**5.1 Starting our first container**

**docker pull ubuntu:**

* This command tells Docker to download the latest Ubuntu image from Docker Hub (or a specified repository).
* Docker checks if the image is already present locally. If not, it fetches the image, layer by layer, from the Docker Hub repository.
* The Ubuntu image is a lightweight version of the Ubuntu operating system (usually a base OS).

**docker run ubuntu:**

* After pulling the image, this command creates and starts a new container based on the Ubuntu image.
* By default, if no additional arguments are provided (such as specifying a command to run), Docker will start the container and immediately exit, as Ubuntu does not have a default long-running process like a web server or shell.
* The container is essentially an isolated environment running Ubuntu.

docker images:

Une image contenant texte, Police, capture d’écran, blanc

Description générée automatiquement

On voit ici les différentes images docker listé qui sont actuellement stocké sur le système en local.

docker ps -a :

Une image contenant texte, reçu, Police, blanc

Description générée automatiquement

The command docker ps -a lists **all** Docker containers on your system, including:

1. **Running containers** (active).
2. **Stopped containers** (inactive or previously run but exited).

The output typically has the following columns:

1. **CONTAINER ID**: A unique ID for each container.
2. **IMAGE**: The image used to create the container (e.g., ubuntu).
3. **COMMAND**: The command the container is executing.
4. **CREATED**: How long ago the container was created.
5. **STATUS**: The current status of the container (e.g., Up for running containers, or exited for stopped ones).
6. **PORTS**: Any ports that are mapped to the host system.
7. **NAMES**: The name given to the container, which is either auto-generated or user-defined.

5.2 docker run -it ubuntu /bin/bash:

En utilisant cette commande, on crée un second container Ubuntu. On arrive directement dedans

Here's a breakdown of **-it** and **/bin/bash** in the command docker run -it ubuntu /bin/bash:

**-it:**

This is a combination of two flags, -i and -t, and it modifies how you interact with the container:

1. **-i (interactive mode)**:
   * Keeps the **STDIN** (standard input) open, allowing you to interact with the container. Without -i, the container would run and then terminate as soon as the default process ends.
2. **-t (pseudo-TTY)**:
   * Allocates a **pseudo-terminal (TTY)**, which provides you with a terminal-like interface. It makes the container’s command-line interface behave like a normal terminal, providing things like command history, auto-completion, etc.

Together, -it gives you an interactive terminal session inside the container, allowing you to execute commands in real-time.

**/bin/bash:**

* **/bin/bash** is the specific command you are telling Docker to run inside the container. In this case, it's starting a Bash shell.
* **Bash** is the default command-line shell in many Linux distributions, including Ubuntu, and /bin/bash refers to the location of the Bash executable.
* This means when the container starts, it opens a **Bash shell**, which allows you to interact with the Ubuntu environment inside the container. You can then execute commands as if you were using a regular terminal.

Without /bin/bash, the container would simply execute the default process for the ubuntu image (which might result in the container starting and stopping immediately). By specifying /bin/bash, you are telling Docker to run a shell and keep it open, allowing for interactive usage.

5.3

On constate que la structures des dossiers sont assez similaire à la base linux standard mais en plus faible nombre.

Pour aller dans un container précis on utilise la commande : docker start <CONTAINER\_ID ou NAME>

docker exec -it <CONTAINER\_ID ou NAME> /bin/bash

Je retrouve bien mon dossier créé auparavant car je reviens bien dans le même container.

Lorsque je crée un nouveau container basé sur la même image, les dossiers créés dans le précédent ne sont pas transmis au nouveau.

Pour supprimer une image de mon ordinateur voici la démarche à suivre :

Une image contenant texte, capture d’écran, Police

Description générée automatiquement

5.4

Si on construit 2 containers avec 2 fois la même image, les containers seront identiques mais bien distinct :

**Key Differences Between Two Containers from the Same Image:**

1. **Unique State**:
   * Each container has its own isolated **file system** and environment, even if they are based on the same image.
   * Any changes you make inside one container (e.g., installing software, modifying files) will not affect the other container.
2. **Container ID**:
   * Each container is assigned a unique **CONTAINER ID** when it is created. This ensures that the two containers are always distinct from each other.
3. **Ephemeral Data**:
   * While the base image is the same, the data or changes you make inside each container can diverge. For example, if you modify files in one container, those changes won't reflect in the other.
4. **Ports & Resources**:
   * Containers can run on different network ports or use different resources. If you expose ports when running the containers (e.g., with -p), they will map to different host ports.
5. **Names**:
   * Even though containers share the same image, Docker assigns a unique **name** (or you can provide one) to each container. For example, if you don’t specify a name, Docker generates one automatically.

Quand on modifie quelque chose à l’intérieur du container, qu’on le quitte et qu’on en crée un autre, les modifications ne sont pas transférées dans le nouveau. Ce sont des containers bien distinct qui n’ont en commun juste l’image de base.

Ce que je pense qu’il manque à ce point-là sont les volumes.

