INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES TOULOUSE

INNOVATIVE SMART SYSTEMS

Cloud and Edge Computing

HATIBI Mohammed Amine
Anka Soubaii Abdelmajid

5ISS

Lab 1 : Introduction to Cloud Hypervisors

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

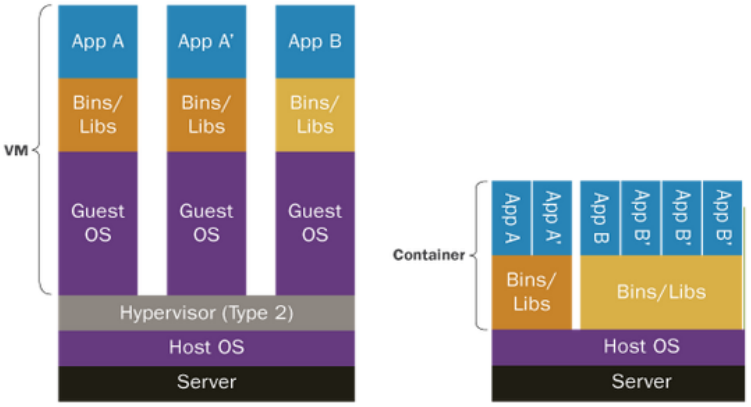


Figure 1: VM vs CT

	Virtual Machine	Container
Virtualizing cost (memory size, and CPU)	Emulate an entire machine at the hardware level, making them more expensive to virtualize.	Lightweight and cost-effective in terms of memory and CPU resources.
Usage of CPU, memory and network for a given application	VMs use a lot more CPU and memory because of their own guest OS.	Containers enable the use of microservices, which reduces CPU and memory usage.
Security for the application	Offer fully isolated virtual machines, which are generally more secure.	Provide process-level isolation and share the underlying OS, potentially exposing some security vulnerabilities.
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.
Tooling for the continuous integration support	Have fewer widely adopted development tools by default.	Typically come with development kits and benefit from widespread development tools.

2. Similarities and differences between the existing CT types

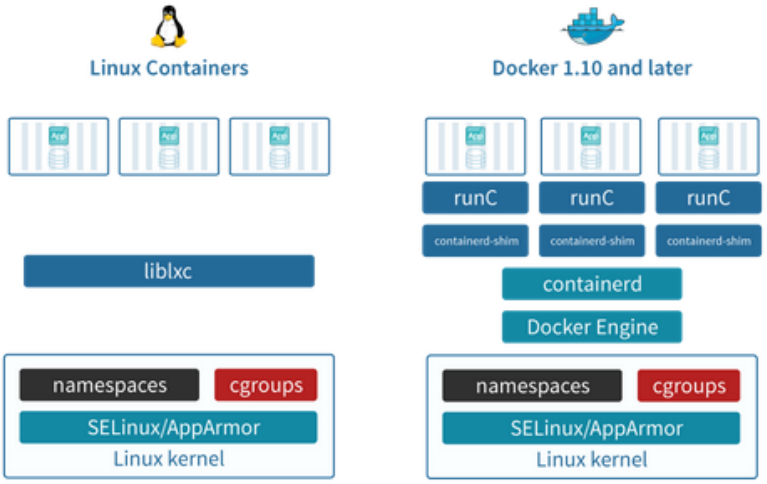
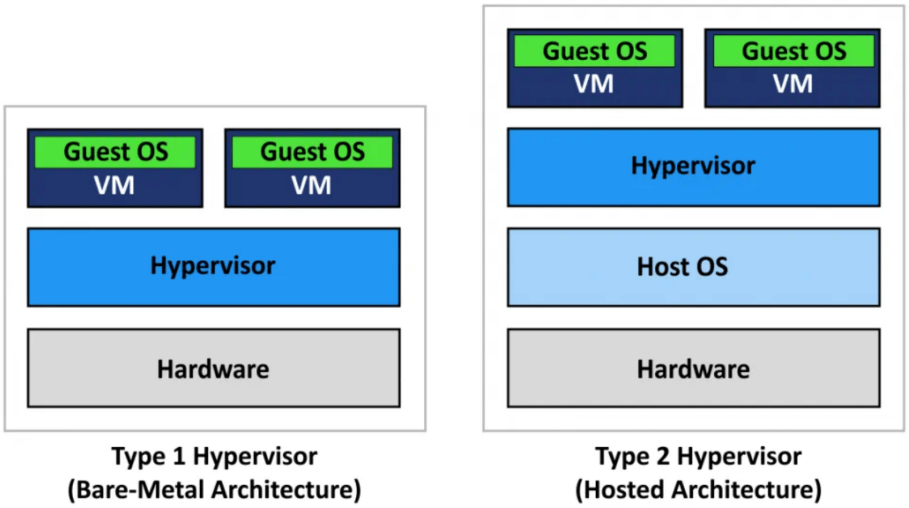


Figure 2 : Linux Lxc vs Docker

	Linux LXC	Docker
Application isolation and ressources	Act more like a light-VM, and is therefore more self-contained. It isn't contained to a single application, and has a higher level of isolation than docker	Works on the application level, and is less isolated. It shares ressources between containers, and also use more resources since it is more complex added to the LXC base
Containerization level	LXC is virtualization on OS level. It provides a means to run multiple Linux distributions on a single host.	Docker is based on LXC, and adds higher level functionalities. Docker provide andengine to supervise containerization, and virtualises on the application level.
Tooling	As LXC is less popular, the documentation isn't as thorough as docker. The LXC project provides base OS container templates and a comprehensive set of tools for container lifecycle management, but doesn't provide nearly as many development tools as docker.	Docker is running on application level and tools are centering on the Docker CLI. Docker Hub provides a public image access for frequently used applications.

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



- In the diagram, it's evident that Type 1 and Type 2 hypervisors differ in the levels at which they operate. Type 1 hypervisors function at the hardware level, bypassing the need for an installed operating system (OS). Consequently, they often deliver superior efficiency and performance. Moreover, they tend to be more secure as they are not susceptible to OS-specific security vulnerabilities.

- Conversely, Type 2 hypervisors operate within an underlying host OS, relying on it to manage certain aspects, including networking, resource allocation, CPU usage, and storage. While Type 2 hypervisors offer compatibility with a wider range of hardware, they can introduce latency due to their interaction with the host OS.

An example of a Type 2 hypervisor is VirtualBox, which can be installed on a variety of machines. On the other hand, OpenStack, a cloud management system, provides the flexibility to choose between Type 1 and Type 2 hypervisors based on your specific requirements.

Practical part (objectives 4 to 5)

First part: Creating and configuring a VM

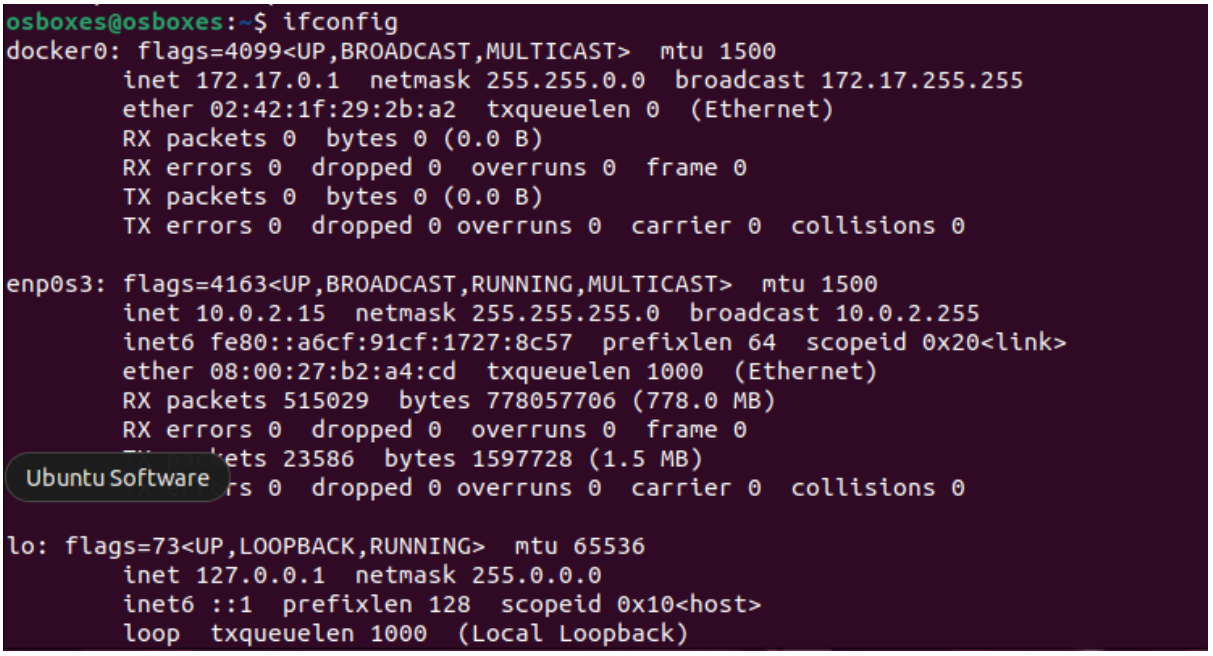
In this section, we will set up a VirtualBox virtual machine (VM) in NAT mode and establish a network configuration that enables two-way communication with external networks. Our initial step involves copying the virtual hard drive containing the Ubuntu operating system and extracting the archives. Next, using VirtualBox, a Type 2 hypervisor, we will create a VM with a Linux/Ubuntu 64-bit configuration. We will allocate essential resources to this VM, such as 1024 MB of RAM, a provided hard drive, and up to 1 CPU core.

Additionally, we will associate peripheral devices like a network board, CD readers, and USB ports. We will configure the network board in NAT mode, which will automatically assign a private IP address to the VM.

Once these configurations are complete, we can initiate the VM using the provided authentication details: our username will be "osboxes," and our password will be "osboxes.org." This setup lays the foundation for seamless communication between our VM and external networks, a crucial aspect of various virtualization and networking scenarios.

Second part: Testing the VM connectivity

In this part of the exercise, we will identify the IP address assigned to the VM using the 'ifconfig' command in the Linux command line and compare it to the host's IP address obtained using the 'ip config' command. This allows us to observe the network configuration and any disparities between the VM and host addresses.



```
U:\>ipconfig

Configuration IP de Windows


Carte Ethernet Ethernet :

   Suffixe DNS propre à la connexion. . . : insa-toulouse.fr
   Adresse IPv6 de liaison locale. . . . .: fe80::86ec:8d41:c26d:80d%7
   Adresse IPv4. . . . . . . . . . . . . .: 10.1.5.87
   Masque de sous-réseau. . . . . . . . . : 255.255.0.0
   Passerelle par défaut. . . . . . . . . : 10.1.0.254

Carte Ethernet VirtualBox Host-Only Network :

   Suffixe DNS propre à la connexion. . . :
   Adresse IPv6 de liaison locale. . . . .: fe80::e1d1:f8bb:b660:a0bd%5
   Adresse IPv4. . . . . . . . . . . . . .: 192.168.56.1
   Masque de sous-réseau. . . . . . . . . : 255.255.255.0
   Passerelle par défaut. . . . . . . . . :
```

- **IP Address of the VM : 10.0.2.15**
- **IP Address of the Host : 10.1.5.87**

The two addresses are different because the VM's IP address is an IP address emulation and not the host IP address.

- **IP Address of the neighboors : 10.1.5.39**

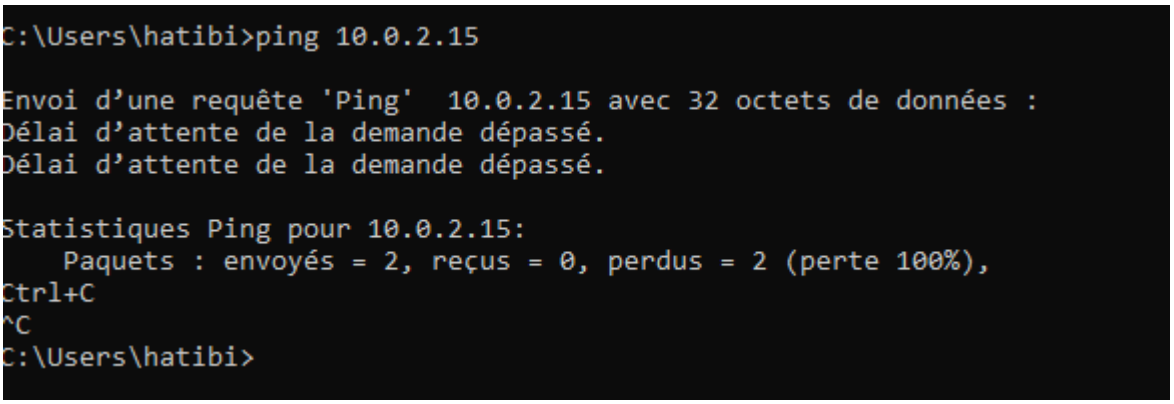
Subsequently, we will use the 'ping' command to perform connectivity checks.

Firstly, we will assess the VM's connectivity to external networks

```
user@tutorial-vm:~$ ping youtube.com
PING youtube.com (142.251.37.238) 56(84) bytes of data.
64 bytes from mrs09s16-in-f14.1e100.net (142.251.37.238): icmp_seq=1 ttl=114 time=6.21 ms
64 bytes from mrs09s16-in-f14.1e100.net (142.251.37.238): icmp_seq=2 ttl=114 time=6.39 ms
64 bytes from mrs09s16-in-f14.1e100.net (142.251.37.238): icmp_seq=3 ttl=114 time=6.35 ms
^C
--- youtube.com ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2003ms
rtt min/avg/max/mdev = 6.210/6.318/6.391/0.077 ms
```

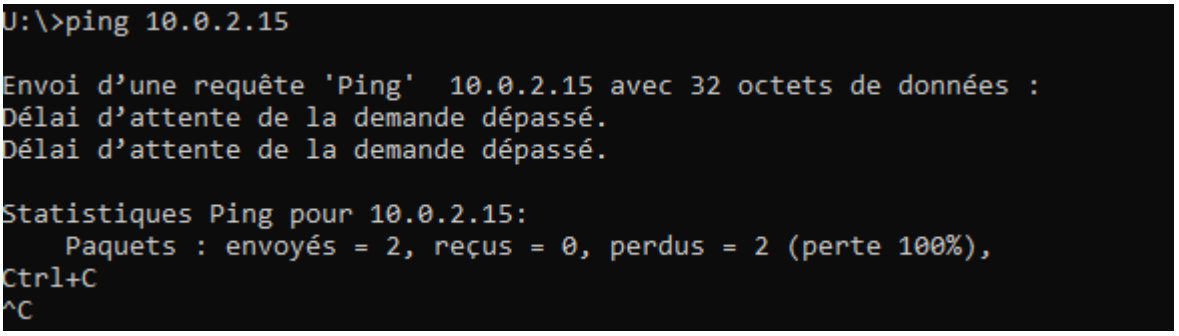
The ping is possible to the outside because even if it is not a routable address, but thanks to NAT, we can connect to the outside and receive the ping back.

Following that, we will investigate the connectivity from our host neighbor to our hosted VM.



—-> Ping is impossible. Because the IP address is not routable

Lastly, we will examine the connectivity from our host to the hosted VM.



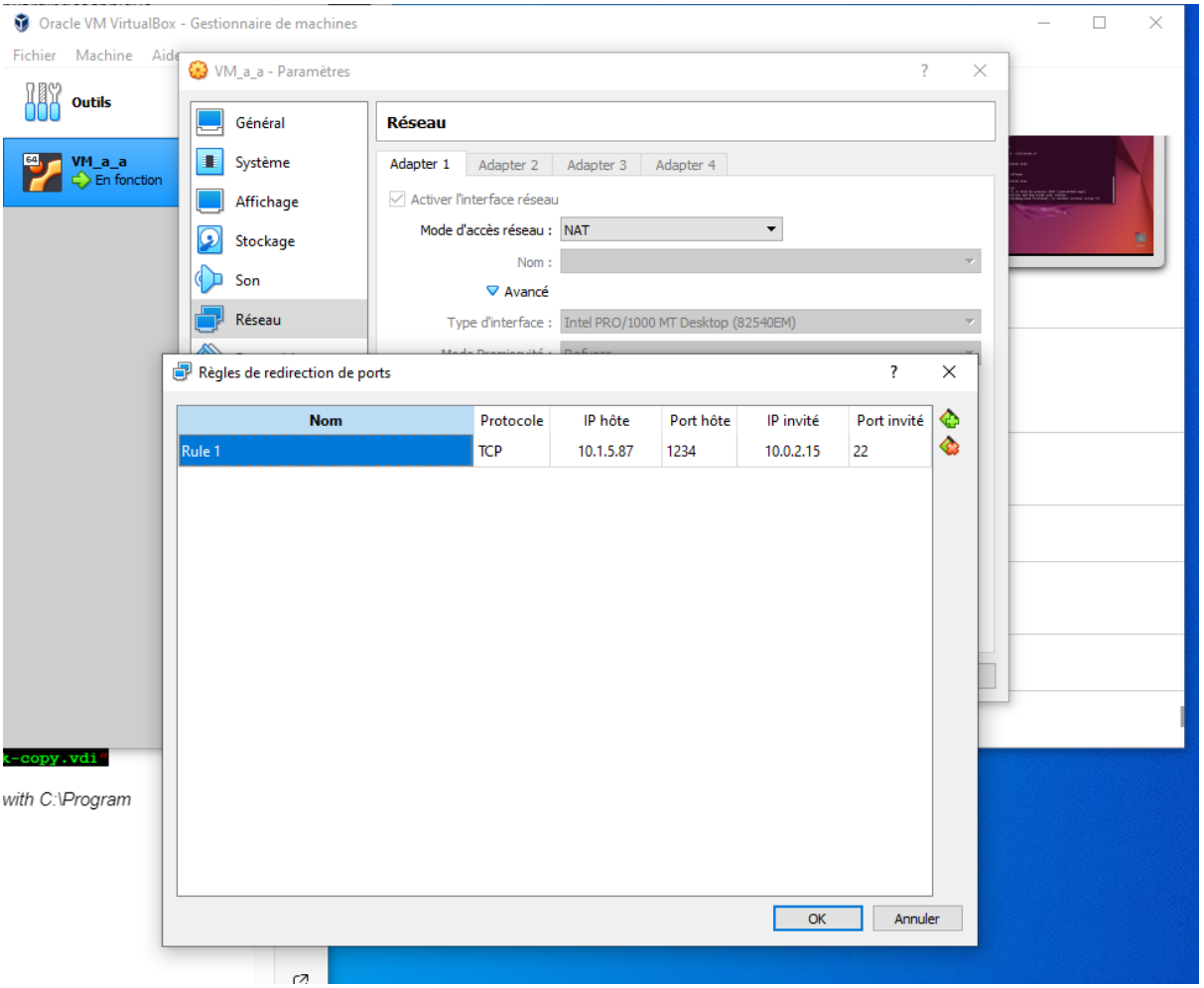
—-> Ping is impossible. This is normal. It is a NAT connection. The same applies to the previous ping.

