Cloud Computing: Adaptability and Autonomic Management

Lab 1 : Introduction to Cloud Hypervisors

Access link : tiny.cc/TP_Cloud

LIEVRE Agathe
NGUYEN Assia

5A ISS 2021/2022

Introduction

The general purpose of this lab is to enhance your knowledge of concepts and technologies for virtualization techniques. These techniques are used in IT environments that support the provisioning (i.e. development, deployment, management) of novel software systems (with their potential constraints such as QoS and security). These environments are typically dynamic and distributed

Theoretical part
(objectives 1 to 3)

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

There are two types of virtualisation hosts (i.e. VMs and CTs). These hosts are depicted in Figure 1.

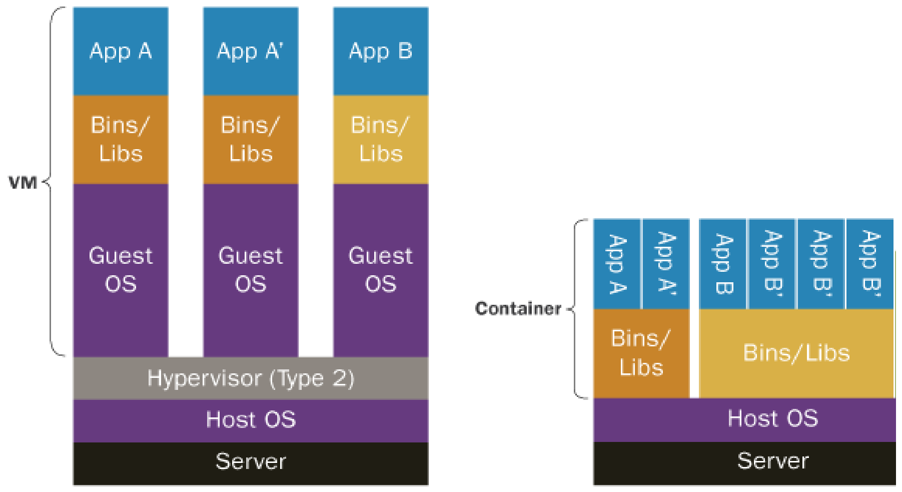


Figure 1: VM vs CT

- The figure on the left represents the stack layers of a virtual machine. It deploys, thanks to the hypervisor layer, different virtual hosts on the same machine. Each host possesses its individual OS. The OS can execute applications and complex processes concurrently.
  The container, on the right, also runs on a physical server but shares only one OS kernel. Therefore, it can execute several applications that use shared binaries/libraries.

- Since each virtual machine contains several virtual hosts, the size needed on the hardware is tens of Gbytes. Therefore, it is slower to run. Meanwhile, the container only needs a few megabytes to run and is thus more agile. It is more adapted to specific and light tasks

	VMs	Containers
Virtualization cost, memory size and CPU	++ since each OS has to be virtualized, the memory size = tens of GBs	-- since every app shares the bins/libs of the host OS (size of images ~ tens of MBs)
Usage of CPU, memory and network for a given application,	++ since all the virtual hosts use the CPU and memory concurrently	-- one OS = less usage of CPU, memory and network
Security for the application (access right, resources sharing, etc.),	++ better isolation because each virtual OS is independant	-- the apps are not isolated from each other since they share ressources
Performances (response time),	-- The server is slower since it has several OS to run (slow to boot)	++ Short response time since having one OS takes up less space
Tooling for the continuous integration support	-- each VM requires its own configuration	++ tooling configuration easier since the bins/libs are shared

Application developers have to prioritize the most agile virtualisation host since they work on the applicative layer. They need high performance, efficiency and the easiest deployment process possible. They also need to run a lot of applications.
For those reasons, they will prefer a CT over a VM. Containers allow developers to easily change and improve their apps.

Infrastructure administrators want the best security possible because they have to guarantee the viability and integrity of the system. They will prefer a virtualisation host that is less prone to security breach so they will choose a Virtual Machine since it has the best isolation out of the two options.

2. Similarities and differences between the existing CT types

Different CT technologies are available in the market (e.g. LXC/LXD, Docker, Rocket, OpenVZ, runC, containerd, systemd-nspawn). Their respective positioning is not obvious, but comparative analyses are available online, such as the one displayed in Figure 2.

