Docker e orchestration tools

Distributed Systems / Technologies

D'Errico Christian
christian.derrico@unibo.it

Dipartimento di Informatica – Scienza e Ingegneria (DISI)
Alma Mater Studiorum – Università di Bologna a Cesena

Academic Year 2020/2021

Outline

Next in Line...

Obiettivi

- Queste slides nascono con lo scopo di presentare Docker e comprendere come la più famosa suite per lo sviluppo del software basato sui microservizi e le tecnologie che vi orbitano attorno possano fornire il loro contributo nello sviluppo su larga scala di architetture distribuite.

- Saranno analizzati e trattati i concetti di:
  - → Docker Image
  - → Container
  - → Service
  - → Stack
  - → Swarm/Cluster

- Parleremo dei due principali tool di orchestrazione basati su tecnologie Docker:
  - Docker Swarm
  - Kubernetes

Next in Line...

Container

Non possiamo parlare di Docker senza definire preliminarmente cosa intendiamo quando utilizziamo la parola container in un qualsiasi contesto in cui adoperiamo anche parole come "macchina virtuale" o "virtualizzazione".

Infatti, quando si parla di virtualizzazione di risorse, definiamo container una partizione del sistema operativo corrente, su cui installare ed eseguire applicazioni che rimangono isolate, pur accedendo ai servizi di uno stesso o.s. sottostante.

- Concettualmente, un container è pensato per pacchettizzare una singola applicazione, rendendola pronta per il deployment.
- Costruito e personalizzato, un container può essere replicato più volte, su una stessa macchina host o su macchine diverse, purchè su di esse giri un Container Engine
- un Container Engine fornisce le funzionalità di base per la gestione dei container: interfacciandosi con il sistema operativo sottostante, espone API che permettono di regolare il ciclo di vita di tali sistemi.

Perché i container?

- Concepiti per essere funzionalmente atomici, i container possono essere combinati nella costruzione di infrastrutture composte da tanti microcomponenti riutilizzabili, ciascuno dei quali isolato in una singola partizione.
- Più container, organizzati con una certa coesione operativa, costituiscono gruppi funzionali denominati stacks, i quali a loro volta vengono gestiti da orchestrators.
- I container offrono basso overhead di memoria e per il context switch; appoggiandosi su uno stesso spazio kernel, ne condividono i servizi di base e soprattuto garantiscono un risparmio di spazio disco.

Con un basso costo in termini di tempo e risorse, è possibile ottenere una configurazione di elementi architettonici che rispondono a qualsiasi tipo di esigenza strutturale.

Perchè no ai container?

- I container sono il risultato di una virtualizzazione operata a livello o.s., che non può ospitare un ulteriore sistema operativo differente dall'unico ospitante.

- Nonostante con i container si abbia l'impressione che le applicazioni siano perfettamente virtualizzate e isolate, non si può dare per scontata l'assoluta completezza della separazione.
  - Es: mentre un container sta eseguendo una system call che richiede una global lock, nessuno degl'altri container sarà mai in grado di eseguire in quell'istante un pezzo di codice che richiede quella specifica istanza di lock.

Per risolvere tali complessità occorre pensare bene a cosa si vuole costruire con i container e monitorare costantemente le prestazioni dei sistemi.

Tecnologie container-based

- Docker:
  - si appoggia sul s.o. Linux che fornisce, a livello kernel, un supporto per container detto LXC (Linux Container).
- HyperV di Microsoft:
  - fornisce unità di isolamento chiamate Container, che però in realtà sono delle macchine virtuali.
- Container di Windows Server:
  - soluzione proposta ancora una volta in ambiente Microsoft, questa volta fornendo un'implementazione "ortodossa" di container.

Tratteremo Docker nel dettaglio, la tecnologia più amata dai service provider di tutto il mondo.

Next in Line...

Docker

Docker ha tre componenti essenziali:

- Docker Engine
  - fornisce le funzionalità di un Container Engine, interfacciandosi con il sistema operativo Linux sottostante.
- Docker tools:
  - sono un set di istruzioni a riga di comando che comunicano con l'API esposta da Docker Engine. Permettono la creazione di container, di nuove immagini, la configurazione di volumi e reti, e performano varie altre operazioni che impattano sul ciclo di vita dei container.
- Docker Registry:
  - si tratta dello spazio in cui sono archiviate le immagini utilizzate dai container messi in esecuzione.
    Ogni immagine è identificata da un proprio tag e gli utenti possono scaricare o condividere le proprie immagini interagendo con il Docker Hub, un *registry* online gestito da Docker Inc. (la società che distribuisce Docker) e condiviso dalla community (è possibile registrarvisi, creando un profilo, ed ottenere uno spazio di archiviazione per le proprie immagini, che diverranno in questo modo disponibili per tutti gli utenti Docker a cui vogliamo permettere di usufruire delle nostre creazioni).

Docker Image

- L'immagine di un container è un package software eseguibile che contiene tutto quello che serve al container per funzionare correttamente: codice, tools di sistema, librerie e impostazioni generiche.
- L'istanziazione dell'immagine di un container, dopo che questa è stata posta in esecuzione dal Container Engine, è costituita dal container stesso:
  - Immagine : Container = File eseguibile : Processo.
- Grazie ai Docker Tools sono disponibili svariati comandi per interagire con le immagini di container:

  `$ docker <cmd>`

  dove <cmd> può essere costituito da:

  `run <image>` — scarica l'immagine <image>, se non presente, e la mette in esecuzione
  `search <image>` — cerca l'immagine <image> sul Docker Hub
  `pull <image>` — scarica un'immagine dal Docker Hub (effettuato il log).
  `push <image>` — carica un'immagine sul Docker Hub (effettuato il log).
  `images` — mostra le immagini salvate in locale
  `rmi <image>` — elimina l'immagine <image>

Docker Image - Esempio

1. Controlliamo la presenza sul Docker Hub dell'immagine `hello-world`.
   - `$ docker search hello-world`
   - viene mostrata a video una lista di tutte le immagini che contengono "*hello-world*" nel proprio nome
2. Scarichiamo l'immagine `hello-world`
   - `$ docker pull hello-world`
   - `hello-world` è un esempio minimale di applicazione "dockerizzata".
3. Controlliamo che il download sia avvenuto correttamente
   - `$ docker images`
   - dovremmo vedere l'immagine appena scaricata fra le nostre.
4. Creare il container corrispondente all'immagine appena scaricata
   - `$ docker run hello-world`
   - viene mostrato a video un messaggio che attesta la corretta installazione di Docker e poi il container termina la sua esecuzione.

Lavorare con i container Docker

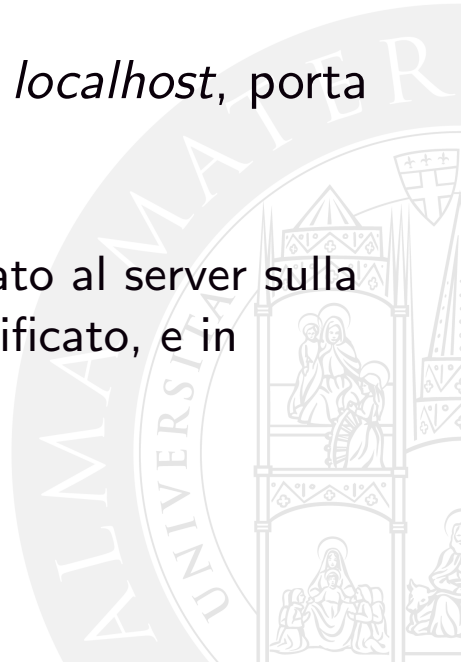
- Docker run è il comando per istanziare un container; eccone la sintassi completa:
- docker [*OPTIONS*] `<image_name>` [*COMMAND*]:
  - [*OPTIONS*]: lista di flag di settings per il lancio del container
  - [*COMMAND*]: comando da eseguire all'avvio

`! parentesi quadre indicano opzionalità`

  - tra le *OPTIONS*, preme sottolineare la presenza di:
    - i : lancia il container in modalità "interattiva" (mantiene aperto STDIN per il container)
    - t : lancia il container in modalità "emulazione di terminale" (allocando uno pseudo terminale TTY)
    - d : lancia il container in modalità "daemon" (in background, stampando il container ID)
    - p `<host>` : `<guest>` — mappa la porta `<guest>` del container nella porta `<host>` dell'host
    - - name `<name>` — assegna un nome al container (PRATICA FORTEMENTE CONSIGLIATA, per evitare di avere container con nomi random poco pratici da manovrare).

Lavorare con i container Docker II – Esempio

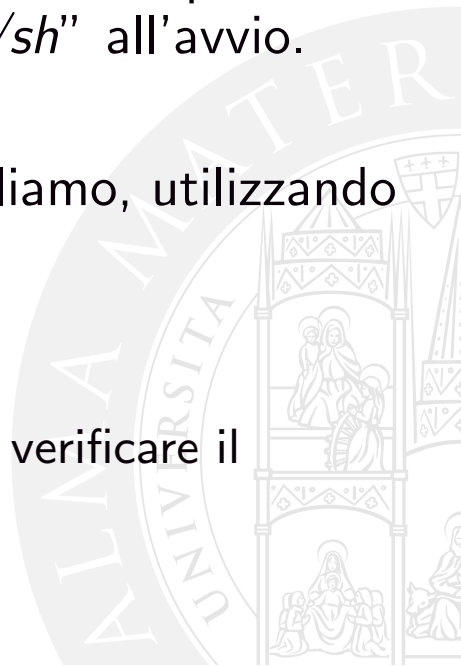
1. Lanciamo il container briceburg/ping-pong in daemon mode, collegando la porta 7777 del localhost (scelta per l'esempio, non essendo fra le porte well-known), alla porta 80 del container.
   - `$ docker run -d -p 7777:80 --name=ping_pong briceburg/ping-pong`
   - viene lanciato in background in un container un server http che risponde con un messaggio pong ad un messaggio ping.
2. Inviamo un messaggio ping con il comando curl all'indirizzo *localhost*, porta 7777.
   - `$ curl localhost:7777/ping`
   - se tutto si è svolto correttamente, il messaggio viene dirottato al server sulla porta 80, come indicato dal mapping precedentemente specificato, e in risposta si ottiene un messaggio pong.



Lavorare con i container Docker III – Esempio I

Prepariamo ora un'immagine di container che ci servirà al termine di questa lezione, partendo da una versione Docker di Alpine, distro lightweight di Linux:

1. Scarichiamo da Docker Hub l'immagine per Alpine
   ```
   $ docker pull alpine
   ```
2. Istanziamo il container a partire dall'immagine appena scaricata: impostiamo modalità "*terminal emulation*", e apriamo l'interpete "*/bin/sh*" all'avvio.
   ```
   $ docker run -it --name=my_alpine alpine /bin/sh
   ```
3. A questo punto, all'avvio del container, scarichiamo e installiamo, utilizzando il comando apk add, l'openjdk-11 e gradle.
   ```
   $ apk add openjdk11-jre
   $ apk add gradle
   ```
   - con `java -version` e `gradle -version` siamo in grado di verificare il corretto esito delle nostre operazioni.



Lavorare con i container Docker III – Esempio II

4. Possiamo "sganciarci" dal container in esecuzione con la shortcut ctrl + p ctrl + q, e salvare l'immagine che abbiamo costruito:
   ```
   $ docker commit -m="msg" <container_name> <new_image_name>
   $ docker commit -m="alpine with jdk and gradle" my_alpine
     my_alpine
   ```
5. Stoppiamo quindi il container in esecuzione e rimuoviamolo:
   ```
   $ docker stop <container_name>
   $ docker stop my_alpine
   $ docker rm <container_name>
   $ docker rm my_alpine
   ```

La meccanicità e sequenzialità di queste operazioni possono risultare ovviamente un ostacolo nella composizione di container più elaborati, e per questo motivo esiste un approccio più pratico nella costruzione delle immagini: i Dockerfile.

Dockerfile I

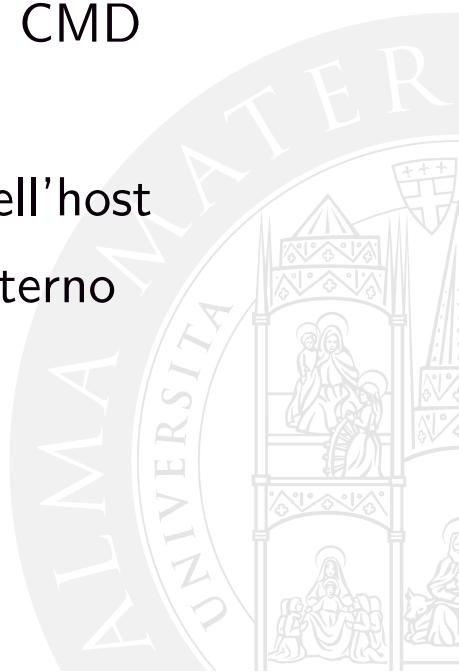
Un Dockerfile è uno script che contiene un insieme di comandi e istruzioni che saranno lanciati automaticamente in sequenza per costruire una nuova immagine.

