ALMA MATER STUDIORUM – UNIVERSITÀ DI BOLOGNA
CAMPUS DI CESENA

DIPARTIMENTO DI INFORMATICA – SCIENZA E INGEGNERIA

CORSO DI LAUREA IN INGEGNERIA E SCIENZE INFORMATICHE

TITOLO DELL’ELABORATO

Virtualizzazione mediante software container: un approfondimento su Docker

Elaborato in
PROGRAMMAZIONE DI SISTEMI EMBEDDED

Relatore
Prof. ALESSANDRO RICCI

Presentata da:
GIANNI SAVINI

Co-relatore
Ing. ANGELO CROATTI

Terza Sessione di Laurea
Anno Accademico 2017 – 2018

PAROLE CHIAVE

Virtualizzazione

Container

Docker

Linux kernel

Servizi

Alla mia famiglia

Indice

Introduzione……………………………………………………..ix

1. Software container……………………………………………1
1.1. Introduzione ai software container…………………………1
1.1.1. Cenni di virtualizzazione………………………....2
1.1.2. Confronto fra virtualizzazione e software container ……………………………………………………...3
1.2. Evoluzione dei software container………………………...5
1.3. I container al giorno d'oggi………………………………..6
1.4. Vantaggi dell'uso dei software container………………...10
2. Docker……………………………………………………..13
2.1. Cos'è Docker……………………………………………..13
2.2. Architettura di Docker……………………………………13
2.3. Come funziona Docker…………………………………..17
2.3.1. Installazione……………………………………..17
2.3.2. Alcuni comandi ed esempi………………………18
2.3.3. Dockerfile……………………………………….20
2.4. Docker compose…………………………………………22
2.5. Docker swarm……………………………………………23
3. Docker e sistemi a micro-servizi……………………………25
3.1. Linee guida………………………………………………25
3.1.1. Premessa sui micro-servizi……………………...26
3.2. Docker e micro-servizi…………………………………..27

Conclusioni……………………………………………………29

Ringraziamenti………………………………………………...31

Bibliografia……………………………………………………33

Introduzione

Il mercato dell’informatica, specialmente quello del cloud computing sta volgendo sempre più lo sguardo alla virtualizzazione tramite container piuttosto che alla virtualizzazione classica implementata tramite macchine virtuali. Le necessità che portano a questo cambio di direzione sono molteplici, quali i carichi di lavoro che queste aziende devono gestire, per soddisfare le richieste sempre maggiore da parte degli utenti. Questo volume di richieste sempre maggiori richiederebbe un commisurato aumento delle risorse disponibili nelle aziende, quali risorse hardware, e software, ai quali bisogna aggiungere una crescita del personale specializzato per gestirle. È quindi necessaria l’adozione di un sistema che possa offrire a queste aziende un modo per ridimensionare gli investimenti da sostenere, che allo stesso tempo non vada a discapito della qualità del servizio offerto. I software container sono buona opzione per lo scopo, che grazie alle loro caratteristiche possono garantire la qualità del servizio che un’azienda ha da offrire, ottimizzando le risorse hardware, già in uso, senza la necessità di effettuare altri ingenti investimenti in hardware o personale aggiuntivo.

Questa tesi si pone come obbiettivo quello di descrivere i software container, prendendo Docker come caso di riferimento, in quanto risulta essere una delle tecnologie a container più diffuse ed utilizzate. Nel primo capitolo si inquadrerà una panoramica generale sui software container, come si sono evoluti nel tempo e le motivazioni principali che hanno portato ad un loro utilizzo nell’ambito IT. Nel secondo capitolo si tratterà di Docker come caso di studio descrivendone una breve storia, il funzionamento di questa piattaforma, con alcuni semplici esempi pratici a titolo dimostrativo, e alcuni strumenti, nativi e non, che la piattaforma offre. Nell’ultimo capitolo si descriverà l’approccio di sistema software a micro-servizi e come questa, possa essere combinata con l’uso dei container, spiegando alcuni punti critici che si possano rilevare, offrendo alcune soluzioni che permettono di agevolarne la gestione.

Capitolo 1

I Software Container

In questo capitolo si descrive in generale cos'è un software container e il concetto che sta alla base di questa tecnologia, e la necessità che ha portato ad un loro sviluppo ed utilizzo nel mondo. Scendendo nel dettaglio nel paragrafo 1.1 si offrirà una panoramica generale di cosa sono i software container, comparandone brevemente gli aspetti con una tecnologia simile: le macchine virtuali. Nel paragrafo 1.2 si esporrà una breve storia sull'evoluzione della tecnologia, nel paragrafo 1.3 verranno spigate le ragioni per utilizzare software container e perché preferirli in alternativa ad altre tecnologie per poi concludere con quale soluzione è possibile o preferibile lavorare con questo approccio, nel paragrafo 1.4 visionando soluzioni proposte da alcune aziende.

1.1 Cosa sono i software container

Un container è uno spazio software in cui viene eseguita un'applicazione in modo isolato ed indipendente, che dispone di proprie risorse software (filesystem, interfaccia di rete, volumi…) rispetto le altre applicazioni e al sistema host sul quale è eseguito. Un container viene avviato a partire da un'immagine che definisce l'applicativo o il servizio che deve essere eseguito, con la relativa configurazione dell'ambiente d'esecuzione, che comprende molte delle componenti sopra citate. L'approccio di eseguire le applicazioni in questo genere di ambienti mira a soddisfare e risolvere determinate necessità e problematiche legate soprattutto alla gestione, distribuzione e manutenzione dei software da parte delle aziende in ambito informatico, e ha già visto la sua utilizzazione pratica con le macchine virtuali.
Un container si può considerare come un server (o un sistema) virtualizzato ma solo per lo spazio utente, ossia la parte virtualizzata è l'ambiente di esecuzione delle applicazioni e non tutti i componenti sottostanti.

Parlando di virtualizzazione è necessario fare una premessa in cui si descrive che cosa sia e i metodi con il quale può essere effettuata, relativamente al caso di studio.

1.1.1 Cenni di Virtualizzazione

La virtualizzazione nasce da varie necessità da parte del mondo IT quali, maggiore fra queste, è l'ottimizzazione delle risorse hardware. In informatica il termine virtualizzazione si riferisce alla possibilità di astrarre le componenti hardware, cioè fisiche, degli elaboratori al fine di renderle disponibili al software in forma di risorsa virtuale. I metodi di virtualizzazione che andremo ad analizzare sono due: Virtualizzazione a livello hardware e a livello di sistema operativo.

Virtualizzazione hardware level

In questo approccio ci si preoccupa di virtualizzare il sistema fisico sul quale eseguire il servizio. Per applicare questo metodo si ricorre all'uso delle macchine virtuali. Una macchina virtuale è un software che simula un sistema operativo, dando possibilità all'utente di ricorrere a tutte le funzionalità offerte dal sistema operativo simulato, chiamato guest; questo sistema è eseguito sul sistema operativo host, garantendo indipendenza fra i due, o meglio garantendo che ciò che viene eseguito e che succede al sistema guest, sia confinato allo stesso senza influenzare il sistema host. Per fare ciò il sistema operativo host dedica una parte di risorse hardware al sistema guest, anche se questo non ne disporrà in maniera completa; inoltre è possibile eseguire più sistemi guest sulla stessa macchina; considerando quanto detto questa tecnologia risulta utile per l'esecuzione di applicazioni ancora in fase di test in ambienti sicuri senza recare danno alla risorse fisiche, o al sistema host, inoltre è possibile eseguire più sistemi virtuali su una stessa macchina fisica piuttosto che su macchine diverse, permettendo di abbattere così i costi legati all'hardware che altrimenti un'azienda dovrebbe sostenere.

Virtualizzazione OS level, o container level

Approccio di eseguire applicazioni e servizi in un ambiente isolato, differente dal modello di virtualizzazione della macchina virtuale, la quale virtualizza fisicamente un intero sistema. Questo metodo di

virtualizzazione si dice che virtualizza il software, in pratica si tratta di creare ambienti isolati chiamati container nel quale eseguire le applicazioni, i container istanziati non sfruttano un sistema operativo ospite, ma sfruttano direttamente il kernel del sistema operativo sottostante, con chiamate di sistema. È possibile eseguire più container nello stesso host, i quali condivideranno le risorse hardware disponibili, ed utilizzandole solo all'occorrenza. Nella figura seguente si può vedere l'esempio dei due approcci.

