Containers

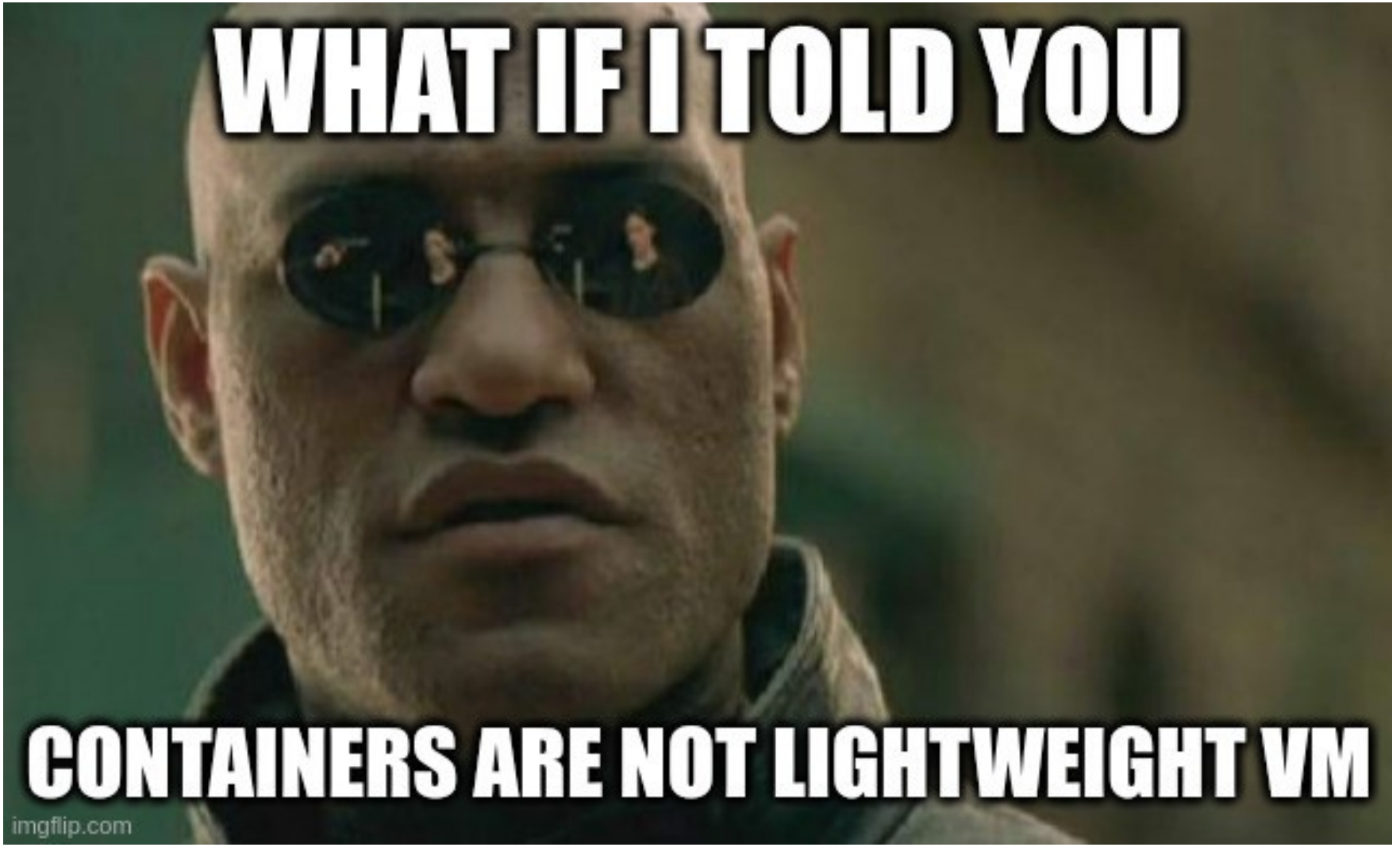
Sonia Zorba

Serate a tema – 11 luglio 2024

Lug Trieste

Scaletta

- Container VS Virtual Machine
- Chroot, cgroup e namespaces
- Docker
- Podman
- LXC
- systemd-nspawn


WHAT IF I TOLD YOU
CONTAINERS ARE NOT LIGHTWEIGHT VM
imgflip.com

Container vs VM

VM:

- In una VM gira un’intero SO
- C’è emulazione dell’hardware

Container:

- Condivide il kernel con l’host
- Non c’è emulazione dell’hardware

C’era una volta chroot...

