IUT de Vélizy-Rambouillet

CAMPUS DE VÉLIZY-VILLACOUBLAY

Durand Antonin
Jougla Maxime
Parciany Benjamin
Zehren William

Compte rendu installation application

Table des matières

- I- Introduction
- II- Installation du kit Cluster hat
  - A) Choix des images pour le kit Cluster Hat
  - B) Installation des images sur chaque Raspberry Pi et premier démarrage
  - C) Configuration du ssh des Raspberry pi du kut Cluster Hat
  - D) Installation du service Fail2Ban sur le Raspeberry Pi host
- III- Installation de l'application
  - A) Introduction
  - B) Création de l'image pour le serveur web
  - C) Création de l'image pour la base de données
  - D) Création du docker swarm
  - E) Création du stack de l'application

I- Introduction

Dans ce rapport seront présentées les différentes étapes et installations effectuées pour mettre en place le kit Cluster hat, ainsi que l'application qui sera hébergée sur ce dernier. En premier lieu, seront montrées les différentes étapes pour mettre en route le kit Cluster, de la création de l'iso pour le Raspberry principal, en passant par l'allumage des 4 Raspberry Pi Zero, à l'installation de différents utilisateurs et logiciels pour manager ce dernier. Puis, dans un second temps, sera présenté comment l'application est installée, hébergée et gérée sur le kit Cluster à l'aide de docker swarm.