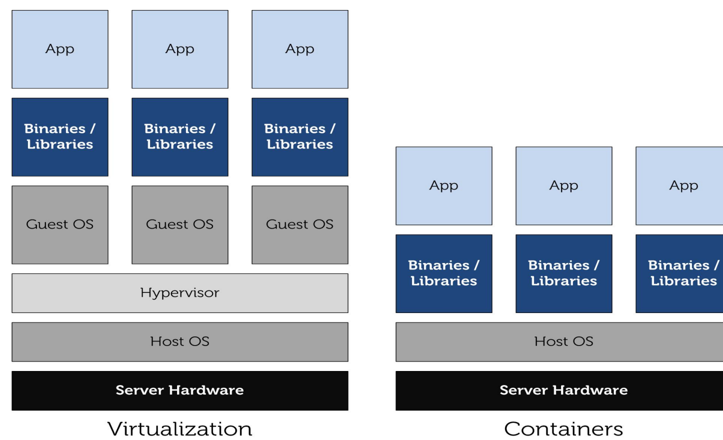

Figura1. 1: Confronto tra virtual machine e container

Definiti questi due concetti, ora si possono fare alcune considerazioni sui casi di utilizzo e quando sia opportuno utilizzare l'una o l'altra.
L'utilizzo della virtualizzazione nasce dall'esigenza delle aziende di gestire intelligentemente il deployment, il rilascio e la manutenzione delle applicazioni o servizi che mettono a disposizione. Senza contare il fatto che servono infrastrutture hardware per l'esecuzione delle stesse. Qui la virtualizzazione trova terreno fertile, in quanto è possibile incapsulare dentro un ambiente isolato e solitamente portabile i servizi di cui si necessita, cosa che è stata gestita per anni dalle macchine virtuali e adesso ci si sta dirigendo verso l'uso dei container.

1.1.2 Confronto fra virtualizzazione e software container

I containers si differenziano dalle macchine virtuali soprattutto per la loro "leggerezza", infatti una macchina virtuale, necessita di un sistema operativo ospite che viene eseguito sul sistema operativo

host, e di un controllore chiamato hypervisor per il suo funzionamento, mentre un container sfrutta il kernel e le librerie del sistema operativo sottostante per virtualizzare un ambiente privato di esecuzione delle applicazioni, condividendo le risorse (CPU, Ram, rete, memorie di massa..) con il resto delle applicazioni/container. La tecnologia più famosa, che ha reso i container una soluzione appetibile per le aziende è Docker. Questo metodo di virtualizzazione non nasce come alternativa o rivale della virtualizzazione con macchina virtuale, infatti possono presentarsi casi in cui è meglio utilizzare una o l'altra, o addirittura un approccio ibrido che vede l'esecuzione di container su macchine virtuali (come in figura 1.2).

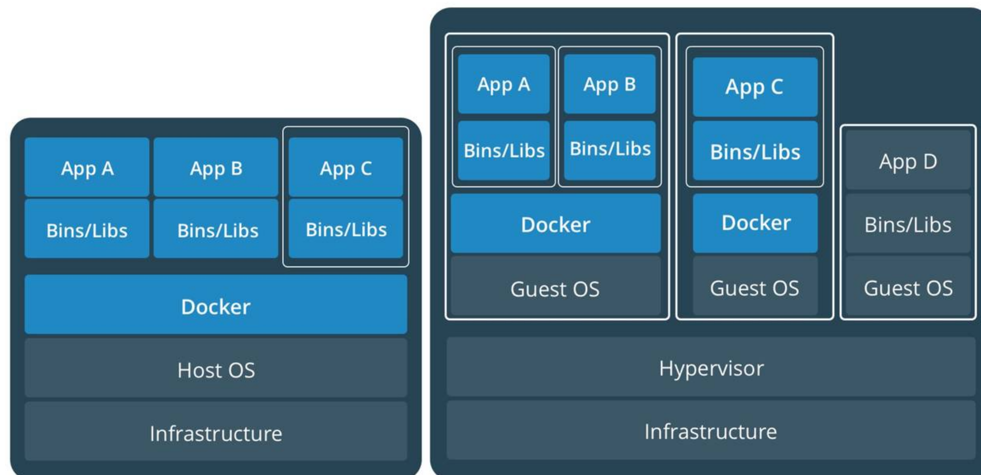

Figura1. 2: Esempio di utilizzo ibrido di virtual machine e container

1.2 Evoluzione dei software container

Nel 2013, il rilascio e la diffusione del progetto Docker portò una vasta gamma di persone a pensare che i container fossero nati con questa piattaforma, in realtà i container sono comparsi prima di Docker, e l'idea della containerizzazione o isolamento dei processi ancora prima. Nel 1979 fu introdotta nei sistemi Unix-like, la chiamata di sistema chroot (change root), la quale è in grado di cambiare la directory di riferimento dei processi in esecuzione e di quelli generati da questi ultimi (detti processi figlio). L'idea era quella di fornire ai processi degli spazi "isolati" nello storage in uso, una caratteristica che rende chroot un "precursore" dei container Linux. Nel 2000 Derrick T. Woolworth ideò FreeBDS jail introdotta nell'omonimo OS; questa soluzione, simile a chroot, aggiungeva funzionalità di sandboxing per l'isolamento dei file di sistema, degli utenti e della rete consentendo ad esempio di assegnare determinate configurazioni software ed IP a ciascuna jail. L'anno dopo uscii Linux VServer, tecnologia simile alla jail ma permetteva inoltre la suddivisione delle risorse di un computer (file di sistema, indirizzi di rete, memoria, CPU time) in partizioni chiamate "security context" e il sistema al suo interno VPS (virtual private server). Nel 2004 Sun Microsystem rilasciò Solaris Container; per la prima volta si parlò di container. Questi container (chiamati zone) combinavano il controllo delle risorse di sistema con ambienti isolati in esecuzione su un'unica istanza del sistema operativo. L'anno dopo, nel 2005, fece la sua comparsa OpenVZ virtualizzazione a container open source offerta da Virtuozzo, offre le caratteristiche comuni dei container, ma richiede una patch al kernel Linux. Nel 2006 Google sviluppa Process container, un progetto che introduce nel Linux kernel 2.6.24 i "Control groups" (cgroups). I cgroups sono una funzionalità che sta alla base di molti software container, si tratta di una funzionalità che permette di limitare, registrare ed isolare l'utilizzo delle risorse (CPU, memoria, disco I/O, ecc) di gruppi di processi, fornendo così un accesso controllato all'hardware da parte dei processi.

Nel 2008 Arrivano i LinuX Container (LXC) sono la prima soluzione che offre un completo sistema di container. Sfruttano i cgroups e i namespace di Linux, e non richiedono un'installazione di componenti aggiuntivi al kernel.

Nel 2011 Warden propone una piattaforma che sfruttasse inizialmente container LXC, per poi utilizzare qualche tempo dopo una sua versione. L'architettura di Warden si basa sul modello client-server, in cui il demone (server) agisce sui container soddisfando le richieste inviate dal client, interfacciandosi con delle API. Warden è inoltre supportato non solo da Linux, ma da più sistemi operativi.
Nel 2013 si vedono protagonisti 2 importanti progetti, il primo LMCTFY (Let Me Contain That For You) versione open source di un progetto di Google, utilizza cgroups, come molte soluzioni container già viste. Particolarità di LMCTFY è che ogni applicazione è "container aware" tradotto consapevoli, consci, significa che le applicazioni eseguite in questi contesti sono in grado di eseguire e gestire dei propri sotto container. Il progetto termina nel 2015, quando Google decide di contribuire alla nuova libreria di Docker libcontainer, inserendo in questa alcune parti appartenenti al progetto ormai chiuso. Docker, il punto centrale di questa Tesi, ora se ne fornirà una breve descrizione per poi entrare nel dettaglio nel seguito. Nasce come progetto open source all'interno dell'azienda dotCloud, e viene pubblicato nel marzo 2013. Inizialmente optò per l'utilizzo di LXC per poi passare alla soluzione coustom libcontainer. Nel 2014 CoreOS pubblicò rkt (Rocket) che si pone come valida alternativa a Docker, focalizzandosi sulla questione della sicurezza.

1.3 I container al giorno d'oggi

Dopo aver illustrato l'evoluzione dei container fino ad oggi, si passa ora ad esaminare quali siano al giorno d'oggi le soluzioni a container più gettonate dalle aziende e dal mercato. Si ricorda inoltre che i container sono tecnologie che hanno tutte alla base lo stesso scopo e quindi le soluzioni offerte sono molto simili fra loro, ma vi sono alcuni fattori che possono portare alla scelta di una piuttosto che altre; oltre a ciò linux fundation diede inizio anni fa al progetto OCI (Open Container Initiative), con lo scopo di definire uno standard per le tecnologie di containerizzazione in modo che la scelta del tipo di container sia il più irrilevante possibile, dando così ai potenziali clienti la possibilità di sfruttare i vantaggi della tecnologia senza incappare nel lock-in. Di seguito si mostreranno alcuni formati di container più diffusi ed utilizzati correntemente.

