TP 3 – Docker Swarm

S4 ASI – 2024 - v1

CAMPUS DE PARIS - Siège Social • 9 rue Vésale - 75005 Paris • +33 (0)1 55 43 23 23
CAMPUS D'IVRY-SUR-SEINE - 74 bis avenue Maurice Thorez - 94200 Ivry-sur-Seine • +33 (0)1 43 90 21 21
CAMPUS DE LAVAL - 38 rue des docteurs Calmette et Guérin - 53000 Laval • +33 (0)2 43 59 24 24
CAMPUS D'AGEN – 156 avenue Jean Jaurès – 47000 Agen • +33 (0)5 53 66 37 70
CAMPUS DE DAX – Immeuble Confluences – 182 avenue Saint Vincent de Paul – 40100 DAX • +33 (0)5 33 09 13 83

ESIEA, 9 rue Vésale 75005 Paris – Association loi 1901 – N° W751037789 – SIRET 311 349 138 00017 – APE 8542Z

Attention pour réaliser ce TP vous n'avez que 4 heures une fois connecté

0. Play with docker

La mise en place d'un cluster peut être contraignant car cela nécessite d'avoir une infrastructure avec différentes machines qui communiquent entre elles. Un outil gratuit et accessible depuis un navigateur nommé « Play With Docker » permet d'instancier facilement des nodes Docker afin de créer un cluster Swarm.

Allez sur https://labs.play-with-docker.com/, identifiez-vous avez votre compte Docker Hub et démarrez une nouvelle session.

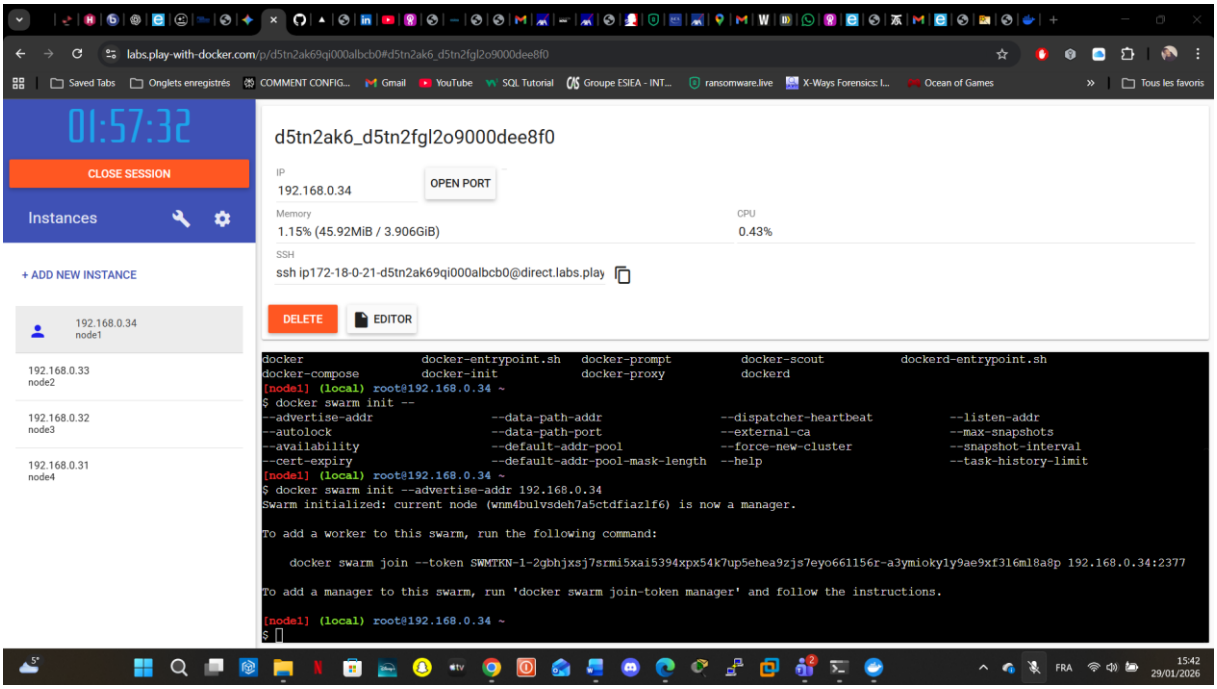
Les sessions durent 4 heures après cela les instances sont supprimées mais vous pouvez en redémarrer une nouvelle. Les instances sont basées sur Alpine Linux et ont déjà Docker et Compose d'installés.

