FONCTIONNEMENT ET MANIPULATION DU RÉSEAU DANS DOCKER

Introduction

Pour que les conteneurs Docker puissent communiquer entre eux mais aussi avec le monde extérieur via la machine hôte, alors une couche de mise en réseau est nécessaire. Cette couche réseau rajoute une partie d'isolation des conteneurs, et permet donc de créer des applications Docker qui fonctionnent ensemble de manière sécurisée.

Docker prend en charge différents types de réseaux qui sont adaptés à certains cas d'utilisation, que nous allons voir à travers ce chapitre.

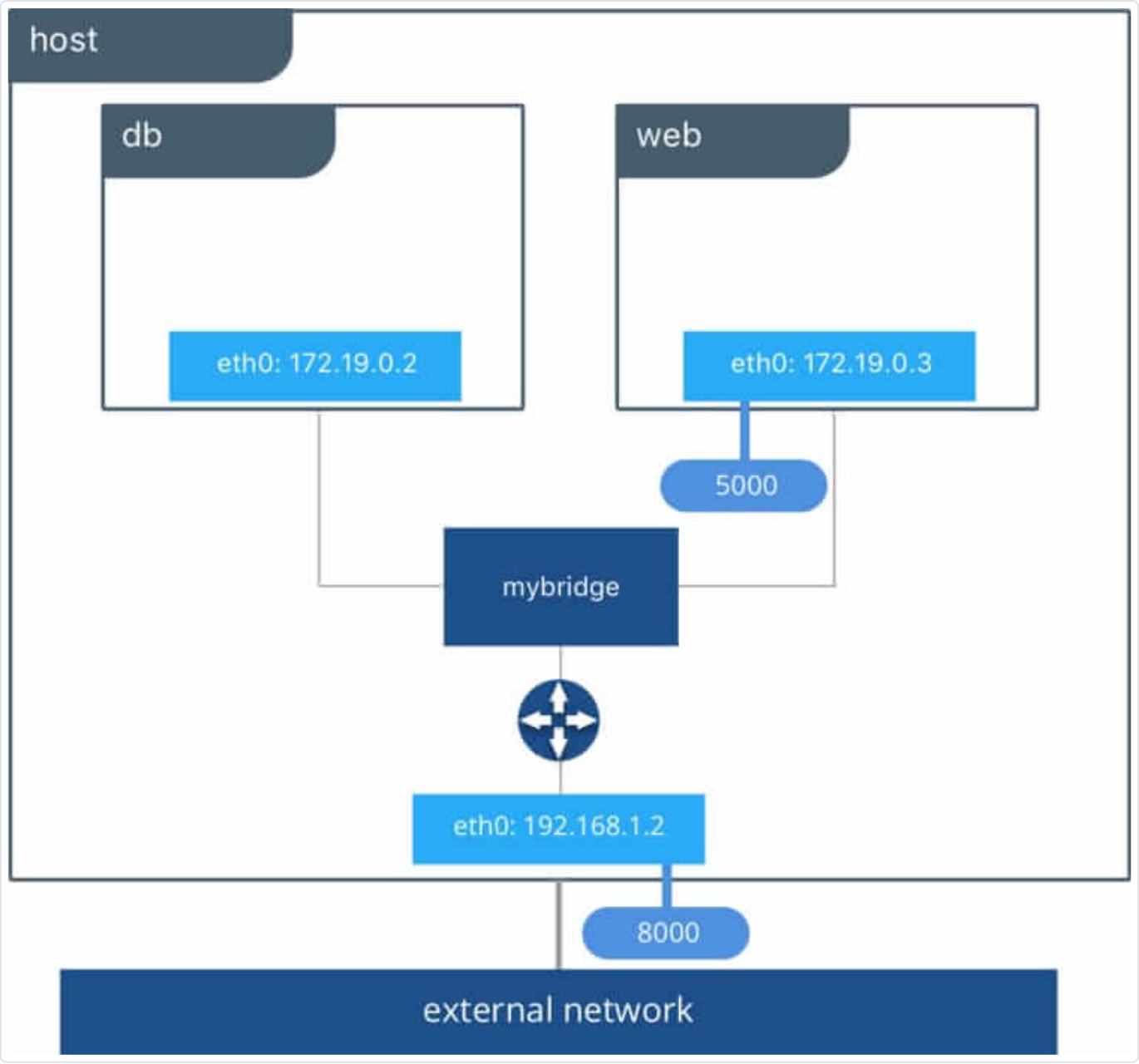
Présentation des différents types de réseau Docker