Linux Container

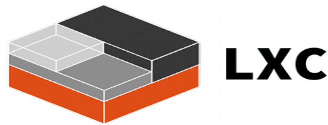

Figura1. 3: Logo LXD

I LinuX Container sono ancora un formato di container molto diffuso, Docker stesso agli albori del suo successo utilizzava LXC come ambiente di esecuzione di default, (ad oggi sostituito dalla propria libreria libcontainer). Un container Linux è un insieme di uno o più processi isolati dal resto del sistema. Poiché tutti i file necessari per eseguire tali processi vengono forniti da un'immagine distinta, i container Linux sono portabili e coerenti in tutti gli ambienti, dallo sviluppo ai test e infine alla produzione. Questo li rende molto più veloci, rispetto alle tradizionali pipeline di sviluppo che dipendono dalla replica degli ambienti di test tradizionali.
I Linux container sfruttano le funzionalità del kernel cgroups e namespace per garantire l'isolamento dell'ambiente di esecuzione dell'applicazione, senza la necessità di installare alcuna componente aggiuntiva al kernel (contrariamente ad OpenVZ). Questo, tuttavia, li rende compatibili solo su sistemi con Linux kernel.

Rkt (rock-it) CoreOS

Figura1. 4: Logo rkt

Rkt è la soluzione di container offerta da CoreOS. Si pone come diretto avversario di Docker, differendosi da questa per alcuni aspetti, principale dei quali che lo rende appetibile rispetto la concorrenza sono le feature di sicurezza aggiuntive. Di queste fanno parte, oltre che un isolamento di container basato su KVM, il supporto dell'estensione kernel SELinux, ma anche una convalida della firma per immagini della specifica di container sviluppata autonomamen-

te, ovvero App container (appc). Questa funzione descrive il formato dell'immagine App Container Image (ACI), l'ambiente di runtime, un meccanismo di image discovery, e infine anche la possibilità di raggruppare immagini in app multi-container, i cosiddetti App Container Pods. Rkt supporta il suo modello di immagini, ma anche altri formati. L'ambiente di runtime è compatibile con le immagini di Docker e con il tool open source *Quay* offre la possibilità di convertire ogni formato container secondo ACI. La struttura di rkt non ruota attorno ad un demone centrale ma i container vengono gestiti direttamente dal client, risolvendo la questione legata ai permessi di root necessari alla gestione dei comandi come fa Docker. Rkt risaluta adatto all'esecuzione di singole applicazioni all'interno dei container, quindi qualora fosse necessario implementare un sistema complesso che comprende un elevato numero di container, con questa soluzione è consigliabile utilizzare tool di orchestrazione compatibile con questo formato, uno di questi è sicuramente Kubernetes.

Docker

Figura1. 5: Logo di Docker

Docker è la piattaforma per gestione di container per eccellenza. Ha un modello di esecuzione centralizzato basato su un demone, il che implica che la gestione dei container possa effettuata con privilegi di root. Scopo di questa piattaforma è quello di creare un'ambiente di esecuzione virtualizzato per delle singole applicazioni, quindi, come per rkt, è necessario ricorre a tool di orchestrazione per operare con sistemi con molti container. Se invece si vuole creare un sistema full-stack che comprende un numero di container che è umanamente gestibili (parliamo di massimo qualche decina) allora si può valutare l'utilizzo di docker-compose, un tool per gestire un sistema che comprende più contianer che solitamente interagiscono fra loro. Docker in quanto piattaforma più diffusa, non ha cercato di

rendere compatibili nel suo ambiente di runtime immagine non native (Dockerfile), supporta quindi solo il suo formato di immagini.

Questi sono le soluzioni maggiormente diffuse, che presentano alcune differenze relativamente alla compatibilità con altri sistemi, e al tipo di container management (tipo di sistema che si vuole containerizzare) e alcune specifiche architetturali. Consideriamo le due soluzioni che ospitano una sola applicazione dentro al container: rkt, e Docker. Queste incapsulano una sola applicazione per container, il che potrebbe sembrare strano confrontandoli con i complessi sistemi che questi vanno a costruire; infatti il punto di forza dei container non sta nella quantità di applicazioni che riescono ad eseguire al loro interno, ma bensì la possibilità di interagire e comunicare fra di loro pur mantenendo un alto grado di isolamento. Questo può essere fatto tramite alcune delle funzionalità offerte da queste piattaforme e per quanto riguarda alla gestione di sistemi con numerosi container, si ricorre, come anticipato, all'uso di tool di orchestrazione. Sono strumenti che permetto di maneggiare, e delle volte automatizzare sistemi di container complessi, in modo agevole. Nel seguito della tesi si tornerà a parlare di tool di orchestrazione, e in particolare di Docker swarm, tool di orchestrazione nativo di Docker, e Kubernetes.

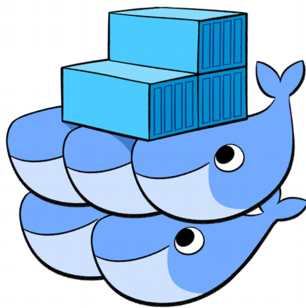

Figura1. 6: Logo Swarm

Figura1. 7: Logo Kubernetes

1.4 Vantaggi dell’uso dei software container

In questo paragrafo saranno esaminati i vantaggi dei container. Di solito i vantaggi si esprimono in relazione qualcos’altro, nel caso specifico i vantaggi dei container possono, spesso, essere messi in relazione alle macchine virtuali. Detto ciò, non si vuole esibire un confronto fra i 2 approcci, ma soltanto mostrare i vantaggi che i container possono portare in relazione o meno di altre tecnologie preferibilmente senza fare paragoni.

I motivi che portano a preferire lo sviluppo di un sistema software con container, sono molteplici:

- Sono più leggeri rispetto ad altre tecnologie (come le macchine virtuali).
- Garantiscono l'isolamento dell'applicazione in esecuzione su di essi.
- Godono di un rapido avvio del servizio contenuto al loro interno
- Sono portabili anche su sistemi differenti
- Dispongono di un controllo di versione
- Le risorse dell'host sono condivise
- Godono di granularità e flessibilità
- Scalabilità

Ora si analizzeranno i vari punti sopracitati

Leggerezza: un container può essere eseguito su una vasta gamma di dispositivi, anche se non dotati di grosse risorse (come ad esempio RaspberryPi), in quanto nel container è contenuto solo lo stretto necessario per il funzionamento dell'applicazione, cosa che non avviene con le macchine virtuali, le quali necessitano di un sistema operativo ospite, e quindi di risorse hardware da virtualizzare, inoltre il numero di container che si possono eseguire in un server, e solitamente molto più alto del numero di macchine virtuali eseguibili nello stesso server.

Isolamento: ogni applicazione/servizio viene eseguita in una propria "sandbox", in maniera indipendente da altri container/applicazioni sfruttando determinate funzionalità del kernel; inoltre lo si può limitare per fare in modo che il container possa interagire con altre ri-

sorse, come la rete, i volumi; tuttavia l'isolamento dei container non è perfetto come quello delle macchine virtuali.

Rapido avvio: L'assenza di una macchina virtuale, implica già l'assenza del tempo di avvio dell'OS ospite della macchina stessa, mentre l'avvio di un container può essere effettuato mediante una riga di comando ed è praticamente istantaneo.

Portabilità: Proprio come i container dei camion, i compute container godono di un approccio standardizzato, ciò significa che possono essere distribuiti su qualunque risorsa di calcolo indipendentemente da configurazioni, SO o hardware. Grazie a questa caratteristica, una volta completato il deployment, il rilascio e la distribuzione dell'applicativo sono velocizzati.

Controllo di versione: La tecnologia dei container permette di gestire le versioni del codice dell'applicazione e delle sue dipendenze. È possibile tenere traccia delle versioni di un container, analizzare le differenze tra di esse ed eventualmente tornare a versioni precedenti in caso di introduzione di bug negli aggiornamenti (similmente a come accade per i DVCS).

Condivisione delle risorse: ogni container vede le risorse come se fossero dedicate, anche se in realtà sono condivise con gli altri processi in esecuzione. In questa maniera le risorse verranno utilizzate solo quando vi è un effettivo carico di lavoro da svolgere, rilasciandole quando non sono utilizzate, permettendo perciò ad altri container di potervi accedere e utilizzarle; ciò tuttavia non viola l'isolamento del contesto di esecuzione del container.

Granularità e flessibilità: i container permettono di organizzare le risorse computazionali in micro-servizi migliorando le prestazioni del sistema, che potrà adattarsi in modo estremamente flessibile alle esigenze dell'azienda.

Scalabilità: Un sistema implementato tramite container, è anche scalabile. La scalabilità è data dal fatto che i servizi che sono inclusi nei container, quindi spesso per aggiungere una feature, basta aggiungere uno container, o un insieme di container che la implementa; Oltre a ciò è possibile allocare o distruggere container al fine di gestire al meglio il carico di lavoro a cui far fronte in momenti di-

versi (scalabilità orizzontale) .Inoltre ad un aumento delle prestazioni hardware, può spesso corrispondere un proporzionale incremento della performance del sistema, ad esempio in termini di velocità (scalabilità verticale).

