## Exercice 8 solution – DNS

Informations

**Évaluation** : formative

**Type de travail** : individuel

**Durée** : 2 heures

**Système d’exploitation** : Linux / Docker

**Environnement** : Docker

Objectifs

Cet exercice a pour objectifs :

* Mieux comprendre le réseau sous Docker.
* Comprendre le DNS sous Docker.

Dans cet exercice, vous allez travailler à mieux comprendre le fonctionnement du réseau sous Docker. La configuration par défaut du réseau est assez facile à travailler avec, mais il est aussi assez facile de changer toutes les options. Vous allez aussi jouer avec le DNS de Docker et voir comment il affecte les paramètres personnalisés et par défauts du réseau. Vous allez aussi voir l’importance du DNS dans Docker, car on ne peut pas se fier aux adresses IP.

#### Partie 1 : Réseau virtuel sous Docker

Chaque conteneur établit une connexion à un réseau privé virtuel «pont» (bridge). Chaque réseau virtuel est routé à travers un mur par feu NAT sur l’IP de l’hôte. Chaque conteneur dans un réseau privé peut se parler entre eux sans utiliser le paramètre -p.

Il est toujours préférable de créer un nouveau réseau virtuel pour chacune de nos applications (app). Par exemple, une «application» contenant Apache/php et MySQL serait dans un réseau «mon\_app\_web».

Par contre, chacun des réseaux virtuels ne peut pas se parler entre eux sans passer par le mur par feu de Docker.

En plus de vous permettre de créer de nouveaux réseaux virtuels, Docker vous permets d’attacher des conteneurs à plusieurs réseaux virtuels (ou aucun), d’ignorer les réseaux virtuels et d’utiliser l’IP de l’hôte (--net=host), d’utiliser différents pilotes réseau Docker pour acquérir de nouvelles fonctionnalités...

Étape 1 : Comprendre le réseau

1. Lancer un conteneur webhost avec l’image nginx:alpine en exposant le port 80 sur votre VM Ubuntu.

docker container run -p 80:80 --name webhost -d nginx:alpine

1. Vous pouvez consulter les ports ouverts sur votre hôte avec la commande suivant.

docker container port webhost

> docker container port webhost

80/tcp -> 0.0.0.0:80

Ce conteneur est relié au réseau virtuel privé de l’engin Docker.

1. Pour connaître l’adresse IP de notre conteneur dans le réseau privé de Docker, vous devez « inspecter » le conteneur. On peut filtrer certaines sorties de Docker avec le paramètre --format qui utilise un gabarit en langage Go.

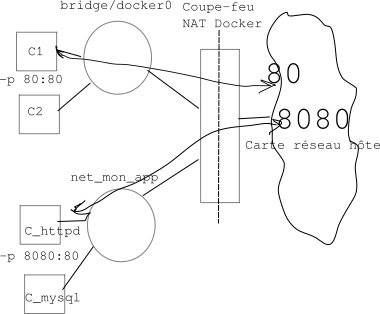
docker container inspect --format '{{ .NetworkSettings.IPAddress }}' webhost

172.17.0.2

Est-ce la même adresse que l’hôte (votre VM Ubuntu) ? Non.

Quelle est l’adresse IP de votre conteneur ? 172.17.0.2

Ce petit schéma résume en gros le fonctionnement du réseau de Docker.



1. Regardez les réseaux que vous avez présentement.

docker network ls

NETWORK ID NAME DRIVER SCOPE

60b1365c69e9 bridge bridge local

b01c46337ebc host host local

ebca4603975a none null local

Il se peut que vous ayez des réseaux qui traînent de vos conteneurs précédents, vous pouvez les enlever avec :

docker network rm *network\_ID*

Le réseau bridge est le réseau par défaut de Docker : NAT avec l’adresse IP de l’hôte.

Le réseau host est relié directement à la carte réseau de l’hôte sans passer par le coupe-feu/NAT de Docker.

Le réseau none n’est pas relié à la carte réseau. C’est l’équivalent de « host only » d’une machine virtuelle.

1. Inspectez le réseau principal : bridge.

docker network inspect bridge

Pouvez retrouver votre conteneur webhost et son adresse? Oui.