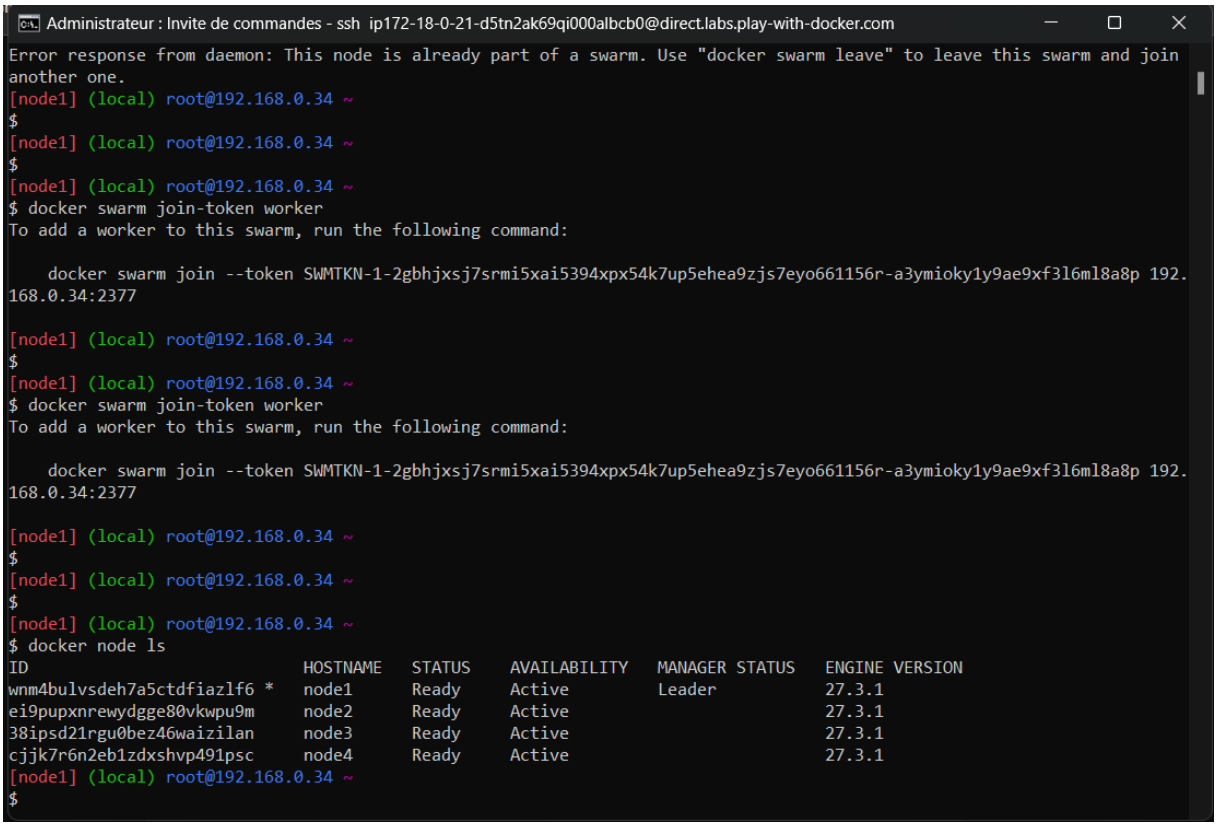
Concernant les aspects réseaux, les instances ont un accès à Internet non filtré et elles sont aussi connectées sur un réseau privé (il y a donc deux cartes réseaux). Les instances sont accessibles en SSH et si des services web sont hébergés ils sont aussi accessibles depuis l'extérieur. Normalement, des liens apparaissent avec le numéro de port lorsqu'un service ou un conteneur est démarré (si ce n'est pas le cas essayez avec un autre navigateur). Dans le cas où les liens n'apparaissent pas, le format de l'URL est le suivant : http://ip172-18-0-75-blke760nddr000cukeag-8080.direct.labs.play-with-docker.com (172-18-0-75 = @IP de l'instance en remplaçant les « . » par des « - » ; blke760… = votre numéro de session ; 8080 = port en écoute sur l'host)..

