Compte rendu TP Cloud

TRUONG - EJIGU ISS B1 Groupe 3

Theoretical part

Objectives 1 to 3

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

Two virtualization options are today available: Virtual machines and containers. They differ from each other by the architecture the systems require to execute one or another. Below a comparison of two concepts.

	VM	CT
virtualization cost, taking into consideration memory size and CPU	Hypervisor and Guest OS have background tasks that use memory and CPU from the Host OS. This will make other applications on the guest OS have less ressource	Without the OS, and the hypervisor, deploying an environment is much lighter in terms of memory and CPU. An engine might run on top of the Host OS but it remains lighter compared to to VM hypervisors
Usage of CPU, memory and network for a given application,	VM uses much more resources than CT since VM runs separate operating systems, and every system call has to go through the virtualization layer. Especially memory usage as VM consumes memory even then it isn’t running any user process. CPU usage demandes less resource since CPU virtualization is relatively cheap.	Since only application is executed in its environment, there are no background tasks like we would find in a full OS. Network usage from these background tasks is also reduces
Security for the application (access right, resources sharing, etc.)	Resources are allocated for the VM. If the hypervisor has no security vulnerabilities, the host and guest OS	Resources are shared between guest os and containers running on OS. Visibility on the used

	should theoretically be maintained separately, as well as all other VMs One should be only accessible through shared files and the network	ressources is not as easily done as with VM’s. Security wise, since all containers share resources, one being vulnerable might make others vulnerable
Performances (response time)	In terms of performance, VM takes much longer than CT since VM needs to start an entire OS, including a full boot process, startup of services...	CT can reduce time required to deploy and run an app. Since the OS is already up and running, the app can start up without any noticeable delay.
Tooling for the continuous integration support	Exportable and can be sent, image size is quite often large since it contains a whole OS	Light export, the whole image doesn't always need to be sent. We can upgrade previous containers or use base containers hosted on the cloud (container image repositories like Docker hub)

	VM	CT
Developers POV	Using a VM doesn’t need any other skills than using an OS. The Developer can (using a GUI or not), maintain his virtual OS as he would maintain his guest OS. If however he decides to share it, the image will be heavy and the process much longer than with containers Distant access is possible, but coworking is hard in this scenario	Initial skills are required and no GUI applications (unless they use web). Deploying and sharing our work is very easy. Containers can also be used to quickly download/execute/deploy apps. For example if we need a mongo db and don't want to be bothered with all the requirements etcc .. (we simply pull an image with all necessary things already packaged and run it as a container
Infrastructure admin POV	As an admin, I can really control what a virtual OS uses as a resource. Even if I over-allocate, as long as my guest OS doesn’t need more resources than the	For an admin maintaining a machine with multiple containers is a nightmare but still better compared to directly running all apps directly on the machine.

	one allocated there should be no problems. On the network, I can see the VM as a PC. Rules applicable to PC (firewall, users..) are as easily applicable to a VM Outils de monitoring disponible	Even if libs/bins are seperated, an impact on the hard disk of one container can affect another container. Network resource allocation can be tricky (no bridge mode for example). Outils de monitoring disponible

2. Similarities and differences between the existing CT types

Different containerization tools are today available. Here is a comparison of the three most known open source/ free tools
Chosen criterias are size of images, creation/ modification ease, resources available in community, security , isolation, management (tooling), performance

	LXC	Docker (uses Linux containers)	Rocket
image size and portability	- portability works well only on Linux machines	- best	
creation/modification use	- easy line commands	- easy line commands - app images often already available	- more of from scratch building strategy
applications	- multiple apps/container	- designed for one app /container	- multiple applications per container
community resource	- good community	- docker hub image sharing - best community since one of the most used	- good community
isolation and security	- runs on machine	- runs as a process, - daemon process manages containers and is necessary (if killed, all containers die) -	- images from repository are signed and checked - security is central to Rocket
management	- can create	- needs sudo	- advanced

	containers as a user - external management tools are quite developed (Kubernetees, Nomad, ..)	access - external manageme nt tools are quite developed (Kubernetee s, Nomad, ..) - best Devops capabilities according to community	management tooling - process management tooling (multiple process' in a container) - external management tools are quite developed (Kubernetees, Nomad, ..)
performance	- best performance		

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

There are two types of hypervisors on top of which run virtual machines. Below a comparison of the two types