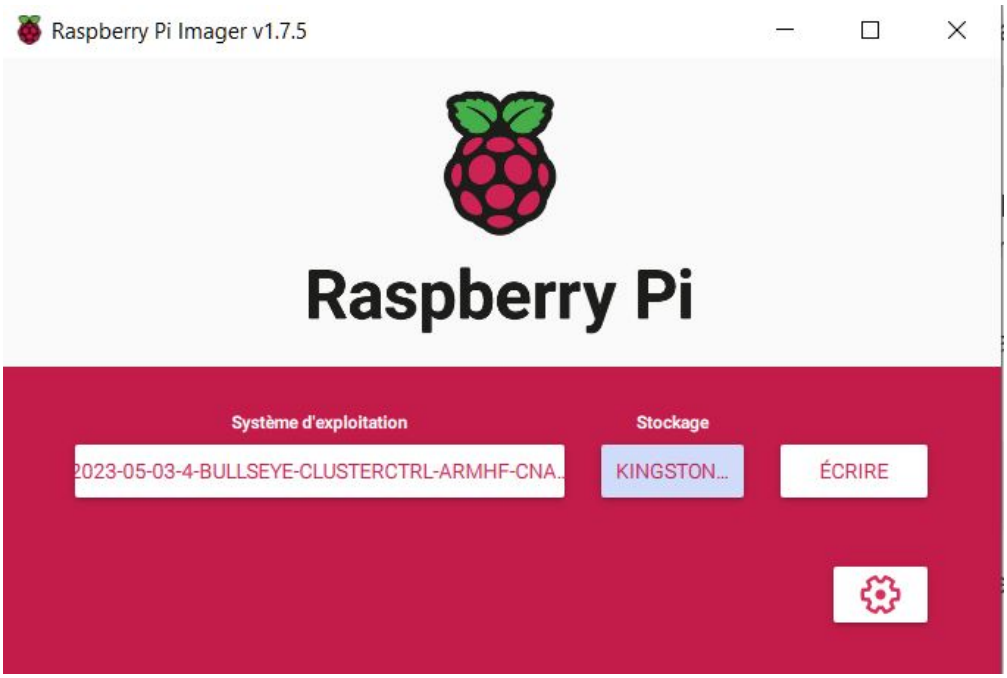
II- Installation du kit Cluster hat

A) Choix des images pour le kit Cluster Hat

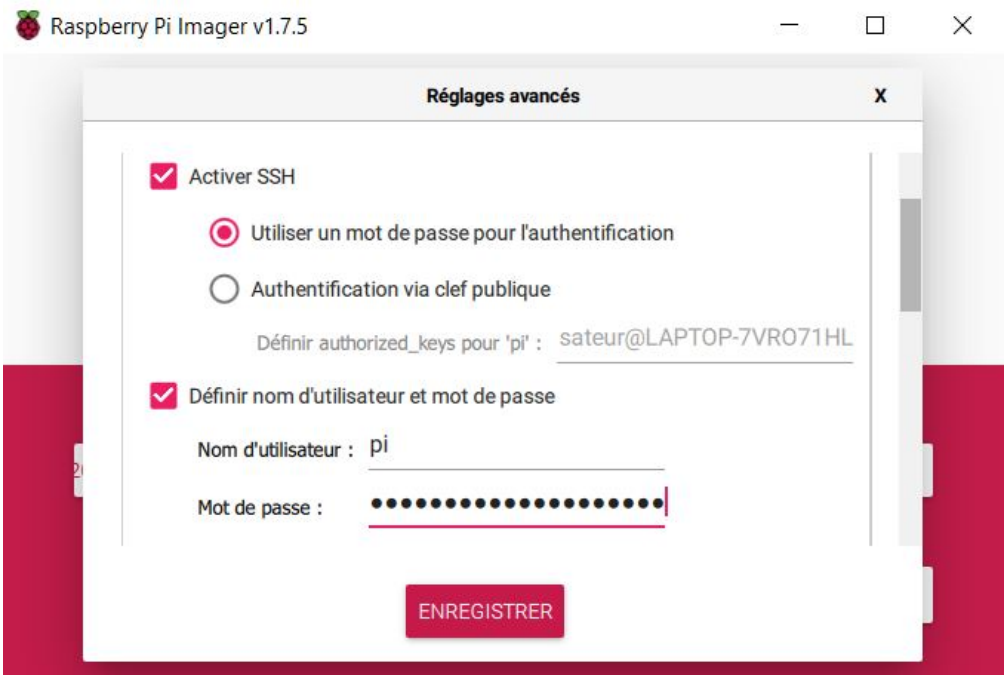
La première étape est le choix des images pour le RPI 4 host, et les RPI Zero. Nous avons choisit de prendre ces dernières sur le site Cluster CTRL , car il fournit des images permettant d'utiliser et contrôler facilement le cluster. Ainsi, **CNAT-Desktop Controller** est l'image du RPI host, et **Lite Bullseye image PN** est l'image pour le N-ième RPI zero. Le premier RPI zero aura l'image *P1*, le deuxième *P2*, le troisième *P3* et le quatrième *P4*. Comme les Raspberry du kit Cluster Hat ont des processeurs 32 bits, par conséquent il est nécessaire de prendre la version 32 bits de ces images, qui est aussi disponible sur le site. L'avantage de ces images et plus précisément l'image du RPI host, est que cette dernière utilise la méthode NAT (Network Address Translation) pour créer un sous-réseau **172.19.181.0/24**, où chaque RPI zero se verra assigner une adresse ip fixe. Par exemple, le premier RPI zero aura l'adresse *172.19.181.1*, et ainsi de suite. Le RPI host aura l'adresse *172.19.181.254*.

B) Installation des images sur chaque Raspberry Pi et premier démarrage

La deuxième étape consiste à l'installation des images sur les cartes micro sd du RPI host et des 4 RPI Zero. Pour ce faire, nous avons utilisé le logiciel **Raspberry Pi Imager**, qui comme **balenaEtcher**, permet de flasher des images sur différents supports physiques comme une carte micro sd ou un disque dur. L'avantage de **Raspberry Pi Imager** est qu'il permet de personnaliser l'image qu'on veut flasher, en activant le *ssh*, et en créant un utilisateur *pi* dans notre cas. Cela nous évite par exemple à avoir à activer manuellement le ssh sur chaque RPI zero, où il aurait fallu monter chaque carte micro sd dans une distribution Linux, pour ensuite créer un fichier *ssh* dans le répertoire *boot*.



Comme le montre l'image ci-dessus, pour flasher l'image à l'aide de Pi Imager, il faut en premier lieu sélectionner le système d'exploitation, dans notre cas l'image CNAT ou les images lite Bullseye.
Puis dans un deuxième lieu, il faut choisir le support sur lequel installer le système d'exploitation, à savoir une carte micro sd dans notre situation. Enfin, on peut modifier certains paramètres du système d'exploitation à installer à l'aide du rouage en bas à droite, et ensuite appuyer sur ***Ecrire*** *pour installer ce dernier sur la carte micro sd.*



L'image ci-dessus permet de montrer les paramètres les plus importants que nous avons utilisé pour chacune des images du kit Cluster Hat, comme l'activation du SSH et la définition d'un utilisateur et de son mot de passe.

Une fois les images créées, installées sur les cartes micro sd, on allume le RPI host. Ensuite, on le met à jour manuellement, puis on crée un script bash pour faire en sorte qu'il soit à jour à chaque fois qu'il démarre. Dans un second temps, on démarre le cluster de RPI zero grâce à la commande ci-dessous qui nous est fournie par l'image **CNAT** :

```
dev@cnat:~$ clusterhat on
```

Si on veut éteindre le cluster, il suffit d'effectuer la commande suivante, qui est similaire à celle pour allumer le cluster.

```
dev@cnat:~$ clusterhat off
```

Pour vérifier que le cluster est allumé, on exécute la commande suivante qui permet d'obtenir toutes les adresses IP qui sont dans le *cache ARP*, et notamment les adresses IP des RPI zero.

```
dev@cnat:~ $ arp -a
? (172.19.181.2) at 00:22:82:ff:ff:02 [ether] on br0
? (172.19.181.4) at 00:22:82:ff:ff:04 [ether] on br0
? (172.19.181.1) at 00:22:82:ff:ff:01 [ether] on br0
? (172.19.181.3) at 00:22:82:ff:ff:03 [ether] on br0
```

Ainsi, le sous-réseau en **172.19.181.0** a été créé par le RPI host, pour contenir les 4 RPI zero.