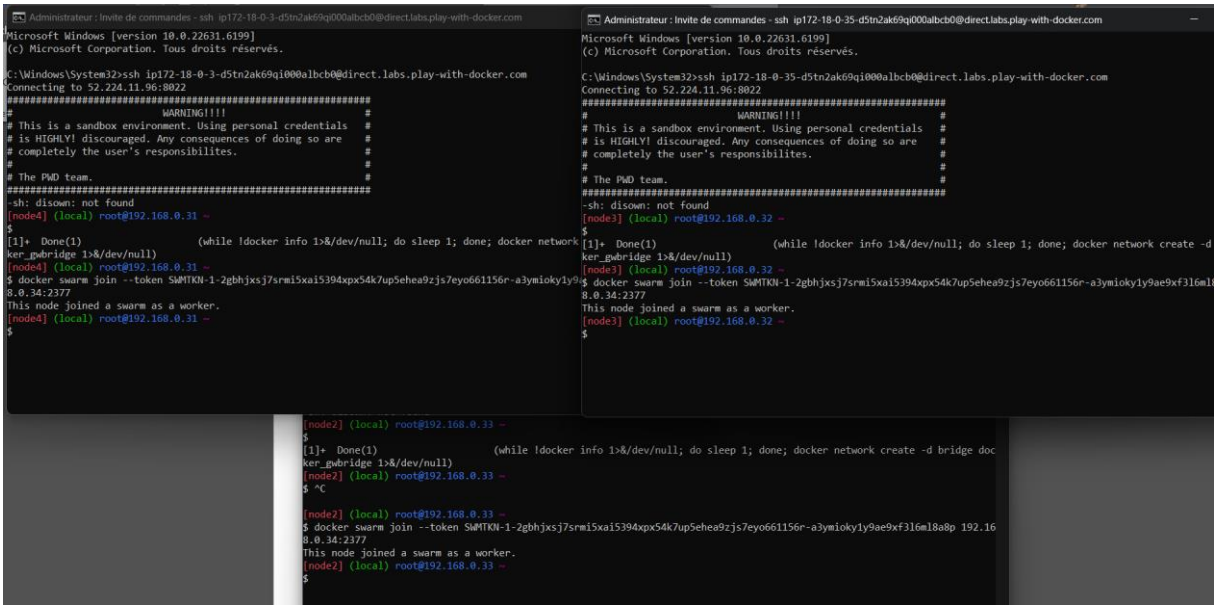
1. Exercice 1 : Cluster

Mettez en place un cluster Swarm composé des nodes suivants :

- node1 : manager et worker
- node2 : worker
- node3 : worker
- node4 : worker







[node1] (local) root@192.168.0.34 ~

$ docker node ls

ID HOSTNAME STATUS AVAILABILITY MANAGER STATUS ENGINE VERSION

wnm4bulvsdeh7a5ctdfiazlf6 * node1 Ready Active Leader 27.3.1

ei9pupxnrewydgge80vkwpu9m node2 Ready Active 27.3.1

38ipsd21rgu0bez46waizilan node3 Ready Active 27.3.1

cjjk7r6n2eb1zdxshvp491psc node4 Ready Active 27.3.1

[node1] (local) root@192.168.0.34 ~

2. Exercice 2 : Replicas

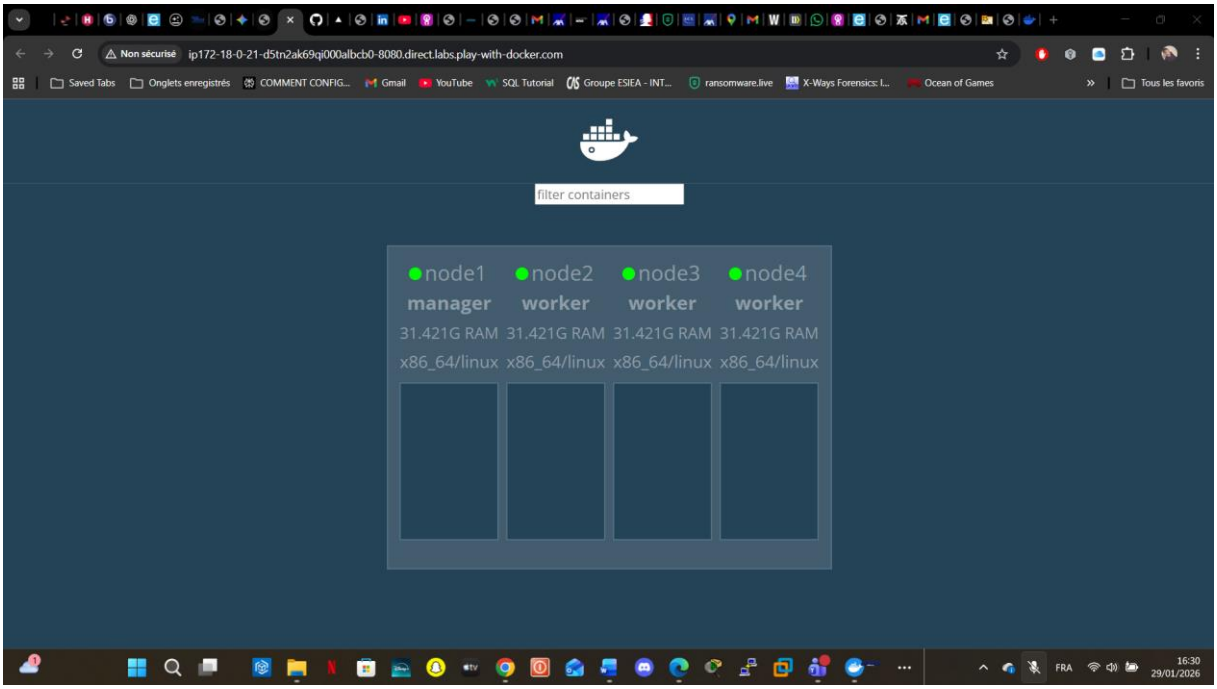
Une des principales fonctionnalités de Docker Swarm et la réplication sur les nodes du cluster. Nous allons voir cela au travers de la création d'un service qui « s'autodétruit » et un outil nous permettant de visualiser le placement des tasks au sein du cluster.