[

{

"Name": "bridge",

"Id": "60b1365c69e9813ed0a4bfab1f54c9a1da3da30b423f42ad3898592d3c3be94f",

"Created": "2020-11-21T14:52:13.883777079-05:00",

"Scope": "local",

"Driver": "bridge",

"EnableIPv6": false,

"IPAM": {

"Driver": "default",

"Options": null,

"Config": [

{

"Subnet": "172.17.0.0/16",

"Gateway": "172.17.0.1"

}

]

},

"Internal": false,

"Attachable": false,

"Ingress": false,

"ConfigFrom": {

"Network": ""

},

"ConfigOnly": false,

**"Containers": {**

**"dfb9ea31634eeca55194dcf908ef1714711d06650b1aebae81f64bfba27ee135": {**

**"Name": "webhost",**

**"EndpointID": "3083daa677d6cef35d93580da6864534c558dfaf15c77ff07d35d823cdcd8528",**

**"MacAddress": "02:42:ac:11:00:02",**

**"IPv4Address": "172.17.0.2/16",**

**"IPv6Address": ""**

**}**

**},**

"Options": {

"com.docker.network.bridge.default\_bridge": "true",

"com.docker.network.bridge.enable\_icc": "true",

"com.docker.network.bridge.enable\_ip\_masquerade": "true",

"com.docker.network.bridge.host\_binding\_ipv4": "0.0.0.0",

"com.docker.network.bridge.name": "docker0",

"com.docker.network.driver.mtu": "1500"

},

"Labels": {}

}

]

1. Arrêter votre conteneur et lancer un deuxième conteneur avec la commande ci-dessous.

docker container stop webhost

docker container run --name webhost2 --network host -d nginx:alpine

1. Vérifier l’adresse du conteneur webhost2.

docker container inspect --format '{{ .NetworkSettings.IPAddress }}' webhost2

Quelle adresse a le conteneur webhost2 ? Aucune, mais en réalité il a l’adresse de l’hôte.

Vérifiez que le conteur s’exécute avec la commande sudo ss -tnap | grep 80. Lancez un navigateur dans votre VM et aller à localhost.

Comme vous pouvez constater, le conteneur s’exécute sans passer par le NAT. Ceci peut aider à améliorer les performances, mais naturellement c’est beaucoup moins sécuritaire.

Arrêter et enlever le conteneur webhost2. Relancer webhost.

docker container stop webhost2

docker container rm webhost2

docker container start webhost

1. Nous allons maintenant nous créer un nouveau réseau.

docker network create app\_net2

docker network ls

1. Il est possible de relier un conteneur existant à un réseau. Exécuter la commande suivante pour relier le conteneur webhost à votre nouveau réseau.

docker network connect app\_net2 webhost

Quelles sont les deux adresses IP utilisées par le conteneur webhost ?

docker container inspect webhost

"Networks": {

"app\_net2": {

"IPAMConfig": {},

"Links": null,

"Aliases": [

"dfb9ea31634e"

],

"NetworkID": "32e0a3dfe61669b6de8cfd49789db0314e0c121b824e7f9c9c23b21a94315824",

"EndpointID": "037598892cd01f61b4cbc6078a6562060b2bce2128b71486019d2ba062f54209",

"Gateway": "172.18.0.1",

"IPAddress": "**172.18.0.2**",

"IPPrefixLen": 16,

"IPv6Gateway": "",

"GlobalIPv6Address": "",

"GlobalIPv6PrefixLen": 0,

"MacAddress": "02:42:ac:12:00:02",

"DriverOpts": {}

},

"bridge": {

"IPAMConfig": null,

"Links": null,

"Aliases": null,

"NetworkID": "60b1365c69e9813ed0a4bfab1f54c9a1da3da30b423f42ad3898592d3c3be94f",

"EndpointID": "4ade034ca012859e2bb57ec47adfbecca71cb87bd80c582b8a496c334bf913e2",

"Gateway": "172.17.0.1",

"IPAddress": "**172.17.0.2**",

"IPPrefixLen": 16,

"IPv6Gateway": "",

"GlobalIPv6Address": "",

"GlobalIPv6PrefixLen": 0,

"MacAddress": "02:42:ac:11:00:02",

"DriverOpts": null

}

}

}

#### Partie 2 : DNS dans Docker

Dans cette partie, vous allez voir l’importance du DNS pour les communications entre conteneurs. Vous allez également voir comment le DNS fonctionne par défaut et vous allez utiliser le DNS pour faire un « LoadBalancing » en Round Robin.

