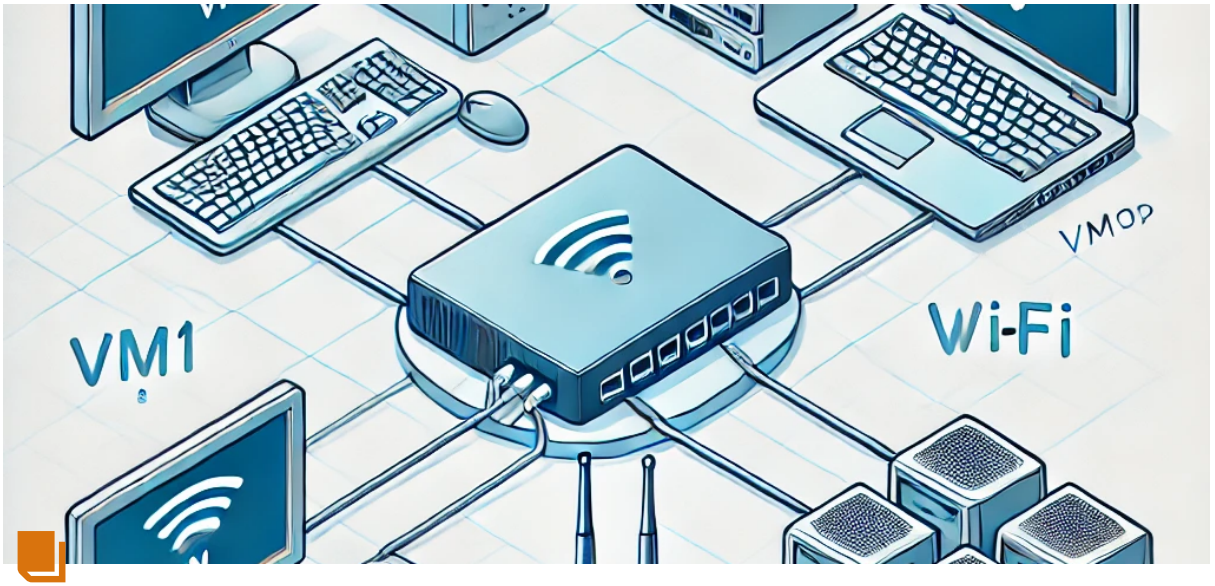
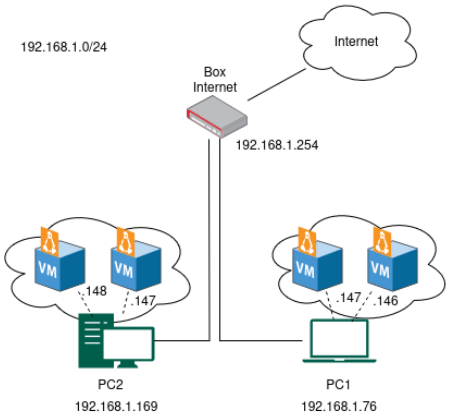


Rapport Projet Cloud

Alban PERSONNAZ - Adrien SOMARANDY

Introduction

Ce projet a pour but de mettre en place une infrastructure système à base de machines virtuelles avec l'hyperviseur KVM de manière automatisé via Terraform qui permet de faire de l'Infrastructure as Code (IaC) tout en ajoutant une partie de configuration automatisée pour le déploiement et la mise en service de Spark de facon distribuée par l'intermédiaire de Ansible qui est un outil de gestion et de configuration des machines.



Sommaire

Infrastructure via Terraform

Introduction

Terraform permet donc de déployer des infrastructures par l'intermédiaire de code au sein de fichiers de configuration (.tf). Pour ce projet nous utilisons l'hyperviseur KVM, donc pour gérer notre infrastructure avec Terraform nous avons besoin d'utiliser le provider *libvirt* qui est une bibliothèque (API) qui permet de communiquer avec notre hyperviseur KVM.