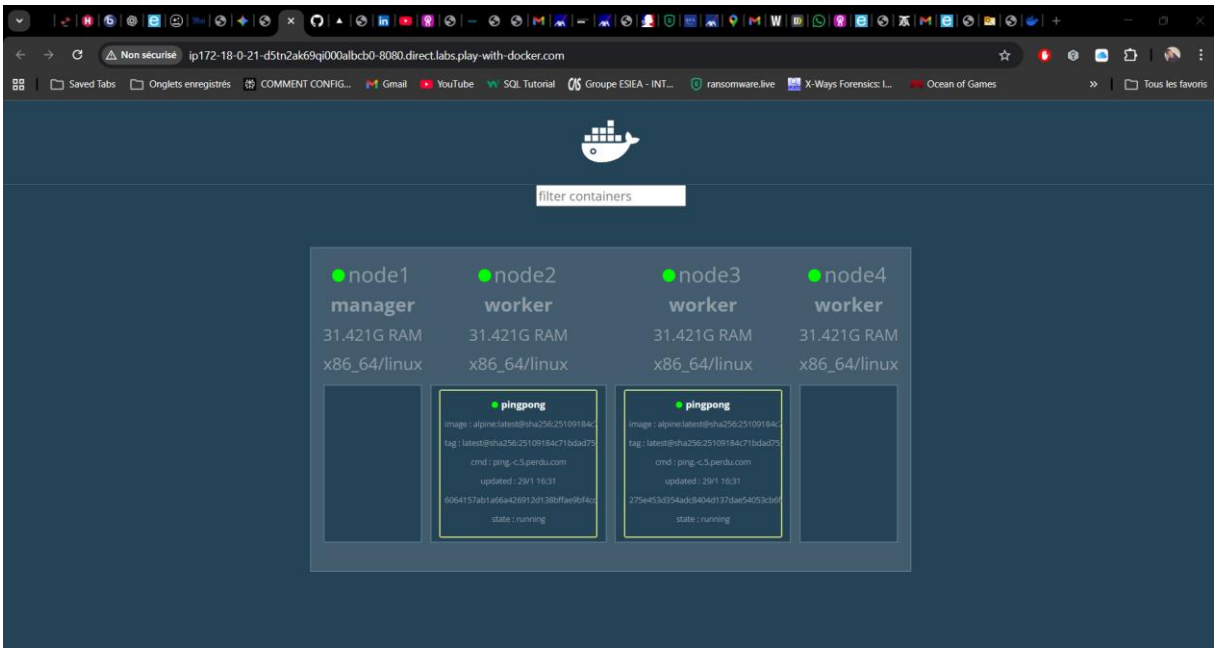
```
[node1] (local) root@192.168.0.34 ~
$ docker run -d -p 8080:8080 -v /var/run/docker.sock:/var/run/docker.sock dockersamples/visualizer
Unable to find image 'dockersamples/visualizer:latest' locally
latest: Pulling from dockersamples/visualizer
ddad3d7c1e96: Pull complete
3a8370f05d5d: Pull complete
71a8563b7fea: Pull complete
119c7e14957d: Pull complete
28bdf67d9c0d: Pull complete
12571b9c0c9e: Pull complete
e1bd03793962: Pull complete
3ab99c5ebb8e: Pull complete
94993ebc295c: Pull complete
021a328e5f7b: Pull complete
Digest: sha256:530c863672e7830d7560483df66beb4cbbcd375a9f3ec174ff5376616686a619
Status: Downloaded newer image for dockersamples/visualizer:latest
cd7dacfe5c2dd438a1defd140a921d63a206c8804368323207043097 43fb2958
[node1] (local) root@192.168.0.34 ~
$
[node1] (local) root@192.168.0.34 ~
$
[node1] (local) root@192.168.0.34 ~
$ docker service create --name pingpong --replicas 2 -d alpine ping -c 5 perdu.com
m2p5duwady1391stf3o4x5y00
[node1] (local) root@192.168.0.34 ~
$
```

Instancier un service sur le cluster dont voici les caractéristiques :

- Nom : pingpong
- Image : alpine
- Réplicas : 2
- Commande : ping -c 5 perdu.com

La commande « ping -c 5 perdu.com » va faire en sorte que le conteneur soit stoppé au bout de 5 pings (car la commande principale s'arrête). Du coup, lors de la création du service, Docker va attendre de manière infinie. Pour éviter ce comportement il faut rajouter l'argument « -d » qui s'occupe de la création du service en tâche de fond.





