LabSO 2024

Laboratorio Sistemi Operativi - Unitn A.A. 2023-2024

Michele Grisafi - michele.grisafi@unitn.it

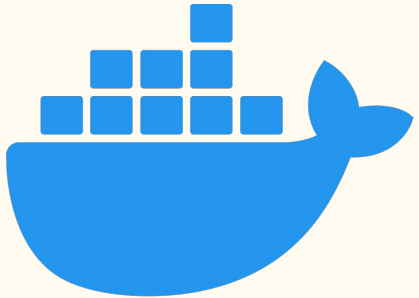
Docker

Docker, cos'è?

Tecnologia di virtualizzazione a livello del sistema operativo che consente la creazione, la gestione e l'esecuzione di **applicazioni** attraverso containers.

I **containers** sono ambienti leggeri, dinamici ed isolati che vengono eseguiti sopra il kernel di Linux.

... e molti altri!



docker

Docker Container

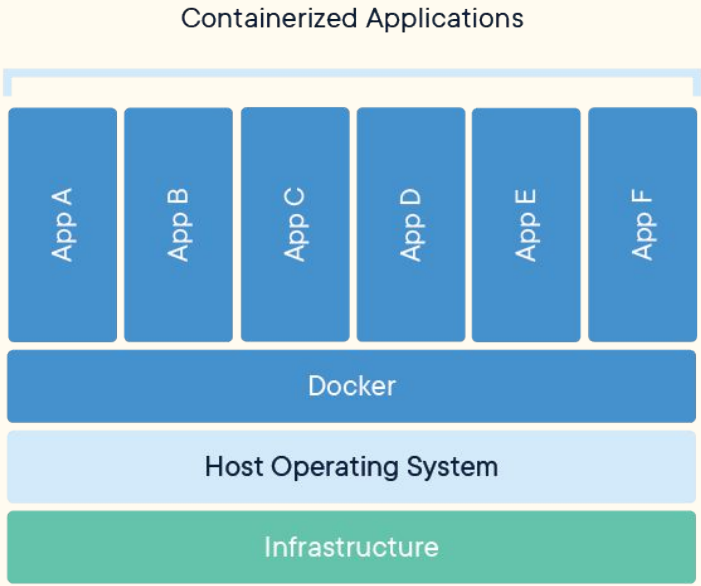
- Virtualizzazione a livello OS
- Containers condividono kernel
- Avvio e creazione in secondi
- Leggere (KB/MB)
- Utilizzo leggero di risorse
- Si distruggono e si rieseguono
- Minore sicurezza

NB: Basati su immagini delle quali se ne trovano tantissime già pronte!

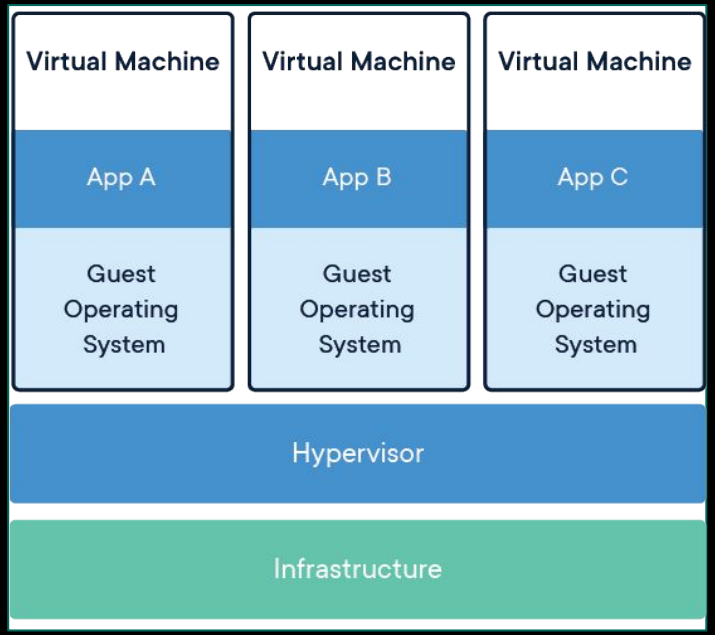
Virtual Machine

- Virtualizzazione a livello HW
- Ogni VM ha il suo OS
- Avvio e creazione in minuti
- Pesanti (GB)
- Utilizzo intenso di risorse
- Si trasferiscono
- Maggiore sicurezza
- Maggiore controllo

Docker Container

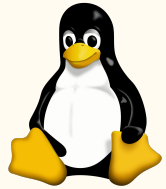


Virtual Machine

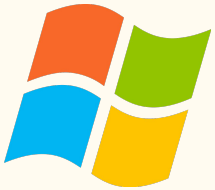


Compatibilità sui vari OS

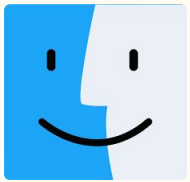
Linux: docker gestisce i containers usando il kernel linux nativo.



