Pose Lorine (pose@etud.insa-toulouse.fr)
Rup-Pedoussat Théo (rup--ped@etud.insa-toulouse.fr)
B1-4

Cloud Computing: Adaptability and Autonomic Management

Lab 1 : Introduction to Cloud Hypervisors

Access link : tiny.cc/TP_Cloud

Lab objectives

The purpose of this lab is to discover and test the basic functionalities (features) that cloud service providers could offer to end-users at the infrastructure level (IaaS perspective).
The lab objectives are detailed in what follows:

- Objective 1: Recognize the fundamental differences between the types of hypervisors' architectures (type 1/type 2).
- Objective 2: Recognize the fundamental differences between the two main types of *virtualisation hosts*, i.e. *virtual machines* (VM) and *containers* (CT).
- Objective 3: Evaluate the two different modes of network connection for VM and CT (i.e. NAT and Bridge modes).
- Objective 4: Operate VirtualBox VMs provisioning, in NAT mode, and make appropriate configurations so that VMs can access and be accessed through the Internet
- Objective 5: Operate Docker containers provisioning, and make appropriate configurations so that containers can access and be accessed through the Internet
- Objective 6: Operate proper VMs provisioning with OpenStack and manage the networking connectivity
- Objective 7: Test and evaluate the supported management operations on the virtualization hosts (e.g. snapshot, restore, backup, resize).
- Objective 8: Use the OpenStack API in order to automate the operations described in objectives 4, 5, 6 and 7.
- Objective 9: Implement a simple orchestration tool in order to provision a Web 2-tier application.
- Objective 10 : Make up a specific network topology on OpenStack
- Objective 11: Configure and deploy this network topology

Pose Lorine (pose@etud.insa-toulouse.fr)
Rup-Pedoussat Théo (rup--ped@etud.insa-toulouse.fr)
B1-4

Theoretical Part (objectives 1 to 3)

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

- Containers :

Containers are directly built on top of the host operating systems. Containers virtualize the software environment on a single host OS. Binaries and libraries are hosted and shared on the same OS so there is no need for them to be duplicated for different apps.

- Virtual Machines :

Virtual Machines are based on the same Server/Host system but the main difference with the CTs is that an hypervisor will allow to emulate a particular hardware system, not only the software environment. Within each virtual machine runs a unique guest operating system. They are all independent from each other. They don't share the binaries and libraries which leads to duplicated services.

The following tables summarize the main differences between the two types of containers, from the point of view of an application developer, and of a system administrator.

<u>Application Developer's point of view</u>

	Virtual Machines	Containers
Virtualization cost	- Higher cost in memory and CPU use	- Lower cost in memory and CPU use
Usage of CPU, memory and network	- Allocated cores - Memory requirements : Guest OS + duplication of binaries and libraries - Memory (Gb) and CPU allocation is not optimized.	- Shared host CPU - Memory used : only (Mb) - Only requires non duplicated binaries and libraries
Security for the application	- Better isolation between hardwares	Application level isolation: lack of security
Performances	- Depends on the part of the native OS allocated to the VM - Supervisor limitation - Slow to start	- Host OS performances - Just some seconds to start
Tooling for the continuous integration support	- Ability to run apps on different OSes - Single operating system needs care and feeding for bug fixes, patches…	- Apps run on the host OS

Pose Lorine (pose@etud.insa-toulouse.fr)
Rup-Pedoussat Théo (rup--ped@etud.insa-toulouse.fr)
B1-4

System Administrator's point of view

	Virtual Machines	Containers
Usage of CPU, memory and network	- Memory and CPU power limited by VMs allocation - Requires more RAM	- More instances on same hardware resources - Optimized used of the CPU and RAM between the CTs
Security for the application	- Isolated VMs from the host operating system and other VMs : strong security boundary	- Lightweight isolation from the host and other containers
Performances	- Slower : runs through the host OS, the hypervisor and the guest OS	- Faster : runs directly on the OS, like an application
Tooling for the continuous integration support	- Support Agile	- Download and install operating system updates on each VM. - Installing a new operating system version requires upgrading or often just creating an entirely new VM.

