Container-based Virtualization and Services Deployment

Advanced Computer Programming

Prof. Luigi De Simone

Sommario

- Introduction to containers
- Containers vs. Virtual Machines
- Kernel namespaces and cgroups
- Container technologies
- Container orchestration

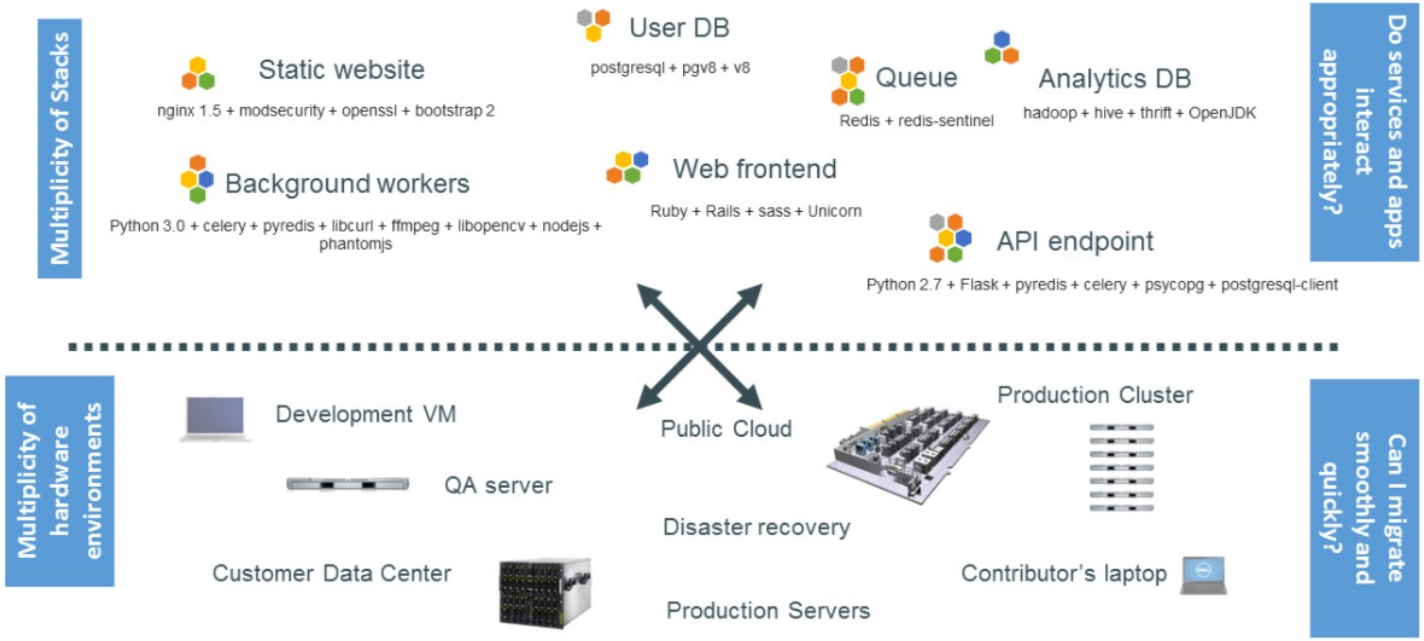
Riferimenti

- Docker documentation: https://docs.docker.com/
- Docker Swarm. https://docs.docker.com/engine/swarm/
- Swarm tutorial. https://rominirani.com/docker-swarm-tutorial-b67470cf8872
- Esempi Docker compose. https://github.com/docker/awesome-compose/

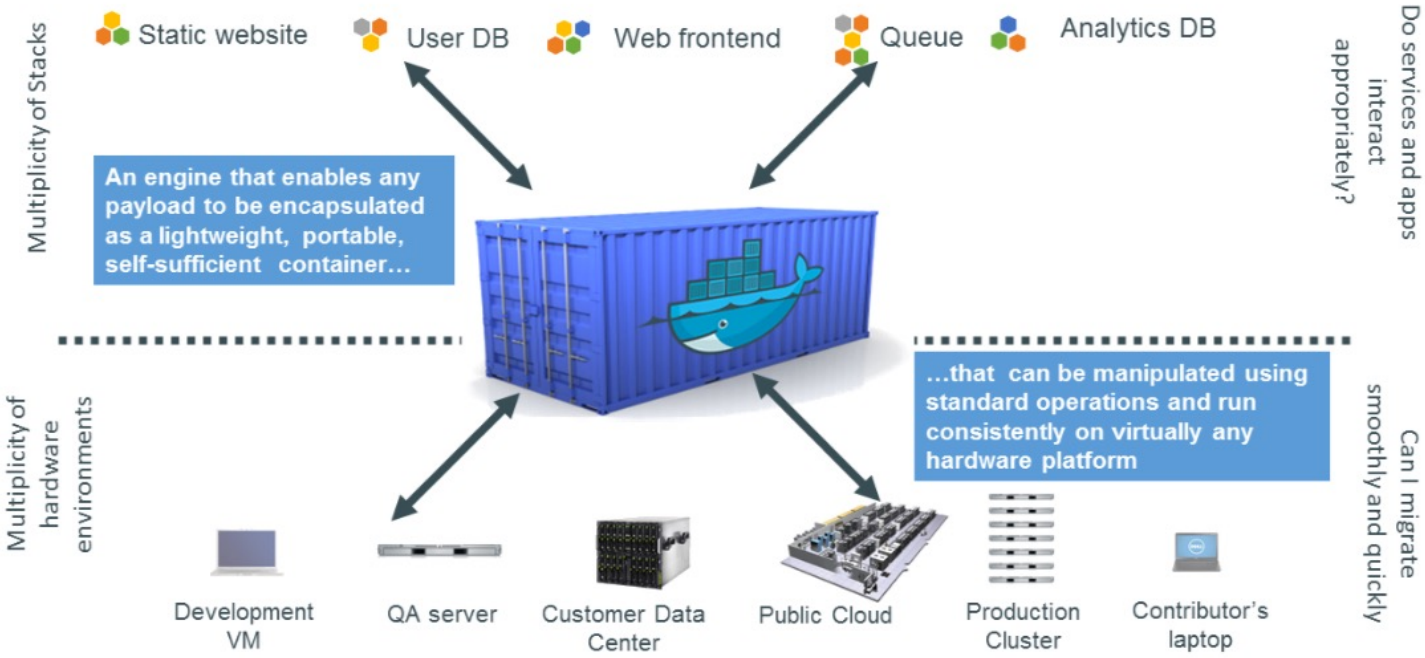
Uso delle virtual machine (VM)

- **Overhead di prestazioni** dovuto alle indirezioni software usate (sistema operativo guest e hypervisor, macchine virtuali innestate)
- **Grande** ingombro di **memoria** e **storage**
- Tempo di avvio **lento** (ordine di minuti)
- **Costi** di **licenza** e **manutenzione** del sistema operativo guest
- È davvero necessario virtualizzare tutti i dispositivi hardware e un sistema operativo completo?