C) Configuration du ssh des Raspberry Pi du kit Cluster Hat

Ensuite, la troisième étape est la configuration du système de connexion en ssh entre le RPI host et les 4 RPI zero, dans les 2 sens, pour faciliter la communication entre le RPI host et ces derniers.

Dans un premier temps, on change le hostname de chaque RPI zero. Le **hostname**, ou nom d'hôte, est une étiquette que l'on peut donner à un appareil dans un réseau. Dans notre cas, nous allons attribuer un hostname pour chaque RPI zero, stockés dans le fichier */etc/hosts/* du RPI host. On peut ainsi utiliser ce hostname au lieu de l'adresse IP du RPI zero lors de la connexion en ssh entre le RPI host et ce dernier. De la même manière, pour pouvoir utiliser ce hostname pour se connecter au RPI host ou autre RPI zero depuis un autre RPI zero, il faut ajouter les adresses IP de ces derniers et les hostnames associés dans */etc/hosts/* du RPI zero en question.

Pour vérifier que cela fonctionne, on regarde les hostnames accessibles depuis le RPI host.

```
dev@cnat:~$ cat /etc/hosts
127.0.0.1       localhost
::1             localhost ip6-localhost ip6-loopback
ff02::1         ip6-allnodes
ff02::2         ip6-allrouters

127.0.1.1       cnat
172.19.181.1 pi1
172.19.181.2 pi2
172.19.181.3 pi3
172.19.181.4 pi4
```

Ensuite on vérifie qu'on peut se connecter à un RPI zero depuis le RPI host en utilisant son hostname.

```
dev@cnat:~ $ ssh pi@pi2
pi@pi2 password:
Linux p2 6.1.21+ #1642 Mon Apr  3 17:19:14 BST 2023 armv6l

The programs included with the Debian GNU/Linux system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.
Last login: Fri Dec  1 14:51:07 2023 from 172.19.181.254
pi@p2:~ $
```

Dans un second temps, on va faire en sorte de pouvoir se connecter en ssh à un RPI zero depuis le host et inversement sans avoir à préciser un mot de passe, ce qui va être important pour certains programmes de calculs distribués à venir, et pour gagner en rapidité. Pour ce faire, depuis le RPI host, nous allons configurer le ssh à l'aide du fichier **.ssh/config** dont le format est le suivant :

```
Host hostname1
    SSH_OPTION value
    SSH_OPTION value
```

Ces 3 lignes permettent d'ajouter des options de connexion en ssh à un autre système. Dans notre cas, pour la connexion au premier RPI zero, on remplace **hostname1** par le nom que nous voulons utiliser pour se connecter à ce dernier. Ensuite, la première option que nous allons mettre est le hostname du RPI zero sur lequel se connecter en ssh, dans notre cas *pi1*. Enfin, on indique l'utilisateur du RPI zero à utiliser pour établir la connexion, qui est dans notre cas *pi*. Après avoir écrit et adapté ces lignes pour les 3 autres RPI zero, on enregistre les modifications et le **.ssh/config** ressemble désormais à cela :

```
dev@cnat:~ $ cat .ssh/config
Host pi1
    Hostname pi1
    User pi
Host pi2
    Hostname pi2
    User pi
Host pi3
    Hostname pi3
    User pi
Host pi4
    Hostname pi4
    User pi
```

Maintenant que l'on peut communiquer rapidement avec ssh on va créer une clé ssh et la partager aux autres nodes avec la commande `ssh-copy-id`. Pour créer une clé ssh on utilise la commande `ssh-keygen -t rsa`, sans lui donner de nom ou de passphrase. On obtient deux clés shh : `id_rsa` et `id_rsa.pub` c'est la clé publique que l'on va partager (ne jamais partager sa clé privé) avec la commande `ssh-copy-id IPNode`.
On va faire cette manipulation une première fois sur le node principal et partager la clé ssh à chaque nodes. Une fois que c'est fait on va se connecter sur chaque node pour faire la même procédure à l'exception que l'on ne partagera la clé publique de chaque node seulement au node principal.

