



# ***Cloud Computing: Adaptability and Autonomic Management***

<b>Introduction</b>	<b>2</b>
<b>Tutorial 1</b>	<b>2</b>
Theoretical part	2
Difference between main virtualization hosts (VM and CT)	2
Differences between the existing CT types	4
Differences between Type 1 & Type 2 for hypervisors' architectures	5
Difference between the two main network connection modes for virtualization hosts	6
Practical part	7
VM creation (NAT mode) and network setting up	7
Creating and configuring a VM	7
Connectivity test	7
Set up the missing connectivity	8
Proxmox CT creation. Snapshot it, restore it and migrate it	9
CT creation and configuration	9
Connectivity test	10
CT migration	10
<b>Tutorial 2</b>	<b>12</b>
First part: Proxmox API	12
Second part: Monitoring application	13

# I. Introduction

For the first lab, we have to understand the concepts of Cloud Hypervisors (software allowing physical hardware to be shared across multiple virtual machines), Virtual machines and containers.

We also have to learn how to create a Container and how to snapshot, restore and migrate this Container.

For the second lab, we have to learn how to use the Proxmox API to automatically create containers, to monitor the resources of the server platform, to migrate containers between servers and to create an application for monitoring the platform resources.

## II. Tutorial 1

### A. Theoretical part

#### 1. Difference between main virtualization hosts (VM and CT)

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

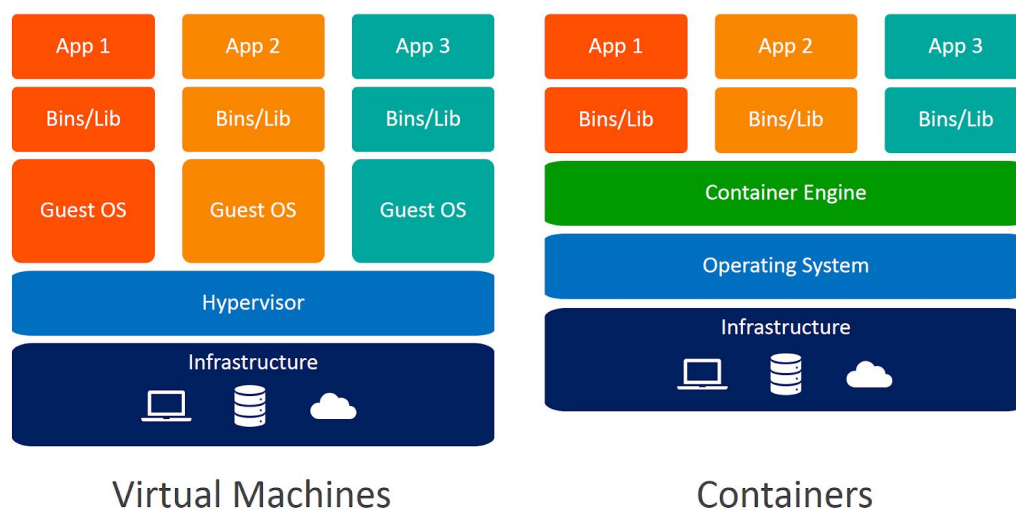


Figure 1: VM vs CT

The aim of these virtualization methods is to allow an easier way for developers to manage (recovery for example), build, deploy (and with continuous integration also test) faster various apps for different hardware systems. Indeed, with the recent necessity of increasing power and capacity for servers, bare metal (method of programming that is specific to the hardware) applications are not able to use the new abundance in resources.

That's why VMs are born, a running software on top of physical servers which emulate a specific hardware system. A hypervisor, or a virtual machine monitor, is software, firmware, or hardware that creates and runs VMs. It's what sits between the hardware and the virtual machine and is necessary to virtualize the server. Each VM runs a unique guest operating system (OS) and has its

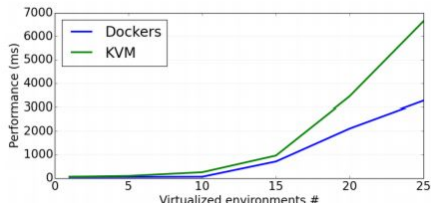
own binaries and libraries associated to the app. VM may be many gigabytes in size.

We can see in this approach that each VM includes a separate operating system image, which adds overhead in memory and storage footprint, and the app portability is also limited.