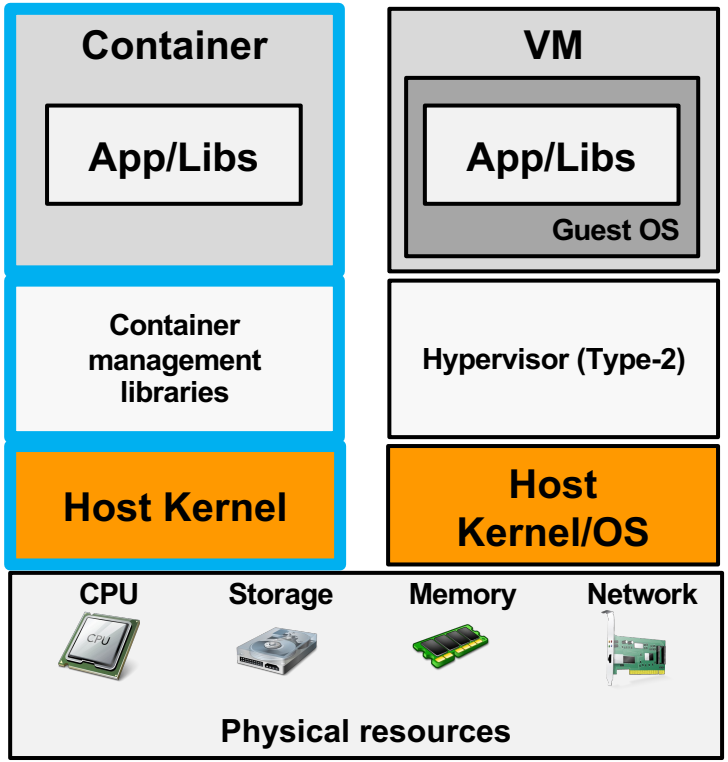
Problematiche



Soluzione basata su *container*



VM vs. Container



- Un container **non è una macchina virtuale** nel senso tradizionale del termine.
  - **Non c'è emulazione** di dispositivi fisici
- **Virtualizzazione leggera**
  - Nessun overhead dovuto al livello di hypervisor
- Sfrutta **l'astrazione dei processi** e le capacità di **gestione delle risorse** del sistema operativo del kernel host
  - Namespace
  - Cgroups

Containers performance

	BARE METAL	VIRTUAL MACHINE	CONTAINER
Boot Time	~5-10 min	~2 min	~2 secs
App Deployment Lifecycle	Deploy in weeks Live for years	Deploy in minutes Live for weeks	Deploy in seconds Live for minutes/hours
Deployment Complexity	**Need to know:** 1. Hardware 2. OS 3. Runtime environment 4. Application code	**Need to know:** 1. OS 2. Runtime environment 3. Application code	**Need to know:** 1. Runtime environment 2. Application code
Investment	Buy/rent dedicated server	Rent a dedicated VM on a shared server	Rent containers and pay for actual runtime
Scaling	Can take months* Should be approved by a panel of experts	Takes hours Should be approved by adminstrators	Takes seconds Policy driven scaling

**For on-premises bare metal. Bare metal cloud services can be dynamically provisioned in minutes*

Sources: Arun Kottolli (average bare metal boot time = multiple sources)

Container footprint

- Utilizzando un tipico server fisico, con risorse di calcolo medie, è possibile eseguire circa **10-100 VMs vs 100-1000 containers**

- Con i container è possibile utilizzare pochi MB di disco virtuale, a seconda dell'**immagine** di partenza

Namespaces

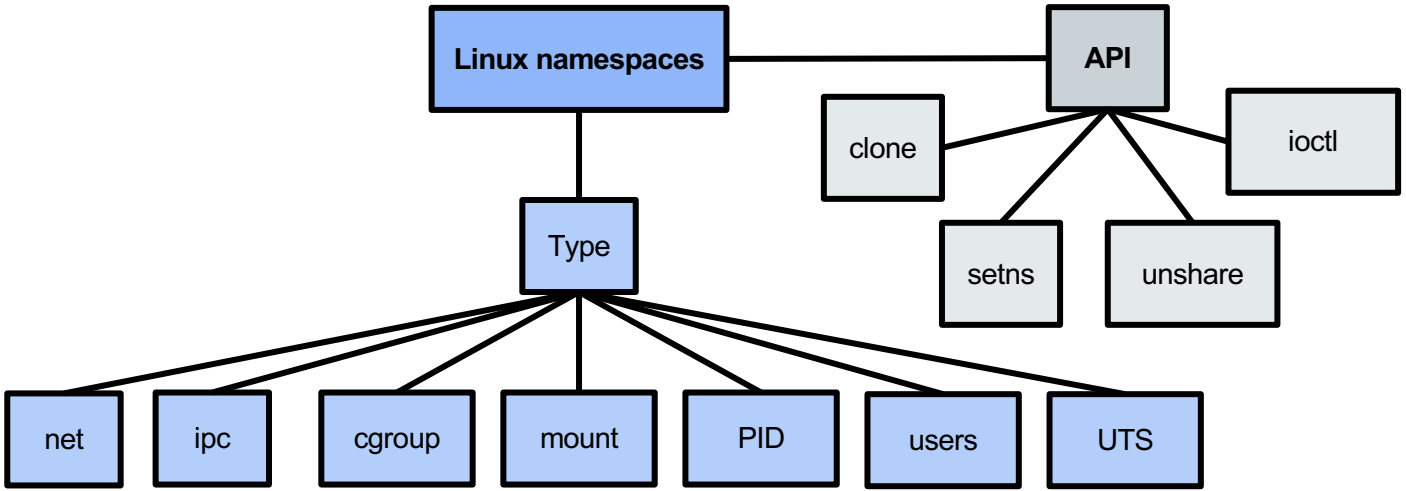
Namespace: dominio di denominazione per varie risorse

- `mnt` (mount points, filesystems)
- `pid` (processes)
- `net` (network stack)
- `ipc` (System V IPC)
- `uts` (hostname)
- `user` (UIDs)