This series of checks will provide insights into the networking setup and potential connectivity issues, enabling us to fine-tune the configuration for optimal communication between the VM and the external world, as well as between different network entities.

Third part: Set up the “missing” connectivity

In this section, our goal is to address the problems identified in the previous part, enabling the virtual machine to be reachable from external networks while still using the NAT protocol. To achieve this, we will implement the Port Forwarding technique. To do so, we'll need to create a new forwarding rule in the configuration of the network interface management provided by VirtualBox within the VM settings:

In our scenario, this technique primarily involves rerouting incoming requests received by the host on a designated port, which in our case is port 1234 (chosen above 1024). These incoming requests are then directed to a specific port on the virtual machine, specifically port 22, which is the SSH port. This approach enables us to initiate an SSH connection to the virtual machine from an external source by sending the request to the redirected port.



We configured PuTTy as an SSH client, then we managed to connect to the VM using SSH. This is the only way to access the VM from the outside, since it does not have a routable IP address.

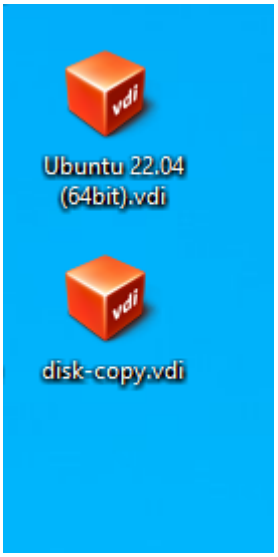
Fourth part: VM duplication

Now we want to create a new (clone) VM with the same disk file. We run the following command:

```
VBoxManage clonemedium "chemin\vers\disk.vdi" "chemin\vers\disk-copy.vdi"
```

```
C:\Program Files\Oracle\VirtualBox>VBoxManage clonemedium "U:\Windows\Bureau\Ubuntu 22.04 (64bit).vdi" "U:\Windows\Bu
u\disk-copy.vdi"
0%...10%...20%...30%...40%...50%...60%...70%...80%...90%...100%
Clone medium created in format 'VDI'. UUID: 3f8ba9f7-46c2-49fa-9a7a-d8eacaa69358
```

The copy of the disk has been done successfully. Now we create the new VM with the disk-copy.vmdk file:



Fifth part: Docker containers provisioning

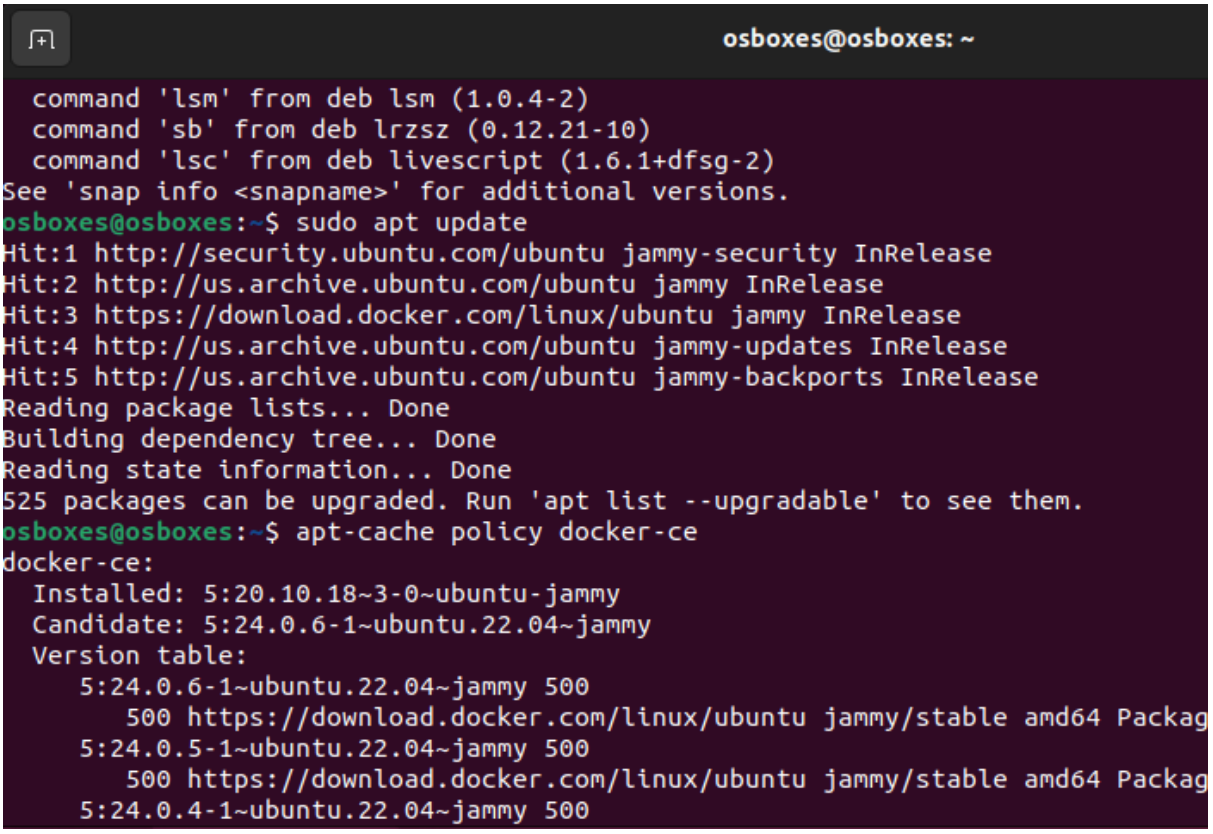
In this section, we will address the provisioning of Docker containers. For practical convenience, we will deploy the Docker environment on VirtualBox VMs, as this approach grants us full administrative privileges over these VMs. While it is technically feasible to provision containers directly over VMs, it is a less common practice in the real-world scenario. Typically, containers are deployed directly on bare-metal physical hosts, similar to the way VMs are deployed.

We will first install the Docker Engine on the VM then we will use it to create and handle Docker containers.

Docker Engine Setup:

To ensure the successful addition of the new packages, it's crucial to update the existing list of packages using the command : `"sudo apt update "`

Subsequently, it's important to verify that the installation will be conducted from the Docker repository rather than the default Ubuntu repository, which can be confirmed by executing "`$ apt-cache policy docker-ce`".



Next, we install a few prerequisite packages which let *apt* use packages over HTTPS :

```
osboxes@osboxes:~$ sudo apt install apt-transport-https ca-certificates curl software-properties-common
Reading package lists... Done
Building dependency tree... Done
Reading state information... Done
The following additional packages will be installed:
  libcurl4 python3-software-properties software-properties-gtk ubuntu-advantage-tools
The following packages will be upgraded:
  apt-transport-https ca-certificates curl libcurl4 python3-software-properties
  software-properties-common software-properties-gtk ubuntu-advantage-tools
8 upgraded, 0 newly installed, 0 to remove and 522 not upgraded.
Need to get 306 kB/946 kB of archives.
After this operation, 1,431 kB disk space will be freed.
Do you want to continue? [Y/n] y
Get:1 http://us.archive.ubuntu.com/ubuntu jammy-updates/main amd64 ubuntu-advantage-tools amd64 29.4~22.04
[190 kB]
Get:2 http://us.archive.ubuntu.com/ubuntu jammy-updates/universe amd64 apt-transport-https all 2.4.10 [1,51
0 B]
Get:3 http://us.archive.ubuntu.com/ubuntu jammy-updates/main amd64 software-properties-common all 0.99.22.7
 [14.1 kB]
Get:4 http://us.archive.ubuntu.com/ubuntu jammy-updates/main amd64 software-properties-gtk all 0.99.22.7 [7
1.3 kB]
Get:5 http://us.archive.ubuntu.com/ubuntu jammy-updates/main amd64 python3-software-properties all 0.99.22.
7 [28.8 kB]
Fetched 306 kB in 1s (399 kB/s)
Preconfiguring packages ...
(Reading database ... 163610 files and directories currently installed.)
Preparing to unpack .../0-ca-certificates_20230311ubuntu0.22.04.1_all.deb ...
Unpacking ca-certificates (20230311ubuntu0.22.04.1) over (20211016) ...
Preparing to unpack .../1-ubuntu-advantage-tools_29.4~22.04_amd64.deb ...
Unpacking ubuntu-advantage-tools (29.4~22.04) over (27.7~22.04.1) ...
Preparing to unpack .../2-apt-transport-https_2.4.10_all.deb ...
Unpacking apt-transport-https (2.4.10) over (2.4.8) ...
Preparing to unpack .../3-curl_7.81.0-1ubuntu1.13_amd64.deb ...
Unpacking curl (7.81.0-1ubuntu1.13) over (7.81.0-1ubuntu1.4) ...
```

Then we add the GPG key for the official Docker repository to our system :

```
osboxes@osboxes:~$ curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo gpg --dearmor -o /usr/sha
re/keyrings/docker-archive-keyring.gpg
File '/usr/share/keyrings/docker-archive-keyring.gpg' exists. Overwrite? (y/N) y
osboxes@osboxes:~$
```

Upon observation, it should be noted that the docker-ce package is not currently installed, but the installation candidate is sourced from the Docker repository specifically tailored for Ubuntu 22.04 (jammy). To complete the installation, we execute the command "`$ sudo apt install docker-ce`"

```
osboxes@osboxes: ~
osboxes@osboxes:~$ sudo apt install docker-ce
Reading package lists... Done
Building dependency tree... Done
Reading state information... Done
Suggested packages:
  aufs-tools cgroupfs-mount | cgroup-lite
The following packages will be upgraded:
  docker-ce
1 upgraded, 0 newly installed, 0 to remove and 524 not upgraded.
Need to get 22.6 MB of archives.
After this operation, 8,496 kB of additional disk space will be used.
Get:1 https://download.docker.com/linux/ubuntu jammy/stable amd64 docker-ce amd
mmy [22.6 MB]
Fetched 22.6 MB in 1s (17.3 MB/s)
(Reading database ... 163747 files and directories currently installed.)
Preparing to unpack .../docker-ce_5%3a24.0.6-1~ubuntu.22.04~jammy_amd64.deb ...
Unpacking docker-ce (5:24.0.6-1~ubuntu.22.04~jammy) over (5:20.10.18~3-0~ubuntu
Setting up docker-ce (5:24.0.6-1~ubuntu.22.04~jammy) ...
Installing new version of config file /etc/default/docker ...
Installing new version of config file /etc/init.d/docker ...
Removing obsolete conffile /etc/init/docker.conf ...
osboxes@osboxes:~$
```

This series of commands ensures the proper recognition and installation of Docker from the designated repository, facilitating a smooth and accurate setup on the Ubuntu 22.04 system.

Docker should have been installed, the daemon started, and the process enabled to start on boot. As we can see, it's running.

```
● docker.service - Docker Application Container Engine
     Loaded: loaded (/lib/systemd/system/docker.service; enabled; vendor preset: enabled)
     Active: active (running) since Wed 2023-10-11 11:50:59 EDT; 2min 34s ago
TriggeredBy: ● docker.socket
       Docs: https://docs.docker.com
   Main PID: 15121 (dockerd)
      Tasks: 9
     Memory: 30.6M
        CPU: 595ms
     CGroup: /system.slice/docker.service
             └─15121 /usr/bin/dockerd -H fd:// --containerd=/run/containerd/containerd.sock

Oct 11 11:50:58 osboxes dockerd[15121]: time="2023-10-11T11:50:58.312637894-04:00" level=info msg="Starting>
Oct 11 11:50:58 osboxes dockerd[15121]: time="2023-10-11T11:50:58.330775521-04:00" level=info msg="detected>
Oct 11 11:50:58 osboxes dockerd[15121]: time="2023-10-11T11:50:58.757815109-04:00" level=info msg="[graphdr>
Oct 11 11:50:58 osboxes dockerd[15121]: time="2023-10-11T11:50:58.801910371-04:00" level=info msg="Loading >
Oct 11 11:50:59 osboxes dockerd[15121]: time="2023-10-11T11:50:59.297779463-04:00" level=info msg="Default >
Oct 11 11:50:59 osboxes dockerd[15121]: time="2023-10-11T11:50:59.369591056-04:00" level=info msg="Loading >
Oct 11 11:50:59 osboxes dockerd[15121]: time="2023-10-11T11:50:59.703107296-04:00" level=info msg="Docker d>
Oct 11 11:50:59 osboxes dockerd[15121]: time="2023-10-11T11:50:59.710979881-04:00" level=info msg="Daemon h>
Oct 11 11:50:59 osboxes dockerd[15121]: time="2023-10-11T11:50:59.861777591-04:00" level=info msg="API list>
Oct 11 11:50:59 osboxes systemd[1]: Started Docker Application Container Engine.
~
```

Docker Containers Provisioning

Once the Docker service is installed, we are ready to provision Docker nodes. Using the `docker` command consists of passing it a chain of options and commands followed by arguments.

- **We get an ubuntu image by executing this command :**

```
osboxes@osboxes:~$ 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
```

- **Then we execute an instance of the ubuntu image (CT1):**

```
osboxes@osboxes:~$ sudo docker start -i ct1
root@bc306bc8711a:/# ls
bin   dev  home  lib32  libx32  mnt  proc  run   srv  tmp  var
boot  etc  lib   lib64  media   opt  root  sbin  sys  usr
root@bc306bc8711a:/#
```

- Then we install the required connectivity testing tools :

```
root@bc306bc8711a:/#
root@bc306bc8711a:/# apt-get -y update && apt-get -y install net-tools iputils-ping
Hit:1 http://archive.ubuntu.com/ubuntu jammy InRelease
Get:2 http://archive.ubuntu.com/ubuntu jammy-updates InRelease [119 kB]
Get:3 http://security.ubuntu.com/ubuntu jammy-security InRelease [110 kB]
Get:4 http://archive.ubuntu.com/ubuntu jammy-backports InRelease [109 kB]
Get:5 http://archive.ubuntu.com/ubuntu jammy-updates/main amd64 Packages [1342 kB]
Get:6 http://archive.ubuntu.com/ubuntu jammy-updates/multiverse amd64 Packages [49.8 kB]
Get:7 http://archive.ubuntu.com/ubuntu jammy-updates/restricted amd64 Packages [1252 kB]
Get:8 http://archive.ubuntu.com/ubuntu jammy-updates/universe amd64 Packages [1269 kB]
Get:9 http://archive.ubuntu.com/ubuntu jammy-backports/universe amd64 Packages [28.1 kB]
Get:10 http://archive.ubuntu.com/ubuntu jammy-backports/main amd64 Packages [50.4 kB]
Get:11 http://security.ubuntu.com/ubuntu jammy-security/multiverse amd64 Packages [44.0 kB]
Get:12 http://security.ubuntu.com/ubuntu jammy-security/main amd64 Packages [1081 kB]
Get:13 http://security.ubuntu.com/ubuntu jammy-security/restricted amd64 Packages [1226 kB]
Get:14 http://security.ubuntu.com/ubuntu jammy-security/universe amd64 Packages [1004 kB]
Fetched 7684 kB in 3s (2426 kB/s)
Reading package lists... Done
Reading package lists... Done
Building dependency tree... Done
Reading state information... Done
iputils-ping is already the newest version (3:20211215-1).
net-tools is already the newest version (1.60+git20181103.0eebece-1ubuntu5).
0 upgraded, 0 newly installed, 0 to remove and 34 not upgraded.
root@bc306bc8711a:/#
```