	Type 1 hypervisor	Type 2 hypervisor
Definition	Run on Bare-metal, it runs directly on the physical hardware of the host machine, so it doesn't have to load an underlying OS (host OS) before that. Type 1 hypervisor is also called Native or Bare-metal hypervisor	Run on top of an OS, it's called a hosted hypervisor because it relies on the host machine's pre-existing OS to manage hardware resources (calls to CPU, memory, storage and network resources). It's capable of supporting a wide range of hardware.
Performance	Best performing and most efficient for enterprise computing since it have direct access to the underlying hardware	Less performing since it has an underlying OS, all the hypervisor's activities have to pass through the host OS. That's why type 2 hypervisor is also called

		Hosted hypervisor
Security	Highly secure since the common flaws and vulnerabilities of OSs are absent (because there is no underlying OS) Each VM is isolated from the other so each VM is guarded against malicious activities or threats	Less secure. Type 2 is often reserved for client or end-user systems.
Hardware support	Use hardware acceleration support. Type 1 relies on hardware acceleration and can’t function without the availability of this technologie.	Use hardware acceleration support if the feature is available. If the feature isn’t available, they can rely on software emulation.
Example	Citrix/Xen server, VMware ESXi, Microsoft Hyper-V...	Microsoft Virtual PC, Oracle Virtual Box, VMware Workstation, Oracle Solaris Zones, VMware Fusion, Oracle Server for x86...

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

VirtualBox is type 2 hypervisor
Openstack is a free open source cloud computing platform that can be used to create private and public clouds. Openstack is not a hypervisor, it’s a software solution that complements the virtual environment, converting into a cloud computing solution. As openstack runs on a variety of hypervisors (VMware, KVM,Xen, Hyper-V…) we can choose what hypervisor we want to work with

Practical Part

Objectives 4 and 5

Virtual Box

First part: Creating and configuring a VM

No issues

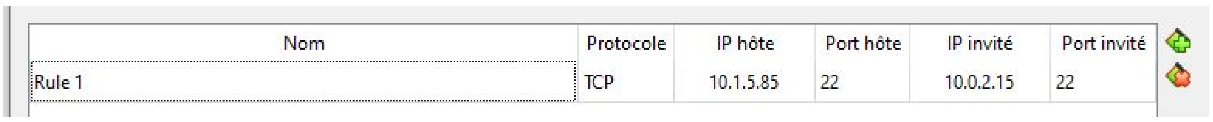
Attention difference virtual hard drive VS virtual image

Second part: Checking the VM's connectivity

IPv4 address of VM: 10.0.2.15/24
IPv4 address of host: 10.1.5.85/16

We can ping the host from our VM and we can also ping our colleague's VM.
From our host, we can not ping our VM but we can't ping our colleague's VM.
This is because our host PC is the one doing the NATing. All machines being NATed can have connectivity towards the machines visible by our host (including the host itself). However all connexions need to be started from inside the NATed network. This is the way NAT works.

Third part: Set up the "missing" connectivity



Nom	Protocole	IP hôte	Port hôte	IP invité	Port invité
Rule 1	TCP	10.1.5.85	22	10.0.2.15	22

The ssh now works fine from our host to our VM,
In cmd, we used this command to connect to our VM by ssh: ssh user@10.1.5.85 (ssh user@adresse_IP_of_host)

By mapping the VMs port to the host's port, anyone who wishes to access port 22 of our VM just needs to access port 22 of our host (by sending the packet to the host's IP), and the host will translate the ip address.

Fourth part: VM duplication

Attention copy pasting a virtual hard drive =/ cloning

Docker

Fifth part: Docker containers provisioning

All of the proposed interconnectivity works as with docker , the host machine creates a virtual network and virtual interface in that network. All containers will also have an interface in this new virtual network.

When launching the snapshot we still have nano as we have exported the container containing nano.

Objectives 6 and 7

Openstack

First part : CT creation and configuration on OpenStack

Done : Network setup and VM creation

Second part: Connectivity test

Connectivity test:
IP address of VM: 192.168.1.169 it's a private IP address of C class
IP address of host: 10.1.5.85 it's also a private IP address of A class
The connectivity doesn't work in both directions (From host to VM and vice-versa) because 192.169.1.169 is on a private network of OpenStack.

