INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES TOULOUSE

INNOVATIVE SMART SYSTEMS

23 January 2018

Adaptability: cloud and autonomic management

Labs report

Enseignant : S. Yangui

Laurent Chasserat, Alex Noize

GitHub repository : https://github.com/TheRatHunter/tp-cloud

Summary

Lab 1 - Introduction to Cloud Hypervisors

Subject link : goo.gl/r5T3cX

Servers assignment : goo.gl/DDbdDt

Theoretical part

1. Differences between the main virtualisation hosts (VM and CT)

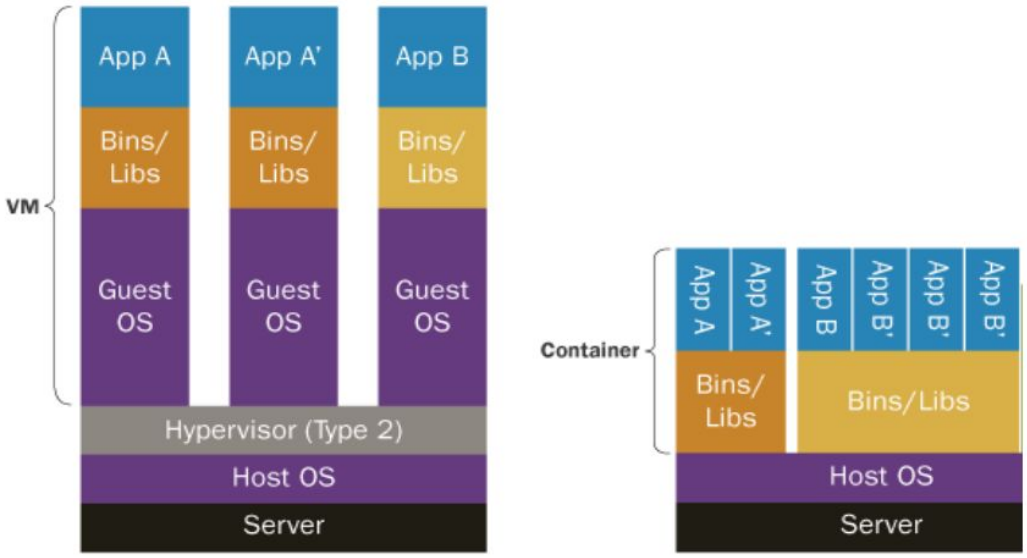


Fig. 1 - VM vs CT

A **Virtual Machine**, or "VM" imitates a computer in its entirety. Indeed, each VM contains a complete operating system, and with it all its binaries, libraries and device drivers. This architecture requires the VM to run on a Type 2 hypervisor, running itself on the host operating system. It is this host OS that runs the actual hardware of the machine. As shown in Figure 1, the hypervisor is able to simulate multiple VMs at a time, and thus several guest OSs, with each their binaries, libraries and applications. However, if these are identical from one VM to another, they will be redundant because an instance of these files will be present for each VM.

A **Container**, runs on the host operating system of the machine, without the need to simulate one. It simply uses the binary files and libraries needed by the applications in addition to this host OS. If several containers on the machine use the same binaries and libraries, then they are able to share

them. Thus, only one instance of these files exists, which avoids the waste of memory present in the case of the VMs. The role of the hypervisor is then provided by a containerization engine, the most well-known of them being Docker. A container is smaller than a VM, being freed from the space that the OS takes. It is therefore faster to migrate or save it. The isolation between containers is however less important than between VMs.

From an **application developer's** point of view:

- Virtualization in VMs offers more freedom in terms of RAM and CPU than that in CTs. Indeed, a container is calibrated to offer only the necessary resources.
- A VM theoretically offers more security than a CT because the VM is more isolated.
- In terms of performance and especially response time, the container is preferable because only one OS runs on the machine. In the case of a VM, moving from one OS to another to execute the code requires a lot more computation.
- The developer works on a VM in a similar way than in a standard environment. However, in the case of a CT he must write additional files defining how the container will be created. In addition, the container works with a "layer" architecture corresponding to the dependencies of the libraries used, which take time to set up before the application can be launched.

