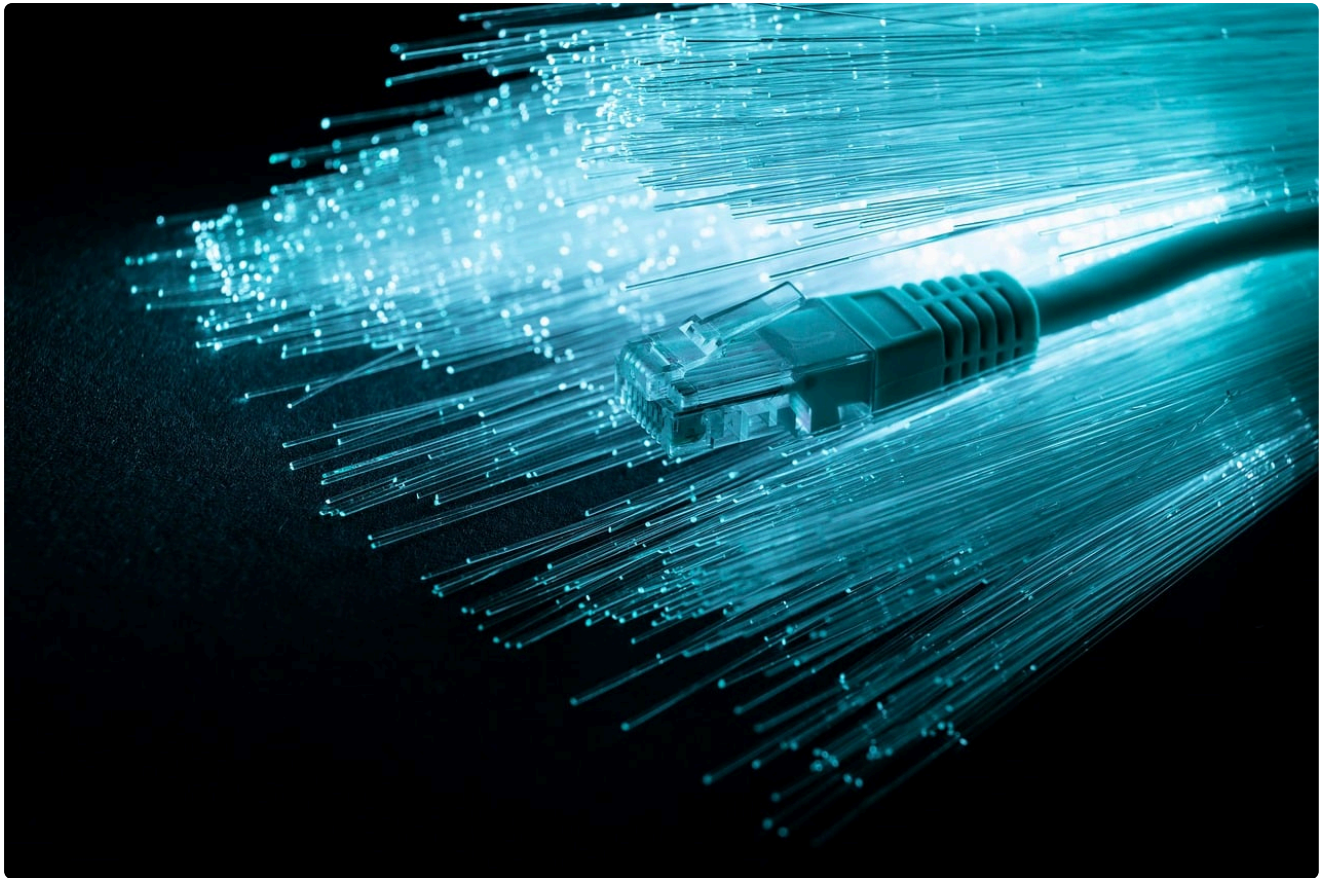




CC_fibre-optique.png

Relier et isoler des conteneurs

Prérequis

Docker Desktop lancé, et les commandes des chapitres précédents en tête. Travaillez dans un terminal Linux ou macOS, pas dans PowerShell.

Ce qu'on cherche à obtenir

Au chapitre 01, vous avez isolé un shell avec `unshare --net` : il n'avait plus aucun réseau et ne pouvait joindre personne. Docker, lui, rebranche quelque chose dès le premier `docker run`. Ce TP sert à voir ce qu'on obtient sans rien demander, ce qu'on obtient en le demandant, et surtout **quoi choisir**.

Vous n'aurez rien à configurer : pas d'adresse à poser, pas de route à écrire. Tout tient dans quelques options de `docker run` et une poignée de commandes `docker network`.

Toujours `--help` d'abord

Une famille de commandes entière apparaît dans ce TP :

```
docker network --help
```

Partie 1 — Ce qu'un conteneur reçoit sans rien demander

Lancez un conteneur ordinaire, et regardez ce que Docker lui a déjà branché.