Nel 1979 (in Unix V7) nasce il tool **`chroot`**, che modifica la root directory di un dato processo (e dei suoi child).

Il concetto verrà poi migliorato da FreeBSD nel 2000 con l’introduzione del comando **`jail`**.

Esempio chroot

Avviamo una shell busybox in chroot:

```
$ mkdir scatola
$ cp /usr/bin/busybox scatola
$ sudo chroot scatola /busybox ash
```

Il comando `ps` non ci mostra nulla perché manca la cartella `/proc`, ma se conosciamo il PID di un processo fuori dalla chroot possiamo comunque terminarlo.

Questo dimostra che l'isolamento è solo a livello di filesystem.

Cgroups

Meccanismo del kernel per controllare le risorse utilizzate dai processi.

I processi vengono organizzati in una struttura gerarchica che viene montata in un'apposita cartella:

```
$ mount | grep cgroup

cgroup2 on /sys/fs/cgroup type cgroup2
(rw,nosuid,nodev,noexec,relatime)
```

Cgroups

I controller sono i componenti di cgroup che si occupano di controllare la distribuzione delle risorse ai processi.

La lista dei controller disponibili si trova nel seguente file:

```
cat /sys/fs/cgroup/cgroup.controllers
```

Esempi:

- limite sull'uso della CPU o della RAM
- limite al numero di fork possibili
- limite al numero di operazioni I/O al secondo

Esempio cgroup

Creiamo un cgroup aggiungendo una cartella:

```
# cd /sys/fs/cgroup
# mkdir pippo
```

Migriamo il processo con PID 20835 nel cgroup:

```
# echo 20835 > /sys/fs/cgroup/pippo/cgroup.procs
```

Limitiamo l'uso della CPU al 50%:

```
# echo "50000 100000" > /sys/fs/cgroup/pippo/cpu.max
```

Namespaces

Meccanismo del kernel che modifica le risorse che un insieme di processi possono vedere, isolandoli così dagli altri namespace e dal resto del sistema.

Esempi:

- PID
- Network
- User ID
- Mount
- Cgroup
- Time

Namespaces

Posso creare un namespace con:

1) comando **`unshare`**
2) comando **`ip netns`** (per network namespace)

Posso usare **`lsns`** per visualizzare i namespace.

Le informazioni sui namespace stanno in **`/proc/<PID>/ns`**

Namespace – User ID

```
user@hostname$ id
uid=1000(user)

user@hostname$ unshare -U bash
nobody@hostname$ id
uid=65534(nobody)

user@hostname$ unshare -U --map-root-user bash
root@hostname$ id
uid=0(root)
root@hostname$ ls /root
Permission denied
```

Network namespace

host

```
# (1) ip netns add pippo
# (4) ip netns exec pippo ip link set lo up
# (5) ip link add veth0 type veth peer veth1 netns pippo
# (6) ip link set veth0 up
# (10) ping6 <IP-CONTAINER>
# (11) ip netns del pippo
```

man veth

container

```
# (2) unshare --net=/var/run/netns/pippo bash
# (3) ip addr
# (7) ip link set veth1 up
# (8) ip addr
# (9) ping6 <IP-HOST>
```

PID namespace

```
# unshare --pid --fork --mount-proc bash
```

Per creare un PID namespace utilizzabile non basta **`--pid`**, ma servono altri 2 flag:

1) **`--mount-proc`**: crea e monta un nuovo proc filesystem (questo implica che abbiamo anche un nuovo mount namespace)
2) **`--fork`**: il programma (bash) non viene eseguito direttamente ma diventa un child di unshare, in modo da creare un nuovo albero dei processi

