
apiVersion: v1
kind: Service
metadata:
  name: fastapi-app-service
spec:
  selector:
    app: fastapi-app
  ports:
    - protocol: TCP
      port: 80
      targetPort: 5000
  type: NodePort
```

Figure 41: Snapshot of the service's yaml file

We can, just like before, call the services:



```
user@Master:/usr/lib/systemd/system/kubelet.service.d$ curl http://192.168.0.197:31114/
{"hello":"world"}user@Master:/usr/lib/systemd/system/kubelet.service.d$ _
```

Figure 42: Snapshot of calling services from Master

Noël Jumin
Clément Gauché

4. Conclusion

In conclusion, this lab allowed us to gain hands-on experience with essential virtualization technologies, including VirtualBox and OpenStack. Working as a team, we successfully met the lab objectives by provisioning and configuring virtual machines and Docker containers, testing various networking modes, and automating operations through OpenStack's API. This practical approach enhanced our understanding of the critical role virtualization plays in cloud computing infrastructures. The insights gained from this lab are invaluable, particularly in terms of understanding how cloud environments are managed, secured, and optimized for performance.