All'interno di questo tipo di file possiamo trovare un insieme di comandi specifici:

- FROM: indica l'immagine di base utilizzata per costruire quella nuova
- MAINTAINER: opzionale, indica il mantainer dell'immagine
- RUN: esegue un comando durante il build
- COPY: copia un file dall'host al container
- ADD: simile al comando COPY, aggiunge un'opzione per scaricare file da host con URL.

Dockerfile II

- ENV: definisce una variabile d'ambiente
- CMD: esegue un comando post-costruzione dell'immagine
- ENTRYPOINT: definisce il comando da eseguire al lancio del container
- WORKDIR: definisce la directory per i comandi eseguiti con CMD
- USER: setta l'user o l'UID per il container
- VOLUME: abilita l'accesso dal container ad una directory dell'host
- EXPOSE: espone una determinata porta del container all'esterno



Dockerfile III

Ecco il Dockerfile che automatizza le procedure che abbiamo svolto precedentemente:

```
#indica che l'immagine di partenza e' alpine
FROM alpine
#lancia l'interprete /bin/sh
RUN /bin/sh
#installa l'open jdk 11
RUN apk add openjdk11-jre
#installa gradle
RUN apk add gradle
#stabilisce che all'instanziazione del container venga aperta una shell sh
ENTRYPOINT /bin/sh
```

Il comando build crea l'immagine a partire da questo file e la salva fra le nostre immagini:

`docker build [OPTIONS] PATH | URL | -`

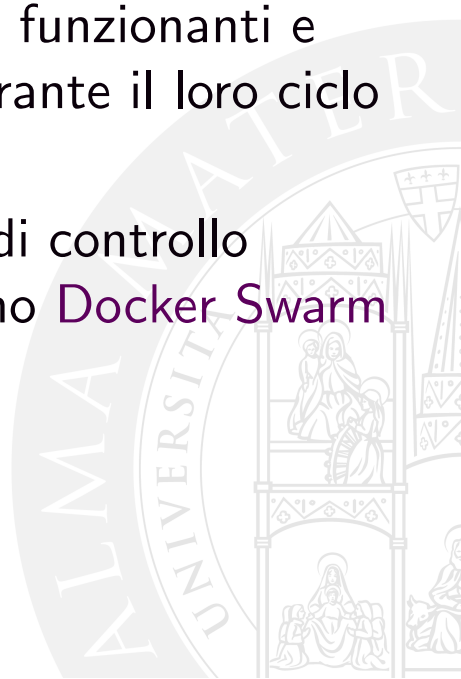
- `$ docker build -t my_alpine .`
- il "." è significativo: indica al Docker Engine che troverà il Dockerfile nella directory corrente.

! Processo : Makefile = Container : Dockerfile

Next in Line...

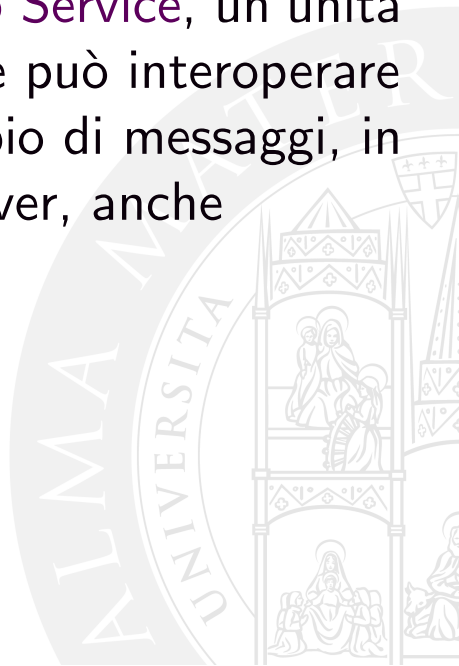
Perchè orchestrare?

- La portabilità e la riproducibilità dei processi containerizzati ci concedono l'opportunità di dispiegare e distribuire le applicazioni attraverso cloud e datacenter.
- Per questo, è necessario avere strumenti per automatizzare la gestione dei nostri sistemi, strumenti in grado di sostituire container mal funzionanti e garantire gli aggiornamenti e le riconfigurazioni di questi durante il loro ciclo di vita.
- La famiglia di tools che realizza questo tipo di funzionalità di controllo prende il nome di *orchestrators*, e i più famosi esponenti sono Docker Swarm e Kubernetes.



Servizi I

- Chi utilizza tool di orchestrazione ha come obiettivo quello di rendere fruibile, ad esempio per i clienti di un'organizzazione, un servizio, che poi molto spesso è contattabile in rete.
- Pertanto, quando si dice Service quello che si intende è Web Service, un'unità logico funzionale "atomica", potenzialmente estendibile, che può interoperare con altri servizi, instaurando comunicazioni basate su scambio di messaggi, in ottica *loose coupling*, per costruire applicazioni, client e server, anche complesse.



Servizi II

Tratti essenziali

- *autonomia*
  - un servizio è pensato per essere autosufficiente
- *accoppiamento lasco*
  - due servizi possono essere "debolmente legati" l'un l'altro, avvalendosi del file di descrizione
- *componibilità*
  - un servizio è un "mattoncino" per applicazioni più complesse
- *riusabilità*
  - un servizio funzionalmente completo può sempre essere riutilizzato in qualsiasi progetto
- *interoperabilità*
  - un servizio è tale da poter facilmente interoperare con altri, a prescindere dall'implementazione

Services Oriented Architecure & Resources Oriented Architecture

A tutte queste esigenze concettuali, fin dagl'anni '90, hanno risposto degli standard ad-hoc:

SOA: Service Oriented Architecture

"Paradigma per l'organizzazione e l'utilizzo delle risorse distribuite che possono essere sotto il controllo di domini di proprietà differenti. Fornisce un mezzo uniforme per offrire, scoprire, interagire ed usare le capacità di produrre gli effetti voluti consistentemente con presupposti e aspettative misurabili." - OASIS Open.

ROA: Resources Oriented Architecture

"Si concentra su quegli aspetti dell'architettura che riguardano le risorse. Le risorse sono un concetto fondamentale su cui si fonda gran parte del Web e gran parte dei servizi Web; ad esempio, un servizio Web è un tipo particolare di risorsa, importante per questa architettura." - W3C

Microservizi I

Oggi, progettare un sistema significa scalare e rendere disponibile un considerevole ammontare di risorse h-24, in ambienti dinamici, che cambiano continuamente e che con le loro modifiche influenzano sempre di più le caratteristiche dei sistemi, per i quali non è spesso nemmeno possibile prevedere i requisiti in fase di progettazione, a fronte di un aumento costante di utenti connessi e una richiesta dunque sempre più ampia.

Con queste premesse, la corrente di pensiero che sta trascinando sempre più addetti ai lavori è che i vecchi standard siano antiquati e non performanti al meglio per il mercato attuale, cosa che si rivolge a favore per un, ormai abbastanza consolidato e non più emergente, stile architetturale con cui realizzare applicazioni, Architettura a Microservizi.

Microservizi II

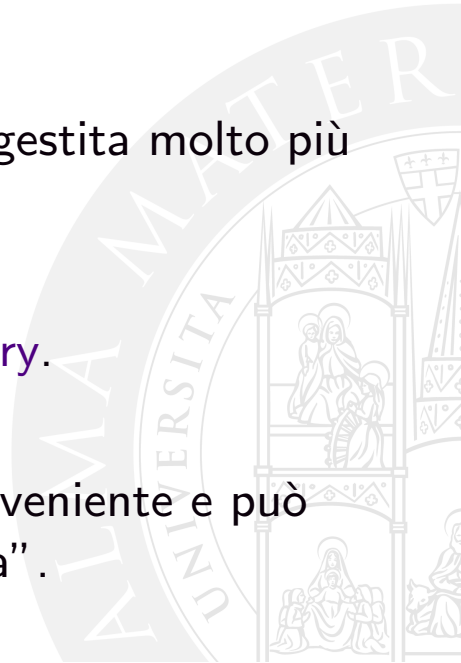
Architettura a microservizi

M. Fowler: "Architettura a microservizi è un approccio allo sviluppo di una singola applicazione come una suite di piccoli servizi, ciascuno in esecuzione nel proprio processo e comunicante con meccanismi semplici, spesso un'API di risorse HTTP".

- Se in un approccio old-fashion, si realizzavano n applicazioni, l'approccio a microservizi tende alla realizzazione di una singola applicazione composta da n servizi sviluppati e implementati secondo il principio della Single Responsability (SRP).
- Nell'architettura a microservizi la comunicazione tra i servizi è basata su HTTP tramite API RESTful, con passaggio di dati in formato JSON, spesso attraverso una coda di messaggi, quando è necessario garantire l'affidabilità.

Benefici delle architetture a microservizi I

- Eliminazione di Single Point of Failure
  - La separazione dei componenti di un'applicazione rende molto meno probabile che un bug o un problema si rifletta sull'intero sistema.
- Servizi lightweight
  - I servizi sono abbastanza piccoli, in modo da essere capiti e gestiti da un team minimo di persone, le quali potrebbero non conoscere l'intero sistema in cui il loro servizio verrà inserito.
- Orchestrazione più "snella"
  - L'automazione dei processi (build, test, deploy) può essere gestita molto più facilmente avendo servizi "snelli".
- Velocità di rilascio del singolo servizio
  - Un' architettura a microservizi favorisce il continuous delivery.
- Scalabilità efficace
  - La scalabilità a livello di servizio individuale diventa più conveniente e può essere fatta "su richiesta" (on demand) in maniera "elastica".



Benefici delle architetture a microservizi II

- Versionamento
  - Le API possono essere versionate in modo più efficace in quanto i singoli servizi possono seguire il loro schema.
- Flessibilità del linguaggio di sviluppo
  - È possibile implementare ogni servizio con un linguaggio diverso, utilizzando quello che più si adatta alla specifica funzionalità.

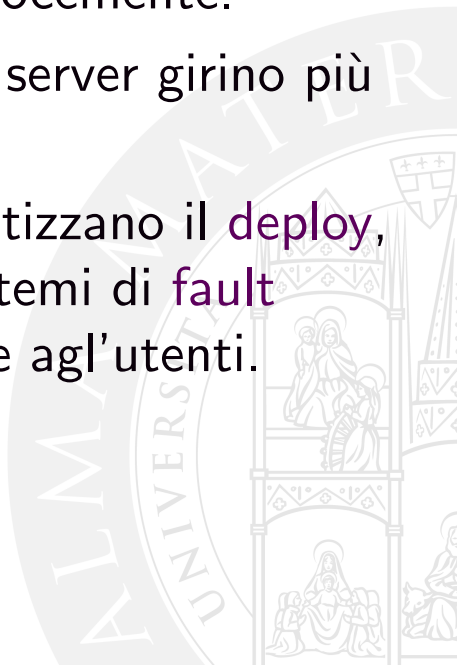
Certamente, molte delle tecniche in uso presso la community sorta attorno al software a Microservizi nascono dall'esperienza con SOA, e possiamo dire che questo nuovo modello ha posto delle etichette su un ben specifico subset di funzionalità e capacità del più vecchio standard.

Dunque, un' Architettura a microservizi non è uno stile innovativo in assoluto; tuttavia le sue implicazioni organizzative e gestionali sono un concetto abbastanza nuovo e sono state il motivo del suo successo oggi.

Container e microservizi

I microservizi trovano nei container le caratteristiche perfette per avere la miglior implementazione possibile:

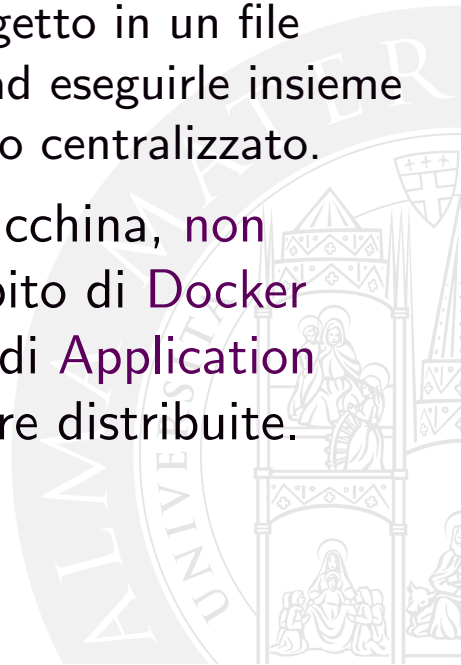
- I container incapsulano un ambiente di esecuzione a livello del sistema operativo. Perciò lanciare più microservizi in container è facile e, anzi, un microservizio in un container si inizializza e si esegue più velocemente.
- L'isolamento offerto dai container permette che in un unico server girino più microservizi e che non vi siano interferenze.
- I tool di orchestrazione che lavorano con i container automatizzano il deploy, lo scaling, effettuano bilanciamento del carico e dotano i sistemi di fault tolerance e meccanismi di resilienza, in maniera transparente agl'utenti.