Étape 1 : On n’utilise pas d’adresse IP

1. Docker a un daemon DNS intégré. Donc, lorsqu’on travaille avec Docker on évite d’utiliser les adresses IP et on privilégie les noms de conteneur. Consulter le réseau créé dans la partie précédente.

docker network inspect app\_net2

Voyez-vous le conteneur webhost relié à ce réseau ? Oui

[

{

"Name": "app\_net2",

"Id": "32e0a3dfe61669b6de8cfd49789db0314e0c121b824e7f9c9c23b21a94315824",

"Created": "2020-11-21T15:57:21.860146771-05:00",

"Scope": "local",

"Driver": "bridge",

"EnableIPv6": false,

"IPAM": {

"Driver": "default",

"Options": {},

"Config": [

{

"Subnet": "172.18.0.0/16",

"Gateway": "172.18.0.1"

}

]

},

"Internal": false,

"Attachable": false,

"Ingress": false,

"ConfigFrom": {

"Network": ""

},

"ConfigOnly": false,

"Containers": {

"dfb9ea31634eeca55194dcf908ef1714711d06650b1aebae81f64bfba27ee135": {

**"Name": "webhost",**

"EndpointID": "d48273381648475c4146985fa0f8939a58b0953ddcca3eeb52af04587270055d",

"MacAddress": "02:42:ac:12:00:02",

"IPv4Address": "172.18.0.2/16",

"IPv6Address": ""

}

},

"Options": {},

"Labels": {}

}

]

1. Le réseau app\_net2 a reçu automatiquement un DNS qui résout les noms de conteneurs aux adresses IP. Par contre, il est possible de créer des alias pour les noms de conteneurs. Créer un nouveau conteneur relié au réseau app\_net2.

docker container run -d --name webhost2 --network app\_net2 nginx:alpine

Consulter le réseau app\_net2.

Voyez-vous les deux conteneurs reliés à ce réseau ? Oui

[

{

"Name": "app\_net2",

"Id": "32e0a3dfe61669b6de8cfd49789db0314e0c121b824e7f9c9c23b21a94315824",

"Created": "2020-11-21T15:57:21.860146771-05:00",

"Scope": "local",

"Driver": "bridge",

"EnableIPv6": false,

"IPAM": {

"Driver": "default",

"Options": {},

"Config": [

{

"Subnet": "172.18.0.0/16",

"Gateway": "172.18.0.1"

}

]

},

"Internal": false,

"Attachable": false,

"Ingress": false,

"ConfigFrom": {

"Network": ""

},

"ConfigOnly": false,

"Containers": {

"0e95eb960c7856976352659c8d1c183ec9e3770d8236c56387d8a8a3a33c047a": {

**"Name": "webhost2",**

"EndpointID": "aa9c8c90738ca4c3107091826b3c5b87794d0ba0c7817d2c2566295ba2b4a99b",

"MacAddress": "02:42:ac:12:00:03",

"IPv4Address": "172.18.0.3/16",

"IPv6Address": ""

},

"dfb9ea31634eeca55194dcf908ef1714711d06650b1aebae81f64bfba27ee135": {

**"Name": "webhost",**

"EndpointID": "d48273381648475c4146985fa0f8939a58b0953ddcca3eeb52af04587270055d",

"MacAddress": "02:42:ac:12:00:02",

"IPv4Address": "172.18.0.2/16",

"IPv6Address": ""

}

},

"Options": {},

"Labels": {}

}

]

1. Faire un ping du nouveau conteneur à webhost en utilisant seulement le nom.

docker container exec -it webhost2 ping webhost

PING webhost (172.18.0.2): 56 data bytes

64 bytes from 172.18.0.2: seq=0 ttl=64 time=0.274 ms

64 bytes from 172.18.0.2: seq=1 ttl=64 time=0.106 ms

64 bytes from 172.18.0.2: seq=2 ttl=64 time=0.093 ms

64 bytes from 172.18.0.2: seq=3 ttl=64 time=0.096 ms

On peut bien voir que le conteneur webhost2 résout l’adresse IP de webhost.

