LabSO 2022

Laboratorio Sistemi Operativi - A.A. 2021-2022

dr. Andrea Naimoli	Informatica LT andrea.naimoli@unitn.it
dr. Michele Grisafi	Ingegneria informatica, delle comunicazioni ed elettronica (LT) michele.grisafi@unitn.it

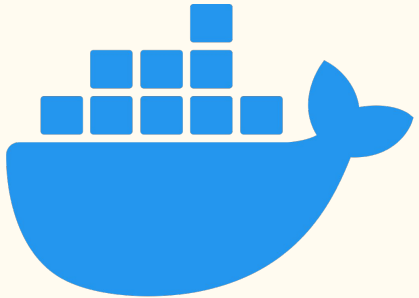
docker

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

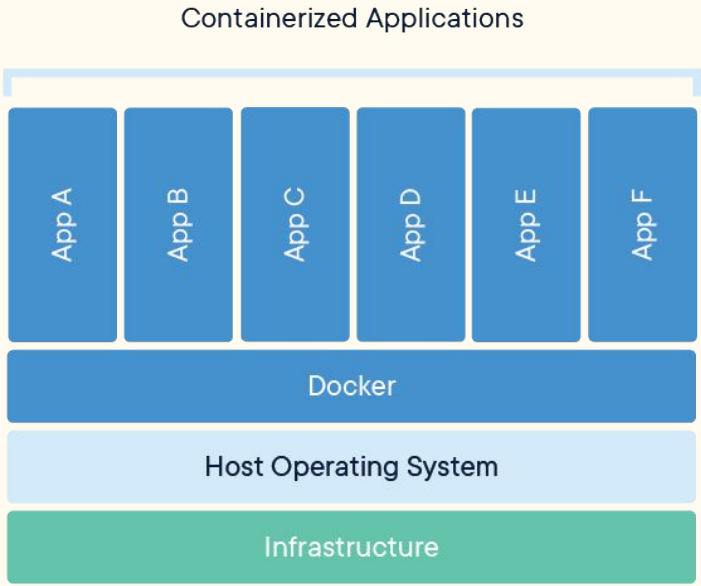
- Virtualizzazione a livello OS
- Containers condividono kernel
- Avvio e creazione in secondi
- Leggere (KB/MB)
- Utilizzo leggero di risorse
- Si distruggono e si rieseguono
- Minore sicurezza

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

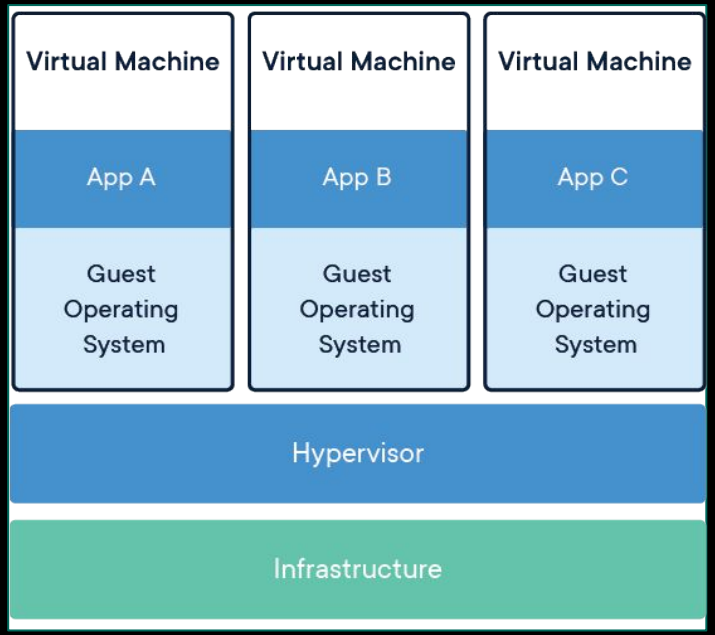
Virtual Machine

- 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

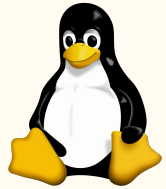


Virtual Machine

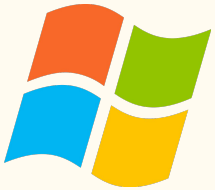


Compatibilità sui vari OS

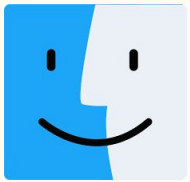
Linux: docker gestisce i containers usando il kernel linux nativo.