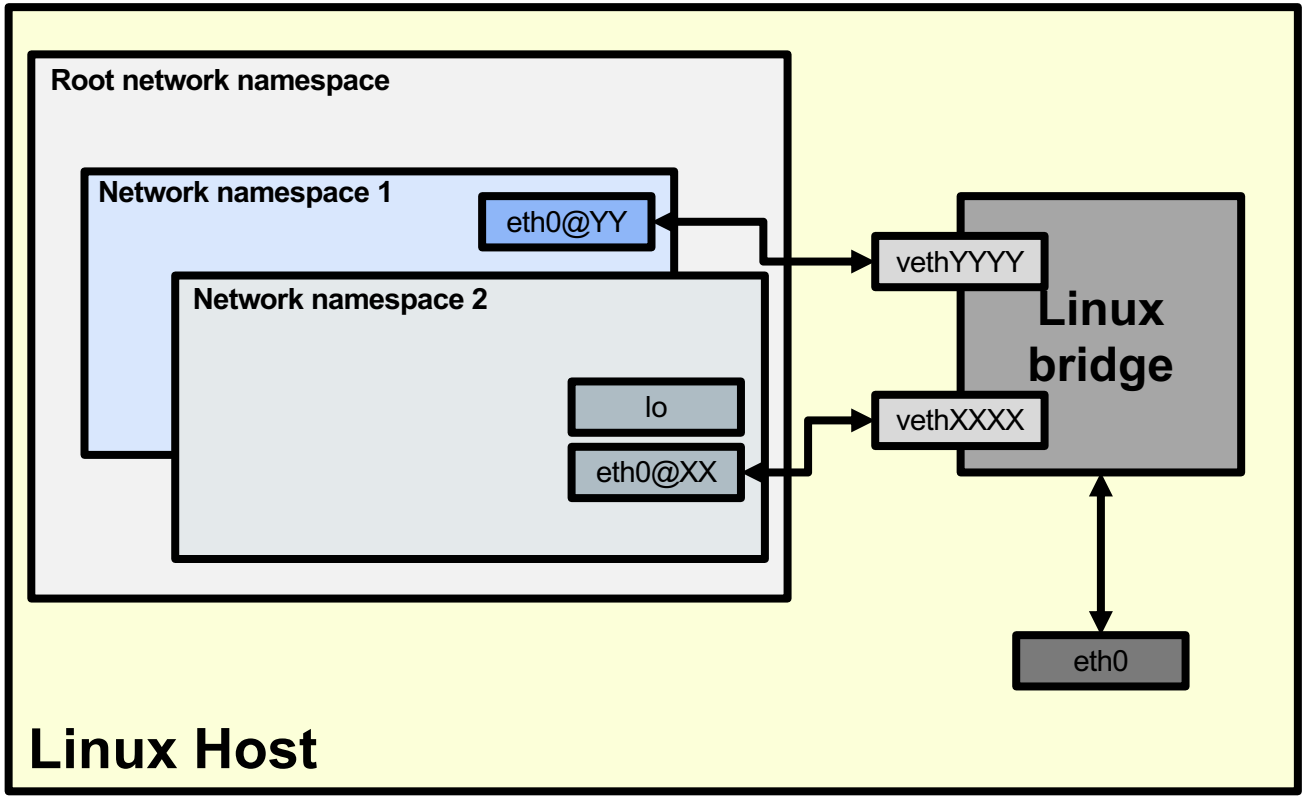
Namespace system calls

- **`clone()`** – crea un **nuovo processo** e un **nuovo namespace**; til processo viene "attaccato" al nuovo namespace
  - **`fork()`** e **`exit()`** sono modificati in modo da gestire nuovi flag per i nasmespace (e.g., `CLONE_NEWxxx`)
- **`unshare()`** - non crea un nuovo processo; crea un nuovo namespace e lo "attacca" al processo corrente
  - Questa system call fu aggiunta nel 2005, non soltanto per gestire i namespace ma anche per questioni legate alla sicurezza
- **`setns()`** - a una nuova system call aggiunta per fare il join di un namespace esistente

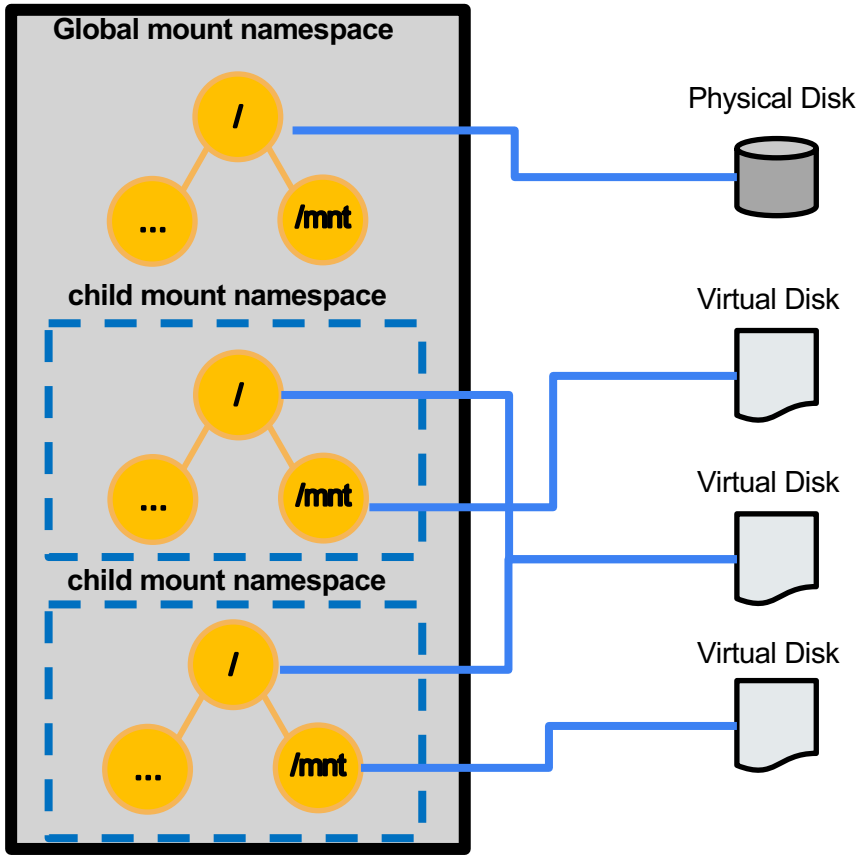
Namespaces and Syscalls



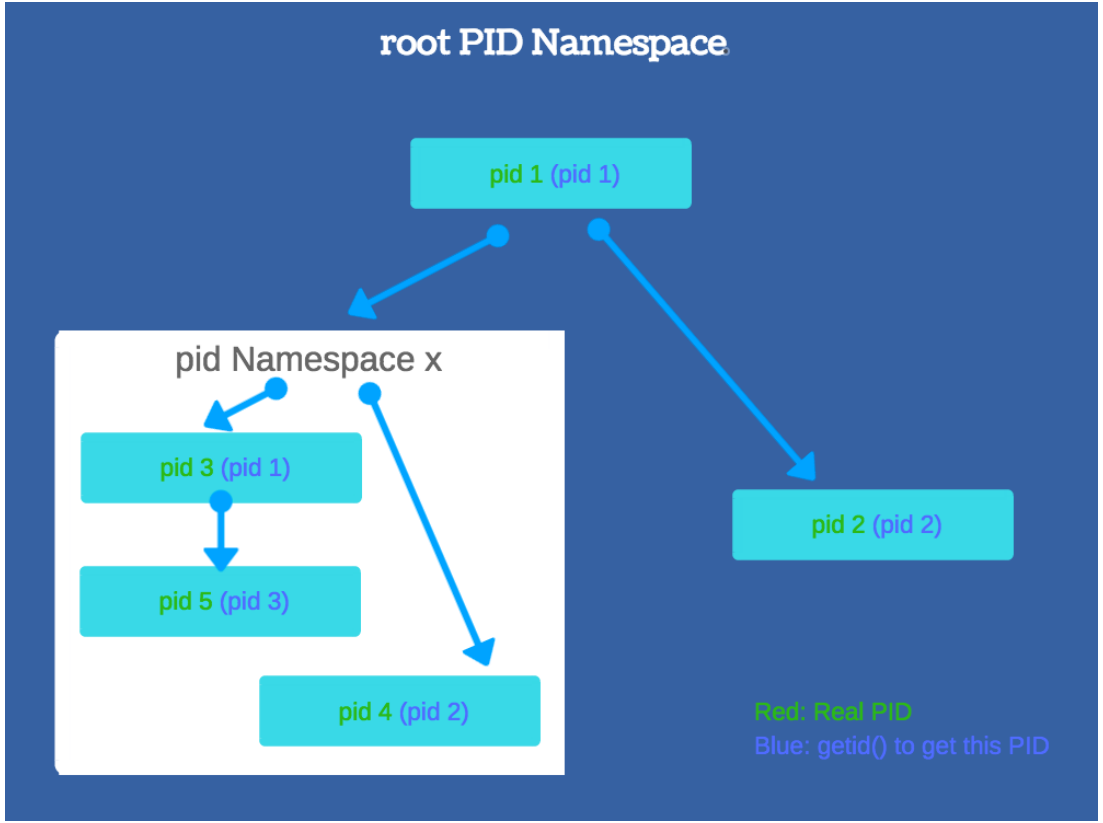
Network namespace example



Mount namespace example



PID namespace example



Cgroups

- Il sottosistema *Cgroups* (gruppi di controllo) è una soluzione di **gestione delle risorse** che fornisce un framework generico per gruppi di processi
- Progetto nato da Google
  - nel 2006 con il nome di *process container*
  - nel 2007, rinominato in ***cgroup***
- Il controller della memoria (memcg) è gestito separatamente
  - **Probabilmente il più complesso**
- I Namespace forniscono una **soluzione di isolamento delle risorse per processo, i Cgroup per gruppi**