Docker Compose

Concepire applicazioni come "pile di servizi" che condividono dipendenze software ha portato gli sviluppatori Docker in primis alla definizione di Docker Compose:

- Compose è un tool che aiuta il deploy di Stack, set di container, che, opportunamente "assemblati", costituiscono un unico applicativo.
  - Permette di definire le diverse funzionalità di un nostro progetto in un file centrale .yaml (il docker-compose.yml) e di iniziare da qui ad eseguirle insieme in un runtime environment isolato, amministrandole in modo centralizzato.
- Di base, questo tool svolge i suoi compiti su una singola macchina, non lavora in cluster e non coinvolge più host: questo sarà compito di Docker Swarm, strumento che permetterà di effettuare il deploying di Application Stack su più macchine, realizzando vere e proprie architetture distribuite.



Sviluppo con Docker Compose - Esempio I

- Per mostrare Docker Compose all'opera, si è scelto di costruire un'applicazione che utilizzi un server Flask e un db Redis per gestire un semplice registro scolastico.
- Nello specifico, l'applicativo consente di inserire un nuovo studente con una richiesta POST e di ottenere la lista di tutti quelli precedentemente inseriti con una richiesta GET.
- I container coinvolti sono due:
  - il web server Flask (flask), a cui vengono inviate le richieste HTTP sulla porta TCP 5000.
  - il db in-memory Redis (redis), che viene interrogato dalla componente Flask per la gestione dei record degli studenti.
- I due container condividono la medesima rete e sfruttano il Dns integrato della piattaforma Docker per comunicare fra di loro, utilizzando i nomi dei servizi.

Sviluppo con Docker Compose - Esempio II

Questo il docker-compose.yml:

```
#versione del Docker Compose file
version: '3'

#servizi appartenenti allo stack
services:
  #componente flask
  flask:
    build:
      #specifica il context per il build, compresa la posizione del
     Dockerfile
      context: .
      #specifica gli argomenti utilizzati per il build
      args:
        - IMAGE_VERSION=3.7.0-alpine3.8
    #definisce immagine utilizzata
    image: takacsmark/flask-redis:1.0
    #definisce variabili d'ambiente per i container
    environment:
      - FLASK_ENV=development
    #mapping fra <porta_host> e <porta_container>
    ports:
      - 8080:5000
    #reti a cui viene associato il servizio
```

Sviluppo con Docker Compose - Esempio III

```
    networks:
      - mynet
  #componente redis
  redis:
    #immagine utilizzata
    image: redis:4.0.11-alpine
    networks:
      - mynet
    #spazio di archiviazione utilizzato dal servizio
    volumes:
      - mydata:/data

#si definiscono le reti e i volumi che lo stack utilizzera'
#una volta effettuato il deploy
networks:
  mynet:
volumes:
  mydata:
```

Sviluppo con Docker Compose - Esempio IV

Il Dockerfile per il web-server Flask:

```
#definisce una variabile passabile in fase di compilazione
ARG IMAGE_VERSION
#Python alla "IMAGE_VERSION" e' l'immagine di base
FROM python:$IMAGE_VERSION
#imposta app come working directory
WORKDIR /usr/src/app
#copia "requirements.txt" nel container
COPY requirements.txt ./
#installa il software in requirements.txt
RUN pip install --no-cache-dir -r requirements.txt
#copia la directory corrente nel container
COPY . .
#setta la variabile d'ambiente "FLASK_APP"
ENV FLASK_APP=app.py
#setta il comando da eseguire allo start del container
CMD flask run --host=0.0.0.0
```

NB: i file *app.py* e *requirements.txt* non vengono mostrati, ma sono forniti ugualmente con le slides.

Sviluppo con Docker Compose - Esempio V

1. Effettuiamo il build dell'applicazione:
   - `$ docker-compose build`
   - non è necessario specificare ulteriori parametri, dal momento che *Compose* automaticamente utilizza il *docker-compose.yml* che abbiamo definito (se, però, questo è effettivamente il nome che abbiamo assegnato a questo tipo di file).
2. Facciamo partire l'esecuzione dello stack:
   - `$ docker-compose up -d`
   - viene lanciato il risultato del precedente build, e, a meno del parametro `-d`, che ci si assicura che i vari servizi girino in background, ancora una volta non si specificano ulteriori parametri.

Testiamo lo stack di cui abbiamo appena fatto il deploy, contattando il server all'indirizzo `localhost` e porta 8080:

- si inoltri una richiesta POST per aggiungere uno studente:
  - `$ curl --header "Content-Type: application/json" --request POST --data '"name":"<student_name>"' localhost:8080`
- con una richiesta GET si ottenga l'elenco degli studenti:
  - `$ curl localhost:8080`

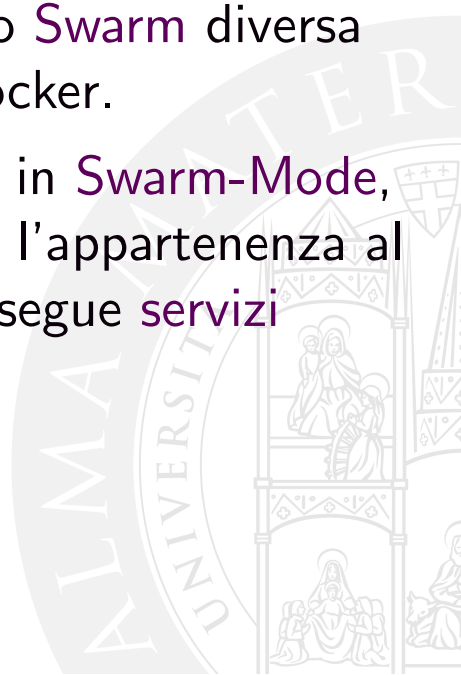
Da Docker Compose a Docker Swarm

- Anche se avessimo più host connessi fra loro in qualunque modo, Docker Compose non distribuirebbe mai su questi i container di un'applicazione, ma si limiterebbe sempre alla macchina su cui stiamo lavorando, anche lanciando multiple istanze di uno stesso container: per questo esiste Docker Swarm.
- I due tool lavorano con applicazioni complesse, composte da più servizi, e possono utilizzare pertanto lo stesso Compose file per i loro scopi: ciò implica che si possono trovare tag che sono compatibili con entrambi e tag limitati a solo uno dei due, con un chiaro riferimento nella documentazione:
  - Esempio lampante, la sezione deploy è disponibile solo per Swarm, poichè è qui che viene specificata la politica con cui vengono distribuiti i container sulle diverse macchine di uno swarm (aspetti che saranno poi più chiari).
  - D'altra parte, nella terza versione del Compose file, build è un'opzione valida solo per Compose, non per Docker Swarm, che non consente di fare il build di un'immagine nel momento in cui viene fatto il deploy dell'app.

Next in Line...

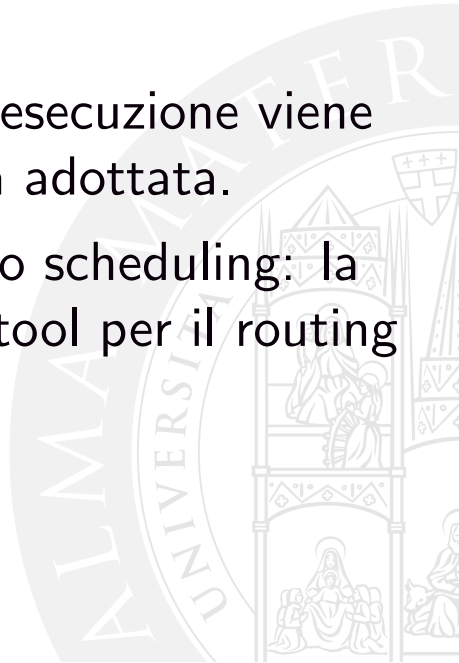
Docker Swarm I

- Swarm è la soluzione nativa proposta da Docker Inc. per l'orchestrazione (configurazione, coordinamento e gestione automatizzati) di sistemi informatici e software su più host.
- Questo, concretamente, significa che il Docker Engine mette a disposizione degl'utilizzatori una modalità per eseguire come nodo di uno Swarm diversa dalla classica modalità di esecuzione come semplice host Docker.
- Con Swarm definiamo dunque un gruppo di nodi che girano in Swarm-Mode, e che si comportano o come manager (il leader, che gestisce l'appartenenza al gruppo e la distribuzione dei compiti) o come worker (che esegue servizi swarm) o nodi che si comportano in entrambi i modi.



Docker Swarm II

- Con servizio swarm definiamo un task che deve essere eseguito sul nodo manager o sui worker. È la struttura centrale del sistema e il punto di interazione con l'utente.
- Creare un servizio significa stabilirne lo stato desiderato: immagine di container da utilizzare, comandi da eseguire, numero di repliche da istanziare, porte da esporre..
- L'istanziazione di un servizio produce uno o più task, la cui esecuzione viene assegnata ad uno o più nodi worker, a seconda della politica adottata.
- In Swarm, un servizio fornisce strutture per il networking e lo scheduling: la comunicazione fra i container creati viene gestita mediante tool per il routing dei messaggi.



Docker Machine

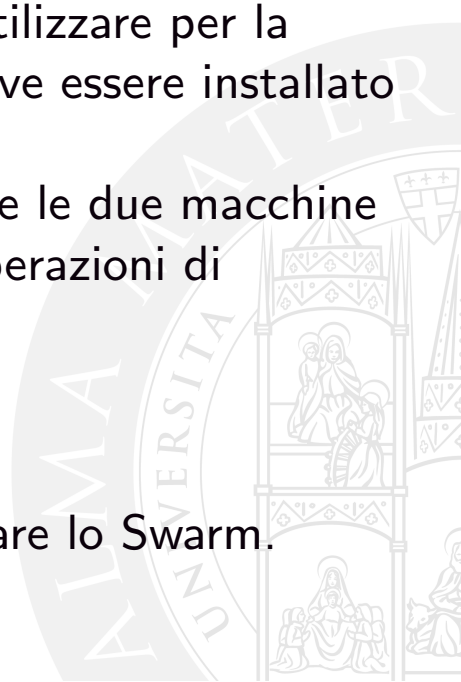
La maniera più rapida e semplice per essere operativi con uno swarm Docker è rappresentata da Docker Machine (per l'installazione: `https://docs.docker.com/machine/install-machine/`)

- Docker Machine è un tool che permette di utilizzare Docker praticamente ovunque, dal proprio pc locale Mac o Windows (fino alla versione 1.12 di Docker questa era l'unica soluzione per ambienti diversi da Linux), alla propria rete aziendale, al proprio data center o un provider cloud come Azure, AWS o DigitalOcean.
- Concretamente, Machine crea un mini host virtuale con un kernel Linux e vi installa il Docker Engine, l'API REST che interagisce con il Docker daemon e un CLI che utilizza l'API per comunicare con il daemon.
  - $ `docker-machine` è il set di comandi che permette di avviare, ispezionare, arrestare e riavviare una macchina.
- Questa soluzione ci permette di ricavare un paio di nodi per lo swarm che vogliamo creare.

Costruire uno Swarm - Creazione dei nodi

Apriamo una shell (Git Bash in ambiente Windows) e iniziamo ad interagire con Docker Machine.

1. Creiamo quello che sarà il nodo manager dello Swarm e un nodo Worker:
   - `$ docker-machine create --driver virtualbox manager`
   - `$ docker-machine create --driver virtualbox worker`
   - con l'opzione `--driver` indichiamo quale supporto driver utilizzare per la virtualizzazione (in questo caso Virtualbox, che pertanto deve essere installato sul nostro sistema).
   - Aprendo la main window di Virtualbox ora si possono vedere le due macchine create, esattamente come se avessimo svolto le consuete operazioni di creazione di una VM con questo software.
2. Creati localmente i nodi, ecco l'indirizzo IP del manager:
   - `$ docker-machine ip manager`
   - L'indirizzo restituito come output ci occorrerà per inizializzare lo Swarm.



Costruire uno Swarm - Inizializzazione I

Siamo pronti per erigere il nostro Swarm:

1. Mediante SSH ci colleghiamo all'host manager:
   - `$ docker-machine ssh manager`
   - In questo modo ci logghiamo "da remoto" al manager node, che a questo punto è pronto per diventare effettivamente tale.
2. Inizializziamo lo Swarm:
   - `$ docker swarm init --advertise-addr <MANAGER-IP>`
     - Crea lo Swarm, per il quale elegge il nodo in cui ci troviamo come manager.
     - `--advertise-addr` indica l'indirizzo che gli altri nodi nello sciame Docker utilizzeranno per connettersi: qui passiamo l'IP del manager che Docker Machine ci aveva mostrato precedentemente.
   - L'output contiene anche il comando da lanciare per aggiungere nodi worker allo Swarm:
     - Ad esempio:
       "To add a manager to this swarm, run the following command: docker swarm join –token SWMTKN-1-3pu6hszjas19x yp7ghgosyx9k8atbfcr8p 2is99znpy26u2lkl-7p73s1dx5in4tatdymyhg9hu2 172.17.0.2:2377"

Costruire uno Swarm - Inizializzazione II

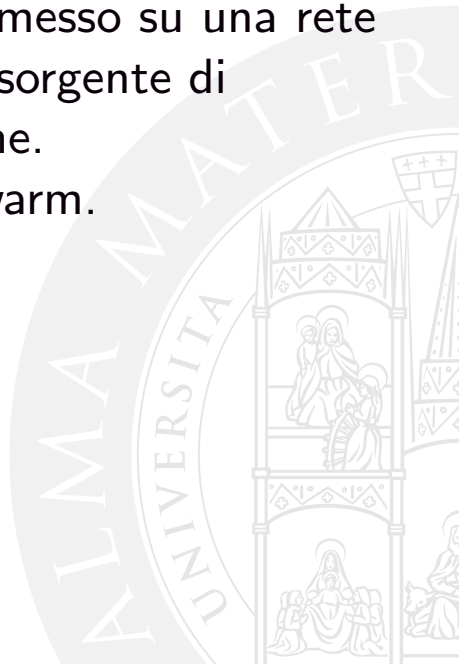
3. In una shell differente, mediante SSH, ci colleghiamo al nodo worker, che sarà arruolato nello Swarm:

   ```
   $ docker-machine ssh worker
   ```

4. Effettuiamo il join:

```
$ docker swarm join --token <TOKEN> <Manager-address>
```

- Tale comando, eseguito in qualsiasi host, lo assegna come worker allo Swarm.
- `<TOKEN>`: è un secret (blob di dati che non deve essere trasmesso su una rete o archiviato non crittografato in un Dockerfile o nel codice sorgente di un'applicazione) che consente a un nodo di unirsi allo sciame.
- `<Manager-address>`: l'indirizzo del nodo manager dello Swarm.