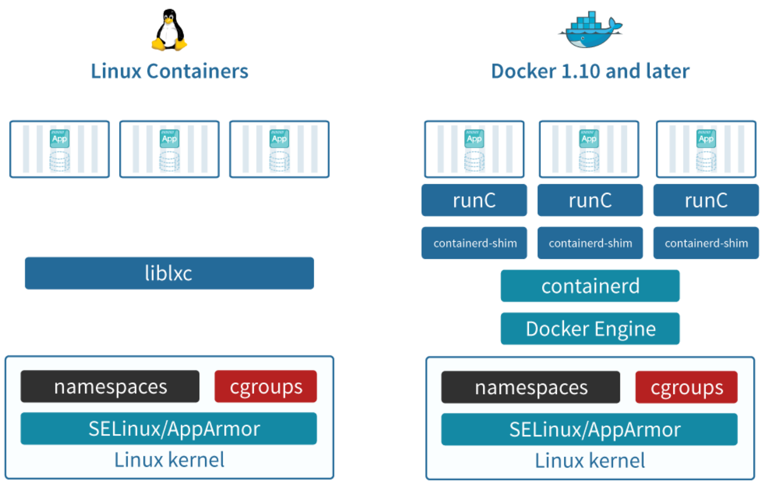


Figure 2 : Linux Lxc vs Docker

These technologies can however be compared based on the following criteria (non-exhaustive list):

- Application isolation and resources, from a multi-tenancy point of view,
- Containerization level (e.g. operating system, application),
- Tooling (e.g. API, continuous integration, or service composition).

Application isolation and resources : it is the containers' degree of separation on the machine. For example, the highest degree of separation is that of the VMs since they do not share the ressources.

Containerization level : it is how much resources are shared between the containers. For example, the level for the VMs is null because the hardware is simulated. If the hardware is shared (System container, Linux LXC), the level is low. If the OS and runtime environment are shared (App container, Docker), the level is high.

Tooling : it is how many sets of plugins and tools that you can have in the containers
.

	Application isolation and resources,	Containerization level	Tooling	One or multiple process
Linux LXC	++ can execute container as non admin users one container can have several applications	System container only supported by linux -> medium level	REST API	Multiple
Docker	-- Docker daemons can only have a single application per container	Application container Portability (supported by windows/linux/macOS) -> high level	SDK automatisation tools for building	One
Rocket	one container can have several applications	Application container -> high level	API	Multiple
OpenVZ	run multiple Operating Systems virtually	System container Operating system level -> medium level	API	Multiple
systemd - nspawn	can execute container as non admin users if only one user is needed	OS-Level	Not enough tooling to manage the containers	Multiple

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

There are two main categories of hypervisors, referred to as type 1 and type 2.

Hypervisors :

- Type 1, Bare-metal (Native or hardware-level), it is executed directly on the hardware layer. Only one hypervisor can be installed on a physical machine.
- Type 2, Hosted (OS-level), allows us to use apps on different levels. It is possible to execute user processes directly on the OS or in a VM controlled by an hypervisor which runs separately from the host machine.

VirtualBox = type 2 because it has be installed on an OS and not be installed on a system as an OS
OpenStack = type 1

4. Difference between the two main network connection modes for virtualization hosts

With some hypervisors, multiple possibilities exist to connect a VM/CT to the Internet, via the host machine (in this case your desktop). The two main modes are the following:

- *NAT mode*: It is the default mode. It does not require any particular configuration. In this mode, the VM/CT is connected to a private IP network (i.e. a private address), and uses a virtual router (managed by the hypervisor) running on the host machine to communicate outside of the network:
  - This router executes what is equivalent to a NAT function (only *postrouting*) allowing the VM to reach the host machine or the outside;
  - However, the VM cannot be reached from the host (and by any another VM hosted on the same machine): to make it accessible from outside, it is necessary to deploy port forwarding (*prerouting*).
- *Bridge mode*, the most widely used (yet not systematically), in which the VM/CT sees itself virtually connected to the local network of its host (see Figure 3): it has an IP address identifying it on the host network, and can access the Internet (or is accessed) as the host.

NB: Other less used modes exist, such as Private Network in VirtualBox, that you can try out.

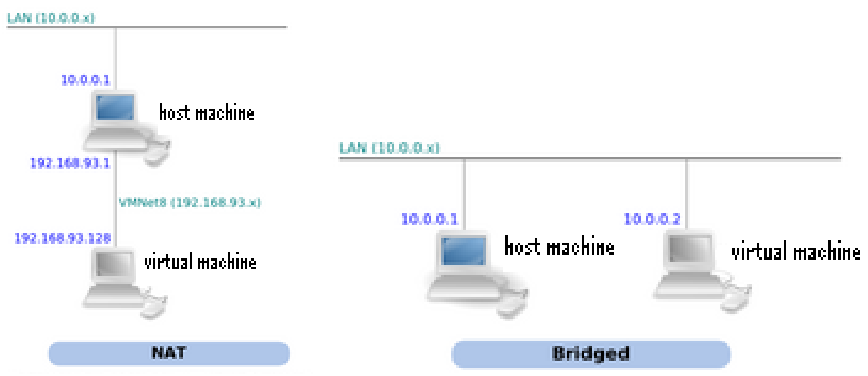


Figure 3 : Mode NAT vs Mode Bridge (the most usual)

Practical part
(objectives 4 to 7)

1. Tasks related to objectives 4 and 5

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

In this part, you will use the VirtualBox hypervisor in NAT mode to connect a VM to the network. The bridge mode will be used in the second part of the lab, with another hypervisor (OpenStack).

