Antonin CHAUVET Ramy CHEMAK Vincent PARENT

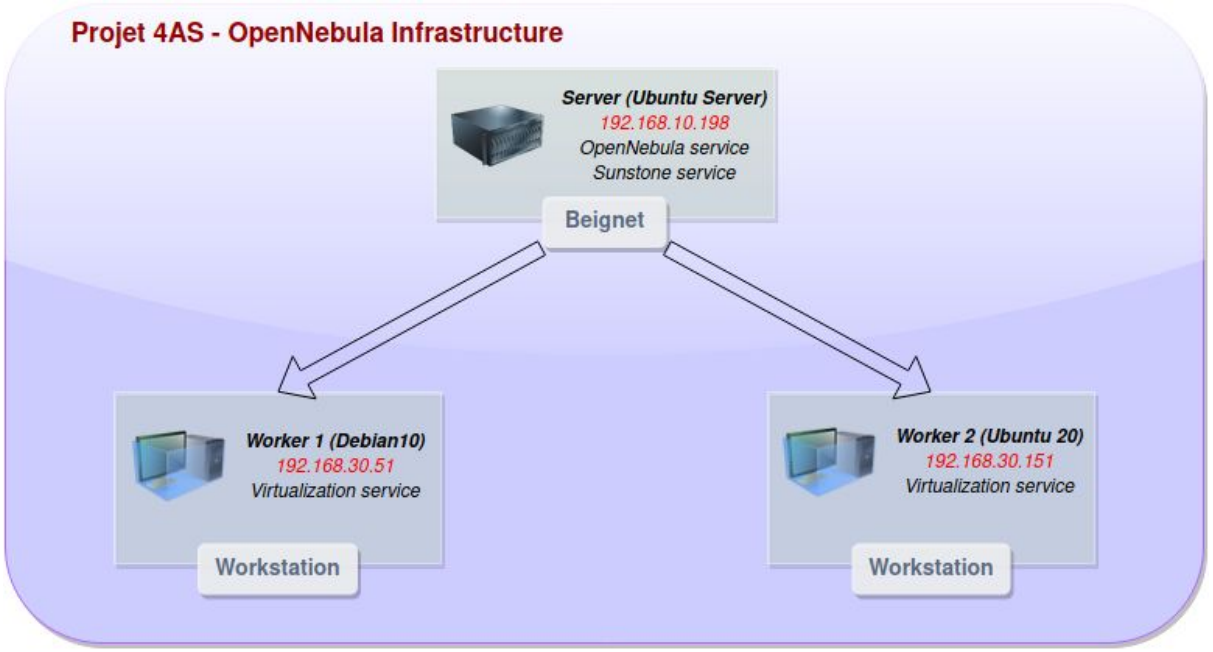
Déploiement OpenNebula et Auto-scale

Objectifs du projet

Le but du projet est de déployer une infrastructure Cloud, utilisant la solution OpenNebula, capable de s'adapter de manière dynamique à la variation de charge.

Dans un exemple plus concret, on imagine un site Web avec un backend susceptible de subir une charge de calcul importante. La solution proposée nous permet par exemple de déployer plus de VMs pour le backend quand la charge monte, et de les supprimer lorsque la charge diminue.

Le projet est réalisé sur l'infrastructure de la salle I204, avec le serveur OpenNebula déployé sur le serveur de virtualisation (Beignet), et les workers déployés sur les machines de travail.



Déploiement du serveur OpenNebula

Le serveur a été déployé sur une VM Ubuntu Server 20, sur le serveur de virtualisation (Beignet).

Afin de pouvoir installer les composants OpenNebula dont on a besoin, il faut tout d'abord rajouter les dépôts, ajouter la clé avec lequel ces dépôts sont signés puis installer les composants:

```
echo "deb https://downloads.opennebula.io/repo/5.12/Ubuntu/20.04 stable
opennebula" > /etc/apt/sources.list.d/opennebula.list

wget -q -O- https://downloads.opennebula.io/repo/repo.key | apt-key add -

apt-get update

apt-get install opennebula opennebula-sunstone opennebula-gate
opennebula-flow
```

L'installation génère la clé RSA privée et publique dans /var/lib/one/.ssh. Cette dernière sera plus tard utilisée par le serveur pour communiquer avec les noeuds. L'installation génère aussi les credentials par défaut pour se connecter à l'interface web Sunstone d'OpenNebula dans /var/lib/one/.one/one_auth. On recopie ensuite ces credentials dans ~/.one/one_auth.