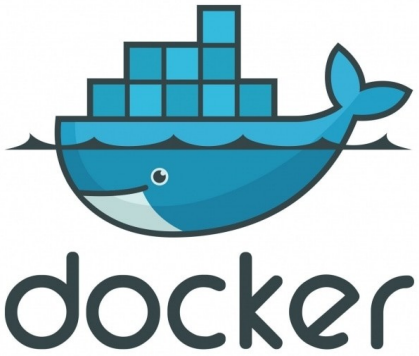
Namespace – (boot)time

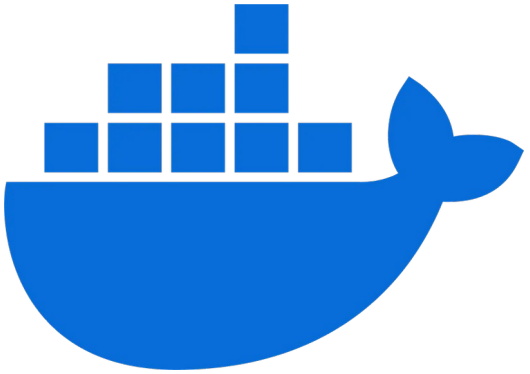
```
# uptime -s
2024-05-22 18:44:42

# unshare -T --boottime 100000000 bash
# uptime -s
2021-03-22 07:58:02
```

Usato per mantenere valori di uptime consistenti quando si effettuano migrazioni e operazioni di checkpoint/restore

Docker

Docker





- Docker Inc: azienda USA
- Docker Engine:
  - Prima release nel 2013
  - Licenza Apache 2.0
- Docker Hub: registry, repository di immagini https://hub.docker.com

Application vs System Containers

Docker

Ha bisogno di un demone (dockerd):

```
# systemctl start docker
```

Facciamo pull di un'immagine da Docker Hub:

```
$ docker pull alpine
```

Questo scarica file binari e librerie della distro Alpine e li salva nella cartella Docker (es: `/var/lib/docker`).

Possiamo ispezionare l'immagine con il tool **`dive`**.

Docker

Visualizzo la lista delle immagini alpine:

```
$ docker image ls | grep alpine
```

Avvio un container e apro una shell al suo interno:

```
$ docker run -it alpine
```

Verifico i container attivi:

```
$ docker ps
```

Eseguo un comando nel container:

```
$ docker exec <CONTAINER-ID> <COMANDO>
```

Volumi

I container Docker sono stateless, ma possiamo rendere persistenti certe cartelle creando dei volumi:

```
$ docker run -it -v ./volume:/root alpine
```

Il formato è <cartella-host>:<cartella-container>

Immagini Docker

Costruite sulla base di un file detto Dockerfile.

```
FROM alpine
RUN apk add fortune
ADD linus linus.dat /usr/share/fortune/
CMD ["/usr/bin/fortune", "linus"]
```

```
$ docker build . -t rantune
```

Comunicazione tra containers

Docker compose

```
services:
  db:
    image: postgres
    environment:
      POSTGRES_PASSWORD: lug
    ports:
      - 5432:5432
  adminer:
    image: adminer
    ports:
      - 8080:8080
```

```
$ docker compose up

$ docker compose down
```

Orchestrazione di container

- Docker Compose
- Docker Swarm
- Kubernetes
- OpenShift

Open Container Initiative (OCI)

- Standard spinto dalla Linux Foundation
- Nato nel 2015 da Docker e altri
- Scopo: avere container compatibili con vari gestori di container
- 3 specifiche:
  - Runtime
  - Image
  - Distribution

Svantaggi (storici) di Docker

- Dipende dal Docker daemon. E se non risponde?
- Deve girare da root* (o da gruppo docker, che ti rende root senza rendertene conto).

```
$ docker run -it -v /:/foo alpine
```

* dalla 20.10 (dicembre 2020) c'è Docker rootless

Podman

Podman

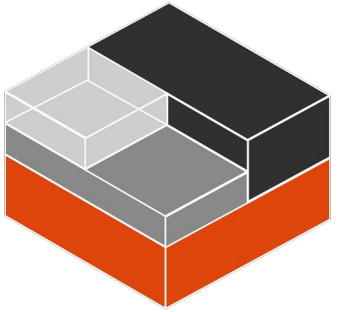
- Sviluppato da RedHat con licenza Apache 2.0
- Implementa standard OCI
- Stesse API di Docker
- Daemonless
- Rootless

```
$ alias docker=podman
```

Per i nomi delle immagini usare docker.io/<image-name>

LXC: Linux Containers

LXC: Linux Containers



- Iniziato a sviluppare da IBM nel 2008, rilasciato in GPLv2
- Usato nelle prime versioni di Docker (rimpiazzato tra la 0.9 e la 1.10 con *libcontainer*)
- Possono essere sia volatili che persistenti ← default!
- Adatto sia per singole applicazioni che per interi sistemi
- Posso far girare anche unprivileged containers

https://linuxcontainers.org

LXC – creazione container

```
# lxc-create -n container1 -t debian
```

L'opzione **-t** di **lxc-create** indica un template (script) che di default viene cercato in `/usr/share/lxc/templates`

```
# lxc-create -n container2 -t download -- \
     -d alpine -r edge -a amd64
```