That's why containers have been created: they sit on top of a physical server and its host OS. Each container shares the OS kernel and the binaries/libraries. Thus, containers are very light (only megabytes compared to VMs), reduce management overhead and are more portable.

To sum up, containers provide a way to virtualize an OS so that multiple workloads can run on a single OS instance. With VMs, the hardware is being virtualized to run multiple OS instances.

Table 1: Comparison between VM and CT

Criteria	Virtual Machine	Container
<i>Virtualization cost (taking into consideration memory size and CPU)</i>	Hardware-level virtualization.	OS virtualization. Gain of efficiency for memory, CPU and storage compared to traditional virtualization.
<i>Usage of CPU, memory and network for a given application</i>	The hypervisor emulates the hardware from CPU, memory and storage. Those can be shared numerous time by each instance of VM. A VM, for a given app, uses more CPU and RAM than a container.	The host OS constrains the container's access to physical resources (CPU, memory and storage) → a single container can not consume all of a host's physical resources
<i>Security for the application (regarding access right and resources sharing)</i>	Base more secure (established security tools), and better known security controls).	Biggest problem of containers. Security between each CT by isolating them (avoid malware spreading). Need a lot of time to secure (but simplified security updates). Necessary to treat CT as a server application: big number of kernel systems out of the container, if a user or an app has privileges the host OS can be compromised.
<i>Performances (regarding response time)</i>	Latency experienced by users:  Bad latency when too much VM.	We can see that the latency of a container is much more low than a VM when the number of environments is high. Faster than VM to boot/start and shut down.
<i>Tooling for the continuous integration support</i>	Less tools.	Lot of tools.

## 2. Differences between the existing CT types

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

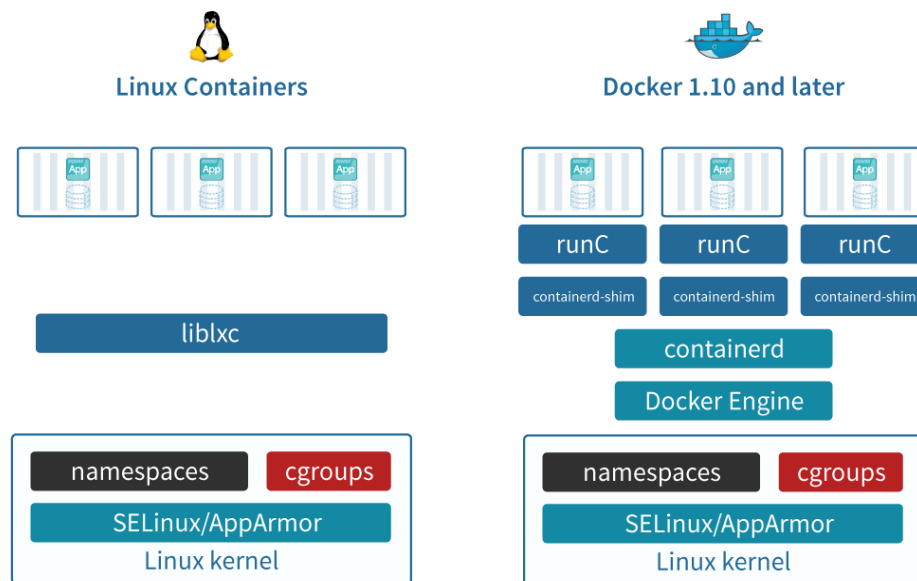


Figure 2 : Linux Lxc vs Docker

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)

Linux kernel has support for Container technologies like namespaces, cgroups etc.

Docker, LXC and Rocket use the technologies available in Linux kernel to manage the lifecycle of the Container.

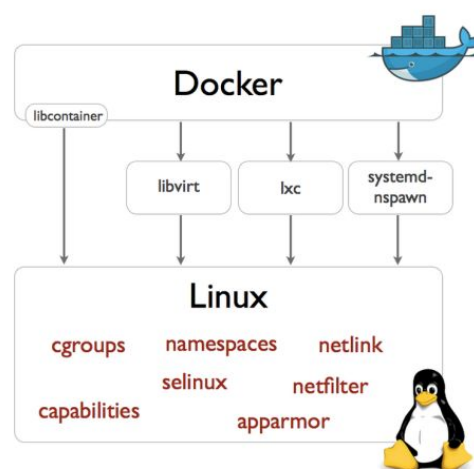


Figure 3 : Linux kernel used by CT technologies

A container management allows Container creation, deletion and modification.