Perché Swarm? I

- L'operatività, come abbiamo visto, è immediatamente raggiungibile:
  - lanciando `$docker node ls` nella shell SSH manager, osserviamo come l'architettura che abbiamo messo in piedi sia attiva e recettiva.
- Ma quali sono concretamente gli aspetti che possiamo sfruttare a nostro favore nel fare Dev-ops con Swarm?
- **Design decentralizzato**: Il Docker Engine gestisce qualsiasi specializzazione in fase di runtime: di default, i task possono essere assegnati sia a worker che manager, distibuendo orizzontalmente i carichi di lavoro.
  Non a caso, tra le network che Docker consente di creare per connettere i container e, in questo caso, i servizi, la network *overlay*, che in letteratura sottende un'architettura decentralizzata peer-to-peer, ha di default scope swarm.
- **Modello dichiarativo dei servizi**: Docker Engine utilizza un approccio dichiarativo per consentire di definire lo stato desiderato dei vari servizi nello stack dell'applicazione.

Perché Swarm? II

- **Scaling**: per ogni servizio è possibile dichiarare il numero di task che si desidera eseguire. Ovviamente, la decentralizzazione by design rende molto più facile lo scaling in uno Swarm.
- **Mantenimento dello stato desiderato**: il manager monitora costantemente, per tutta la durata dell'esecuzione dei servizi (l'obiettivo è la Reliability del sistema), lo stato dello Swarm e interviene qualora lo stato desiderato espresso non sia stato raggiunto.
  Ad esempio, se si richiedono 10 repliche di un servizio e una macchina worker che ospita due di queste repliche si arresta in modo anomalo, il manager crea due nuove repliche per recuperare il fault.
  Transparentemente anche all'admin dello Swarm, si garantisce Availability e Maintainability: in caso di failure, immediatamente e senza interventi alcuni, parte la procedura di sostituzione dei task problematici, in maniera del tutto Safety (la strategia di recovery cerca di nascondere il fault all'utente).

Perché Swarm? III

- **Networking multi-host**: come già enunciato, è possibile istanziare una rete overlay per i propri servizi. Il manager assegna automaticamente gli indirizzi ai container sulla rete overlay quando inizializza o aggiorna l'applicazione.
- **Service discovery**: Il manager dello Swarm assegna a ciascun servizio nello swarm un nome DNS univoco e si occupa del bilanciamento di carico. A questi scopi, si utilizza il servizio DNS embedded dello Swarm.
- **Bilanciamento di carico**: è possibile esporre le porte per i servizi ad un bilanciatore di carico esterno. Il nostro esempio successivo si muoverà in questa direzione.
- **Sicurezza by default**: Ogni nodo nello Swarm applica l'autenticazione reciproca e la crittografia TLS per proteggere le comunicazioni tra se stesso e tutti gli altri nodi.
- **Aggiornamenti periodici**: è sempre possibile tornare ad una versione precedente di un servizio se qualcosa va storto.

Sviluppo con Swarm - Esempio I

- Ciò che segue è un esempio di applicazione dispiegata in Swarm alquanto semplice ma al contempo istruttiva.
- Si tratta di uno stack composto da un frontend costituito da un service Nginx, server proxy, che riceve le richieste dal client, le invia ad un backend costituito da un server HTTP, da cui attende la risposta da presentare poi al client.
- Nginx nasce come web server open source e, secondo uno studio del 2016, è il secondo più diffuso al mondo. Negli ultimi anni il tasso di adozione è rapidamente aumentato con l'avvento delle applicazioni progettate secondo il paradigma a microservizi. Nel nostro caso, la funzione a cui dovrà rispondere è quella di Reverse Proxy Server, frapponendosi fra client e backend Http.
- Il server HTTP risponderà alle richieste inoltrate da Nginx indicando l'indirizzo IP dell'host su cui è in esecuzione e altre informazioni di acknowledgment.

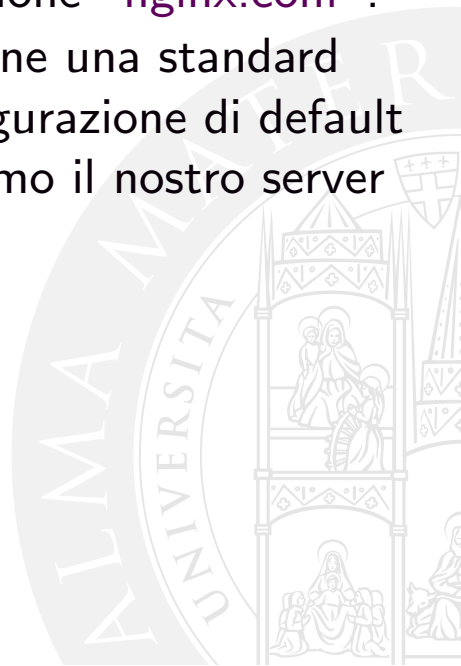
Sviluppo con Swarm - Esempio II

Nella pratica:

1. Per prima cosa, ci colleghiamo mediante SSH al nostro nodo manager, esattamente come abbiamo fatto per inizializzare lo Swarm:
   - `$ docker-machine ssh manager`
   - in ambiente Windows, da una qualsiasi Git Bash o Cygwin.
2. A questo punto, effettuiamo un paio di operazioni di "configurazione logistica":
   - alla creazione di una macchina virtuale, Docker Machine abilita la condivisione con il nuovo host della folder `/c/Users/`.
   - quello che dobbiamo fare è rendere visibile alla macchina virtuale la cartella condivisa, andando a "montare" il volume:
     - Creiamo la folder di destinazione:
     - `$ mkdir projects`
     - "Mounting" della cartella condvisa:
     - `$ sudo mount -t vboxsf c/Users $PWD/projects`
   - il risultato è che ora la folder *projects* del nodo manager è logicamente corrispondente alla directory `c/Users/` della nostra macchina fisica.

Sviluppo con Swarm - Esempio III

3. Definiamo un file di configurazione per il nostro server Nginx:
   - Nginx può assolvere a varie funzioni differenti (viene spesso adottato come entry point HTTP e load balancer, oltre che proxy server, ad esempio): il ruolo viene formalizzato nella specifica del file di configurazione "nginx.conf".
   - Costruiremo la nostra immagine Nginx andando ad utilizzarne una standard già presente sul Docker Hub e sovrascrivendo il file di configurazione di default con un nostro file che esprime il comportamento che vogliamo il nostro server esegua.



Sviluppo con Swarm - Esempio IV

Questo il nostro nginx.conf:

```
# upstream server che contattiamo con una richiesta ad nginx
upstream backend {
    server whoami;
}

server {
    # si specificano nome del server e porta
    listen 80 default_server;

    location / {
        # indica l'url a cui inoltrare le richieste e il protocollo
    utilizzato:
        # "backend" e' l'upstream server gia' definito
        proxy_pass http://backend;
        # aggiunge campo "Host" con nome del server che processa la
    richiesta
        proxy_set_header Host $host;
        # aggiunge campo "X-Real-IP" con indirizzo del client
        proxy_set_header X-Real-IP $remote_addr;
        #aggiunge campo "X-Forwarded-For" con l'indirizzo del client
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
    }
}
```

Sviluppo con Swarm - Esempio V

4. Prepariamo la docker image di Nginx:
   - il deploy di stack applicativi in Docker Swarm avviene mediante l'utilizzo di file che permettono la definizione del *desired_state* del nostro sistema, con la stessa logica operativa del tool Docker Compose, efficace in un contesto locale e non distribuito.
   - Tuttavia, non è possibile fare il build delle immagini dei container utilizzati nello stack al momento della creazione di quest'ultimo; pertanto, vanno definite preliminarmente.
   - Non solo: per effetture correttamente tutte le operazioni, salveremo il risultato nel nostro registro Docker locale e poi in remoto, sfruttando il nostro profilo su Docker Hub (è necessario preliminarmente loggarsi, da CLI, con il comando `docker login`, oppure a questo link: https://hub.docker.com/).

Sviluppo con Swarm - Esempio VI

$ `docker build -t <tag_name> -f <Dockerfile_name> <path>`

- per il salvataggio in remoto, occorre che il <tag_name> sia strutturato con una ben precisa sintassi: <docker_ID>\<image_name>
- <docker_ID>: l'ID con il quale ci siamo registrati a Docker Hub.
- <image_name>: nome assegnato all'immagine.
- Se utilizziamo un nome differente da "Dockerfile" (default) per il file utilizzato dal comando build questo va indicato con il parametro -f.
- <path> è il percorso a cui trovare il file.

Questo è il contenuto di Dockerfile_nginx, il Dockerfile che utilizziamo:

```
FROM nginx:alpine

# Sostituisce la configurazione di default di nginx con la nostra
RUN rm /etc/nginx/conf.d/default.conf
COPY ./nginx.conf /etc/nginx/conf.d/default.conf
```

$ `docker push <tag_name>`

- L'immagine viene pubblicata sul registro online ed è pronta per l'uso.

Sviluppo con Swarm - Esempio VII

5. Creiamo la rete overlay alla quale saranno connessi i servizi del nostro stack:
   - `$ docker network create -d overlay mynet`
   - Docker ci permette di specificare varie tipologie di network per i nostri container, ma in questo caso, come abbiamo già detto prima, utilizziamo overlay come driver perchè offre un supporto nativo alla Swarm Mode.
6. Creiamo lo stack e mettiamolo in esecuzione nello Swarm:
   - `$ docker stack deploy -c <file_name>.yml <stack_name>`
   - il parametro -c permette di specificare che stiamo utilizzando un Compose file, con nome `<file_name>`.
   - `<stack_name>` è il nome che assegniamo allo stack.
   - Con il comando `$ docker stack ls` possiamo controllare gli stack in esecuzione e vedere quello che abbiamo appena lanciato.
   - Lanciando il comando `$ docker stack ps <stack_name>` e inserendo il nome del nostro stack possiamo vederne lo stato e controllare l'esecuzione dei singoli servizi.

Sviluppo con Swarm - Esempio VIII

Vediamo dunque il contenuto di proxy.yml e facciamo alcune osservazioni:

```
#versione di Docker Compose adottata
version: '3.3'
#la sezione indica i servizi che saranno inclusi nello stack
services:
    #service nginx che fara' da superserver/reverse-proxy per lo stack
    nginx:
        #immagine che abbiamo precedentemente buildato
        image: christianderrick/nginx
        #matching fra la porta 80 del container e la 8000 dell'host, alla
    quale contattare il service
        ports:
            - 8000:80
        #SEZIONE SPECIFICA PER SWARM - contiene le opzioni per il deploying
    su Swarm
        deploy:
            #obblighiamo il service sul nodo manager dello Swarm, che
    vogliamo diriga le comunicazioni
            placement:
                constraints:
                    - "node.role==manager"
        depends_on:
            - whoami
        #reti a cui il servizio viene connesso
```

Sviluppo con Swarm - Esempio IX

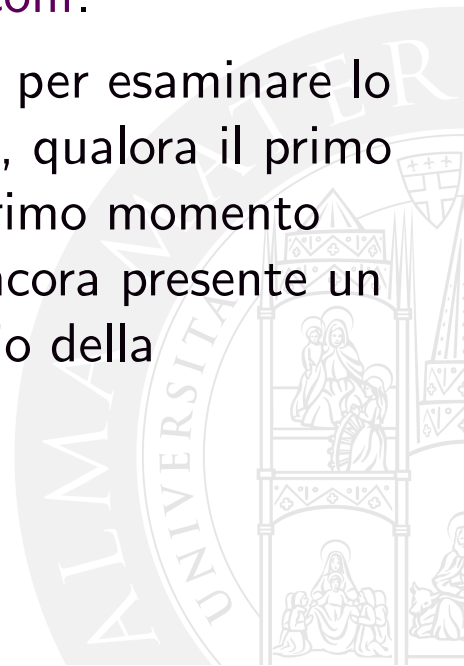
```
        networks:
            - mynet
    whoami:
        image: nginxdemos/hello:plain-text
        ports:
          - 8080:80

        networks:
            - mynet

#reti per lo stack
networks:
  mynet:
  #indica che la rete e' stata creata precedentemente e non deve essere
     creata appositamente per lo stack
    external: true
```