Une fois cela fait, on peut se connecter en ssh au RPI host depuis n'importe quel RPI zero sans avoir à fournir un mot de passe comme le montre la commande ci-dessous :

```
dev@cnat:~ $ ssh pi1
Linux p1 6.1.21+ #1642 Mon Apr  3 17:19:14 BST 2023 armv6l

The programs included with the Debian GNU/Linux system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.
Last login: Fri Dec  1 20:13:44 2023 from 172.19.181.254
pi@p1:~ $
```

D) Installation du service Fail2Ban sur le Raspeberry Pi host

On installe **Fail2ban** grâce à la commande ci-dessous, qui est une application analysant les différents logs de nombreux services, comme SSH, Apache, FTP, et bannissant temporairement les adresses ip à l'origine de motifs de connexions, de requêtes au préalablement indiqués comme suspectes et non désirables.

```
dev@cnat:~ $ sudo apt install fail2ban
dev@cnat:~ $ sudo systemctl start fail2ban
dev@cnat:~ $ sudo systemctl enable fail2ban
```

Dans notre cas, l'objectif est de bannir temporairement les adresses ip à l'origine des tentatives de connexions en ssh au rpi host échouées. Pour ce faire, nous avons modifier le fichier *_/etc/fail2ban/jail.d/defaults-debian.conf*, pour créer une 'prison' pour le service sshd.

```
dev@cnat:~ $ sudo cat /etc/fail2ban/jail.d/defaults-debian.conf
[sshd]
enabled = true
port = 22
logpath = /var/log/auth.log
banaction = iptables
maxretry = 5
bantime = 120
```

Comme indiqué dans la commande ci-dessus, nous avons spécifié de nombreux paramètres pour la prison, comme le port du service sshd, le fichier du log de ce dernier à écouter, la méthode bannissement utilisée qui est iptables et qui permet de bannir seulement un port ou tous les ports de l'adresse ip suspecte. On précise ensuite le nombre d'essais maximum après quoi fail2ban bannit l'adresse ip avec le paramètre *maxretry*. Et enfin, on indique la durée de bannissement de cettre adresse ip, qui est mis à 120s soit 2min.

On active aussi le filtre récidive, qui permet de poursuivre le banissement si l'adresse ip essaie encore de se connecter en ssh sans succès. Pour ce faire, on ajoute les lignes suivante dans le fichier *_/etc/fail2ban/jail.d/defaults-debian.conf*.

```
[recidive]
enabled = true
logpath = /var/log/fail2ban.log
bantime = 1h
```

Ainsi, fail2ban va bannir l'adresse ip pendant 1h si cette dernière récidive.

Pour que fail2ban prenne en compte ces changements, on redémarre le service à l'aide de la commande suivante :

```
dev@cnat:~ $ sudo systemctl reload fail2ban
```

III- Installation de l'application

A) Introduction

Une fois le kit Cluster configuré, et les 4 pi zero accessibles, nous allons installer l'application en utilisant docker swarm. Docker swarm est une fonctionnalité avancée de docker permettant de gérer un cluster de containers et un ensemble de services. Il est composé d'un manager, le RaspberryPi principal dans notre cas, qui s'occupe de gérer les différents workers, et services du docker swarm. Il peut ajouter, supprimer un worker, ajouter, modifier et supprimer un service. Un worker ne possède pas de droits et son seul rôle est d'exécuter les tâches données par le manager, soit un ou plusieurs services. L'avantage de docker swarm est qu'il permet d'assurer la haute disponiblité des services en gérant automatiquement le fail-over et le load-balancing, dans notre cas, l'application doit être disponible en permanance. Dans notre situation, chaque worker est un RaspberryPi zero.

Nous aurons un service web qui sera le front-end et le back-end du site. Il aura 4 réplicas hébergés sur chaque noeud worker RPI zero, ce qui permet au site d'être accesssible si l'un de ces derniers rencontre une erreur. Ensuite, un autre service sera la base de données MySql de l'application, avec 1 réplica qui sera hébergé sur le noeud manager RPI Host.

Avant d'initialiser la swarm, il nous faut tester les différentes images sur le RPI Host et les RPI zero, pour s'assurer de leur bon fonctionnement. On créé donc un nouveau dossier dockerConfig au même niveau que le src dans le RPI Host pour y mettre tous les dockerfiles et les différents fichiers de configuration.

B) Création de l'image pour le serveur web

Pour créer le service web de notre application, nous avons besoin d'une image docker personnalisée.

Nous avons décidé de choisir l'image de base **php:8.2-apache** car cette dernière est compatible avec l'architecture armv6 des quatre RPI zero et qu'elle contient à la fois un serveur web capable de lire et d'exécuter des fichiers html et du js ainsi que des fichiers php grâce au module présent dans Apache. Elle doit contenir l'ensemble du front et back-end de l'application, soit les fichiers du site web.

Ensuite, on crée notre image personnalisée à l'aide d'un fichier **dockerfile**. Ce dockerfile va contenir des instructions pour nous permettre de créer une image standardisée et contenant les différents fichiers dont nous avons besoin sans devoir les intégrer manuellement à nos conteneurs. Chaque conteneur / service créé à partir de cette image personnalisée sera par conséquent identique et contiendra tout le nécéssaire pour l'éxécution de notre site web.