Il template "download" scarica l'immagine da https://images.linuxcontainers.org/

LXC – Immagini

- Create con il tool **distrobuilder** sulla base di file di configurazione YAML nei quali possiamo specificare:
  - URL da cui scaricare l'immagine ISO base della distro
  - Chiavi PGP per verificare l'immgine e i pacchetti
  - Pacchetti aggiuntivi da installare
  - Modifiche a file di configurazione e script

  https://github.com/lxc/lxc-ci/tree/main/images
- Non c'è ereditarietà

LXC vs Docker – Comandi

Docker	LXC
`docker start`	`lxc-start`
`docker stop`	`lxc-stop`
`docker ps --all`	`lxc-ls -f`
`docker exec`	`lxc-attach`
`docker rm`	`lxc-destroy`

LXC – Supporto OCI

LXC supporta anche immagini OCI (es: Docker)


```
# lxc-create -n prova-oci -t oci -- \
    --url docker://alpine
```

LXD e Incus

- **LXD**: Gestore di container e VM, inizialmente parte del progetto LXC
- Preso in gestione da Canonical nel 2023

  → fork chiamato **Incus**

systemd-nspawn

be an
init system

be a time-sync
daemon, device manager,
Message-oriented
middleware mechanism,
login manager,
bootloader, DNS resolver

systemd-nspawn

2 comandi principali:

- **`systemd-nspawn`**: fa partire un comando o un SO dentro un container
- **`machinectl`**: gestisce immagni e container

+ integrazione con **`journalctl`**, **`systemd-analyze`**, ...

```
journalctl -M my-container
```

Creare le immagini

Non c'è alcun tipo di registry.

Posso crearle da tar o immagine qcow:

```
# machinectl pull-raw \
https://cloud.debian.org/[...].qcow2 debian12
```

Oppure posso usare dei tool indipendenti che installano la struttura di una distro in una data cartella:

```
# debootstrap bookworm /var/lib/machines/debian12
```

Tools: **debootstrap** (Debian), **pacstrap** (Arch), **dnf** --installroot (Fedora)

Avviare un'immagine

Avvia un singolo comando (di default bash, con PID 1) nel container:

```
# systemd-nspawn -M debian12
```

Fa il boot (avvia init come PID 1) dentro il container:

```
# systemd-nspawn -M debian12 --boot
```

Usando **machinectl start** internamente viene chiamato **systemd-nspawn** con **--boot** e altri flag:

```
# machinectl start debian12
```

Similitudini con servizi systemd

Abilitare il container a partire automaticamente all'avvio:

```
# machinectl enable debian12
```

Questo crea un symlink che punta a `/lib/systemd/system/systemd-nspawn@.service`

Vedere lo status del container:

```
# systemctl status systemd-nspawn@debian12
```