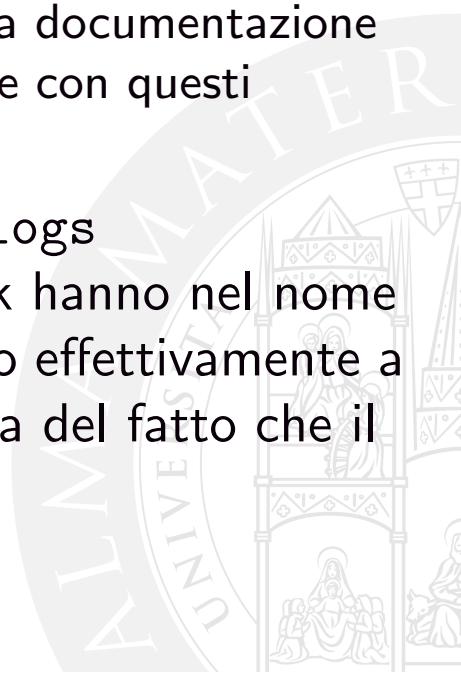
Osservazioni I

- È possibile testare da shell SSH nel nodo manager la nostra semplice applicazione con il comando `curl *:8000`
- La richiesta è indirizzata al nodo manager su cui abbiamo vincolato il task relativo al service Nginx (con l'opzione constraint placement) e poi reindirizzata al service whoami, come stabilito nel file ngix.conf.
- Utilizzando il comando `$docker stack ps <stack_name>` per esaminare lo stato dei task del nostro stack, può capitare di vedere come, qualora il primo task messo in esecuzione fosse Nginx, questo risulti in un primo momento terminato, per il fatto che sulla sua medesima rete non è ancora presente un servizio whoami, indicato nella configurazione come bersaglio della comunicazione con il client.



Osservazioni II

- Lavorare con stack non è l'unico modo di interagire e gestire service in uno swarm:
  - `$ docker service <command>` è un set di comandi che consente di manipolare le singole unità service che invece nel Compose file precedente abbiamo elencato come nodi figli del tag *servizi* e che abbiamo manipolato nel contesto di stack (quindi non direttamente), utilizzando il set di comandi
  - `$ docker stack <command>` (consultare a questo proposito la documentazione ufficiale Docker, per sapere quali comandi potere performare con questi aspetti).
- Nello specifico, utilizziamo il comando `$docker service logs <stack_name>_nginx` (i servizi che fanno parte di uno stack hanno nel nome un prefisso indicante lo stack di appartenenza), ed otteniamo effettivamente a video i messaggi di log di Nginx, che costituiscono una prova del fatto che il problema segnalato sia l'assenza di un service whoami.



Osservazioni III

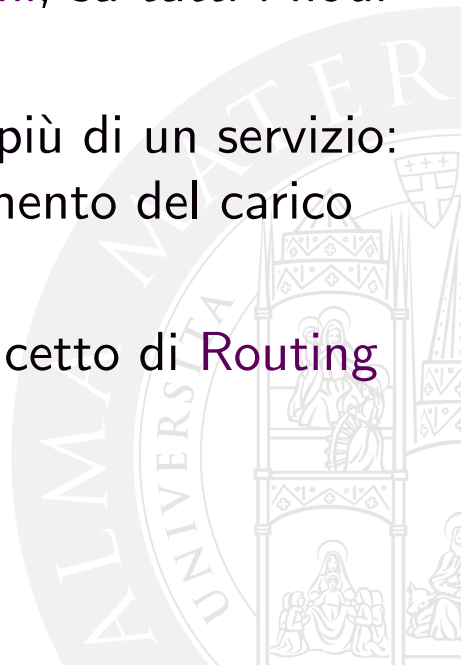
- Per questo, Swarm, con assoluta trasparenza, termina questo container segnalando l'errore, lancia whoami e rilancia Nginx, che questa volta non segnala anomalie; dal punto di vista del client che richiede il servizio però, il failure rimane nascosto, e non ci si accorge mai che qualcosa non sia andato subito come dovesse.

Bilanciare il carico di lavoro con Swarm I

- Proviamo ora ad "esplodere" l'architettura che abbiamo appena costruito, andando a lanciare multiple istanze (in questo caso 10) del servizio `whoami`:

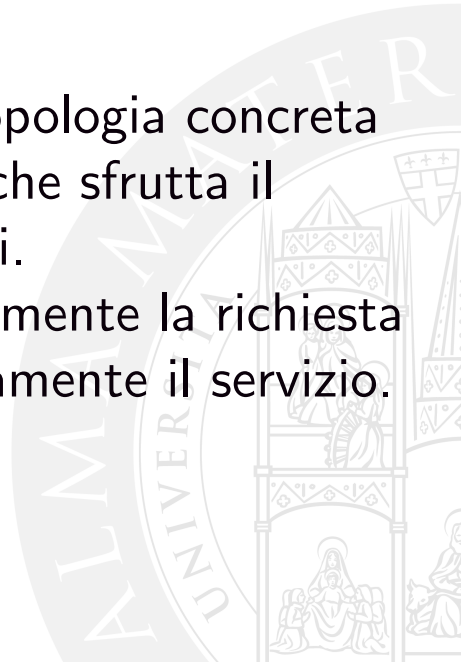
```
$ docker service scale <stack_name>_whoami=10
```

- In questo modo viene effettuato lo scaling del service whoami, su tutti i nodi del cluster in grado di fornire il proprio supporto alla causa.
- Ciò significa che, ora, ad una richiesta client può attendere più di un servizio: il manager deve organizzare una propria politica di bilanciamento del carico per gestire "l'affluenza".
- Ed è in questo contesto che si inserisce perfettamente il concetto di Routing Mesh.



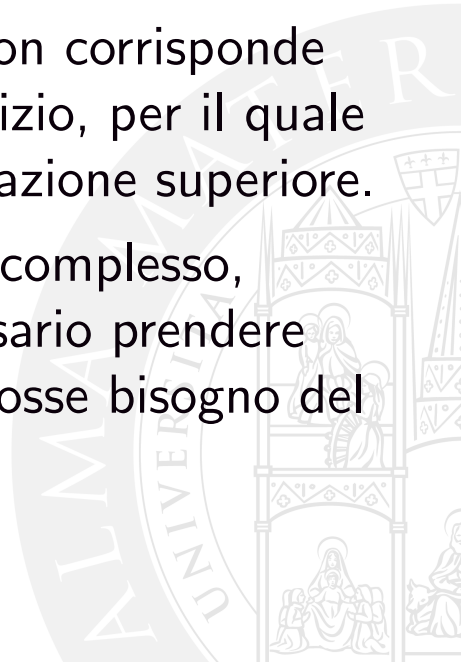
Bilanciare il carico di lavoro con Swarm II

- Routing Mesh è la modalità con cui Docker Swarm gestisce le richieste ai propri servizi: di default, un qualsiasi nodo dello Swarm venga contattato da un client può rispondere direttamente, qualora su di esso si trovi in esecuzione il task "adatto" a fornire la risposta, altrimenti "passare" la palla (in maniera, ancora una volta, del tutto trasparente al client) ad un nodo in grado di soddisfare l'esigenza.
- Questo perchè tale soluzione permette di "astrarre" dalla topologia concreta dello Swarm: quando si specifica una porta per un servizio che sfrutta il Routing Mesh, si espone la determinata porta su tutti i nodi.
  Un utilizzatore può richiedere tale servizio inviando semplicemente la richiesta allo swarm e non allo specifico nodo su cui si trova concretamente il servizio.



Bilanciare il carico di lavoro con Swarm III

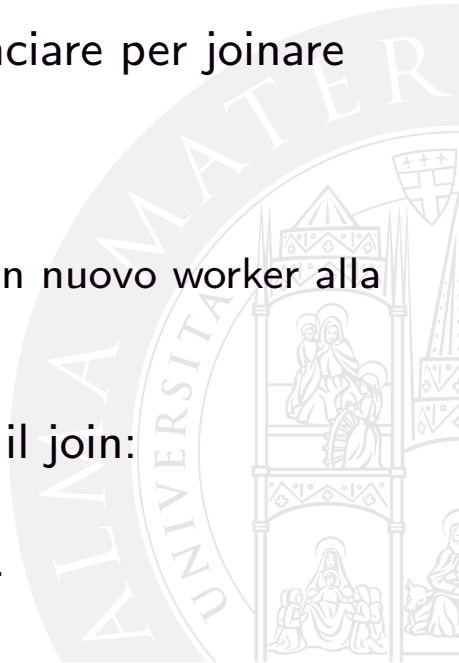
- A tutto questo si accompagna anche una strategia di service discovery: la modalità adottata di default è l'utilizzo di un indirizzo IP virtuale per ogni service dello Swarm, al quale viene associata una entry nel DNS embedded di Docker (motivo per cui nel nostro primo scenario d'esempio abbiamo indicato nel blocco "upstream" di Nginx il servizio con il nome assegnato nel Compose File).
- Tale indirizzo è quello che ci viene mostrato da whoami e non corrisponde all'indirizzo reale del o dei container che sottostanno al servizio, per il quale sono stati lanciati, e che, pertanto, occupa un livello di astrazione superiore.
- Questo rimane però un approccio base: in uno scenario più complesso, potrebbe non essere la scelta giusta, qualora si reputi necessario prendere decisioni di routing in base allo stato dell'applicazione o ci fosse bisogno del controllo totale del processo delle richieste.



Bilanciare il carico di lavoro con Swarm IV

In alternativa, potremo voler escogitare una strategia per manipolare tutti gli aspetti sopra elencati e prenderne il controllo.

1. Partiamo con l'aggiungere un nuovo nodo al nostro Swarm:
   1. creiamo una nuova docker-machine:
      ```
      $ docker-machine create <new_worker_machine_name>
      ```
   2. accediamo in ssh al manager e chiediamo il comando da lanciare per joinare come nuovo worker la macchina appena creata:
      ```
      $ docker-machine ssh <manager_machine_name>
      $ docker swarm join-token worker
      ```
      - l'output ci fornisce il comando da lanciare per aggiungere un nuovo worker alla nostra architettura.
   3. accediamo in ssh all'ultima macchina creata ed effettuiamo il join:
      ```
      $ docker-machine ssh <new_worker_machine_name>
      ```
      - *copy and paste* dell'output fornito dal comando precedente.



Bilanciare il carico di lavoro con Swarm V

2. Modifichiamo il Compose File della nostra applicazione:

```
# ...
    whoami:
        # ...
        ports:
          - published: 8080
            target: 80
            mode: host
        deploy:
            mode: global
        # ...
```

- La modalità *global* specifica l'istanziazione del service su tutti i nodi dello Swarm.
- La sintassi estesa del tag *ports* ci permette di impostare la "*mode: host*" e disabilitare il Routing Mesh: le porte dei container vengono pubblicate direttamente sull'host, "saltando" il networking di default.

Bilanciare il carico di lavoro con Swarm VI

3. Modifichiamo perciò anche il file nginx.conf.

```
# ...
upstream backend {
    server 192.168.99.121:8080 weight=2;
    server 192.168.99.122:8080 weight=4;
    server 192.168.99.123:8080 weight=4;
}
# ...
```

- Data l'introduzione della modalità *host*, indichiamo ora come rotte upstream, a cui perverranno le richieste, gli indirizzi IP dei nodi con i task in esecuzione e le porte esposte.
- Il parametro *weight* specifica il il "peso" dei server. Nell'esempio, su 10 richieste, il primo risponderà 2 volte, mentre gli altri 2 risponderanno 4 volte ciascuno.

Bilanciare il carico di lavoro con Swarm VII

4. Effettuate le modifiche, effettuiamo nuovamente il build dell'immagine e il deploy dello stack come fatto in precedenza.
   - Nella shell SSH, testiamo contattando il proxy alla porta 8000 10 volte e constatiamo che effettivamente le proporzioni sono quelle che abbiamo annunciato.
   - Contattando infatti anche da fuori lo Swarm (in un'altra shell non ssh, con il comando `$ curl <ip_machine>:8000`, indicando uno qualsiasi degl'ip, visto che il service nginx è in ascolto sulla porta 8000 e sfrutta il networking di default e di conseguenza il Routing Mesh), possiamo capire su quale nodo sono instanziati i container che ci indicano con whoami il loro indirizzo e controllare la bontà di quanto abbiamo fino ad ora sostenuto e dimostrato.
   - Allo stesso tempo, la risposta del server HTTP conterrà questa volta non un ip virtuale associato al service, ma bensì l'ip del container all'interno del quale è in esecuzione il server che ha risposto.

Swarm - Pro & Contro

PRO

- facile da installare con una configurazione rapida.
- installazione leggera: è più semplice da distribuire e la modalità Swarm è inclusa nel motore Docker.
- apprendimento più semplice di Kubernetes.
- integrazione perfetta con Docker Compose e Docker CLI (strumenti Docker nativi).