Le système réseau de Docker utilise des drivers (pilotes). Plusieurs drivers existent et fournissent des fonctionnalités différentes.

Le driver Bridge

Tout d'abord, lorsque vous installez Docker pour la première fois, il crée automatiquement un réseau bridge nommé `bridge` connecté à l'interface réseau `docker0` (consultable avec la commande `ip addr show docker0`). Chaque nouveau conteneur Docker est automatiquement connecté à ce réseau, sauf si un réseau personnalisé est spécifié.

Par ailleurs, **le réseau bridge est le type de réseau le plus couramment utilisé**. Il est limité aux conteneurs d'un hôte unique exécutant le moteur Docker. Les conteneurs qui utilisent ce driver, ne peuvent communiquer qu'entre eux, cependant ils ne sont pas accessibles depuis l'extérieur.



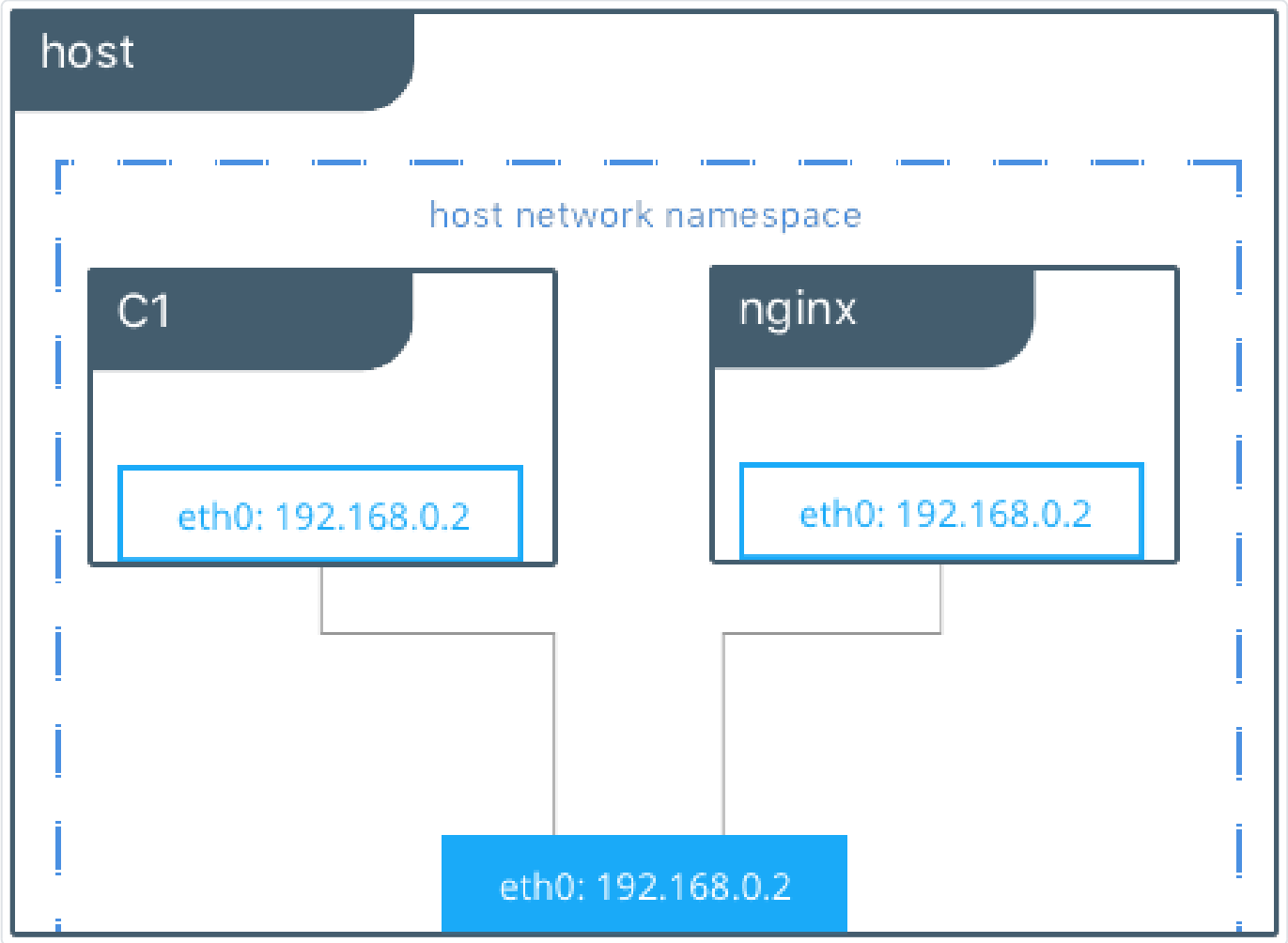
Pour que les conteneurs sur le réseau bridge puissent communiquer ou être accessibles du monde extérieur, vous devez configurer le mappage de port.