To make our VM reachable, we first set up a gateway router connected to both the public network and our private network.
This allowed connectivity from the VM to public (principle of NAT)
We then requested a floating address on the public network and linked it to our VM.

This made the connectivity possible from the public to the VM (through its floating address)

Third part: Snapshot, restore and resize a VM

We are actually in small2 size with 20GB root and 1GB RAM. We let our VM runs. when we resize our running VM2 from small 2 to medium, OpenStack tells us to confirm or revert resize/migrate. And our VM is forced to disconnect from the GUI. However (according to teacher) the services continue to be executed
It would be a problem if to resize we had to shut down the machine. To solve this problem we can clone the VM and use this new VM while we shut off and resize our original VM

The Snapshots size is 0 bytes. This is because it is not a backup but a saved state so that using this snapshot and the VM (which has evolved), we can revert to a previous version of the VM

Objectives 8 and 9

Part one : OpenStack client installation

Done, we had no issues with this task.

Part two: Web 2-tier application topology and specification

We tested the service on local Ubuntu VM and it works like we expected.

Part three: Deploy the Calculator application on OpenStack

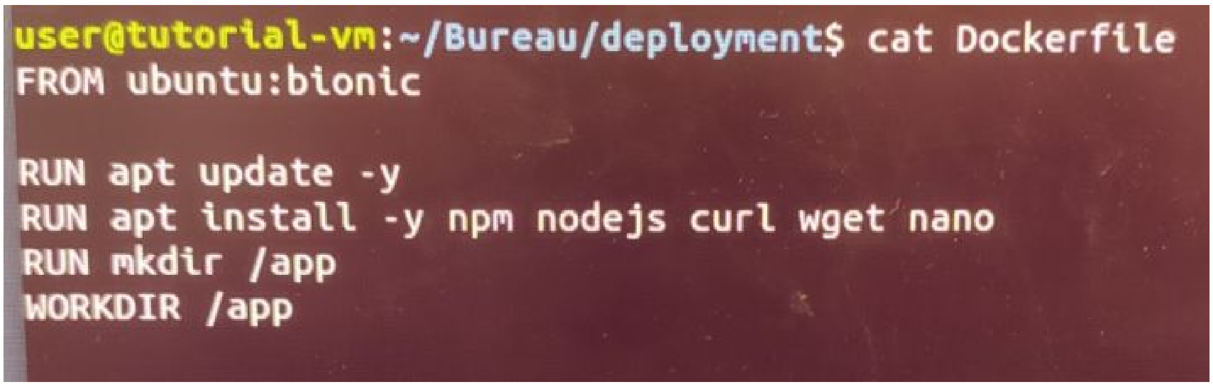
We put all our machines in the same network, and made sure the VM containing the Calculator Service had the address of all the other SubServices.

We did this part using only the Openstack platform (instead of Ubuntu OpenStack Client)

Part four: Automate/Orchestrate the application deployment

We then did the same thing with Docker and made the deployment process automatized.