- **We check the connectivity with the newly instantiated Docker:**
  - The docker IP is **172.17.0.1**

```
osboxes@osboxes:~$ ifconfig
docker0: flags=4099<UP,BROADCAST,MULTICAST>  mtu 1500
        inet 172.17.0.1  netmask 255.255.0.0  broadcast 172.17.255.255
        ether 02:42:1f:29:2b:a2  txqueuelen 0  (Ethernet)
        RX packets 0  bytes 0 (0.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 0  bytes 0 (0.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

- **We ping an Internet resource from Docker (Youtube as an example)**

```
root@bc306bc8711a:/# ping youtube.com
PING youtube.com (172.217.18.46) 56(84) bytes of data.
64 bytes from ham02s12-in-f14.1e100.net (172.217.18.46): icmp_seq=1 ttl=113 time=6.69 ms
64 bytes from ham02s12-in-f14.1e100.net (172.217.18.46): icmp_seq=2 ttl=113 time=7.05 ms
64 bytes from mrs08s01-in-f14.1e100.net (172.217.18.46): icmp_seq=3 ttl=113 time=6.97 ms
64 bytes from mrs08s01-in-f14.1e100.net (172.217.18.46): icmp_seq=4 ttl=113 time=6.85 ms
64 bytes from ham02s12-in-f46.1e100.net (172.217.18.46): icmp_seq=5 ttl=113 time=6.96 ms
64 bytes from ham02s12-in-f46.1e100.net (172.217.18.46): icmp_seq=6 ttl=113 time=6.91 ms
```

—> As we can see below, the Ping works !

- **We ping the VM from Docker**

```
root@bc306bc8711a:/# 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=0.062 ms
64 bytes from 10.0.2.15: icmp_seq=2 ttl=64 time=0.119 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
^C
--- 10.0.2.15 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3066ms
rtt min/avg/max/mdev = 0.062/0.086/0.119/0.020 ms
root@bc306bc8711a:/#
```

—> The Ping works successfully.

- **We the docker from the VM**

```
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=0.055 ms
64 bytes from 172.17.0.1: icmp_seq=2 ttl=64 time=0.070 ms
64 bytes from 172.17.0.1: icmp_seq=3 ttl=64 time=0.069 ms
^C
--- 172.17.0.1 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2052ms
rtt min/avg/max/mdev = 0.055/0.064/0.070/0.006 ms
osboxes@osboxes:~$
```

—> It also works !

Conclusion: Provide further detail on the results obtained. As outlined in the theoretical section, the key distinction between virtual machines (VMs) and containers (Docker) lies in their virtualization approach. While VMs virtualize at the hardware level, containers operate at the operating system level. The practical implication of this distinction is evident in the results, where all pings are achievable without the necessity of employing SSH technology for external accessibility, a requirement often associated with VMs. Notably, containers exhibit the capability to seamlessly access both the resources and network of their host.

- **We execute a new instance (ct2) of the ubuntu Docker :**

```
osboxes@osboxes:~$ sudo docker run --name ct2 -p 2223:22 -it ubuntu
[sudo] password for osboxes:
root@98014186962f:/# apt -get -y update && apt install nano
E: Command line option 'g' [from -get] is not understood in combination with the other options.
root@98014186962f:/# apt-get -y update && apt install nano
Get:1 http://security.ubuntu.com/ubuntu jammy-security InRelease [110 kB]
Get:2 http://archive.ubuntu.com/ubuntu jammy InRelease [270 kB]
Get:3 http://archive.ubuntu.com/ubuntu jammy-updates InRelease [119 kB]
Get:4 http://archive.ubuntu.com/ubuntu jammy-backports InRelease [109 kB]
Get:5 http://security.ubuntu.com/ubuntu jammy-security/universe amd64 Packages [1004 kB]
Get:6 http://security.ubuntu.com/ubuntu jammy-security/restricted amd64 Packages [1226 kB]
Get:7 http://security.ubuntu.com/ubuntu jammy-security/multiverse amd64 Packages [44.0 kB]
Get:8 http://security.ubuntu.com/ubuntu jammy-security/main amd64 Packages [1081 kB]
Get:9 http://archive.ubuntu.com/ubuntu jammy/main amd64 Packages [1792 kB]
Get:10 http://archive.ubuntu.com/ubuntu jammy/restricted amd64 Packages [164 kB]
Get:11 http://archive.ubuntu.com/ubuntu jammy/universe amd64 Packages [17.5 MB]
Get:12 http://archive.ubuntu.com/ubuntu jammy/multiverse amd64 Packages [266 kB]
Get:13 http://archive.ubuntu.com/ubuntu jammy-updates/restricted amd64 Packages [1252 kB]
Get:14 http://archive.ubuntu.com/ubuntu jammy-updates/multiverse amd64 Packages [49.8 kB]
Get:15 http://archive.ubuntu.com/ubuntu jammy-updates/universe amd64 Packages [1269 kB]
Get:16 http://archive.ubuntu.com/ubuntu jammy-updates/main amd64 Packages [1342 kB]
Get:17 http://archive.ubuntu.com/ubuntu jammy-backports/main amd64 Packages [50.4 kB]
Get:18 http://archive.ubuntu.com/ubuntu jammy-backports/universe amd64 Packages [28.1 kB]
Fetched 27.6 MB in 5s (5407 kB/s)
Reading package lists... Done
Reading package lists... Done
Building dependency tree... Done
Reading state information... Done
Suggested packages:
```

- **We Install nano on Ct2**
- **The Ct2 is obtained by running this command `$ sudo docker ps`**

—> The Ct2 ID is `98014186962f`

- **Then we make a snapshot of Ct2 :**

```
osboxes@osboxes:~$ sudo docker commit 98014186962f photo:01
sha256:04006735b41eae542d45a5c7a3bb1fec9b544d92a7e0a4afa13a1c72b955f36f
osboxes@osboxes:~$
```

- **We Stop and terminate CT2 by executing those commands :**

```
osboxes@osboxes:~$ sudo docker stop ct2
ct2
osboxes@osboxes:~$
```

```
osboxes@osboxes:~$ sudo docker rm 98014186962f
98014186962f
osboxes@osboxes:~$
```

- **We list the available Docker images in the VM**

```
osboxes@osboxes:~$ sudo docker images
REPOSITORY   TAG       IMAGE ID       CREATED         SIZE
photo        01        04006735b41e   3 minutes ago   124MB
ubuntu       latest    3565a89d9e81   2 weeks ago     77.8MB
ubuntu       <none>    216c552ea5ba   12 months ago   77.8MB
osboxes@osboxes:~$
```

- **We execute a new instance (CT3) form the snapshot that we previously created from CT2**

```
osboxes@osboxes:~$ sudo docker run --name ct3 -it photo:01
root@4d2dc59ad5da:/#
```

We still have nano installed on CT3. This is because containers use shared resources on the host machine. This means that if one container is using a service on the host machine, another container on that machine will have the rights to use that service. When we installed nano in Container2, we uploaded the files to the shared resources and said that Container2 had the right to access them. When we created the Snapshot, we copied these rights and our new container had the right to access the files and had nano configured correctly.

```
osboxes@osboxes:~$ touch myDocker.dockerfile
osboxes@osboxes:~$ ls
Desktop    Downloads  myDocker.dockerfile  Public  Templates
Documents  Music      Pictures             snap    Videos
osboxes@osboxes:~$
```

- Docker enables “recipes” sharing to create persistent images (as a second alternative of snapshots). To make a proper recipe, we have written a Dockerfile (myDocker.dockerfile):

```
FROM ubuntu
RUN apt update -y
RUN apt install -y nano
CMD ["/bin/bash"]
```

Then, we have built the image in the VM :

```
osboxes@osboxes:~/Desktop$ sudo docker build -t container3:version2 -f myDocker.
dockerfile .
Sending build context to Docker daemon  2.048kB
Step 1/4 : FROM ubuntu
 ---> 216c552ea5ba
Step 2/4 : RUN apt update -y
 ---> Running in df3ab2da64d8
```

Objectives 6 and 7

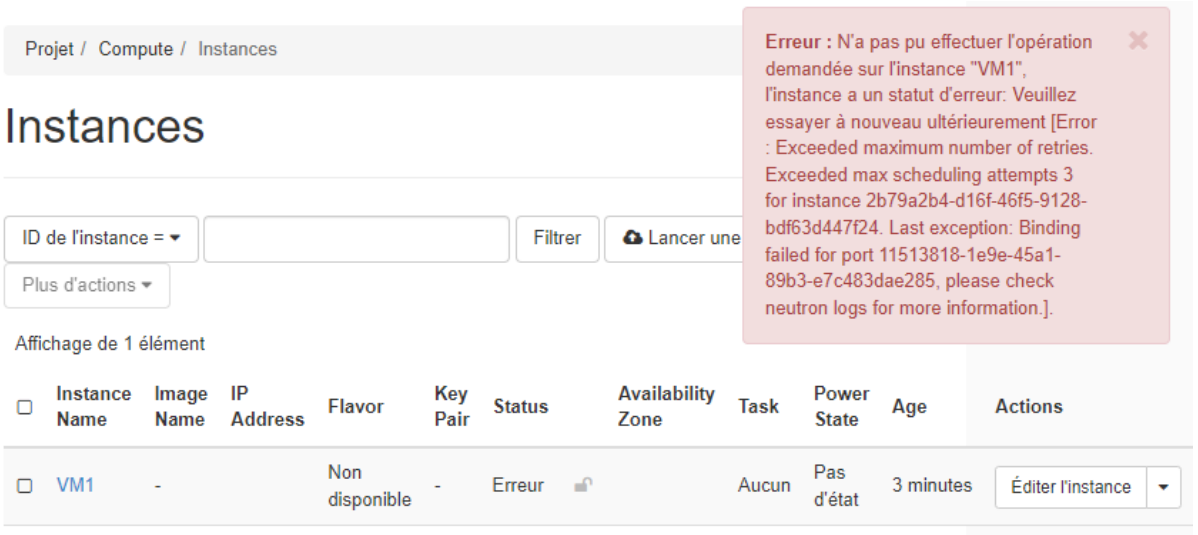
First part : CT creation and configuration on OpenStack

The initial step involves accessing the OpenStack web interface, facilitated by utilizing our INSA login credentials through the CSH (Cloud Service Hub). This secure login mechanism ensures that only authorized users can interact with the OpenStack platform. Once logged in, we navigate to our default project, a designated workspace within OpenStack that encapsulates our specific resources and configurations.

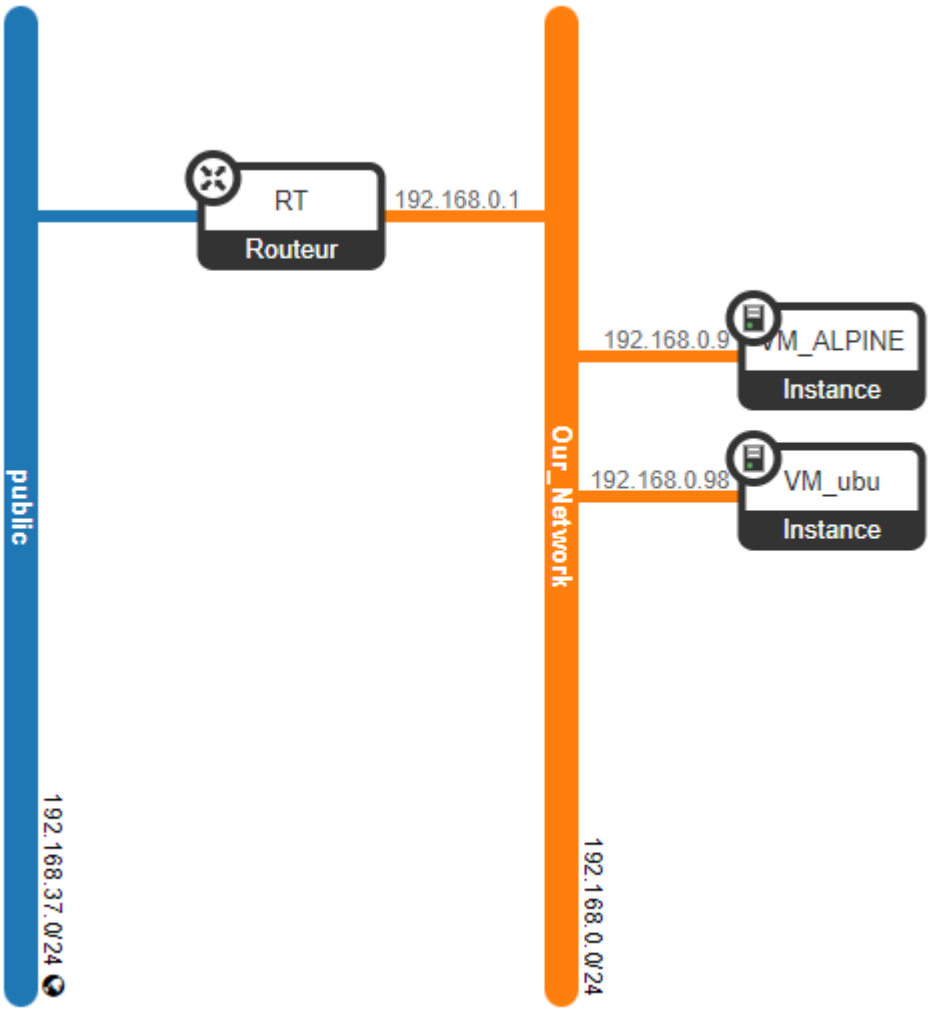
Next, we proceed to the "instances" tab, a section dedicated to managing virtual machine instances. Within this tab, we initiate the creation of a new virtual machine (VM) by selecting the desired specifications.

Specifically, the VM is configured using the available "ubuntu4CLV" image, which serves as the base operating system for our VM.

Additionally, we choose the "small2" flavor, indicating the predefined set of computing resources (CPU, memory, etc.) allocated to our VM. This step ensures that our VM is provisioned with the necessary attributes to meet our requirements.



The VM can’t launch because the network can’t give it an IP address: because of the public network’s security settings, it doesn’t accept the new VM as part of the network and doesn’t give it an IP address. We need to set up our private network to be able to add our VMs to it. To do so, we add two rules to the security group: SSH and ICMP. We then setup the network topology, with a private network containing our VM, and a gateway between the public and private networks



Second part: Connectivity test

First, we start by running the "**ipconfig**" command in our command terminal to get the ip address of our openstack VM

```
U:\>ipconfig

Configuration IP de Windows


Carte Ethernet Ethernet :

   Suffixe DNS propre à la connexion. . . : insa-toulouse.fr
   Adresse IPv6 de liaison locale. . . . .: fe80::3f1c:5559:9cf3:73d4%4
   Adresse IPv4. . . . . . . . . . . . . .: 10.1.5.234
   Masque de sous-réseau. . . . . . . . . : 255.255.0.0
   Passerelle par défaut. . . . . . . . . : 10.1.0.254

Carte Ethernet VirtualBox Host-Only Network :

   Suffixe DNS propre à la connexion. . . :
   Adresse IPv6 de liaison locale. . . . .: fe80::2693:9263:322c:50f2%6
   Adresse IPv4. . . . . . . . . . . . . .: 192.168.56.1
   Masque de sous-réseau. . . . . . . . . : 255.255.255.0
   Passerelle par défaut. . . . . . . . . :
```

IP PC : 10.1.5.234
IP VM : 192.168.56.1

- **Ping From VM to PC**

```
user@tutorial-vm:~$ ping 10.1.5.234
PING 10.1.5.234 (10.1.5.234) 56(84) bytes of data.
64 bytes from 10.1.5.234: icmp_seq=1 ttl=126 time=2.53 ms
64 bytes from 10.1.5.234: icmp_seq=2 ttl=126 time=1.53 ms
64 bytes from 10.1.5.234: icmp_seq=3 ttl=126 time=1.42 ms
64 bytes from 10.1.5.234: icmp_seq=4 ttl=126 time=3.27 ms
64 bytes from 10.1.5.234: icmp_seq=5 ttl=126 time=1.60 ms
^C
--- 10.1.5.234 ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 4005ms
rtt min/avg/max/mdev = 1.420/2.074/3.277/0.721 ms
```

—> It is working

- **Ping from Vm to Internet**

```
user@tutorial-vm:~$ ping youtube.com
PING youtube.com (142.251.37.238) 56(84) bytes of data.
64 bytes from mrs09s16-in-f14.1e100.net (142.251.37.238): icmp_seq=1 ttl=114 time=6.21 ms
64 bytes from mrs09s16-in-f14.1e100.net (142.251.37.238): icmp_seq=2 ttl=114 time=6.39 ms
64 bytes from mrs09s16-in-f14.1e100.net (142.251.37.238): icmp_seq=3 ttl=114 time=6.35 ms
^C
--- youtube.com ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2003ms
rtt min/avg/max/mdev = 6.210/6.318/6.391/0.077 ms
```

—> The ping is working

```
user@tutorial-vm:~$ ifconfig
ens3: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1450
        inet 192.168.0.98  netmask 255.255.255.0  broadcast 192.168.0.255
        inet6 fe80::f816:3eff:feb0:2f7c  prefixlen 64  scopeid 0x20<link>
        ether fa:16:3e:b0:2f:7c  txqueuelen 1000  (Ethernet)
        RX packets 7856  bytes 49793096 (49.7 MB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 2278  bytes 173996 (173.9 KB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING>  mtu 65536
        inet 127.0.0.1  netmask 255.0.0.0
        inet6 ::1  prefixlen 128  scopeid 0x10<host>
        loop  txqueuelen 1000  (Boucle locale)
        RX packets 100  bytes 9042 (9.0 KB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 100  bytes 9042 (9.0 KB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

- **Then we ping The Vm from the Host**

```
U:\>ping 192.168.0.98

Envoi d'une requête 'Ping'  192.168.0.98 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.

Statistiques Ping pour 192.168.0.98:
    Paquets : envoyés = 3, reçus = 0, perdus = 3 (perte 100%),
```

—> The attempt to ping from a PC to a VM does not work , signifying that the two entities are not in the same network address. The inability to establish a successful connection indicates that the VM is not reachable from the PC. This lack of reachability may stem from the fact that the VM belongs to a private network, a network configuration often used to enhance security by isolating internal resources. In this context, the VM's network address may not be directly accessible from external sources, such as the PC attempting the ping. The private network configuration restricts the routing of any packets or communication attempts, making it challenging to establish a direct connection or "root any packet" to the VM from the PC.