Le driver none

C'est le type de réseau idéal, si vous souhaitez interdire toute communication interne et externe avec votre conteneur, car votre conteneur sera dépourvu de toute interface réseau (sauf l'interface loopback).

Le driver host

Ce type de réseau permet aux conteneurs d'utiliser la même interface que l'hôte. Il supprime donc l'isolation réseau entre les conteneurs et seront par défaut accessibles de l'extérieur. De ce fait, il prendra la même IP que votre machine hôte.



Me concernant sur mon pc perso j'utilise le réseau wifi, plus précisément l'interface `wlp3s0`. Voici les informations retournées par la commande `ip add show wlp3s0` depuis ma machine hôte :

```
wlp3s0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group defau
link/ether dc:85:de:ce:04:55 brd ff:ff:ff:ff:ff:ff
inet 192.168.0.11/24 brd 192.168.0.255 scope global dynamic noprefixroute wlp3s0
    valid_lft 54874sec preferred_lft 54874sec
inet6 fe80::335:f1f5:127d:b62c/64 scope link noprefixroute
    valid_lft forever preferred_lft forever
```

Je vais lancer la même commande dans un conteneur basé sur l'image alpine avec un driver de type host :

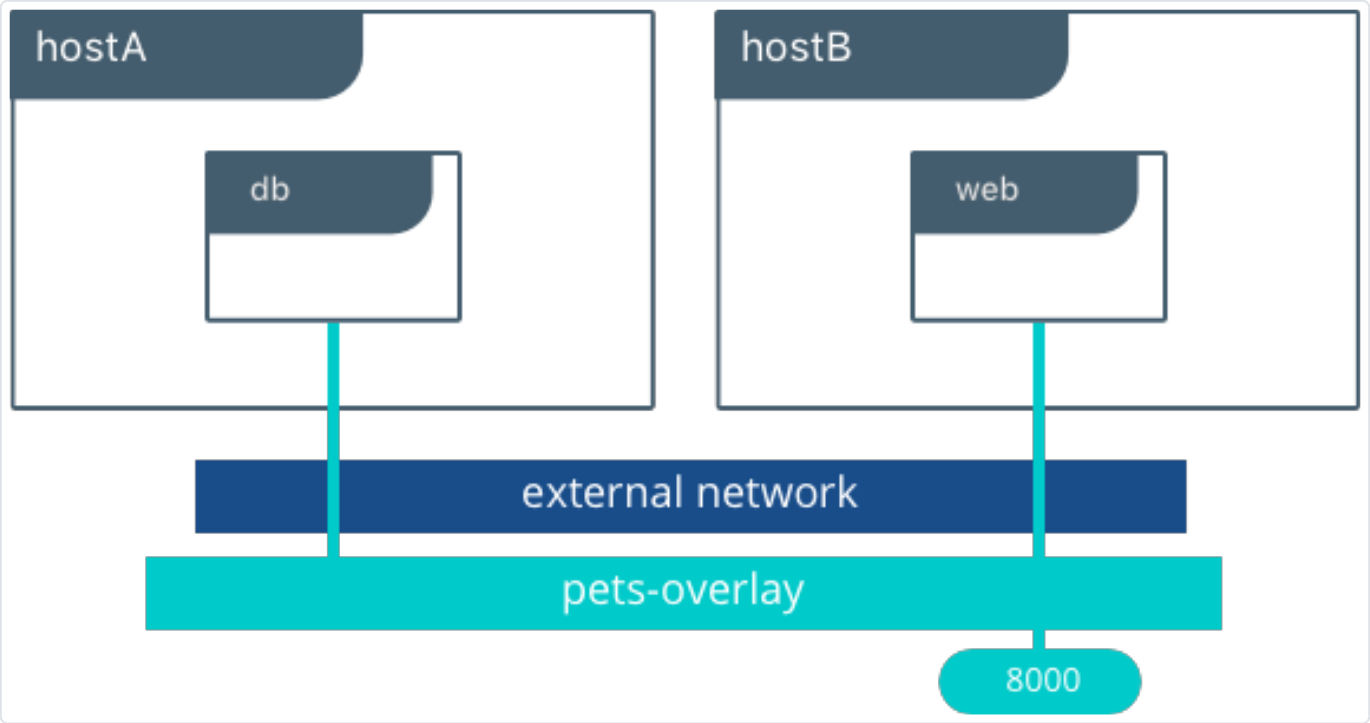
```
docker run -it --rm --network host --name net alpine ip add show wlp3s0
```

Sans surprise, j'obtiens les mêmes informations que sur ma machine hôte (normal car ils utilisent tous les deux l'interface `wlp3s0` grâce au driver host):

```
wlp3s0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP qlen 1000
    link/ether dc:85:de:ce:04:55 brd ff:ff:ff:ff:ff:ff
    inet 192.168.0.11/24 brd 192.168.0.255 scope global dynamic wlp3s0
       valid_lft 54711sec preferred_lft 54711sec
    inet6 fe80::335:f1f5:127d:b62c/64 scope link
       valid_lft forever preferred_lft forever
```