La structure du projet Terraform est découpée en un fichier main.tf et une partie modules/instances/ qui permet de gérer l'instanciation de plusieurs VMs facilement en changeant des paramètres au besoin, en fonction de ces différentes instances, par l'intermédiaire de variables initialisées dans main.tf ou définit par défaut dans le fichier /modules/instance/variable.tf.

```
├── main.tf
├── modules
│   └── instances
│       ├── cloud_init.cfg
│       ├── cloudinit.tf
│       ├── instance.tf
│       ├── variables.tf
│       └── volume.tf
```

Lors de l'instanciation d'une VM, plusieurs "Ressources" ont besoin d'être définies, certaines sont obligatoires, d'autres optionnelles, permettant ainsi de configurer chacune de nos VM comme l'on veut. Par exemple, certains points sont très importants dans le cadre de ce projet.

Provider

Le provider permet donc d'interagir avec KVM, ici on en a deux car il faut aussi administrer l'infrastructure sur le PC2 et pour se faire Terraform va utiliser l'API *libvirt* qui, à son tour, enverra les instructions via SSH à l'hyperviseur distant.

```
provider "libvirt" {
  alias = "local"
  uri = "qemu:///system"
}
provider "libvirt" {
  alias = "remote"
```

```
  uri   = "qemu+ssh://cross@192.168.1.169/system?no_verify=1"
  }
```

Volumes

On a un premier volume qui nous permet d'importer notre image depuis internet, on place ce volume dans la pool default qui est celle qui existe par defaut avec KVM, il n'est pas nécessaire, dans ce projet, de créer une nouvelle pool.

```
resource "libvirt_volume" "ubuntu-qcow2" {
  name = "ubuntu-${var.name}.qcow2"
  pool = "default"
  source = "https://cloud-images.ubuntu.com/noble/current/noble-server-cloudimg-amd64.img"
  format = "qcow2"}
```

Dans le code ci-dessous l'on donne un nouveau volume à notre VM d'une taille définie dans le fichier /modules/instances/variables.tf. Dans notre cas, nous avons fixé cette variable à 10 Go, à cause des différentes installations liées à Spark notamment.

```
resource "libvirt_volume" "disk" {
name           = "vm-disk-${var.name}"
base_volume_id  = libvirt_volume.ubuntu-qcow2.id
pool           = "default"
size           = var.disk_size
}
```

Un des problèmes rencontré fut de ne pas rajouter de volume aux VMs créant des erreurs systèmes lors de l'installation de Spark car la taille par défaut du volume de l'image était trop faible.

Module

Voici l'instanciation d'une VM qui elle sera hébergée par le PC2 (via *libvirt.remote*) qui est donc l'hôte physique distant et ne possédant pas Terraform. Plusieurs paramètres sont à noter, comme la mémoire des VMs qui sera pratique pour Spark ainsi que les 2 cœurs virtuels (en terme de performance). Mais on peut aussi voir que l'adresse MAC est notée en dur ce qui est pratique si l'on utilise un bail statique avec un serveur DHCP pour garder la même adresse IP et ce pour chacune des VMs. Le dernier élément important dans les différentes variable est le "*net-mode*" qui est "*br-test*". Il s'agit d'un bridge créé sur l'hote et déclaré dans *libvirt* pour avoir les VMs en pont et les rendre accessible au sein du réseau comme toute autre machine.

```
module "remote_instance1" {
source    = "./modules/instances"
providers = {
libvirt = libvirt.remote
}
name = "remote_vm1"
cpu  = 2
memory = 2048
mac= "52:54:00:AB:65:F5"
net_mode = "br-test"
}
```

Cloud Init

Un élément de configuration très pratique est le fichier *cloud_init.cfg* ainsi que la ressource *libvirt_cloudinit_disk*. Si l'on utilise une image spécifique de type cloud image on va pouvoir avoir un fichier cloud init (.cfg) qui nous permettra d'ajouter de la configuration au système de nos VMs tel que l'ajout d'un utilisateur avec ses logins et mot de passe mais aussi rajouter des clés SSH (très pratique pour la suite avec Ansible)ou encore installer des paquets.

```
data "template_file" "user_data" {
  template = file("${path.module}/cloud_init.cfg")
}

resource "libvirt_cloudinit_disk" "commoninit" {
  name      = "commoninit-${var.name}.iso"
```

```
  user_data = data.template_file.user_data.rendered
  }
```