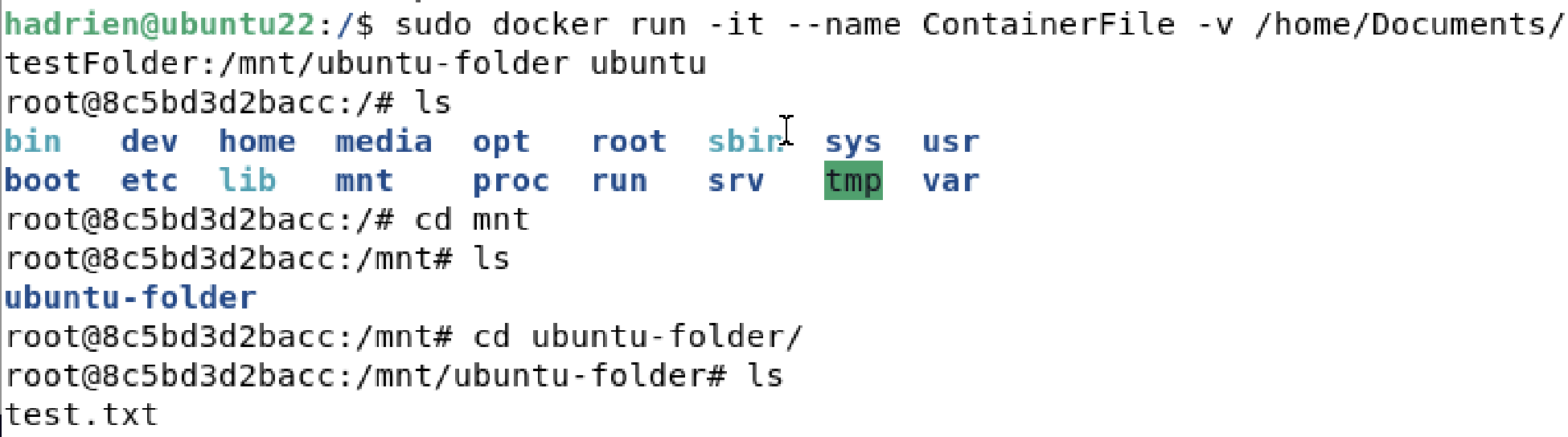
5.6

Une image contenant texte, Police, capture d’écran

Description générée automatiquement

On ne peut pas ajouter un container déjà existant dans le volume. Un container doit être créé à partir du volume en question. Pour transférer les données d’un container existant dans un volume, il faut copier les données et créer un container temporaire relié au volume dans lequel on copie les données à transférer.

5.7

1) 

2) Quand on modifie un fichier existant sur l’ordinateur en local, le fichier se modifie aussi dans le conteneur et prend la même valeur.

Quand on modifie un fichier dans un container, le fichier en local sur la machine se modifie également et prend la même valeur.

3) Une image contenant texte, Police, Page web, capture d’écran

Description générée automatiquement

Une image contenant texte, capture d’écran, Police

Description générée automatiquement

Une image contenant texte, Police, Page web, capture d’écran

Description générée automatiquement

Ici on voit bien que le dossier est persistant dans le volume

4)Le meilleur stockage pour moi est le téléversement dans le volume car à chaque modification dans le volume, le document existant en local ne se modifie pas et surtout que je ne peux pas y avoir directement accès depuis mon pc mais seulement en étant en admin avec docker.

5.8

1) **Typical Output Structure**

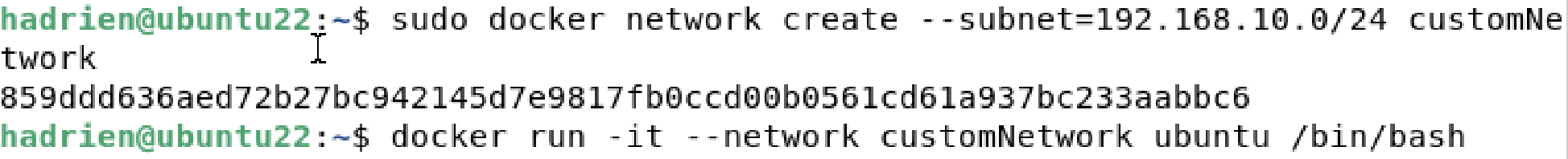
1. **Interface Name**: Each network interface is listed with its name (e.g., eth0, lo, wlan0).
2. **IP Address**: The inet field shows the IPv4 address assigned to the interface. You may also see inet6 for IPv6 addresses.
3. **MAC Address**: The ether field displays the MAC (Media Access Control) address for the interface.
4. **Status Information**: Additional information may include:
   * **RX packets**: The number of packets received.
   * **TX packets**: The number of packets transmitted.
   * **Errors**: Any errors encountered during transmission or reception.
   * **Collisions**: If applicable, the number of collisions that occurred.
5. **Loopback Interface**: The lo interface represents the loopback network, typically assigned the IP address 127.0.0.1.

**Analysis of Interfaces**

* **Active vs. Inactive**: You can tell which interfaces are active based on the presence of an IP address. Interfaces without an IP address are generally inactive.
* **Types of Interfaces**:
  + **Ethernet Interfaces (e.g., eth0, eth1)**: Common for wired connections.
  + **Wireless Interfaces (e.g., wlan0)**: Used for Wi-Fi connections.
  + **Loopback Interface (lo)**: Used for local communications within the host.
* **Configuration**: You can identify whether interfaces are configured correctly by checking if they have valid IP addresses and network masks.
* **Traffic Statistics**: The RX and TX packet counts help you understand the traffic handled by each interface, which can be useful for diagnosing network issues.

**Conclusion**

The ifconfig -a command provides a comprehensive overview of all network interfaces, their statuses, and their configurations. This information is essential for troubleshooting network connectivity issues and for understanding the network configuration of a system.

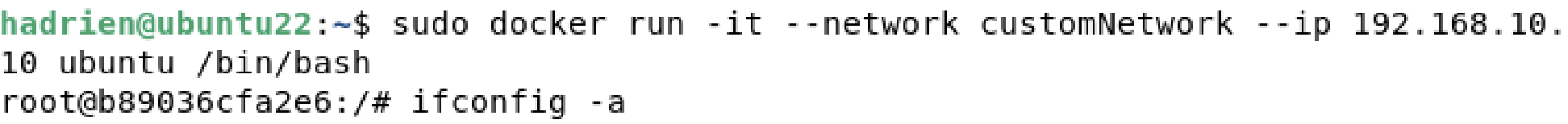
2) Création d’un conteneur avec un réseau custom

Une image contenant texte, capture d’écran, Police, document

Description générée automatiquement

On s’aperçoit ici que l’adresse prend l’identité de du réseau customisé.