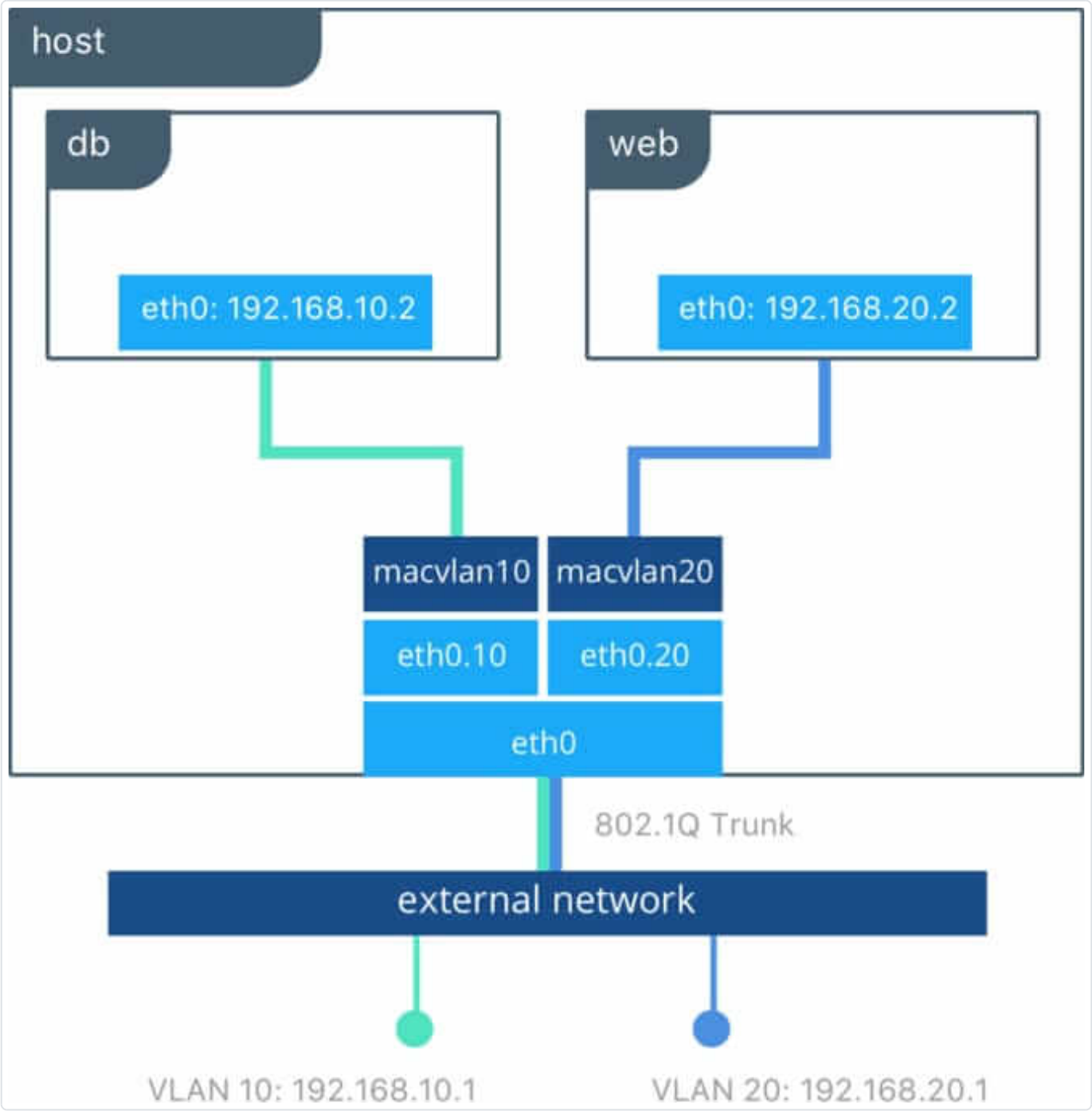
Le driver overlay

Si vous souhaitez une mise en réseau multi-hôte native, vous devez utiliser un driver overlay. Il crée un réseau distribué entre plusieurs hôtes possédant le moteur Docker. Docker gère de manière transparente le routage de chaque paquet vers et depuis le bon hôte et le bon conteneur.



Le driver macvlan

L'utilisation du driver macvlan est parfois le meilleur choix lorsque vous utilisez des applications qui s'attendent à être directement connectées au réseau physique, car le driver Macvlan vous permet d'attribuer une adresse MAC à un conteneur, le faisant apparaître comme un périphérique physique sur votre réseau. Le moteur Docker route le trafic vers les conteneurs en fonction de leurs adresses MAC.



Manipulation du réseau dans Docker

Une fois les présentations finies, il est temps de pratiquer un peu en manipulant le réseau dans Docker.

Créer et récolter des informations d'un réseau Docker

La **commande pour créer un réseau Docker** est la suivante :

```
docker network create --driver <DRIVER TYPE> <NETWORK NAME>
```

Dans cet exemple nous allons **créer un réseau de type bridge** nommé `mon-bridge` :

```
docker network create --driver bridge mon-bridge
```

On va ensuite **lister les réseaux docker** avec la commande suivante :

```
docker network ls
```

Résultat :

```
NETWORK ID          NAME                    DRIVER              SCOPE
58b8305ce041        bridge                  bridge              local
91d7f01dad50        host                    host                local
ccdbdbf708db        mon-bridge              bridge              local
10ee25f56420        monimagedocker_default  bridge              local
6851e9b8e06e        none                    null                local
```

Il est possible de **récolter des informations sur le réseau docker**, comme par exemple la config réseau, en tapant la commande suivante :

```
docker network inspect mon-bridge
```