2. Similarities and differences between the existing CT types

	LXC	Docker	Rocket
Application isolation and resources	It allows you to not only isolate applications, but even the entire OS. The environment is very close to a VM but without the cost associated with the kernel splitting.	Containers: read only layers ("Docker images") Isolated by the kernel namespace (LXC concept) Performance penalty : using overlay or layers of read only file systems.	Rocket: inter-operable, and open container solution More secure (signature verification, separate privileges...) Entire process hierarchy encapsulated inside a KVM process (container's contents are firewalled off from the host)
Containerization level	Operating-system-level virtualization	Containers restricted to a single application Separate container storage from application	Applications containerization level

Tooling	Several language bindings for the API: python3, lua, Go, ruby, python2, Haskell Better-suited for I/O-intensive data applications LXC project provides base OS container templates and tools for container lifecycle management. Only runs on Linux	Docker Hub: public and private registry where users can push and pull images from Runs on Linux but also on windows and MacOs Docker Command Line Interface allows to list the images, gather and hande Docker images	Standard container format : App Container (appc) spec, but can also execute other container images, like those created with Docker.

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

The main difference between Type 1 and Type 2 hypervisors is that Type 1 runs on bare metal and Type 2 runs on top of an OS.

The Type 1 Hypervisors are considered the most efficient because they have direct access to the underlying hardware, so it doesn't rely on the OSes and device drivers. An other particularity is the high security provided by this Hypervisor since security flaws and vulnerabilities are usually related to OSes. Every guest machine is isolated and protected from malicious softwares and activities.

The Type 2 Hypervisor is, as previously said, installed on top of an existing OS. It relies on this same OS when it comes to manage calls to CPU, memory, storage and network resources. Opposed to Type 1 Hypervisor, any security flaws or vulnerabilities in the host OS could potentially compromise all of the VMs running above it. It is also slower since everything has to go through the OS first.

VirtualBox belongs to Type 2 Hypervisors when OpenStack can be developed with both types but are mainly developed with Type 1.

Pose Lorine (pose@etud.insa-toulouse.fr)
Rup-Pedoussat Théo (rup--ped@etud.insa-toulouse.fr)
B1-4

Practical Part (objectives 4 to 11)

1. Tasks related to objectives 4 and 5

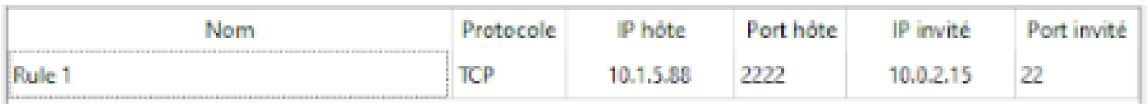
Creating a VirtualBox VM and setting up the network to enable two-way communication with the outside

In this first part, we created a Virtual Machine via VirtualBox in which we will create the containers. It works on a Linux environment and has been allocated one core of the CPU and 512 Mo of RAM. Finally, the NAT mode is selected for the network connexion.

Virtual Machine's IP address : 10.0.2.15
Host's IP address : 10.1.5.88

Due to the hardware virtualization, the VM has a different IP address than the host machine. Moreover, the network of our VM is based on NAT mode, so the VM is connected to a private IP network.
By using the ping command, it was proved that the VM could ping the host. This is due to the virtual router which runs on the host machine and connects the VM to the outside.
We tried to ping the VM from our host which failed. The VM and the host are isolated.
In the same way, the connexion does not exist between our hosted VM and the neighbours host. They are on different private networks.

In order to fix the lack of connectivity between the host and the VM it was necessary to deploy port forwarding as shown below. It redirects a communication request from the host address (10.1.5.88) and a chosen port number combination (2222) to our VM IP address (10.0.2.15). The invited port number is 22 since it is the port associated to SSH applications. This way the connexion is established, and the SSH application to the VM is enabled.