Cgroups implementation

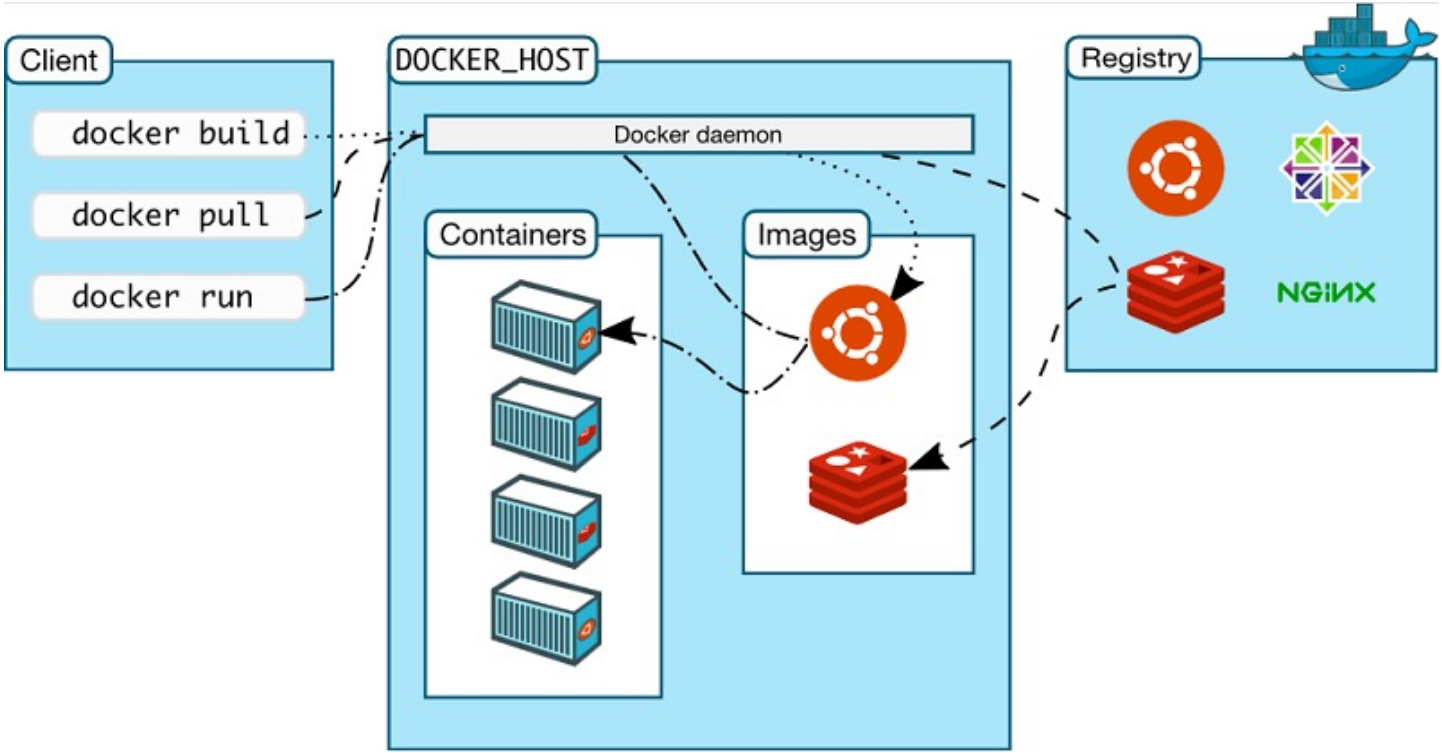
- I moduli di cgroup non si trovano in un'unica cartella, ma sono sparsi nell'albero del kernel in base alla loro funzionalità

  - **memory** - imposta limiti all'uso della memoria da parte dei task di un cgroup e genera report automatici sulle risorse di memoria utilizzate da tali task. (`mm/memcontrol.c`)
  - **cpuset** - assegna le singole CPU (in un sistema multicore) e i nodi di memoria ai task di un cgroup. (`kernel/cpuset.c`)
  - **net_prio** - fornisce un modo per impostare dinamicamente la priorità del traffico di rete per ogni interfaccia di rete (`net/core/netprio_cgroup.c`).
  - **devices** - consente o nega l'accesso ai dispositivi da parte dei task di un cgroup. (`security/device_cgroup.c`)
  - **blkio** - imposta limiti sull'accesso di input/output a e da dispositivi a blocchi come le unità fisiche (dischi, stato solido, USB, ecc.).
  - E così via (cpu, cpuact, freezer, net_cls, ns)

Docker

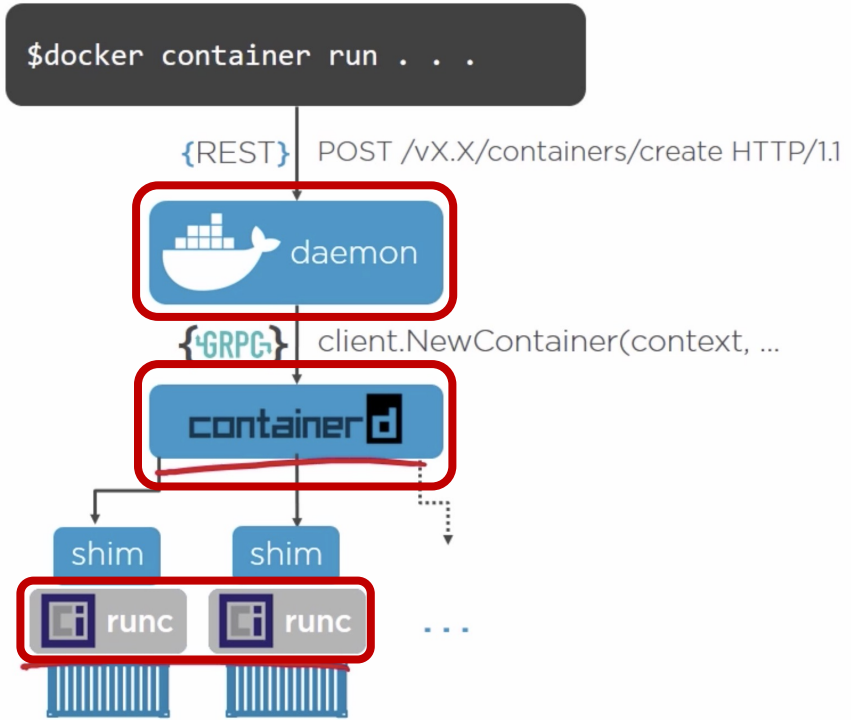
- **Docker Inc.**
  - Fondata come dotCloud, Inc. nel 2010 da Solomon Hykes (rinominata in Docker Inc. nel 2013)
  - Valutazione stimata di oltre 1 miliardo di dollari (101-250 dipendenti)
- **Docker il software**
  - Un motore di container scritto in **Go** (basato su container Linux)
  - Docker è costruito per eseguire un'applicazione in un container
- **Comunità Docker**
  - Attualmente 1851 collaboratori, 16,2K fork del motore Docker su GitHub (chiamato Moby, https://github.com/moby/moby)