Ici on a donc l'ajout de la ressource avec le fichier *cloud-init.cfg* pour configurer un utilisateur. Cela n'a pas été fait durant le projet car nous utilisons des bails statiques avec le DHCP mais on aurait pu avoir un fichier *.cfg* dédié à la configuration réseau de nos VMs via la variable *meta-data*.

Instance

le fichier /modules/instance/instance.tf est le fichier où l'on applique toute la configuration des fichiers précédents pour l'instanciation personnalisée de la VM. On retrouve donc le nom, les vcpus... Mais aussi la configuration réseau, les disques etc...

```
resource "libvirt_domain" "instance" {
  name   = var.name
  memory = var.memory
  vcpu   = var.cpu

  ...
cloudinit = libvirt_cloudinit_disk.commoninit.id

network_interface {
  network_name = var.net_mode
  mac = var.mac
}

disk {
  volume_id = libvirt_volume.disk.id
}
```

Configuration via Ansible

Introduction

Ansible permet donc de configurer des machines par l'intermédiaire de playbooks, donc des fichiers de configurations qui sont idempotents c'est-à-dire que pour la même entrée on a toujours la même sortie ce qui est très pratique quand on applique plusieurs fois le même playbook pour faire de la débug. Ansible utilise SSH pour communiquer avec les machines, c'est pour cela que donner sa clé publique via le fichier cloud init dans terraform est très utile. Grâce à cet outil on peut installer des paquets et configurer les machines comme l'on veut, soit par l'édition des fichiers de configurations ou l'exécution des commandes. Dans ce projet Ansible a permis d'installer et de configurer Spark sur les différentes VMs mais aussi d'installer et de configurer les PC physiques pour le lancement et l'administration de l'infrastructure comme par exemple automatiser l'installation et la configuration de Terraform ou de KVM.

Dans ce projet on retrouve un fichier inventaire *hosts.ini* qui permet de cibler les machines sur lesquels appliquer les playbooks, ensuite il y a un playbook pour le PC1, un pour le PC2, ainsi que deux playbook pour les VMs dont un qui configure les 4 VMs pour l'installation et le fonctionnement de Spark et un autre playbook dédié à la VM Master pour Spark. On a aussi plusieurs fichiers de configuration des répertoires */kvm-config* et *spark-config* qui sont utiles à la configuration des machines et qui seront téléversés via Ansible.

```
├── host_pb.yml
├── hosts.ini
├── kvm-config
│   └── qemu.conf
├── pc_distant_pb.yml
├── spark-config
│   ├── filesample.txt
│   ├── master-spark
│   │   ├── info.md
│   │   ├── slaves
│   │   └── spark-env.sh
│   ├── slaves-spark
```

```
│   │   ├── slave1
│   │   │   ├── slaves
│   │   │   └── spark-env.sh
│   │   ├── slave2
│   │   │   ├── slaves
│   │   │   └── spark-env.sh
│   │   └── slave3
│   │       ├── slaves
│   │       └── spark-env.sh
│   └── WordCount.java
├── spark_config_pb.yml
├── spark_launch_pb.yml
└── variables.yml
```

PC 1

On retrouve donc le fichier *host_pb.yml* qui permet d'installer et de configurer tout ce dont on a besoin par exemple les différents paquets pour Terraform et KVM.

```
---
- name: Install KVM on pc portable
hosts: localhost
become: true
tasks:
#### Installation KVM
        - name: Update packages
          apt:
              update_cache: yes

    - name: Install KVM and dependencies
      apt:
          name:
              - qemu-kvm
              - libvirt-daemon-system
              - libvirt-clients
              - bridge-utils
              - virt-manager
      state: present

    ...
```

On a ensuite la configuration de Terraform via des fichiers grâce à l'option copy, ou encore l'ajout d'utilisateurs dans les groupes grâce à des commandes shell via *command*.