Nom	Protocole	IP hôte	Port hôte	IP invité	Port invité
Rule 1	TCP	10.1.5.88	2222	10.0.2.15	22

Figure 1 : Port forwarding rule used

Docker containers provisioning

In this part, we deployed the Docker environment on the VM we just created in order to have the admin privileges. First, we got the ubuntu image from DockerHub and used it to create our container CT1. With the necessary testing tools, we could define that the Docker IP address is 172.17.0.2. By checking the connectivity, we highlighted that the container can ping an internet resource (8.8.8.8 : IP address delivered by the Google DNS server). We can also ping the VM from Docker, and Docker from the VM. These results show that there is a virtualized network layer between the VM and Docker.
Another way to proceed is to create your own image by making a snapshot of a container that has the libraries you want. As an example, we created a second container CT2 with an ubuntu image, installed a text editor on it (nano) and made a snapshot of it. Then we created a container CT3 with the image we just created. This third container had nano pre-installed just as expected.
Finally, a third option is to directly write a dockerfile which will define all the libraries and binaries added to the image. Once it is built, it can be used to create a new container CT4.

2. Expected work for objectives 6 and 7

First part : CT creation and configuration on OpenStack

In this part we used OpenStack. We created a private network and an instance connected to that network. Then we unblock the network traffic on the virtualized private network by adding security rules. We added more particularly the ping (ICMP) and the SSH connection for all ports.

Second part: Connectivity test

We notice that the IP address associated by the hypervisor with the VM is in between the address range of the new private network. This explains that when we tried to ping the VM from the host, or the host from the VM it did not work. By going to the network topology tab, we can see clearly that the private network is isolated and has no connection to the public network, hence the lack of connectivity between instances.
In order to fix this issue, we simply have to create a router that will create a bridge between the two networks. This router will have an IP address from the private network. This allows the VM to ping the host. To make this connectivity bi-directional we have to associate a Floating IP to the VM. This way, we can from the host create a SSH client to the VM.

Third part: Snapshot, restore and resize a VM

While the instance was running and then while being shut off, we resized the VM :

- (RUNNING) making it bigger, which worked easily (confirm VM restart)
- (RUNNING) making it smaller, which did not work
- (SHUT OFF) making it bigger, which worked easily
- (SHUT OFF) making it smaller, which did not work

This last try should have worked but we learned that CSN restrictions blocked the process. One of the technical limitations is that in order to resize, the VM has to "freeze" and restart since it will rebuild the instance. It can not be running during this process.

3. Expected work for objectives 8 and 9

OpenStack method

The objective here is to implement a simple orchestration tool in order to provision a Web 2-tier application. The idea is to create 5 different VMs on a private network named here “PrivateNetwork”.

Four of these machines contain microservices for basic operations :

- SumService VM for addition with the local IP address 10.0.2.50 on port 50001
- SubService VM for subtraction with the local IP address 10.0.2.152 on port 50002
- MulService VM for multiplication with the local IP address 10.0.2.130 on port 50003
- DivService VM for division with the local IP address 10.0.2.19 on port 50004

Pose Lorine (pose@etud.insa-toulouse.fr)
Rup-Pedoussat Théo (rup--ped@etud.insa-toulouse.fr)
B1-4

On each of those 4 VMs, we downloaded the javascript file associated to the corresponding microservice from the Laas website. After downloading the required runtime environment, we have runned each microservice with the commande *node <service>.js*.

The last VM is the CalculatorService on which we have downloaded and executed the CalculatorService.js file. Its local IP address is 10.0.2.230 and it uses the port 50000. This service uses the 4 microservices available on the 4 other VMs placed in the private network to make a mathematical calcul. On the file, the 4 VMs IP addresses and ports are registered to make the link.

To test this system locally, we have at first created a Client VM that has been used as a local client (belonging to the same network). This test was a success.

The second step was to make the access to the calculator possible from the public network. To do so, at first we created a router to make the link between our "PrivateNetwork" network and the public network. Then, we assigned a floating IP to the CalculatorService VM so that it could be accessible from the public network when a client will send a calcul request. Finally, we have configured our security groups with the protocol TCP for the port 50000 corresponding to the CalculatorService VM.

To do the test, we used cURL to execute the calculator service from the public network and successfully managed to solve calculus.

You can find below the representation of our networks and the disposition of the VMs on it.