Capitolo 2

Docker

In questo capitolo si tratterà di Docker come piattaforma di riferimento dei software container, nel paragrafo 2.1 si offrirà una panoramica generale su Docker introducendo l’argomento, nel paragrafo 2.2 si descriverà la struttura architetturale di Docker, con le componenti e le relative funzioni. Dopo queste spiegazioni nel paragrafo 2.3 si mostrerà qualche esempio di utilizzo, a partire dall’installazione. Negli ultimi 2 paragrafi si mostreranno sue tool nativi di gestione di sistemi multi container di Docker: Compose nel paragrafo 2.4, e Swarm, nel paragrafo successivo.

2.1 Cos’è Docker

Docker è una delle piattaforme software container più utilizzate e diffuse nell'ambito della virtualizzazione a livello OS. Il progetto Docker nacque nell’azienda dotCloud, e fu pubblicato come progetto open source a marzo 2013. Il successo del progetto fu tale che nell’ottobre dello stesso anno la società fu rinominata con il nome dell’omonima tecnologia. Il successo lo si può attribuire a molteplici fattori, principali dei quali la condivisione di progetti su DockerHub. DockerHub è il registry nativo offerto da Docker ove è possibile caricare/scaricare i progetti sviluppati con la piattaforma, consentendo a moltissimi interessati al progetto di conosce e sperimentare questa tecnologia in modo gratuito. Inizialmente Docker sfruttava i container LXC, per poi passare all’utilizzo della libreria libcontainer scritta in linguagio GO

2.2 Architettura di Docker

Docker utilizza un’architettura client-server. Il Docker client comunica con il Docker deamon, chiamato dockerd i quale fa “il grosso del lavoro” come buildare, mandare in esecuzione e distribuire i container. Il client e il demone Docker possono essere eseguiti nella stessa macchina oppure un Docker client può contattare un Docker

deamon remoto. I due comunicano utilizzando un set di API REST, tramite una socket unix o un'interfaccia di rete.

Il Docker deamon: Il Docker deamon (demone Docker o dockerd) ascolta le richieste delle Docker API
e gestisce gli oggetti Docker, quali le immagini, i container, le porte, i volumi e la rete. Il demone può anche comunicare con un demone remoto per la gestione di sevizi docker, come la condivisione o il download di immagini.

Docker client: Il Docker client (docker) è il modo di interfacciarsi con il sistema docker, ovvero comprende i comandi da inviare al Docker deamon quale li processerà. I comandi docker usano le api di Docker. Un docker client può comunicare con più di un demone Docker.

Docker registries: Il Docker registry (registro) memorizza le immagini Docker. Un esempio di registry è certamente DockerHub, elemento citato più volte nel corso della tesi, è uno dei punti di forza di Docker, consultabile da tutti in maniera gratuita, ha dato una grossa mano nella diffusione del progetto, offrendo immagini già pronte all'uso e innumerevoli progetti ed esempi di semplici applicazioni. La presenza di queste immagini già pronte è essenziale per la comunità Docker, in quanto permette di avere delle basi di partenza di un servizio già testate e in costante manutenzione e aggiornamento, liberando così gli sviluppatori delle fasi preliminari dello sviluppo dei container consentendogli di concentrarsi solo sulla customizzazione dell'applicativo.

Docker Object: Utilizzando Docker si va ad utilizzare una serie di oggetti quali immagini, container, reti, volumi, plugin ecc… Di seguito saranno descritti alcuni di questi oggetti focalizzandosi su quelli principali.

Immagini: Un'immagine è un file di solo lettura contenente le istruzioni che hanno lo scopo di creare container. Solitamente un'immagine si basa su una o più immagini già esistenti, e a partire da questa si susseguono le istruzioni che servono a personalizzare l'immagine al lavoro che dovrà svolgere. Tutto questo viene gestito tramite il Dockerfile, scrivendo passo passo tutte le istruzioni da eseguire per creare l'immagine ed eseguirla. Ogni istruzione del Dockerfile si

traduce in uno strato dell'immagine. Se un'istruzione viene modificata, alla ricompilazione dell'immagine le modifiche saranno apportate solo allo strato corrispondente all'istruzione modificata.

Container: Un container è un'istanza eseguibile dell'immagine. Tramite le API di Docker, utilizzabili da terminale o eventuali tool grafici, è possibile gestire le operazioni sui container come l'esecuzione, fermarli, metterli in pausa, riavviarli, cancellarli. Normalmente un container gode di un buon isolamento, e lo si può tramite le configurazioni espresse nel Dockerfile, come ad esempio la rete, archiviazione o altri sottosistemi sottostanti. Un container è definito dalla sua immagine esattamente dalle configurazioni che gli vengono fornite alla sua creazione e al suo avvio. In fugura 2.1 si può vedere l'architettura di Docker.

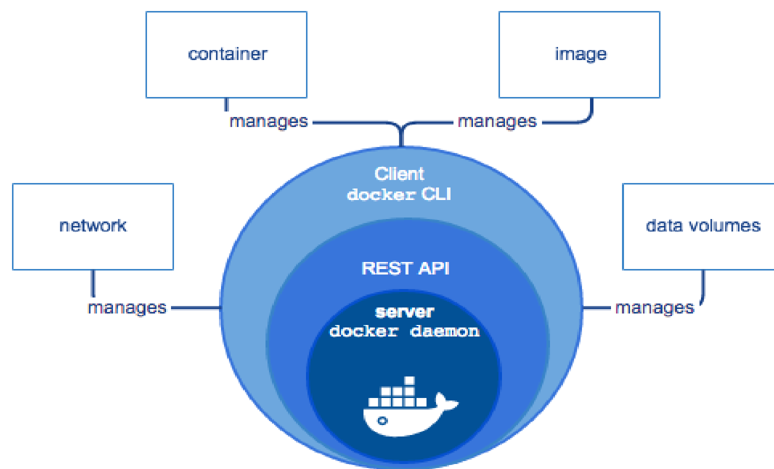

Figura2. 1: Architettura Docker

Teconologia sottostante: Docker è scritto in linguaggio Go a sfrutta diverse funzionalità del kernel Linux per fornire il suo servizio.

Namespace: Come anticipato, Docker ha bisogno di un kernel Linux per la sua esecuzione per il fatto che sfrutta delle sue funzionalità, namespace è una di queste e se ne esamineranno altre. Si tratta di una tecnologia che fornisce uno spazio utente isolato chiamato container. Questo fornisce uno strato di isolamento. Ogni aspetto del container è eseguito in un namespace separato e il suo accesso è

limitato a quel namespace. Docker engine usa i seguenti namespace di Linux:

- pid namespace: isolamento del processo (Process ID)
- net namespace: gestione dell'interfaccia di rete (networking)
- ipc namespace: gestione dell'accesso alle risorse IPC (Interr process comunication)
- mnt namespace: gestione del filesystem e unità montabili (mount)
- uts namespace: isolamento del kernel (unix timsharing system)

Control Group: Docker engine si basa anche su questa funzionalità di linux che sono i control groups (cgroups). Cgourps limita l'utilizzo da parte del container delle risorse hardware. I Cgroups consentendo a Docker engine di condividere/concedere le risorse hardware disponibili ai container, e opzionalmente limitarne e vincolarne l'utilizzo (come ad esempio settare il numero di processori che un container può utilizzare).

Union file systems: Union file systems o UnionFS sono dei file systems che si occupano di creare strati leggeri e veloci. Per lo scopo, Docker engine utilizza UnionFs per fornire una costruzione a blocchi dei container. Docker engine utilizza varie possibilità di UnionFS quali: AUFS, btrfs, vfs, and DeviceMapper.

Container format: Docker engine combina i namespace, i cgroups, e UnionFS in un involucro chiamato container format. Di default è libcontainer. In futuro Docker potrebbe supportare altri formati di container per integrarsi con altre tecnologie come BSD, jails o Solaris Zone.

