



Red Hat



Linux Day
2025 Torino



Containers from Scratch: cgroups & Namespaces

Linux Day Torino



Alex Guidi
Ingegnere Software Senior
presso Red Hat



[linkedin.com/in/alex-guidi/](https://www.linkedin.com/in/alex-guidi/)



Agenda

- Contesto – Fondamenta dei Container
- Costruiamo un container - senza podman o docker
- Conclusione



Fondamenta dei Container

La "Magia" di podman o docker run

- Probabilmente avete già usato `podman run` o `docker run`
- Sembra magia: un ambiente completo e isolato appare in pochi secondi.
- Ma non è magia. Non è una Macchina Virtuale.
- **Obiettivo di questo talk:** Demistificare i container costruendo uno da zero, usando solo strumenti Linux standard su fedora.



podman

Cos'è un container?

Un container è solo un **processo Linux in sandbox**.

Questa sandbox è costruita usando due caratteristiche principali del kernel Linux:

1. **Namespaces:** Limitano ciò che un processo può **VEDERE**.
2. **cgroups (Control Groups):** Limitano quanto un processo può **USARE**.

Insieme, creano un ambiente completo e isolato.

I Pilastri dell'Isolamento: Namespaces

I Namespaces separano le risorse di sistema globali, dando ad un processo una prospettiva privata.

- **pid (ID di processo)**: Il suo proprio albero di processi, a partire da PID 1.
- **mnt (Mount)**: La sua propria prospettiva del filesystem.
- **net (Rete)**: Le sue proprie interfacce di rete e indirizzi IP.
- **uts (Nome Host)**: Il suo proprio nome host.
- **ipc (Comunicazione Inter-Processo)**: Memoria condivisa isolata
- **user (ID Utente)**: Mappa il root del container (UID 0) a un utente non-root sull'host.

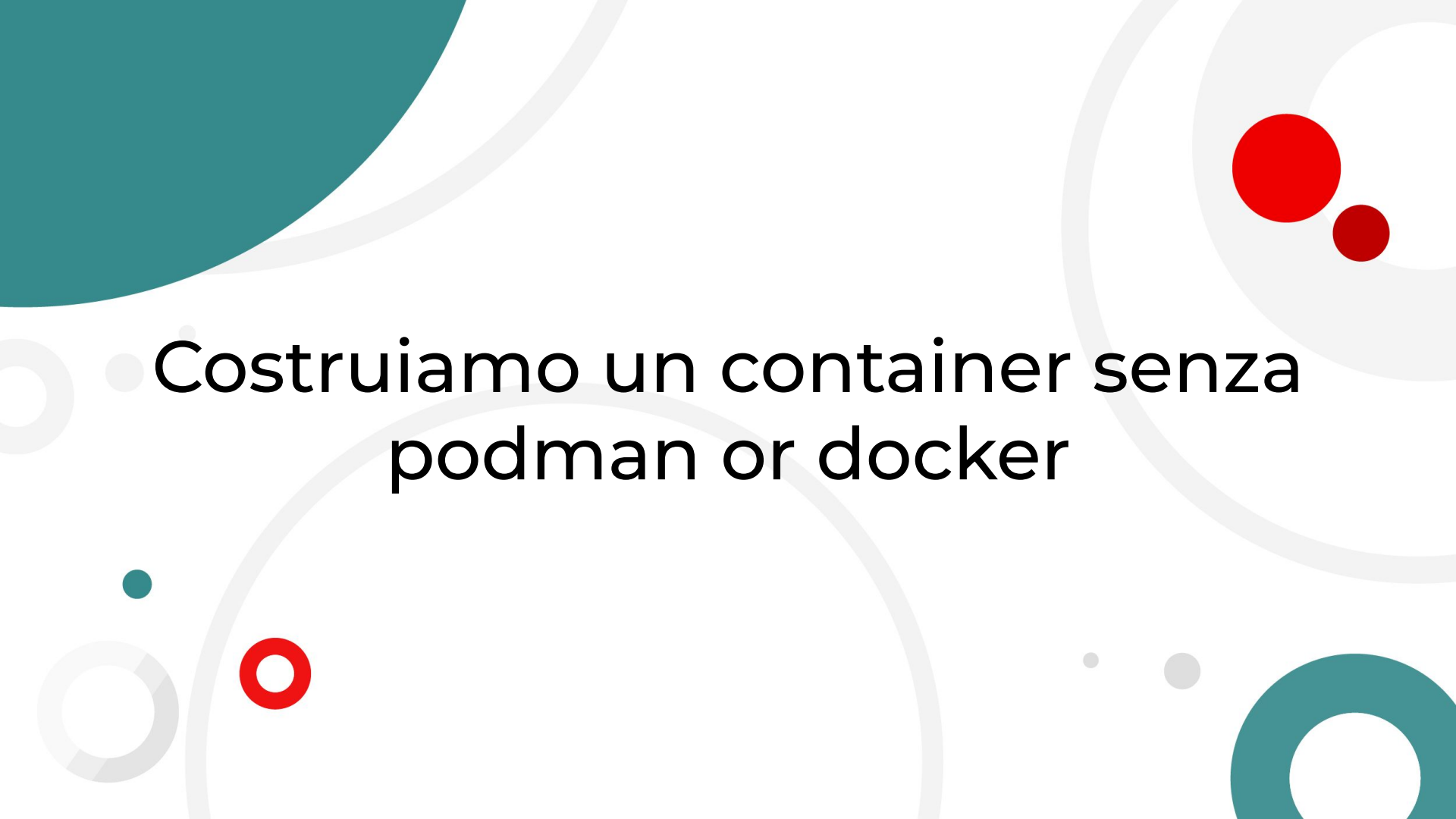
Analogia: L'host è una casa; i namespaces sono stanze isolate con le loro proprie utenze.