Commande pour personnaliser l’IP d’un conteneur :



Une image contenant texte, capture d’écran, Police, document

Description générée automatiquement

3) Pour montrer que la connexion entre les deux conteneurs est établie on fait un ping.

D’abord on ping le premier conteneur au sein du second.

Une image contenant texte, Police, blanc, algèbre

Description générée automatiquement

Puis on ping le deuxième au sein du premier.

Une image contenant texte, Police, capture d’écran, document

Description générée automatiquement

On constate que les données sont envoyées sans perte, les deux conteneurs peuvent donc communiquer.

4) Dans Docker, les conteneurs peuvent être connectés à différents réseaux pour gérer les communications entre eux, leur accès à l'extérieur, ou les isoler.

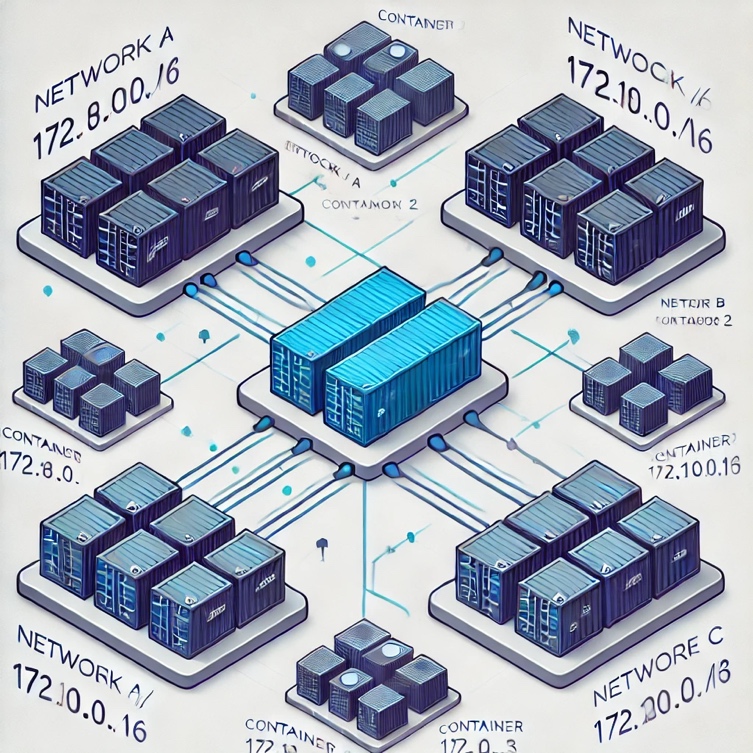
Voici un exemple d'une telle configuration avec trois réseaux et six conteneurs.

**Réseaux et Conteneurs Docker**

| **Réseau** | **Plage IP** | **Conteneurs (Nom/IP)** | **Communication Interne** |
| --- | --- | --- | --- |
| **Network A** | 172.18.0.0/16 | - Container1 (172.18.0.2)  - Container2 (172.18.0.3) | Oui (entre les deux) |
| **Network B** | 172.19.0.0/16 | - Container3 (172.19.0.2)  - Container4 (172.19.0.3) | Oui (entre les deux) |
| **Network C** | 172.20.0.0/16 | - Container5 (172.20.0.2)  - Container6 (172.20.0.3) | Oui (entre les deux) |

**Communication inter-réseaux**

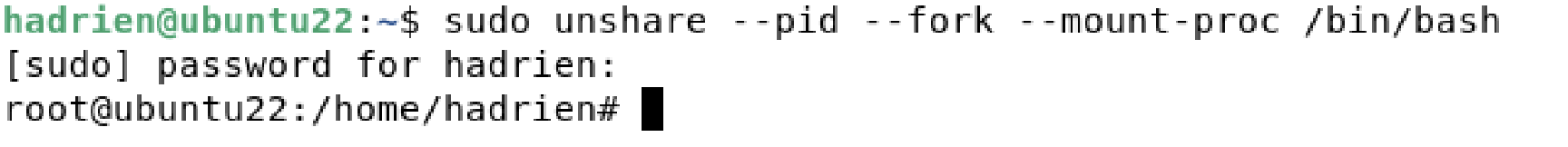
* Les conteneurs sur le **même réseau** peuvent communiquer directement entre eux. Par exemple, Container1 et Container2 peuvent se pinguer.
* Par défaut, les conteneurs sur **des réseaux différents** ne peuvent pas se communiquer (par exemple, Container1 ne peut pas communiquer avec Container3). Mais si vous configurez des routes entre les réseaux ou utilisez le mode **bridge**, il est possible de créer des ponts de communication.



Voici le diagramme montrant la configuration des réseaux Docker avec les conteneurs affectés à différents réseaux, leurs plages d'IP, et les relations de communication. Il illustre les trois réseaux distincts, chacun avec deux conteneurs pouvant communiquer entre eux sur le même réseau, mais isolés des autres réseaux.

5.9

1) sudo unshare --pid --fork --mount-proc /bin/bash



Cette commande crée un namespace de type PID, process namespaces, avec un linux system call unshare().

Cette commande crée un environnement isolé avec un nouvel espace de noms pour les PIDs (processus). Le processus Bash qui est lancé croira qu'il est le processus racine (PID 1), comme s'il se trouvait dans un système fraîchement démarré. Cela permet de tester des processus ou d'exécuter des commandes dans un environnement contrôlé sans affecter le reste du système.