Mais on retrouve aussi la configuration réseau qui a été automatisé dans ce playbook avec la création d'un bridge et sa configuration ainsi que sa déclaration dans *libvirt*.

```
    ...
        - name: Bridge creation checked
            command: nmcli connection show br-test
            register: bridge_check
            failed_when: bridge_check.rc != 0
            changed_when: false

            - name: Create bridge br-test
                command: nmcli connection add type bridge con-name br-test ifname br-test
                when: bridge_check.rc != 0
      ...
```

PC2

Tout comme le PC1, la configuration nécessaire pour PC2 est décrite dans le fichier *pc_distant_pb.yml*.

```
---
- name: Install KVM and dependancies on pc distant
  hosts: pc_distant
  become: true
  tasks:
#### Installation KVM
    - name: MAJ packages
      apt:
        update_cache: yes

    - name: Installation KVM et dependances
      apt:
        name:
          - qemu-kvm
          - libvirt-daemon-system
          - libvirt-clients
          - bridge-utils
          - virt-manager
        state: present

 ...
```

Tout comme le fichier de configuration du PC1, il comprend l'installation de KVM et la configuration du réseau avec la création et configuration d'un bridge br-test comme sur PC1, sauf que cette fois via un script shell pour ne pas perdre la connection au réseau entre les étapes avec ansible.builtin.command.

En revanche, le playbook n'installe pas Terraform. En effet comme le déploiement de l'infrastructure est lancé par le PC1, il n'est pas requis d'installer Terraform sur le PC2.

Ainsi les VMs de notre infrastructure peuvent fonctionner sur le PC1 et PC2 et communiquer entre elles grâce au mode bridge les mettant sur le même réseau.

VM Master et Slaves

Sur les VMs que ce soit master ou slaves la configuration de Spark sera la même, ainsi le playbook *spark_config_pb.yml* sera exécuté sur les 4 VMs. On y retrouve l'installation des différents packages dont on a besoin ainsi que la configuration des différents fichiers necessaires à l'execution de l'application Spark (voir la partie configuration Spark).

VM Master

La configuration de la VM Master pour Spark est identique à celle des slaves comme dit précédemment mais dans le playbook *spark_config_pb.yml* on retrouvera un ajout spécifique à la VM master, l'ajout des paires de clés SSH qui lui permettront d'exécuter les tâches sur les slaves au lancement de l'application Spark.

```
#### master -> copie cle ssh
     - name: Copy clé privé
             copy:
             src: /home/cross/.ssh/id_rsa
             dest: /home/cross/.ssh/id_rsa
             mode: '0600'
             when: ip_address == master_spark

    - name: Copy clé public
            copy:
            src: /home/cross/.ssh/id_rsa.pub
            dest: /home/cross/.ssh/id_rsa.pub
            mode: '0644'
            when: ip_address == master_spark
```

Le playbook spark_launch_pb.yml permet d'ajouter les fichiers nécessaire au lancement de l'application ainsi que toutes les opérations nécessaire à l'execution de Spark dans le cadre du Wordcount (voir lancement Spark)

Spark

Spark est une application permettant de faire du traitement big data. Dans le cadre de ce projet il est utilisé en mode Standalone et donc sans HDFS, qui est un système de fichiers distribués. Via le code Java et les librairies Spark, il est possible de répartir les traitements sur des fichiers en mode maître-esclave, ce qui permet de traiter beaucoup de données en parallèle sur différentes machines.

Pour fonctionner Spark a besoin d'avoir Hadoop d'installé, Spark et Java 8. Ensuite il faut configurer les fichiers Spark et compiler le code avant de l'exécuter, ces différentes étapes sont réalisées dans les playbooks *spark_config_pb.yml* et *spark_launch_pb.yml* .