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

`docker run [options] <image>`: crea un nuovo container da un'immagine

`docker container ls [options]`: mostra i containers attivi ([-a] tutti)

`docker start/stop <container>`: avvia/ferma l'esecuzione del container

`docker exec [options] <container> <command>`: esegue il comando all'interno del container

`docker stats`: mostra le statistiche di utilizzo dei containers

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 possibile crearne di nuove.

`docker images`: mostra le immagini salvate

`docker rmi <imageID>`: elimina un'immagine (se non 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

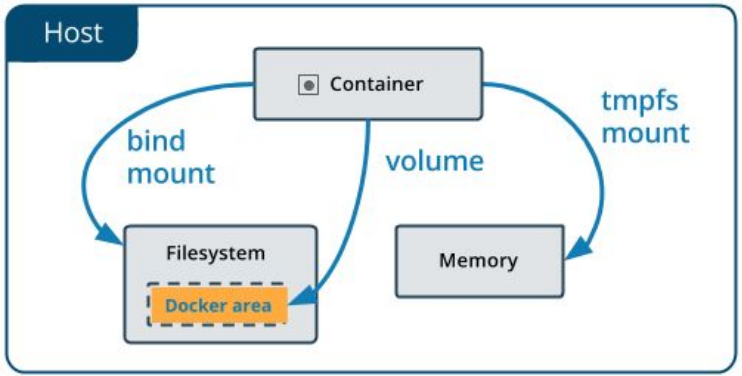
I dockerfile sono dei documenti testuali che raccolgono una serie di comandi necessari alla creazione di una nuova immagine. Ogni nuova immagine sarà generata a partire da un'immagine di base, come Ubuntu o l'immagine minimale 'scratch'. La creazione a partire da un docker file viene gestita attraverso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

```
FROM ubuntu:20.04
RUN apt update && apt install build-essential nano -y
RUN mkdir /home/LabOS
CMD cd /home/labOS && bash
```

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

Gestione dei volumi

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:20.04 /bin/bash
```

Ed eseguire:

```
apt update && apt 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:labso2022`)