Guardrails: cgroups v2

Mentre i namespaces forniscono i muri, i cgroups forniscono i contatori delle utenze.

- **cgroups (Control Groups)** limitano, contabilizzano e isolano l'uso delle risorse.
- Fedora usa il moderno **cgroup v2**, gestito attraverso una gerarchia unificata a `/sys/fs/cgroup/`.
- **Controllers per le risorse chiave:**
 - **cpu**: Tempo di CPU, quote (`cpu.max`).
 - **memory**: Limiti di utilizzo della memoria (`memory.max`, `memory.swap.max`).
 - **io**: Blocchi I/O (accesso al disco).
 - **pids**: Numero di processi.

Quando esegui `podman/docker run --memory=1g`, Podman sta solo scrivendo valori in questo file cgroup.

- 
- Costruiamo un container senza podman or docker

Costruiamo un Container!

Eseguiamo manualmente i passaggi che un runtime di container compie.

Il Piano:

1. **Prepara il Terreno:** Crea un filesystem root (`dnf`).
2. **Costruisci i Muri:** Isola il processo con i namespaces (`unshare`).
3. **Stabilisci la Connettività:** Crea una rete privata (coppie `veth`).
4. **Imposta le Regole:** Applica i limiti di risorse (`cgroups v2`).

Step 1: II Filesystem Root

Ogni container ha bisogno del suo set di file e librerie. Su Fedora usiamo **dnf**.

```
1 # Create a directory for our new root filesystem
2 mkdir fedora-rootfs
3
4 # Use dnf to install a minimal Fedora system
5 sudo dnf install --installroot=$(pwd)/fedora-rootfs --releasever=42 --use-host-config @core --
  nogpgcheck
```

Step 1: Il Filesystem Root

Ora, entriamo in questo filesystem usando **mount namespace** per isolare cosa può vedere il processo.

```
1 # Create a new mount namespace and set our root to the new directory
2 sudo unshare --mount --root=./fedora-rootfs /bin/bash
```

Step 2: Isolamento del Processo e del Sistema

Aggiungiamo altri namespaces per creare un sandbox più completo.

- `--uts`: Isola il nome host.
- `--ipc`: Isola la comunicazione inter-processo.
- `--pid --fork --mount-proc`: Crea un nuovo albero di processi. La nostra shell diventa **PID 1**.

```
1 sudo unshare --mount --uts --ipc --pid --fork --mount-proc --root=./fedora-rootfs /bin/bash
```

Step 2: Isolamento del Processo e del Sistema

All'interno della nuova shell:

- `ps aux` mostrerà solo la shell e il comando ps stesso.
- `hostname my-container` non cambierà il nome host effettivo dell'host.