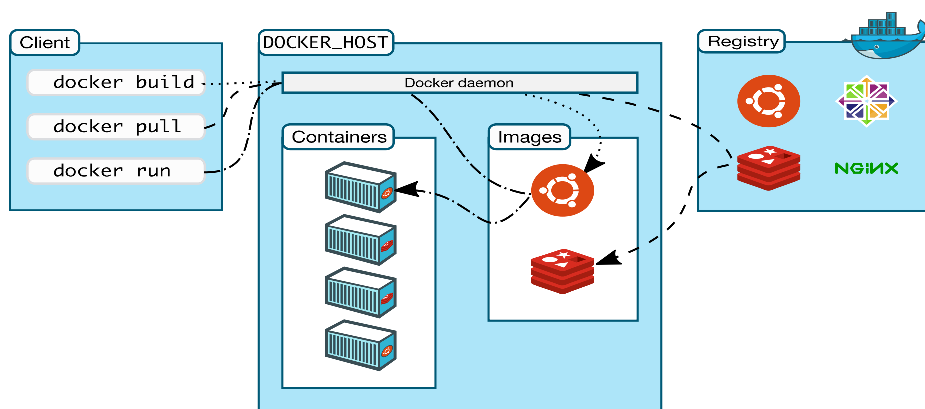

Figura2. 2: Fuzionamento della piattaforma Docker

2.3 Come funziona Docker

Definita l'architettura e la parti di cui è composta la piattaforma Docker se ne darà un esempio pratico, illustrando la fase di installazione e di operazioni preliminari all'uso nella sottosezione successiva, mentre nella sottosezione 2.3.2 si mostreranno alcuni comandi su come iniziare a conoscere ed utilizzare la piattaforma con l'esecuzione di alcune immagini disponibili su Docker Hub. Si conclude il paragrafo parlando di Dockerfile, il concetto che sta alla base delle immagini di Docker e quindi alla creazione dei container

2.3.1 Installazione

Ecco l'installazione docker tramite package manager di un sistema Ubuntu:

```
$ sudo apt-get update

$ sudo apt-get install --no-install-recommends \
      apt-transport-https \
      curl \
      software-properties-common
```

Opzionalmente si possono i moduli del kernel per aggiungere il supporto AUFS.

```
$ sudo apt-get install -y --no-install-recommends \
      linux-image-extra-$(uname -r) \
      linux-image-extra-virtual
```

Scaricare ed importare la chiave pubblica di Docker:

```
$ curl -fsSL 'https://sks-keyservers.net/pks/lookup?
op=get&search=0xee6d536cf7dc86e2d7d56f59a178ac6c6238
f52e' | sudo apt-key add -
```

Aggiungere il repository, all'interno del comando sotto, si usa il sottocomando lsb_release -cs che ritorna il nome della versione di Ubuntu in uso.

```
$ sudo add-apt-repository \
      "deb https://packages.docker.com/1.13/apt/repo/ \
      ubuntu-$(lsb_release -cs) \
      main"
```

Installare l'ultima versione di Docker engine:

```
$ sudo apt-get update
```

```
$ sudo apt-get -y install docker-engine
```

Confermare l'esecuzione del demone docker:

```
$ sudo docker info
```

Opzionale, la gestione dei container avviene tramite il demone docker, che di default richiede i privilegi di root, questo comando dà all'utente attuale i permessi di eseguire docker (altrimenti è necessario eseguire i comandi di gestione dei container con seguiti da sudo)

```
$ sudo usermod -a -G docker $USER
```

2.3.1 Alcuni comandi

Ora che il tool è stato installato, si esporranno alcuni dei comandi base per utilizzare la piattaforma. Una prima prova che si può fare è eseguire la shell di linux in un container Docker:
digitando nel terminale il comando `$ sudo docker run ubuntu -it`. Il prefisso *sudo* è necessario in quanto per comunicare con il demone docker sono necessari permessi di amministratore, *docker* indica che si vuole richiamare un'API di docker, *run* esegue l'immagine in un container, i due flag che forniscono al container alcune caratteristiche -*i* (interactive) serve per interagire con il container, e -*t* serve per usufruire della tipica interfaccia del terminale. Infine, *Ubuntu* è l'immagine che si desidera eseguire. Una precisazione è necessaria per quanto riguarda l'immagine, se l'immagine è disponibile in locale, allora questa verrà eseguita, altrimenti dockerd consulterà il registry per cercarla fra i progetti della commuty nel caso specifico l'immagine è stata scaricata al lancio del comando run. Ora il container è in esecuzione, il risultato si può vedere nell'immagine successiva (Figura 2.3).

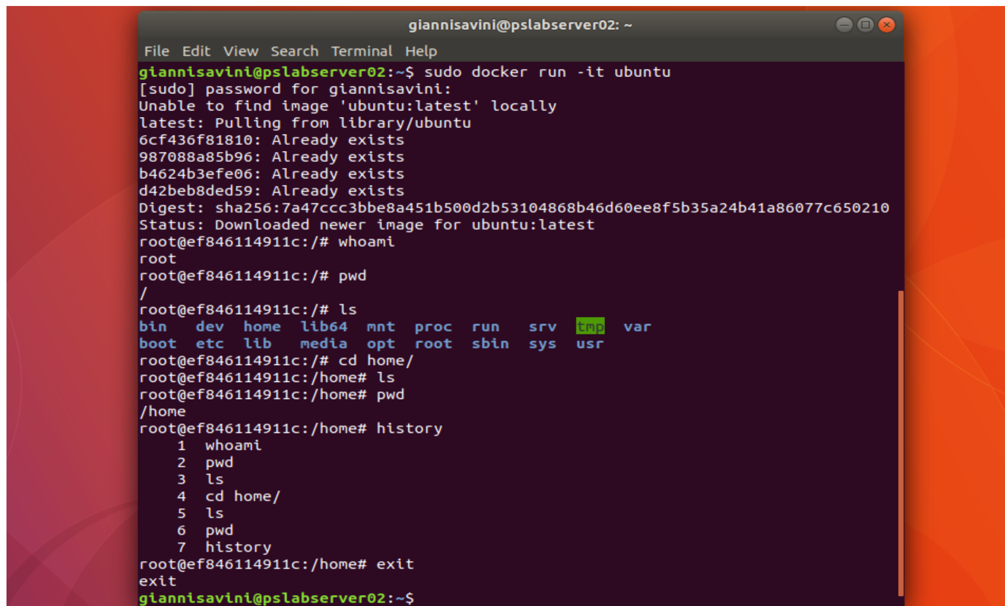

Figura2. 3: Esecuzione di immagine Ubuntu su container

Lanciando il comando *exit* si nota il ritorno al terminale del sistema host e non del container. Ora il container è stato di stop.
Nell'esecuzione dell'immagine non ci siamo preoccupati di gestire comandi relativi ai container, infatti, docker ha costruito automaticamente un container per l'esecuzione dell'immagine, con le caratteristiche specificate dai flag. Nel prossimo esempio si vedrà un utilizzo di porta. Questa volta si scaricherà l'immagine in precedenza con il comando `$ sudo docker pull brice-burg/ping-pong` da Docker hub. Dopo di che, seguendo la documentazione fornita dai mantenitori dell'immagine, si lancia il comando `$ sudo docker sudo docker run -p 7777:80 -it --rm briceburg/ping-pong` per avviare il container. I flag che compaiono sono -p 7777:80, il quale apre una porta, opportunamente specificata, per comunicare con il container. -rm è un flag che elimina il container al termine della sua esecuzione. Infine, si apre un altro terminale digitando: `$ curl localhost:7777/ping`. Si nota in figura 2.4 che il servizio ha risposto ping. Concludo questo esempio con considerazioni: se il container è in stato di running, questo risponde pong alla ricezione di ping, se è in pausa, la porta rimane aperta ma il servizio non disponibile; in fase di stop il container non risponde (In quanto è non solo non attivo, ma eliminato).