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

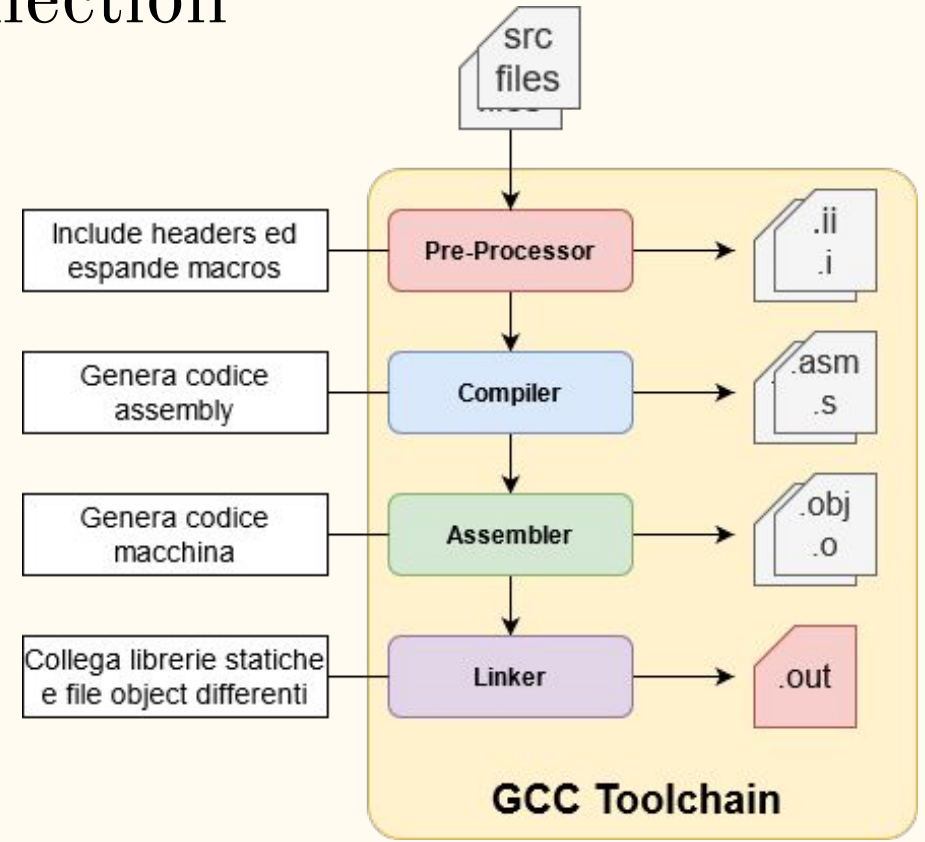
```
docker run -ti --rm --name="labOS" --privileged      \
-v /:/host    -v "$(pwd):/home/labOS"    \
--hostname "labOS" labos/ubuntu
```

gcc

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

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 i quali 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, prerequisite and recipes

Una *ricetta* è una lista di comandi bash che vengono eseguiti indipendentemente dal resto del makefile.
I *target* sono generalmente dei files generati da uno specifico insieme di regole.
Ogni target può specificare dei *prerequisiti*, ovvero degli altri file che devono esistere affinchè le regole di un target vengano eseguite. Un prerequisito può essere esso stesso un target!

L'esecuzione di un file make inizia specificando uno o più target `make -f makefile target1 ...` e prosegue a seconda dei vari prerequisiti.

```
target: prerequisite
→   recipe
→   recipe
    ...
```

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

target2: target3
    rule (1)

target3:
    rule (2)
```

Sintassi

Un makefile è un file di testo “plain” in cui righe vuote e parti di testo dal carattere “**#**” fino alla fine della riga non in una ricetta (considerato un commento: sempre che non sia usato l’escaping con “**\#**” o che compaia dentro una stringa con **'** o **"**) sono ignorati.

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

Una ricetta che (a parte il TAB) inizia con @ non viene visualizzata in output, altrimenti i comandi sono visualizzati e poi eseguiti.

Una riga con un singolo TAB è una ricetta vuota.

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

Target speciali

Il target di default eseguito quando non ne viene passato alcuno è il primo disponibile.

`.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: target1 …

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

Variabili utente e automatiche

Le variabili utente si definiscono con la sintassi `nome:=valore` o `nome=valore` e vengono usate con `$(nome)`. Inoltre, possono essere sovrascritte da riga di comando con `make nome=value`.

Le variabili automatiche possono essere usate all'interno delle regole per riferirsi ad elementi specifici relativi al target corrente.

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

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

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

Funzioni speciali

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

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

`$(wildcard *)`: restituisce 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 - Esempio