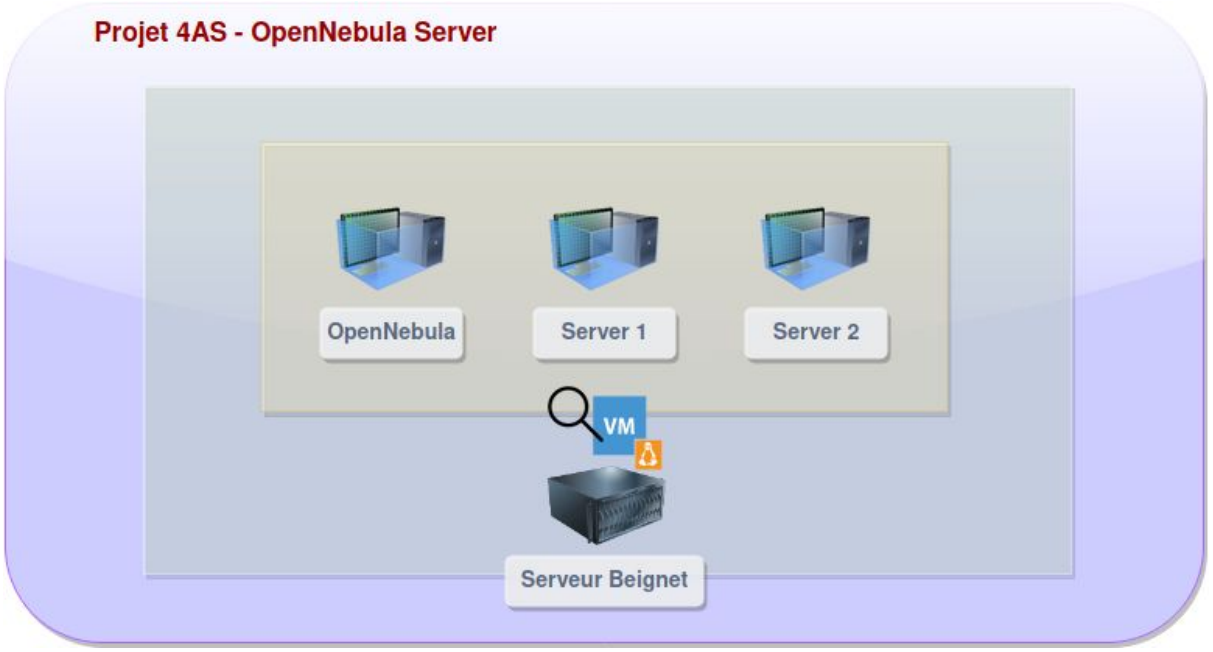
Une fois l'installation est terminée, on peut ainsi démarrer le service principal d'OpenNebula, ainsi que le service de l'interface Sunstone:

```
systemctl start opennebula
systemctl start opennebula-sunstone
```

On peut vérifier que notre installation s'est bien déroulé en affichant l'utilisateur crée par défaut:

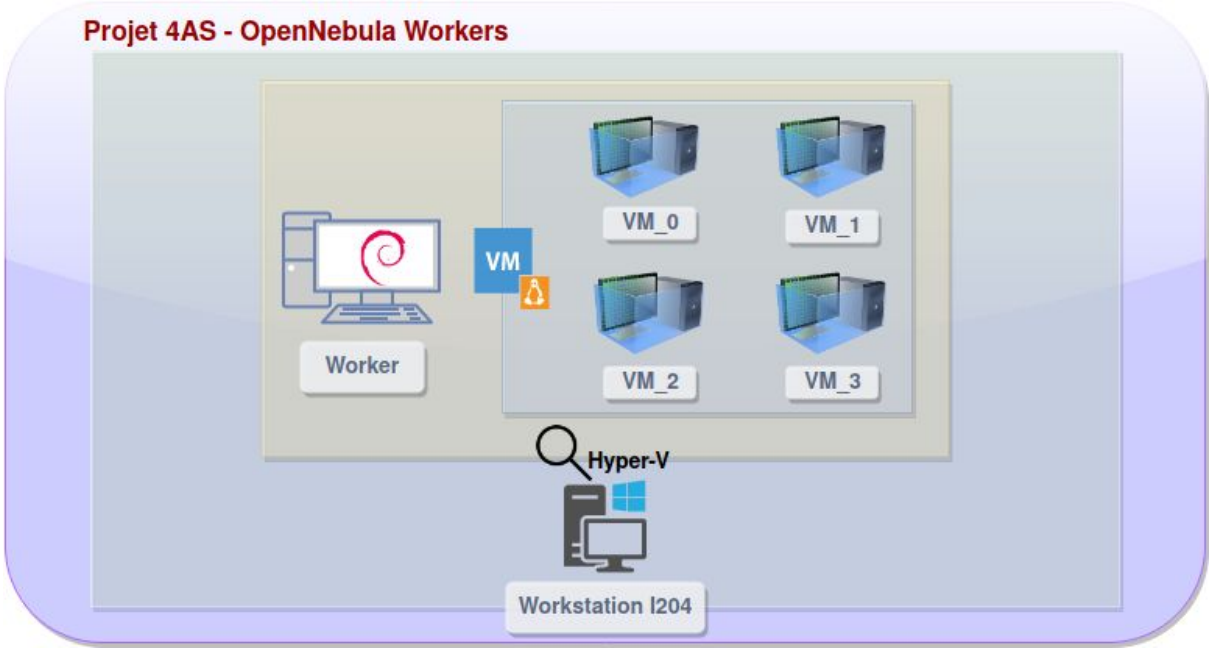
```
oneuser show
```

On peut se connecter à l'interface web sur le port 9869.



Déploiement des Workers

On déploie deux workers OpenNebula sur les workstations de la salle. Comme nous n'avons pas accès aux machines en physique, nous avons dû installer les worker dans des machines virtuelles. Pour éviter le nesting, nous avons d'abord commencé par installer des workers lxd, cependant, nous avons eu plusieurs problèmes empêchant le démarrage des conteneurs.
Nous avons donc décidé de faire du nesting et d'installer des workers KVM. Après avoir activé le nesting dans hyper-v et créé une VM, il s'agit alors d'installer le paquet opennebula-node, puis de distribuer les clés ssh. Par défaut, le paquet opennebula-node configure un worker kvm. Il existe un paquet opennebula-node-lxd et un paquet opennebula-node-firecracker qui configure lxd ou firecracker respectivement comme backend.
Concernant le réseau, nous avons opté pour un bridge simple. OpenNebula offre d'autre possibilité de gestion du réseau, notamment une segmentation du réseaux en utilisant des VLAN.
OpenNebula permet de configurer le stockage des images de vm d'une part et des données persistantes de vms d'autre part. Il est possible de configurer des datastores qui peuvent être partagé entre le serveur et les noeuds. Plusieurs solutions de stockage sont possible, nous avons choisi la plus simple en utilisant un datastore pour les images de vm et un datastore 'system' servant à stocker les éventuelles données persistantes.



Une fois les workers déployés, ceux-ci sont gérés par OpenNebula, soit via l'interface Sunstone, soit via l'utilitaire CLI.