Pour visualiser le placement des tasks sur notre cluster, nous pouvons utiliser l'image dockersamples/visualizer qui fournit une interface web où est affiché le placement des tasks en temps réel. À noter, le conteneur doit être lancé sur le manager pour pouvoir obtenir les informations nécessaires. Aussi, le conteneur doit être exécuté en « standalone », c'est-à-dire qu'il n'est pas exécuté sur le cluster mais directement sur la machine node1. Vous trouverez un exemple d'utilisation sur la page Docker Hub : https://hub.docker.com/r/dockersamples/visualizer

Une fois le conteneur lancé accédez à l'interface web et analysez le placement des tasks.

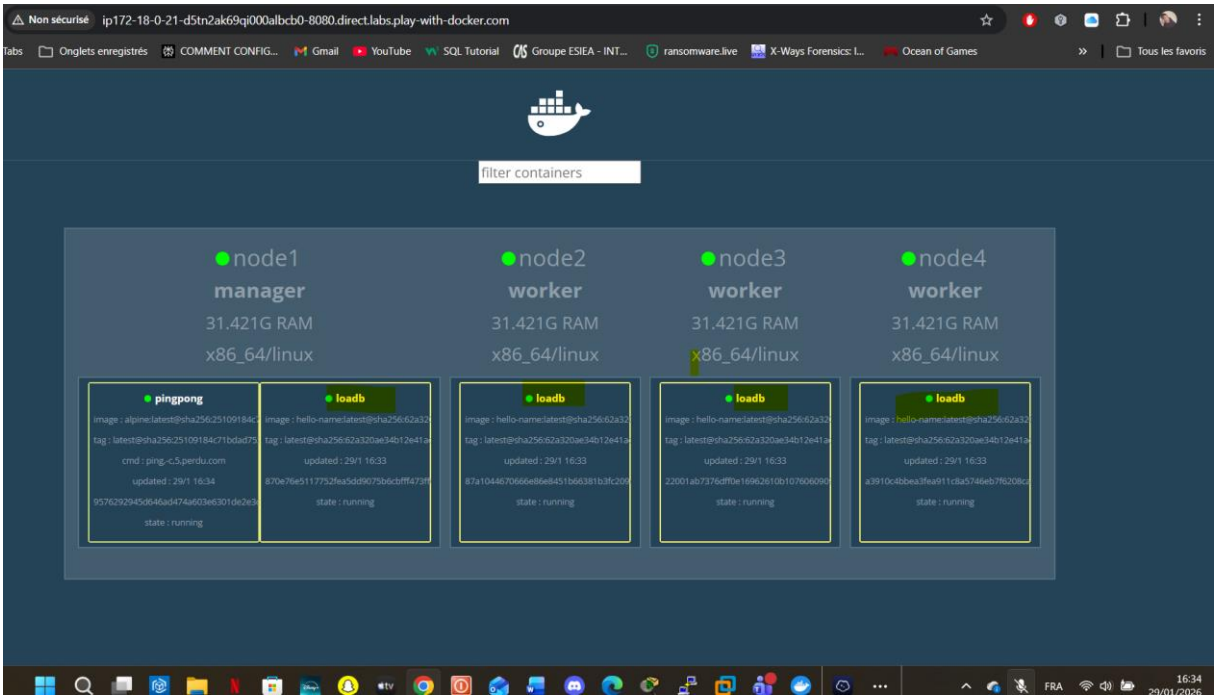
3. Exercice 3 : Load Balancing

Une autre fonctionnalité intéressante est celle du load-balancing nativement intégré à Docker. Nous allons expérimenter cela au travers d'un service web qui affichera une information unique (ex : son nom d'hôte) et en essayant de se connecter via les différents nodes.

Créez un service dont voici les spécificités :