Cet isolement est utile pour le développement, les tests ou la sécurité, par exemple pour exécuter des applications de manière isolée ou simuler un environnement de type conteneur léger.

sudo : Cette commande nécessite les privilèges d'administrateur, car elle modifie des espaces de noms et accède à des fonctionnalités sensibles du système.

unshare : C'est un outil Linux qui permet de dissocier (ou "unshare") certains aspects d'un processus en créant de nouveaux espaces de noms. Un espace de noms est une fonctionnalité d'isolation qui permet de séparer les ressources du système (comme les processus, le réseau, les utilisateurs, etc.) pour un processus donné ou un groupe de processus.

--pid : Cette option crée un nouvel espace de noms PID (Process Identifier). Cela signifie que les processus dans ce nouvel espace de noms auront leur propre arborescence de processus, séparée de l'espace de noms PID du système hôte. Le processus lancé avec cette option (dans ce cas, un nouveau shell) apparaîtra comme ayant le PID 1 dans cet espace de noms.

--fork : Cela indique que le processus doit être immédiatement forké (dupliqué). Le nouveau processus enfant sera exécuté dans l'espace de noms créé, tandis que le processus parent reste inchangé.

--mount-proc : Cette option monte un système de fichiers spécial /proc (qui contient des informations sur les processus en cours) dans le nouvel espace de noms. Sans cela, le nouvel espace de noms de PID ne pourrait pas voir une arborescence correcte des processus, car le système de fichiers /proc est lié à l'espace de noms PID dans lequel il est monté.

/bin/bash : Lance un nouveau shell Bash dans cet espace de noms isolé.

2)

Une image contenant texte, Police, blanc

Description générée automatiquement

On vérifie d’abord le PID dans un namespace courant

Ensuite on vérifie le PID vu par l’hôte

If the first number is 1, that means you are in an isolated PID namespace because in the new PID namespace, your process is treated as the initial process (PID 1).

3)

In a standard Linux system, **PID 1** refers to the **init** process (or systemd in modern systems). This is the first process started by the kernel during boot, and it is responsible for:

* Initializing the system.
* Spawning all other processes.
* Reaping zombie processes (processes that have terminated but whose parent has not yet read their exit status).

Since **PID 1** has special responsibilities, it is always present in a Linux system and is central to system management

Without namespaces, a Linux system would have a single process tree, with only one **PID 1**, making it impossible to run isolated environments (like containers) on the same kernel. All processes would share the same PID space, making process management difficult, as containers could not have their own isolated set of processes starting at PID 1.

The **PID namespace** solves this problem by creating an isolated environment where processes inside the namespace have their own independent PID tree. When a new PID namespace is created (such as when a container is started):

* The process that starts the namespace (the container's entry point) becomes **PID 1** in that namespace.
* This **PID 1** process inside the namespace acts as the "init" process for the container. It may start other processes inside the container, manage them, and reap zombie processes.
* The container's processes can only see and interact with other processes inside the same PID namespace, while being completely unaware of processes running on the host or in other containers.

The **PID namespace** ensures that each container has:

* **A separate process tree**: Every container has its own independent process hierarchy starting from **PID 1**, which gives it the illusion of being an independent operating system.
* **Process isolation**: Processes inside one PID namespace cannot see or interfere with processes in another PID namespace, ensuring secure isolation between containers and the host.
* **Reaping of zombie processes**: Inside each container, **PID 1** is responsible for managing child processes, reaping zombie processes, and ensuring proper resource cleanup. This is important for the stability of the container's processes.

4) **Diagramme de l'arbre PID pour l'hôte et un conteneur avec un namespace de processus isolé**

**Hôte (arbre PID réel) :**

* **PID 1** : systemd (processus principal de l'hôte)
  + **PID 100** : docker-containerd (gère les conteneurs)
    - **PID 500** : dockerd (démon Docker)
      * **PID 1500** : Processus du conteneur (comme vu par l'hôte)

**Conteneur (arbre PID dans le namespace isolé) :**

* **PID 1** : Processus principal du conteneur (vu depuis le conteneur)
  + **PID 2** : Sous-processus A
  + **PID 3** : Sous-processus B

**Explication du diagramme :**

* Le côté gauche montre l’arbre PID de l'hôte, qui commence avec **PID 1** (systemd). Le conteneur apparaît ici avec un PID hôte réel, tel que **1500**.
* Le côté droit montre l’arbre PID isolé à l'intérieur du namespace du conteneur, où le processus principal du conteneur a le PID **1**, suivi de sous-processus avec les PIDs **2** et **3**.