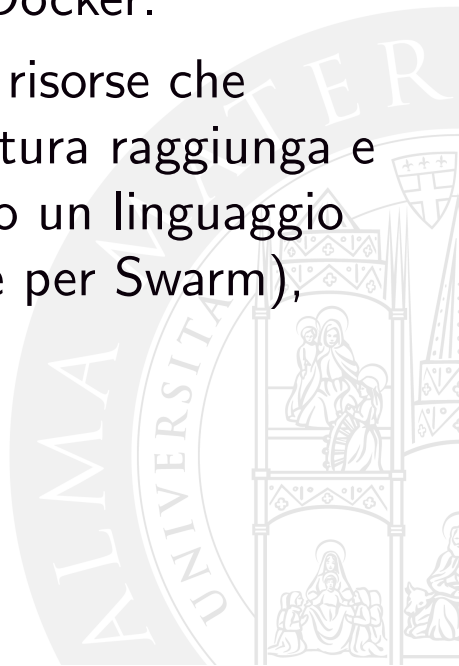
CONTRO

- fornisce funzionalità limitate.
- ha una tolleranza ai guasti limitata.
- community e un progetto più piccoli rispetto a Kubernetes.
- servizi ridimensionabili manualmente.

Next in Line...

Kubernetes

- Kubernetes (abbreviato K8s) è un sistema open-source di orchestrazione, un' architettura software con una struttura client-server che esegue su un cluster in cui dispiega la sua applicazione. Inizialmente sviluppato da Google, mantenuto ora da Cloud Native Computing Foundation, Kubernetes funziona con molti sistemi di containerizzazione, fra cui ovviamente Docker.
- Progettare un sistema in Kubernetes significa descriverne le risorse che compongono il *desired state* che vogliamo la nostra architettura raggiunga e lasciare che K8s faccia il resto: questo è possibile utilizzando un linguaggio dichiarativo (tipicamente tramite file di configurazione come per Swarm), senza la necessità di conoscere la tecnologia sottostante.



Architetture kubernetes I

Vediamo dunque quali sono i componenti della architettura Kubernetes:

- Il cluster è l'insieme di sistemi nodo che collaborano nell'esecuzione di applicazioni containerizzate.
- Il Control Plane è il centro nevralgico del cluster, in cui troviamo i componenti di controllo e i dati sul suo stato e sulla sua configurazione: il compito principale che viene svolto è controllare che il numero di container in esecuzione sia sufficiente e che siano disponibili le risorse necessarie.
  I processi di controllo sono in esecuzione su una o più macchine, che assumono ruolo di master (che si occupa di orchestrare), separato dai restanti nodi, che assumono ruolo di worker (suddivisione presente già in Docker Swarm).
- Questo ultimo è infatti il ruolo assegnato ai sistemi di elaborazione, o nodi, che compongono il data plane: concretamente, il livello che fornisce CPU, memoria, rete e archiviazione ai container in esecuzione.

Architetture kubernetes II

Vediamo dunque quali processi formano "l'ecosistema" Kubernetes:

Control Plane Components

- kube-apiserver: front end del control plane di Kubernetes, espone le Kubernetes API.
- etcd: database key-value presente in multiple copie, usato da Kubernetes per salvare tutte le informazioni del cluster.
- kube-scheduler: controlla i pod appena creati ai quali non è stato assegnato un nodo, e, dopo averlo identificato, provvede all'assegnamento.
- kube-controller-manager: gestisce i controllers, processi che monitorano lo stato del cluster e si assicurano raggiunga e mantenga il suo *desired state*.
- cloud-controller-manager: aggiunge logiche di controllo specifiche per il cloud.

Architetture kubernetes III

Node Components

- kubelet: un agente che è eseguito su ogni nodo del cluster. Si assicura che i container siano eseguiti in un pod.
- kube-proxy: proxy eseguito su ogni nodo del cluster, dedicato all'inoltro del traffico fra i nodi e alla configurazione delle regole di networking.
- container-runtime: è il software che è responsabile per l'esecuzione dei container.

Addons

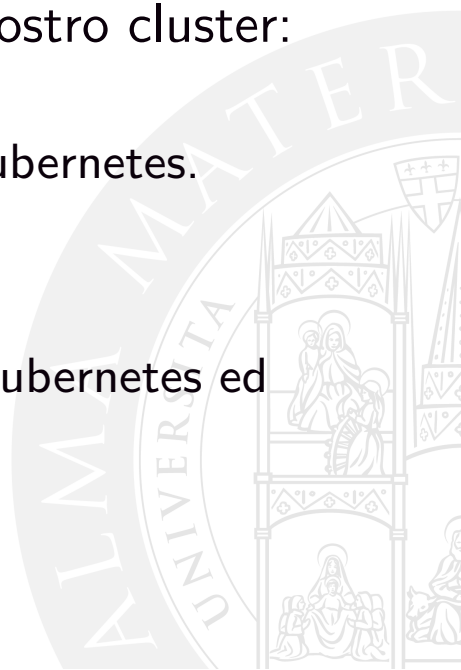
- cluster DNS: server DNS per i servizi Kubernetes.
- Dashboard: interfaccia web per cluster Kubernetes.
- Container Resource Monitoring: registra metriche generiche sui container in un database centrale e fornisce un'interfaccia utente per navigarvici.
- Cluster-level Logging: salva i log dei container in un unico log centralizzato la cui interfaccia ne permette l'esplorazione.

Costruire un cluster I

- Le vie di Kubernetes sono infinite: per costruire un cluster è possibile adottare una soluzione cloud (i principali approcci offerti da Google, Amazon AWS e Microsoft Azure) oppure on premise (i nodi sono di proprietà dell'utente e dotati di distribuzioni Linux qualunque o addirittura progettate proprio per eseguire su nodi di cluster Kubernetes, quali Rancher Os o Tectonic).
- In tutti i casi, viene messa a disposizione una dashboard di controllo con cui monitorare i servizi, presente anche per Minikube.
- Minikube è la nostra soluzione: si tratta di un applicativo che genera macchine virtuali in cui emulare un cluster e mettere in esecuzione i container, grazie agli strumenti a riga di comando di cui dispone: si tratta della "controparte" Kubernetes di Docker-Machine, con la differenza che le operazioni di creazione di un cluster e il join di nodi worker sono gestiti automaticamente dal tool stesso.
- Assieme a Minikube si installa KubectI, che fornisce l'interfaccia necessaria all'orchestrazione dei servizi dispiegati dall'utente nel cluster (Per i download, link: `https://minikube.sigs.k8s.io/docs/start/`).

Costruire un cluster II

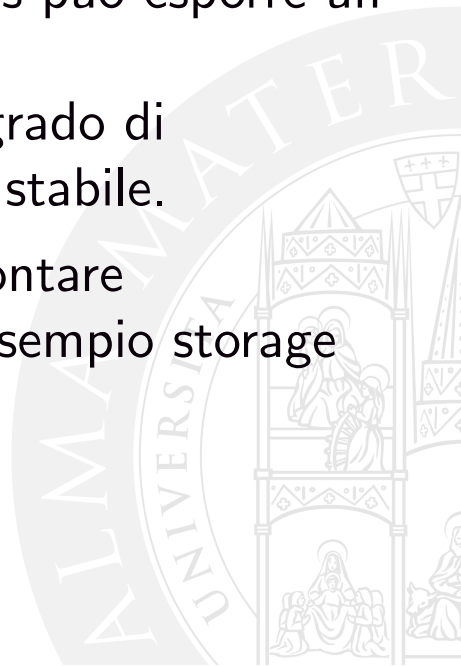
1. Creiamo il cluster:
   - `$ minikube start --nodes=2`
   - Minikube cercherà un hypervisor per la virtualizzazione: tra questi, la scelta di default è Virtualbox, compatibile con pressochè tutti i processori.
   - il risultato di questa operazione sarà la creazione di un cluster composto di due nodi (due macchine virtuali appositamente create).
2. Otteniamo dettagli e informazioni sullo stato di salute del nostro cluster:
   - `$ kubectl cluster-info`
   - Possiamo vedere l'IP del master e l'IP del server DNS di Kubernetes.
3. Vediamo i nodi appartenenti al cluster:
   - `$ kubectl get nodes`
   - Possiamo aver un risultato analogo aprendo la dashboard Kubernetes ed esaminando la sezione *Cluster*:
   - `$ minikube dashboard`



Perché Kubernetes? I

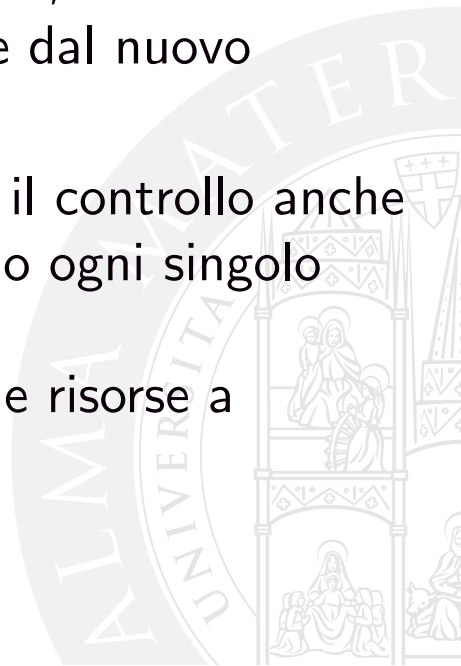
Al giorno d'oggi, tante aziende si affidano a Kubernetes per gestire le loro applicazioni. Cerchiamo di scoprire insieme le ragioni che stanno dietro a questo grande successo:

- **Service discovery & Bilanciamento di carico**: Kubernetes può esporre un container usando un nome DNS o il suo indirizzo IP.
  Se il traffico verso un container è elevato, Kubernetes è in grado di distribuirlo su più container in modo che il servizio rimanga stabile.
- **Orchestrazione dello storage**: Kubernetes permette di montare automaticamente diversi sistemi di archiviazione, come ad esempio storage locale, dischi forniti da cloud pubblici, e altro ancora.



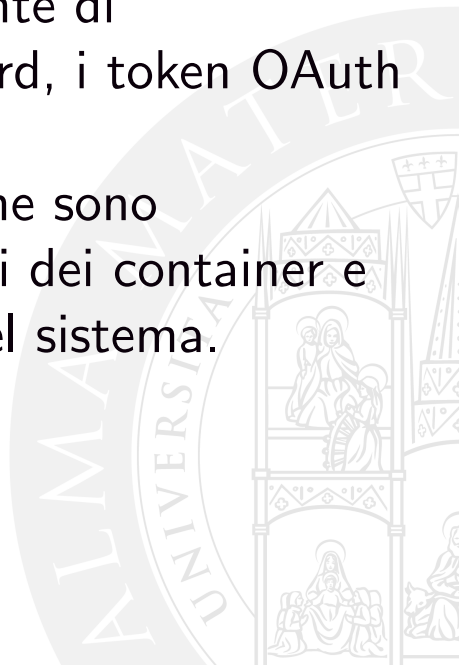
Perché Kubernetes? II

- **Aggiornamenti rollouts and rollbacks automatizzati**: comparabilmente con la policy di mantenimento dello stato desiderato adottata da Docker Swarm, anche Kubernetes si occupa di garantire il raggiungimento e il mantenimento dello *stato desiderato* (definito sulla base delle specifiche fornite dal developer) di un sistema dispiegato con questo tool.
  Per esempio, si possono creare nuovi container per un servizio, rimuovere container esistenti e adattare le loro risorse a quelle richieste dal nuovo container.
- **Automatic bin packing**: Con Kubernetes è possibile avere il controllo anche su aspetti hardware, come di quanta CPU e RAM ha bisogno ogni singolo container per eseguire.
  L'orchestrator allocherà i container massimizzando l'uso delle risorse a disposizione.



Perché Kubernetes? III

- **Self-healing**: Kubernetes riavvia i container che si bloccano per un qualche failure, sostituisce o termina quelli che non rispondono agl' health checks, ed evita di far arrivare traffico ai container che non sono ancora pronti per rispondere correttamente.
- **Configurazione e gestione dei Secret**: Kubernetes consente di memorizzare e gestire informazioni sensibili, come le password, i token OAuth e le chiavi SSH.
  Tali informazioni sensibili e la configurazione dell'applicazione sono distribuibili e aggiornabili senza dover ricostruire le immagini dei container e senza svelare le informazioni sensibili nella configurazione del sistema.



Perché Kubernetes? IV

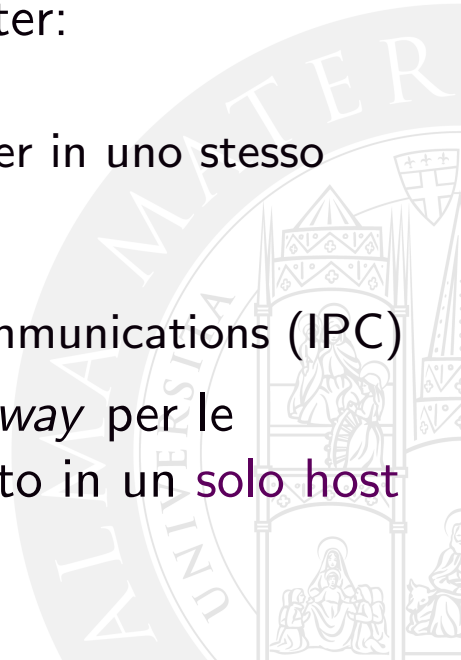
Kubernetes non è un sistema PaaS (Platform as a Service) tradizionale e completo: fornisce gli elementi base per la costruzione di piattaforme di sviluppo, ma conserva le scelte dell'utente e la flessibilità dove è importante.