Floating IP:

In response to the challenge of the PC being unable to ping the VM due to network address disparities and the VM belonging to a private network, a solution is proposed. The resolution involves navigating to the network settings and creating a floating IP, which is subsequently linked to the VM. This action effectively introduces an additional layer of connectivity to the VM.

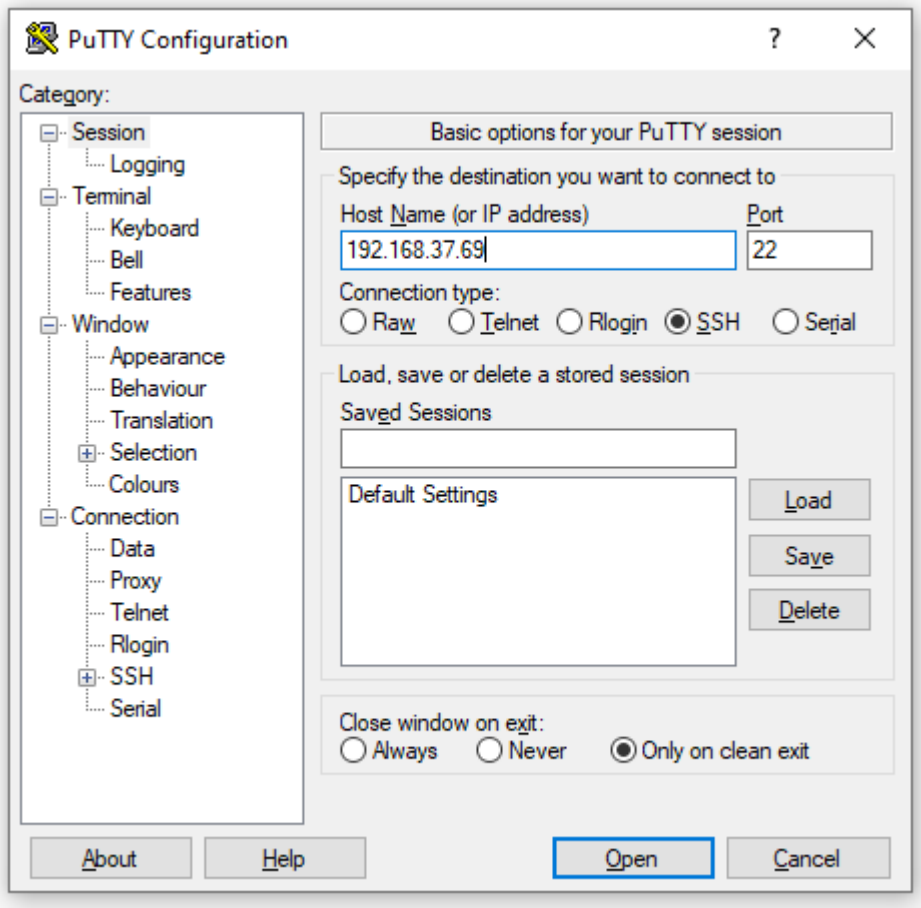
☐	IP Address	Description	DNS Name	DNS Domain	Mapped Fixed IP Address	Pool	Status	Actions
☐	192.168.37.69				VM_ubu 192.168.0.98	public	Active	Dissocier ▾

- Our Floating address is : **192.168.37.69**

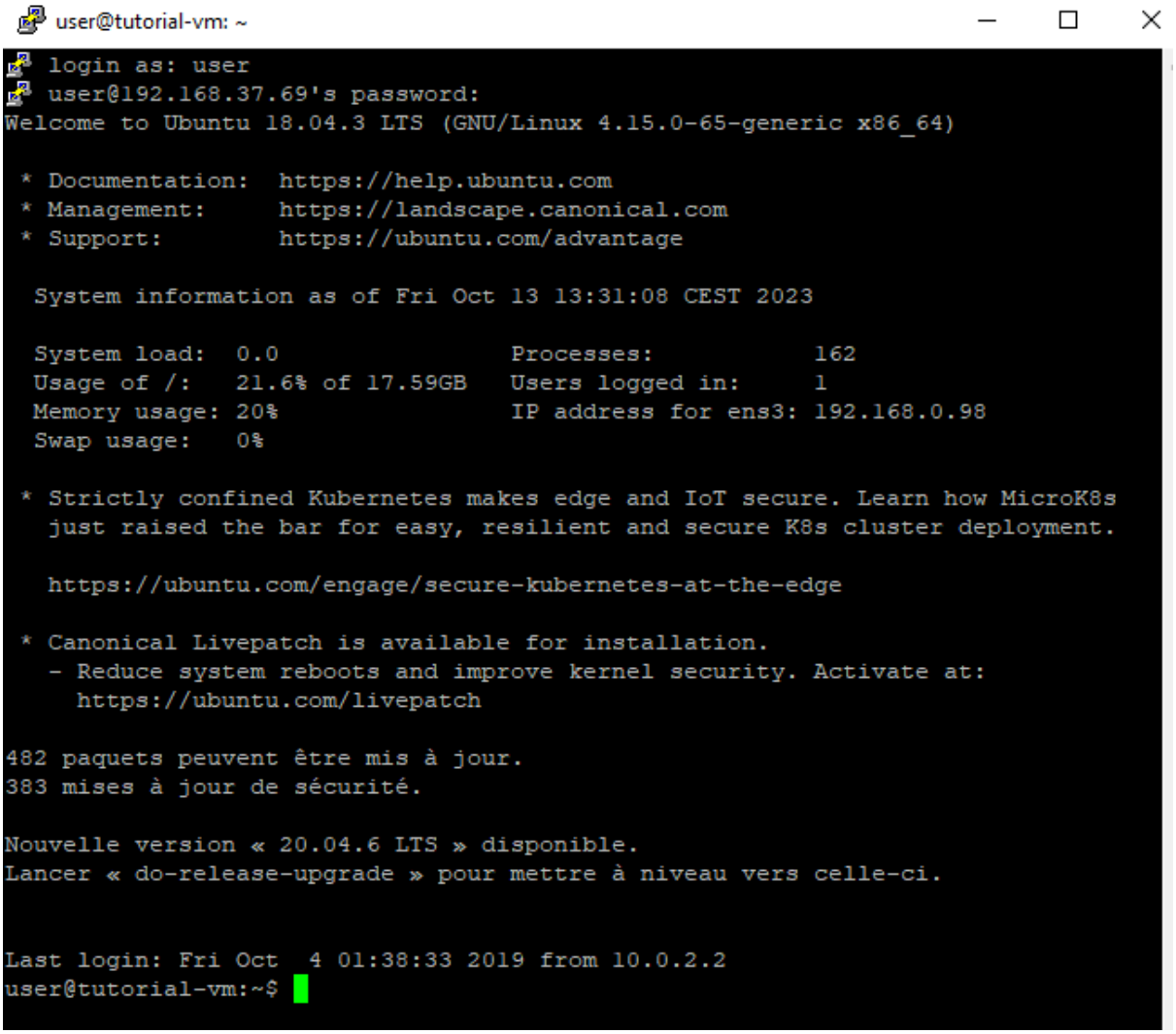
We were required to capture a screenshot while pinging 192.168.37.69 from the local host (our PC), but unfortunately, we overlooked this step. Nonetheless, the absence of the screenshot will not hinder the success of the operation.

- **SSH CLIENT**

To test the connectivity of the VM from outside, we utilised an SSH client, specifically PuTTY. The results of this test were successful, confirming that we could establish an SSH connection with the VM from an external source.

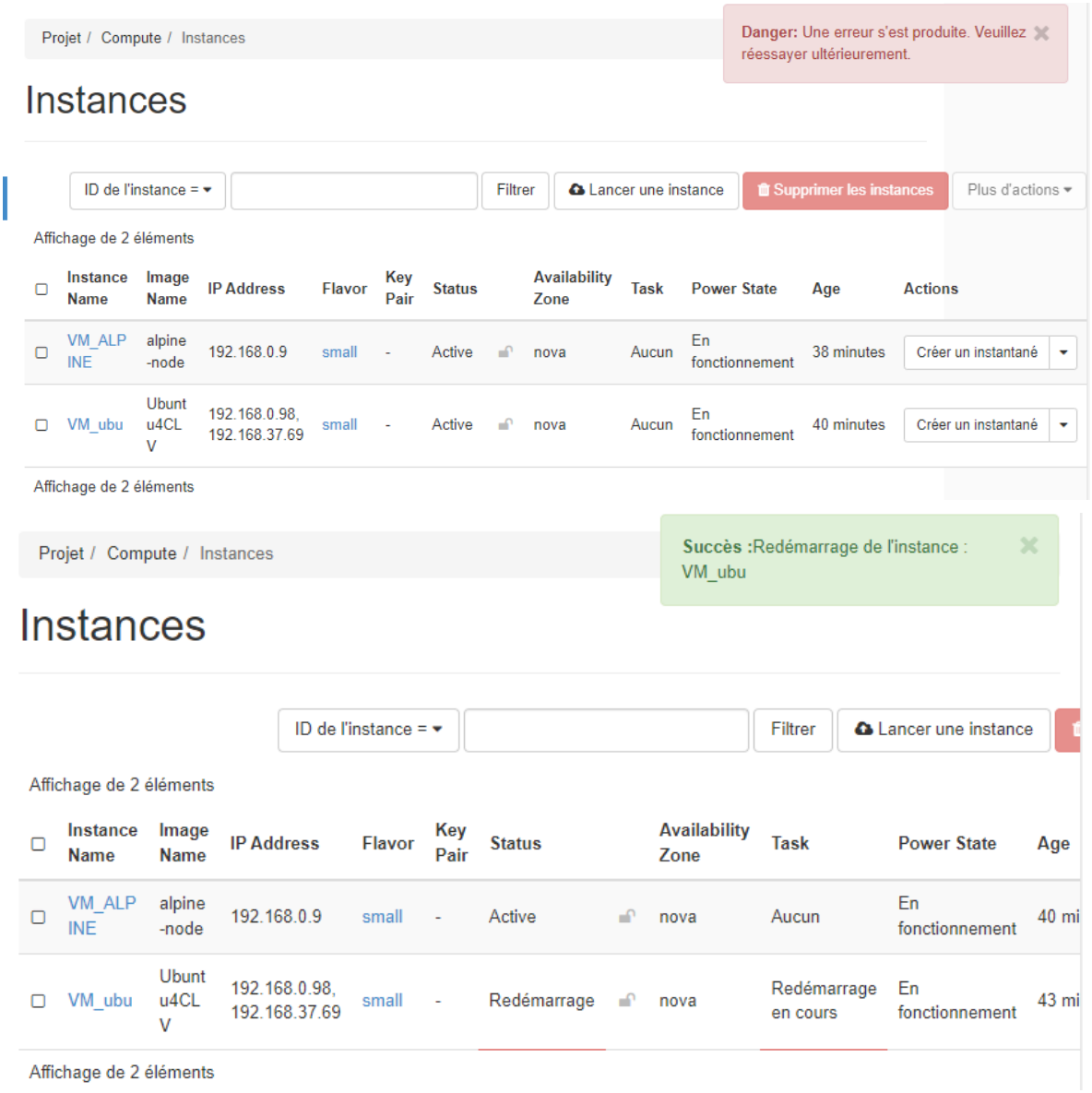


Additionally, a noteworthy recommendation is provided for future operations on VMs. To streamline and enhance security in SSH interactions, it's suggested to create and generate SSH keys. These keys can be associated with the VMs during the creation process, and this can be conveniently managed from the "Key pairs" section in the Compute tab. By implementing SSH key pairs, we can enhance authentication and facilitate secure access to VMs in subsequent operations.



Third part: Snapshot, restore and resize a VM

In the process of resizing a running VM from the Instances tab, I initially observed limitations that hindered the direct adjustment of the VM's configuration while it was in an active state.



Subsequently, after shutting down the VM and attempting the same operation, I noticed increased flexibility, allowing for more extensive modifications to the VM's size and resources. These actions underscored the inherent flexibility and agility within virtualization settings, as elucidated in previous lectures.

However, it's crucial to acknowledge an existing technical limitation within OpenStack—specifically, the inability to resize a running VM to a larger flavor directly. To address this constraint, a viable solution involves creating a new VM with the desired flavor, migrating the data, and subsequently decommissioning the original VM. This strategic workaround enables resizing without compromising the operational state of the VM.

Additionally, making a snapshot of the VM provides a frozen image capturing its current configuration, and when comparing it with the original image, differences may arise due to changes made post-snapshot.

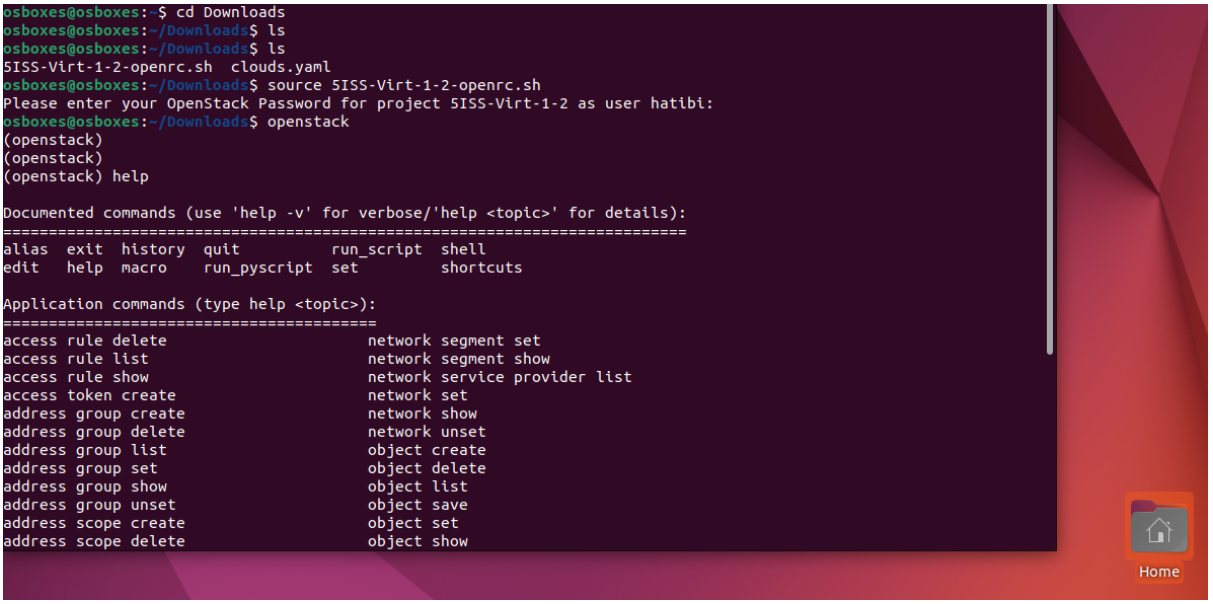
Objectives 8 and 9

Part one : OpenStack client installation

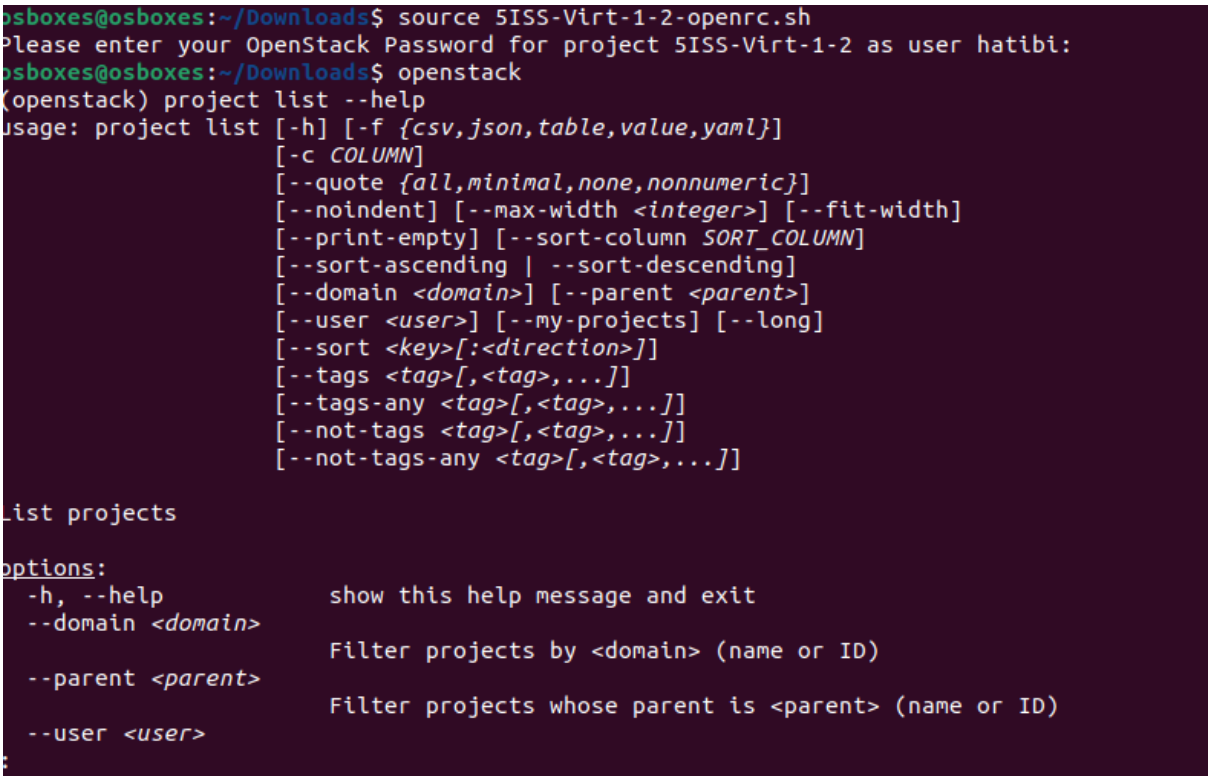
In the first step of the process, we focus on the installation of the OpenStack command-line client, a vital tool facilitating the invocation of remote REST operations. This client plays a pivotal role in managing and interacting with OpenStack resources from the command line. To proceed with the installation, the command " `$ sudo apt install python3-openstackclient"` is executed within the Ubuntu VM hosted on VirtualBox.