Architecture Docker



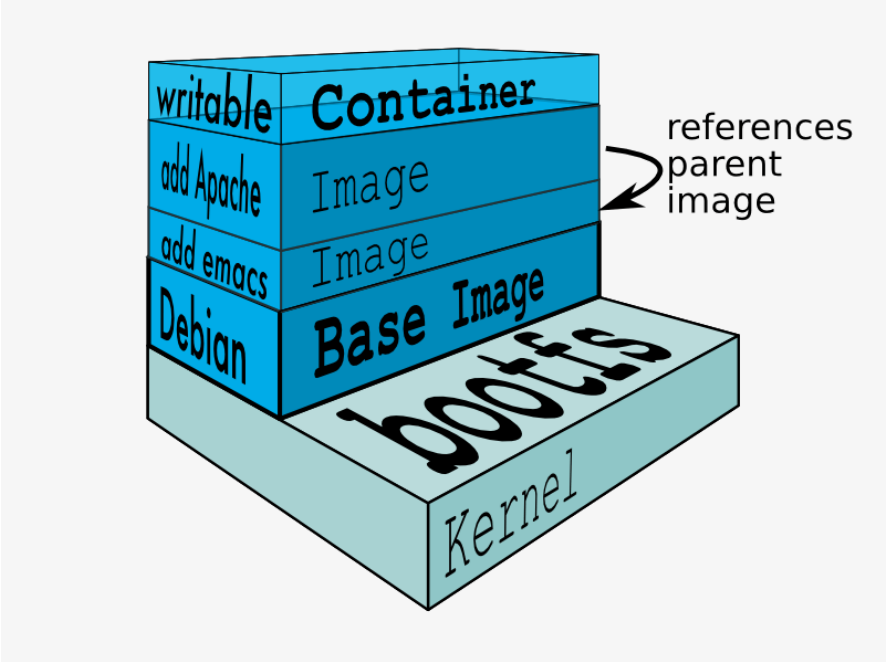
Docker *engine*

- **daemon**: API REST (ricezione di istruzioni) e altre funzionalità
- **containerd**: Logica di esecuzione (ad esempio, avvio, arresto, pausa, disattivazione, cancellazione di contenitori)
- **runc**: Un'interfaccia a riga di comando (CLI) leggera per il runtime conforme a OCI

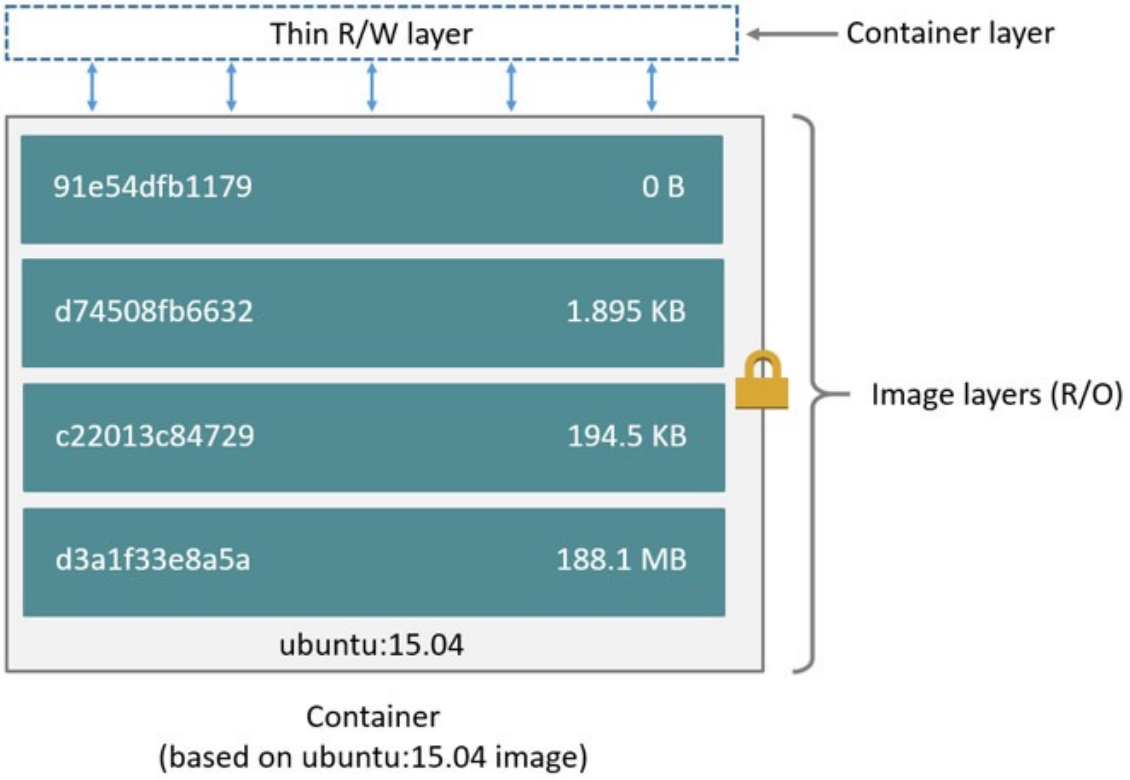


Immagini Docker

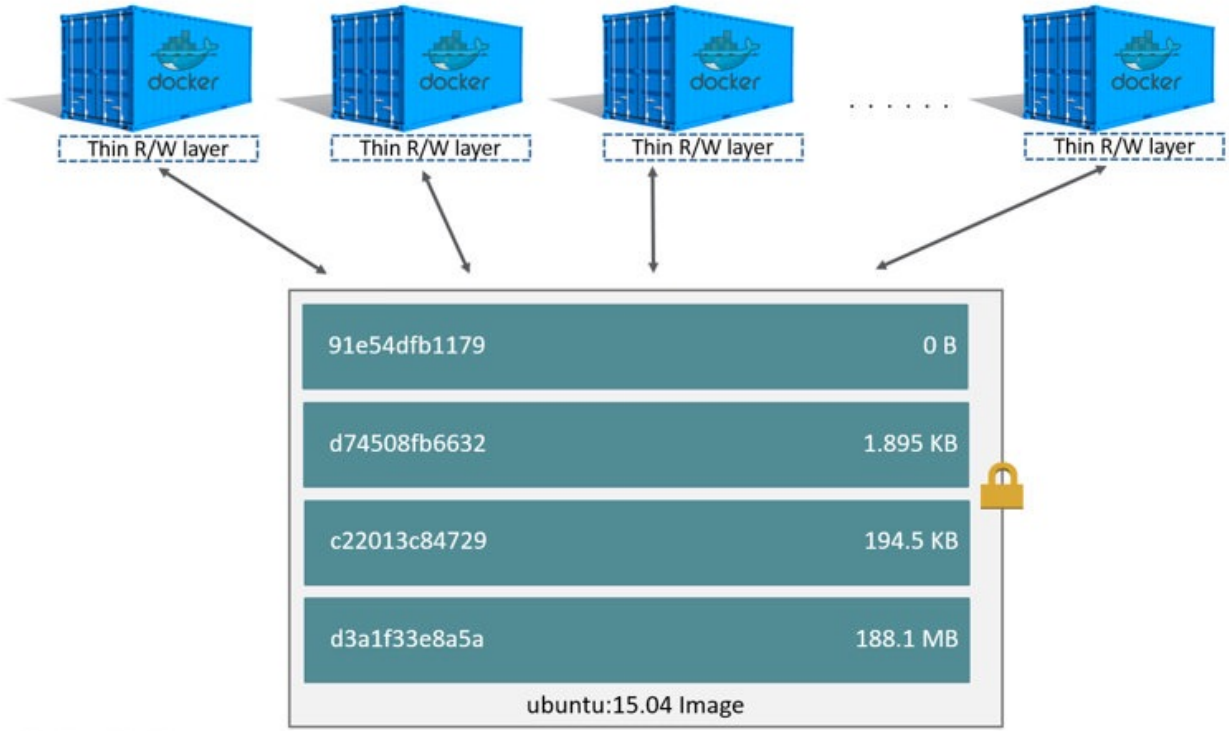
- Non è un disco rigido virtuale, nè un file system
- Utilizza un **Union File System**
- È un **layer di sola lettura**
- **Non ha stato**
- Fondamentalmente un **file compresso**
- **Ha una gerarchia** (profondità arbitraria)



Overlay filesystem (OverlayFS)