Déploiement d'un service OneFlow

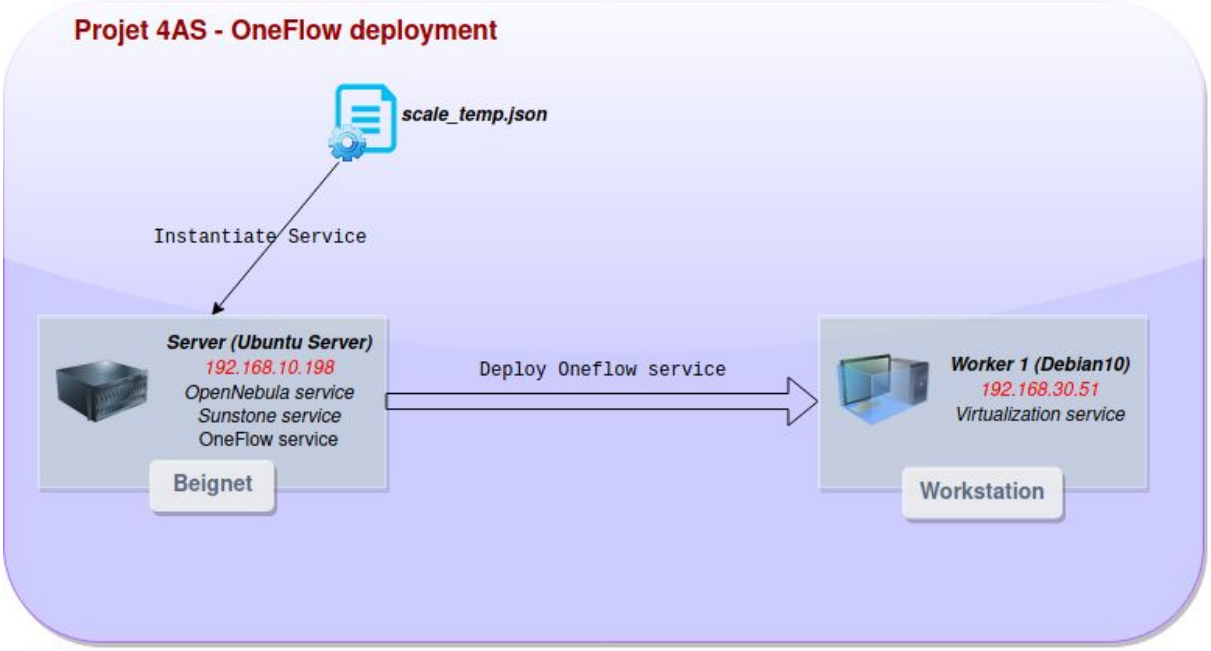
Afin de tester l'Auto-Scale, on déploie le serveur de services OneFlow, on crée un template de service qui supporte l'auto-scaling, et on lance ce dernier. On observe alors l'ajout et la suppression de VMs au sein de notre service, vis-à-vis des manipulations qu'on exécute pour déclencher les différents triggers d'auto-scaling.

Alors on commence tout d'abord par lancer le serveur OneFlow:

```
systemctl start opennebula-flow.service
```

Puis on prépare un template de service qui supporte l'auto-scaling (Annexe A). Ici on choisit un scénario simple, où on définit un service, théoriquement Web, avec deux rôles. D'un côté, le *frontend* qui ne supporte pas l'élasticité, avec une VM allouée. D'un autre côté, le *backend* qui supporte l'élasticité, avec une augmentation des ressources dédiées au rôle jusqu'à 5 VMs.

On règle les conditions d'élasticité vis-à-vis de l'utilisation du CPU. Vu qu'on a un CPU de 0.5 pour les VMs du *backend*, on définit une règle d'ajout de ressource si l'utilisation du CPU dépasse à deux reprises 0.4 pendant 10s. Une règle similaire est définie pour désallouer une ressource si l'utilisation du CPU descend au-dessous du seuil de 0.3.



Dans un scénario similaire (Annexe B), c'est la MEMORY qui sert d'office pour les seuils des règles d'élasticité. Pour une mémoire totale de 768 M, si dépasse 500 M utilisée pendant deux périodes de 10s, on alloue une ressource supplémentaire au "backend". Alors que si on descend au-dessous de 400 M pendant deux périodes de 10s, on désalloue une VM.