In this phase, the focus shifts towards configuring the OpenStack client with essential variables such as the OpenStack address, port, and the designated project. To achieve this, a crucial step involves generating and downloading the RC v3 file from the OpenStack dashboard. This file encapsulates all the necessary information required for client configuration. Following the download, the next action involves executing the RC v3 file locally from a terminal. This process effectively establishes the necessary environment variables, allowing the OpenStack client to authenticate and interact with the specified OpenStack instance seamlessly.

By incorporating the details from the RC v3 file, users can ensure that their OpenStack client is appropriately configured, enabling efficient and secure command-line operations tailored to their specific OpenStack environment.



- We start the client by the following command “`$ openstack "` and we can display the help by typing `help`

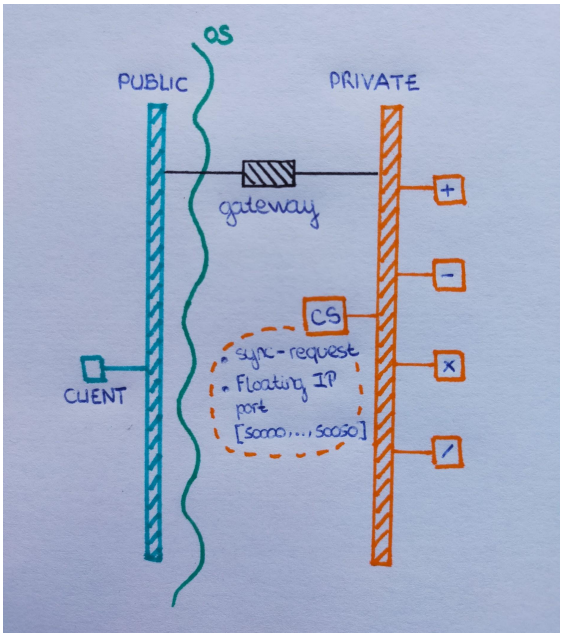


Part two : Web 2-tier application topology and specification

In this section, our proposal involves the deployment of a 2-tier Web application on the OpenStack platform. The application is designed to execute arithmetic operations, including addition, subtraction, multiplication, and division, specifically for integer numbers.

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

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



The client is responsible for communicating with the central server (CS), transmitting each calculation operation, and prompting the CS to invoke various web services when necessary.

Before achieving seamless functionality, modifications need to be made to the CS script, including:

- Updating the IP addresses associated with each virtual machine containing the diverse web services.
- Adjusting the port from 80 to a port within the range of 50,000 to 50,050, as this is the only port accepted by the CSN (Central Server Node). Using other ports may result in communication being blocked by the gateway controlled by the CSN.
- Altering the floating IP address assigned to the CS VM.

Moreover, an additional step involves adding a rule to activate port 50000.

To ensure seamless communication within the system, a few adjustments were made : Initially, the listening port in the source code of CalculatorService.js was modified from 80 to 50000. This alteration ensures compatibility with the specified requirements of the system.

Moreover, given the distinct local IP addresses of the services, it was imperative to update these IPs in CalculatorService.js accordingly. This step ensures accurate identification and interaction between the various components of the system.

Lastly, to facilitate external communication with the frontend machine, a floating IP was assigned. This enables communication from outside the private network, allowing us to effectively ping it from our separate Ubuntu VM. These modifications collectively contribute to the enhanced functionality and connectivity of the system components.

Installation of curl :

```
root@bc306bc8711a:/# apt install curl
Reading package lists... Done
Building dependency tree... Done
Reading state information... Done
The following additional packages will be installed:
  ca-certificates libbrotli1 libcurl4 libldap-2.5-0 libldap-common
  libnghttp2-14 libpsl5 librtmp1 libsasl2-2 libsasl2-modules
  libsasl2-modules-db libssh-4 openssl publicsuffix
Suggested packages:
  libsasl2-modules-gssapi-mit | libsasl2-modules-gssapi-heimdal
  libsasl2-modules-ldap libsasl2-modules-otp libsasl2-modules-sql
The following NEW packages will be installed:
  ca-certificates curl libbrotli1 libcurl4 libldap-2.5-0 libldap-common
  libnghttp2-14 libpsl5 librtmp1 libsasl2-2 libsasl2-modules
  libsasl2-modules-db libssh-4 openssl publicsuffix
0 upgraded, 15 newly installed, 0 to remove and 34 not upgraded.
```

Port list for each application :

- **SumService**: 50001
- **SubService**: 50002
- **MulService**: 50003
- **DivService**: 50004
- **CalculatorService**: 50000

As the local IP address of services are different, we need to change those IPs in CalculatorService.js as following :

```
const SUM_SERVICE_IP_PORT = 'http://192.168.0.79:50001';
const SUB_SERVICE_IP_PORT = 'http://192.168.0.22:50002';
const MUL_SERVICE_IP_PORT = 'http://192.168.0.64:50003';
const DIV_SERVICE_IP_PORT = 'http://192.168.0.186:50004';
```

Part three: Deploy the Calculator application on OpenStack

Using the recently installed OpenStack client, we'll instantiate 5 virtual machines (VMs) designated to host the five microservices of the Calculator application.

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

Instance Name	Image Name	IP Address
calculatorService	calcSnap	192.168.0.90, 192.168.37.155
divService	calcSnap	192.168.0.186, 192.168.37.51
MulService	calcSnap	192.168.0.64, 192.168.37.177
subService	calcSnap	192.168.0.22, 192.168.37.183
sumService	ubuntu4CLV	192.168.0.79, 192.168.37.13

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, 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
```

Because of the port listening we needed to add a new rule to “Security Group” in order to allow packet transfert on port range 50000 to 50004 as following :

☐ Ingress	IPv4	TCP	50000 - 50004	0.0.0.0/0

We can now request all the services with the following command :

```
CalculatorService : $ curl -d "(5+6)*2" -X POST http://<ip>:50000
SumService :        $ curl -d "2 3" -X POST http://<ip>:50001
SubService :        $ curl -d "5 3" -X POST http://<ip>:50002
MulService :        $ curl -d "12 3" -X POST http://<ip>:50003
DivService :        $ curl -d "15 5" -X POST http://<ip>:50004
```

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

```
curl -d "(5+6)*2" -X POST http://192.168.37.155:50000
```

```
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
```

.

Request example :

```
curl -d "(5+6)*2" -X POST http://192.168.37.155:50000
```

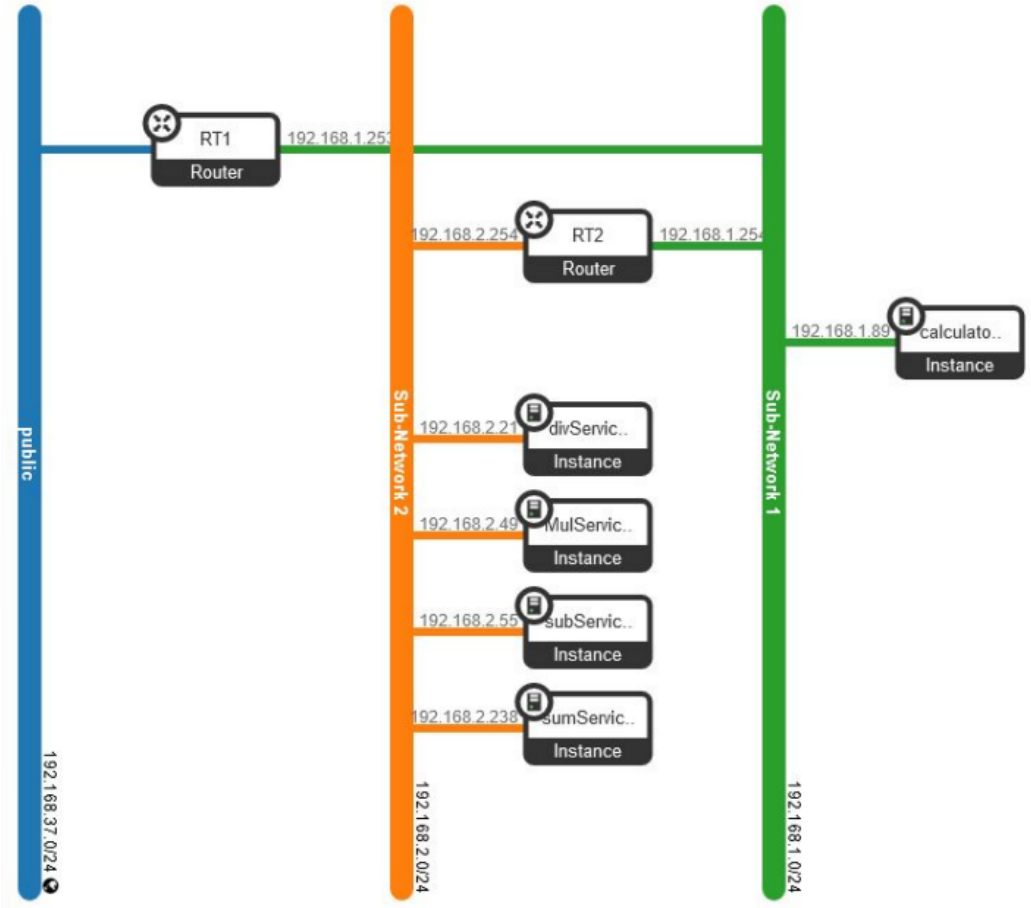
```
user@tutorial-vm:~/Téléchargements$ curl -d "2*3" -X POST http://192.168.37.96:5
0000
result = 6
```

When we change the source code while the service is running (adding a console.log) , it receives the request, does the operation, but doesn’t send the result back. If we comment that extra line

Part four: Automate/Orchestrate the application deployment (optional)

In this part we will redeploy and re-execute the same application using a set of Docker nodes.

We created a new network on openstack following the topology given in the tutorial



We used the virtual machines with the services already running on them to avoid having to configure them again.

Let's examine the connectivity between the virtual machine hosting the calculator

front end and any of the machines responsible for hosting the arithmetic operations services. What observations can you make? Please provide a detailed elaboration on your findings.

→The front end machine and the calculating machines can't communicate. Since they're on different networks, we need to set up routes so that the different machines know how to reach each other despite being in different networks.

To do so, we must first add a route that tells our front end machine to access network2 any time it tries to communicate with one of the 4 services.
The default route in network1 also seemed to be a problem, as after setting everything up, we could ping the router but not the calculator. By rewriting the default route manually, we fixed this issue and were able to send requests to the service.

```
sudo route add -net default gw 192.168.1.253
sudo route add -net 192.168.2.0/24 gw 192.168.1.254
```

We also need to specify the services' new IP addresses in the calculator's source code

```
const SUM_SERVICE_IP_PORT = 'http://192.168.2.238:50001';
const SUB_SERVICE_IP_PORT = 'http://192.168.2.55:50002';
const MUL_SERVICE_IP_PORT = 'http://192.168.2.49:50003';
const DIV_SERVICE_IP_PORT = 'http://192.168.2.21:50004';
```

When the calculator service is requested, it replies with the right result :

```
curl -d "(5+6)*2" -X POST http://192.168.37.82:50000
result = 22
```

The calculator service receive the request and answer as following :

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

Objectives 10 and 11

We utilised the rc file to extract essential information, including the domain name, OS domain ID, and OS project ID.

```
export OS_AUTH_URL=https://os-api-ext.insa-toulouse.fr:5000/v3
# With the addition of Keystone we have standardized on the term **project**
# as the entity that owns the resources.
export OS_PROJECT_ID=17463813d1d54dce90bdef797b96806d
export OS_PROJECT_NAME="5ISS-Virt-1-2"
export OS_USER_DOMAIN_NAME="insa"
if [ -z "$OS_USER_DOMAIN_NAME" ]; then unset OS_USER_DOMAIN_NAME; fi
export OS_PROJECT_DOMAIN_ID="ef8de9847762471f9f8cea12458550d2"
if [ -z "$OS_PROJECT_DOMAIN_ID" ]; then unset OS_PROJECT_DOMAIN_ID; fi
# unset v2.0 items in case set
unset OS_TENANT_ID
unset OS_TENANT_NAME
# In addition to the owning entity (tenant), OpenStack stores the entity
# performing the action as the **user**.
export OS_USERNAME="hatibi"
# With Keystone you pass the keystone password.
echo "Please enter your OpenStack Password for project $OS_PROJECT_NAME as user $OS_USERNAME: "
read -sr OS_PASSWORD_INPUT
export OS_PASSWORD=$OS_PASSWORD_INPUT
# If your configuration has multiple regions, we set that information here.
# OS_REGION_NAME is optional and only valid in certain environments.
export OS_REGION_NAME="RegionINSA"
# Don't leave a blank variable, unset it if it was empty
if [ -z "$OS_REGION_NAME" ]; then unset OS_REGION_NAME; fi
export OS_INTERFACE=public
export OS_IDENTITY_API_VERSION=3
```

Leveraging this extracted data, we implemented the following Python code to create the networks:

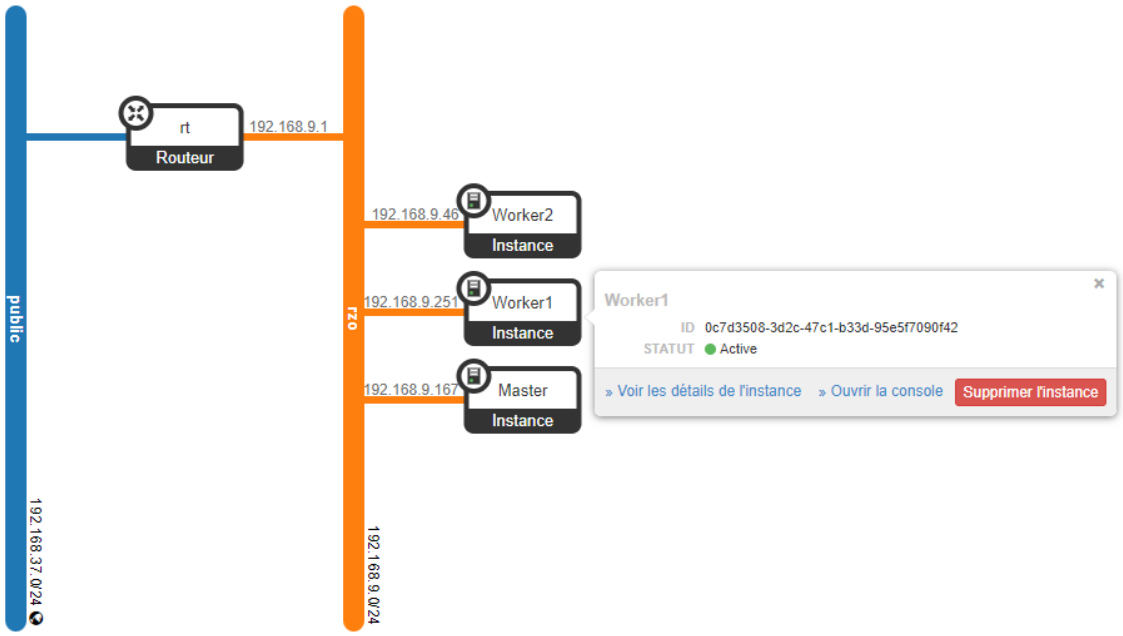
https://github.com/amine0561/Cloud.git

Unfortunately, due to time constraints, the testing of services has not been conducted yet. The primary focus was on establishing the foundational components, and the next steps involve thorough testing to ensure the robust functionality of the services within the specified environment.

Lab 2 : Orchestrating services in hybrid cloud/edge environment

Part 1: Cloud infrastructure setup

In this phase, we're aiming to set up our Kubernetes cluster with three active nodes.
Let's start by connecting to the OpenStack web interface. We'll then authenticate using our INSA login details within the "INSA" group/domain and ensure we're connected to our default project on OpenStack.
Now, let's create a new private network and link it to the public network using a well-configured gateway. We'll instantiate the master-node VM in this network, specifying it to use the Ubuntu_20 cloud image (login=root, password=root) and a medium flavour. Additionally, we'll instantiate two worker node VMs in the same network, using the Ubuntu20 cloud image (login=root, password=root) and a small2 flavour.



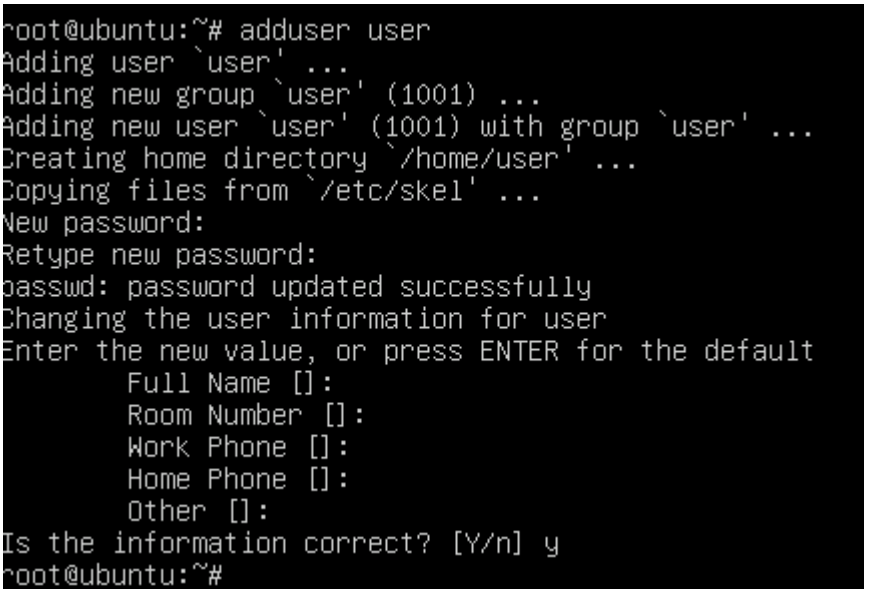
Finally, let's generate three floating IP addresses and associate each with its respective VM.

IP Address	Description	DNS Name	DNS Domain	Mapped Fixed IP Address	Pool	Status	Actions
192.168.37.186				Worker2 192.168.9.46	public	Active	Dissocier
192.168.37.69				Worker1 192.168.9.251	public	Active	Dissocier
192.168.37.62	Master			Master 192.168.9.167	public	Active	Dissocier

Affichage de 3 éléments

On each machine, let's go ahead and create a fresh "user" with the password set to "user" by executing the command: `$ adduser user`

After creating the user, we'll want to grant sudo privileges to our newly added user. We can achieve this by running the following command:`$ usermod -aG sudo user`



Let's establish the SSH connection on the VM by configuring the sshd_config file. We use the nano editor to open the file with the following command: **$ nano /etc/ssh/sshd_config**