```
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
```

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)

C

Perchè C?

- Struttura minimale
- Poche parole chiave (con i suoi pro e contro!)
- Unix compliant, alla base di Unix, nato per scrivere Unix
- Organizzato a passi, con sorgente, file intermedi ed eseguibile finale
- Disponibilità di librerie conosciute e standard
- Nessuna struttura di alt(issim)o livello, come classi o altro
- Efficiente perché di basso livello
- Pieno controllo del programma e delle sue risorse

Direttive

Il compilatore, nella fase di preprocessing, elabora tutte le direttive presenti nel sorgente. Ogni direttiva viene introdotta con '#' e può essere di vari tipi:

- `#include <lib>`: copia il contenuto del file *lib* (cercando nelle cartelle delle librerie) nel file corrente
- `#include "lib"`: come sopra ma cerca prima anche nella cartella corrente
- `#define VAR VAL`: crea una costante VAR con il contenuto VAL, e sostituisce ogni occorrenza di VAR con VAL.
- `#define MUL(A,B) A*B`: dichiara una funzione con parametri A e B. Queste funzioni hanno una sintassi limitata!
- `#ifdef, #ifndef, #if, #else, #endif`: rende l'inclusione di parte di codice dipendente da una condizione.

Macro possono essere passate a GCC con `-D NAME=VALUE`

Tipi e Casting

C è un linguaggio debolmente tipizzato che utilizza 8 tipi fondamentali. È possibile fare il casting tra tipi differenti:

```
float a = 3.5;
int b = (int)a;
```

La grandezza delle variabili è **dipendente dall'architettura di riferimento** i valori massimi per ogni tipo cambiano a seconda se la variabile è *signed* o *unsigned*.

- void (0 byte)
- char (1 byte)
- short (2 bytes)
- int (4 bytes)
- float (4 bytes)
- long (8 bytes)
- double (8 bytes)
- long double (8 bytes)

NB: non esiste il tipo boolean, ma viene spesso emulato con un char.

Array e stringhe

C supporta l'uso di stringhe che, tuttavia, corrispondono a degli array di caratteri.

```
int nome[DIM];
long nome[] = {1,2,3,4};
char string[] = “ciao”;
char string2[] = {‘c’,’i’,’a’,’o’};
nome[0] = 22;
```

Gli array sono generalmente di dimensione statica e non possono essere ingranditi durante l'esecuzione del programma. Per array dinamici dovranno essere usati costrutti particolari (come malloc).

Le stringhe, quando acquisite in input o dichiarate con la sintassi `"stringa"`, terminano con il carattere '`\0`' e sono dunque di grandezza *str_len+1*

Puntatori di variabili

C si evolve attorno all'uso di puntatori, ovvero degli alias per zone di memorie condivise tra diverse variabili/funzioni. L'uso di puntatori è abilitato da due operatori: '*' ed '&'.

'*' ha significati diversi a seconda se usato in una dichiarazione o in un'assegnazione:

`int *punt;` Crea un puntatore ad intero

`int valore = *(punt);` Ottiene valore puntato

'&' ottiene l'indirizzo di memoria in cui è collocata una certa variabile.

`long whereIsValore = &valore;`

Esempi:

```
float   pie   =   3.4;
float  *pPie  =  &pie;
pie       *=       2;
float pie4 = *pPie * 2;

char  *array  =  "str";
*array     =      's';
array[1]   =      't';
*(array + 2) = 'r';
```

Puntatori di variabili

```
int      i      =      42
int   *   punt   =   &i;
int b = *(punt);
```

Tipo	Nome	Valore	Indirizzo
int	i	42	0xaaaabbbb
int *	punt	0xaaaabbbb	0xccccdddd
int	b	42	0x11112222

```
i = 20;
```

Nome	Valore
i	**20**
punt	0xaaaabbbb
b	**?**

Puntatori di funzioni

```
#include <stdio.h>
float xdiv(float a, float b) {
  return a/b;
}
float xmul(float a, float b) {
  return a*b;
}
void main() {
  float (*punt)(float,float);
  punt = xdiv;
  float res = punt(10,10);
  punt = &xmul;
  res = (*punt)(10,10);
  printf("%f\n", res);
}
```

Main.c

```
#include <stdlib.h>
#include <stdio.h>
#define DIVIDENDO 3

int division(float var1, float var2, float *result){
        *result = var1/var2;
        return 0;
}
int main(int argc, char * argv[]){
        float var1 = atof(argv[1]);
        float result = 0;
        division(var1, DIVIDENDO, &result);
        printf("%f \n", result);
}
```

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.