Table 2: Comparison between CT technologies

Criteria	LXC	Docker
Containerization level	<b>OS container</b> = OS virtualization, the kernel of the OS allows the existence of multiple isolated containers instances. Allow to run multiple environment on a single host.	<b>Application container</b> = application virtualization, encapsulating computer programs from the OS on which they are executed. Allow to run a single service (single process). Before version 0.9, Docker was using LXC to interact with Linux Kernel. Since, it directly interacts with Linux kernel thanks to the libcontainer interface).
Application isolation and resources	Different applications can be installed and configured. Resources assigned to a container are visible to this container only. Total isolation from HW.	Docker is in top of LXC but increases the security by creating isolation layers between app and between the app and the host.
Tooling (API, continuous integration, service composition)	For IT operators. Useful to create several OS with identical configurations. Light virtualization and easy to use. Continuous integration (security, developers can collaborate)	For developer community. Supports continuous integration (developers can collaborate to build code, very easily).
System organization	No layered filesystems by default.	Layered filesystems.

**Rocket** = Application container (more secure, inter-operable and open solution compared to Docker)

**OpenVZ** = OS container (live-migration, userspace tools)

**runC** = expects a user to understand low-level details of the host OS

**containerd** = a standard runtime used to control runC

**systemd-nspawn** = linux container working with systemd (to manage services to handle components used for container isolation)

### 3. Differences between Type 1 & Type 2 for hypervisors' architectures

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

**Type 1: Bare-metal (a.k.a Native or Hardware-level)**

This type is run directly on the host's hardware (not requiring admins to install a server OS first) to control the hardware and to manage guest operating systems. The direct access to HW gives better performance, scalability and stability, but the HW support (drivers) is more limited.

#### Type 2: Hosted (a.k.a OS-level)

It is run on an OS as a computer program (we need to install this OS). For this one, there is a better HW compatibility (because the OS is responsible of HW drivers), but to access this HW we need more resources (going through the OS) and it can steal resources from other app running on the OS.

**VirtualBox** = type 2, hosted

**Proxmox** = type 1, bare-metal

## 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. The two main modes are the following:

- **NAT mode**: 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*) in VirtualBox.
  - **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 modes exist in VirtualBox, such as Private Network, that you can try out.

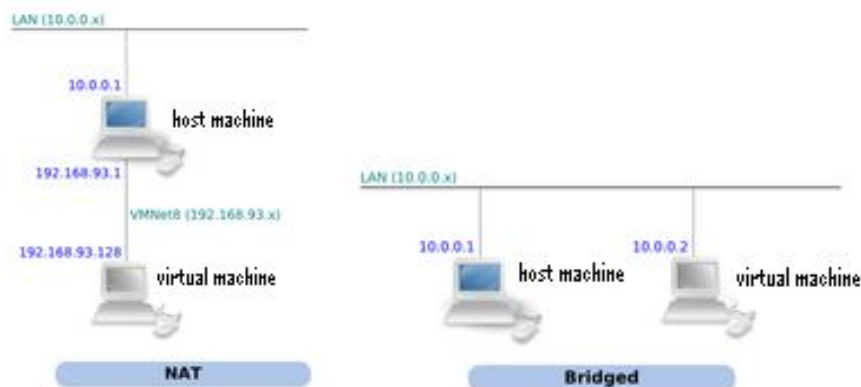


Figure 4 : Mode NAT vs Mode Bridge (the most usual)

## B. Practical part

### 1. VM creation (NAT mode) and network setting up

#### a) Creating and configuring a VM

In this part, we just have followed the creating and configuring tutorial for having and running a VM.