```
#       $OpenBSD: sshd_config,v 1.103 2018/04/09 20:41:22 tj Exp $

# This is the sshd server system-wide configuration file.  See
# sshd_config(5) for more information.

# This sshd was compiled with PATH=/usr/bin:/bin:/usr/sbin:/sbin

# The strategy used for options in the default sshd_config shipped with
# OpenSSH is to specify options with their default value where
# possible, but leave them commented.  Uncommented options override the
# default value.

Include /etc/ssh/sshd_config.d/*.conf

#Port 22
#AddressFamily any
#ListenAddress 0.0.0.0
#ListenAddress ::

#HostKey /etc/ssh/ssh_host_rsa_key
#HostKey /etc/ssh/ssh_host_ecdsa_key
#HostKey /etc/ssh/ssh_host_ed25519_key

# Ciphers and keying
#RekeyLimit default none

# Logging
#SyslogFacility AUTH
#LogLevel INFO

# Authentication:

#LoginGraceTime 2m
#PermitRootLogin prohibit-password
#StrictModes yes
#MaxAuthTries 6
#MaxSessions 10

#PubkeyAuthentication yes

# Expect .ssh/authorized_keys2 to be disregarded by default in future.
#AuthorizedKeysFile     .ssh/authorized_keys .ssh/authorized_keys2

#AuthorizedPrincipalsFile none
                                                    [ Read 123 lines ]
```

Inside the sshd_config file, we locate the line for PasswordAuthentication and we set it to "**yes.**" Once you've made this adjustment, we save and exit the nano editor:

```
PasswordAuthentication yes
```

After modifying the configuration, we'll need to reload the SSH service to apply the changes. Execute the following command to restart the SSHD service:`$ systemctl restart sshd`

```
root@ubuntu:~#
root@ubuntu:~# systemctl restart sshd
root@ubuntu:~#
```

This ensures that the SSH connection is set up on the VM with the specified configuration changes.

From our computer, we open three terminal tabs connected to our deployed servers using SSH.

```
U:\>ssh user@192.168.37.62
The authenticity of host '192.168.37.62 (192.168.37.62)' can't be established.
ECDSA key fingerprint is SHA256:JuG7yak+TaLrke7MUipazGhdFEmJmKJiYS+fa0PgJjA.
Are you sure you want to continue connecting (yes/no/[fingerprint])? y
Please type 'yes', 'no' or the fingerprint: y
Please type 'yes', 'no' or the fingerprint: yes
Warning: Permanently added '192.168.37.62' (ECDSA) to the list of known hosts.
user@192.168.37.62's password:
Welcome to Ubuntu 20.04.6 LTS (GNU/Linux 5.4.0-163-generic x86_64)

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

  System information as of Tue Nov  7 11:53:16 UTC 2023

  System load:  0.0               Processes:             117
  Usage of /:   3.8% of 38.58GB   Users logged in:       1
  Memory usage: 10%               IPv4 address for ens3: 192.168.9.167
  Swap usage:   0%


Expanded Security Maintenance for Applications is not enabled.

0 updates can be applied immediately.

Enable ESM Apps to receive additional future security updates.
See https://ubuntu.com/esm or run: sudo pro status


The list of available updates is more than a week old.
To check for new updates run: sudo apt update
New release '22.04.3 LTS' available.
Run 'do-release-upgrade' to upgrade to it.



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.

To run a command as administrator (user "root"), use "sudo <command>".
See "man sudo_root" for details.

user@ubuntu:~$
```

To ensure proper operation of the kubelet in each node, we turn off swap by executing this command : `$ sudo swapoff -a`
We change also the name of our hostnames by this command `$ sudo hostnamectl set-hostname <node_name>` and check if the change is considered by typing `hostname`

- Here is an example of changing the master name

```
user@ubuntu:~$  sudo swapoff -a
[sudo] password for user:
user@ubuntu:~$ sudo hostnamectl set-hostname master
user@ubuntu:~$ hostname
master
user@ubuntu:~$
```

So, first things, let's get our system all up to date and ready for Kubernetes.

```
user@ubuntu:~$  sudo apt update && sudo apt install -y curl
Hit:1 http://archive.ubuntu.com/ubuntu focal InRelease
Get:2 http://security.ubuntu.com/ubuntu focal-security InRelease [114 kB]
Get:3 http://archive.ubuntu.com/ubuntu focal-updates InRelease [114 kB]
Get:4 http://archive.ubuntu.com/ubuntu focal-backports InRelease [108 kB]
Get:5 http://archive.ubuntu.com/ubuntu focal/universe amd64 Packages [8628 kB]
Get:6 http://security.ubuntu.com/ubuntu focal-security/main amd64 Packages [2569 kB]
Get:7 http://archive.ubuntu.com/ubuntu focal/universe Translation-en [5124 kB]
Get:8 http://security.ubuntu.com/ubuntu focal-security/main Translation-en [398 kB]
Get:9 http://security.ubuntu.com/ubuntu focal-security/main amd64 c-n-f Metadata [13.2 kB]
```

Now, we need the key to access the Kubernetes repository

```
user@ubuntu:~$ curl -s https://packages.cloud.google.com/apt/doc/apt-key.gpg | sudo apt-key add
OK
```

Next, we're telling our system where to find all the Kubernetes goodies

```
user@ubuntu:~$ cat << EOF | sudo tee /etc/apt/sources.list.d/kubernetes.list
> deb https://apt.kubernetes.io/ kubernetes-xenial main
> EOF
deb https://apt.kubernetes.io/ kubernetes-xenial main
user@ubuntu:~$
```

Then we upgrade our packages by running this :

```
user@ubuntu:~$ sudo apt update
Hit:1 http://security.ubuntu.com/ubuntu focal-security InRelease
Hit:2 http://archive.ubuntu.com/ubuntu focal InRelease
Hit:4 http://archive.ubuntu.com/ubuntu focal-updates InRelease
Hit:5 http://archive.ubuntu.com/ubuntu focal-backports InRelease
Get:3 https://packages.cloud.google.com/apt kubernetes-xenial InRelease [8993 B]
Get:6 https://packages.cloud.google.com/apt kubernetes-xenial/main amd64 Packages [69.9 kB]
Fetched 78.9 kB in 2s (46.4 kB/s)
Reading package lists... Done
Building dependency tree
Reading state information... Done
23 packages can be upgraded. Run 'apt list --upgradable' to see them.
```

Now, let's bring in the star of the show - Kubernetes:

```
user@ubuntu:~$ sudo apt install -y kubeadm=1.27.6-00 kubelet=1.27.6-00 kubectl=1.27.6-00
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following additional packages will be installed:
  conntrack cri-tools ebtables kubernetes-cni socat
Suggested packages:
  nftables
The following NEW packages will be installed:
  conntrack cri-tools ebtables kubeadm kubectl kubelet kubernetes-cni socat
0 upgraded, 8 newly installed, 0 to remove and 23 not upgraded.
Need to get 85.9 MB of archives.
After this operation, 329 MB of additional disk space will be used.
Get:1 http://archive.ubuntu.com/ubuntu focal/main amd64 conntrack amd64 1:1.4.5-2 [30.3 kB]
Get:2 http://archive.ubuntu.com/ubuntu focal/main amd64 ebtables amd64 2.0.11-3build1 [80.3 kB]
Get:3 http://archive.ubuntu.com/ubuntu focal/main amd64 socat amd64 1.7.3.3-2 [323 kB]
```

Lastly, we want to make sure these components stay put, no matter what , we then run this command :

```
user@ubuntu:~$ sudo apt-mark hold kubeadm kubelet kubectl
kubeadm set on hold.
kubelet set on hold.
kubectl set on hold.
```

And there we have it —> the system is geared up and ready to roll with Kubernetes!

Let's dive into setting up the container runtime. Since we're opting for Docker containers to host our services, we follow these steps:

First,we get Docker installed and ready to roll:

```
user@ubuntu:~$  sudo apt install -y docker.io
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following additional packages will be installed:
  bridge-utils containerd dns-root-data dnsmasq-base libidn11 pigz runc ubuntu-fan
Suggested packages:
  ifupdown aufs-tools cgroupfs-mount | cgroup-lite debootstrap docker-doc rinse zfs-fuse | zfsutils
The following NEW packages will be installed:
  bridge-utils containerd dns-root-data dnsmasq-base docker.io libidn11 pigz runc ubuntu-fan
0 upgraded, 9 newly installed, 0 to remove and 26 not upgraded.
Need to get 63.2 MB of archives.
After this operation, 267 MB of additional disk space will be used.
Get:1 http://archive.ubuntu.com/ubuntu focal/universe amd64 pigz amd64 2.4-1 [57.4 kB]
Get:2 http://archive.ubuntu.com/ubuntu focal/main amd64 bridge-utils amd64 1.6-2ubuntu1 [30.5 kB]
Get:3 http://archive.ubuntu.com/ubuntu focal-updates/main amd64 runc amd64 1.1.7-0ubuntu1~20.04.1 [3819 kB]
Get:4 http://archive.ubuntu.com/ubuntu focal-updates/main amd64 containerd amd64 1.7.2-0ubuntu1~20.04.1 [32.5 MB]
Get:5 http://archive.ubuntu.com/ubuntu focal/main amd64 dns-root-data all 2019052802 [5300 B]
Get:6 http://archive.ubuntu.com/ubuntu focal/main amd64 libidn11 amd64 1.33-2.2ubuntu2 [46.2 kB]
Get:7 http://archive.ubuntu.com/ubuntu focal-updates/main amd64 dnsmasq-base amd64 2.80-1.1ubuntu1.7 [315 kB]
Get:8 http://archive.ubuntu.com/ubuntu focal-updates/universe amd64 docker.io amd64 24.0.5-0ubuntu1~20.04.1 [26.4
MB]
```

```
user@ubuntu:~$ sudo usermod -aG docker $USER
user@ubuntu:~$ newgrp docker
user@master:~$
```

We ensure Docker initialises with system startup by running this command : `sudo systemctl enable docker`

To preempt the issue of "Docker group driver detected cgroups instead systemd" during cluster initialization, it is imperative to modify the Docker cgroup driver on all nodes (both master and workers). We execute the following commands:

```
user@worker2:~$ sudo cat <<EOF | sudo tee /etc/docker/daemon.json
> { "exec-opts": ["native.cgroupdriver=systemd"],
> "log-driver": "json-file",
> "log-opts":
> { "max-size": "100m" },
> "storage-driver": "overlay2"
> }
> EOF
{ "exec-opts": ["native.cgroupdriver=systemd"],
"log-driver": "json-file",
"log-opts":
{ "max-size": "100m" },
"storage-driver": "overlay2"
}
user@worker2:~$
```

Subsequently, we restart the Docker service:

```
user@worker1:~$ sudo systemctl restart docker
user@worker1:~$
```

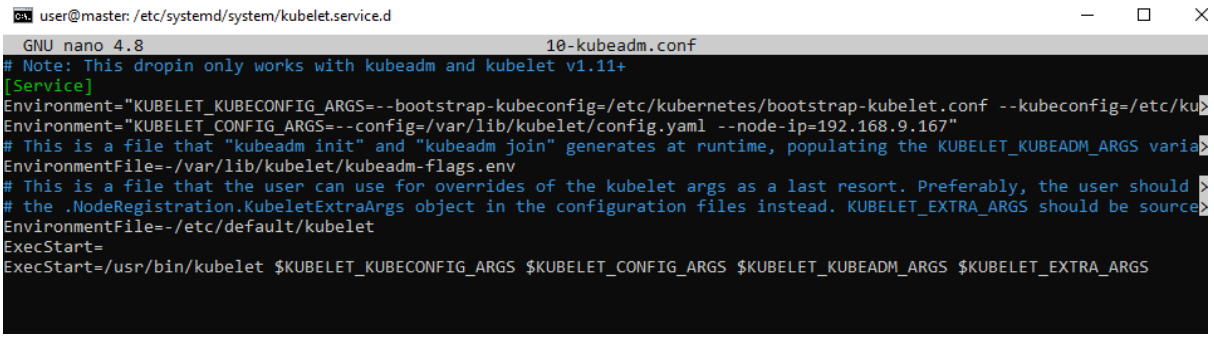
We Verify the change in cgroup driver to systemd by running this command `sudo docker info | grep -i cgroup`

Then , We Enable IP forwarding in the kernel:

```
user@worker2:~$ sudo modprobe overlay
user@worker2:~$ sudo modprobe br_netfilter
user@worker2:~$ sudo tee /etc/sysctl.d/kubernetes.conf<<EOF
> net.bridge.bridge-nf-call-ip6tables = 1
> net.bridge.bridge-nf-call-iptables = 1
> net.ipv4.ip_forward = 1
> EOF
net.bridge.bridge-nf-call-ip6tables = 1
net.bridge.bridge-nf-call-iptables = 1
net.ipv4.ip_forward = 1
```

```
user@worker2:~$ sudo sysctl --system
* Applying /etc/sysctl.d/10-console-messages.conf ...
kernel.printk = 4 4 1 7
* Applying /etc/sysctl.d/10-ipv6-privacy.conf ...
net.ipv6.conf.all.use_tempaddr = 2
net.ipv6.conf.default.use_tempaddr = 2
* Applying /etc/sysctl.d/10-kernel-hardening.conf ...
kernel.kptr_restrict = 1
* Applying /etc/sysctl.d/10-link-restrictions.conf ...
fs.protected_hardlinks = 1
fs.protected_symlinks = 1
* Applying /etc/sysctl.d/10-magic-sysrq.conf ...
kernel.sysrq = 176
```

Before initiating the cluster, we edit the kubelet config file



Then we restart the daemon and kubelet to ensure that the configuration is effective , by running this two command : -**sudo systemctl daemon-reload / - sudo systemctl daemon-reload**

To initialise our master we run this following command :

```
user@master:~$ sudo kubeadm init --apiserver-advertise-address 192.168.9.167 --control-plane-endpoint 192.168.9.16
7
```

```
To start using your cluster, you need to run the following as a regular user:

  mkdir -p $HOME/.kube
  sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
  sudo chown $(id -u):$(id -g) $HOME/.kube/config