Commencez par la liste des réseaux que le démon connaît :

```
docker network ls
```

- Combien y en a-t-il ? Reconnaissez-vous certains noms du cours ?

Essayez d'en supprimer un :

```
docker network rm bridge
```

- Que répond Docker ? Qu'en déduisez-vous sur ces trois réseaux ?

Vu de l'intérieur

```
docker run -dit --name solo alpine sh
docker exec solo ip addr
```

- Combien d'interfaces voyez-vous ?
- Quelle adresse porte `eth0` ? Ressemble-t-elle à celle de votre machine ?

Une adresse qui n'existe que là

Cette adresse appartient au réseau privé que Docker gère sur votre machine. Elle n'a aucun sens en dehors : vous ne la retrouverez pas sur votre box, et personne d'autre ne peut s'en servir pour joindre le conteneur.

Sortir vers l'extérieur

```
docker exec solo ping -c 2 1.1.1.1
docker exec solo wget -q0- -T 5 https://search.brave.com/ | head -3
```

- Les deux fonctionnent-elles ?
- Vous n'avez rien configuré. D'après le cours, qui relaie pour le conteneur ?

Validation — `solo` a une adresse, il joint internet, et vous savez dire qui s'en est chargé à votre place.

Partie 2 — Deux conteneurs sur le réseau par défaut

Un conteneur seul ne prouve rien. On en ajoute un second.

```
docker run -dit --name duo alpine sh
```

Récupérez les adresses de `solo` et `duo`. La commande `inspect` du chapitre 02 sait l'extraire toute seule :

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

Depuis `solo`, essayez de joindre `duo` — d'abord par son adresse, puis par son nom :

```
docker exec solo ping -c 2 <adresse-de-duo>
docker exec solo ping -c 2 duo
```

- Laquelle des deux fonctionne ?
- Que répond exactement la seconde ?

Pourquoi c'est un problème

Détruisez `duo`, créez une autre machine nommez `test` (avec la même image de base que `solo`) et enfin recréez `duo` à l'identique, et regardez à nouveau son adresse :

- A-t-il la même adresse qu'avant ?
- Si vous aviez écrit cette adresse dans un fichier de configuration, que se passerait-il maintenant ?

Nettoyez :

```
docker rm -f solo test duo
```

Validation — vous avez joint un conteneur par son adresse, échoué à le joindre par son nom, et constaté que son adresse n'est pas stable.

Partie 3 — Un réseau à soi, et les noms

On refait exactement la même chose sur un réseau qu'on crée soi-même.

```
docker network create atelier
docker network ls
```

- Votre réseau apparaît-il à côté des trois autres ?

Relancez les deux conteneurs, cette fois attachés à `atelier`. L'option qui choisit le réseau d'un conteneur se trouve dans l'aide de `docker run` :

```
docker run --help | grep -i network
```

Créez `solo` et `duo` comme en Partie 2, mais sur le réseau `atelier`. Puis refaites le test par nom :

```
docker exec solo ping -c 2 duo
```

- Cette fois, est-ce que ça passe ?
- Vous n'avez écrit aucune adresse nulle part. Qui a fait la correspondance ?

La stabilité du nom

Détruisez `duo`, recréez-le sur `atelier`, puis recommencez le test :

```
docker rm -f duo
docker run -dit --name duo --network atelier alpine sh
docker exec solo ping -c 2 duo
```

- L'adresse a sans doute changé. Le `ping` fonctionne-t-il quand même ?
- C'est exactement le problème de la Partie 2. Qu'est-ce qui l'a résolu ?

C'est tout l'intérêt du chapitre

En production, on recrée des conteneurs en permanence : mise à jour, redémarrage, remplacement. Si les services se cherchaient par adresse, il faudrait réécrire une configuration à chaque fois. Avec un réseau à soi, on écrit le nom une fois pour toutes.

Un nom de plus

Un conteneur peut répondre à un autre nom que le sien. Lancez un troisième conteneur nommé `trio`, sur `atelier`, en lui donnant l'alias `base` — cherchez l'option dans l'aide :

```
docker run --help | grep -i alias
```

Vérifiez ensuite les deux noms depuis `solo` :

```
docker exec solo ping -c 1 trio
docker exec solo ping -c 1 base
```

- Les deux répondent-elles ? Avec la même adresse ?

Nettoyez les conteneurs, mais **gardez le réseau** :

```
docker rm -f solo duo trio
```

Validation — un `ping` par nom fonctionne sur `atelier`, échouait sur le réseau par défaut, et continue de fonctionner après recréation du conteneur visé.

Partie 4 — Publier un port

Jusqu'ici tout se passe entre conteneurs. On veut maintenant être joint depuis votre navigateur.