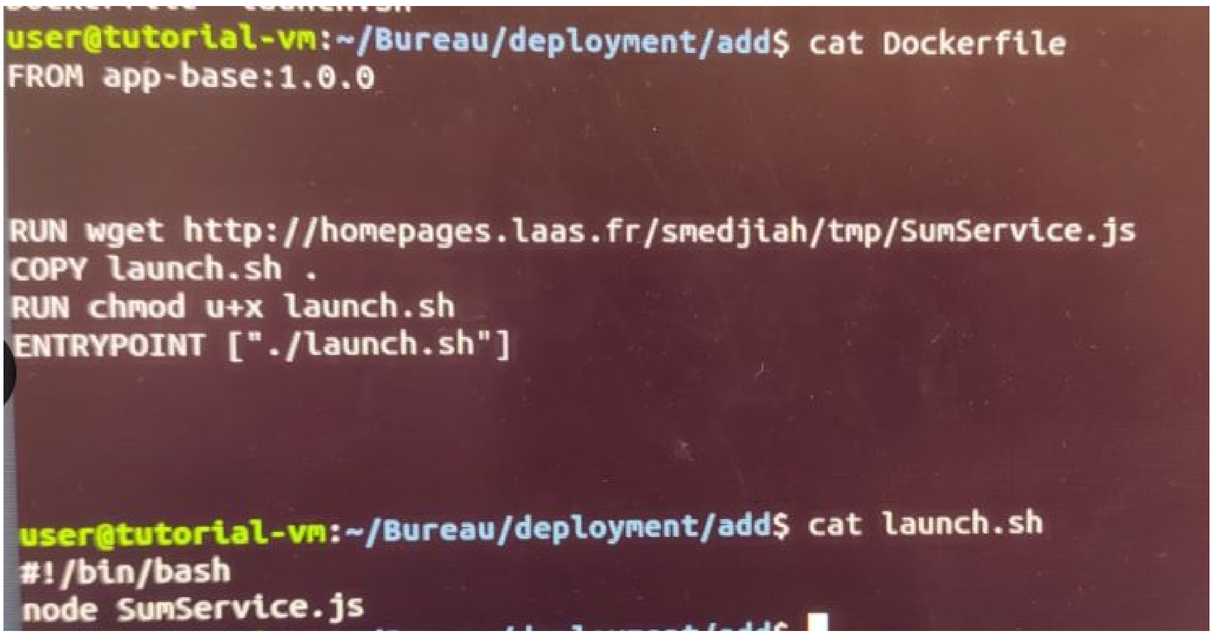
We start by building an app-base image starting from a ubuntu (latest version from Docker Hub) with the necessary applications (npm, nodejs, curl, wget and nano).



```
user@tutorial-vm:~/Bureau/deployment$ cat Dockerfile
FROM ubuntu:bionic

RUN apt update -y
RUN apt install -y npm nodejs curl wget nano
RUN mkdir /app
WORKDIR /app
```

In four different folders (one per subservice), we put a Dockerfile downloading the corresponding Javascript Service and setting up a launch.sh file as an entrypoint (executed when the image will be run into a container).

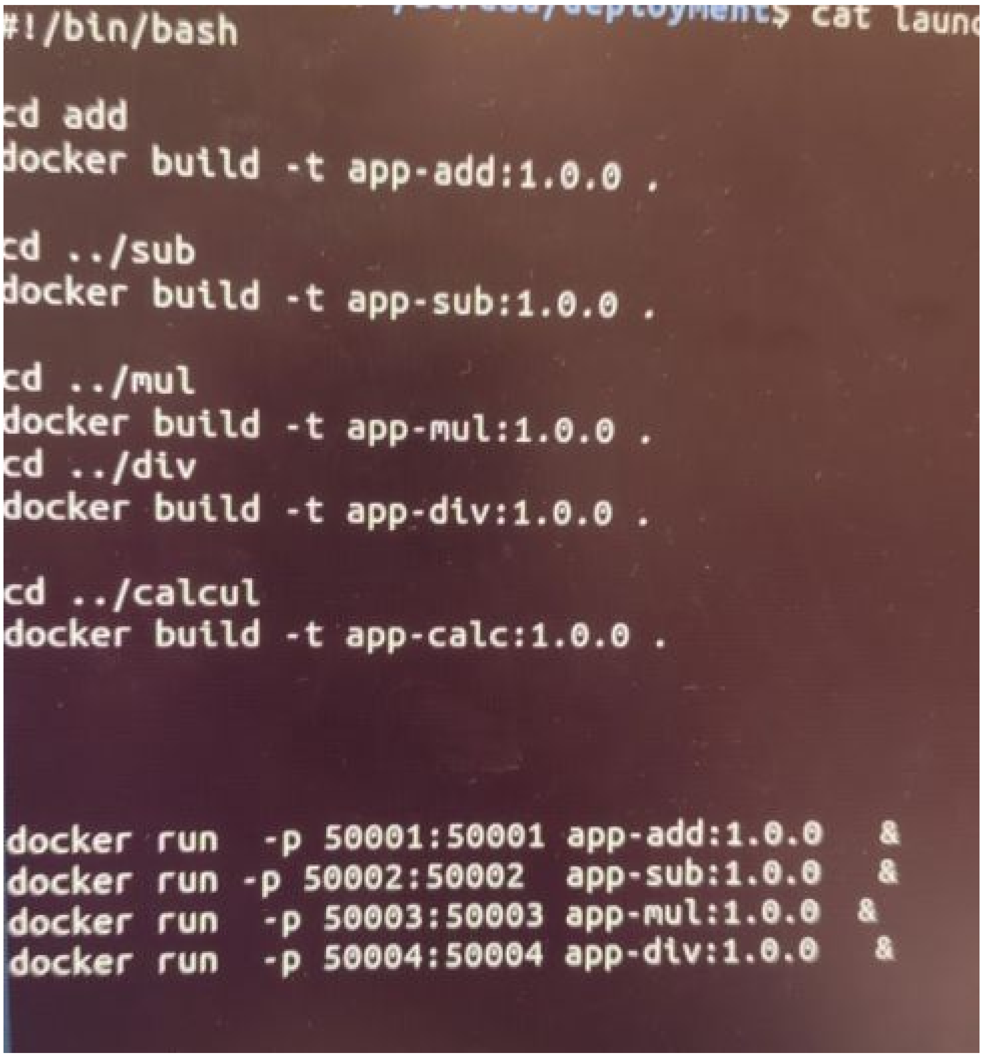
The launch.sh file just launches the nodejs service. Four example for the SumService :



```
user@tutorial-vm:~/Bureau/deployment/add$ cat Dockerfile
FROM app-base:1.0.0


RUN wget http://homepages.laas.fr/smedjiah/tmp/SumService.js
COPY launch.sh .
RUN chmod u+x launch.sh
ENTRYPOINT ["./launch.sh"]


user@tutorial-vm:~/Bureau/deployment/add$ cat launch.sh
#!/bin/bash
node SumService.js
```