From the point of view of the **infrastructure administrator**:

- A VM is more expensive than a container because it uses more space on the disk, and requires more computing power for the same application, because of the presence of the guest OS. A VM also costs in network performance because the hypervisor needs to perform additional processing on the packets, for example through NAT, to route to and from the different VMs.
- As for the developer, using CTs requires more security precautions because access to libs and binaries is shared between different applications. In the case of a VM it is easier because they always has a local copy of these files.
- The performance of a CT is generally easier to administrate. Indeed, their smaller use of disk space allows faster backups, restorings and migrations.
- CTs usually work with an orchestration interface that allows an easy administration and support for continuous integration. In this lab, we will use the one coming with the Proxmox hypervisor. One could also mention Kubernetes who is the most widespread orchestrator. With tools such as these, it is easy to scale an application, that is to say to provide computing power according to the demand, by multiplying the number of containerized micro-services assigned to a task for example. VMs do not allow such an easy management of these modern problems.

2. Differences between the existing CT types

We compare CT technologies according to the following criteria:

- **Isolation of applications / resources**: Isolation is an essential criteria in terms of security. It means minimizing the number of resources shared between applications of different containers. Thus, it minimizes the number of external access to it and therefore the number of

potential security vulnerabilities. In the sense of "multi-tenancy application", ie. a single software application that shares its resources among several external clients, low isolation of resources can be a problem. Indeed, possible security vulnerabilities could allow a client to access the data of another. Moreover, without mentioning security, a strong use of the service by one customer could result in a loss of quality of service for the others.

- **Containerization level**: It characterizes the nature of what will be containerized. It can act from a wide container at the system level for several applications, or a multitude of smaller containers centered around each application.
- **Tooling**: This criteria characterizes the functions available to developers and administrators to manage their containers. It can be a simple command-line interface or an elaborated graphical interface. Various functions such as ease of backup, scaling or migration also fall within this criteria.

3. Differences between Type 1 and Type 2 hypervisors' architectures

A hypervisor is a program that allows different operating systems to run on the same machine. It serves as a base for the underlying OS.

A **type 1 hypervisor** is called "native". It is a software running directly on a real hardware. It is usually optimized for its hardware.

A **type 2 hypervisor** is said to be "hosted". It runs inside a "host" operating system, and hosts "guests" OSs.

In our case, VirtualBox is a type 2 hypervisor while Proxmox works more like a type 1 hypervisor.

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

In **NAT mode**, the virtualization containers are on a private IP network, and have a virtual router within the host PC that takes care of address translation. VMs / CTs are not visible from the outside.

In **bridge mode**, VMs / CTs are virtually connected to the same local network as the host PC, can access the Internet and can be found as any conventional machine.

Practical part

1. Tasks related to objectives 3 and 4

Creating and configuring a VM :

We created a VM on VirtualBox thanks to the Alpine image.

Testing connectivity :

Connectivity VM -> out : Works correctly
Connectivity PC of the neighbor -> VM : Doesn't work because a basic NAT only allows outgoing connections
Connectivity Host PC -> VM : Doesn't work because a basic NAT only allows outgoing connections

To get a working connectivity, the connection must be initiated by the VM, and the NAT takes care of the address translation to handle the responses to it. An outdoor machine can not find the VM because its IP address is masked by the NAT.

VM Duplication :

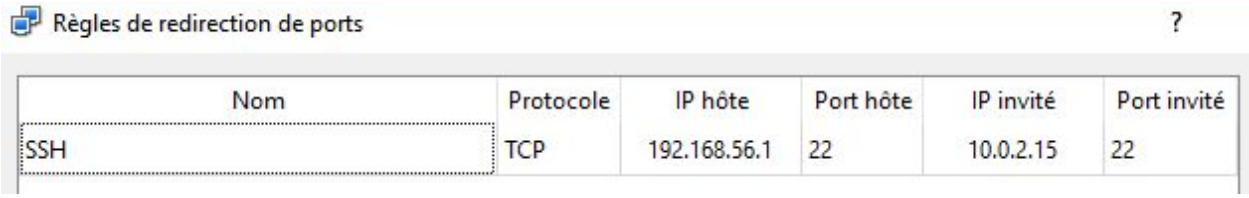
We then followed the instructions to duplicate our VM.