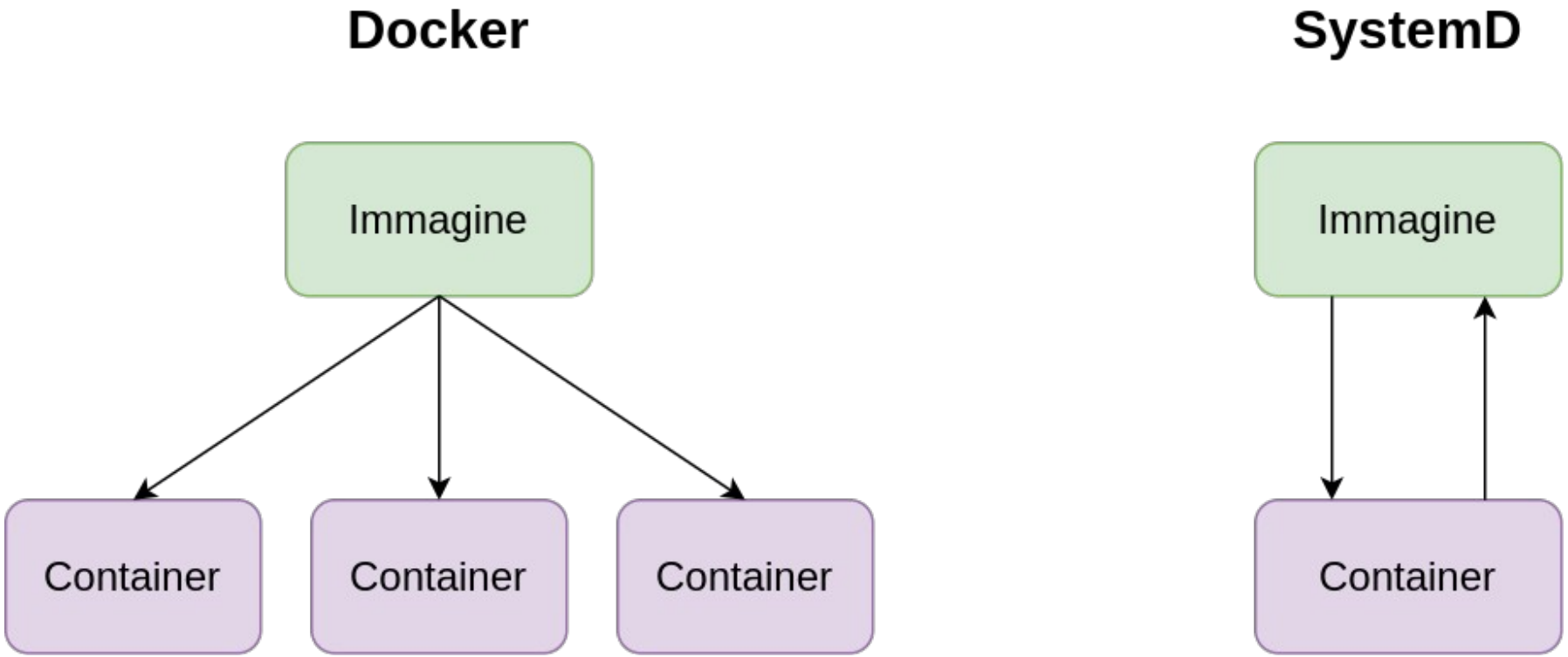
Similitudini con servizi systemd

Posso creare un file .nspawn, simile agli unit file dei servizi:

```
# vim /etc/systemd/nspawn/debian12.nspawn
```

```
[Exec]
User=pippo
Ephemeral=on
Environment="KEY=foo"
```

Immagine vs Container

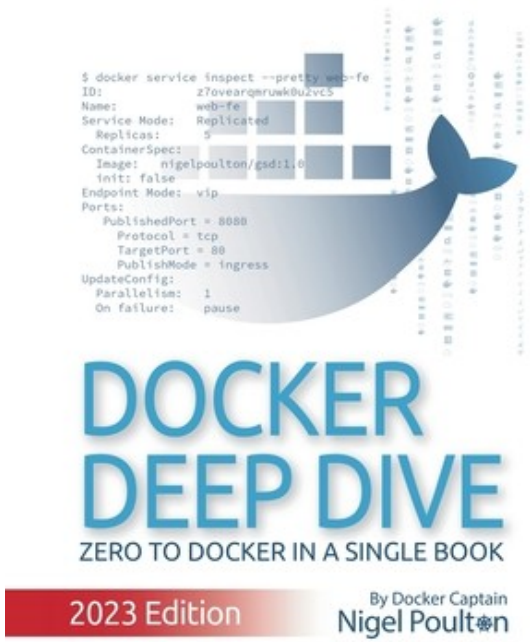


Supporto OCI

C’è un flag **`--oci-bundle`**, ma prima devo scaricare ed estrarre l’immagine OCI con altri tool (skopeo e umoci).

```
# skopeo copy docker://alpine oci:alpine:latest
# umoci unpack --image alpine:latest alpine_unpacked
# systemd-nspawn --oci-bundle alpine_unpacked
```

Domande?


$ docker service inspect --pretty web-fe
ID: z7ovearqmruwk0u2vc5
Name: web-fe
Service Mode: Replicated
Replicas: 5
ContainerSpec:
Image: nigelpoulton/gsd:1.0
init: false
Endpoint Mode: vip
Ports:
PublishedPort = 8080
Protocol = tcp
TargetPort = 80
PublishMode = ingress
UpdateConfig:
Parallelism: 1
On failure: pause
DOCKER
DEEP DIVE
ZERO TO DOCKER IN A SINGLE BOOK
2023 Edition
By Docker Captain
Nigel Poulton