En premier lieu, on se retrouve donc avec un dockerfile nommé dockerfilePHP tel que :

```
FROM php:8.2-apache

COPY ./src/ /var/www/html/

COPY ./dockerConfig/php.ini-development /usr/local/etc/php/

COPY ./dockerConfig/php.ini-production /usr/local/etc/php/
```

```
RUN chown -R www-data:www-data /var/www/html/

RUN docker-php-ext-install mysqli && docker-php-ext-enable mysqli
```

Ce dockerfile va donc créer une image personnalisée à partir de l'image existante **php:8.2-apache**, puis va copier l'intégralité des pages du site dans le répertoire */var/www/html/* du conteneur utilisant cette image, car c'est le répertoire par défaut dans lequel apache va chercher les fichiers web. Ensuite, on copie les différents fichiers de configuration pour les pages en PHP, qui vont par exemple nous permettre d'afficher les erreurs et de charger les extensions PHP, dans notre cas, l'extension MySQLI pour se connecter à un serveur MySQL depuis un script PHP. Puis, on attribue le répertoire de l'application, on donne les droits à l'utilisateur *www-data* qui execute le serveur apache au répertoire de l'application. Enfin, on installe puis on active l'extension PHP de MySQLi.

Après avoir écrit ce dockerfile, on va donc tester son bon fonctionnement sur le RPI Host.

Pour tester cette image personnalisée, on se met sur un des RPI zero et on créé l'image depuis ce dernier pour s'assurer que l'image à bien été créée et est bien compatible avec les RPI zero.

On commence donc par créer l'image qui découle de ce dockerfile avec la commande suivante (cette commande doit être exécutée au même niveau que le répertoire *src*) :

```
docker build -t phpimage -f dockerConfig/dockerfilePHP .
```

Lorsqu'on a créé l'image, on démarre un conteneur avec la commande suivante :

```
docker run --name phpcont -dit phpimage
```

```
dev@cnat:~/Documents/SAE_S5_CalculsDistribues $ docker ps
CONTAINER ID   IMAGE      COMMAND                  CREATED       STATUS       PORTS                  NAMES
73c6c9a98a1e   phpimage   "docker-php-entrypoi…"   3 hours ago   Up 3 hours   0.0.0.0:80->80/tcp   phpcont
```

Après le lancement du container, on s'y connecte en bash pour vérifier que les changements ont bien été pris en compte avec la commande :

```
docker exec -it phpcont bash
```

On peut donc déjà vérifier que les fichiers ont bien été insérés dans le repertoire /var/www/html/ avec la commande `pwd` :

```
pi@p3:~ $ docker exec -it a7cf98001e7a bash
root@a7cf98001e7a:/var/www/html#
```

On vérifie ensuite les droits ont bien été modifiés pour www-data sur les différents répertoires des fichiers du site web :

```
pi@p2:~/SAE_S5_CalculsDistribues $ docker exec -it 74ec609b37ee sh
# ls -lh
total 44K
drwxr-xr-x 1 www-data www-data 4.0K Dec  9 14:06 CSS
drwxr-xr-x 1 www-data www-data 4.0K Dec  9 14:06 Config
drwxr-xr-x 1 www-data www-data 4.0K Dec  9 14:06 JS
drwxr-xr-x 1 www-data www-data 4.0K Dec  9 14:06 LOGS
drwxr-xr-x 1 www-data www-data 4.0K Dec  9 14:06 PHP
drwxr-xr-x 1 www-data www-data 4.0K Dec  9 14:06 PICTURES
drwxr-xr-x 1 www-data www-data 4.0K Dec  9 14:06 SQL
drwxr-xr-x 1 www-data www-data 4.0K Dec  9 14:06 dist
-rw-r--r-- 1 www-data www-data 9.3K Dec  9 14:06 index.html
#
```

Enfin, on vérifie que MySQLi à bien été installé :

```
pi@p2:~/SAE_S5_CalculsDistribues $ docker exec -it 74ec609b37ee sh
# php -r "print_r(get_loaded_extensions());"
Array
(
    [0] => Core
    [1] => date
    [2] => libxml
    [3] => openssl
    [4] => pcre
    [5] => sqlite3
    [6] => zlib
    [7] => ctype
    [8] => curl
    [9] => dom
    [10] => fileinfo
    [11] => filter
    [12] => ftp
    [13] => hash
    [14] => iconv
    [15] => json
    [16] => mbstring
    [17] => SPL
    [18] => session
    [19] => PDO
    [20] => pdo_sqlite
    [21] => standard
    [22] => posix
    [23] => random
    [24] => readline
    [25] => Reflection
    [26] => Phar
    [27] => SimpleXML
    [28] => tokenizer
    [29] => xml
    [30] => xmlreader
    [31] => xmlwriter
    [32] => mysqlnd
    [33] => mysqli
    [34] => sodium
)
#
```

C) Création de l'image pour la base de données

Après avoir créé le service web de notre application et confirmé son bon fonctionnement, nous avons pu passer à la réalisation du service bd de notre application.

Pour créer le service bd de notre application, nous avons aussi besoin d'une image docker personnalisée.