Résultat :

```
[
    {
        "Name": "mon-bridge",
        "Id": "ccdbdbf708db7fa901b512c8256bc7f700a7914dfaf6e8182bb5183a95f8dd9b",
        ...
        "IPAM": {
            "Driver": "default",
            "Options": {},
            "Config": [
                {
                    "Subnet": "172.21.0.0/16",
```

```
                    "Gateway": "172.21.0.1"
                }
            ]
        },
        ...
        "Labels": {}
    }
]
```

Information

Vous pouvez **surcharger la valeur du Subnet et de la Gateway** en utilisant les options `--subnet` et `--gateway` de la commande `docker network create`, comme suit :

```
docker network create bridge --subnet=172.16.86.0/24 --gateway=172.16.86.1 mon-bridge
```

Pour cet exemple, nous allons **connecter deux conteneurs à notre réseau bridge** créé précédemment :

```
docker run -dit --name alpine1 --network mon-bridge alpine
```

```
docker run -dit --name alpine2 --network mon-bridge alpine
```

Si on inspecte une nouvelle fois notre réseau `mon-bridge`, on verra nos deux nouveaux conteneurs dans les informations retournées :

```
docker network inspect mon-bridge
```

Résultat :

```
[
    {
        "Name": "mon-bridge",
```

```
        "Id": "ccdbdbf708db7fa901b512c8256bc7f700a7914dfaf6e8182bb5183a95f8dd9b",
        ...
        "Containers": {
            "1ab5f1815d98cd492c69a63662419e0eba891c0cadb2cbdd0fb939ab25f94b33": {
                "Name": "alpine1",
                "EndpointID": "5f04963f9ec084df659cfc680b9ec32c44237dc89e96184fe4f231
                "MacAddress": "02:42:ac:15:00:02",
                "IPv4Address": "172.21.0.2/16",
                "IPv6Address": ""
            },
            "a935d2e1ddf76fe49cdb1950653f4a093928020b49ebfea4130ff9d712ffb1d6": {
                "Name": "alpine2",
                "EndpointID": "3e009b56104a1bf9106bc622043a2ee06010b102279e24b4807c7b
                "MacAddress": "02:42:ac:15:00:03",
                "IPv4Address": "172.21.0.3/16",
                "IPv6Address": ""
            }
        },
        ...
    }
]
```

D'après le résultat, on peut s'apercevoir que notre conteneur `alpine1` possède l'adresse IP `172.21.0.2`, et notre conteneur `alpine2` possède l'adresse IP `172.21.0.3`.

Tentons de les faire communiquer ensemble à l'aide de la commande ping :

```
docker exec alpine1 ping -c 1 172.21.0.3
```

Résultat :

```
PING 172.21.0.3 (172.21.0.3): 56 data bytes
64 bytes from 172.21.0.3: seq=0 ttl=64 time=0.101 ms
```

```
docker exec alpine2 ping -c 1 172.21.0.2
```

Résultat :

```
PING 172.21.0.2 (172.21.0.2): 56 data bytes
64 bytes from 172.21.0.2: seq=0 ttl=64 time=0.153 mss
```

Pour information, vous ne pouvez pas créer un network host, car vous utilisez l'interface de votre machine hôte. D'ailleurs si vous tentez de le créer alors vous

recevrez l'erreur suivante :

```
docker network create --driver host mon-host
```

Erreur :

```
Error response from daemon: only one instance of "host" network is allowed
```

On peut ne peut qu'utiliser le driver host mais pas le créer. Dans cet exemple nous allons démarrer un conteneur Apache sur le port 80 de la machine hôte. Du point de vue de la mise en réseau, il s'agit du même niveau d'isolation que si le processus Apache s'exécutait directement sur la machine hôte et non dans un conteneur. Cependant, le processus reste totalement isolé de la machine hôte.

Cette procédure nécessite que le port 80 soit disponible sur la machine hôte :

```
docker run --rm -d --network host --name my_httpd httpd
```

Sans aucun mappage, vous pouvez accédez au serveur Apache en accédant à http://localhost:80/, vous verrez alors le message "It works!".