Kubernetes:

- Non limita i tipi di applicazioni supportate.
- Non fornisce servizi a livello applicativo (come middleware) framework di elaborazione dati, database, cache, né sistemi di storage distribuito come servizi integrati.
- Non impone soluzioni di logging, monitoraggio o di gestione degli alert.
- Fornisce un'API dichiarativa che può essere richiamata da qualsiasi sistema.
- Non è un semplice sistema di orchestrazione: i processi di controllo indipendenti e componibili che guidano costantemente lo stato attuale verso lo stato desiderato e la non necessità di un controllo centralizzato lo rendono un sistema più potente, robusto, resiliente ed estensibile.

Le risorse di Kubernetes - i Pods

- In Kubernetes lo stato del sistema è descritto mediante l'utilizzo del concetto di risorsa.
- Il più piccolo esempio di risorsa è il Pod: si tratta di uno o potenzialmente più container che collaborano nello svolgimento di una o più funzionalità logiche e che pertanto necessitano di lavorare a stretto contatto tra loro, come se fossero in uno stesso host virtuale.
- I container di uno stesso Pod, infatti, sono realizzati per poter:
  - condividere uno stesso indirizzo IP.
  - avere lo stesso spazio delle porte di protocollo (due container in uno stesso pod non possono attestarsi sulla stessa porta di protocollo)
  - comunicare tra di loro mediante *localhost*
  - interagire tra di loro mediante le classiche Inter Process Communications (IPC)
- Ogni pod ha un proprio indirizzo IP (che utilizza come *gateway* per le comunicazioni fra container di Pod differenti), ed è contenuto in un solo host fisico, che però a sua volta può contenere più Pod.



Fare il deploy di un Pod - Esempio I

1. Creiamo un pod da eseguire nel nostro cluster:

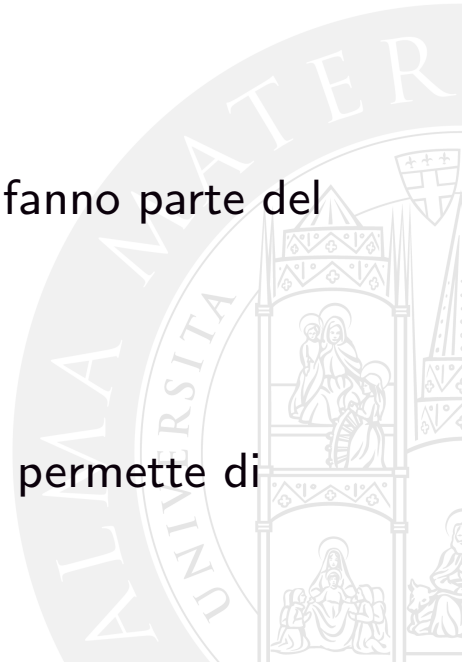
```
$ kubectl create -f single-container.yml
```

```
apiVersion: v1
#tipo di risorsa
kind: Pod
#METADATA: descrive univocamente la risorsa
metadata:
  #nome della risorsa
  name: nginx
  #definiamo una label "name" con valore "nginx"
  labels:
    name: nginx
#SPEC: definisce le specifiche della risorsa, il desired state
spec:
  #containers appartenenti al pod
  containers:
  #nome del container
  - name: nginx
    #immagine utilizzata
    image: nginx
    #porte esposte
    ports:
    - containerPort: 80
```

Fare il deploy di un Pod - Esempio II

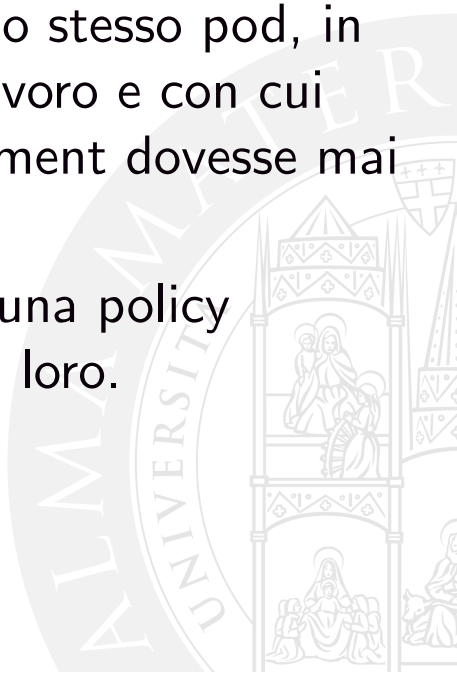
- Questo file specifica un container NGiNX molto semplice.
- Generalmente, un pod si lancia partendo da file simili, poiché in questo modo viene promossa la trasparenza e la riproducibilità.

2. Verifichiamo che l'operazione è avvenuta correttamente
   - $ `kubectl get pods`.
   - l'output ci mostra i pod in esecuzione nel cluster, e, fra questi, quello che abbiamo appena istanziato.
3. Visualizziamo le informazioni legate al nostro pod "nginx":
   - $ `kubectl describe pod nginx`.
   - vengono elencate le informazioni sul pod, sui container che fanno parte del pod, sui volumi e gli eventi corredati di messaggi di log.
4. VIsualizziamo i log prodotti dal pod:
   - $ `kubectl logs nginx`.
   - attività fondamentale per il debug delle applicazioni, poichè permette di intercettare un errore attraverso i messaggi emessi.



Le risorse di Kubernetes - i Deployments

- Dispiegare un' applicazione in un cluster Kubernetes come pod può essere una soluzione efficace, ma qualora si verifichi un fail il pod non sarà riavviato.
- Per rimediare a questo, Kubernetes ha creato un oggetto chiamato deployment.
- Dispiegare un deployment significa lanciare più istanze di uno stesso pod, in modo da creare delle repliche su cui distribuire il carico di lavoro e con cui garantire la fault tolerance: se un pod creato con un deployment dovesse mai andare in crash, verrà riavviato automaticamente.
- Ovviamente, un deployment comporta l'esigenza di definire una policy secondo cui selezionare i diversi pod e bilanciare il carico tra loro.

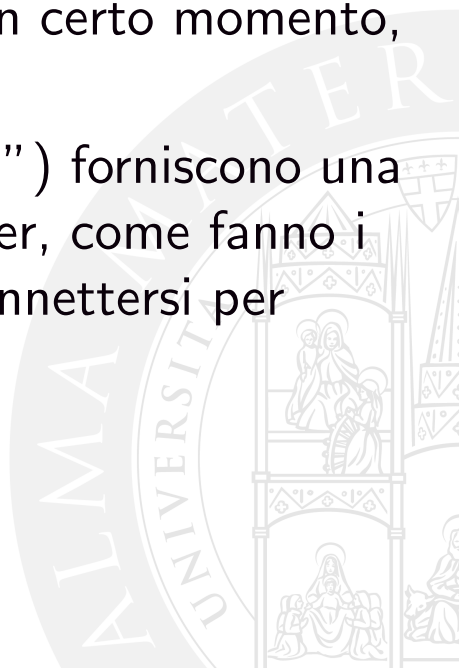


Fare il deploy di un Deployment: Esempio

1. Lanciamo subito un deployment nel nostro cluster e controlliamo sia effettivamente presente:
   ```
   $ kubectl create deployment nginx --image=nginx --port 80
   $ kubectl get deployments
   ```
2. Se tutto è ok, e fra i deployment figura "*nginx*", esiminiamone i pod creati:
   ```
   $ kubectl get pods
   ```
3. "Testiamo" il deployment, andando ad eliminarne il pod a cui è associato:
   ```
   $ kubectl delete pod $(kubectl get pods --no-headers=true | awk '{print $1;}')
   ```
   - il comando in questione (da eseguirsi in una shell bash o emulatrice di shell bash) ottiene il nome del pod per poi eliminarlo.
4. Controlliamo l'effettiva eliminazione:
   ```
   $ kubectl get pods
   ```
   - Magia dei deplyoment: il pod esegue ancora, poichè è stato riavviato al fail, cosa che non avviene con un pod autonomo.

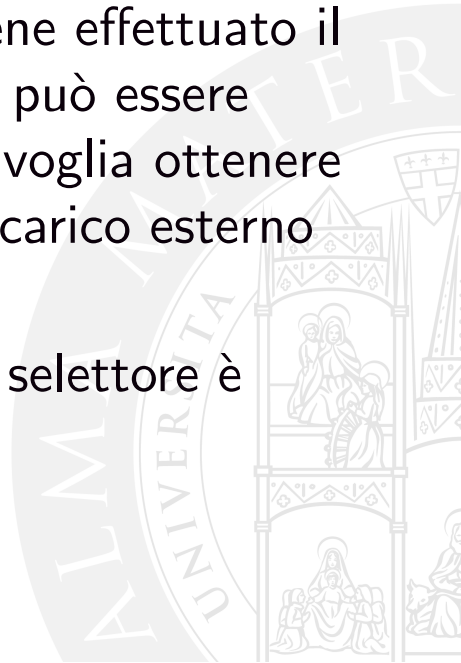
Le risorse di Kubernetes - i Services I

- I pod sono unità non permanenti, mentre se si dà alla nostra applicazione la forma di un deployment, questo può creare e distruggere dinamicamente i pod su cui si appoggia.
- Ogni pod, qualsiasi sia il contesto in cui è stato aggiunto al cluster, ha un suo indirizzo ip, e, se ne consideriamo un insieme corrispondente ad un deployment sul quale sta girando la nostra applicazione in un certo momento, può capitare che questo cambi in un momento successivo.
- Questo porta a un problema: se alcuni set di Pod ("backend") forniscono una funzionalità ad altri Pod ("frontend") all'interno di un cluster, come fanno i frontend a scoprire e tenere traccia dell'indirizzo IP a cui connettersi per sfruttare le funzionalità offerte dal "backend"?
- Qui entrano in gioco i service.



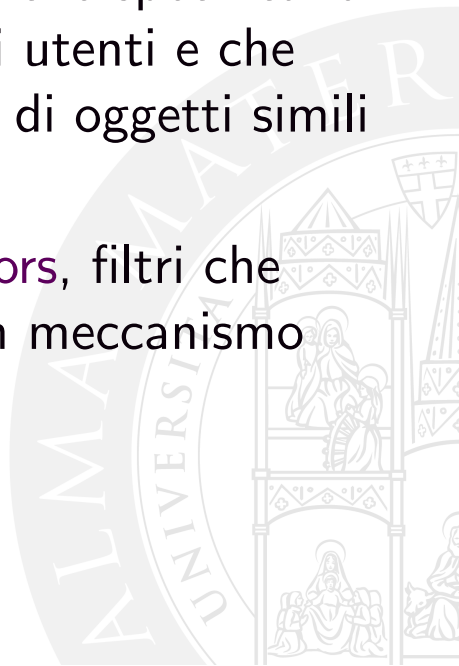
Le risorse di Kubernetes: - i Services II

- In Kubernetes, un service è un'astrazione che definisce un insieme logico di pod, individuato da un selettore, e una politica con cui accedervi (a volte questo modello è chiamato **micro-service**).
- Per "politica con cui accedervi" di solito si intende l'indirizzo IP o il nome con cui il DNS mappa il service.
- Tipicamente, un service include anche la politica con cui viene effettuato il bilanciamento di carico delle richieste tra i diversi pods, che può essere gestita e implementata da Kubernetes stesso ma, in caso si voglia ottenere qualche effetto specifico, da un servizio di bilanciamento di carico esterno (esattamente quanto valeva per i service Swarm).
- Di solito, quindi, ogni gruppo di pods definiti da uno stesso selettore è protetto da un servizio che ne regola l'accesso.



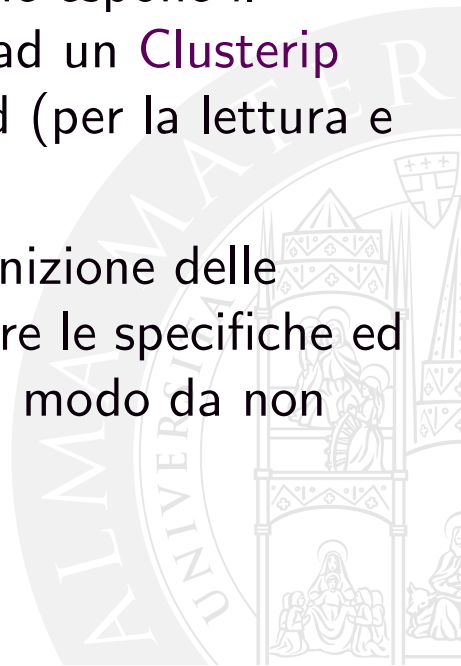
Fare il deploy di un app con Kubernetes - Esempio I

- Operativamente, quando si dispiega un' applicazione in un cluster, prima si istanziano i singoli Pods (o le repliche), creando il deployment, poi davanti a ciascun gruppo si pone il servizio che ne regola l'accesso, il tutto ovviamente utilizzando un file che descrive le proprietà di ogni risorsa.
- Ai pods vengono associate una o più label, coppie key-value che specificano attributi identificativi che sono significativi e rilevanti per gli utenti e che vengono utilizzate per organizzare e selezionare sottoinsiemi di oggetti simili per funzionalità offerte.
- Per associare un servizio alle altre risorse si utilizzano selectors, filtri che specificano una label e il suo valore richiesto e forniscono un meccanismo utile per la ricerca di una risorsa.