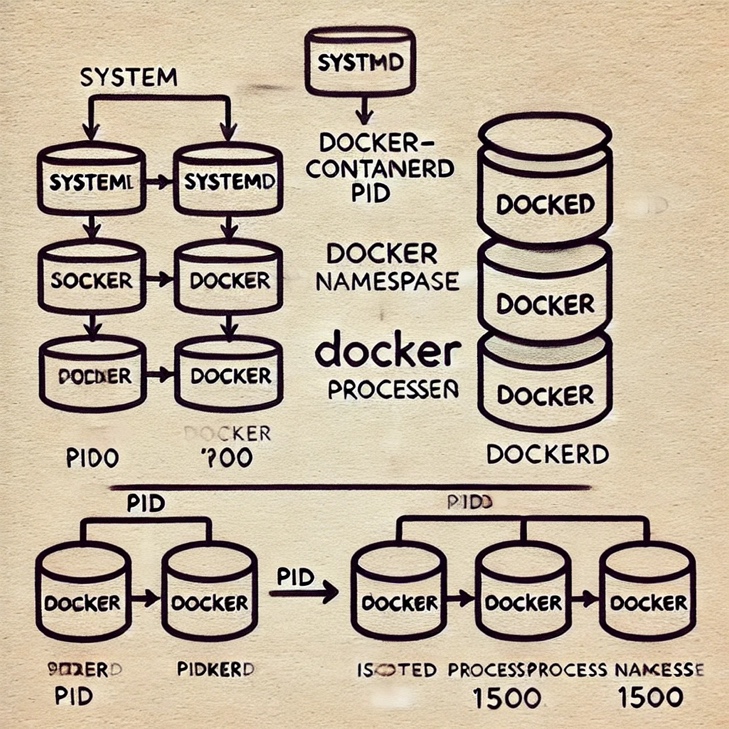


Diagramme simple montrant l'arbre PID du système hôte à gauche et l'arbre PID isolé dans le namespace du conteneur Docker à droite. Le côté gauche représente les PIDs vus depuis l'hôte, tandis que le côté droit montre les PIDs isolés dans le conteneur.

5.10

1)

Une image contenant texte, Police, blanc, typographie

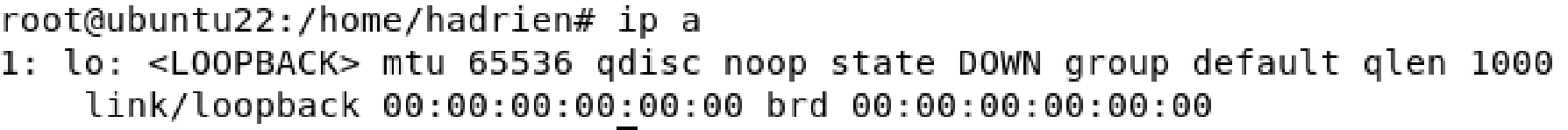
Description générée automatiquement

Ici on crée un namespace avec un call linux unshare qui permet de dissocier certains apsects

--net : This option creates a new **network namespace**. A network namespace provides an isolated network stack, meaning it has its own independent set of network interfaces, routing tables, firewall rules, and so on.

The command sudo unshare --net /bin/bash creates a new network namespace, launching a Bash shell in which you are now operating in a completely isolated network environment. This network isolation means that, initially, you won’t have access to network interfaces or the internet unless you configure it. This is a key feature of Linux namespaces used to create network isolation for containers and other lightweight virtualization techniques.

2)



Si vous travaillez dans un espace de noms réseau (par exemple après avoir utilisé la commande unshare), vous pouvez également utiliser ip a pour vérifier les interfaces disponibles dans cet espace de noms particulier. Vous verrez souvent que seule l'interface de bouclage (lo) est présente par défaut dans un nouvel espace de noms réseau.

3)

Il n’est pas possible de communiquer avec les réseaux extérieurs car la seule interface disponible là où l’on se trouve est l’interface de loopback.

Lorsque vous créez un nouvel espace de noms réseau avec la commande unshare --net, vous isolez le réseau de ce shell par rapport au reste du système. Dans cet espace de noms :

* Vous n'avez **aucune interface réseau** active à part l'interface de bouclage (loopback, lo), qui est locale et permet uniquement la communication à l'intérieur de l'espace de noms lui-même.
* Aucune interface Ethernet (eth0, par exemple) ou d'accès à Internet n'est disponible par défaut.

Dans un nouvel espace de noms réseau, il n'y a pas de **route par défaut** qui permette d'accéder à des réseaux extérieurs, comme Internet ou le réseau local. Cela signifie que même si vous configurez une interface réseau dans cet espace, vous devez aussi configurer des routes pour qu'il puisse savoir comment atteindre d'autres réseaux.

4) faire le schéma

5.11

La liste des différents namespaces avec leur définition et leur utilité :

**1. PID (Process ID) namespace – Espace de noms des processus**

* **Définition** : L'espace de noms des PID isole l'arborescence des processus. Dans un conteneur, le processus initial (souvent l'application exécutée par le conteneur) commence avec le PID 1 à l'intérieur du conteneur, même si sur le système hôte, il peut avoir un autre PID. Cela permet aux conteneurs d'avoir leur propre hiérarchie de processus, sans interférer avec ceux de l'hôte ou d'autres conteneurs.
* **Utilité** : Isoler les processus pour qu'ils soient invisibles en dehors du conteneur.

**2. Network namespace – Espace de noms réseau**

* **Définition** : L'espace de noms réseau offre une isolation des interfaces réseau, des adresses IP, des ports, des routes, et des règles de pare-feu. Chaque conteneur peut avoir ses propres interfaces réseau virtuelles, et ses communications réseau sont séparées du système hôte.
* **Utilité** : Permet à chaque conteneur d'avoir son propre sous-réseau, ses adresses IP, et ses règles de routage, et empêche l'interférence entre les communications réseau des différents conteneurs.

**3. Mount namespace – Espace de noms des montages**

* **Définition** : L'espace de noms de montage isole le système de fichiers vu par un processus. Chaque conteneur peut avoir son propre ensemble de systèmes de fichiers montés, ce qui permet, par exemple, d'avoir un système de fichiers différent de celui de l'hôte ou d'autres conteneurs.
* **Utilité** : Assure que les conteneurs puissent avoir des systèmes de fichiers distincts et isolés, avec leurs propres montages (par exemple, dossiers spécifiques ou volumes partagés), sans affecter l'hôte ou d'autres conteneurs.

**4. UTS (UNIX Time-sharing System) namespace – Espace de noms UTS**

* **Définition** : L'espace de noms UTS permet d'isoler les informations système liées au nom de l'hôte et au domaine. Chaque conteneur peut ainsi avoir un nom d'hôte distinct (vu avec la commande hostname) différent de celui de l'hôte physique.
* **Utilité** : Permet aux conteneurs d'utiliser un nom d'hôte distinct, créant l'illusion qu'ils fonctionnent sur une machine séparée.