Depuis votre machine hôte, vous pouvez vérifier quel processus est lié au port 80 à l'aide de la commande `netstat` :

```
sudo netstat -tulpn | grep :80
```

C'est bien le processus httpd qui utilise le port 80 sans avoir recours au mappage de port :

```
tcp        0      0 127.0.0.1:8000          0.0.0.0:*               LISTEN      5084/
tcp6       0      0 :::80                   :::*                    LISTEN      11133
tcp6       0      0 :::8080                 :::*                    LISTEN      3122/
```

Enfin arrêtez le conteneur qui sera supprimé automatiquement car il a été démarré à l'aide de l'option `--rm` :

```
docker container stop my_httpd
```

Supprimer, connecter et connecter un réseau Docker

Avant de supprimer votre réseau docker, il est nécessaire au préalable de supprimer tout conteneur connecté à votre réseau docker, ou sinon il suffit juste de **déconnecter votre conteneur de votre réseau docker sans forcément le supprimer**.

Nous allons choisir la méthode 2, en déconnectant tous les conteneurs utilisant le réseau docker `mon-bridge` :

```
docker network disconnect mon-bridge alpine1
```

```
docker network disconnect mon-bridge alpine2
```

Maintenant, si vous vérifiez les interfaces réseaux de vos conteneurs basées sur l'image alpine, vous ne verrez que l'interface loopback comme pour le driver none :

```
docker exec alpine1 ip a
```

Résultat :

```
lo:  mtu 65536 qdisc noqueue state UNKNOWN qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
        valid_lft forever preferred_lft forever
```

Une fois que vous avez déconnecté tous vos conteneurs du réseau docker `mon-bridge`, vous pouvez alors le supprimer :

```
docker network rm mon-bridge
```

Cependant vos conteneurs se retrouvent maintenant sans interface réseau bridge, il faut donc **reconnecter vos conteneurs au réseau bridge par défaut** pour qu'ils puissent de nouveau communiquer entre eux :

```
docker network connect bridge alpine1
```

```
docker network connect bridge alpine2
```

Vérifiez ensuite si vos conteneurs ont bien reçu la bonne Ip :

```
docker inspect -f '{{.Name}} - {{range .NetworkSettings.Networks}}{{.IPAddress}}{{end
```

Résultat :

```
/alpine2 - 172.17.0.3
/alpine1 - 172.17.0.2
```

Conclusion

Vous pouvez créer autant de réseaux bridge que vous souhaitez, ça reste un bon moyen pour sécuriser la communication entre vos conteneurs, car les conteneurs connectés au bridge1 ne peuvent pas communiquer avec les conteneurs du bridge2, limitant ainsi les communications inutiles.

Concernant le driver overlay, j'essayerais de vous montrer son utilisation dans un autre article car le sujet est très vaste et demande des connaissances sur d'autres sujets que nous n'avons pas eu encore l'occasion de voir, notamment le docker swarm.

Comme d'habitude, voici l'aide-mémoire de ce cours :

```
## Créer un réseau docker
docker network create --driver <DRIVER TYPE> <NETWORK NAME>
```

```
# Lister les réseaux docker
docker network ls

## Supprimer un ou plusieurs réseau(x) docker
docker network rm <NETWORK NAME>

## Récolter des informations sur un réseau docker
docker network inspect <NETWORK NAME>
    -v ou --verbose : mode verbose pour un meilleur diagnostique

## Supprimer tous les réseaux docker non inutilisés
docker network prune
    -f ou --force : forcer la suppression

## Connecter un conteneur à un réseau docker
docker network connect <NETWORK NAME> <CONTAINER NAME>

## Déconnecter un conteneur à réseau docker
docker network disconnect <NETWORK NAME> <CONTAINER NAME>
    -f ou --force : forcer la déconnexion

## Démarrer un conteneur et le connecter à un réseau docker
docker run --network <NETWORK NAME> <IMAGE NAME>
```