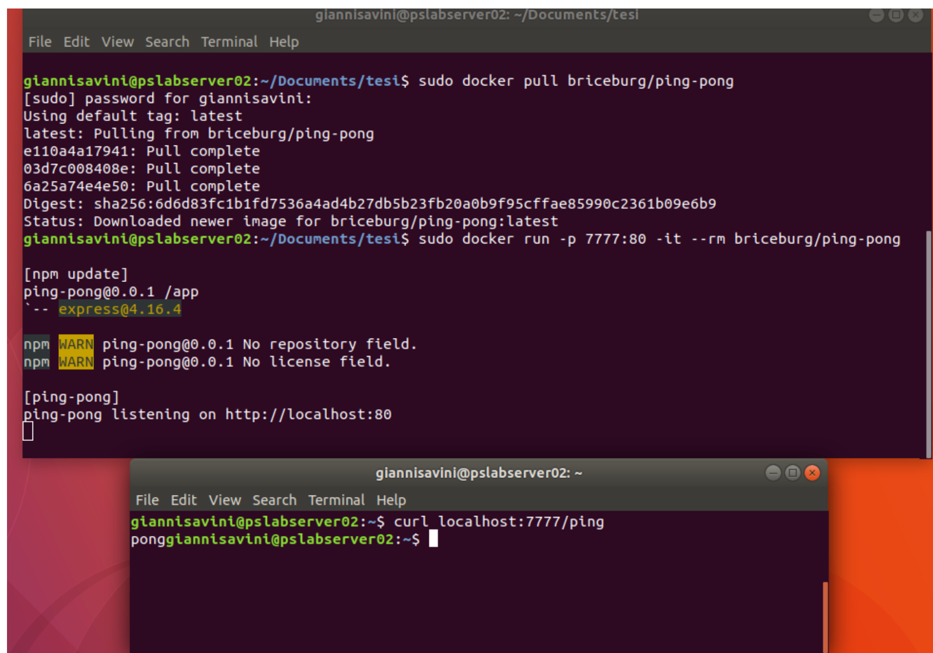

Figura2. 4: Esempio di esecuzione su container di applicazione ping-pong

2.3.3 Dockerfile

Il Dockerfile è ciò che definisce ciò che un container docker deve svolgere quando è in esecuzione. Lo si può associare ad uno “stampo” per container, infatti da un’immagine possono essere creati più container. L’uso di un Dockerfile si vede necessario qualora si debba personalizzare un servizio già esistente oppure che non esista nel Docker Hub. Nel Dockerfile sono descritte le funzionalità che l'applicazione containerizzata avrà una volta in esecuzione, nello specifico si tratta di utilizzare una sequenza di comandi che ne definisce gli strati. Ogni strato è indipendente dagli altri, e qualora sia necessario modificare uno strato, in fase i build solo lo strato modificato sarà soggetto a ricompilazione. Di seguito si riportano alcuni comandi del Dockerfile come i seguenti:

FROM <image>:<tag-version>: si specifica la base di partenza dell'immagine che si sta creando e la versione che si desidera utilizzare. Tuttavia, c'è modo di creare immagini di base facendo a meno dell'utilizzo di questo comando.

ENTRYPOINT ["cmd", "parameter"]: è il primo comando che viene eseguito quando l'applicazione sarà eseguita sul container.

RUN <command>: esegue un comando del kernel linux, utile per esempio ad installare e risolvere dipendenze una volta che l'immagine base è stata scaricata.

EXPOSE <port> [<port>/<protocol>...]: permette al container di rimanere in ascolto nella <port> porta tramite protocollo /protocol

COPY <source> <detination> copia un file dal filesystem host, al filesystem del container.

ADD: Come COPY, ma supporta la possibilità di recuperare pacchetti da URL remoti

ENV: setta una variabile d’ambiente nel nuovo container

MANTAINER: Opzionale, definisce nome ed e-mail del mantenitore dell’immagine

VOLUME: crea un volume condivisibile con altri container e il sistema host

WORKDIR: setta la cartella di lavoro di default

#: Simbolo per l’inserimento dei commenti

Ecco come compare un Dockerfile (Figura 2.5)

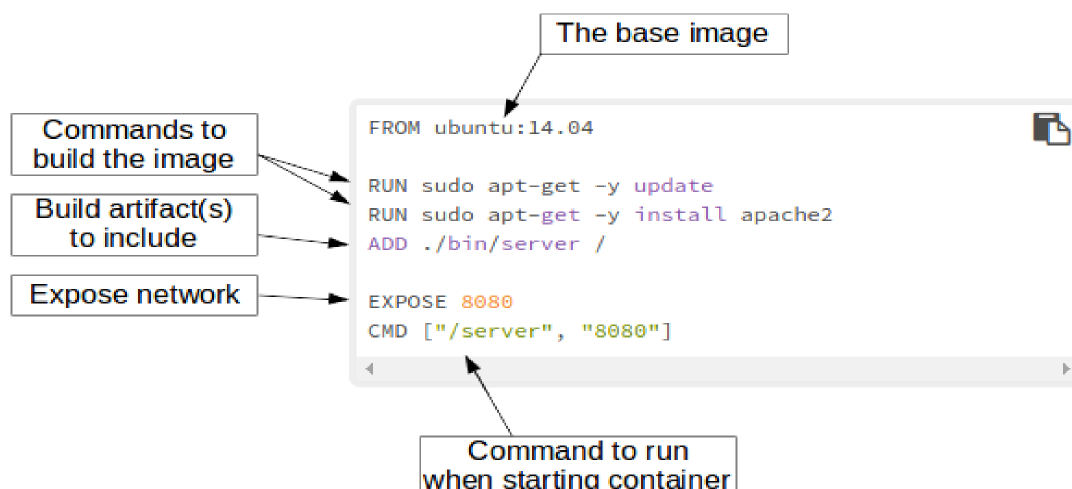

Figura2. 5: Come si presenta un Dockerfile

Una volta scritto il Dockerfile si esegue il comando $ sudo docker build nella directory ove il Dockerfile è locato, e di seguito verrà mostrato nel terminale la compilazione dell'immagine comando per comando, al termine del quale l'immagine è creata. Ora che l'immagine è buildata, la si può eseguire in un container con il comando $ sudo docker run imageid o $ sudo docker run imagename , in cui imageid è l'id assegnata all'immagine e imagename è il nome dell'immagine, con possibilità di specificare tramite i flag altri parametri di esecuzione del container come quelli visto nel precedente paragrafo. Ora che il container è in esecuzione lo è anche la nostra applicazione/servizio. C'è anche la possibilità di creare immagini senza partire da una base già esistente.

2.4 Docker Compose

Docker compose è uno strumento per gestire applicazioni Docker multi-container.
Di fatto, un container Docker è istanza eseguibile di una solo immagine, quindi una sola applicazione, questo potrebbe sembrare un limite per la piattaforma, mentre se ben sfruttato più rivelarsi il suo punto più forte. Un'applicazione separata in unità logiche come i container permette di suddividere le responsabilità di questa su più personale, riducendone per ognuno il carico di lavoro da svolgere, parallelizzando il lavoro e di liberarsi da problematiche legate alle dipendenze da altre parti di sistema. Docker compose è un tool che a partire da immagini già pronte, realizzate personalmente o di implementazione esterna, permette di mettere in piedi un sistema di container indipendenti e che interagiscono fra di loro al fine offrire il servizio completo. "Il tutto è più della somma delle parti" calza a pennello nei sistemi multicontainer. L'utilizzo di compose non è compreso nella distribuzione di base di Docker, ma è necessario installarlo a parte. Come per i classici container, per creare uno stack è necessario definire un file YAML (docker-compose.yml) in cui sono specificati quali container deve avere il servizio, e le configurazioni relative alle porte, volumi e altre risorse di cui l'applicativo disporrà. Uno volta definito il file, si lancia il comando $ sudo docker-compose build nella directory del file, e $ sudo docker -compose up per avviare il servizio, ovvero avviare tutti i container che lo compongono.

2.5 Docker Swarm

Docker Swarm permette di scalare le applicazioni in contenitori, eseguendole in un numero qualsiasi di istanze su qualsiasi numero di nodi su un network. Swarm, nato dagli stessi sviluppatori di Docker, raggruppa un qualsiasi numero di host docker in un unico cluster, permettendo la gestione centralizzata dei cluster e l'orchestrazione dei contenitori. Fino alla versione di Docker 1.11, Swarm doveva essere installato come tool separato, mentre le versioni più recenti della piattaforma swarm come funzionalità nativa, così è possibile utilizzarlo installando Docker Engine. Swarm si basa su un'architettura master-slave. Ogni cluster di Docker (lo sciame) dispone di almeno un nodo manager e di una quantità a piacere di nodi lavoratori (worker). Mentre il manager è responsabile per l'amministrazione del cluster e la delegazione di compiti, i worker prendono in carico l'esecuzione di unità lavorative ("tasks"). Un task è container in esecuzione che è parte di un servizio dello swarm ed è gestito dallo swarm manager, contrariamente a quanto accade per un servizio a container unico. Inoltre, le applicazioni del contenitore vengono suddivise come cosiddetti servizi ("services") in un numero qualunque di account Docker. Nella terminologia di Docker il concetto di "servizio" indica una struttura astratta mediante la quale potete definire compiti che devono essere eseguiti nel cluster. Ogni servizio consta di un set di task singoli, che vengono elaborati ciascuno nel proprio contenitore su uno dei nodi del cluster. Docker Swarm supporta due modalità nelle quali sono definiti i servizi Swarm: servizi replicati e globali.

*Servizi replicati***:** un servizio replicato è un task che viene eseguito in un numero di repliche definite dall'utente. Ogni replicato è un'istanza del container definito nel servizio. I servizi replicati vengono scalati man mano che gli utenti creano più repliche. Un server web come NGINX si può ad esempio impostare a una sola riga di comando a 2, 4 o 100 istanze a seconda delle esigenze.

Servizi globali: eseguendo un servizio in modalità globale, ogni nodo disponibile nel cluster inizia un task per il relativo servizio. Se aggiungete un nuovo nodo al cluster, lo swarm manager gli assegna immediatamente un'attività del service globale. I service globali si prestano ad esempio ad applicazioni di monitoraggio o programmi antivirus.