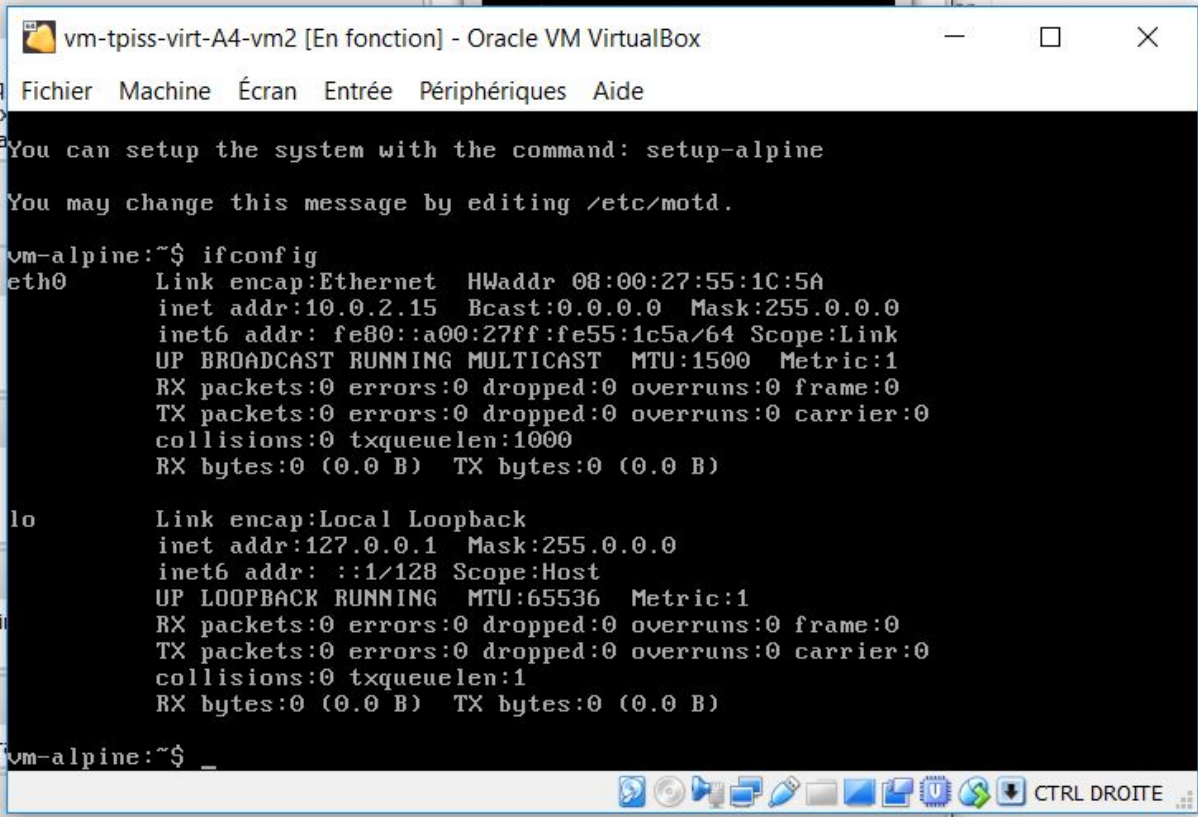
- Copy the virtual hard drive (vm-alpine.vmdk.zip).
- Unzip the archives.
- Start VirtualBox (available on Windows sessions).
- In VirtualBox, create a VM and configure it.
- Run the VM.

#### b) Connectivity test

We first used 'ifconfig' command line to identify the IP address attributed to the VM and the IP address attributed to the host (computer, machine):

**@Computer :** 192.168.56.1

**@VM:** 10.0.2.15



```
vm-tpiss-virt-A4-vm2 [En fonction] - Oracle VM VirtualBox
Fichier Machine Écran Entrée Périphériques Aide
You can setup the system with the command: setup-alpine
You may change this message by editing /etc/motd.
vm-alpine:~$ ifconfig
eth0      Link encap:Ethernet  HWaddr 08:00:27:55:1c:5a
          inet addr:10.0.2.15  Bcast:0.0.0.0  Mask:255.0.0.0
          inet6 addr: fe80::a00:27ff:fe55:1c5a/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)

vm-alpine:~$
```

Figure 5 : VM console with ifconfig

The two IP addresses belong to two different classes: A and C, and are private addresses. They are different because not on the same network.

With ping commands we have tested:

- Connectivity from the VM to the outside.
- VM → machine: ping is working (OK)
- Connectivity from a neighbour's host to our hosted VM.
- (neighbor machine → machine: ping is working (OK))  
 neighbor machine → VM: ping is not working (NO)
- Connectivity from our host to the hosted VM.
- machine → VM: ping is not working (NO)

Those tests allow to check if the NAT mode is setting up because it has to allow our VM to connect to the outside but to block anyone trying to connect to this VM, that is the case here.