First part: Creating and configuring a VM

We created and configured our VM.

Second part: Testing the VM connectivity

Identify the IP address attributed to the VM using ifconfig (in Linux command line), and compare it to the host address (ip config in the command line). What do you observe.

IP address of the host : 10.1.5.235

IP address of the VM : 10.0.2.15

With a ping command (or any other command of your choice) :

- Check the connectivity from the VM to the outside. What do you observe?

The ping command is successful. The router executes the NAT function which allows the VM to reach the outside.

- Check the connectivity from your neighbour's host to your hosted VM. What do you observe?

The ping command is not successful. Indeed, the VM is not accessible from the outside.

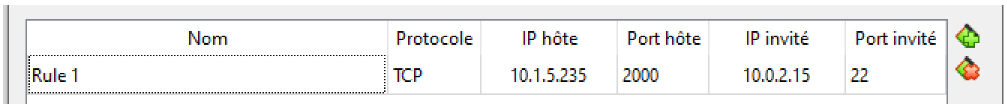
- Check the connectivity from your host to the hosted VM. What do you observe?

The ping command is not successful. The reason is the same as before.

So, the VM has access to the outside thanks to the NAT router but the opposite is not possible. Indeed, we have to implement port forwarding.
(A VM cannot virtualize a network card)

Third part: Set up the "missing" connectivity

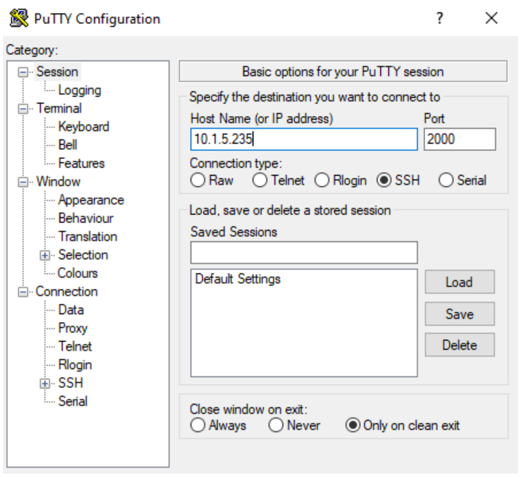
To implement the port forwarding, we have to add a new rule of port forwarding with these settings :



Nom	Protocole	IP hôte	Port hôte	IP invité	Port invité
Rule 1	TCP	10.1.5.235	2000	10.0.2.15	22

Figure 4 : configuration of the new rule

The port forwarding will transfer every packet the port 2000 receives to the VM's port 22. To check if the port forwarding is working, we used PuttY to connect in SSH the machine host to the VM.



PuTTY Configuration

Basic options for your PuTTY session

Specify the destination you want to connect to

Host Name (or IP address): 10.1.5.235

Port: 2000

Connection type: Raw Telnet Rlogin [✓ SSH] Serial

Saved Sessions: Default Settings

Close window on exit: Always Never [✓ Only on clean exit]

Figure 5 : PuttY configuration

Then we use the PuttY to login.

Fourth part: VM duplication

To create a new (clone) VM with the same disk file, we ran the following command:

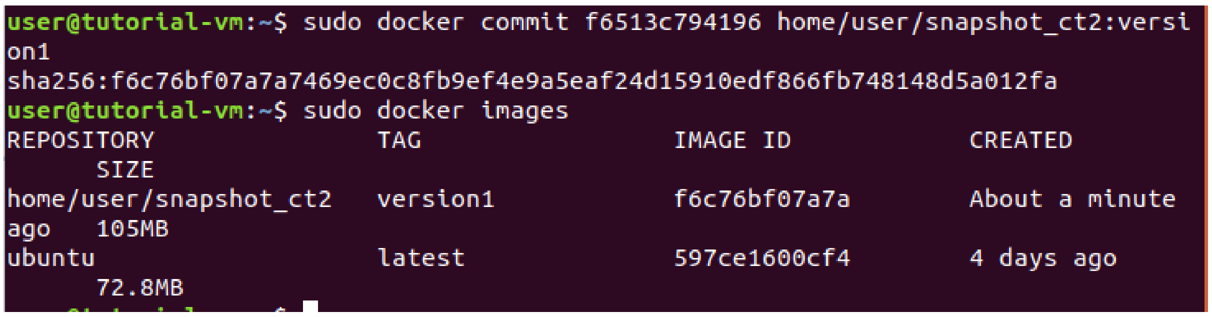
```
"C:\Program Files\Oracle\VirtualBox\VBoxManage.exe" VBoxManage
clonemedium "C:\User\asnguyen\Downloads\disk.vmdk\disk.vmdk"
"C:\User\asnguyen\Downloads\disk_copy.vmdk"
```

Then, we created the new VM with the disk_copy.vmdk file.

