Tirocinio curriculare svolto presso il DISI - PSLab

Lorenzo Valentini

05/28/2017

Sommario

Il tema da me scelto per il tirocinio corrisponde alla proposta B punto 2.

A Revisione dell'interfaccia grafica 2D/Swing di Alchemist con focus sull'usabilità. Questo integra una serie di passi semi-indipendenti, di cui più se ne fanno meglio è:

1- stabilizzazione e rilascio del plugin IntelliJ-Alchemist (uscita dallo stadio prototipale con una tesi in completamento)

2- sistema per la modifica dello stato dei nodi (attualmente molto caotico, è praticamente una proof of concept)

3- revisione del sistema di creazione e modifica degli effetti

B Sperimentazione (studio, installazione e prove) dei seguenti sistemi cloud-oriented (uno su tre, ma anche di più):

1- Confronto fra Paas Open Source (e.g. Cloud Foundry, AppScale, Tsuru, Apache Stratos ...): sviluppo di un'applicazione minimale e deploy su due o più PaaS per confrontarne features, pro e contro di ognuno.

2- Docker e Container Scheduling: sviluppo di un'applicazione distribuita minimale che sfrutti la containerizzazione e Docker, deploy utilizzando due o più framework di scheduling (e.g Docker Swarm, Kubernetes, Apache Mesos ...) per effettuarne un confronto.

3- Real-Time Data Processing: installazione e prove di strumenti per il processing di dati near real-time , e.g Apache Storm, Kafka, Spark ...,

Lo scopo di questo tirocinio è stato incentrato principalmente sull'apprendimento di alcune tecnologie open-source riguardanti la virtualizzazione di servizi e sistemi. Ciò è stato fatto utilizzando strumenti per automatizzare il deploy su varie architettetture di natura disomogenea, riducendo in questo modo problemi e limitazioni dettate da fattori come: compatibilità, ergonomia di utilizzo e campo applicativo.

I test effettuati con questi strumenti sono stati eseguiti su un singolo calcolatore in cui veniva virtualizzato ogni elemento di un'architettura distribuita.

Obittivo principale nella fase iniziale è stato quello di acquisire familiarità con gli stumenti e le tecnologie del caso di studio, in modo da individuare, tramite una successiva fase di analisi le peculiarità di ognuno di essi. Dopo di che è stato possibile provarne il funzionamento mediante esempi ed esercitazioni pratiche reperiti dalle documentazioni ufficiali, sul web o prodotte da me.

In termini generali si è cercato di valutarne alcune caratteristiche quali: praticità di utilizzo, difficoltà di installazione, limitazioni e potenzialità che li contraddistinguono uno dall'altro.

Introduzione

Vi fornirò una breve introduzione su che cosa verrà trattato sucessivamente più in dettaglio all'interno di questo report, in modo da darvi subito un chiara idea su alcuni concetti che verrano trattati nel corso della documentazione.
Inizierò con il descrivere qual è il paradigma architetturale alla base del concetto dei containers e quali siano le differenze tra questi e una comune macchina virtuale. Poi andremo più nel dettaglio applicando ciò appreso fino ad ora per capire come "Docker-whale" sfrutta al suo interno questo paradigma architetturale.
Dopo aver familiarizzato con i container ho potuto apprezzare i vantaggi di questa tecnlogia anche in un ambiente distribuito in cui vengono messi a disposizione server, capacità di rete e sistemi di memoria, per esempio secondo il modello IaaS(infrastructur as a service). Questo non è altro che un insieme di servizi incapsulati in container distribuiti su un cluster di macchine che cooperano grazie ad un'architettura master-slave, mettendo in esecuzione una o più istanze di essi al proprio interno secondo delle politiche di scheduling per la gestione del carico di lavoro.

Indice

Capitolo 1

Container

La continua espansione dei sistemi cloud e la neccessità di poter scalare servizi e applicazioni in modi sempre più efficienti ed in tempi più ridotti, ha fatto si che venisse rivisto il concetto di macchine virtuali che venivano usate per svolgere questo lavoro.
Perché virtualizzare un'intera macchina, quando sarebbe possibile isolare solamente una piccola parte di essa?
Questa idea ha condotto gli sviluppatori a trovare delle strade alternative alla virtualizzazione completa e lo stesso Google, confrontandosi con questo problema, ha sviluppato un'aggiunta al kernel Linux chiamata **cgroups**, "gruppi di controllo" (control groups) che permette di gestire l'accesso alle risorse. I control groups permettono di impostare le quote di utilizzo, le priorità e misurare le risorse quali: la memoria, l'utilizzo della CPU, gli accessi al disco e così via. Questo modulo ha permesso al team di sviluppo californiano di rilasciare software in maniera più veloce, ed economica e con una scalabilità senza precedenti, permettendogli così di creare un contesto di esecuzione isolato, con un alto livello di astrazione, tanto da imporsi come una sorta di sistema operativo semplificato e virtualizzato sul quale sono basate tutte le applicazioni da loro sviluppate.