### VM duplication

Then, we have created a duplicate (vm2), by going in the VBoxManage folder (of VM2) and running the command `'VBoxManage internalcommands sethduuid chemin\vers\vm-alpine-copy.vmdk'`

### c) Set up the missing connectivity

We have a NAT mode working but a neighbor machine can't talk to our machine. So we want to use the Bridge mode to connect the machines to a LAN.

For this, we will use the Port Forwarding technique for a dedicated application (SSH port 22).

We are redirecting requests done at the host computer (@192.168.56.1) to the virtual machine (@10.0.2.15):

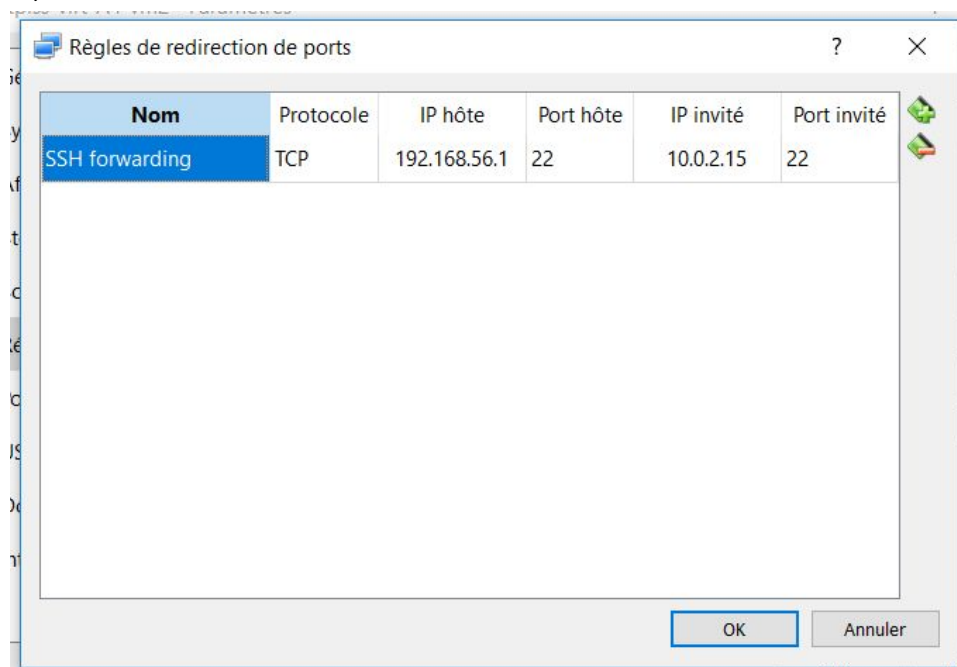


Figure 6 : Forwarding rule

We can connect in SSH to the VM using PuTTY:



```

taillard@LAPTOP-MAP8JT9Q:/mnt/c/Users/elief$ ssh user@192.168.56.1
user@192.168.56.1's password:
Welcome to Alpine!

The Alpine Wiki contains a large amount of how-to guides and general
information about administrating Alpine systems.
See <http://wiki.alpinelinux.org>.

You can setup the system with the command: setup-alpine

You may change this message by editing /etc/motd.

vm-alpine:~$ ifconfig
eth0      Link encap:Ethernet  HWaddr 08:00:27:55:1C:5A
          inet addr:10.0.2.15  Bcast:0.0.0.0  Mask:255.0.0.0
          inet6 addr: fe80::a00:27ff:fe55:1c5a/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)

vm-alpine:~$

```

Figure 7 : SSH connection to the VM

## 2. Proxmox CT creation. Snapshot it, restore it and migrate it

### a) CT creation and configuration

We used the tutorial to create and configure a container:

- Connect to the Proxmox Web interface: <https://srv-px1.insa-toulouse.fr:8006/>
- Authenticate with the INSA login/pwd.
- Create a CT (in the resource pool "admin") based on the available template **debian-8-turnkey-nodejs\_14.2-1\_amd64.tar.gz**, and configure it:
  - Set up a password
  - Ressources : Disk space (< 3 Go), RAM (< 512 Mo), CPU (1 core)
  - Bridge mode network (vmbri1 and VLAN ID = 2028) with dynamic IP addresses (dhcp)
  - Go to the CT view (via the Pool View or the Server view)
  - Start the CT, and activate the Console
  - Follow the "TurnKey" initial configuration steps.