Fifth part: Docker containers provisioning

We had to follow the next steps :

1. Get an ubuntu image
2. Execute an instance of the ubuntu image
3. Install the required connectivity testing tools
4. Check the connectivity (through ping) with the newly instantiated Docker
    a. What is the Docker IP address : 172.17.0.2
    b. Ping an Internet resource from Docker : successful
    c. Ping the VM from Docker : successful
    d. Ping the Docker from the VM : successful
5. Execute a new instance (CT2) of the ubuntu Docker.
6. Install nano (text editor) on CT2
7. Make a snapshot of CT2



```
user@tutorial-vm:~$ sudo docker commit f6513c794196 home/user/snapshot_ct2:versi
on1
sha256:f6c76bf07a7a7469ec0c8fb9ef4e9a5eaf24d15910edf866fb748148d5a012fa
user@tutorial-vm:~$ sudo docker images
REPOSITORY                TAG                 IMAGE ID            CREATED
      SIZE
home/user/snapshot_ct2    version1            f6c76bf07a7a        About a minute
ago    105MB
ubuntu                    latest              597ce1600cf4        4 days ago
      72.8MB
```

Figure 6: parameters of the snapshot

8. Stop and terminate CT2 (sudo stop ct2)
9. List the available Docker images in the VM
10. Execute a new instance (CT3) from the snapshot that you previously created from CT2 using the run command. Do you still have nano installed on CT3 ? Since we created a snapshot from CT2 and used it to create a new instance (CT3), CT3 will also possess Nano.
11. Docker enables “recipes” sharing to create persistent images (as a second alternative of snapshots).

https://docs.docker.com/engine/reference/commandline/build/

2. Expected work for objectives 6 and 7

First part : CT creation and configuration on OpenStack

On OpenStack, we created a private network with a sub-network with the following address : 10.1.0.0/24.Then, we created and associated a VM to this network.

Second part: Connectivity test

The IP address of the VM is 10.1.0.108. It belongs to the range of addresses created with the private network we created.
We then created a router to link the public and private network by adding an interface.

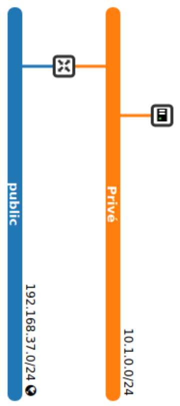
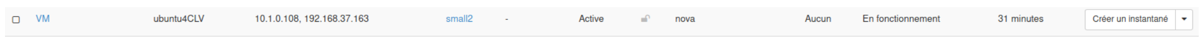


Figure 7: network topology

Using a Ping (or any other appropriate command) :

- Check the connectivity from the VM to the desktop: it is successful.
- Check the connectivity from the desktop to the VM: it is unsuccessful. The VM's IP address is non routable since OpenStack created a Virtual Local Network. To reach the VM from outside, we have to allocate a floating IP address to the VM.

Instance name	Image name	IP address	Flavor	Key pair	Status	Availibility zone	Task	Power state
VM	ubuntu4CLV	10.1.0.108 192.168.37.163	small2	-	Active	nova	Aucun	En fonctionnement



```
VM    ubuntu4CLV    10.1.0.108, 192.168.37.163    small2    -    Active    nova    Aucun    En fonctionnement    31 minutes    Créer un instantané
```

Figure 8: parameters of the VM

Third part: Snapshot, restore and resize a VM

First, we resized a running VM. If the selected size is inferior to the one used when creating the instance and the size is too small to contain the VM, the resizing doesn't work. The operation succeeded when we chose a superior size or an inferior size big enough to contain the VM.

Then, we tried resizing it while it was turned off and it also succeeded. This shows the flexibility of OpenStack since we can resize while the VM is on. We also took a snapshot of the VM. We compared it to the included material/software.

	Type	Visibility	Disk format	Size
ubuntu4CLV	Image	Public	QCOW2	3,66 Go
Snapshot	Snapshot	Private	QCOW2	3,78 Go

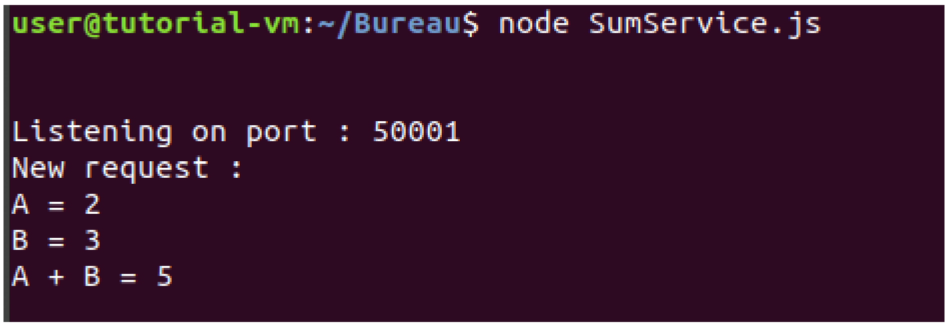
Finally, we restored the VM from the snapshot. Some flavors were not available when we configured the VM. (IP : 10.1.0.202)