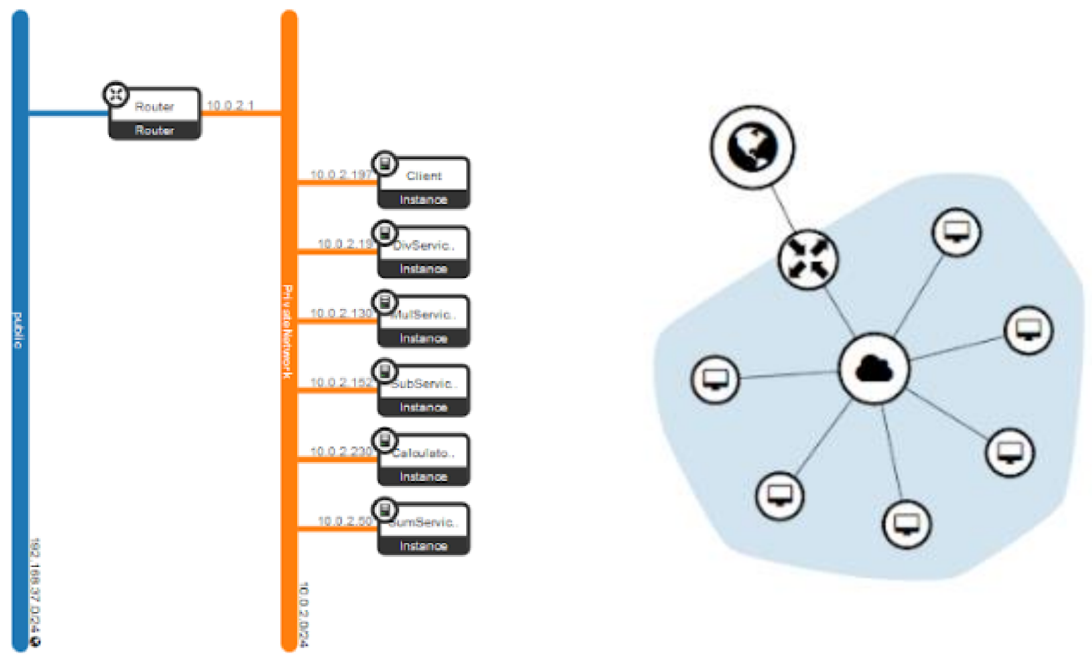
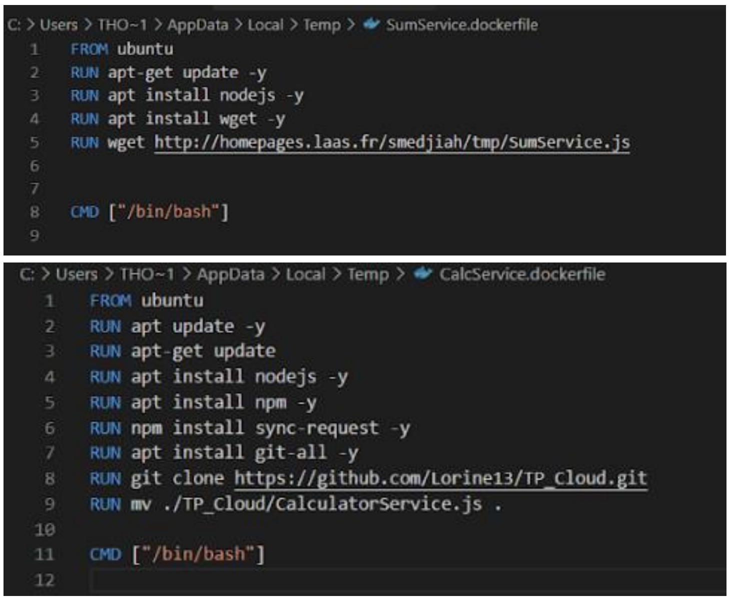


Figure 2 : OpenStack network topology for this task

Pose Lorine (pose@etud.insa-toulouse.fr)
Rup-Pedoussat Théo (rup--ped@etud.insa-toulouse.fr)
B1-4

Docker method

The objective here is to reproduce the same system seen in the previous part by using Docker in an automatic way.
First, we created a dockerfile for each service seen previously. Those dockerfiles will be built as images to implement the five dockers.
Therefore, each dockerfile includes the environment necessary to run the services on the docker and the javascript file containing the service. Below, you can see both SumService.dockerfile and CalcService.dockerfile that we have implemented.