**5. IPC (Inter-Process Communication) namespace – Espace de noms IPC**

* **Définition** : L'espace de noms IPC isole les mécanismes de communication inter-processus comme les files de messages, les sémaphores, et les segments de mémoire partagée. Cela signifie que les processus dans un conteneur ne peuvent pas interagir via IPC avec des processus en dehors du conteneur.
* **Utilité** : Garantit que la communication entre les processus à l'intérieur du conteneur reste isolée des autres conteneurs et de l'hôte.

**6. User namespace – Espace de noms utilisateur**

* **Définition** : L'espace de noms utilisateur permet l'isolation des ID utilisateurs (UID) et des ID groupes (GID). Les processus à l'intérieur d'un conteneur peuvent avoir des UID/GID différents de ceux de l'hôte, même si, dans le conteneur, ils apparaissent comme des utilisateurs root (UID 0). Cela permet d'exécuter des processus en tant que root dans un conteneur sans avoir les privilèges root sur l'hôte.
* **Utilité** : Renforce la sécurité en permettant aux conteneurs de fonctionner avec des utilisateurs "root" internes sans donner des privilèges root réels sur l'hôte.

**7. Cgroup (Control Groups) namespace – Espace de noms des groupes de contrôle**

* **Définition** : Les cgroups ne sont pas à proprement parler un namespace, mais ils sont souvent utilisés conjointement. Ils permettent de limiter et de surveiller les ressources (CPU, mémoire, disque, réseau) qu'un groupe de processus peut utiliser. L'espace de noms des cgroups isole la vue des cgroups pour que chaque conteneur voie uniquement les ressources allouées à ses propres processus.
* **Utilité** : Contrôle la quantité de ressources qu'un conteneur peut consommer, pour éviter qu'un conteneur monopolise les ressources de l'hôte.

**8. Time namespace – Espace de noms temporel**

* **Définition** : L'espace de noms temporel, introduit plus récemment, permet d'isoler la vue de l'horloge système pour les processus dans un conteneur. Cela permet de définir une horloge différente pour un conteneur sans affecter l'horloge globale du système hôte.
* **Utilité** : Utile dans des environnements virtualisés où les conteneurs peuvent avoir besoin d'horloges ou de fuseaux horaires spécifiques.

5.12

1)

**1. FROM nginx:1.21-alpine**

* **Explication** : Cette ligne indique à Docker quelle image de base utiliser pour créer le conteneur. Ici, il s'agit de l'image nginx:1.21-alpine.
* **Nginx** est un serveur web populaire, et alpine est une version très légère de Linux. Cela signifie qu'on commence avec un serveur Nginx léger pour construire notre conteneur.

**2. COPY . /usr/share/nginx/html**

* **Explication** : Cette commande copie tous les fichiers du répertoire actuel (indiqué par .) dans le dossier /usr/share/nginx/html à l'intérieur du conteneur.
* **Pourquoi ?** : Nginx utilise ce dossier pour servir les fichiers web (comme des pages HTML). Donc, en copiant les fichiers ici, ils seront accessibles via le serveur Nginx.

**3. EXPOSE 80**

* **Explication** : Cette commande indique à Docker que le conteneur va utiliser le **port 80**.
* **Pourquoi ?** : Le port 80 est le port standard utilisé pour les serveurs web (HTTP). Cela permet à Nginx de répondre aux requêtes web depuis ce port.

**4. CMD ["nginx", "-g", "daemon off;"]**

* **Explication** : Cette commande spécifie ce qui doit être exécuté lorsque le conteneur démarre. Ici, elle indique à Docker de lancer Nginx avec l'option -g "daemon off;".
* **Pourquoi ?** : L'option "daemon off;" empêche Nginx de fonctionner en arrière-plan (daemon mode) afin que Docker puisse garder le conteneur actif et gérer Nginx correctement.

5.13

docker build -t html-hello-world:v1 .

**-t html-hello-world:v1**

* **Explication** : L'option -t (abréviation de "tag") permet de donner un **nom** et une **étiquette** (tag) à l'image que vous construisez.
  + **html-hello-world** est le nom que vous donnez à l'image Docker.
  + **v1** est la version ou le tag que vous assignez à cette image (ici, version 1).
* **Pourquoi ?** : Cela vous permet d'identifier l'image plus facilement plus tard, par exemple pour l'exécuter ou la partager. Vous pourriez avoir plusieurs versions de la même image, d'où l'utilité des tags.

**2. . (le point)**

* **Explication** : Le point . indique le **contexte de construction** (build context), c'est-à-dire le répertoire à partir duquel Docker va chercher les fichiers nécessaires pour créer l'image.
* **Pourquoi ?** : Ici, le point . signifie que Docker doit utiliser le **répertoire courant** (celui où vous exécutez la commande) pour trouver le **Dockerfile** et les fichiers associés.

5.14

docker run -p 8080:80 html-hello-world:v1

**-p 8080:80** redirige le port 8080 de votre machine vers le port 80 dans le conteneur.

Cela permet d'accéder au serveur web Nginx du conteneur en tapant http://localhost:8080 dans un navigateur.

5.15

1)

Une image contenant texte, capture d’écran, Police, information

Description générée automatiquement

L'erreur se produit parce que **le port 8080 sur votre machine hôte** est déjà utilisé par la première instance du conteneur. Vous ne pouvez pas avoir deux conteneurs essayant d'écouter le même port sur l'hôte en même temps.