Set up the "missing" connectivity :

We have the following IPs :

Host Windows PC	10.1.5.45
Carte Réseau de VirtualBox (NAT)	192.168.56.1
VM	10.0.2.15

We had the following port forwarding rule:



With this setup, when launching a SSH connection from the host OS to the NAT interface, we are redirected to the VM and are able to connect to the Alpine Linux. We have set up the "missing connectivity".

2. Expected work for objectives 5, 6 and 7

Creation and configuration of a CT :

We have created a container on the Proxmox platform and we connected on it as root.

During the first session there was a DHCP problem so we couldn't get an IP.

On the following session we did the manipulation again and DHCP worked this time. We got the IP **10.20.28.110** !

Testing connectivity :

The connectivity works well. Thanks to bridge mode, it works just as if we were on a regular LAN (the same as the server hosting it).

We can :

- Ping the outside (for example google.fr) from the CT
- Ping the CT from the Lab machine.

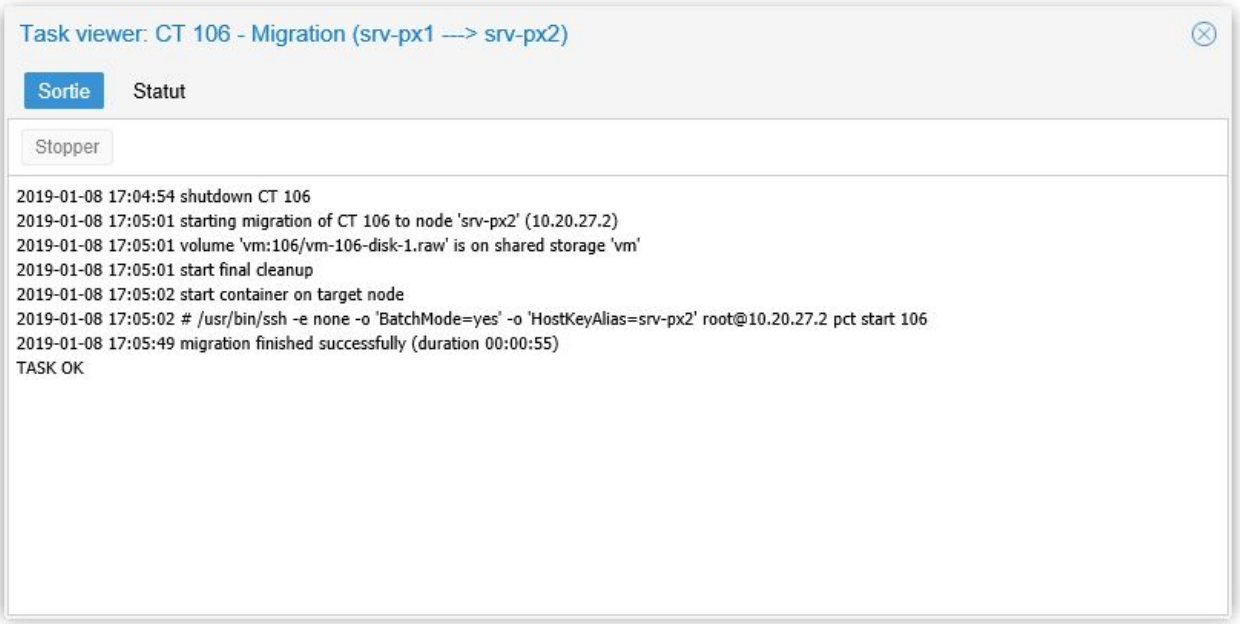
Snapshot and restoring a CT :

We couldn't try this manipulation because we didn't have the writing rights on the local disk.

Taking a snapshot of a CT enables us to restore it to this state later. For example, if we snapshot our CT then uninstall nano, and then restore it to the snapshot, anno will be back ! This can be useful when making mistakes !

CT migration :

We used the "migrate" tool to move our CT from the server srv-px1 to the server srv-px2.



Conclusion

In this lab we have acquired theoretical and practical knowledge about two types of virtualization hosts : Virtual Machines and Containers. We learnt how to set them up with specific softwares (VirtualBox and Proxmox respectively).