Lancez un serveur web **sans rien publier** :

```
docker run -d --name site --network atelier nginx
docker ps
```

- Que contient la colonne `PORTS` ?

Ouvrez <http://localhost:8080>.

- Que se passe-t-il ?

Le serveur tourne-t-il vraiment ? Demandez à un conteneur du même réseau d'aller voir :

```
docker run --rm --network atelier alpine wget -q0- http://site
```

- Obtenez-vous la page d'accueil de Nginx ?
- Le serveur fonctionne donc. Qu'est-ce qui manque pour que votre navigateur y accède ?

Ouvrir la porte

Détruisez le conteneur et relancez-le en publiant son port 80 sur le port 8080 de la machine — c'est l'option vue en cours, déjà croisée au chapitre 02 :

```
docker rm -f site
docker run -d --name site --network atelier -p ???:??? nginx
docker ps
docker port site
```

Rechargez <http://localhost:8080>.

- La page s'affiche-t-elle ?
- Que répond `docker port` ?

 **0.0.0.0 ne veut pas dire « moi seulement »**

Une porte publiée de cette façon est ouverte à **tout le réseau local** : un collègue sur le même Wi-Fi peut ouvrir votre page. C'est le point « publier, c'est ouvrir » du cours.

Un port de la machine ne sert qu'une fois

Essayez de publier un second conteneur sur le même port :

```
docker run -d --name site2 --network atelier -p 8080:80 nginx
```

- Que se passe-t-il ? Lisez le message d'erreur en entier.
- Les deux conteneurs écoutent pourtant tous les deux sur le port 80. Pourquoi celui-là ne pose-t-il aucun problème ?

 **Le nom est déjà pris**

Le conteneur n'a pas démarré, mais il a bien été créé : `docker ps -a` le montre. Son nom reste donc réservé, et le relancer tel quel échouera pour une tout autre raison. C'est le point du chapitre 02 — un conteneur arrêté existe toujours. Supprimez-le avant de réessayer.

Relancez `site2` sur un **autre** port de la machine, et vérifiez que les deux pages répondent.

Fermer la porte au reste du réseau

Lancez un troisième serveur, publié cette fois uniquement pour votre machine — le cours en donne la forme :

```
docker run -d --name site3 --network atelier -p 127.0.0.1:8082:80 nginx
docker port site3
```

- Comparez la sortie avec celle de `docker port site`. Qu'est-ce qui change ?
- Lequel des deux publieriez-vous sur un serveur partagé ?

Nettoyez les trois serveurs :

```
docker rm -f site site2 site3
```

Validation — vous savez expliquer pourquoi la même image était joignable par un autre conteneur mais pas par votre navigateur, et pourquoi un seul conteneur à la fois peut prendre le port 8080.

Partie 5 — Isoler des services

On monte maintenant l'architecture du cours : une porte d'entrée publiée, une base de données que personne ne doit joindre de l'extérieur, et un service intermédiaire qui parle aux deux.

Créez les deux réseaux :

```
docker network create front
docker network create back
```

Lancez les trois conteneurs. Le serveur web est sur `front` et publié, la base est sur `back` et ne publie rien, l'API démarre sur `front` seulement :

```
docker run -d --name web --network front -p 8080:80 nginx
docker run -dit --name db --network back alpine sh
docker run -dit --name api --network front alpine sh
```

Ce qui marche, ce qui ne marche pas

```
docker exec api wget -qO- http://web
docker exec api ping -c 2 db
```

- La première commande passe-t-elle ? Pourquoi ?
- La seconde échoue. Est-ce seulement le nom qui manque ?

Tranchez la question en essayant par l'adresse :

```
docker inspect -f '{{range .NetworkSettings.Networks}}{{.IPAddress}}{{end}}' db
docker exec api ping -c 2 <adresse-de-db>
```

- Le résultat est-il différent ?
- Conclusion : connaître l'adresse suffit-il à franchir la séparation ?

Brancher l'API sur le second réseau

Le cours donne la commande qui attache un conteneur déjà en marche à un réseau. Utilisez-la pour mettre `api` sur `back`, sans le détruire ni le redémarrer. Puis vérifiez :

```
docker exec api ip addr
docker exec api ping -c 2 db
```

- Combien d'interfaces `api` a-t-il maintenant ?
- Le `ping` par nom passe-t-il ?

Ce que l'extérieur voit

```
docker ps
```

- Quels conteneurs ont une entrée dans la colonne `PORTS` ?
- Depuis votre navigateur, par quel moyen pourriez-vous joindre `db` ? Cherchez : il n'y en a aucun, et c'est le but.

Débrancher

```
docker network disconnect back api
docker exec api ping -c 2 db
```

- L'isolation est-elle revenue immédiatement ?