Windows: docker gestisce i containers usando il kernel linux tramite WSL2 (originariamente virtualizzato tramite Hyper-V). Gestito da un'applicazione.



Mac: docker gestisce i containers usando il kernel linux virtualizzato tramite xhyve hypervisor. Gestito da un'applicazione.



Containers e immagini

Un'immagine docker è un insieme di "istruzioni" per la creazione di un container. Essa consente di raggruppare varie applicazioni ed eseguirle, con una certa configurazione, in maniera rapida attraverso un container.

I containers sono invece gli ambienti virtualizzati gestiti da docker, che possono essere creati, avviati, fermati ed eliminati. I container devono essere basati su un'immagine!

(esiste un'opzione di creazione da zero - FROM scratch - che però è raramente usata in pratica)

Gestione dei containers

Al contrario delle VMs, i containers sono elementi quasi effimeri: è normale eliminarli e ricrearli. È possibile creare ed eseguire un nuovo container da un'immagine con:

```
docker run [options] <image>
```

Al container viene assegnato un ID ed un nome univoco. Una volta creato, un container può essere arrestato e riattivato con:

```
docker start/stop <ID/name of container>
```

Per eliminare un container inattivo, è possibile usare:

```
docker rm <ID/name of container>
```

Gestione dei containers

Per vedere i containers attivi è possibile usare:

`docker container ls [options]` `-a`: mostra anche inattivi

Mentre per visualizzare le statistiche di utilizzo delle varie risorse esiste il comando:

`docker stats`

Infine, per interagire con un container in esecuzione, ed eseguire un certo comando, si può usare:

`docker exec [options] <container> <command>`

E la panacea di tutti i dubbi... `docker <command> --help`

Parametri ‘run’ opzionali

- `--name <nome>`: assegna un nome specifico al container
- `-d`: detach mode → scollega il container (ed il suo input/output) dalla console*
- `-ti`: esegue container in modalità interattiva*
- `--rm`: elimina container all’uscita
- `--hostname <nome>`: imposta l’hostname nel container
- `--workdir <path>`: imposta la cartella di lavoro nel container
- `--network host`: collega il container alla rete locale **
- `--privileged`: esegue il container con i privilegi dell’host

*Per collegarsi `docker attach <container>`. Per scollegarsi *`Ctrl+P, Ctrl+Q`*

** la modalità host non funziona su W10 e MacOS a causa della VM sottostante

Esempi

- Esegui “`docker run hello-world`”
- Esegui “`docker run -d -p 80:80 docker/getting-started`” e collegati alla pagina “localhost:80” con un qualunque browser

Gestione delle immagini

La community di docker offre migliaia di immagini pronte all'uso (ma è sempre possibile crearne di nuove)

`docker images`: mostra le immagini salvate

`docker rmi <imageID>`: elimina un'immagine (se non è in uso!)

`docker search <keyword>`: cerca un'immagine nella repository di docker

`docker commit <container> <repository/imageName>`: crea una nuova immagine dai cambiamenti nel container

Altrimenti si possono creare nuove immagini con dei dockerfile...

Dockerfile

I dockerfile sono dei documenti testuali che raccolgono una serie di comandi necessari alla creazione di una nuova immagine. Ogni nuova immagine sarà generata a partire da un'immagine di base, come Ubuntu o l'immagine minimale 'scratch'. La creazione a partire da un docker file viene gestita attraverso del caching che ne permette la ricompilazione rapida in caso di piccoli cambiamenti.

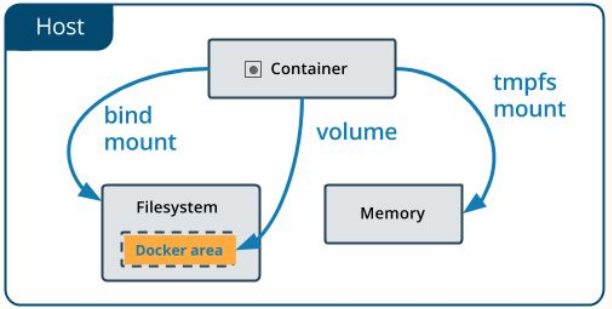
```
FROM ubuntu:22.04
RUN apt-get update && apt-get install build-essential nano -y
RUN mkdir /home/labOS
CMD cd /home/labOS && bash
```

```
docker build -t labos/ubuntu - < dockerfile
```

Gestione dei volumi

Tutto ciò che viene scritto su un container è legato alla vita di quel container. Una volta eliminato (prassi comune) gli eventuali dati vengono persi.