3. Expected work for objectives 8 and 9

Part one : OpenStack client installation

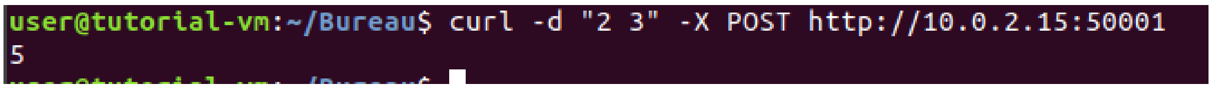
We installed the REST client on our Ubuntu VM. We configured it with the RC file we downloaded from the OpenStack website.

Part two : Web 2-tier application topology and specification



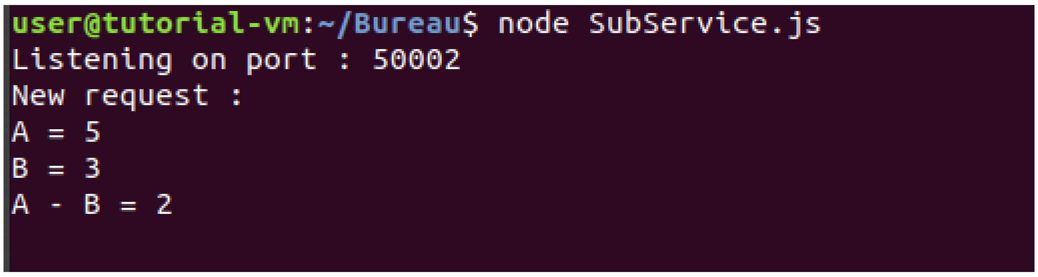
```
user@tutorial-vm:~/Bureau$ node SumService.js


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

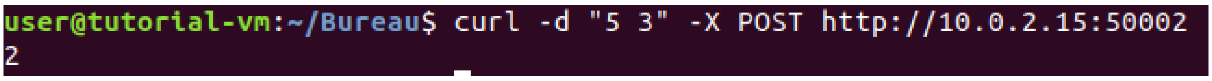


```
user@tutorial-vm:~/Bureau$ curl -d "2 3" -X POST http://10.0.2.15:50001
5
```

Figure 9: test of the SumService.js

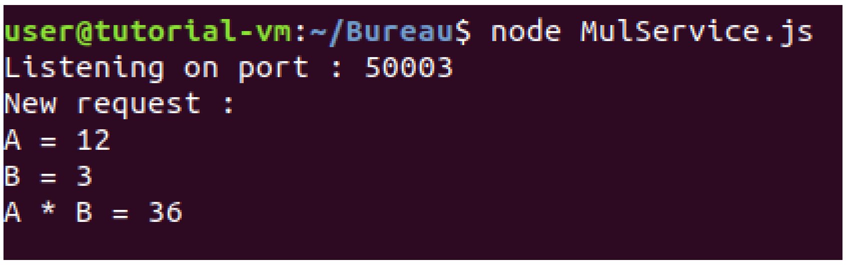


```
user@tutorial-vm:~/Bureau$ node SubService.js
Listening on port : 50002
New request :
A = 5
B = 3
A - B = 2
```

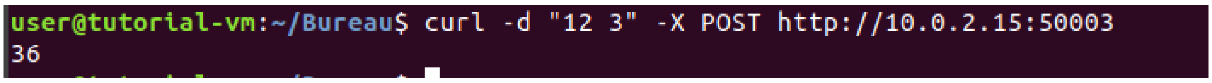


```
user@tutorial-vm:~/Bureau$ curl -d "5 3" -X POST http://10.0.2.15:50002
2
```

Figure 10: test of the SubService.js

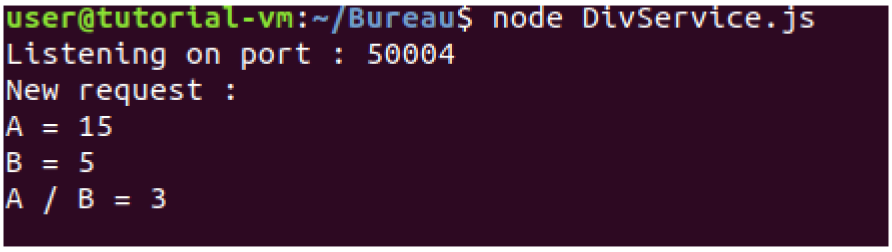