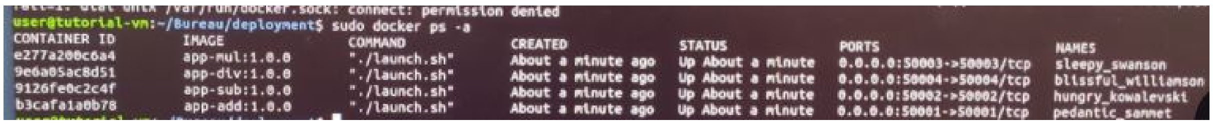
Finally we made a script to build all the images (four sub services and calculator service) and to launch the subservices.

To make sure the SubServices are accessible from the Calculator Service, we map their port (port the service launches on by default) to the host’s port.



```
#!/bin/bash

cd add
docker build -t app-add:1.0.0 .

cd ../sub
docker build -t app-sub:1.0.0 .

cd ../mul
docker build -t app-mul:1.0.0 .
cd ../div
docker build -t app-div:1.0.0 .

cd ../calcul
docker build -t app-calc:1.0.0 .



docker run  -p 50001:50001 app-add:1.0.0   &
docker run -p 50002:50002  app-sub:1.0.0   &
docker run  -p 50003:50003 app-mul:1.0.0  &
docker run  -p 50004:50004 app-div:1.0.0   &
```

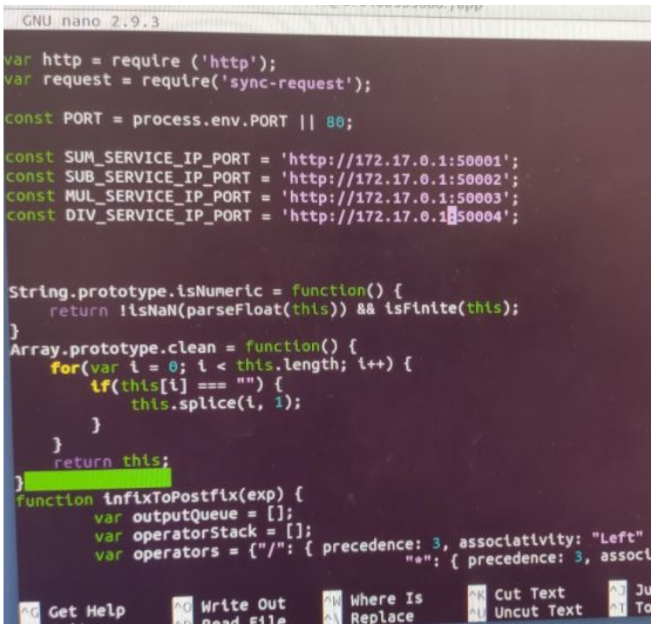
Result :