Cette dernière se base sur l'image existante **mysql**, car elle contient déjà un système de gestion de base de données (SGBD). Elle doit contenir la base de données de notre application.

Ensuite, on crée notre image personnalisée à l'aide d'un fichier **dockerfile**.

On créé donc un dockerfile dockerfileMYSQL tel que :

```
FROM clover/mysql:5.7

COPY ./database_script.sql /docker-entrypoint-initdb.d/

RUN chown -R mysql:mysql /docker-entrypoint-initdb.d/ \
    && chmod 755 -R /docker-entrypoint-initdb.d/

WORKDIR /docker-entrypoint-initdb.d/
```

Dans le dockerfile dockerfileMYSQL, il nous a fallu choisir une image mysql compatible avec l'architecture 32 bits armv7 de notre RPI Host. Cependant cette dernière n'existant pas dans la version officielle de l'image mysql, il a fallu choisir une image non officielle. Nous avons donc décidé de choisir l'image clover/mysql. Cette image ayant plus de 10000 pulls, nous en avons déduit qu'elle était fiable et fonctionnelle.

Il nous faut également placer notre script .sql dans un répertoire bien particulier du conteneur qui sera créé à partir de cette image, le répertoire **/docker-entrypoint-initdb.d/**. Ce répertoire permet d'exécuter facilement le script .sql pour créer la base de données après la création du container. Le script quant à lui créé un utilisateur pour manipuler la base de données, deux utilisateurs de base dans l'application et instancie les trois tables SQL.

On peut alors, comme pour notre dockerfilePHP, commencer par build l'image à partir du dockerfileMYSQL :

```
docker build -t sqlimage -f ../../dockerConfig/dockerfileMYSQL .
```

On créé donc une image personnalisée nommée sqlimage à partir du dockerfileMYSQL. Cette commande doit être exécutée a l'endroit ou est situé le script .sql.

Lorsqu'on a build l'image sql on peut démarrer un container avec la commande suivante :

```
docker run --name mysqlcont -dit sqlimage
```

On créé donc un conteneur nommé mysqlcont à partir de l'image sqlimage. Puis on se connecte en sh (bash n'existant pas sur cette image) à ce conteneur :

```
docker exec -it mysqlcont sh
```

Après cela, il faut executer la commande suivant pour se accèder à mysql :

```
mysql -h 127.0.0.1 -u root -proot
```

Pour exécuter le script .sql et donc l'insérer dans le conteneur, il faut effectuer la commande suivante dans le dossier `/docker-entrypoint-initdb.d/` :

```
\. database_script.sql
```

Le script .sql à bien été executé au lancement du conteneur comme le montre l'image suivante :

```
/ # mysql -h 127.0.0.1 -u root -proot
mysql: [Warning] Using a password on the command line interface can be insecure.
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 14
Server version: 8.0.25-0ubuntu0.20.04.1 (Ubuntu)

Copyright (c) 2000, 2021, Oracle and/or its affiliates.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql> show databases
    -> ;
+--------------------+
| Database           |
+--------------------+
| BlitzCalc_DB       |
| information_schema |
| mysql              |
| performance_schema |
| sys                |
+--------------------+
5 rows in set (0.00 sec)

mysql> use BlitzCalc_DB
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A

Database changed
mysql> show tables;
+------------------------+
| Tables_in_BlitzCalc_DB |
+------------------------+
| Logging                |
| Users                  |
| Weak_passwords         |
+------------------------+
3 rows in set (0.01 sec)
```

D) Création du docker swarm

On installe docker s'il ne l'est pas encore. Ensuite, on initialise un docker swarm, et on rajoute 4 workers, où le manager est le RasperryPi principal, et chaque worker est un RaspberryPi zero. L'objectif ensuite est d'avoir 1 stack contenant l'application déployée et 1 autre stack contenant l'application en production. Ce docker swarm contiendra deux services, respectivement un service pour le serveur web et un service pour le serveur sql.

Après avoir vérifié le bon fonctionnement de nos conteneurs et des deux services php et mysql, on crée le swarm :

On commence donc par initialiser le swarm dans le RPI Host avec la commande :

```
docker swarm init --advertise-addr 172.19.181.254
```

On fait ensuite rejoindre les workers (donc les 4 RPI0) avec la commande :

```
docker swarm join --token SWMTKN-1-
2lcnd86mpg7d8x64b1nyspdm6ezhujg6tbdqzxk4ugr3vhio2r-ctzz5dlvv7q0b73bl7kgc5n6n
192.168.0.25:2377
```

Etant donné que l'on veut héberger notre service web uniquement dans les quatre RPI Zero, on doit leur donner à chacun un label spécifique. Lorsqu'un service sera créé, on pourra lui indiquer une contrainte de placement en fonction d'un label. Par exemple, on associe aux quatre RPI zero un label service=web, si on créé un service avec pour contrainte de placement service=web, ce dernier ira placer ses réplicas sur les quatre RPI Zero et pas sur le RPI Host.