Pour la corriger on peut utilité un port différent comme par exemple

le port 8081 au lieu de 8080 :

docker run -p 8081:80 html-hello-world:v1

2)

En remplaçant COPY par ADD et en reconstruisant l'image, vous ne verrez probablement aucune différence directe à moins que vous n'ayez des fichiers spéciaux (comme des archives ou des liens externes) dans le répertoire que vous ajoutez.

Dans la plupart des cas, il est recommandé d'utiliser COPY pour sa simplicité, à moins que vous ayez besoin des fonctionnalités supplémentaires de ADD.

3)

On voit une image apparaître sur l’écran avec le mot-clé ADD provenant d’un dossier en cloud.

Avec le mot-clé COPY, cela ne fonctionne pas.

* ADD fonctionne avec les URLs et téléchargera le fichier à partir du lien fourni pour le placer dans l'emplacement spécifié dans le conteneur.
* COPY échoue lorsqu'il est utilisé avec une URL, car il ne prend pas en charge le téléchargement de fichiers externes. Il fonctionne uniquement avec des fichiers locaux situés dans le répertoire de contexte du build Docker.

Vous devez donc d'abord télécharger le fichier manuellement sur votre machine locale avant de pouvoir utiliser COPY pour le copier dans le conteneur.

5.16

1)

1 **. FROM ubuntu:14.04**

* Cette ligne spécifie l'image de base à utiliser pour la création de cette image Docker, ici Ubuntu version 14.04. Cela signifie que l'image Docker sera construite sur un système d'exploitation Ubuntu 14.04.

**2. RUN**

* Les différentes commandes suivantes exécutées sous cette instruction permettent de configurer le système.
* **sed -i ’s/# .\*multiverse$/\1/g’ /etc/apt/sources.list** : Cette commande utilise sed pour modifier le fichier /etc/apt/sources.list. Elle décommente les lignes contenant "multiverse", activant ainsi les dépôts "multiverse", qui contiennent des logiciels non-libres et propriétaires.
* **apt-get update** : Cette commande met à jour la liste des paquets disponibles à partir des dépôts activés.
* **apt-get -y upgrade** : Cela met à jour tous les paquets déjà installés sur l'image à leurs dernières versions disponibles.
* **apt-get install -y build-essential** : Installe le package "build-essential", qui est un ensemble de logiciels et outils essentiels pour compiler du code en C et C++.
* **apt-get install -y software-properties-common** : Installe des utilitaires pour gérer les dépôts logiciels et les PPA (Personal Package Archives).
* **apt-get install -y byobu curl git htop man unzip vim wget** : Cette commande installe des outils courants comme :
  + **byobu** : un gestionnaire de session terminal,
  + **curl** : un outil de transfert de données avec URL,
  + **git** : un système de gestion de versions,
  + **htop** : un moniteur interactif des processus,
  + **man** : pour accéder aux manuels des commandes,
  + **unzip** : un utilitaire pour décompresser des fichiers ZIP,
  + **vim** : un éditeur de texte,
  + **wget** : un outil pour télécharger des fichiers depuis le web.
* **rm -rf /var/lib/apt/lists/** : Cette commande supprime les fichiers de liste de paquets pour économiser de l'espace disque dans l'image Docker.

**3. ADD root/.bashrc /root/.bashrc**

* Cette ligne copie un fichier .bashrc personnalisé depuis votre machine locale (situé dans le répertoire root) vers le répertoire /root/.bashrc dans l'image Docker. Ce fichier configure des options et des alias pour le shell Bash.

**4. ADD root/.gitconfig /root/.gitconfig**

* Cela copie le fichier de configuration Git personnalisé .gitconfig depuis votre machine locale vers /root/.gitconfig dans le conteneur.

**5. ADD root/.scripts /root/.scripts**

* Cette commande copie le dossier scripts de votre machine locale vers /root/.scripts. Ce dossier pourrait contenir des scripts bash ou d'autres scripts utilitaires à exécuter dans le conteneur.

**6. ENV HOME /root (marqué par ???)**

* Cette instruction définit la variable d'environnement HOME, qui pointe vers le répertoire /root. Cela indique au système que le répertoire /root est le répertoire personnel de l'utilisateur root.

**7. WORKDIR /root (marqué par ???)**

* Cette instruction change le répertoire de travail par défaut du conteneur en /root. Cela signifie que toute commande exécutée après cette ligne se fera à partir de ce répertoire.

**8. CMD ["bash"]**

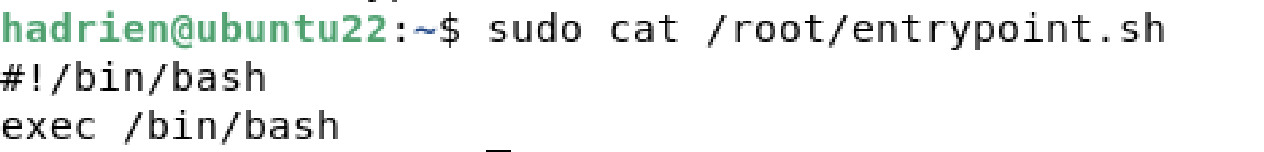
* Définit la commande par défaut à exécuter lorsque le conteneur démarre. Ici, le conteneur ouvrira un shell Bash.

2)

* **CMD ["bash"]** : Cette ligne signifie que par défaut, lorsque le conteneur démarre, il ouvre un shell Bash. Cependant, l'utilisateur peut remplacer cette commande par une autre au moment de l'exécution (par exemple, en passant une commande différente via docker run).
* **ENTRYPOINT** : Contrairement à CMD, ENTRYPOINT définit un point d'entrée fixe pour le conteneur, que l'utilisateur ne peut pas remplacer facilement. C'est utile pour forcer un comportement particulier, comme l'exécution automatique d'un script.