```
user@tutorial-vm:~/Bureau$ node MulService.js
Listening on port : 50003
New request :
A = 12
B = 3
A * B = 36
```

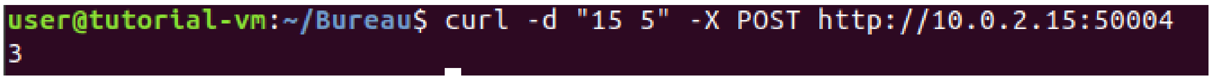


```
user@tutorial-vm:~/Bureau$ curl -d "12 3" -X POST http://10.0.2.15:50003
36
```

Figure 11: test of the MulService.js



```
user@tutorial-vm:~/Bureau$ node DivService.js
Listening on port : 50004
New request :
A = 15
B = 5
A / B = 3
```



```
user@tutorial-vm:~/Bureau$ curl -d "15 5" -X POST http://10.0.2.15:50004
3
```

Figure 12: test of the DivService.js

We needed to modify the CalculatorService.js. We changed the service IP port to "http://10.0.2.15:50000/1/2/3/4" + port "50000"

```
sudo apt update
sudo apt install nodejs npm
npm install sync-request
```

We had to add access authorizations for the 50001, 50002, 50003, 50004 ports because, by default, Openstack blocks the traffic.

Part three: Deploy the Calculator application on OpenStack

We created 5 VMS, one for each application.

4. Expected work for objectives 10 and 11

In this step, as NaaS (Network as-a-Service) provider, we propose to process a prospective client request and deliver on-demand virtualized network topologies that include virtual hosts, routers, and communication functions implementing network levels from 3 to 7.

Part one: The client requirements and target network topology

From now on, the results we obtained are the same as those of Thomas Berton and Aymen Boukezzata because Thomas and Assia were working for their apprenticeship so Aymen and Agathe worked together.

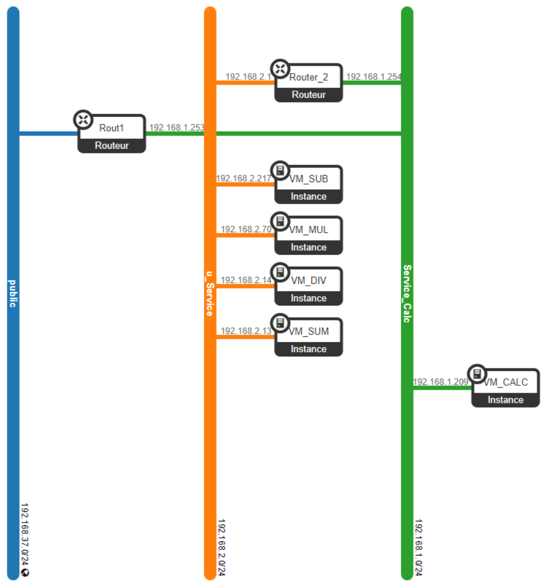


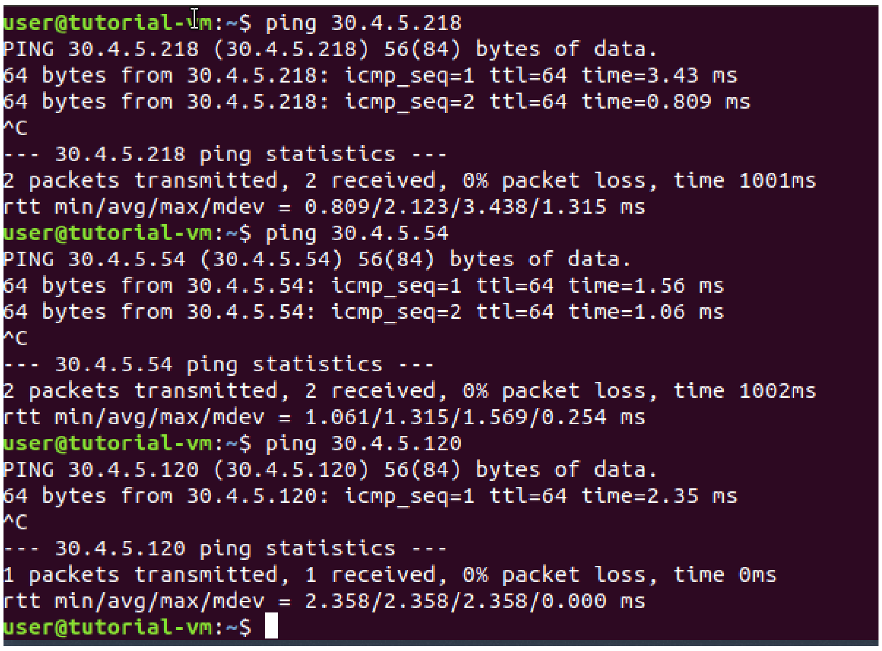
Figure 13: Network topology

Part two: Deployment of the target topology

We deployed each node.js service on each VM with the following command: wget **[lien]**

SumService: http://homepages.laas.fr/smedjiah/tmp/SumService.js
SubService: http://homepages.laas.fr/smedjiah/tmp/SubService.js
MulService: http://homepages.laas.fr/smedjiah/tmp/MulService.js
DivService: http://homepages.laas.fr/smedjiah/tmp/DivService.js
CalculatorService: http://homepages.laas.fr/smedjiah/tmp/CalculatorService.js

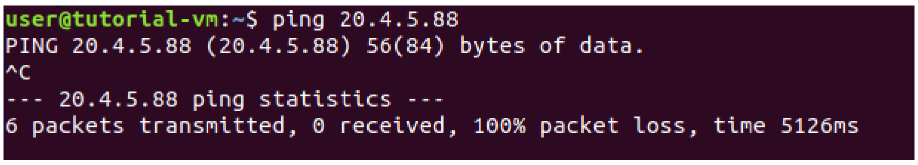
Part three: Configuration of the topology and execution of the services