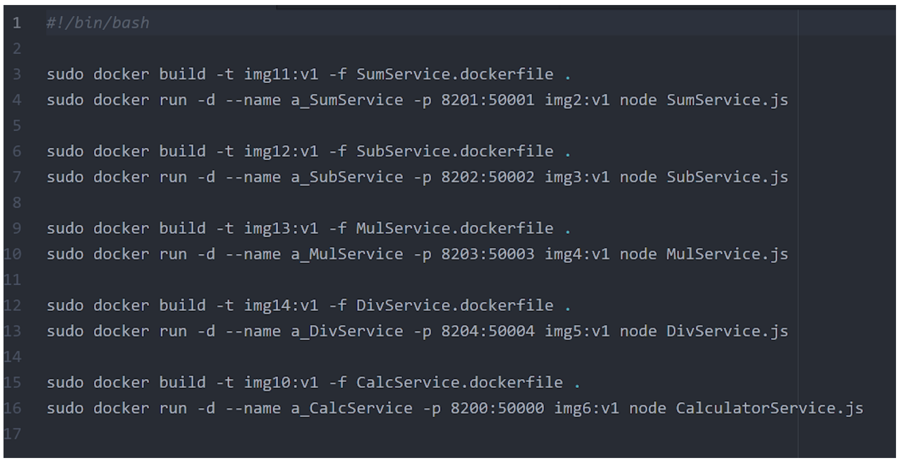


```
C: > Users > THO~1 > AppData > Local > Temp > SumService.dockerfile
FROM ubuntu
RUN apt-get update -y
RUN apt install nodejs -y
RUN apt install wget -y
RUN wget http://homepages.laas.fr/smedjiah/tmp/SumService.js


CMD ["/bin/bash"]
```

```
C: > Users > THO~1 > AppData > Local > Temp > CalcService.dockerfile
FROM ubuntu
RUN apt update -y
RUN apt-get update
RUN apt install nodejs -y
RUN apt install npm -y
RUN npm install sync-request -y
RUN apt install git-all -y
RUN git clone https://github.com/Lorine13/TP_Cloud.git
RUN mv ./TP_Cloud/CalculatorService.js .

CMD ["/bin/bash"]
```

Figure 3 : Dockerfiles used for one microservice and the calculator service

Secondly, we have implemented a bash file "script.sh" (see below) that will build the docker files, run the five dockers with the created image and run the services in each docker.
As the figure shows, when we run each container, we associate a port from the host machine to the port defined in the javascript files, so we can access the service from the host.



```
#!/bin/bash

sudo docker build -t img11:v1 -f SumService.dockerfile .
sudo docker run -d --name a_SumService -p 8201:50001 img2:v1 node SumService.js

sudo docker build -t img12:v1 -f SubService.dockerfile .
sudo docker run -d --name a_SubService -p 8202:50002 img3:v1 node SubService.js

sudo docker build -t img13:v1 -f MulService.dockerfile .
sudo docker run -d --name a_MulService -p 8203:50003 img4:v1 node MulService.js

sudo docker build -t img14:v1 -f DivService.dockerfile .
sudo docker run -d --name a_DivService -p 8204:50004 img5:v1 node DivService.js

sudo docker build -t img10:v1 -f CalcService.dockerfile .
sudo docker run -d --name a_CalcService -p 8200:50000 img6:v1 node CalculatorService.js
```

Figure 4 : Our script.sh bash file

Using this method, we easily managed to use the different microservices (Sum, Sub, Div and Mul) from the host with the cURL command.
However, we have had a problem when it came to the CalculatorService which couldn't access the microservices to make calculus.