Une image contenant texte, capture d’écran, Police, algèbre

Description générée automatiquement



5.17

1)

Une image contenant texte, capture d’écran, Police, nombre

Description générée automatiquement

Historique pour html-hello-world :v1 :

Une image contenant texte, capture d’écran, Police, document

Description générée automatiquement

Historique pour ubuntu-junia :v1 :

Une image contenant texte, capture d’écran, Police, document

Description générée automatiquement

Le Dockerfile est un ensemble d'instructions pour construire une image Docker, tandis que l'historique d'une image montre chaque commande exécutée lors de sa création, étape par étape.

Différences entre l'historique et le Dockerfile :

1. Historique :
   * L'historique d'une image Docker (docker history <image\_name>) montre chaque commande exécutée, y compris celles qui peuvent avoir été exécutées en dehors du Dockerfile (par exemple, via des sessions interactives).
   * Il inclut des couches supplémentaires créées par chaque commande RUN ou d'autres opérations. On y voit aussi des métadonnées comme la taille de la couche et le moment où elle a été créée.
2. Dockerfile :
   * Le Dockerfile est une liste d'instructions déclaratives pour définir ce qui doit se produire lors de la construction de l'image.
   * Il reflète la structure prévue, mais ne montre pas les couches intermédiaires ou les résultats concrets des opérations, comme la taille de l'image finale.

En résumé, l'historique Docker montre les commandes réellement exécutées, tandis que le Dockerfile représente les instructions prévues pour construire l'image.

2)

Dockerfile édité avec la mise à jour apk :

Une image contenant texte, blanc, Police, reçu

Description générée automatiquement

Voici l’historique de l’image html-hello-world :v8 :

Une image contenant texte, capture d’écran, Police, document

Description générée automatiquement

On voit la ligne de commande qui exécute l’upgrade d’apk.

5.18

1)

Une image contenant texte, Police, capture d’écran

Description générée automatiquement

2)

Intérieur du dossier app :



**Explication des instructions**

1. **FROM node:alpine** :
   * Utilise l'image **Node.js** basée sur **Alpine Linux**, qui est une distribution très légère, idéale pour des images Docker plus petites.
2. **ENV APPDIR /APP** :
   * Définit une variable d'environnement **APPDIR** qui pointe vers le répertoire /APP. Cela permet de réutiliser cette variable plus tard sans répéter le chemin.
3. **WORKDIR ${APPDIR}** :
   * Définit le répertoire de travail à **/APP**. Toute commande RUN, CMD, ou COPY sera exécutée dans ce répertoire.
4. **COPY package.json package.json** :
   * Copie le fichier **package.json** de votre projet local dans le conteneur, ce qui est nécessaire pour installer les dépendances Node.js avec **npm**.
5. **RUN apk add --no-cache --update curl && rm -rf /var/cache/apk && npm install** :
   * Installe **curl** dans l'image Alpine et nettoie le cache des paquets pour garder l'image aussi légère que possible.
   * Exécute ensuite **npm install** pour installer les dépendances spécifiées dans **package.json**.
6. **COPY app.js app.js et COPY index.html index.html** :
   * Copie vos fichiers d'application (**app.js**) et votre fichier HTML (**index.html**) dans le conteneur.
7. **CMD ["node", "app.js", "3000"]** :
   * Définit la commande par défaut exécutée lorsque le conteneur démarre, ici pour lancer votre serveur Node.js avec le fichier **app.js** en écoutant sur le port 3000.
8. **EXPOSE 3000** :
   * **Expose** le port **3000**, indiquant que l'application écoute sur ce port. Cela permet à Docker de comprendre à quel port le conteneur est censé recevoir des connexions.

**Que fait ce Dockerfile ?**

Ce Dockerfile crée une image Docker pour une application Node.js légère basée sur Alpine Linux. Il fait les actions suivantes :

* Il installe les dépendances Node.js spécifiées dans le fichier **package.json**.
* Il copie votre code d'application (fichiers **app.js** et **index.html**) dans l'image.
* Il lance l'application Node.js à l'intérieur du conteneur et expose le port **3000** pour les connexions réseau externes.

Ce Dockerfile est conçu pour fournir une application Node.js prête à être déployée dans un environnement Docker.

5.19

1)

Une image contenant texte, capture d’écran, Police

Description générée automatiquement

La commande curl 0.0.0.0:PORT/text est utilisée pour effectuer une requête HTTP vers le serveur local qui fonctionne sur le port spécifié. Voici ce que fait cette commande en détail :

1. **curl** :
   * C'est un outil en ligne de commande qui permet d'effectuer des requêtes HTTP et d'afficher les réponses. Il est couramment utilisé pour tester et déboguer des serveurs web ou des APIs.
2. **0.0.0.0** :
   * Cette adresse IP spéciale signifie "toutes les interfaces réseau". Lorsque le serveur écoute sur 0.0.0.0, cela signifie qu'il accepte les connexions sur toutes les interfaces réseau disponibles, y compris localhost(127.0.0.1) et toutes les adresses IP externes.
3. **PORT** :
   * Vous devez remplacer PORT par le numéro de port sur lequel votre serveur fonctionne. Par exemple, si votre serveur fonctionne sur le port 3000, la commande serait curl 0.0.0.0:3000/text.
4. **/text** :
   * C'est le chemin sur votre serveur web. Il peut correspondre à un point de terminaison ou une ressource spécifique fournie par votre serveur. Si votre serveur est configuré pour répondre aux requêtes sur /text, vous recevrez une réponse (habituellement du texte ou des données JSON).

2)

Une image contenant texte, capture d’écran, document, Police

Description générée automatiquement