Lab 2 - Paravirtualization and Autonomic Management

Subject link : goo.gl/H7YEVD

Servers assignment : goo.gl/DDbdDt

The code linked to this project is available on GitHub :
https://github.com/TheRatHunter/tp-cloud

First part (objectives 1 to 4)

1. Collect information about a Proxmox server:

Information about the servers is retrieved in the Monitor class loop inside of the manager package. We loop through all the servers and containers, insulates the ones that are ours and put them in an HashMap. The Map has the following entries :

Key	Value
Server	List of all our containers in this server

We access the values in the hashmap in the Analyzer class (analyze method) called by the monitor after retrieving these informations. We then launch other calls, focused on the servers that we need, to know their characteristics. The additional information that we gather about the servers on which are our containers are :

- CPU Usage
- Disk usage (used, not max)
- Memory usage (used, not max)

2. Create and start containers.

We wrote the API calls allowing :

- Creation of a CT
- Starting of a CT
- Stopping of a CT
- Deletion of a CT

We tested this in the Main class of the tester package.

3. Collect information about a container

As explained above, we have stored information about all our containers in a HashMap. We use these values in the analyze method of the Analyze class (still in the manager package).

We compute :

- CT Status
- % CPU usage of the CT within the server
- % Used disk of the CT within the server
- % Used RAM of the CT within the server
- We know which server the CT is attached to because we memorized this information when constructing the HashMap (it is the Key of the CT list). No need to make one more API call !

Second part (objective 5)

1. CT Generator

a. What was implemented

We created a Java program that interacts with the Proxmox API in order to create CTs automatically on the two servers allocated to us. The CT creation is in a loop, and has 66% of chances of occuring on server 1 and 33% on server 2. It is made so that the CTs can't use more of 3% of the server's RAM. In the subject, it was asked to do it with 16%, but the limit took too long to be reached for an efficient testing.

Our CTs have the following characteristics :

- **ID** : XXYY, where XX is the base ID of our group (31), and YY the CT number
- **Name** : ct-tpgei-virt-C1-ctYY
- They don't use a lot of RAM because no application is running on them

You may find below pseudo-code of this generator :

```
Function CTGenerator:
    Get(ourCTs, ourServers)
    For all (ourServers):
        computeUsage(ourCTs, thisServer)
    If (usage(ourServers)<MAX_THRESHOLD):
        chosenServer = pick(ourServers, percentages)
        CreateCT(ct, chosenServer)
        While not (ct.created):
            Wait
        Start(ct, chosenServer)
        While not (ct.running):
            Wait
```

b. Possible improvements

At each CT creation, we implemented a blocking loop that awaits the mounting of the CT in order to start it. Indeed, the CT creation can take some time to the Proxmox platform, and if we send the starting command to quickly, it fails. It is only a prototype version of the program, we thought about using additional threads to start the containers. This way, the process would be non-blocking and the program would be able to create CTs faster.

2. Cluster manager

The cluster manager runs in a monitoring loop, which calls analyzing loop that takes the load balancing and offloading actions depending on the situation. The analysis loop is the following :

```
Function Analyze:
    Get(ourCTs, ourServers)
    For all (ourServers):
        computeUsage(ourCTs, thisServer)
        If (usage(thisServer)>MIGRATION_THRESHOLD):
            LoadBalancer
        If (usage(thisServer)>DROPPING_THRESHOLD):
            OffLoad
```

a. Load balancer

For this part, we first created a load balancer that migrates CTs from a server that has reached its capacity to another. Then, we implemented a program that stops the oldest CTs to save memory, if another threshold is reached.

```
Function LoadBalancer:
    Get(ourCTs, srcServer)
    ctToMigrate = ourCTs.first();
    Stop(ctToMigrate,srcServer)
    While not (ctToMigrate.stopped):
        Wait
    Migrate(ctToMigrate,srcServer,dstServer)
    Wait
    Start(ctToMigrate,dstServer)
    While not (ctToMigrate.running):
        Wait
```

Right now, the program loops over try/catch blocks, skipping exception until the operation was successful. We should then find a cleaner way to manage server errors.

To test our program efficiently, we wanted to migrate and stop our programs often. Since our containers did not used a lot of RAM, we set the migrating threshold at 0.1% of the server's RAM.

b. Oldest CTs stopping

For this part, we created a offloader that stops the oldest running CTs on overloaded servers.