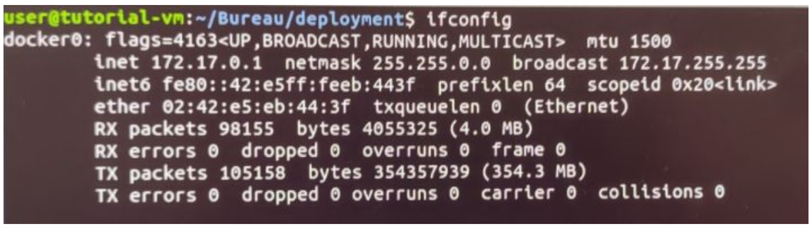
```
[illegible] /var/run/docker.sock: connect: permission denied
user@tutorial-vm:~/Bureau/deployment$ sudo docker ps -a
CONTAINER ID     IMAGE           COMMAND          CREATED              STATUS              PORTS                           NAMES
e277a200c6a4     app-mul:1.0.0   "./launch.sh"    About a minute ago   Up About a minute   0.0.0.0:50003->50003/tcp   sleepy_swanson
9e6a05ac8d51     app-div:1.0.0   "./launch.sh"    About a minute ago   Up About a minute   0.0.0.0:50004->50004/tcp   blissful_williamson
9126fe0c2c4f     app-sub:1.0.0   "./launch.sh"    About a minute ago   Up About a minute   0.0.0.0:50002->50002/tcp   hungry_kowalevski
b3cafa1a0b78     app-add:1.0.0   "./launch.sh"    About a minute ago   Up About a minute   0.0.0.0:50001->50001/tcp   pedantic_sammet
```

Before launching the calculator service, we modify the Javascript Service to make sure he knows where to find the SubServices (on the hosts address , on the mapped port).

We then launch the Calculator service and map it’s port as well to make sure it is accessible from our host PC.

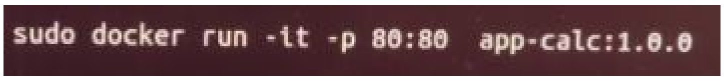


```
GNU nano 2.9.3

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

const PORT = process.env.PORT || 80;

const SUM_SERVICE_IP_PORT = 'http://172.17.0.1:50001';
const SUB_SERVICE_IP_PORT = 'http://172.17.0.1:50002';
const MUL_SERVICE_IP_PORT = 'http://172.17.0.1:50003';
const DIV_SERVICE_IP_PORT = 'http://172.17.0.1:50004';


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;
}
function infixToPostfix(exp) {
        var outputQueue = [];
        var operatorStack = [];
        var operators = {"/": { precedence: 3, associativity: "Left"
                        "*": { precedence: 3, associ

^G Get Help      ^O Write Out     ^W Where Is      ^K Cut Text      ^J Ju
                                  ^\ Replace       ^U Uncut Text    ^T To
```

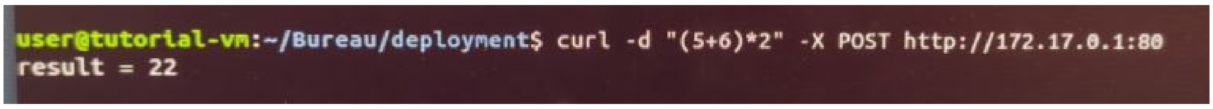


```
user@tutorial-vm:~/Bureau/deployment$ ifconfig
docker0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 172.17.0.1  netmask 255.255.0.0  broadcast 172.17.255.255
        inet6 fe80::42:e5ff:feeb:443f  prefixlen 64  scopeid 0x20<link>
        ether 02:42:e5:eb:44:3f  txqueuelen 0  (Ethernet)
        RX packets 98155  bytes 4055325 (4.0 MB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 105158  bytes 354357939 (354.3 MB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

Finally we launch and test the Calculator Service from our host PC :