```
user@tutorial-vm:~$ ping 30.4.5.218
PING 30.4.5.218 (30.4.5.218) 56(84) bytes of data.
64 bytes from 30.4.5.218: icmp_seq=1 ttl=64 time=3.43 ms
64 bytes from 30.4.5.218: icmp_seq=2 ttl=64 time=0.809 ms
^C
--- 30.4.5.218 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1001ms
rtt min/avg/max/mdev = 0.809/2.123/3.438/1.315 ms
user@tutorial-vm:~$ ping 30.4.5.54
PING 30.4.5.54 (30.4.5.54) 56(84) bytes of data.
64 bytes from 30.4.5.54: icmp_seq=1 ttl=64 time=1.56 ms
64 bytes from 30.4.5.54: icmp_seq=2 ttl=64 time=1.06 ms
^C
--- 30.4.5.54 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1002ms
rtt min/avg/max/mdev = 1.061/1.315/1.569/0.254 ms
user@tutorial-vm:~$ ping 30.4.5.120
PING 30.4.5.120 (30.4.5.120) 56(84) bytes of data.
64 bytes from 30.4.5.120: icmp_seq=1 ttl=64 time=2.35 ms
^C
--- 30.4.5.120 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 2.358/2.358/2.358/0.000 ms
user@tutorial-vm:~$
```

Figure 14: pinging from VM of the u_Service network to others of the same network

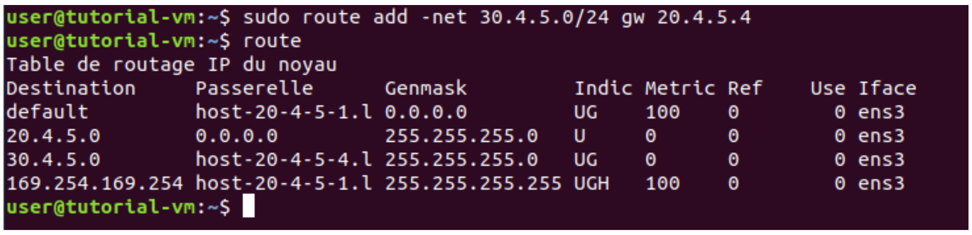
We successfully pinged from one VM the other VM from the same network (u_Service). However, when on the network “u_Service”, we cannot access a VM from another network even if they are linked by a gateway. We have to specify a path for the data to route.



```
user@tutorial-vm:~$ ping 20.4.5.88
PING 20.4.5.88 (20.4.5.88) 56(84) bytes of data.
^C
--- 20.4.5.88 ping statistics ---
6 packets transmitted, 0 received, 100% packet loss, time 5126ms
```

Figure 15: pinging from VM_CALC to VM_SUM

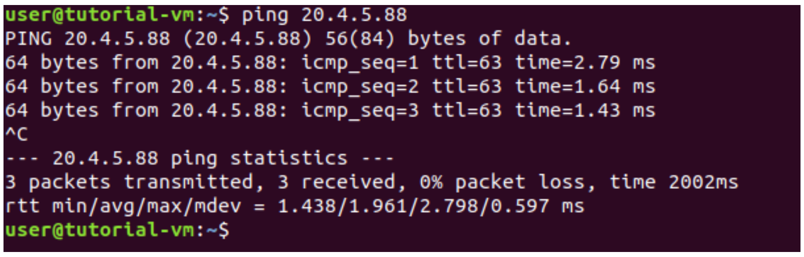
In the VM_CALC we added the gateway “Router_2” as the route to access the 30.4.5.0 network.