Docker salva i file persistenti su *bind mount* o su dei *volumi*. Sebbene i **bind mount** siano strettamente collegati con il filesystem dell'host OS, consentendo dunque una facile comunicazione con il containers, i **volumi** sono ormai lo standard in quanto indipendenti, facili da gestire e più in linea con la filosofia di docker.



Sintassi dei comandi

`docker volume create <volumeName>`: crea un nuovo volume

`docker volume ls`: mostra i volumi esistenti

`docker volume inspect <volumeName>`: esamina volume

`docker volume rm <volumeName>`: rimuovi volume

`docker run -v <volume>:</path/in/container> <image>` : crea un nuovo container con il **volume** specificato montato nel percorso specificato

`docker run -v <pathHost>:<path/in/container> <image>` : crea un nuovo container con un **bind mount** specificato montato nel percorso specificato

Il nostro ambiente

```
docker run -ti --rm --name="labOS" --privileged      \
-v /:/host -v "$(pwd):/home/labOS"   \
--hostname "labOS" --workdir /home/labOS    \
ubuntu:22.04 /bin/bash
```

Ed eseguire:

```
apt-get update && apt-get install -y nano build-essential
```

Quando il container è pronto si può fare il commit per salvare le modifiche in una nuova immagine (es.: `docker commit ..... localhost.ext/unitn:labso2023`)

NB: se non aggiungete il flag `--rm`, ogni volta che uscite il container verrà fermato e potrà essere riavviato con `docker start labOS`

Il nostro ambiente... oppure

Usare il dockerfile in slide #12 per creare un'immagine del laboratorio:

```
docker build -t labOS/ubuntu - < dockerfile
```

E poi è possibile eseguire un container basato sulla nuova immagine 'labOS/ubuntu': usare il comando

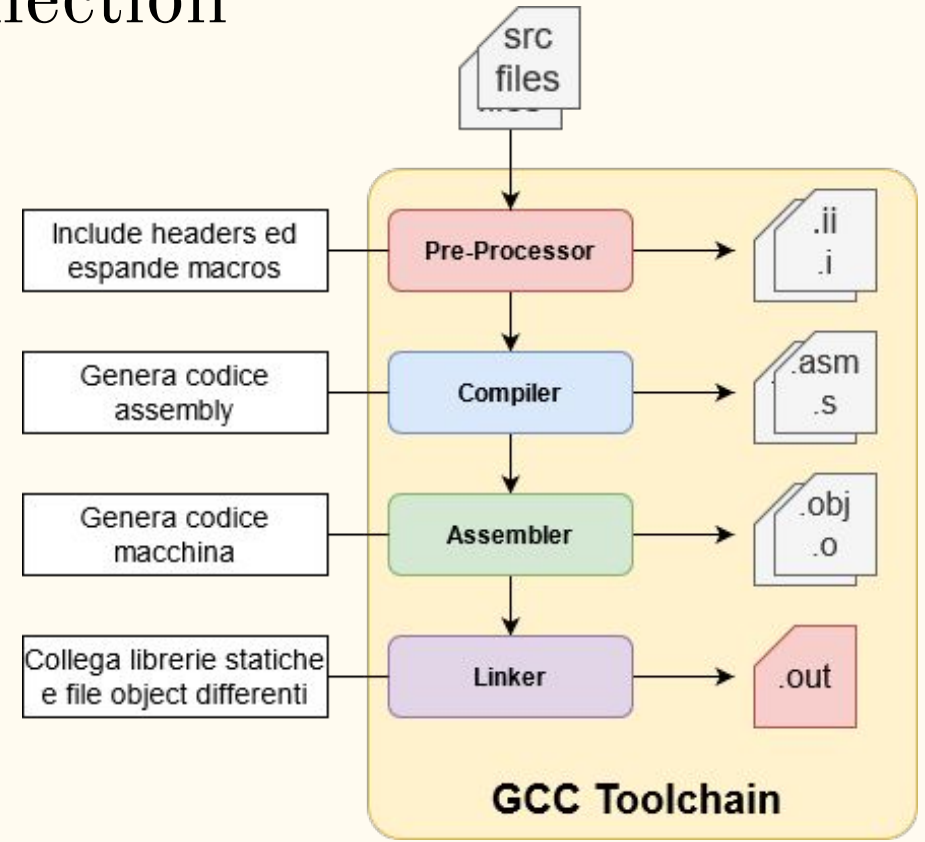
```
docker run -ti --rm --name="labOS" --privileged      \
-v /:/host    -v "$(pwd):/home/labOS"    \
--hostname "labOS" labos/ubuntu
```

GCC
GNU compiler collection

GCC = Gnu Compiler Collection

Insieme di strumenti open-source che costituisce lo standard per la creazione di eseguibili su Linux.