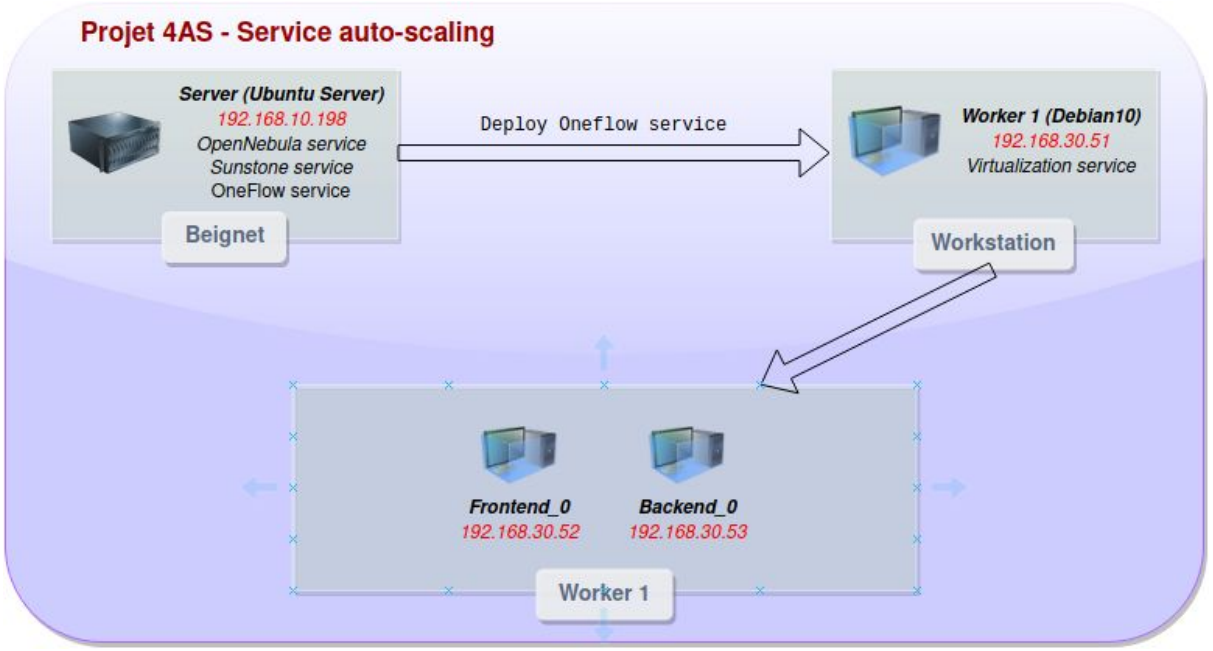
Une fois qu'on définit le template, on l'ajoute à notre serveur. On peut afficher le template, si on a besoin de récupérer son ID. Puis on instancie un service basé sur ce template.

```
oneflow-template create scale_cpu.json
oneflow-template list
oneflow-template instantiate 0 // replace 0 with corresponding template ID
```

On peut afficher les services déployés, ainsi que les VMs déployés. Éventuellement, on peut afficher en détail le service ou une des VMs.

```
oneflow list
oneflow show 1 // replace 1 with corresponding service's ID
onevm list
onevm show 26 // replace 26 with corresponding VM's ID
```

Note: Toutes manipulations précédentes sont aussi bien réalisables depuis l'interface Sunstone (tel est le cas de la démo), que depuis la CLI.



Test de l'Auto-Scale

Afin de tester les règles d'élasticité précédemment définies. On se connecte sur la VM de *backend* déployée par défaut. Pour le template défini dans l'annexe A, on cherche à stresser le processeur:

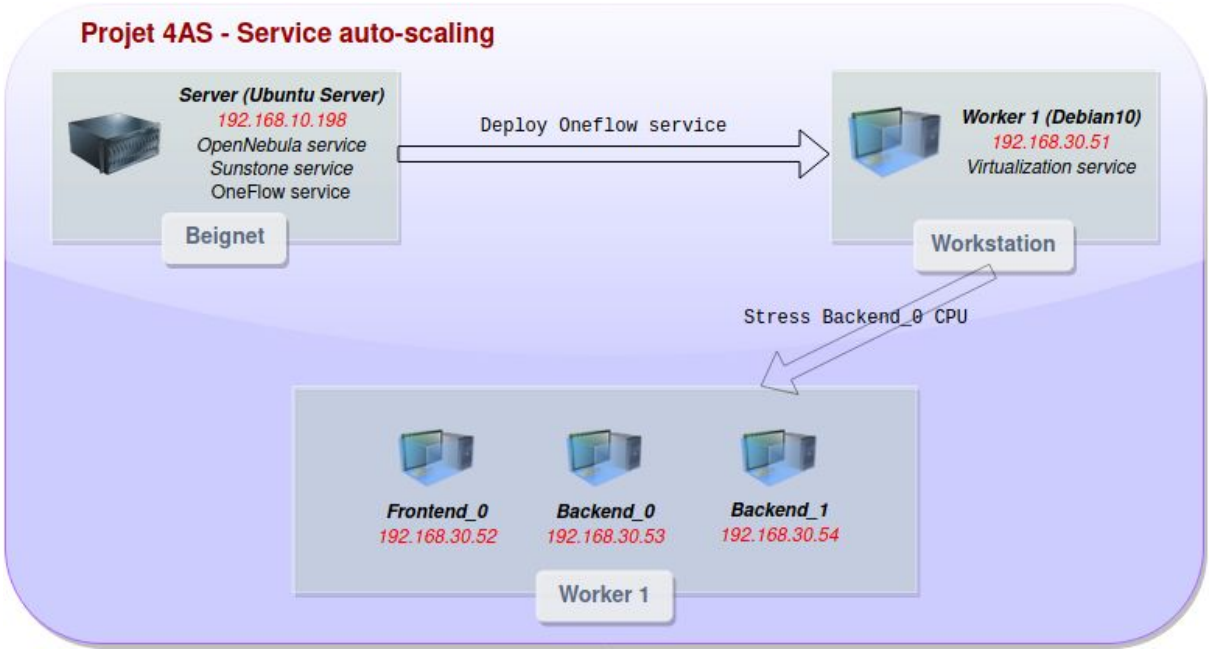
```
while true; do cat /proc/cpuinfo > /dev/null; done
```

Pour le template de service défini dans l'annexe B, c'est plutôt la mémoire RAM qu'on cherche à saturer (même si cela sature également le CPU):

```
:() { :|:& };:
```

Cette commande bash est une "fork-bomb", elle va créer une infinité de processus fils jusqu'à la saturation des ressources du système.

Avec la commande `onevm list`, on peut surveiller le déploiement ou la suppression d'une nouvelle VM.



Nous avons eu des problèmes liés aux métriques utilisées pour l'auto-scale. En effet, la variable "CPU" est celle qui est utilisée dans la plupart des exemples sur internet et une de celles mentionnées dans la documentation officielle. En revanche, chez nous, cette variable est constante et est égale à la quantité de CPU alloué.

Le même problème est à noter pour la RAM.

Annexe A

```
{
   "name": "Test",
   "deployment": "straight",
   "description": "",
   "roles": [
             {
                "name": "Frontend",
                "cardinality": 1,
                "vm_template": 0,
                "elasticity_policies": [],
                "scheduled_policies": []
             },
             {
                "name": "Backend",
                "cardinality": 1,
                "vm_template": 4,
                "min_vms": 1,
                "max_vms": 5,
                "cooldown": 10,
                "elasticity_policies": [
                   {
                      "type": "CHANGE",
                      "adjust": 1,
                      "expression": "CPU>0.4",
                      "period_number": 2,
                      "period": 10
                   },
                   {
                      "type": "CHANGE",
                      "adjust": -1,
                      "expression": "CPU<0.3",
                      "period_number": 1,
                      "period": 10
                   }
      ],
      "scheduled_policies": []
       }
   ],
   "networks": {
      "Bridge": "M|network|| |id:1"
   },
   "ready_status_gate": false
}
```

Annexe B

```
{
   "name": "Test",
   "deployment": "straight",
   "description": "",
   "roles": [
             {
                "name": "Frontend",
                "cardinality": 1,
                "vm_template": 0,
                "elasticity_policies": [],
                "scheduled_policies": []
             },
             {
                "name": "Backend",
                "cardinality": 1,
                "vm_template": 4,
                "min_vms": 1,
                "max_vms": 5,
                "cooldown": 10,
                "elasticity_policies": [
                   {
                      "type": "CHANGE",
                      "adjust": 1,
                      "expression": "MEMORY>500",
                      "period_number": 2,
                      "period": 10
                   },
                   {
                      "type": "CHANGE",
                      "adjust": -1,
                      "expression": "MEMORY<400",
                      "period_number": 1,
                      "period": 10
                   }
      ],
      "scheduled_policies": []
       }
   ],
   "networks": {
      "Bridge": "M|network|| |id:1"
   },
   "ready_status_gate": false
}
```