Fare il deploy di un app con Kubernetes - Esempio II

- L'esempio proposto anche in questo caso è una semplice applicazione Flask con db Redis: il server Flask incrementa e registra il numero di visite che riceve, ogni volta leggendo e salvando il valore nel db Redis, e poi visualizza un messaggio in HTML che mostra il risultato calcolato.
- Ciascun componente viene realizzato da una coppia deployment-service: alla controparte Flask è preposto un service di tipo Nodeport, che espone il deployment all'esterno del cluster, mentre il db è associato ad un Clusterip service, che lo rende raggiungibile internamente agl'altri pod (per la lettura e l'aggiornamento del record con le visite).
- I file di configurazione .yml che seguono racchiudono la definizione delle coppie di risorse: in Kubernetes è possibile infatti raggruppare le specifiche ed evitare di dover trattare separatamente ogni componente, in modo da non rischiare una proliferazione di file.



Fare il deploy di un app con Kubernetes - Esempio III

- Lanciamo i componenti delle nostra applicazione:

```
$ kubectl apply -f <redis_conf>.yml
$ kubectl apply -f <flask_conf>.yml
```

Questo il file per il db Redis:

```
apiVersion: v1
#tipo di risorsa: Service
kind: Service
metadata:
  #nome del service
  name: redis
spec:
  #definito il tipo di selettore per cercare la risorsa
  selector:
    name: redis
  #porta alla quale trovare in ascolta la risorsa
  ports:
  - port: 6379
---
apiVersion: apps/v1
#tipo di risorsa: Deployment
kind: Deployment
```

Fare il deploy di un app con Kubernetes - Esempio IV

```
metadata:
  #nome del deployment
  name: redis
  #label "name: redis" esposta dal deployment
  labels:
    name: redis
spec:
  selector:
    #indica le label dei pod che saranno coinvolti nel depl.
    matchLabels:
      name: redis
  #viene definito il template per i pod del deployment
  template:
    metadata:
      labels:
        name: redis
    spec:
      #container dei pod
      containers:
      - image: redis:4.0.11-alpine
        name: redis
        ports:
        - name: tcp
          containerPort: 6379
```

Fare il deploy di un app con Kubernetes - Esempio V

E questa la configurazione del server Flask:

```
apiVersion: v1
#tipo di risorsa: Service
kind: Service
metadata:
  #nome del service nodeport
  name: nodeport-web-service
spec:
  #definito il tipo di selettore per cercare la risorsa
  selector:
    name: web
  ports:
  - protocol: TCP
    port: 80
    #la porta sulla quale si cercano depl. in ascolto
    targetPort: 5000
  #Nodeport: l'applicazione e' raggiungibile all'esterno del cluster
  type: NodePort
---
apiVersion: apps/v1
#tipo di risorsa: deployment
kind: Deployment
metadata:
  #nome del deployment e labels associate
```

Fare il deploy di un app con Kubernetes - Esempio VI

```
  name: app-web
  labels:
    name: web
spec:
  selector:
    #indica le label dei pod che saranno coinvolti nel depl.
    matchLabels:
      name: web
  #viene definito il template per i pod del depl.
  template:
    metadata:
      labels:
        name: web
    spec:
      #container dei pod
      containers:
      #immagine per app flask disponibile online
      - image: christianderrick/web
        name: web
        env:
        - name: GET_HOSTS_FROM
          value: dns
        ports:
        - name: tcp
          containerPort: 5000
```

Fare il deploy di un app con Kubernetes - Esempio VII

- Il db Redis è in ascolto per connessioni sulla porta 6379, e sulla stessa porta si posiziona anche il service - interfaccia anteposto: il match fra i due componenti viene effettuato sulla base della label "name: redis", su cui il service registra il proprio selettore e trova il deployment a cui deve fare riferimento per l'inolto delle richieste.
- Analogamente accade per il server Flask (match service-deployment sulla label "name: web"); tuttavia viene specificata la targetPort 5000 per il Nodeport, alla quale il service trova in ascolto il server Flask.
- Questo si connette a Redis (in realtà al service Redis che funge da entrypoint), utilizzando il servizio DNS built-in di Kubernetes, che anche qui, come già accaduto per Docker Swarm, al deploy di un service registra una entry da utilizzare per il routing delle comunicazioni fra componenti.

Fare il deploy di un app con Kubernetes - Esempio VIII

- Possiamo testare dunque la nostra applicazione in due modi differenti:
  - `$ curl <minikube_ip>:<nodeport_port>`
  - viene inviata una richiesta all'external ip di minikube, alla porta esposta dal service Nodeport per la comunicazione extra-cluster.
    - l'indirizzo ip da utilizzare è recuperabile con:
    - `$ minikube ip`
    - la porta dal campo Nodeport visualizzabile con:
    - `$ kubectl describe svc <nodeport_service_name>`
  - `$ minikube service <service_name>`
  - Un'alternativa è costituita da questa shortcut, che realizza esattamente una chiamata `curl` come l'abbiamo definita sopra.
- Può accadere che il Dns di Kubernetes non risolva correttamente il nome dei service: in tal caso, è possibile effettuare un rolling update del kube-dns, monitorarne lo stato e poi effettuare nuovamente il test.
  - `$ kubectl -n kube-system rollout restart deployment coredns`
  - `$ kubectl logs --namespace=kube-system -l k8s-app=kube-dns`

Kubernetes: Pro & Contro

PRO

- Kubernetes è supportato dalla Cloud Native Computing Foundation (CNCF).
- Kubernetes ha una community straordinariamente vasta tra gli strumenti di orchestrazione dei container. Oltre 50.000 commit e 1200 collaboratori.
- Kubernetes è uno strumento open source e modulare che funziona con qualsiasi sistema operativo.
- Kubernetes fornisce un'organizzazione di servizi semplice, grazie ai modelli di risorse che introduce.

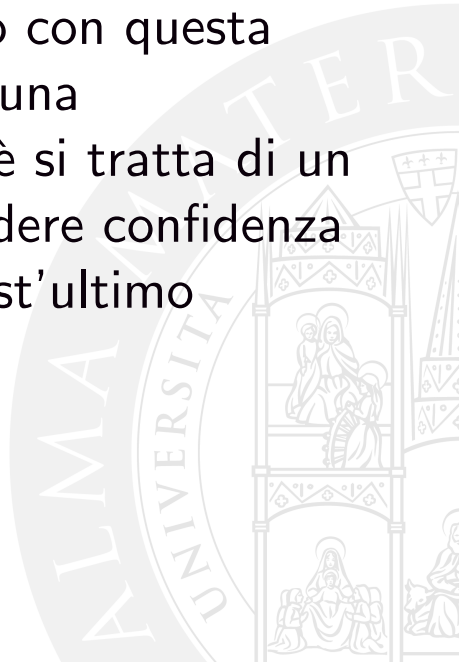
CONTRO

- L'apprendimento di Kubernetes può risultare ostico, e la curva di apprendimento ripida.
- In Kubernetes è necessario disporre di un set separato di strumenti per la gestione, inclusa la CLI di kubectl.
- Non è compatibile con gli strumenti Docker CLI e Compose esistenti.

Next in Line...

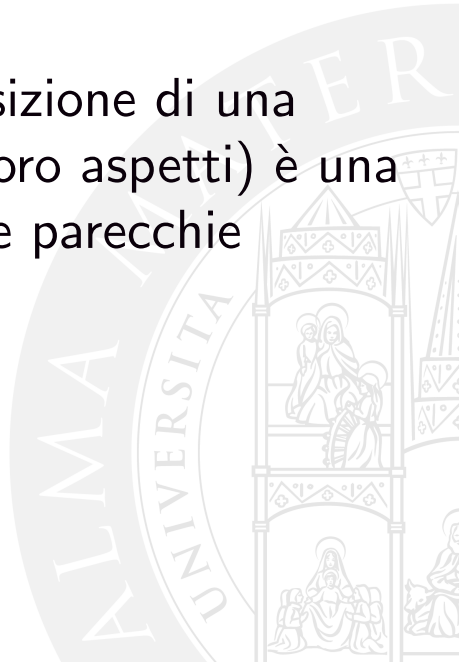
Ultime considerazioni I

- Ovviamente la trattazione degl'argomenti non copre in toto quello che sono Kubernetes e Docker Swarm: tuttavia quanto è stato proposto è sufficiente per avere un'idea di cosa significhi utilizzare questi sistemi.
- Docker Swarm è garante di un'operatività immediata con quello che sono gli Swarm, gli Stack e i Service; a fronte di ciò, però, lavorando con questa tecnologia sono emerse in primis delle difficoltà nel reperire una documentazione efficace e consigli utili dai developer, poichè si tratta di un tool "più marginale" rispetto a Kubernetes, più utile a prendere confidenza con alcuni meccanismi che poi vengono approfonditi da quest'ultimo orchestrator.



Ultime considerazioni II

- Effettivamente, lavorare con Swarm permette di sviluppare un know-how che poi aiuta notevolmente con l'utilizzo di Kubernetes, che, essendo professionalmente utilizzato dalle aziende (a fronte di una minore adozione di Swarm) e ampiamente diffuso, offre un set alquanto corposo di strumenti il cui apprendimento può effettivamente risultare difficile.
- Tuttavia, padroneggiare questo strumento (insieme all'acqusizione di una certa disinvoltura con la manipolazione dei container e dei loro aspetti) è una skill attualmente molto richiesta dalle aziende e può regalare parecchie soddisfazioni.



Ora tocca a voi I

A questo punto siete in grado di utilizzare container ed orchestrators per simulare un ambiente distribuito su più macchine che sfrutti l'implementazione LINDA realizzata durante il corso: l'idea è quella di sviluppare una soluzione che miri ad avere agenti su host differenti in grado di comunicare tra loro:

1. Sfruttando l'implementazione fornita qui, si costruiscano ambienti di esecuzione per i chatter agent che realizzeranno la comunicazione e per il server LINDA che medierà l'accesso agli spazi di tupla remoti.
   - Concretamente, costruire le container image per ogni componente software dell'architettura che vogliamo mettere in piedi, sfruttando tutti gli strumenti che sono stati mostrati e le slides fornite per il final-lab del corso di SD.
   - Si sfrutti l'immagine con OS Alpine, ambiente Gradle e Jdk11, già creata durante la lettura di queste slides.
2. Successivamente, occorre dichiarare il *desired state* del sistema, in modo da gestirne il deploy all'interno di un Cluster / Swarm di macchine.
   - Si scelga un orchestrator e si dispieghi, coerentemente con la decisione presa, il gruppo di macchine sulle quali si vuole distribuire l'applicazione.

Ora tocca a voi II

  - Si scriva poi un file di configurazione che dichiari le specifiche delle componenti del sistema distribuito: si sfruttino le immagini definite al punto 1 e si stabiliscano correttamente le relazioni fra le risorse coinvolte.

- L'implementazione fornita per il final-lab `https://gitlab.com/pika-lab/courses/ds/ay2021/lab-9` è stata "adattata" per l'esercizio proposto:
  - Il server Javalin non si arresta più in assenza di input da tastiera, per evitare che in sede di deploy possa arrestare la sua esecuzione.
  - Il comportamento degl'agenti è stato automatizzato: ora non sono in attesa di ricevere il messaggio da inviare da standard input, ma, in sequenza, spediscono un messaggio "hard-coded", attendono il messaggio in entrata e poi terminano la propria esecuzione.
- Il corretto funzionamento dell'architettura prodotta al termine è osservabile mediante i log prodotti dalle componenti software dispiegate.

Docker e orchestration tools

Distributed Systems / Technologies

D'Errico Christian
christian.derrico@unibo.it

Dipartimento di Informatica – Scienza e Ingegneria (DISI)
Alma Mater Studiorum – Università di Bologna a Cesena

Academic Year 2020/2021

References I

[1] Kubernetes Authors.
Kubernetes documentation.

[2] Kubernetes Authors.
Kubernetes project.

[3] Mateo Burillo.
Docker security vulnerabilities and threats.

[4] Bret Fisher.
Dog vs. cat: Docker swarm stacks on stacks on stacks.

[5] James Lewis & Martin Fowler.
Microservices.

[6] W3C Working Group.
Web services architecture - the resource oriented model.

[7] Docker Inc.
Docker docs.

References II

[8] Docker Inc.
Docker hub.

[9] F5 Inc.
Nginx docs.

[10] Stefano Mariani.
Docker for devs.

[11] O'Reilly Media.
Katakoda learning platform.

[12] Chris Richardson.
Pattern: Microservice architecture.

[13] Andrea Omicini & Andrea Santi.
Standard services for distributed systems web services.