```
sudo docker run -it -p 80:80  app-calc:1.0.0
```



```
user@tutorial-vm:~/Bureau/deployment$ curl -d "(5+6)*2" -X POST http://172.17.0.1:80
result = 22
```

Objectives 10 and 11

OpenStack

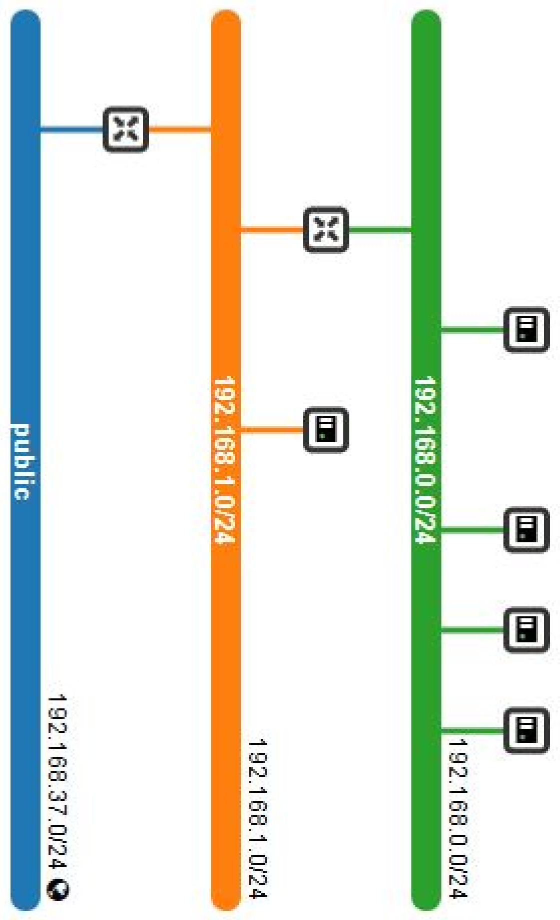
We place our SubServices in the 192.168.0.0/24 network,and our Main Calculator service on it's own network 192.168.1.0/24.
These two networks are inter-accessible through a router which is **directly** connected to both networks and through following rules put on the VMs:

- on the machines hosting the subservices, we setup the router between networks 192.168.0.0/24 and 192.168.1.0/24 as their default gateway

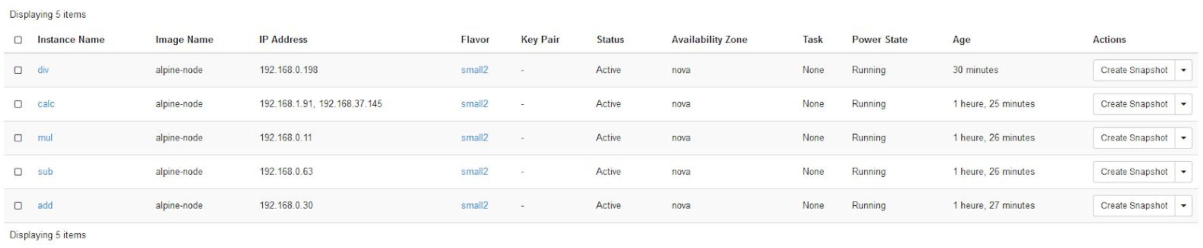
- on the VM containing the calculator services we set up the router (between networks 192.168.0.0/24 and 192.168.1.0/24) as the next hop to reach the 192.168.0.0/24 network, and the gateway router (connected to the public) as our default router.

To make the Main Calculator Service accessible from the outside, we requested and attached a floating address (accessible from the public) to our VM.
Our host PC (used for final test) having a firewall, we also change the port the service listen's to to port 80.
This made the Calculator service accessible from the outside, whilst still keeping the SubServices un-accessible from the outside.

Below the topology of our network:



The VMs and their IP address':

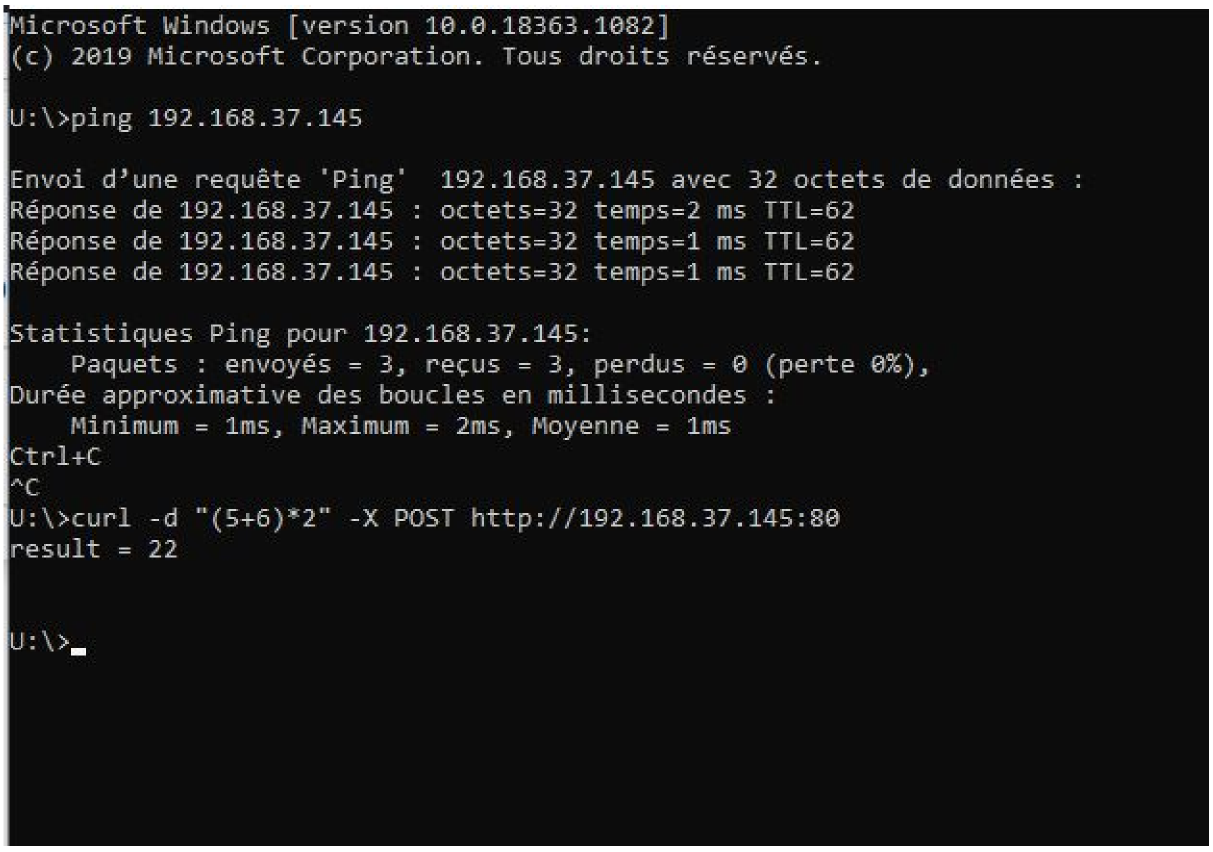


Displaying 5 items

Instance Name	Image Name	IP Address	Flavor	Key Pair	Status	Availability Zone	Task	Power State	Age	Actions