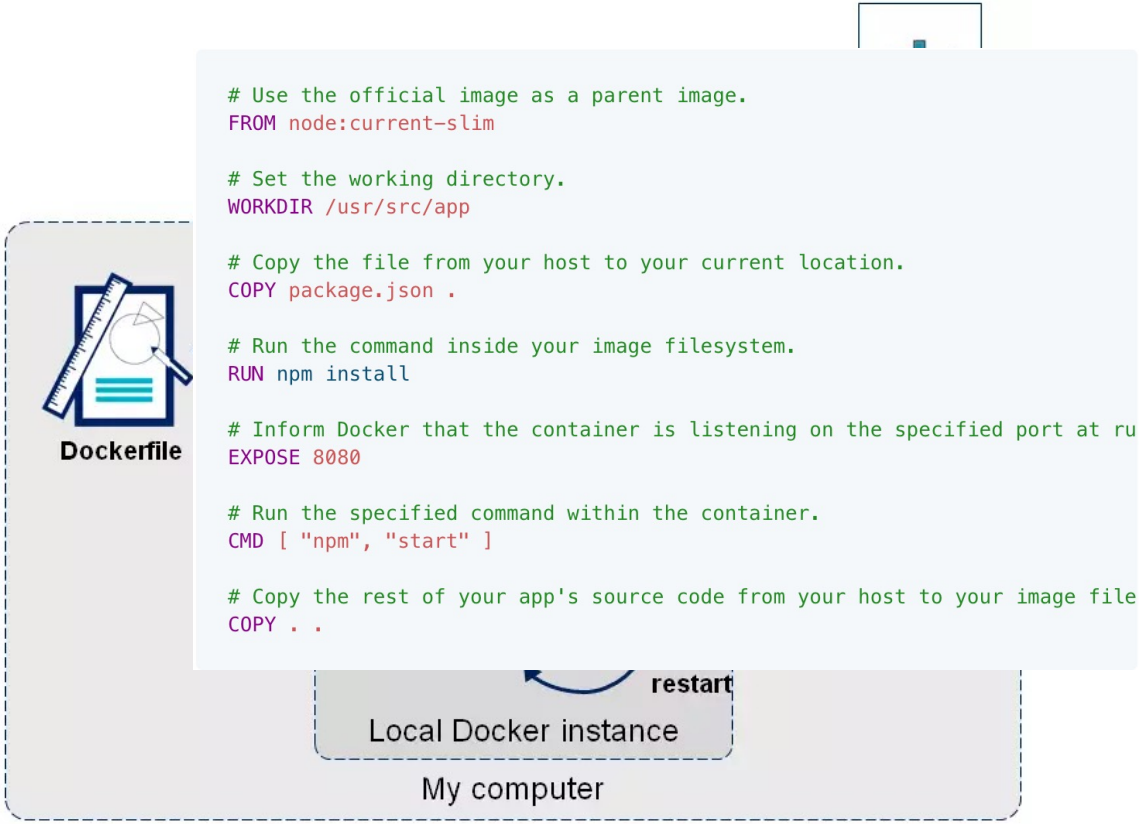
Container and layers



Docker *registry*

- Un **registro** contenente immagini Docker
  - Registro locale sullo stesso host
  - Registro di Docker Hub
  - Registro privato condiviso a livello globale su docker.com
- Controlla **dove vengono archiviate le immagini**
- Controlla completamente la **pipeline di distribuzione delle immagini**
- Integra l'archiviazione e la distribuzione delle immagini nello specifico flusso di lavoro di sviluppo

Docker workflow




```
# Use the official image as a parent image.
FROM node:current-slim

# Set the working directory.
WORKDIR /usr/src/app

# Copy the file from your host to your current location.
COPY package.json .

# Run the command inside your image filesystem.
RUN npm install

# Inform Docker that the container is listening on the specified port at ru
EXPOSE 8080

# Run the specified command within the container.
CMD [ "npm", "start" ]

# Copy the rest of your app's source code from your host to your image file
COPY . .
```

Installazione di Docker (*engine*)

- Fare riferimento alle **guide ufficiali**
  - https://docs.docker.com/engine/install/
- **Per sistemi Ubuntu-based seguire la guida al link**:

  https://docs.docker.com/engine/install/ubuntu/
  - Impostare i repository apt
  - Installare i package Docker
  - Verificare che l'installazione del Docker Engine è andata a buon fine
- E' possibile anche utilizzare uno **script automatico non interattivo**

```
$ curl -fsSL https://get.docker.com -o get-docker.sh
$ sudo sh ./get-docker.sh --dry-run
```

Comandi di base in Docker

- **`docker build –`** costruire un'immagine docker a partire dal Dockerfile
- **`docker run`** - avviare un nuovo container Docker da un'immagine.
- **`docker ps`** - elencare tutti i container Docker in esecuzione.
- **`docker stop`** - arrestare un container in esecuzione.
- **`docker rm -`** rimuovere un container Docker
- **`docker images`** - elencare tutte le immagini Docker attualmente disponibili sul sistema
- **`docker pull`** - scaricare un'immagine Docker da un registro
- **`docker exec`** - eseguire un comando in un container in esecuzione

- **Consultare il cheatseet**: https://docs.docker.com/get-started/docker_cheatsheet.pdf

Dockerfile

- **Riferimento per Dockerfile**: https://docs.docker.com/reference/dockerfile/

- **Direttive principali**
  - `FROM:` crea un nuovo layer da aggiungere all'immagine specificata
  - `WORKDIR:` cambia la working directory di default
  - `COPY:` copia file in directory
  - `RUN:` esegue i comandi specificati creando un nuovo layer al di sopra di quella da cui si sta partendo
  - `ENTRYPOINT:` specifica l'eseguibile di default lanciato all'avvio del container
  - `CMD:` specifica i comandi di default
  - `EXPOSE`: definisce i porti ai quali l'app nel container deve mettersi in ascolto

Esempio: creazione di un app Flask containerizzata

1. Implementazione dell'applicazione Fask

```
# flask_app.py
from flask import Flask, request
import socket

app = Flask(__name__)

@app.route("/")
def index():

    return "This is an example containerized Flask app served from {} to
{}".format(socket.gethostname(), request.remote_addr)


if __name__ == "__main__":
    app.run(host='0.0.0.0', port=5001, debug=True)
```

Esempio: creazione di un app Flask containerizzata

2. Definizione del Dockerfile

```
FROM python:3.10-alpine

WORKDIR /app # assumiamo che l'app flask sia all'interno della dir app

COPY requirements.txt /app
RUN  pip3 install -r requirements.txt

COPY flask_app.py /app

ENTRYPOINT ["python3"] # ENTRYPOINT viene trattato come parametro da dare al docker run

CMD ["flask_app.py"] # viene ignorato nel caso in cui passiamo un argomento (oltre a quello specificato in ENTRYPOINT, tramite docker run
```

Esempio: creazione di un app Flask containerizzata

3. Build dell’immagine

```
$ cd /path/to/Dockerfile
$ docker build –t flask_image_hello_world .
```

3. Run del container

```
$ docker run –it ––name flask_app_container –p 5001:5001 flask_image_hello_world
```

```
root@dockertest1:~# docker ps –a
CONTAINER ID   IMAGE                   COMMAND            CREATED         STATUS         PORTS                                        NAMES
d0a3f0b406a9   flask_image_hello_world   "python3 app.py"   3 seconds ago   Up 2 seconds   0.0.0.0:5001->5001/tcp, :::5001->5001/tcp   flask_app_container
root@dockertest1:~#
```