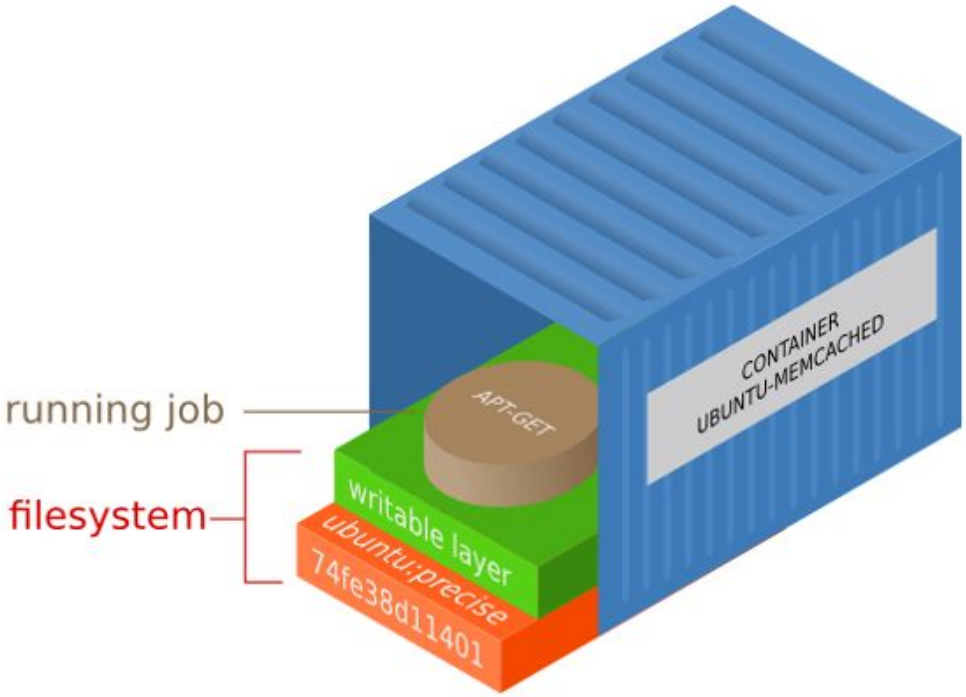

Figura 1.1: Interpretazione grafica di un container partendo da un layer di base che comprende un'immagine di filesystem ubuntu e uno strato che mostra il livello con i layer personalizzati ed infine l'applicativo o il servizio che si vuole utilizzare in esecuzione nel container. Il container non è altro che un contenitore di immagino e programmi.

Storicamente, i container sono stati sviluppati nell'ambito del progetto Linux Containers **LXC**. Per quanto rappresenti una soluzione valida, **LXC** non è mai diventata popolare, principalmente per ragioni di usabilità e per la limitata portabilità dei container creati tramite questa tecnologia anche se in alcuni casi specifici preferibili.
Dopo pochi anni dai primi sviluppi di questo nuovo paradigma architetturale, la volontà di rendere questa tecnologia uno standard, ha condotto il team di Docker a sviluppare un formato di containerizzazione interoperabile, capace di pacchettizzare le applicazioni ed effettuarne il deploy in qualsiasi ambiente di esecuzione, senza doversi preoccupare delle condizioni di eseguibilità.

Vedi figura: 1.1.
Sucessivamente userò in modo del tutto analogo i termini host e server in caso contrario andrò a difefferenziare i concetti nel dettaglio. In pratica, la containerizzazione permette di eseguire un qualsiasi **processo linux** in **ambiente isolato**.
Quindi avremo che un qualsiasi processo "containerizzato" in esecuzione, avrà un proprio ***file system*** privato completamente indipendente da qualsiasi altro processo linux in esecuzione sullo stesso server. Questo tipo di isolamento potrebbe risultare una limitazione ma, in realtà, è proprio grazie a questa caratteristica che si riesce a sfruttare una peculiarità del kernel linux ovvero il **namespaces**, per le quali è possibile sfruttare le seguenti risorse:

- **Inter Process Comunications** (IPC)
- configurazione di rete
- il punto di mount della root
- l'albero dei processi
- gli utenti e i gruppi di essi
- la risoluzione del nome di rete

Senza addentrarsi in eccessivi tecnicismi, il vantaggio principale sta nel fatto che, con l'uso dei namespace, diventa possibile isolare i processi in modo molto efficace. L'unica cosa che il processo isolato nel container condivide con il sistema operativo ospitante è il kernel Linux.