## b) Connectivity test

We can see the Proxmox server IP address:



Figure 7 : Proxmox server page

**@Proxmox:** 10.20.27.7

We use ifconfig to identify the IP address attributed to the container:

**@CT:** 10.20.28.195

The two addresses are on the same LAN, they have the same network address 10.20.xx.xx

However, they are in different subnetworks: 10.20.28.xx and 10.20.27.xx

Using a ping command we can check different connectivities :

- From the CT to a Proxmox server:

CT → Proxmox : ping is working (OK)

- From our own machine to the CT:

Machine → CT : ping is working (OK)

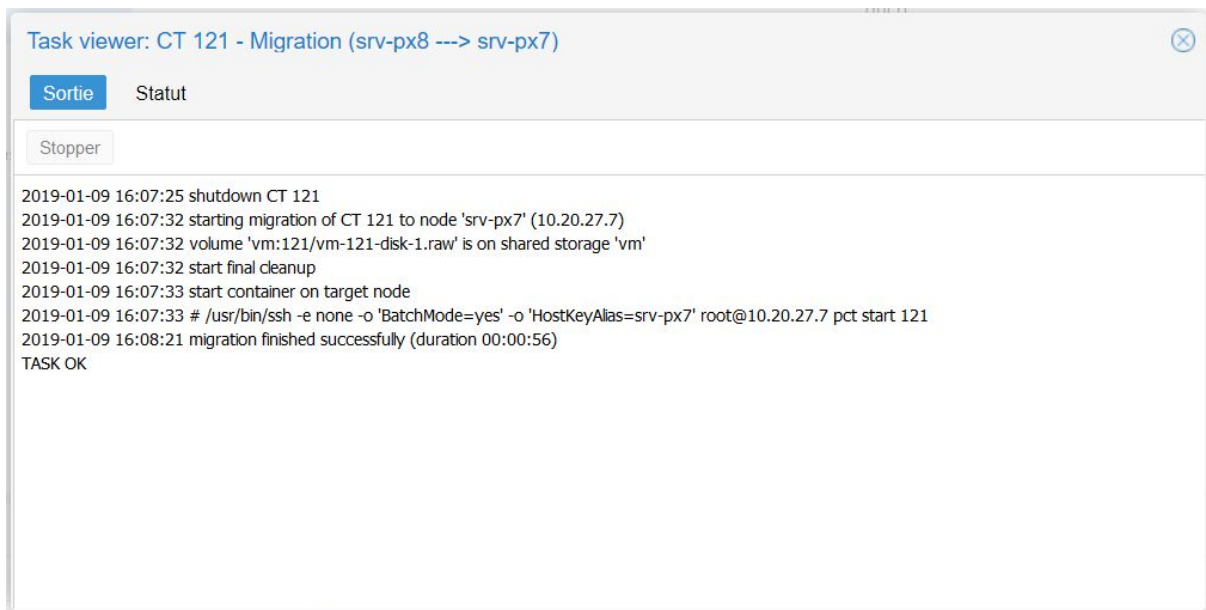
However, we also have tested other ping just to see what will happen:

- CT → machine : not working (NO)
- VM → CT : not working (NO)

## c) CT migration

In this last part, we are migrating the CT from a server to another (CT home page, button migrate).

The server we have used is the second one that we have (server 7).



*Figure 8 : Container migration page*

We have finally observed that the IP address of the migrated container is still the same: only the link of the CT changed.

## III. Tutorial 2

### A. First part: Proxmox API

In this part, we have used the Proxmox API. For this, we have downloaded the Java library (Eclipse project) proxmox app base. This API allow to do automatically the tasks seen in Tutorial 1 and the Web interface is one of the prospective clients of this API.

We used a HTTP REST client to interact with this API (using a provided native Java API).

We have observed different functions, for containers in LXC.java and for servers in Node.java :

- LXC : getCpu(), getStatus(), getName(), getMem(), getMaxmen(),...
- Node: getMemory\_used(), getCpu(),...