Deploy di un'applicazione multi-container
Docker compose

- Per fare il **deploy** di un'applicazione multi-container bisogna
  - Farne il *build* e il *run* di tutti i container che costituiscono l'applicazione
  - Eventualmente collegare (manualmente) i container per la comunicazione
  - Fare attenzione alle dipendenze per l'ordine di avvio dei container
- **Docker compose** ovvia queste limitazioni permettendo di
  - Specificare un'applicazione multi-container definendola in un **file di specifica .yaml**
  - Utilizzare un **singolo comando per fare il deploy** dell'applicazione
  - **Gestire le dipendenze** di avvio per i container in maniera programmatica
  - Supporto nativo all'orchestratore Docker Swarm (più avanti nelle slide)

Docker compose file

Alcuni dei parametri più importanti sono i seguenti

- **build**: specifica la configurazione (Dockerfile) di compilazione per la creazione di un'immagine del container
  - **context:** è possibile usarlo per definire o un percorso ad una directory che contiene il Dockerfile, oppure un URL ad un repo git
- **command**: sovrascrive il comando predefinito dichiarato dall'immagine del contenitore, ad esempio dal CMD di Dockerfile
- **ports**: espone i porti per il container
- **volumes**: definisce i percorsi del filesystem dell'host (mount point) accessibili dal container
- **links**: definisce un collegamento di rete ai contenitori di un altro servizio.
- **image**: specifica l'immagine da cui avviare il container
- **environment**: definisce le variabili d'ambiente impostate nel contenitore.

```
app:
  build: .
  command: python -u app.py
  ports:
    - "5000:5000"
  volumes:
    - .:/app
  links:
    - db
db:
  image: mongo:latest
  hostname: test_mongodb
  environment:
    - MONGO_INITDB_DATABASE=animal_db
    - MONGO_INITDB_ROOT_USERNAME=root
    - MONGO_INITDB_ROOT_PASSWORD=pass
  volumes:
    - ./init-db.js:/docker-entrypoint-initdb.d/init-db.js:ro
  ports:
    - 27017:27017
```

Installazione e comandi in Docker compose

- **Installazione** (Linux)

```
sudo apt-get update
sudo apt-get install docker-compose-plugin
```

- **Comandi** (un subset)
  - **`docker compose up`:** build e run di tutti i container (usare –d per la modalità daemonized)
  - **`docker compose down`:** stop e distruzione di tutti i container
  - **`docker compose ps`:** lista i container legati al progetto Compose il cui file di config è nella *current dir*
  - **`docker compose logs`:** stampa i log legati al progetto Compose il cui file di config è nella *current dir*
  - *. . .*

Esempio: creazione di un app Flask containerizzata tramite Docker compose

- **Definizione del compose file**

compose.yaml

```
services:
  web:
    build:
     context: app
    ports:
      - '5001:5001'
```

- **Avvio della composizione e run del container**

```
$ docker compose -f compose.yaml up -d
```

```
root@dockertest1:~/ACP/docker_deploy_examples/dockerized_flask_service# docker compose -f compose.yaml up -d
[+] Building 0.0s (0/0)
[+] Running 2/2
 ✔ Network dockerized_flask_service_default  Created
 ✔ Container dockerized_flask_service-web-1  Started
root@dockertest1:~/ACP/docker_deploy_examples/dockerized_flask_service#
```

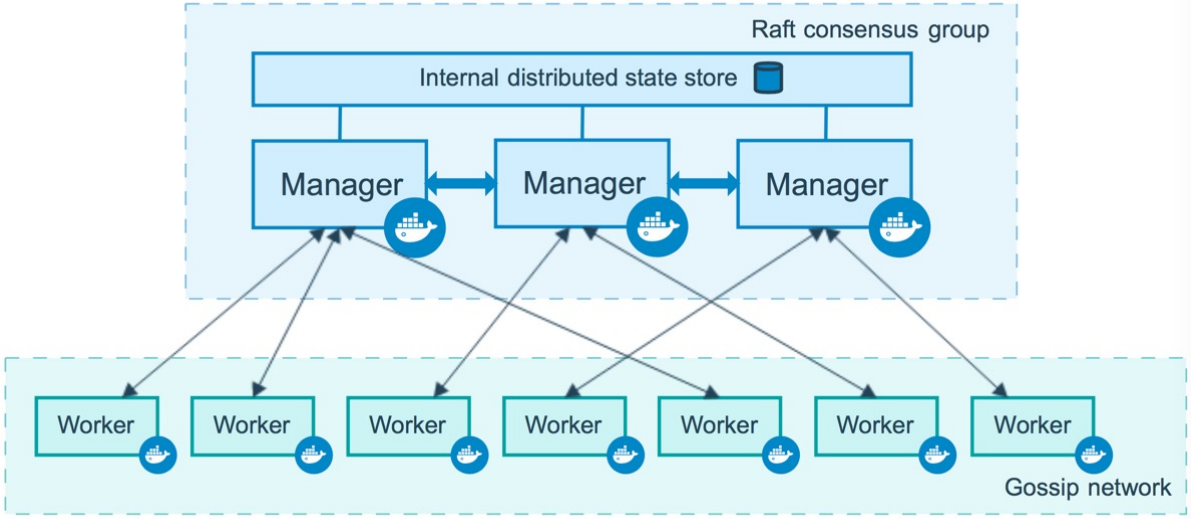
Docker Swarm

- Swarm **orchestra un cluster di istanze Docker**
  - Le funzionalità di gestione e orchestrazione del cluster sono incorporate nel Docker engine
- Uno swarm è costituito da **più host Docker che vengono eseguiti in modalità swarm**
  - Agiscono come **manager** (per gestire il join e la delega allo swarm)
  - o **worker** (che eseguono i servizi dello swarm)
- Un dato host Docker può essere un manager, un worker o svolgere entrambi i ruoli

Swarm managers e tasks

I nodi manager implementano l'**algoritmo di consenso Raft** per gestire lo **stato globale del cluster**

- Tollera fino a **(N-1)/2 fallimenti di un Manager**
  - **Se il manager di uno swarm a singolo manager si guasta, i servizi continuano a funzionare, ma è necessario creare un nuovo cluster per recuperare**
- Richiede una **maggioranza** o un **quorum** di (N/2)+1 membri per accordarsi sui valori



Servizio Swarm

- Per un servizio possiamo **definire il suo stato ottimale (desiderato)**
  - Numero di repliche
  - Risorse di rete e di storage disponibili
  - Porte che il servizio espone al mondo esterno
  - Vincoli di risorse e preferenze di posizionamento
  - etc.
- Docker lavora per **mantenere lo stato desiderato**
  - Ad esempio, se un nodo worker diventa indisponibile, Docker pianifica le attività di quel nodo su altri nodi

Deploy di un servizio con Docker Swarm

- Inviare una **definizione di servizio** a un nodo manager
- Il nodo manager distribuisce ai nodi worker unità di lavoro chiamate **task**
- I nodi manager eseguono anche le funzioni di **orchestrazione** e **gestione del cluster** necessarie per mantenere lo stato desiderato dello swarm
  - I nodi **manager** eleggono un **singolo leader** per condurre le attività di orchestrazione.
  - I nodi **worker ricevono ed eseguono i task** inviati dai nodi manager
  - **Un agente viene eseguito su ogni nodo worker** e riferisce sui compiti ad esso assegnati.