1.1 Paradigma architetturale

Le differenze sostanziali fra i container e le macchine virtuali sono diverse, entrambe le architetture hanno dei propri pregi e difetti, imputabili all'utilizzo e alle esigenze che essi debbono soddisfare. La creazione di una macchina virtuale è molto diversa da quella di un container poichè lavorano su livelli di astrazione differenti.
Le VM virtualizzano completamente tutto il sistema dallo strato hardware fino al livello applicativo in modo da andare a ricostruire nel proprio ambiente una vera e propria macchina indipendente dall'host su cui essa è ospitata, questa è un'operazione onerosa in termini di tempo e risorse, poichè non banale. Questo fa si che ogni macchina dovrà avere una propria porzione di memoria centrale, risorse computazionali e un quantitativo di memoria di massa tutto riservato per le proprie esigenze. Importante portare attenzione al fatto che tutto ciò se per qualche motivo non viene utilizzato rimane allocato per tutto il tempo in cui il sistema virtualizzato rimane in esecuzione privando in questo modo le altre eventuali macchine o applicazioni e servizi in esecuzione sul medesimo server o cluster.
Grazie alla tecnologia dei container che non necessita di dover inglobare tutte le risorse di un server, in particolare il kernel del sistema operativo, i container sono molto più "leggeri" delle macchine virtuali, richiedono poche risorse di CPU e possono essere attivati in pochi istanti. Questo li rende particolarmente adatti alle situazioni in cui il carico di elaborazione da sostenere è fortemente variabile nel tempo con picchi di lavoro. Alcuni scenari applicativi sono la gestione del traffico per siti web di e-commerce o social-media ma anche in molti altri casi dove numeri elevati di utenti accedono simultanea e si connettono richiedendo l'utilizzo di alcune risorse o servizi, facendo si che i sistemi siano sottoposti a picchi di connessioni in entrata, che se gestiti in malo modo in certe situazioni potrebbero compromettere la stabilità del sistema e certe volte rappresentare il fallimento di un progetto o semplicemente il malcontento delle utenze. In questi casi grazie alla tecnologia dei container si possono gestire una moltitudine di richieste attivando

in qualche frazioni di secondo un numero elevato di macchine anche decine o centinaia di istanze, ciascuna delle quali esegue il proprio compito gestendo una porzione del carico di lavoro. Un esempio lampante lo si ha nel caso di un web server sul quale esegue dei web services se ad un certo punto viene contattato da un numero elevato di visitatori in un breve periodo. Esempio "offerte last-minute su un e-commerce" .