Step 3: Rete Privata

Aggiungi il flag `--net` per dare al container il suo proprio stack di rete.

```
1 sudo unshare --mount --uts --ipc --pid --fork --mount-proc --net --root=./fedora-rootfs /bin/bash
```

Step 3: Rete Privata

```
1 # (From host) Find the container's PID
2 CONTAINER_PID=$(ps -o pid,comm --ppid $(ps -o pid,comm -C unshare | awk '/unshare/{print $1}') | awk
  '/bash/{print $1}')
```

```
1 # (From host) Create a virtual ethernet pair
2 sudo ip link add veth-host type veth peer name veth-container
```

```
1 # (From host) Move one end into the container's network namespace
2 sudo ip link set veth-container netns $CONTAINER_PID
```

```
1 # (From host) Configure the host side
2 sudo ip addr add 10.0.0.1/24 dev veth-host
3 sudo ip link set veth-host up
```

Step 3: Rete Privata

```
1 # (Inside container) Configure the container side
2 ip link set lo up
3 ip link set veth-container up
4 ip addr add 10.0.0.2/24 dev veth-container
5 ip route add default via 10.0.0.1
```

Step 4: Applicazione dei Limiti di Risorse (cgroup v2)

Il nostro container è isolato, ma non vincolato. Limitiamo la sua memoria e CPU.

```
1 # (From host) Create a cgroup directory  
2 sudo mkdir /sys/fs/cgroup/my-container
```

```
1 # (From host) Enable memory and cpu controllers for the new cgroup  
2 sudo sh -c 'echo "+cpu +memory" > /sys/fs/cgroup/cgroup.subtree_control'
```

Step 4: Applicazione dei Limiti di Risorse (cgroup v2)

```
1 # (From host) Set a 100MB memory limit and disable swap
2 sudo sh -c 'echo 100000000 > /sys/fs/cgroup/my-container/memory.max'
3 sudo sh -c 'echo 0 > /sys/fs/cgroup/my-container/memory.swap.max'
```

```
1 # (From host) Set a 50% CPU limit
2 sudo sh -c 'echo "50000 100000" > /sys/fs/cgroup/my-container/cpu.max'
```

```
1 # (From host) Assign our container process to these cgroups
2 sudo sh -c "echo $CONTAINER_PID > /sys/fs/cgroup/my-container/cgroup.procs"
```

Step 4: Verifica dei Limiti (Memoria)

Come sappiamo che i limiti di memoria stanno funzionando ? Testiamoli.

- **Test di Memoria (All'interno del Container):** Prova ad allocare 200MB di memoria con Python.

```
1 python3 -c "mem = bytearray(200 * 1024 * 1024)"
```

Risultato: **Killed**. L'OOM killer è stato invocato! **Prova (Sull'Host):**

```
1 sudo dmesg | grep -i "out of memory" or cat /sys/fs/cgroup/my-container/memory.events
```

Step 4: Verifica dei Limiti (CPU)

Come sappiamo che i CPU limiti stanno funzionando? Testiamoli.

- **Test CPU (All'interno del Container):** Esegui un loop infinito.

```
1 while true; do ;; done
```

Prova (Sull'Host): `sudo systemd-cgtop` mostrerà il cgroup `my-container` cgroup limitato al **50% di CPU**.

Conclusione: La magia rivelata

- Non c'è magia. L'intero ecosistema dei container è costruito su queste primitive fondamentali del kernel Linux.
- `podman/docker run` è un comando di alto livello che automatizza questi esatti passaggi:
 - Chiama un runtime come `runc`.
 - `runc` esegue le chiamate di sistema `unshare` e scrive nei file `/sys/fs/cgroup` basandosi su un file di configurazione.
- Comprendere questi blocchi fondamentali è la chiave per la risoluzione avanzata dei problemi, l'ottimizzazione delle prestazioni e la sicurezza.

Domande?

Grazie!

Codice:

<https://gist.github.com/aguidirh/29175845d6f7f349d08807011cf4d7f8>



Alex Guidi

Ingegnere Software Senior
presso Red Hat



[linkedin.com/in/alex-guidi/](https://www.linkedin.com/in/alex-guidi/)