We have been able to collect informations about a Proxmox server thanks to these functions (you can find our code in the Github repository (<https://github.com/elietailardat/proxmox-app-base>) at '*proxmox-app-base/src/org/ctlv/proxmox/tester/Main.java*'.

- CPU usage (%) : 0.81% (very low)
- disk usage (%) : 7.47% (dividing rootfs used by total)
- memory usage (%) : 4.68% (dividing memory used by memory total)

Then, we have created and start containers with CT\_ID between 1400 and 1499.

Here is our console output (where you can find both server and container informations):

Console output log:

-----  
**Proxmox server informations:**

```
srv-px7 cpu usage: 0.7793349%
srv-px8 cpu usage: 0.59995997%
srv-px7 disk usage: 7.4835253%
srv-px7 memory usage: 5.0064387%
```

Create and start containers...

**Container informations:**

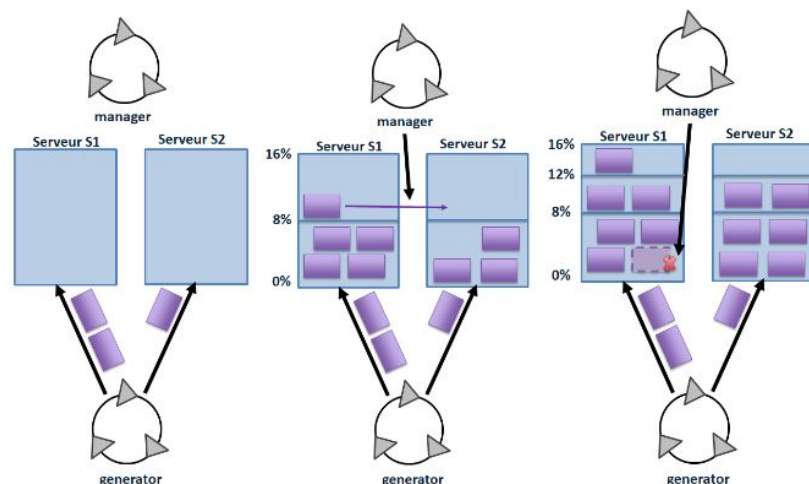
```
ct-tpiss-virt-A4-ct1 cpu usage: 0.0%
ct-tpiss-virt-A4-ct1 disk usage: 29.131187%
ct-tpiss-virt-A4-ct1 memory usage: 14.868164%
Host server: srv-px7
-----
```

## B. Second part: Monitoring application

In this part, we had to automate the processes of containers management in our servers thanks to the API of Proxmox and Java functions.

We chose to built our main process in the *GeneratorMain.java* file, in which we have coded 2 main functionalities:

- **The automatic generation of containers** (creation and start, but the two operations had sometimes difficulties to operate one after the other, so for testing purpose later on we started some containers via the Proxmox interface): in charge of the creation of CT, randomly on the two servers (66% on server 1, 33% on server 2) attributed to our team, for servers 7 & 8. The creation of a new CT may be periodical, or driven by a statistic distribution (uniform, exponential, ...).
- **The cluster management** (following the diagram below): As soon as the load of one of our CT exceeds **8%** of the host memory (i.e. half the resources accessible on the server), we perform load balancing between our servers. As soon as the load of one of the servers exceeds **12%** of the total load, we stop the oldest CT to support the creation of new ones.



Here are the different choices we've made for the different algorithm:

- First, we created an *Analyze.java* class to check the memory used by all the containers inside both our servers, and then, if enough memory is free (16% maximum) we generate **randomly** a container on either server 7 or 8 (66% and 33% of probability). That's why we chose that this analyze function returns a boolean variable.
- To manage the servers, we next compared the memory used inside a server with the two given thresholds (8% and 12% of the memory allocation) to perform either **migrating** or **dropping** operation. These functions are implemented inside the *Controller.java* class, and called inside our *GeneratorMain*.
  - The migration consists in first, stopping a random container (if it is running), then call the API migrate function, and finally restart it in its new server.

- To achieve the drop container method, we first created a function to return **only the running** containers on the concerned server, then find the **oldest** one (with the ID, starting at 1400 for us) and **stop** it.

You can find our Github repository link here:

<https://github.com/elietailardat/proxmox-app-base>