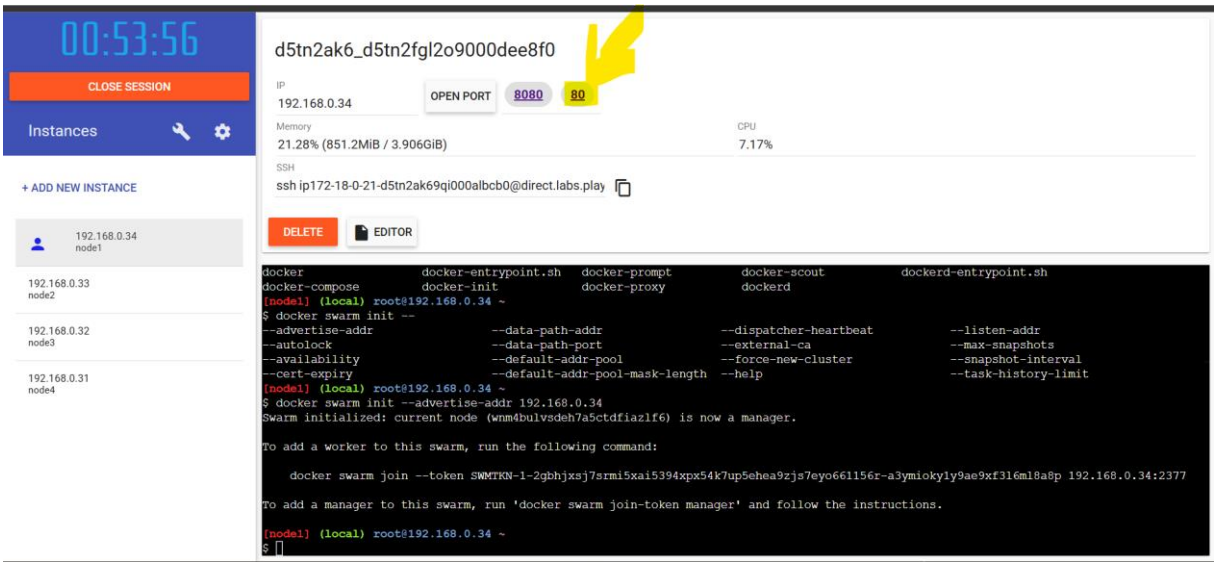
- Image : https://hub.docker.com/r/flointech/hello-name
- Réplicas : 4
- Nom : loadb
- Port : 80:80

```
[node1] (local) root@192.168.0.34 ~
$ docker service create --name loadb --replicas 4 -p 80:80 flointech/hello-name
8fe1sw9rm9u8deixh7ctk8y1y
overall progress: 4 out of 4 tasks
1/4: running   [==================================================>]
2/4: running   [==================================================>]
3/4: running   [==================================================>]
4/4: running   [==================================================>]
verify: Service 8fe1sw9rm9u8deixh7ctk8y1y converged
[node1] (local) root@192.168.0.34 ~
$
```

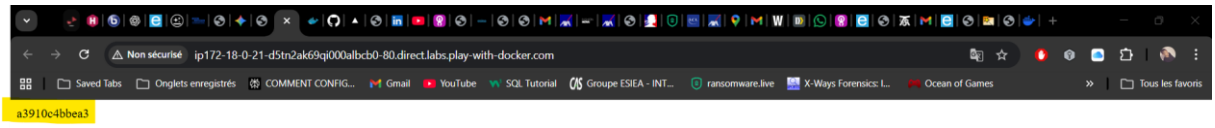


L'image utilisée est un serveur web qui s'occupe d'afficher le nom d'hôte de la machine. Dans le cas de Docker, le nom d'hôte est son identifiant tel qu'il est visible sur l'host.

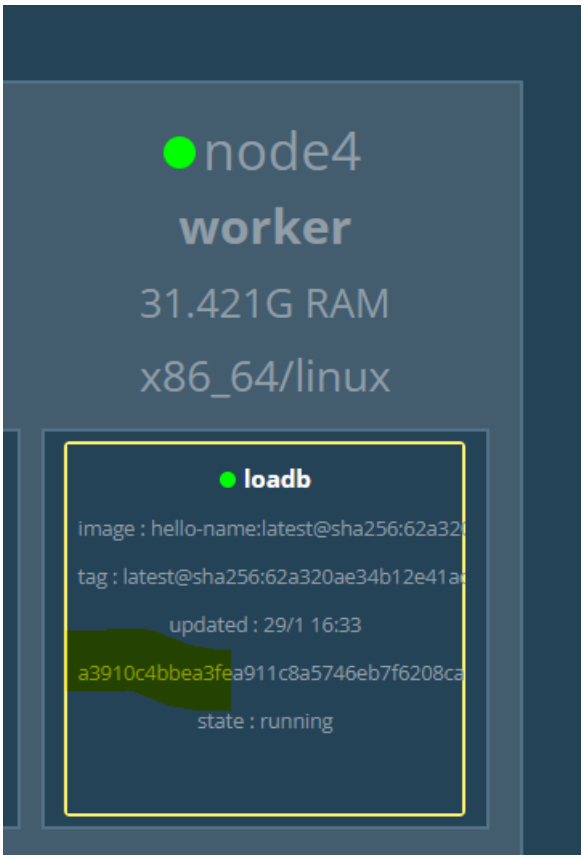
Accédez au service via l'URL du manager et rafraichissez la page (ctrl+F5) plusieurs fois afin de vous assurer que le load balancing fonctionne bien. Pour chacun des identifiants retrouvez le node sur lequel le conteneur est hébergé.



Essayez d'accéder au service via les URLs des workers : ça devrait aussi fonctionner !



a3910c4bbea3



CORRESPOND A l'ID du Node 4 après chaque réactualisation il retourne un ID différent correspondant a l'une des worker ce qui confirme que le loadbalancing fonctionne bien.

4. Exercice 1 : Private Registry

Utiliser un registry public pour stocker ses propres images peut être parfois problématique pour des raisons de confidentialité. Nous allons donc mettre en place on registry local sur le manager. Ainsi, il sera possible d'y déposer des images et de les récupérer depuis les autres workers.