Installation des packages

La première étape est d'installer Hadoop, Spark et Java 8 ce qui se fait depuis le site de Daniel Hagimont car il y a les bonnes versions compatibles. Ensuite il faut configurer le fichier ~/.bashrc pour ajouter les variables d'environnement et ajouté ces variables au PATH, ces opérations sont automatisées dans le playbook *spark_config_pb.yml*.

```
    ...
    - name: Ajout à bashrc
        lineinfile:
            path: /home/{{ ansible_user }}/.bashrc
            line: "export JAVA_HOME=/home/{{ ansible_user }}/jdk1.8.0_202"
            state: present

    - name: Ajout à bashrc
            lineinfile:
            path: /home/{{ ansible_user }}/.bashrc
            line: "export PATH=$JAVA_HOME/bin:$PATH"
            state: present
        ...
```

Configuration Spark

Pour la configuration de Spark c'est le playbook *spark_config_pb.yml* qui s'en charge, il permet d'éditer */spark-2.4.3-bin-hadoop2.7/conf/spark-env.sh* et ce pour chacune des machines, qu'elle soit maitre ou esclave. Chaque VM a un fichier *spark-env.sh* différent car l'adresse IP locale est inscrite dans ce fichier.

```
    - name: Copy fichier spark-env.sh
            copy:
            src: /home/cross/Bureau/cours_N7/s9/Projet_cloud/ansible-host-conf/spark-config/master-sp
ark/spark-env.sh
            dest: /home/cross/spark-2.4.3-bin-hadoop2.7/conf/spark-env.sh
            mode: '0755'
            when: ip_address == master_spark
```

```
export SPARK_MASTER_HOST=192.168.1.146
export JAVA_HOME=/home/cross/jdk1.8.0_202
export SPARK_LOCAL_IP=192.168.1.146
```

Dans ce même playbook on ajoute les fichiers nécessaire tel que *filesample.txt* ou */spark-2.4.3-bin-hadoop2.7/conf/slaves* à toutes les VMs.

```
### master ; slave1 ; slave2 ; slave3
- name: Copy filesample
        copy:
        src: "/home/cross/Bureau/cours_N7/s9/Projet_cloud/ansible-host-conf/spark-config/filesample.t
xt"
        dest: "/home/cross/spark-2.4.3-bin-hadoop2.7/bin/filesample.txt"
        owner: cross
        group: cross
        mode: '0644'
```

Le fichier */spark-2.4.3-bin-hadoop2.7/conf/slaves* est utile uniquement pour le master, il spécifie les IP pour les différents slaves.

```
cross@localhost
cross@192.168.1.147
cross@192.168.1.148
cross@192.168.1.149
```