```
user@tutorial-vm:~$ sudo route add -net 30.4.5.0/24 gw 20.4.5.4
user@tutorial-vm:~$ route
Table de routage IP du noyau
Destination     Passerelle      Genmask         Indic Metric Ref    Use Iface
default         host-20-4-5-1.l 0.0.0.0         UG    100    0        0 ens3
20.4.5.0        0.0.0.0         255.255.255.0   U     0      0        0 ens3
30.4.5.0        host-20-4-5-4.l 255.255.255.0   UG    0      0        0 ens3
169.254.169.254 host-20-4-5-1.l 255.255.255.255 UGH   100    0        0 ens3
user@tutorial-vm:~$ 
```

Figure 16: adding of the route in the routing table of VM_CALC

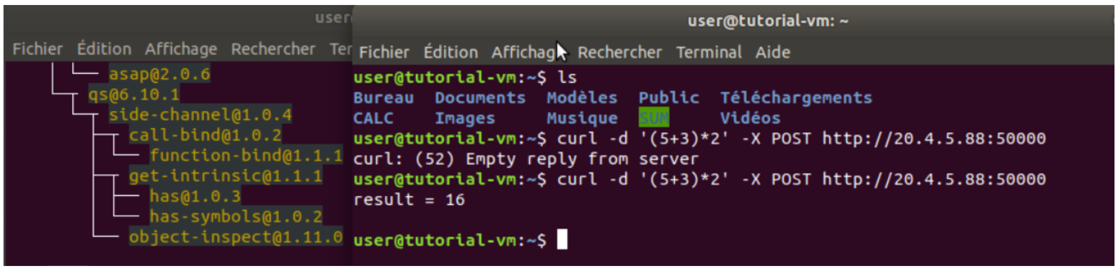
We were then able to ping the other VM from VM_CALC and also ping VM_CALC from the other VM, in this case the VM_SUM:



```
user@tutorial-vm:~$ ping 20.4.5.88
PING 20.4.5.88 (20.4.5.88) 56(84) bytes of data.
64 bytes from 20.4.5.88: icmp_seq=1 ttl=63 time=2.79 ms
64 bytes from 20.4.5.88: icmp_seq=2 ttl=63 time=1.64 ms
64 bytes from 20.4.5.88: icmp_seq=3 ttl=63 time=1.43 ms
^C
--- 20.4.5.88 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2002ms
rtt min/avg/max/mdev = 1.438/1.961/2.798/0.597 ms
user@tutorial-vm:~$
```

Figure 17: pinging from VM_CALC to VM_SUM

We could then try the CalculatorService on the VM_CALC, and it worked:

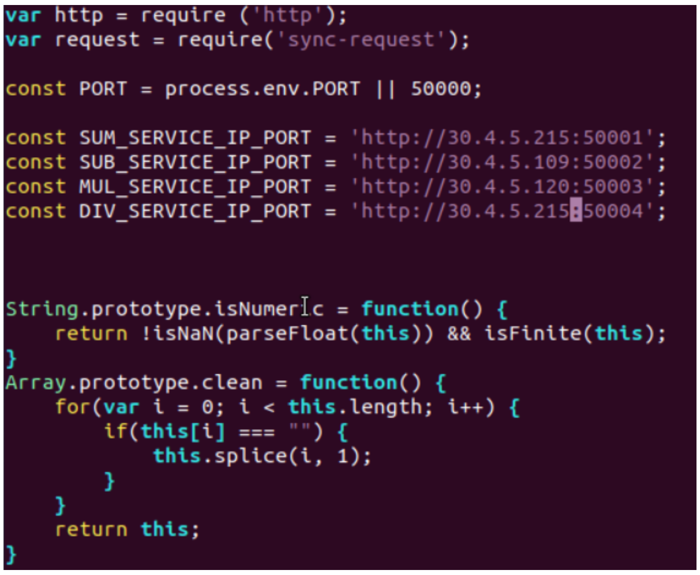


```
  │ └── asap@2.0.6
  ┌ qs@6.10.1
  └┬ side-channel@1.0.4
   ├┬ call-bind@1.0.2
   │└── function-bind@1.1.1
   ├┬ get-intrinsic@1.1.1
   │├── has@1.0.3
   │└── has-symbols@1.0.2
   └── object-inspect@1.11.0
```

```
user@tutorial-vm:~$ ls
Bureau  Documents  Modèles  Public  Téléchargements
CALC    Images     Musique  SUM     Vidéos
user@tutorial-vm:~$ curl -d '(5+3)*2' -X POST http://20.4.5.88:50000
curl: (52) Empty reply from server
user@tutorial-vm:~$ curl -d '(5+3)*2' -X POST http://20.4.5.88:50000
result = 16

user@tutorial-vm:~$ 
```

Figure 18: Calculating form the VM_CALC



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

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

const SUM_SERVICE_IP_PORT = 'http://30.4.5.215:50001';
const SUB_SERVICE_IP_PORT = 'http://30.4.5.109:50002';
const MUL_SERVICE_IP_PORT = 'http://30.4.5.120:50003';
const DIV_SERVICE_IP_PORT = 'http://30.4.5.215:50004';


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

Figure 19: CalculatorService code modifications