Normalement, la connexion à un registry est censée se faire au travers de HTTPS. Cependant, pour des besoins de test, nous allons forcer Docker à utiliser HTTP. Sur chaque node, modifiez la directive « insecure-registries » du fichier /etc/docker/daemon.json pour rajouter le registry qui sera hébergé sur le manager : "insecure-registries": ["127.0.0.1", "192.168.0.38:5000"], (remplacer 192.168.0.38 par l'IP de votre manager). Une fois le fichier modifié, rechargez la configuration Docker sur chaque node via la commande kill -HUP $(pidof dockerd).

Lancez un conteneur « classique » basé sur l'image registry (vous trouverez un exemple ici : https://hub.docker.com/_/registry) sur le manager.

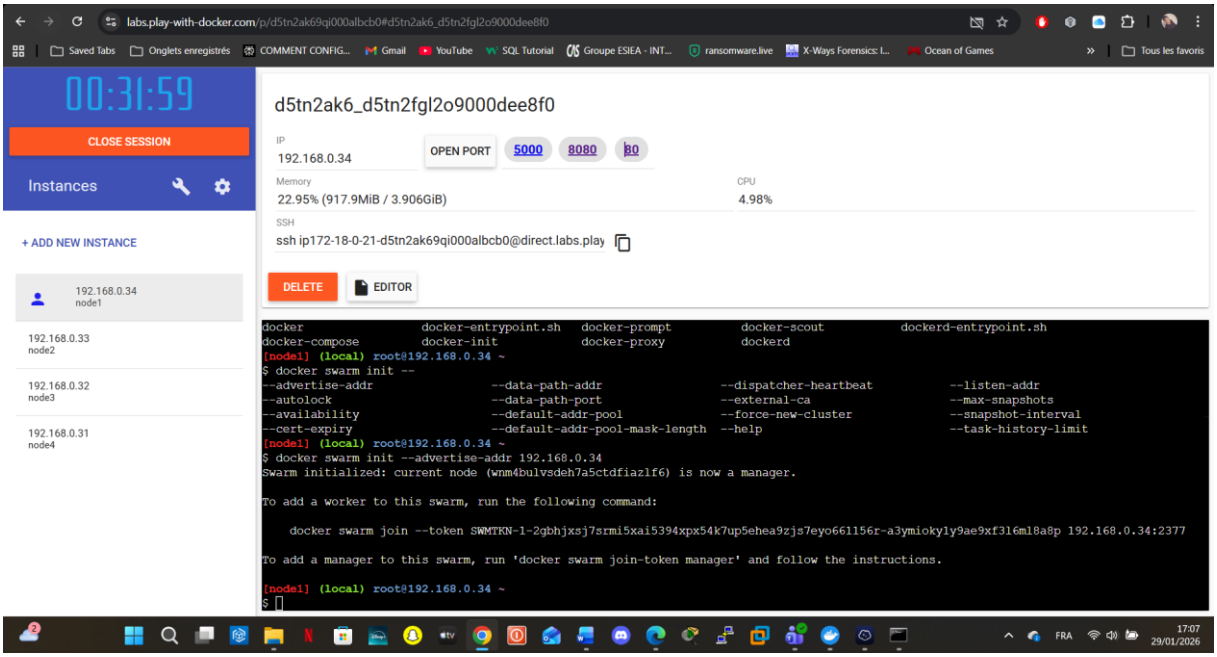
Créez une image « basique » (ex : ping à l'infini sur un serveur) que vous déposerez sur le registry. Voici un exemple :