GCC supporta diversi linguaggi, tra cui C, e consente la modifica dei vari passaggi intermedi per una completa personalizzazione dell'eseguibile.



La compilazione

Gli strumenti GCC possono essere chiamati singolarmente:

```
gcc -E <sorgente.c> -o <preProcessed.ii|.i>
```

```
gcc -S <preProcessed.i|.ii> -o <assembly.asm|.s>
```

```
gcc -c <assembly.asm|.s> -o <objectFile.obj|.o>
```

```
gcc <objectFile.obj|.o> -o <executable.out>
```

NB: l'input di ogni comando può essere il file sorgente, e l'ultimo comando è in grado di creare direttamente l'eseguibile.
NB: l'assembly ed il codice macchina generato dipendono dall'architettura di destinazione

GCC warnings and errors

Quando il file sorgente contiene un errore, il compilatore lo rileva e fa fallire la compilazione. Tuttavia, è molto più comune che vengano generati dei warnings.

Un warning comunica una condizione particolare che non impedisce il funzionamento del codice ma che potrebbe compromettere le funzionalità. Esempi sono:

- Uso improprio di variabili
- Dead code
- Conversioni implicite
- Format string sbagliati
- Funzioni implicite

È buona prassi gestire ogni warning!

Un esempio

1. Provate a compilare una semplicissima applicazione, invocando ogni step singolarmente osservandone l'output.
2. Provate ad aggiungere `#include <stdio.h>` ad inizio file e ripetete il tutto

```c
//main.c
void main(){
    return;
}
```

```c
//main.c
#include <stdio.h>
void main(){
    return;
}
```

Make

Make tool

Il Make tool è uno strumento della collezione GNU che può essere usato per gestire la compilazione *automatica* e *selettiva* di grandi e piccoli progetti. **Make** consente di specificare delle dipendenze tra i vari file, per esempio consentendo solo la compilazione di librerie i cui sorgenti sono stati modificati.

Make può anche essere usato per gestire il deployment di un'applicazione, assumendo alcune delle capacità di uno script bash.

Makefile

Make può eseguire dei makefiles, dei file testuali che contengono tutte le direttive utili alla compilazione di un'applicazione (o allo svolgimento di un altro task).

```
make -f makefile
```

In alternativa, il comando `make` senza argomenti processerà il file `'makefile'` presente nella cartella di lavoro (nell'ordine cerca: `GNUmakefile`, `makefile` e `Makefile`)

Makefiles secondari possono essere inclusi nel makefile principale con delle direttive specifiche, consentendo una gestione più articolata di grandi progetti.

Target e ricette

Una *ricetta o regola* è una lista di comandi shell che vengono eseguiti indipendentemente dal resto del makefile: vengono eseguiti in una sottoshell diversa, e non possono comunicare tra di loro.

Le ricette sono raggruppate all'interno di un **target** che può essere eseguito da linea di comando quando viene invocato make:

```
make -f makefile <target1> <target2> ...
```

```
target:
→   recipe/rule
→   recipe/rule
...
```

Prequisiti e catene

Ogni target può specificare dei **prerequisiti**, ovvero delle altre ricette che devono essere eseguite affinchè le regole del target vengano anch'esse eseguite. Un prerequisito può essere esso stesso un target!

L'ordine di esecuzione dei prerequisiti non è definito.

```
target1: target2 target3
    rule (3)
    rule (4)
    ...

target2: target3
    rule (1)

target3:
    rule (2)
```

Makefile e la compilazione

Il Makefile è pensato per la gestione della compilazione di grandi progetti. Ogni target è generalmente il nome di un file, e le regole sotto di esso i comandi per la generazione di quel file.