Un campo centrale di applicazione di Docker Swarm è la ripartizione del carico. Nella modalità swarm, Docker dispone di funzioni integrate di load balancing, ovvero bilanciamento del carico di lavoro. Se per esempio eseguite un server web NGINX con 4 istanze, Docker distribuisce in modo intelligente le richieste in arrivo alle istanze dei server disponibili.

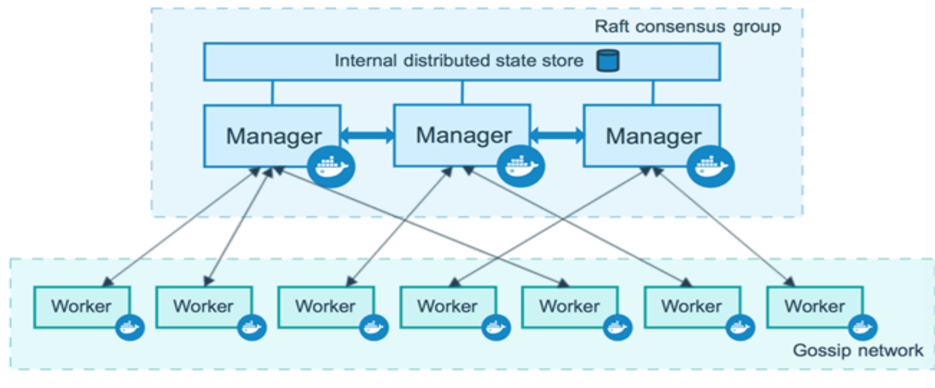

Figura2. 6: Architettura Docker Swarm

Capitolo 3

Docker e i micro-servizi

Ora che abbiamo una panoramica generale di cosa sia la piattaforma Docker, e di come la si utilizza, si possono fare alcune considerazioni di come utilizzarla nell'ambito di sistemi a micro-servizi. In questo capitolo si offrirà una panoramica nel primo paragrafo sui micro-servizi e come questi incontrino Docker, facendo una premessa su questi nel sotto paragrafo 3.1.1. Nel secondo paragrafo si parlerà di come questi si possano incapsulare dentro i container, e quali tool possono essere utili allo scopo, finendo di parlare di orchestrazione con Kubernetes.

3.1 Linee guida

Lo sviluppo di applicazioni è uno scenario ampio e estremamente diversificato, il che rende ampie le soluzioni e gli approcci che si possono adottare. Uno degli approcci possibile è quello "monolitico", ovvero si tratta di sviluppare applicazioni in cui ogni componente viene creato all'interno di un unico elemento. Questo approccio risulta essere più facile da affrontare e permette uno sviluppo iniziale del progetto più rapido, a scapito di agilità nei lavori futuri e manutenzione. Tendenzialmente può trovare applicazione allo sviluppo di progetti di dimensioni ridotte, e/o di progetti soggetti a pochi cambiamenti. Un approccio più interessante è quello di suddividere un sistema monolitico in micro-servizi. In questo modo è possibile la ripartizione delle responsabilità del sistema in più parti di applicazione, generalmente indipendenti fra di loro. Questo approccio richiede una fase iniziale più lunga rispetto all'approccio monolitico che comprende una parta di analisi e di progettazione del sistema, ma traendo benefici in termini di manutenibilità e predisposizione dell'applicativo a lavori futuri.
In questo paragrafo si darà una panoramica sui micro-servizi, con annesso la combinazione che questi trovano con la containerizzazione in particolare con la piattaforma Docker.

3.1.1 Premessa sui micro-servizi

La piattaforma Docker incontra numerosi casi di utilizzo che comprende anche il mercato di aziende medio/grandi. Risulta molto utile qualora si abbia necessità di suddividere un sistema "monolitico" in molti micro-servizi, o nella gestione di sistemi che presentano uno stack di servizi, come servizi web. Inoltre, questa separazione ed indipendenza fra le varie componenti di un’applicazione facilità il riuso di parti di applicazione in modo agevole e semplificato
I micro-servizi sono un approccio architetturale alla realizzazione di applicazioni. Quello che distingue l'architettura basata su micro-servizi dagli approcci monolitici tradizionali è la suddivisione dell'app nelle sue funzioni di base. Ciascuna funzione, denominata servizio, può essere compilata e implementata in modo indipendente. Pertanto, i singoli servizi possono funzionare, o meno, senza compromettere gli altri. Tutte caratteristiche comuni ai container, cosa che rende Docker e i software container una perfetta combinazione nello sviluppo di sistemi software complessi che necessitano di una gestione da team di sviluppo ed un continuo aggiornamento del sistema. Se pensiamo ad una azienda che deve gestire il suo sito web personale, che opera con un’architettura a micro-servizi, Docker si rivela una valida opzione: un sito comprende un server, un database, possibili applicazioni ecc.… come si è già visto nei capitoli precedenti, con Docker è possibile incapsulare i vari micro-servizi in container che comunicano tra di loro con delle porte specifiche, e nel caso usare tool per la gestione di uno stack o orchestrare i container. Ai fini di aver un sistema facile da amministrare/manutenere è bene incapsulare al meglio i servizi nei relativi container; ciò significa che se in sistema multi-container un container viene aggiornato/sostituito, questo non deve recare danno agli altri container.

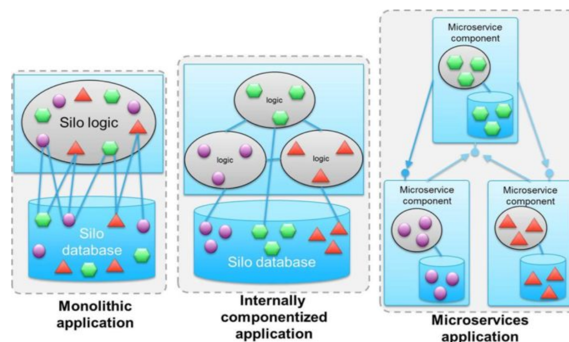

Figura3. 1: Architettura di sistemi monolitici e a microservizi

3.2 Docker e Micro-servizi

Visti i vantaggi dell’uso dei micro-servizi e dell’importanza che hanno al giorno d’oggi, lo step successivo è quello di incapsularli in ambienti virtualizzati, per migliorare ulteriormente la loro scalabilità e distribuzione. L’utilizzo di macchine virtuali è una soluzione, ma molti servizi necessitano di poche risorse per il loro funzionamento, e quindi questa soluzione risulterebbe ingombrante, e poco ottimizzante delle risorse hardware. Si può invece valutare l'utilizzo dei container, in particolare, Docker. Così facendo si può semplicemente scalare orizzontalmente le risorse disponibili, in quanto l’aggiunta di un container, ha un dispendio di risorse hardware minore rispetto la macchina virtuale. Scalare orizzontalmente si intende l’aggiunta di una o più risorse al pool di risorse già disponibili per gestire la quantità crescente di traffico. Questo approccio risulta complicato da gestire se i container da gestire sono molti, come accade nel mondo del cloud computing. Allo scopo, è possibile gestire i container per effettuare scaling con l’ausilio di tool di orchestrazione come Swarm, già citato nel capitolo 2, e Kubernetes. Kubernetes è il tool di orchestrazione offerto da Google, ed è anche il più utilizzato, anche più di Swarm. Swarm offre una gestione dei container più semplice, rispetto a Kubernetes, mentre quest’ultimo è significativamente più complesso di Swarm e richiede più lavoro per la sua implementazione, ma questo sforzo iniziale è destinato a fornire un grande profitto nel lungo periodo. In generale, per lavori di sviluppo e cluster di container più piccoli si usa tendenzialmente Swarm, mentre per sistemi più grandi e complessi, Kubernetes è la soluzione preferibile. Kubernetes è un progetto open source realizzato da Google che automatizza il processo di distribuzione e gestione di applicazioni multi-container su vasta scala. Sebbene Kubernetes funzioni principalmente con Docker, può funzionare anche con qualsiasi sistema di container conforme agli standard OCI (Open Container Initiative) per i formati di immagine e i runtime dei container. Come anticipato, Kubernetes è un progetto open source, ciò significa che chiunque voglia utilizzarlo, per gli scopi più vari, può farlo (con esigue restrizioni). Kubernetes ha un’architettura complessa che permette di organizzare l’esecuzione dei container in *cluster.* Un cluster è una macchina (fisica o virtuale) sul quale è in esecuzione Kubernet, Ogni cluster ha più nodi, di cui almeno uno è il *master*, ovvero il nodo che dirige altri nodi chiamati *worker.* Il master si occupa di gestire il lavoro dei worker, mentre i worker

sono le unità lavorative. Nei worker (anche detti *minion)* viene distribuito il carico di lavoro che deve essere elaborato, in delle unità dette *pod.* Un pod è la minima unità logica disponibile in kubernetes, in pratica si tratta di un container o un agglomerato di questi. Questi container vengono continuamente creati e distrutti in base alle esigenze dei carichi da gestire, o in base alla distribuzione di versioni aggiornate o altri compiti di gestione; Queste volatilità dei container, in quanto entità non persistenti, rende necessario, un livello di astrazione che non dipenda dai pod. Questa astrazione si chiama service, la quale definisce un insieme logico di pod e una politica con cui accedervi.