Pose Lorine (pose@etud.insa-toulouse.fr)
Rup-Pedoussat Théo (rup--ped@etud.insa-toulouse.fr)
B1-4

4. Expected work for objectives 10 and 11

The objective was, at first, to follow the specific network topology given in the booklet by using once again the 5 VMs created in part 3.a. (SumService, SubService, MulService, DivService and CalculatorService). In this topology, the 5 VMs are splitted in 2 privates networks :

- "Sub-Network 1" (IP = 192.168.1.0/24) with the CalculatorService VM
- "Sub-Network 2" (IP = 192.168.2.0/24) with SumService, SubService, MulService and DivService VMs

Thus, this time, the 4 microservices required by the CalculatorService are not on the same network as him.
To make the link between these 2 private networks and with public network, 2 routers have been added :

- RT1 between public network and Sub-Network 1 (with IP = 192.168.1.1)
- RT4 between Sub-Network 1 (with IP = 192.168.1.254) and Sub-Network 2 (with IP = 192.168.2.1)

Note that we have changed some IP addresses compared to the booklet as the system was not working with it. You can see below the final topology :

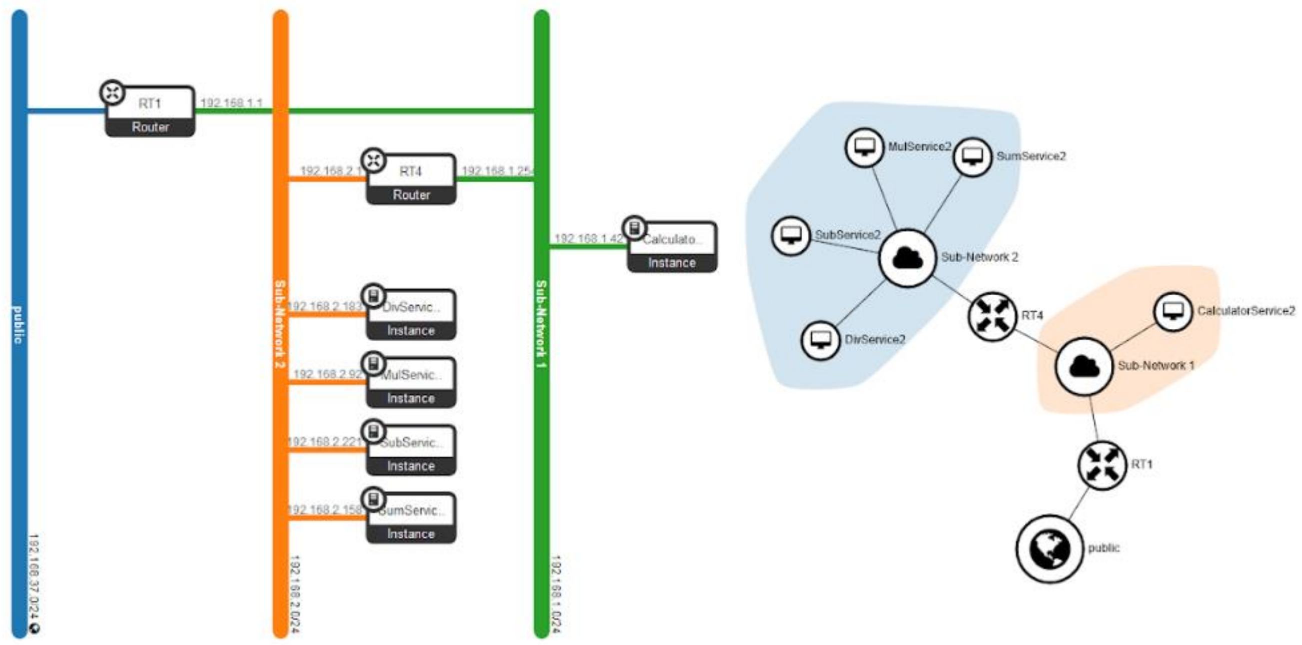


Figure 5 : Implementation of the target topology on OpenStack

From the CalculatorService VM, we have managed to ping the public network (ping 8.8.8.8). However, it is not possible to ping the 4 VMs located on Sub-Network 2. It seems that the CalculatorService sends the packets by default to RT1 so that they are not reaching RT4 and the Sub-Network 2. To fix this issue, we have defined a traffic route between the 2 sub-networks with the command line *route add -net 192.168.2.0/24 gw 192.168.1.254.*
After this, we managed to ping each VM belonging to Sub-Network 2 with the CalcultatorService VM.
Finally, to be accessible from the public network, we have assigned a floating IP to the CalculatorService VM. We have successfully managed to solve calculus in this new network topology by executing the calculator using cURL from the host machine !