Alternatively, if you are the root user, you can run:

  export KUBECONFIG=/etc/kubernetes/admin.conf

You should now deploy a pod network to the cluster.
Run "kubectl apply -f [podnetwork].yaml" with one of the options listed at:
  https://kubernetes.io/docs/concepts/cluster-administration/addons/

You can now join any number of control-plane nodes by copying certificate authorities
and service account keys on each node and then running the following as root:

  kubeadm join 192.168.9.167:6443 --token 7x8bcj.3qtd0g118xhh5cr0 \
        --discovery-token-ca-cert-hash sha256:85fd70f24628e233f61b6386c3a79e92c8eb4f1590a3638b1ea2265ba293b9c3 \
        --control-plane

Then you can join any number of worker nodes by running the following on each as root:

kubeadm join 192.168.9.167:6443 --token 7x8bcj.3qtd0g118xhh5cr0 \
        --discovery-token-ca-cert-hash sha256:85fd70f24628e233f61b6386c3a79e92c8eb4f1590a3638b1ea2265ba293b9c3
user@master:~$
```

In the course of configuring the Kubernetes cluster, a comprehensive series of steps was meticulously followed. Initial scrutiny of the existing nodes was conducted using the command:

```
user@master:~$ kubectl get nodes -o wide
NAME     STATUS     ROLES           AGE   VERSION   INTERNAL-IP     EXTERNAL-IP   OS-IMAGE             KERNEL-VERSION      CONTAINER-RUNTIME
master   NotReady   control-plane   10m   v1.27.6   192.168.9.167   <none>        Ubuntu 20.04.6 LTS   5.4.0-163-generic   containerd://1.7.2
user@master:~$
```

It was observed that the master node was initially in an "unready" state. Recognizing this, the subsequent command played a pivotal role in rectifying the readiness state of the master node:

```
user@master:~$ kubectl apply -f https://docs.projectcalico.org/manifests/calico.yaml
poddisruptionbudget.policy/calico-kube-controllers created
serviceaccount/calico-kube-controllers created
serviceaccount/calico-node created
serviceaccount/calico-cni-plugin created
configmap/calico-config created
customresourcedefinition.apiextensions.k8s.io/bgpconfigurations.crd.projectcalico.org created
customresourcedefinition.apiextensions.k8s.io/bgpfilters.crd.projectcalico.org created
customresourcedefinition.apiextensions.k8s.io/bgppeers.crd.projectcalico.org created
customresourcedefinition.apiextensions.k8s.io/blockaffinities.crd.projectcalico.org created
customresourcedefinition.apiextensions.k8s.io/caliconodestatuses.crd.projectcalico.org created
customresourcedefinition.apiextensions.k8s.io/clusterinformations.crd.projectcalico.org created
customresourcedefinition.apiextensions.k8s.io/felixconfigurations.crd.projectcalico.org created
customresourcedefinition.apiextensions.k8s.io/globalnetworkpolicies.crd.projectcalico.org created
customresourcedefinition.apiextensions.k8s.io/globalnetworksets.crd.projectcalico.org created
customresourcedefinition.apiextensions.k8s.io/hostendpoints.crd.projectcalico.org created
customresourcedefinition.apiextensions.k8s.io/ipamblocks.crd.projectcalico.org created
customresourcedefinition.apiextensions.k8s.io/ipamconfigs.crd.projectcalico.org created
customresourcedefinition.apiextensions.k8s.io/ipamhandles.crd.projectcalico.org created
customresourcedefinition.apiextensions.k8s.io/ippools.crd.projectcalico.org created
customresourcedefinition.apiextensions.k8s.io/ipreservations.crd.projectcalico.org created
customresourcedefinition.apiextensions.k8s.io/kubecontrollersconfigurations.crd.projectcalico.org created
customresourcedefinition.apiextensions.k8s.io/networkpolicies.crd.projectcalico.org created
```

This strategic application of Calico resources not only facilitated the establishment of network communication between nodes but also notably transitioned the master node to a "ready" state :

```
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   14m   v1.27.6   192.168.9.167   <none>        Ubuntu 20.04.6 LTS   5.4.0-163-generic   containerd://1.7.2
user@master:~$
```

A crucial verification step was conducted by inspecting the status of Calico pods:

```
user@master:~$ kubectl get pods -A
NAMESPACE     NAME                                       READY   STATUS    RESTARTS   AGE
kube-system   calico-kube-controllers-85578c44bf-588xm   1/1     Running   0          3m40s
kube-system   calico-node-65w56                          1/1     Running   0          3m40s
kube-system   coredns-5d78c9869d-cbl6j                   1/1     Running   0          15m
kube-system   coredns-5d78c9869d-dqg8w                   1/1     Running   0          15m
kube-system   etcd-master                                1/1     Running   0          15m
kube-system   kube-apiserver-master                      1/1     Running   0          15m
kube-system   kube-controller-manager-master             1/1     Running   0          15m
kube-system   kube-proxy-wn9wf                           1/1     Running   0          15m
kube-system   kube-scheduler-master                      1/1     Running   0          15m
user@master:~$
```

The list of initialization tokens was acquired with:

```
user@master:~$ kubeadm token list
TOKEN                     TTL         EXPIRES                USAGES                   DESCRIPTION                                                EXTRA GROU
PS
7x8bcj.3qtd0g118xhh5cr0   23h         2023-11-08T12:21:58Z   authentication,signing   The default bootstrap token generated by 'kubeadm init'.   system:boo
tstrappers:kubeadm:default-node-token
user@master:~$
```

In preparation for the worker nodes' integration into the cluster, the public key was extracted using:

```
user@master:~$ openssl x509 -pubkey -noout -in /etc/kubernetes/pki/ca.crt | openssl rsa -pubin -outform der 2>/dev/null |openssl dgst -sha256 - -hex | sed
's/^.* //'
85fd70f24628e233f61b6386c3a79e92c8eb4f1590a3638b1ea2265ba293b9c3
```

Leveraging the obtained token and public key, the process of initiating the worker nodes' membership in the cluster was executed using the following command on each worker node:

- **Example of worker2 :**

```
user@worker2:~$ sudo kubeadm join 192.168.9.167:6443 --token 7x8bcj.3qtd0g118xhh5cr0 --discovery-token-ca-cert-hash sha
56:85fd70f24628e233f61b6386c3a79e92c8eb4f1590a3638b1ea2265ba293b9c3
[preflight] Running pre-flight checks
[preflight] Reading configuration from the cluster...
[preflight] FYI: You can look at this config file with 'kubectl -n kube-system get cm kubeadm-config -o yaml'
[kubelet-start] Writing kubelet configuration to file "/var/lib/kubelet/config.yaml"
[kubelet-start] Writing kubelet environment file with flags to file "/var/lib/kubelet/kubeadm-flags.env"
[kubelet-start] Starting the kubelet
[kubelet-start] Waiting for the kubelet to perform the TLS Bootstrap...

This node has joined the cluster:
* Certificate signing request was sent to apiserver and a response was received.
* The Kubelet was informed of the new secure connection details.

Run 'kubectl get nodes' on the control-plane to see this node join the cluster.
```

Here, **192.168.9.167** corresponds to the address of the master node

Following the successful integration of the worker nodes, a comprehensive overview of the cluster's status was obtained with:

```
user@master:~$ kubectl get nodes
NAME      STATUS   ROLES           AGE     VERSION
master    Ready    control-plane   60m     v1.27.6
worker1   Ready    <none>          5m35s   v1.27.6
worker2   Ready    <none>          36s     v1.27.6
user@master:~$
```

This examination effectively confirmed the successful addition of the worker nodes to the cluster. To delve deeper into the specifics, this command was employed:

```
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   62m     v1.27.6   192.168.9.167   <none>        Ubuntu 20.04.6 LTS   5.4.0-163-generic   containerd://1.7.2
worker1   Ready    <none>          7m29s   v1.27.6   192.168.9.251   <none>        Ubuntu 20.04.6 LTS   5.4.0-163-generic   containerd://1.7.2
worker2   Ready    <none>          2m30s   v1.27.6   192.168.9.46    <none>        Ubuntu 20.04.6 LTS   5.4.0-163-generic   containerd://1.7.2
user@master:~$
```

Additionally, a crucial verification step was conducted by inspecting the status of Calico pods:

```
user@master:~$ kubectl get nodes
NAME      STATUS   ROLES           AGE     VERSION
master    Ready    control-plane   63m     v1.27.6
worker1   Ready    <none>          8m32s   v1.27.6
worker2   Ready    <none>          3m33s   v1.27.6
user@master:~$ kubectl get podes -n calico
error: the server doesn't have a resource type "podes"
user@master:~$ kubectl get pods -n calico
No resources found in calico namespace.
user@master:~$ kubectl get pods -A
NAMESPACE     NAME                                       READY   STATUS    RESTARTS   AGE
kube-system   calico-kube-controllers-85578c44bf-588xm   1/1     Running   0          52m
kube-system   calico-node-65w56                          1/1     Running   0          52m
kube-system   calico-node-9r8kd                          1/1     Running   0          4m20s
kube-system   calico-node-q87s6                          1/1     Running   0          9m19s
kube-system   coredns-5d78c9869d-cbl6j                   1/1     Running   0          63m
kube-system   coredns-5d78c9869d-dqg8w                   1/1     Running   0          63m
kube-system   etcd-master                                1/1     Running   0          63m
kube-system   kube-apiserver-master                      1/1     Running   0          63m
kube-system   kube-controller-manager-master             1/1     Running   0          63m
kube-system   kube-proxy-gbwtb                           1/1     Running   0          4m20s
kube-system   kube-proxy-t5525                           1/1     Running   0          9m19s
kube-system   kube-proxy-wn9wf                           1/1     Running   0          63m
kube-system   kube-scheduler-master                      1/1     Running   0          63m
user@master:~$ kubectl get nodes
NAME      STATUS   ROLES           AGE     VERSION
master    Ready    control-plane   64m     v1.27.6
worker1   Ready    <none>          9m43s   v1.27.6
worker2   Ready    <none>          4m44s   v1.27.6
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   64m     v1.27.6   192.168.9.167   <none>        Ubuntu 20.04.6 LTS   5.4.0-163-generic   containerd://1.7.2
worker1   Ready    <none>          9m55s   v1.27.6   192.168.9.251   <none>        Ubuntu 20.04.6 LTS   5.4.0-163-generic   containerd://1.7.2
worker2   Ready    <none>          4m56s   v1.27.6   192.168.9.46    <none>        Ubuntu 20.04.6 LTS   5.4.0-163-generic   containerd://1.7.2
```

This final command provided specific insights into the status of each node, alongside the status of Calico pods, thereby concluding the process of extending and configuring the Kubernetes cluster.

ClusterIP and NodeIP

Our exploration into ClusterIP and NodePort services begin with the deployment of a straightforward FastAPI application, designed to provide a "hello world" response. To initiate this process, Docker was configured to connect to the designated repository:

```
user@master:~$ docker login
Login with your Docker ID to push and pull images from Docker Hub. If you don't have a Docker ID, head over to https://hub.docker.com to create one.
Username: yxos
Password:
WARNING! Your password will be stored unencrypted in /home/user/.docker/config.json.
Configure a credential helper to remove this warning. See
https://docs.docker.com/engine/reference/commandline/login/#credentials-store

Login Succeeded
user@master:~$
```

Following this, a pivotal step involved creating a secret to grant the master access to resources within the specified repository:

```
Login Succeeded
user@master:~$ kubectl create secret docker-registry repo-secret --docker-username=yxos --docker-password=Dockwhizzer007! --docker-email=jean.cedricsanou@gmail.com
secret/repo-secret created
user@master:~$ kubectl get secret repo-secret --output=yaml
apiVersion: v1
data:
  .dockerconfigjson: eyJhdXRocyI6eyJodHRwczovL2luZGV4LmRvY2tlci5pby92MS8iOnsidXNlcm5hbWUiOiJ5eG9zIiwicGFzc3dvcmQiOiJEb2Nrd2hpenplcjAwNyEiLCJlbWFpbCI6ImplYW4uY2Vkcmljc2Fub3VAZ21haWwuY29tIiwiYXV0aCI6ImVYaHZjenBFYjJOcmQyaHBlbnBsY2pBd055RT0ifX19
kind: Secret
metadata:
  creationTimestamp: "2023-11-07T12:42:16Z"
  name: repo-secret
  namespace: default
  resourceVersion: "2160"
  uid: ced09775-5504-46c8-b4c5-6bcb51bc35af
type: kubernetes.io/dockerconfigjson
```

This credential management ensures a secure and authenticated connection. Subsequently, for organisational clarity and workload distribution within the cluster, worker nodes were strategically labelled:

```
user@master:~$ kubectl label node worker1 PoP=space_1
node/worker1 labeled
user@master:~$ kubectl label node worker2 PoP=space_2
node/worker2 labeled
user@master:~$
```

- **Cluster:**

The designated folder, ClusterIP_service, was downloaded for subsequent deployment. We copy this file from our PC to our VM using this command :

```
U:\Windows\Bureau>scp -r ./ClusterIP user@192.168.37.62:/home/user/
user@192.168.37.62's password:
app_deployment_1.yaml
app_ingress.yaml
app_service_cluster_ip.yaml

U:\Windows\Bureau>
```

We verify that the copy was done successfully :

```
user@master:/$ cd home
user@master:/home$ cd user
user@master:~$ ls
ClusterIP
user@master:~$
```

The ClusterIP services were deployed using the command:

```
user@master:~$ ls
ClusterIP
user@master:~$ kubectl  apply -f ./ClusterIP
deployment.apps/fastapi-app created
Warning: annotation "kubernetes.io/ingress.class" is deprecated, please use 'spec.ingressClassName' instead
ingress.networking.k8s.io/fastapi-app-ingress created
service/fastapi-app-clusterip-service created
user@master:~$
```

The status and characteristics of the deployed pods were examined:

```
user@master:~$ kubectl get pods -o wide
NAME                          READY   STATUS              RESTARTS   AGE   IP       NODE      NOMINATED NODE   READINESS
 GATES
fastapi-app-5476944694-8q2cs   0/1     ContainerCreating   0          53s   <none>   worker1   <none>           <none>
fastapi-app-5476944694-dljfj   0/1     ContainerCreating   0          53s   <none>   worker1   <none>           <none>
fastapi-app-5476944694-xtdcs   0/1     ContainerCreating   0          53s   <none>   worker1   <none>           <none>
user@master:~$
```

Notably, it was observed that some of the pods were not in a 'Ready' state. Services were not attainable.

To access the deployed services, the following command was executed:

```
user@master:~$ kubectl get services -o wide
NAME                          TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)   AGE    SELECTOR
fastapi-app-clusterip-service   ClusterIP   10.102.146.75   <none>        80/TCP    73s    app=fastapi-app
kubernetes                    ClusterIP   10.96.0.1       <none>        443/TCP   125m   <none>
user@master:~$
```

The details of a specific service were explored using:

```
user@master:~$ 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:                10.102.146.75
IPs:               10.102.146.75
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>
```

Attention was given to the 'endpoints' section to determine the location of the application.In our case we choose this endpoint : **172.16.235.129**

A curl request was made from the master node to one of the service endpoints:

```
user@master:~$ curl http://172.16.235.129:5000
{"hello":"world"}user@master:~$
```

We find out our Application response means that the service is reachable .

A new pod, named 'testpod,' was instantiated on the master node utilizing the Nginx image:

```
user@master:~$ curl http://172.16.235.129:5000
{"hello":"world"}user@master:~$  kubectl run testpod --image=nginx
pod/testpod created
user@master:~$
```

The list of all existing pods was obtained for verification:

```
user@master:~$  kubectl get pods
NAME                              READY   STATUS    RESTARTS   AGE
fastapi-app-5476944694-8q2cs      1/1     Running   0          4m58s
fastapi-app-5476944694-dljfj      1/1     Running   0          4m58s
fastapi-app-5476944694-xtdcs      1/1     Running   0          4m58s
testpod                           1/1     Running   0          22s
user@master:~$
```

- We can see that the testpod was correctly added .

Access was gained inside the created 'testpod' using the interactive shell:

```
user@master:~$ kubectl exec -it testpod -- bash
root@testpod:/# curl http://172.16.235.129:5000
{"hello":"world"}root@testpod:/#
```

- We can see that the application is also reachable from the new pod created .

To ensure a clean environment for subsequent operations, all previously deployed resources were systematically deleted:

```
root@testpod:/# exit
exit
command terminated with exit code 127
user@master:~$  kubectl delete -f ./ClusterIP
deployment.apps "fastapi-app" deleted
ingress.networking.k8s.io "fastapi-app-ingress" deleted
service "fastapi-app-clusterip-service" deleted
user@master:~$
```

- **Node:**

On the master node, NodePort services were deployed using the command:

```
user@master:~$ kubectl apply -f ./NodePort
deployment.apps/fastapi-app created
Warning: annotation "kubernetes.io/ingress.class" is deprecated, please use 'spec.ingressClassName' instead
ingress.networking.k8s.io/fastapi-app-ingress created
service/fastapi-app-service created
user@master:~$
```

We copy this file from our PC to our VM using this command :