On utilise donc la commande suivante pour donner un label aux différents nodes qui vont héberger le service web :

```
docker node update --label-add service=web p2
```

On effectue la même opération pour le service sql avec la commande suivante :

```
docker node update --label-add service=sql cnat
```

E) Création du stack de l'application

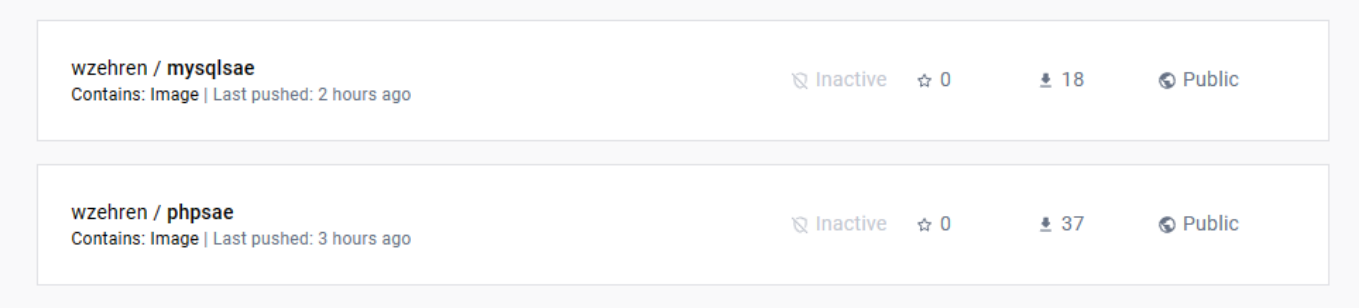
Après avoir créé les deux images et initialisé le swarm, nous allons créer un stack dans le swarm nous permettant de déployer deux services liés aux deux images.

Un stack est un outil permettant de gérer un ensemble de services, sur plusieurs machines, sur plusieurs noeuds, et selon certaines contraintes au sein d'un swarm. Un service correspond à une définiton de taches à exécuter par un ou plusieurs nodes du swarm.

Ainsi, dans notre stack, nous allons créer un service BD qui lancera un conteneur se basant sur l'image sqlimage précédemment créée, ainsi qu'un service web qui lancera un conteneur se basant sur l'image phpimage précédemment créée.

Cependant, comme énoncé précédemment, on souhaite déployer le service web sur quatre noeuds, l'image PHP doit donc également être présente sur ces quatre noeuds. Pour ce faire, nous avons choisi de mettre en ligne nos deux images PHP et MySQL sur la banque d'images docker DockerHub.

Pour mettre nos images en ligne, il a fallu créer un compte docker ainsi qu'un repository docker par images a stocker :



En local, pour mettre nos images en ligne, il a fallu se connecter a notre compte docker hub avec la commande suivante :

```
docker login --username=wzehren --email=williamzehrentravail@gmail.com
```

Et entrer le mot de passe du compte.

Puis il a fallu créer un alias aux images à push de la même manière que le repository avec la commande suivante :

Pour l'image phpimage

```
docker tag phpimage wzehren/phpsae
```

Pour l'image sqlimage

```
docker tag sqlimage wzehren/mysqlsae
```

On peut ensuite les push sur docker hub avec la commande suivante :

Pour l'image phpimage

```
docker push wzehren/phpsae
```

Pour l'image sqlimage

```
docker push wzehren/mysqlsae
```

Après avoir réalisé ces étapes, on peut se placer sur chacun des RPI (zero et host) pour pull, et donc installer leurs images respectives (wzehren/phpsae pour les RPI zero et wzehren/mysqlsae pour le RPI Host) avec la commande suivante :

Pour l'image phpimage

```
docker pull wzehren/phpsae
```

Pour l'image sqlimage

```
docker pull wzehren/mysqlsae
```

Ensuite, on créé un fichier de configuration .yml pour le stack qui va permettre de créer automatiquement les deux services, en spécifiant le nombre de réplicas, leurs redirections de ports ainsi que les contraintes de placement de ces derniers sur les noeuds :

```
version: "3.0"

services:
  serviceweb:
    image: wzehren/phpsae
    ports:
      - "80:80"
    deploy:
      placement:
        constraints: [node.labels.service == web ]
      replicas: 4

  servicebd:
    image: wzehren/mysqlsae
    ports:
      - "3306:3306"
    deploy:
      placement:
        constraints: [node.labels.service == sql ]
      replicas: 1
```

Ensuite, pour déployer le stack sur le swarm, on utilise la commande :

```
docker stack deploy --compose-file docker-compose-stack.yml appli
```