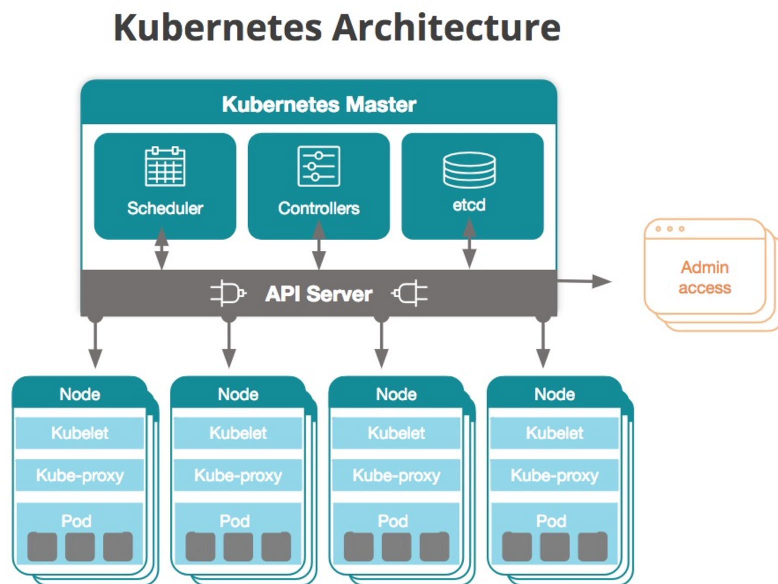

Figura3. 2: Architettura Kubernetes

Conclusioni

Ora che è stato definito un panorama generale sulle tecnologie di containerizzazione, con Docker come riferimento, si possono fare alcune considerazioni. Nell'introduzione si è parlato delle esigenze delle aziende di disporre di un sistema che potesse garantire il servizio offerto, contenendo i capitali da investire; ed ecco che i software container si propongono come la miglior strada percorribile in questo ambito, grazie agli ingenti vantaggi che sono in grado di offrire. In molti ambiti sono in grado di sostituire i sistemi basati sulle macchine virtuali, senza andare a discapito delle performance del sistema, e garantendo un migliore sfruttamento dell'hardware sottostante, infatti il numero di container eseguibili in un host, è solitamente maggiore rispetto al numero di macchine virtuali eseguibili sullo stesso, incrementando così il numero di risorse IT che servono al servizio. In ciò altri fattori si aggiungono a decretare i container come scelta miglior opzione in questo ambito quali la scalabilità, creando risorse aggiuntive solo quando è necessario, la portabilità, garantendo che il servizio sia multipiattaforma, e la separazione del sistema in più parti indipendenti le une dalle altre rende i servi facilmente riusabili in altri progetti eseguendo un numero esiguo, se non nullo di modifiche. Questa tecnologia è uno strumento formidabile per soddisfare la sempre crescente domanda di servizi digitali, che è destinato crescere ancora negli anni a venire, e a consolidarsi sempre di più. "*Everything at Google runs in a container*" è una frase che Joe Beda affermò in una conferenza del 2014, che fa capire a che punto fossero arrivati i container già anni fa, quando ancora erano tecnologie poco conosciute. Nel 2017 meno del 20% delle organizzazioni globali eseguiva applicazioni containerizzate e si stima che questo numero potrebbe arrivare fino al 50% entro il 2020. A fronte di questi numeri e dei benefici esplorati negli scorsi capitoli, è facile pensare che la distribuzione di software tramite container diventerà uno standard per la stragrande maggioranza degli ambienti informatici, non solo per "rimanere al passo" con lo sviluppo, ma proprio per questioni di convenienza nell'uso di queste nuove tecnologie. Tuttavia, è bene analizzare tutti gli aspetti di queste tecnologie, una in particolare che nel corso della tesi è stata trattata più marginalmente è la questione della sicurezza, infatti uno dei motivi per il quale molte organizzazioni temono un intensivo utilizzo dei

container all'interno delle proprie strutture è la sicurezza. In linea di principio un container è un ambiente isolato da tutto il resto del sistema, inaccessibile dall'esterno se non tramite delle porte che offrono un modo di comunicare, o l'accesso a risorse condivise delle quali i container non ne bloccano l'utilizzo. Questi timori si sono rivelati fondati, dalla recente scoperta a Febbraio 2019 da parte di due ricercatori di sicurezza, Adam Iwaniuk e Borys Popławski, su una grave vulnerabilità identificata come CVE-2019-5736 in *runc,* la libreria utilizzata per implementare l'astrazione dei container, riconducibile ad una errata gestione dei link simbolici. "*La vulnerabilità permette ad un container malevolo di riscrivere (con una minima interazione dell'utente) i binari runc dell'host ed in questo modo guadagnare i privilegi di root sullo stesso.*". Docker ha già rilasciato la versione v18.09.2 che integra la soluzione per la vulnerabilità della tecnologia container. Gli altri produttori sono al lavoro sugli aggiornamenti necessari. Questa vulnerabilità potrebbe lasciare fare dei passi indietro agli utenti delle piattaforme, tuttavia gli aggiornamenti sulla correzione del bug sono in corso, e inoltre si può valutare di integrare l'utilizzo delle macchine virtuali con i container, usufruendo così dei vantaggi dell'una e dell'altra tecnologia, ovvero del perfetto isolamento di un sistema offerto dalle macchine virtuali e di tutti i pregi già verificati da queste, e dei vantaggi già citati dei container. Questo appacio infatti non risulta troppo differente dal sistema di kubernet, in cui i nodi dove vengono eseguiti i compiti possono essere macchine sia fisiche che virtuali. Vedremo quindi nel prossimo futuro un continuo evolversi e diffusione dell'utilizzo di queste piattaforme.

Ringraziamenti

Vorrei ringraziare innanzitutto la mia famiglia, in particolare i miei genitori Gianluigi ed Ivana che mi hanno sostenuto in questo percorso, moralmente ed economicamente anche nei momenti più critici della carriera. Un sentito ringraziamento va a tutti i miei professori che ho incontrato in questi anni che mi hanno formato, in particolare vorrei ringraziare il Prof. Alessandro Ricci, il mio relatore, il Prof. Angelo Croatti, mio correlatore, ed il professore Ghini per avermi seguito con pazienza ed attenzione nella fase finale del mio percorso. Inoltre, vorrei ringraziare anche i miei colleghi, in particolar modo, Nicola Stradaioli, Luca Casamenti, Orjada Bardhi e Olexander Melny Chuck, i miei compagni di progetti, nonché compagni di classe alle superiori.

Bibliografia

cwi.it. (s.d.). *Kubernetes: tutto quello che bisogna sapere sulla container orchestration.* Tratto da https://www.cwi.it/applicazioni-enterprise/sviluppo-app/kubernetes-111490

hane, O. (2015). *Build Your Own PaaS with Docker.*

internetpost.it. (s.d.). *Container Linux: storia della tecnologia.* Tratto da https://www.internetpost.it/container-linux-storia-tecnologia/

ionos.it. (s.d.). *Alternative a Docker: una rassegna delle piattaforme per la gestione dei container.* Tratto da https://www.ionos.it/digitalguide/server/know-how/una-panoramica-delle-alternative-a-docker/

Kyoung-Taek Seo, H.-S. H.-Y.-Y. (s.d.). *Performance Comparison Analysis of Linux Container and Virtual Machine for Building Cloud.*

Maida, I. (s.d.). *I VANTAGGI DEI CONTAINER.* Tratto da https://www.criticalcase.com/it/blog/i-vantaggi-dei-container.html

Mocevicius, R. (2015). *CoreOS Essentials.*

Rajdeep Dua, A. R. (s.d.). *Virtualization vs Containerization to support PaaS.*

Sito ufficiale Amazon Web Service. (s.d.). Tratto da https://aws.amazon.com/it/docker/

Sito ufficiale di 01net.it. (s.d.). Tratto da https://www.01net.it/

Sito ufficiale di Docker. (s.d.). Tratto da https://www.docker.com/

Sito Ufficiale di RedHat. (s.d.). Tratto da https://www.redhat.com/it/topics/containers

Tarsitano, P. (s.d.). *Tecnologia container, scoperta vulnerabilità in RunC: ecco i rischi per il cloud.* Tratto da https://www.cybersecurity360.it/news/falla-nella-tecnologia-container-sistemi-cloud-a-rischio-attacco-che-ce-da-sapere/