Questo consente a make di saltare l'esecuzione dei prerequisiti già presenti nel filesystem. In caso il prerequisito non dovesse esistere sul file system e qualora non vi fosse associata alcun target, il make uscirà con errore.

```
bin.out: main.o lib.o
    gcc main.o lib.o -o bin.out

main.o: main.c
    gcc -c main.c -o main.o

lib.o: lib.c
    gcc -c lib.c -o lib.o
```

Sintassi

Un makefile è un file di testo in cui righe vuote e commenti (preceduti da #) vengono ignorati. Commenti all'interno di ricette sono ignorati dalla sottoshell, non da make.

Le ricette **DEVONO** iniziare con un carattere di **TAB** (**non spazi**).

Una ricetta preceduta da @ non viene visualizzata in output, altrimenti i comandi sono visualizzati e poi eseguiti (ed il loro risultato stampato).

Esistono costrutti più complessi per necessità particolari (ad esempio costrutti condizionali).

Target speciali

Se non viene passato alcun target al comando make, viene eseguito il primo disponibile. Esistono poi dei target che hanno un significato e comportamento speciale.

`.INTERMEDIATE` e `.SECONDARY:` hanno come prerequisiti i target “intermedi”. Nel primo caso sono poi rimossi, nel secondo sono mantenuti a fine esecuzione .

`.PHONY:` ha come prerequisiti i target che non corrispondono a dei files, o comunque da eseguire “sempre” senza verificare l’eventuale file omonimo.

In un target, `%` sostituisce qualunque stringa. In un prerequisito corrisponde alla stringa sostituita nel target.

```
target: prerequisite
→   rule
→   rule
    ...
```

```
all: ...
  rule

.SECONDARY: target1 ..

.PHONY: target2 …

%.s: %.c
  #prova.s: prova.c
  #src/h.s: src/h.c
```

Variabili utente e automatiche

```
ONCE:=hello $(LATER)
EVERY=hello $(LATER)
LATER=world

target1:
  echo $(ONCE) # ‘hello’
  echo $(EVERY) # ‘hello world’
```

Le variabili utente si definiscono con la sintassi

- `nome:=valore`
- `nome=valore`

Le prime vengono popolate al momento della dichiarazione, le seconde al momento della lettura. La lettura è eseguita con “`$(nome)`”. Inoltre, possono essere sovrascritte da riga di comando con `make nome=value`

Per usare il carattere `$` nella sottoshell delle regole, è necessario precederlo da un altro carattere `$`. Per esempio:

```
echo $$$ ; var = ciao ; echo $$var
```

Variabili automatiche

Le variabili automatiche possono essere usate all'interno delle regole per riferirsi ad elementi specifici relativi al target corrente. Esempi sono `$@`, `$^`, `$<`

```
target: pre1 pre2 pre3
  echo $@ is 'target'
  echo $^ is 'pre1 pre2 pre3'
  echo $< is 'pre1'
```

La variabile `SHELL` può essere usata per specificare la shell di riferimento per l'esecuzione del makefile. Per esempio, `SHELL=/bin/bash` abilita bash anzichè SH.

Funzioni speciali

`$(eval …)`: consente di creare nuove regole make dinamiche.

`$(shell …)`: cattura l'output di un commando shell.

`$(wildcard *)`: crea un elenco di file che corrispondono alla stringa specificata.

```
LATER=hello
PWD=$(shell pwd)
OBJ_FILES:=$(wildcard *.o)

target1:
  echo $(LATER) #hello
  $(eval LATER += world)
  echo $(LATER) #hello world
```

Make file - Esempi

```
all: main.out
     @echo “Application compiled”

%.s: %.c
     gcc -S $< -o $@

%.out: %.s
     mkdir -p build
     gcc $< -o build/$@

clean:
     rm -rf build *.out *.s

.PHONY: clean

.SECONDARY: make.s
```

```
FILES=$(subst .c,.out,$(wildcard *.c))

all: $(FILES)
     @echo all files built
     @chmod +x $(FILES)

%.out: %.c
     @echo building file $<
     @gcc $< -o $@

clean:
     @rm *.out
```

Esercizio per casa

Creare un makefile con una regola help di default che mostri una nota informativa, una regola backup che crei un backup di una cartella appendendo ".bak" al nome e una restore che ripristini il contenuto originale. Per definire la cartella sorgente passarne il nome come variabile, ad esempio:

```
make -f mf-backup FOLDER=...
```

(la variabile FOLDER è disponibile dentro il makefile)

CONCLUSIONI

Docker, GCC e make possono essere utilizzati per la gestione delle varie applicazioni in C. Ognuno di questi strumenti non è indispensabile ma permette di creare un flusso di lavoro coerente e strutturato.