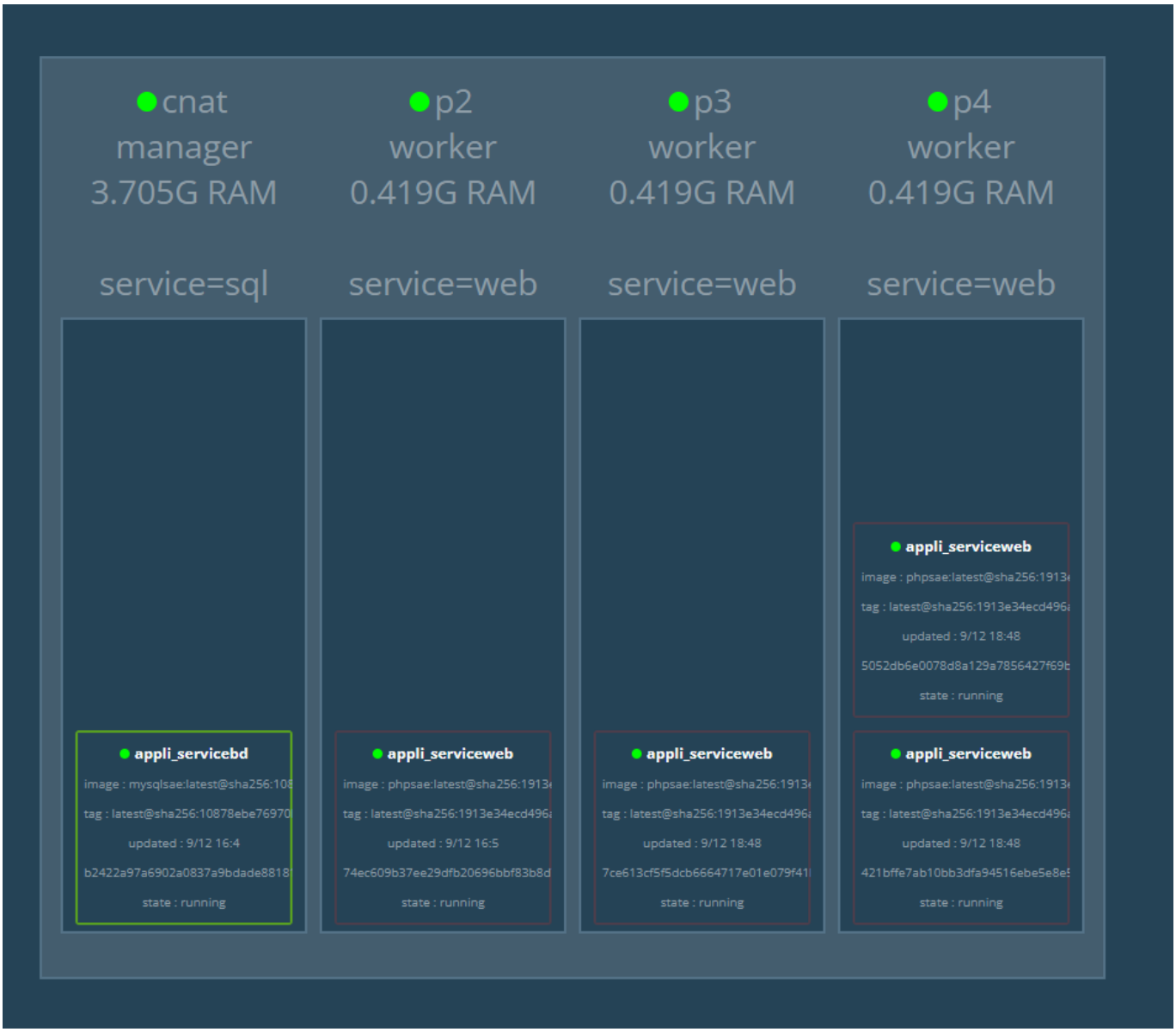
Pour voir que les services sont bien déployés, on peut effectuer la commande :

```
docker service ls
```

Ce qui donne le résultat suivant :

```
dev@cnat:~/Documents/sae/SAE_S5_CalculsDistribues/dockerConfig $ docker service ls
ID             NAME                MODE         REPLICAS   IMAGE                          PORTS
wmvuoldj5cjk   appli_servicebd     replicated   1/1        wzehren/mysqlsae:latest        *:3306->3306/tcp
ij2zhzmlj0br   appli_serviceweb    replicated   4/4        wzehren/phpsae:latest          *:80->80/tcp
```

On voit bien que les deux services ont été créés et ont bien été répliqués. Pour visualiser les différents services de manière plus claire, on créé un conteneur visualizer qui permet d'observer les différents services présents dans le swarm et leur répartition dans les différents nodes.



Une fois l'application déployée, on peut accèder au site web à partir de l'adresse ip suivante :
http://85.170.243.176/

Le site web est donc bien accessible et on peut se connecter à son compte normalement

On peut également accèder au visualizer depuis l'adresse ip suivante : http://85.170.243.176:5000/

(Cette adresse n'est accessible qu'a des fins de test et ne sera plus accessible lors du deploiement du site web)