Arrêter et effacer vos conteneurs et votre réseau.

Étape 2 : DNS Round Robin

1. Nous avons fait un répartiteur de charge avec nginx, mais il est possible de faire quelque chose de semblable, mais moins résilient, avec un DNS (https://en.wikipedia.org/wiki/Round-robin\_DNS).

Je vous donne ici la description de ce qu’on veut faire et vous devez commencer par l’essayer par vous-même. Dans la prochaine étape, je vais vous donner la solution.

Vous allez créer deux conteneurs qui vont répondre au même alias DNS. Vous allez voir que le «Round Robin» n’est pas aussi efficace qu’avec nginx : le même conteneur/serveur peut répondre plusieurs fois de suite.

Commencer par créer un nouveau réseau virtuel dns\_rr.

Créer deux conteneurs elasticsearch:2 reliés au réseau dns\_rr. Nous utilisons l’image elasticsearch, car elle crée automatiquement des noms différents pour ses serveurs.

Dans la création de vos conteneurs, utiliser le paramètre -network-alias search pour que les deux conteneurs soient associés au nom search (-net-alias fonctionne également). Les conteneurs vont répondre à leur nom créer, mais également au nom search.

Après, exécuter la commande Docker alpine nslookup search. avec le paramètre --net pour voir que les deux conteneurs sont associés avec la même entrée DNS.

Ensuite, nous allons répéter plusieurs fois la commande Docker centos curl -s search:9200 (curl est incluse dans l’image centos) avec le paramètre --net. Vous allez voir le champ «name» changer pour chacun des serveurs. Le 9200 est nécessaire, car elasticsearch s’exécute sur le port 9200. Vous allez également voir qu’elasctisearch retourne une courte réponse json facile à interpréter.

Étape 3 : DNS Round Robin solution

1. Création du réseau virtuel.

docker network create dns\_rr

1. Création des conteneurs elasticsearch:2.

docker container run -d --net dns\_rr --net-alias search elasticsearch:2

docker container run -d --net dns\_rr --net-alias search elasticsearch:2

docker container ls

Je n’expose pas de ports, car on veut seulement faire un test de DNS Round Robin dans le réseau virtuel.

1. On vérifie l’entrée DNS search. Attention de ne pas oublier le point à la fin.

docker container run --rm --net dns\_rr alpine nslookup search.

Server: 127.0.0.11

Address: 127.0.0.11:53

Non-authoritative answer:

Non-authoritative answer:

Name: search

Address: 172.18.0.3

Name: search

Address: 172.18.0.2

1. On vérifie le Round Robin DNS.

docker container run --rm --net dns\_rr centos curl -s search:9200

{

"name" : "**Peregrine**",

"cluster\_name" : "elasticsearch",

"cluster\_uuid" : "**UZZSfdrTRT-fH99zzd0RIg**",

"version" : {

"number" : "2.4.6",

"build\_hash" : "5376dca9f70f3abef96a77f4bb22720ace8240fd",

"build\_timestamp" : "2017-07-18T12:17:44Z",

"build\_snapshot" : false,

"lucene\_version" : "5.5.4"

},

"tagline" : "You Know, for Search"

}

docker container run --rm --net dns\_rr centos curl -s search:9200

{

"name" : "**Lianda**",

"cluster\_name" : "elasticsearch",

"cluster\_uuid" : "**eFs-jnMNRJyk5Iw9OSI\_dg**",

"version" : {

"number" : "2.4.6",

"build\_hash" : "5376dca9f70f3abef96a77f4bb22720ace8240fd",

"build\_timestamp" : "2017-07-18T12:17:44Z",

"build\_snapshot" : false,

"lucene\_version" : "5.5.4"

},

"tagline" : "You Know, for Search"