docker build -t 192.168.0.38:5000/mon-image

docker push 192.168.0.38:5000/mon-image

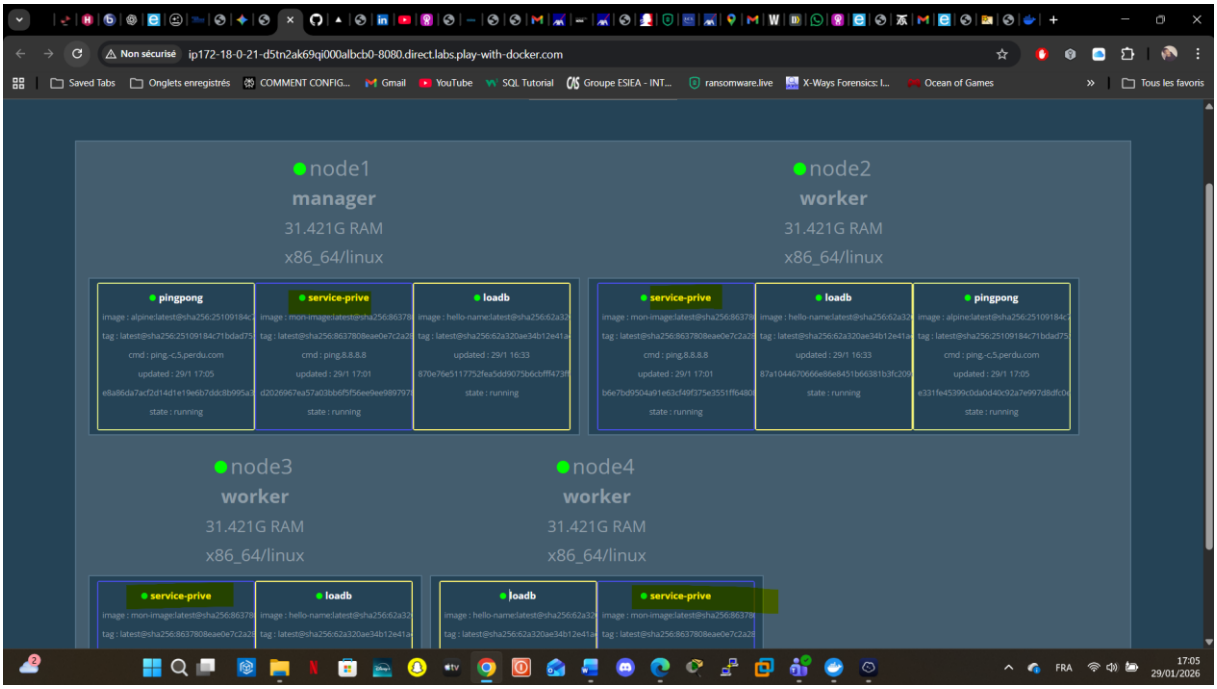
```
[node1] (local) root@192.168.0.34 ~
$ docker pull alpine
Using default tag: latest
000/mon-imagelatest: Pulling from library/alpine
Digest: sha256:25109184c71bdad752c8312a8623239686a9a2071e8825f20acb8f2198c3f659
Status: Downloaded newer image for alpine:latest
docker.io/library/alpine:latest
[node1] (local) root@192.168.0.34 ~
$ docker tag alpine 192.168.0.34:5000/mon-image
[node1] (local) root@192.168.0.34 ~
$ docker push 192.168.0.34:5000/mon-image
Using default tag: latest
The push refers to repository [192.168.0.34:5000/mon-image]
989e799e6349: Pushed
latest: digest: sha256:8637808eae0e7c2a2875ab58a1c7f74999823afdb67b4570e3bf778eeccc119b size: 527
[node1] (local) root@192.168.0.34 ~
$
[node1] (local) root@192.168.0.34 ~
$ docker service create --name service-prive --replicas 4 192.168.0.34:5000/mon-image ping 8.8.8.8
vpo1bf8ljf3i48xtebrhdpva8
overall progress: 4 out of 4 tasks
1/4: running   [==================================================>]
2/4: running   [==================================================>]
3/4: running   [==================================================>]
4/4: running   [==================================================>]
verify: Service vpo1bf8ljf3i48xtebrhdpva8 converged
[node1] (local) root@192.168.0.34 ~
$
```

Vous pouvez maintenant créer un service basé sur l'image « 192.168.0.38:5000/mon-image » : les workers iront chercher l'image sur le repository hébergé sur le manager.



Cette interface (outil Visualizer) présente l'état final du cluster composé d'un Manager (node1) et de trois Workers (node2, node3, node4). On y observe la répartition intelligente des tâches effectuée par l'orchestrateur:

- ***Service pingpong (Exercice 2) : Deux réplicas basés sur l'image alpine. Ils apparaissent en mouvement car ils se relancent automatiquement après chaque cycle de 5 pings.***
- ***Service loadb (Exercice 3) : Quatre réplicas du serveur web répartis sur l'ensemble des nœuds du cluster (un par node) pour assurer l'équilibrage de charge.***
- ***Service service-prive (Exercice 4) : Quatre réplicas déployés à partir de notre Registry local hébergé sur le manager. Cela démontre que les workers parviennent à récupérer l'image personnalisée mon-image via la configuration insecure-registries***



Ce TP a permis d'appréhender les concepts fondamentaux de l'orchestration avec **Docker Swarm** à travers la mise en place d'un cluster multi-nœuds. Les expérimentations réalisées ont démontré les capacités natives de l'outil dans trois domaines clés :

- **Haute Disponibilité et Résilience** : L'exercice sur le service pingpong a illustré comment Swarm maintient l'état souhaité (2 réplicas) en relançant automatiquement les tâches dès qu'un conteneur s'arrête.
- **Équilibrage de Charge (Load Balancing)** : Grâce au *Routing Mesh*, nous avons pu accéder au service loadb depuis n'importe quel nœud du cluster, Docker répartissant de manière transparente les requêtes entre les 4 réplicas.
- **Gestion d'Infrastructure Privée** : L'installation d'un **Registry local** a prouvé qu'il est possible de s'affranchir de Docker Hub pour stocker et déployer des images personnalisées de manière sécurisée et centralisée au sein de sa propre infrastructure.

En conclusion, Docker Swarm s'avère être une solution d'orchestration légère et efficace, permettant de transformer un groupe de machines isolées en une entité unique, robuste et scalable.