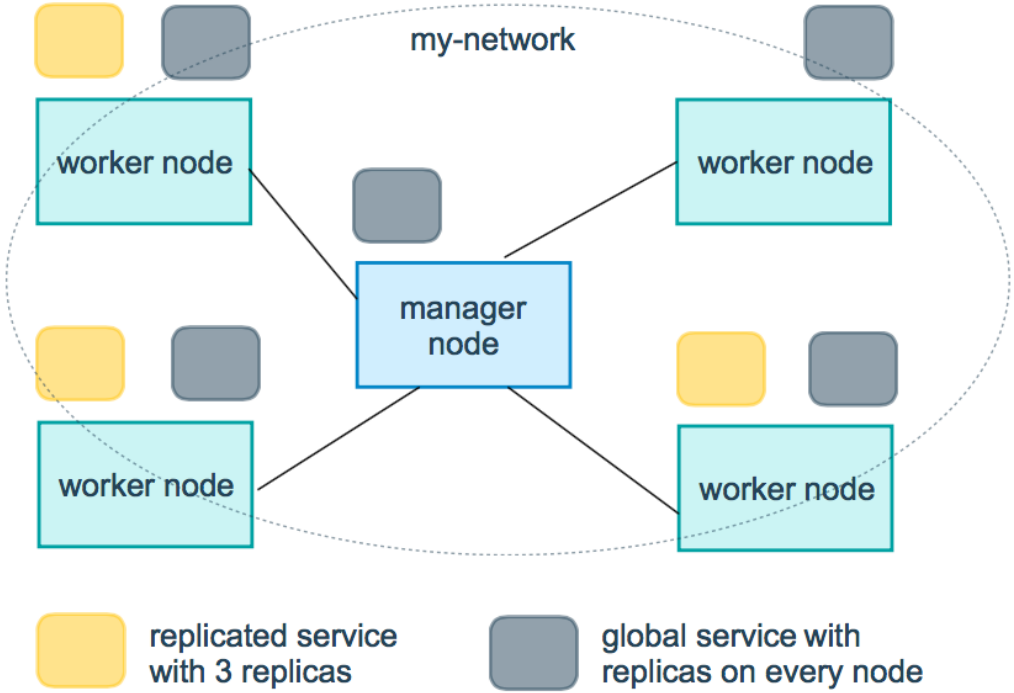
Tipi di servizio in Docker Swarm

- Un servizio è la **definizione dei task da eseguire** sui nodi manager o worker

  - **Modello di servizi replicati**: lo swarm manager distribuisce un **numero specifico di task di replica** tra i nodi in base alla scala impostata nello stato desiderato

  - **Modello di servizi globali**: lo swarm esegue un task del servizio su ogni nodo disponibile nel cluster. Quando si aggiunge un nodo a allo swarm, viene assegnato un task al nuovo nodo

Servizio Replicato vs. Globale in Swarm



Controllo del *placement* del servizio

- Docker Swarm fornisce meccanismi per controllare la scala e il posizionamento (placement) dei servizi su nodi diversi
  - Servizio replicato o globale
  - Riserva di memoria o CPU per un servizio
  - Vincoli di placement
    - Il servizio viene eseguito solo su nodi con un'etichetta specifica
  - Preferenze di placement
    - Cerca di posizionare i task sui nodi appropriati in modo uniforme

Swarm tasks

- Un task **implementa** un container Docker e i comandi da eseguire all'interno del container
- È l'**unità atomica di scheduling** in Docker Swarm
- I **nodi manager assegnano i task ai nodi worker** in base al numero di repliche impostato nel parametro di scala nella definizione del servizio
- Una volta che un task è assegnato a un nodo, **non può spostarsi su un altro nodo**. Può essere eseguito solo sul nodo assegnato o fallire

Service update in Docker Swarm

- Permette gli aggiornamenti online della configurazione di rete e dei volumi legati ad un servizio
  - Docker Swarm avvierà nuovi nodi con la nuova configurazione
  - Arresta automaticamente i vecchi nodi utilizzando la vecchia configurazione
  - Gestione dei rollback e degli aggiornamenti dei servizi

Utilizzo di Docker Swarm

- **Creazione dello swarm**
  - `$ docker swarm init --advertise-addr IP_NODO_MASTER`
- **Join allo swarm**
  - `$ docker swarm join --token TOKEN_GENERATO IP_NODO_MASTER:2377`
  - **P.S.: Utilizzare comando generato dallo swarm init**
- **Creazione/Rimozione del servizio**
  - **`docker service`**
    - `$ docker service create ...`
      - E' possibile specificare il grado di replicazione, e altri parametri
    - `$ docker service rm ...`
  - **`docker stack`**
    - `$ docker stack deploy --compose-file=compose.yaml ...`
      - E' possibile specificare un file yaml per le caratteristiche del servizio
    - `$ docker stack rm ...`
- **Rimozione dello swarm**
  - `$ docker swarm leave`

Esempio. Orchestrazione di un app Flask replicata

1. Creazione dello swarm
2. Join allo swarm
3. Build dell'immagine su tutti i nodi dello swarm (vede slide precedenti)
4. Creazione del servizio (**docker service**)
   - `docker service create --replicas 3 --name flask_helloworld_service --publish 5001:5001 flask_image_hello_world`

```
image flask_image_hello_world:latest could not be accessed on a registry to record
its digest. Each node will access flask_image_hello_world:latest independently,
possibly leading to different nodes running different
versions of the image.

xvymsur4ggtblw0y2sheuub2z
overall progress: 3 out of 3 tasks
1/3: running   [==================================================>]
2/3: running   [==================================================>]
3/3: running   [==================================================>]
verify: Service converged
root@dockertest1:~/ACP/docker_deploy_examples/dockerized_flask_service/app#
```

Esempio. Orchestrazione di un app Flask replicata

```
~ while True; do curl http://192.168.100.101:5001/; echo; sleep 1; done
This is an example wsgi app served from 697b6c7a50d3 to 10.0.0.2
This is an example wsgi app served from 1b139379fb74 to 10.0.0.2
This is an example wsgi app served from 8df0ff0e9494 to 10.0.0.2
This is an example wsgi app served from 697b6c7a50d3 to 10.0.0.2
This is an example wsgi app served from 1b139379fb74 to 10.0.0.2
This is an example wsgi app served from 8df0ff0e9494 to 10.0.0.2
This is an example wsgi app served from 697b6c7a50d3 to 10.0.0.2
This is an example wsgi app served from 1b139379fb74 to 10.0.0.2
This is an example wsgi app served from 8df0ff0e9494 to 10.0.0.2
This is an example wsgi app served from 697b6c7a50d3 to 10.0.0.2
This is an example wsgi app served from 1b139379fb74 to 10.0.0.2
```

- Le richieste di GET verso il nodo manager sono bilanciate tra tutti i nodi che hanno un container in esecuzione per il nostro servizio
- Ad ogni GET risponde uno dei 3 container in esecuzione, permettendo sia un bilanciamento delle richieste che una disponibilità (availability) del servizio

Esempio. Orchestrazione di un app Flask replicata

- L’availability di 3 repliche è garantita automaticamente dallo swarm se uno o più nodi falliscono
  - E’ possibile emulare una non disponibilità di un nodo impostando lo stato di **DRAIN** per quel nodo
  - `$ docker node update --availability drain IP_HOST`

Esempio. Orchestrazione di un app Flask replicata

- E’ possibile effettuare il deploy dell’applicazione anche attraverso **`docker stack`** che utilizza un file di configurazione `.yaml`
- Il comando **deploy** di docker stack accetta una descrizione dello stack sotto forma di file Compose

```
...
deploy:
     replicas: 5
     restart_policy:
       condition: on-failure
       max_attempts: 3
       window: 120s
...
```

Esempio. Orchestrazione di un app Flask replicata

- Per fare il deploy basta eseguire creare l'immagine del serivizio in tutti i nodi del cluster e lanciare il deploy

```
# // On all worker nodes
# docker build –t /PATH/TO/flask_image_hello_world_DOCKERFILE

# // On master node
# docker stack deploy ––compose–file=/PATH/TO/compose_with_stack.yaml
dockerized_flask_service_with_stack
Creating network dockerized_flask_service_replicated_default
Creating service dockerized_flask_service_replicated_web
```