```
U:\Windows\Bureau>scp -r ./NodePort user@192.168.37.62:/home/user/
user@192.168.37.62's password:
app_deployment_1.yaml                                         100%  545    33.6KB/s   00:00
app_ingress.yaml                                              100%  452    28.7KB/s   00:00
app_service_node_port.yaml                                    100%  202    12.6KB/s   00:00
```

The status and characteristics of the deployed pods were examined:

```
user@master:~$ kubectl get pods -o wide
NAME                              READY   STATUS    RESTARTS   AGE     IP               NODE      NOMINATED NODE   READINESS
 GATES
fastapi-app-5476944694-7fj74      1/1     Running   0          63s     172.16.235.134   worker1   <none>           <none>
fastapi-app-5476944694-hf6pl      1/1     Running   0          63s     172.16.235.133   worker1   <none>           <none>
fastapi-app-5476944694-pf9w2      1/1     Running   0          63s     172.16.235.132   worker1   <none>           <none>
testpod                           1/1     Running   0          7m22s   172.16.189.65    worker2   <none>           <none>
user@master:~$
```

To access the deployed services, the following command was executed:

```
user@master:~$ kubectl get services -o wide
NAME                  TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)        AGE    SELECTOR
fastapi-app-service   NodePort    10.110.139.220   <none>        80:32347/TCP   97s    app=fastapi-app
kubernetes            ClusterIP   10.96.0.1        <none>        443/TCP        136m   <none>
user@master:~$
```

- The details of the services were explored, noting the presence of multiple services. Specifically, services 'fastapi-app-service' with type NodePort and 'kubernetes' with type ClusterIP were observed. The NodePort service had an external IP (Cluster IP) of 10.110.139.220 and was exposing port 80 on the host at port 32347.

The details of a specific service were explored using:

```
user@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:                       10.110.139.220
IPs:                      10.110.139.220
Port:                     <unset>  80/TCP
TargetPort:               5000/TCP
NodePort:                 <unset>  32347/TCP
Endpoints:                172.16.235.132:5000,172.16.235.133:5000,172.16.235.134:5000
Session Affinity:         None
External Traffic Policy:  Cluster
Events:                   <none>
user@master:~$
```

- The observed difference in the NodePort values between the YAML file and the output of kubectl describe services fastapi-app-service is a result of Kubernetes dynamic assignment of NodePort values. In the service YAML file, the port is explicitly set to 80, indicating the desired service port within the cluster.

  However, when deploying a NodePort service, Kubernetes dynamically allocates a high port 32347 in our case for external access. The dynamic assignment is reflected in the output of kubectl describe services, where you see the actual externally accessible NodePort, such as 32347/TCP in your case

```
user@master: ~/NodePort
 GNU nano 4.8                          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
```

A curl request was initiated from the master node to one of the NodePort service endpoints: we choose this endpoint : **172.16.235.132**

```
user@master:~$ curl http://172.16.235.132:5000
{"hello":"world"}user@master:~$
```

To maintain a clean environment for subsequent operations, all previously deployed resources were systematically deleted : $ kubectl delete -f ./NodePort
This step facilitated the removal of all associated resources related to NodePort services.

Part 2: Services deployment in the hybrid cloud/edge

In the process of setting up a Kubernetes cluster on two virtual machines, detailed steps were followed to organise and transfer essential resources for deployment and configuration.

The resources folder, containing necessary files, was placed within the /documents directory created under the user account on both the master and worker nodes.

On the master node, SCP commands were executed from the folder where the resources were downloaded. The command is the following :

```
U:\>scp -r ./Windows/Bureau/tp_kubernetes user@192.168.37.62:/home/user/documents/
user@192.168.37.62's password:
car_client.py                                 100% 2295    71.0KB/s   00:00
cloud_printer.py                              100%  315     0.3KB/s   00:00
controller.py                                 100% 7716   493.0KB/s   00:00
data.json                                     100%   99     0.1KB/s   00:00
gateway.py                                    100% 2304     2.3KB/s   00:00
listener.py                                   100%  872    54.5KB/s   00:00
publisher.py                                  100%  752     0.7KB/s   00:00
cloud_deployment.yaml                         100%  531     0.5KB/s   00:00
cloud_ingress.yaml                            100%  448     0.4KB/s   00:00
cloud_service.yaml                            100%  198     0.2KB/s   00:00
edge_deployment_1.yaml                        100%  526    32.9KB/s   00:00
edge_ingress.yaml                             100%  446     0.4KB/s   00:00
edge_service.yaml                             100%  196     0.2KB/s   00:00
edge_deployment_2.yaml                        100%  529    32.3KB/s   00:00
sla_manager.py                                100% 1598   102.2KB/s   00:00
sla_printer.py                                100%  298     0.3KB/s   00:00
```

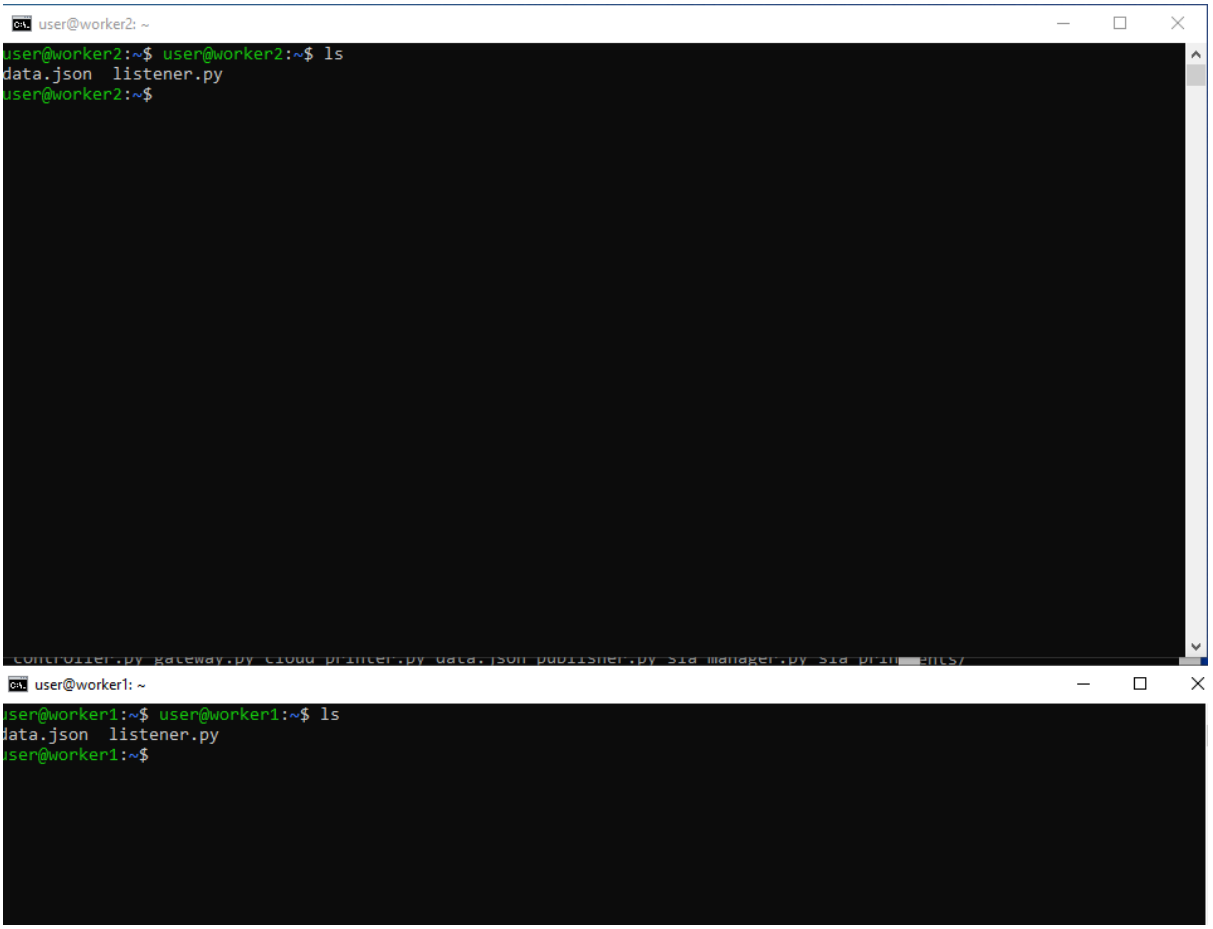
```
user@master:~$ ls
ClusterIP  NodePort  documents
user@master:~$
```

For the worker node, the SCP command was employed to transfer relevant files, including listener.py and data.json, to the /home/user/documents/ directory on the worker VM.

```
U:\Windows\Bureau>cd tp_kubernetes

U:\Windows\Bureau\tp_kubernetes>scp car_client.py controller.py gateway.py cloud_printer.py data.json publisher.py sla_m
anager.py sla_printer.py user@192.168.37.62:/home/user/documents/
user@192.168.37.62's password:
car_client.py                                 100% 2295   143.1KB/s   00:00
controller.py                                 100% 7716   483.8KB/s   00:00
gateway.py                                    100% 2304     2.3KB/s   00:00
cloud_printer.py                              100%  315     0.3KB/s   00:00
data.json                                     100%   99     6.2KB/s   00:00
publisher.py                                  100%  752     0.7KB/s   00:00
sla_manager.py                                100% 1598     1.6KB/s   00:00
sla_printer.py                                100%  298     0.3KB/s   00:00
```

```
U:\Windows\Bureau\tp_kubernetes>scp listener.py data.json user@192.168.37.69:/home/user/
user@192.168.37.69's password:
listener.py                                                   100%  872    54.5KB/s   00:00
data.json                                                     100%   99     0.1KB/s   00:00

U:\Windows\Bureau\tp_kubernetes>scp listener.py data.json user@192.168.37.186:/home/user/
user@192.168.37.186's password:
listener.py                                                   100%  872    54.7KB/s   00:00
data.json                                                     100%   99     0.1KB/s   00:00

U:\Windows\Bureau\tp_kubernetes>
```

We verify that the files exist on the directory :



In preparation for the car's decision service deployment, installations of Python libraries (fastapi, redis, and uvicorn) were executed on the master node:

```
user@master:~$  sudo apt install python3-pip
[sudo] password for user:
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following additional packages will be installed:
```

```
user@master:~$ pip3 install fastapi redis uvicorn
Collecting fastapi
  Downloading fastapi-0.104.1-py3-none-any.whl (92 kB)
     |████████████████████████████████| 92 kB 879 kB/s
Collecting redis
  Downloading redis-5.0.1-py3-none-any.whl (250 kB)
     |████████████████████████████████| 250 kB 13.9 MB/s
Collecting uvicorn
  Downloading uvicorn-0.24.0.post1-py3-none-any.whl (59 kB)
     |████████████████████████████████| 59 kB 6.2 MB/s
Collecting pydantic!=1.8,!=1.8.1,!=2.0.0,!=2.0.1,!=2.1.0,<3.0.0,>=1.7.4
  Downloading pydantic-2.4.2-py3-none-any.whl (395 kB)
     |████████████████████████████████| 395 kB 30.2 MB/s
Collecting anyio<4.0.0,>=3.7.1
```

We try to install python in another master Bash to verify that it's already existing or not :

```
user@master:~$  sudo apt install python3-pip
[sudo] password for user:
Reading package lists... Done
Building dependency tree
Reading state information... Done
python3-pip is already the newest version (20.0.2-5ubuntu1.9).
0 upgraded, 0 newly installed, 0 to remove and 26 not upgraded.
user@master:~$
```

The data.json file in the resources folder was updated with master and worker IP information. Additionally, listener.py was executed on both worker nodes to participate in the migration process.

Two Redis databases, cloud_db and sla_db, were initiated on the master node for cloud service and service management, respectively:

user@master: ~/documents

```
 GNU nano 4.8                                                  data.json
{
   "master_ip":"192.168.9.167",
   "worker_1_ip":"192.168.9.251",
   "worker_2_ip":"192.168.9.46"
}
```

The status of pods to be deployed was continuously monitored using the command:

```
$ watch kubectl get pods -o wide
```

Car Microservices Deployment and Configuration Report:

Upon the successful setup of the Kubernetes cluster, the deployment of car microservices was undertaken, guided by an in-depth understanding of key Kubernetes concepts. Kubernetes serves as an orchestrator for containers, managing their workload within a cluster. The fundamental building blocks are pods, containers running inside these pods, and deployments that handle the replication of pods for scalability.

- The car initiates a /init request to describe the outsourced decision service, specifying latency tolerances, remote data gathering service, and the gateway for communication.
- The controller orchestrates the deployment as follows:

- **Gateway**: A Docker image with the gateway application is deployed on the master node using the command: **$ uvicorn gateway:app --host 0.0.0.0 --port 8000**

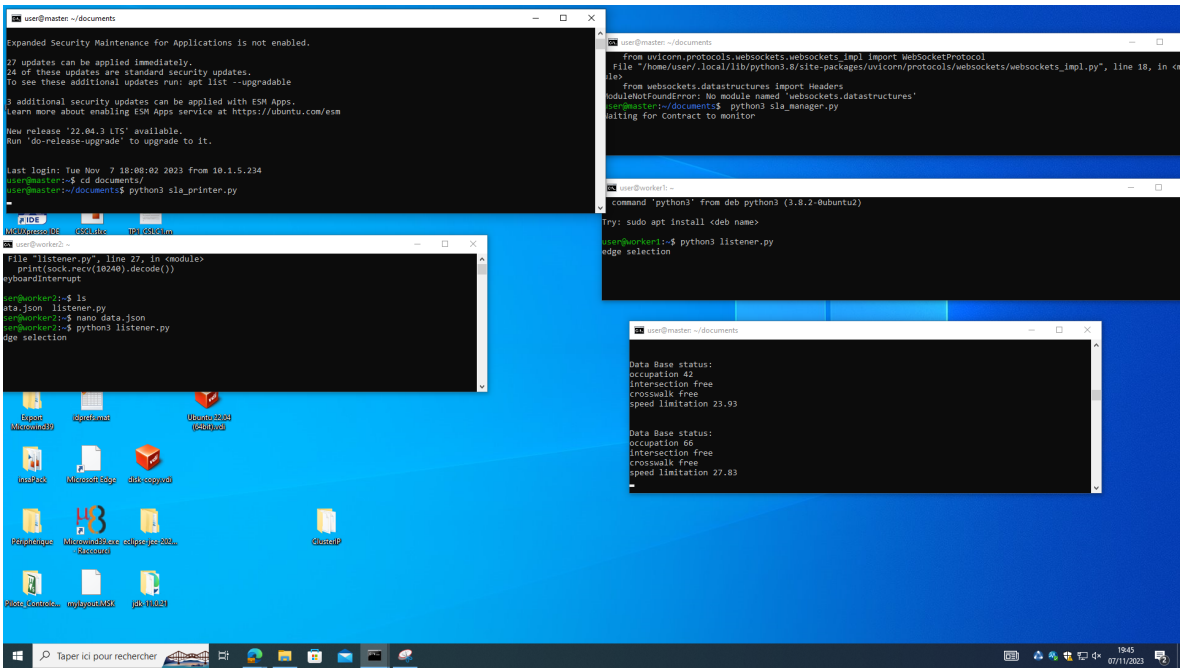
- **Decision Service**: A pod deployment object is created in the node labeled space_1, accompanied by a Kubernetes service and an ingress route for load balancing between pods running the same application.

The deployment command is: **$ uvicorn controller:app --host 0.0.0.0 --port 7000**

- **Data Gathering Service:** A pod is deployed in a distant cloud, providing additional information about the surrounding traffic.

```
user@Master:~$ cd documents
user@Master:~/documents$ uvicorn gateway:app --host 0.0.0.0 --port 8000
INFO:     Started server process [2677280]
INFO:     Waiting for application startup.
INFO:     Application startup complete.
INFO:     Uvicorn running on http://0.0.0.0:8000 (Press CTRL+C to quit)
```

The final steps involve running the sla_manager and car_client applications on the master node: **$ python3 sla_manager.py / $ python3 car_client.py**



Following the implementation of services within the Kubernetes cluster, it is noteworthy that some services were successfully deployed, while others encountered challenges, as evident in the previously provided screenshot. The disparity in outcomes may be attributed to various factors, including specific configurations, missing dependencies, or other service-specific issues.

Due to time constraints, a detailed identification of the error sources was not feasible; however, a fundamental understanding of the principles underlying this step was gained.

Conclusion :

The initial lab provided us with a comprehensive understanding of virtualization technologies, encompassing both theoretical concepts and practical implementations. From creating and managing Virtual Machines (VMs) to deploying distributed applications on cloud platforms like OpenStack, the lab successfully achieved its objectives. The hands-on experience gained is crucial in navigating the dynamic landscape of virtualization and cloud infrastructure management.

Moving on to the lab on orchestrating services in a hybrid cloud/edge environment, we were guided through the practical implementation of Kubernetes for service management, with a specific focus on a next-generation autonomous car case study. Key aspects covered included infrastructure setup, Kubernetes cluster deployment, essential Kubernetes concepts, and hands-on exercises deploying various service types, including microservices tailored for the autonomous car scenario.

The lab effectively demonstrated the adaptability of the infrastructure, especially in managing service migration to optimise performance as the vehicle moves. This experience equipped us with valuable skills in deploying, scaling, and migrating services within a dynamic hybrid cloud/edge computing environment. The practical insights gained address real-world challenges, offering a holistic perspective on the intricacies of managing services in the context of autonomous vehicles.