You may find below the pseudo-code of this method :

```
Function OffLoad:
    Get(ourCTs, server)
    Sort(ourCTs, vmID)
    Stop(ourCTs.first)
    While not (ourCTs.stopped):
         Wait
```

To test our program efficiently, we wanted to migrate and stop our programs often. Since our containers did not used a lot of RAM, we set the stopping threshold at 0.2% of the server's RAM.

c. Cleanup function

To make testing easier, we also created a method allowing us to clean all of our CTs on the server :

```
Function CleanUp:
    Get(ourCTs, ourServers)
    Stop(ourCTs)
    While not (ourCTs.stopped):
         Wait
    Delete(ourCTs)
    While not (ourCTs.deleted):
         Wait
```

Conclusion

During this lab, we could manipulate the concepts of containers seen in the previous one thanks to a REST API. We have implemented the automatization of processes such as creation, migration and stopping of containers according to the available resources of each server.

Lab 3 - Provisioning End-User Applications in Cloud Platforms

Subject link : goo.gl/EPQfwA

Servers assignment : goo.gl/DDbdDt

The code linked to this project is available on GitHub :
https://github.com/TheRatHunter/tp-cloud

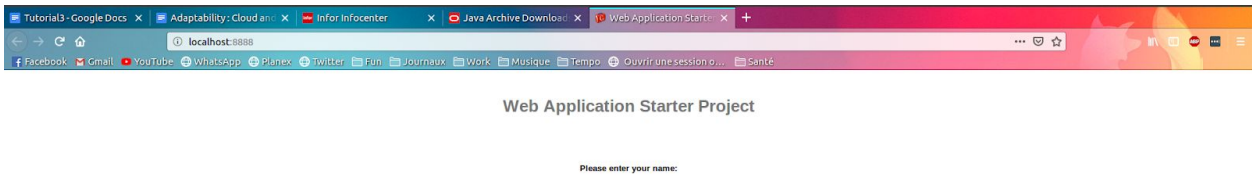
In this lab we have tried to provide a very simple on the Cloud. We proceeded in three phases:

- Development of a HelloWorld servlet application
- Deployment of this application on the Google Cloud Platform
- Management of our application

Providing a HelloWorld app on Google Cloud

Deployment of a HelloWorld servlet application

We have installed the Google Eclipse Plugin and created a Google Development Project sample. When running it in localhost, we manage to display the following page :



Sample project running in localhost

Deployment of the application on the Google Cloud Platform

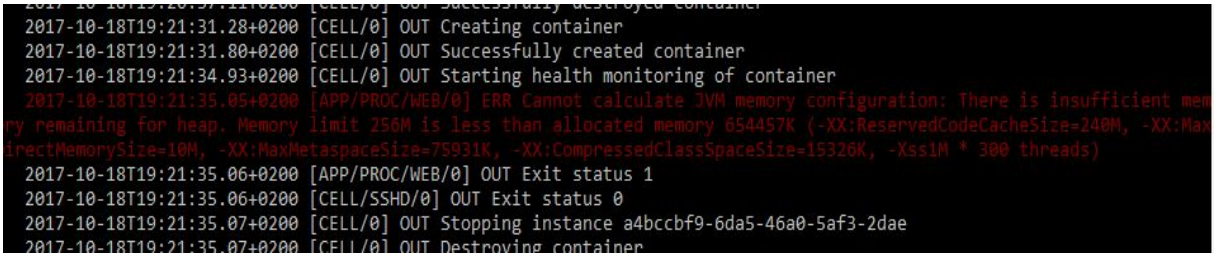
We then deployed the app on the Google Cloud platform.

Management of our application

After deploying an application on the Google Cloud, it is possible to manage several parameters (scaling, migration…) with the provided dashboard.

Providing a HelloWorld app on Cloud Foundry

We started by setting up the suitable environment and pushing the provided artifacts onto Cloud Foundry. We had to update the manifest.yml file with our personal data. We encountered the error :



To solve this, we augmented the memory limit of our container, and it worked.

We then tried scaling up our application by enabling more instances of it thanks to the web interface. The server can then compute as many requests in parallel as there are instances of our container.