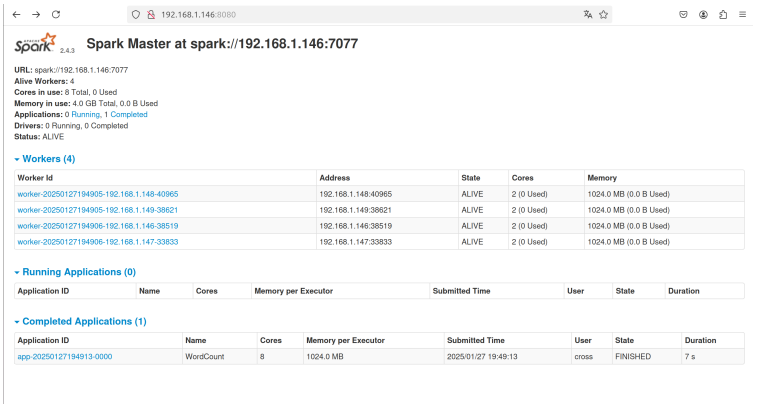
Lancement Spark

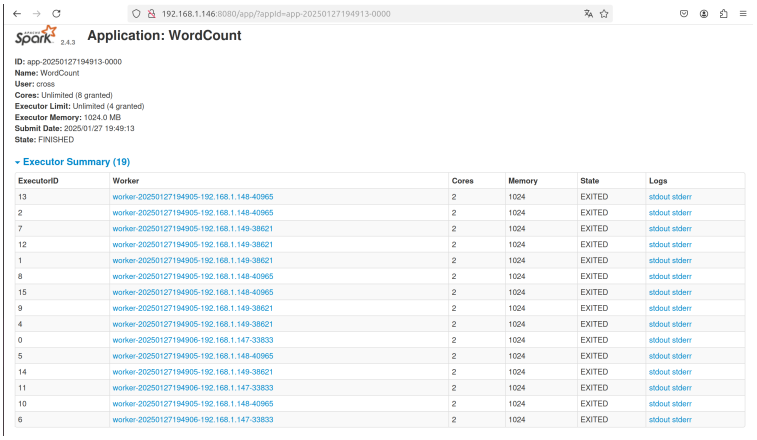
Le lancement de Spark se fait via le playbook *spark_launch_pb.yml*, il permet de créer le projet java, de compiler le code avec les bonnes librairies Spark et enfin de le spark-submit dans le cluster.

```
    ...
    - name: Run le jar dans le cluster spark
            shell: |
            ./spark-submit --class WordCount --master spark://{{ master_spark }}:7077 /home/cross/wor
dcount/wc.jar /home/cross/spark-2.4.3-bin-hadoop2.7/bin/filesample.txt
            args:
            chdir: "/home/cross/spark-2.4.3-bin-hadoop2.7/bin"
    ...
```

Résultat

Une fois tous les playbooks lancés, que ce soit à la main ou via un script init.sh notre application Spark finit par être déployée et exécutée jusqu'à la fin de sa tâche, dans notre cas la tâche Wordcount.





Configuration réseau

Plusieurs problématiques se sont posées durant ce projet particulièrement sur la partie réseau. Le problème étant que pour réaliser ce projet on utilise un pc portable et un pc physique (gaming). Ces deux PCs sont sous ubuntu en dual-boot car ce sont la tout ce qu'on avait pour réaliser ce projet.

Dans un premier temps, nous pensions laisser le PC2 au domicile d'Alban, et déployer l'infrastructure avec Terraform et la configurer avec Ansible depuis son portable à l'ENSEEIHT en utilisant des redirections de port via la box Free. Mais nous avons rencontré des problèmes au moment de la configuration de Spark, en effet la VM Master et les VMs slaves doivent pouvoir se joindre de manière bidirectionnelle.

Bridge

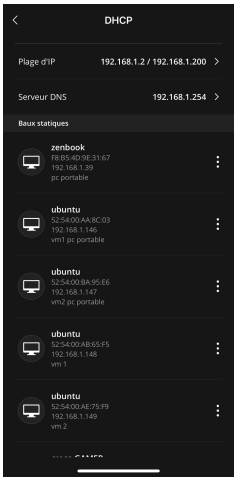
On est donc partie sur des VMs en mode pont, ainsi toutes les VMs se trouvent dans le même réseau et surtout tout le monde peut joindre tout le monde, ce qui est nécessaire pour Spark. La création du pont se fait via des commandes au sein des deux playbooks pour le PC1 et PC2 (host_pb.yml et pc_distant_pb.yml)

```
2: enx3c18a0d4cf8f: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel master br-test state UP
group default qlen 1000
   link/ether 3c:18:a0:d4:cf:8f brd ff:ff:ff:ff:ff:ff

7: br-test: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default qlen 1000
link/ether 36:8d:c4:af:2b:f9 brd ff:ff:ff:ff:ff:ff
inet 192.168.1.76/24 brd 192.168.1.255 scope global dynamic noprefixroute br-test
```

IP fixe via bail statique

Grâce au serveur DHCP, on peut associer des IPs aux adresses MAC et ainsi garder toujours la même IP et cela est pratique pour prendre le contrôle en SSH que ce soit pour Ansible, Terraform ou encore pour les communications Maitre - esclaves avec Spark.



Bails statiques du serveur DHCP

Redirection de ports

L’idée étant que les deux PC, portable et physique, soient dans le même réseau et lors de la présentation via des redirections de ports sur la box on puisse administrer les PCs en SSH, ainsi que les VMs.