Validation — `api` joint `db` quand il est attaché à `back` et plus du tout quand il en est détaché, et aucune porte de la machine ne mène à `db`.

Partie 6 — `host` et `none`

Les deux réseaux dont on ne se sert pas tous les jours, mais qu'il faut savoir reconnaître.

`none` : aucun réseau

```
docker run --rm --network none alpine ip addr
docker run --rm --network none alpine ping -c 2 1.1.1.1
```

- Quelles interfaces restent ? Comparez avec la Partie 1.
- Ce conteneur peut-il encore joindre quoi que ce soit sur internet ?

`host` : le réseau de la machine

```
docker run --rm --network host alpine ip addr
```

- Combien d'interfaces voyez-vous, par rapport à la Partie 1 ?

⚠ Sur Mac et Windows, ce ne sont pas vos interfaces

Docker tourne dans une VM Linux sur ces systèmes, et c'est le réseau de cette VM que le conteneur partage, pas celui de votre machine. Sur un poste Linux, ce serait vraiment le vôtre. C'est la raison pour laquelle `host` n'est pas un choix portable.

Essayez de publier un port dans ce mode :

```
docker run -d --name hote --network host -p 8080:80 nginx
```

- Docker affiche un avertissement. Que dit-il ?
- D'après le cours, pourquoi publier n'a-t-il aucun sens ici ?

Nettoyez :

```
docker rm -f hote
```

Validation — vous savez dire ce qu'un conteneur en `none` peut encore faire, et pourquoi `-p` est ignoré en `host`.

Partie 7 — Choisir

Rien à taper dans cette partie. Pour chacune des quatre situations, dites quel réseau vous choisiriez — un réseau à vous, le réseau par défaut, `host`, ou `none` — et en une phrase, pourquoi.

1. Une application web et sa base de données, à faire tourner ensemble sur votre poste.
2. Un outil de supervision qui doit découvrir tous les équipements du réseau de l'entreprise.
3. Un convertisseur d'images récupéré sur internet, à qui vous confiez un dossier de photos.
4. Un `alpine` lancé trente secondes pour vérifier le comportement d'une commande.

🔗 La question à se poser en premier

« Ce conteneur a-t-il besoin de parler à quelqu'un, et à qui ? » Si la réponse est « à d'autres conteneurs », c'est un réseau à vous. Si c'est « à personne », c'est `none`. Si c'est « au vrai réseau de la machine, et je sais pourquoi », c'est `host`.

Validation — vos quatre réponses tiennent en une phrase chacune, et vous pouvez les défendre.

Partie 8 — Ranger derrière soi

Ce TP a laissé des conteneurs et trois réseaux.

Faites le ménage vous-même. Vous avez toutes les commandes : celles des chapitres précédents pour les conteneurs, celles de ce TP pour les réseaux.

- Supprimez les conteneurs encore présents.

- Supprimez les réseaux `atelier`, `front` et `back`.
- Vérifiez qu'il ne reste que les trois réseaux du départ.
- **Gardez** les images : elles resserviront au chapitre suivant.

Un réseau occupé ne se supprime pas

Si Docker refuse de supprimer un réseau, c'est qu'un conteneur y est encore attaché — même arrêté. Supprimez le conteneur d'abord, ou détachez-le. La commande de ménage groupée de la famille `network` ne touche, elle, que les réseaux que plus personne n'utilise.

Validation — `docker network ls` ne montre plus que `bridge`, `host` et `none`, et `docker ps -a` plus aucun conteneur de ce TP.

Mémo des commandes

- **Réseaux** : `docker network ls`, `create`, `rm`, `prune`
- **Attacher** : `--network <nom>` au lancement, `docker network connect` / `disconnect` ensuite
- **Nommer** : `--name` donne le nom par lequel on joint le conteneur, `--network-alias` en ajoute un autre
- **Publier** : `-p 8080:80`, `-p 127.0.0.1:8080:80`, puis `docker port`
- **Observer** : `docker ps`, `docker inspect -f`, et depuis le conteneur `ip addr`, `ping`, `wget -q0-`

Pour finir

Vous avez vérifié les deux affirmations centrales du chapitre. Sur le réseau par défaut, deux conteneurs se joignent par une adresse qui change à chaque recreation ; sur un réseau que vous créez, ils se joignent par un nom qui, lui, ne bouge pas. Et un conteneur n'est joignable depuis votre machine que si vous avez explicitement demandé à Docker de lui ouvrir une porte.

La Partie 5 est celle que vous retrouverez partout : on ne protège pas une base de données en oubliant de la publier, on la protège en la mettant sur un réseau où seuls ses clients légitimes sont attachés.

Rien ici n'a demandé de connaissance en réseau, et c'est volontaire. Ce qui se passe réellement sous le capot — comment la machine relaie, comment la porte est installée — est une autre histoire, qu'on pourra ouvrir une autre fois.