div	alpine-node	192.168.0.198	small2	-	Active	nova	None	Running	30 minutes	Create Snapshot
calc	alpine-node	192.168.1.91, 192.168.37.145	small2	-	Active	nova	None	Running	1 heure, 25 minutes	Create Snapshot
mul	alpine-node	192.168.0.11	small2	-	Active	nova	None	Running	1 heure, 26 minutes	Create Snapshot
sub	alpine-node	192.168.0.63	small2	-	Active	nova	None	Running	1 heure, 26 minutes	Create Snapshot
add	alpine-node	192.168.0.30	small2	-	Active	nova	None	Running	1 heure, 27 minutes	Create Snapshot

Displaying 5 items

Testing our Main Service from our host PC (TP room PC):



```
Microsoft Windows [version 10.0.18363.1082]
(c) 2019 Microsoft Corporation. Tous droits réservés.

U:\>ping 192.168.37.145

Envoi d'une requête 'Ping'  192.168.37.145 avec 32 octets de données :
Réponse de 192.168.37.145 : octets=32 temps=2 ms TTL=62
Réponse de 192.168.37.145 : octets=32 temps=1 ms TTL=62
Réponse de 192.168.37.145 : octets=32 temps=1 ms TTL=62

Statistiques Ping pour 192.168.37.145:
    Paquets : envoyés = 3, reçus = 3, perdus = 0 (perte 0%),
Durée approximative des boucles en millisecondes :
    Minimum = 1ms, Maximum = 2ms, Moyenne = 1ms
Ctrl+C
^C
U:\>curl -d "(5+6)*2" -X POST http://192.168.37.145:80
result = 22


U:\>_
```

Conclusion

We have through this TP learned how to set up virtual machines and containers from images.
We have also set up networks (networks, routing, Nating, bridging, port mapping), environments (linux environments for the services exercise).
These methods can be applied on other virtualizations and containerization platforms/applications.