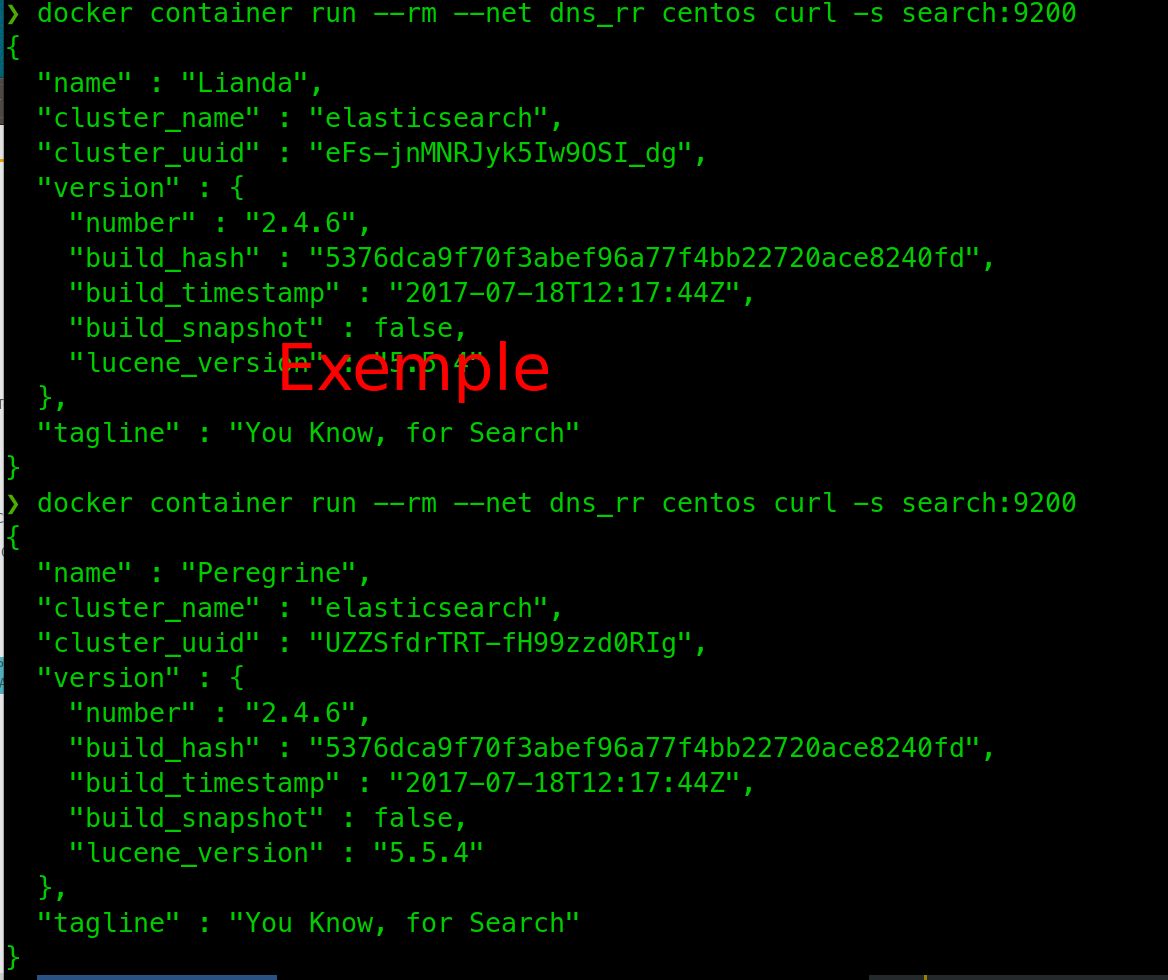
}

Comme le Round Robin DNS n’est pas très efficace, vous allez devoir répéter la commande plusieurs fois avant de voir les deux noms différents.

1. Après avoir pris la capture d’écran pour la vérification, arrêter et effacer vos conteneurs et votre réseau virtuel.

Pour vérification

Remettre une capture d’écran où l’on voit vos deux serveurs elasticsearch répondre en Round Robin.



Compétences développées

|  |  |
| --- | --- |
| **00SJ** – Effectuer le déploiement de serveurs intranet. | **00SJ # 1** – Analyser le projet de déploiement.  **00SJ # 2** – Monter les serveurs intranet.  **00SJ # 3** – Installer les services intranet.  **00SJ # 5** – Participer à la mise en service des serveurs intranet. |

**Note** : les compétences sont développées en partie.

Références

<https://en.wikipedia.org/wiki/Round-robin_DNS>