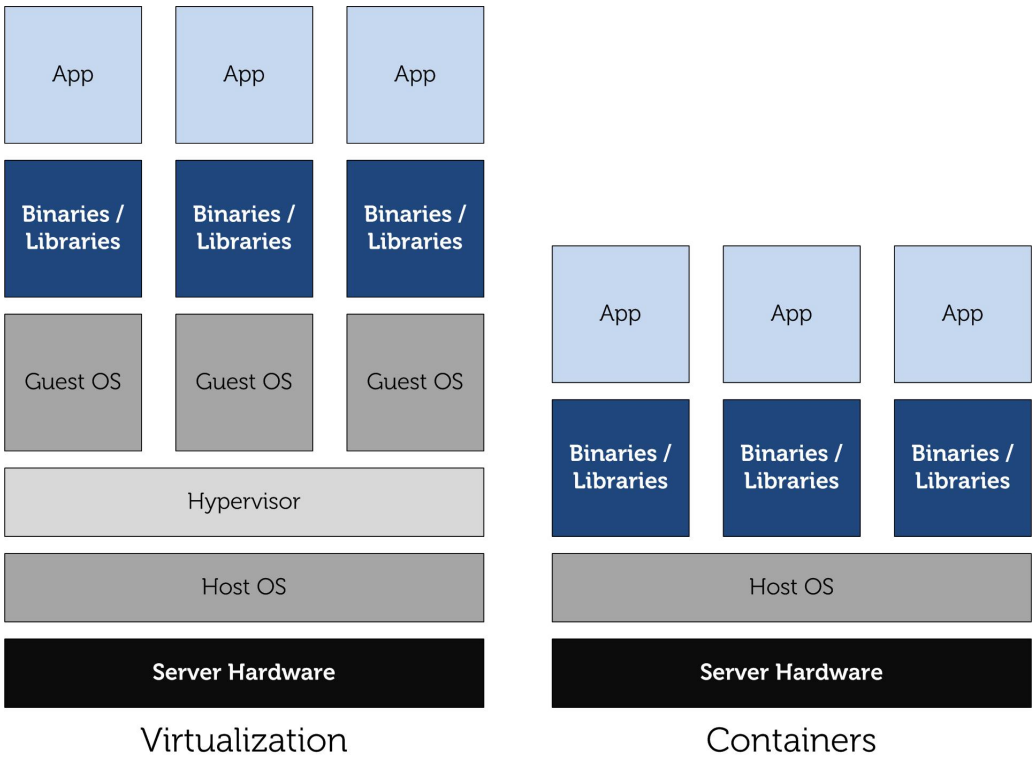

Figura 1.2: Container e VM a confronto con le rispettive architetture.

Sopra detti alcuni dei vantaggi offerti da questa nuova tecnologia per ora in stato embrionale ma che nel breve periodo si prevede sostituirà le VM in molti scenari applicativi grazie a queste sue caratteristiche.

I vantaggi della containerizzazione sono:

Poter contenere un numero elevato di container in esecuzione su una singola macchina.
Deployment semplificato: impacchettando un'applicazione in un singolo componente distribuibile e configurabile con una sola linea di comando, la tecnologia a container permette di semplificare il deployment di qualsiasi applicazione, senza doversi preoccupare della configurazione dell'ambiente di runtime;
Una disponibilità rapida: virtualizzando ed astraendo solo il sistema operativo e le componenti necessarie all'esecuzione dell'applicazione, invece che l'intera macchina, l'intero package si avvia in un tempo molto ridotto, rispetto ai tempi di avvio di una VM;
Un controllo più granulare: i container consentono agli operatori e agli sviluppatori di suddividere ulteriormente le risorse computazionali in microservizi, garantendo così un controllo superiore sull'eseguibilità delle applicazioni e un miglioramento delle prestazioni dell'intero sistema. Per quanto sia possibile eseguire anche su un laptop diverse virtual machine, questa operazione non è mai veloce e semplice ed impatta sulle prestazioni.
Inoltre si traggono vantaggi anche sull'amministrazione dei cicli di rilascio delle applicazioni, in quanto distribuire una nuova versione di un container è pari al tempo speso per digitare in console una singola linea di comando.
Le attività di testing traggono un beneficio economico da un ambiente containerizzato. Se si effettua il test di un'app direttamente su un cloud server in un ambiente di cloud computing pubblico, sarà necessario sostenere i costi relativi ai tempi di occupazione delle risorse computazionali. Questo costo aumenta all'aumentare del numero di test che devono essere eseguiti. Con un container è possibile effettuare una serie di semplici test programmati mantenendo costante il costo, in quanto si userebbero sempre le stesse risorse di calcolo. Infine, non si possono ignorare i guadagni in termini di componibilità dei sistemi applicativi, specialmente per le applicazioni open source. In pratica, invece di obbligare gli sviluppatori a installare e configurare i più disparati servizi che potrebbero richiedere molto tempo in termini di messa

in installazione come per esempio MySQL, memcached, MongoDB, nginx, node.js e via discorrendo, per avere la giusta piattaforma esecutiva per le proprie applicazioni, sarebbe meno rischioso e più veloce avviare ed eseguire con piccoli script quei pochi container che ospitano queste stesse applicazioni. Sul lungo periodo, il più importante vantaggio che questa tecnologia promette è la portabilità e la consistenza di un formato che consente l'esecuzione applicativa su diversi host. Infatti, con la standardizzazione dei container, i workload possono essere facilmente spostati lì dove vengono eseguiti in modo più economico e veloce, evitando anche i lock-in (blocchi e rallentamenti) dovuti alle peculiarità delle piattaforme dei singoli provider.
Breve panoramica per avere un quadro completo sul fatto che esistono anche container in ambiente Windows, si distinguono in due diversi tipi:

Windows Server Container: forniscono uno strato di isolamento tra gli applicativi usando tecnologie basate sui namespace e la separazione dei processi, ma condividono il kernel tra tutti i container in esecuzione sullo stesso host.

Hyper-V Container: eseguono ogni container in una macchina virtuale ottimizzata, in cui viene eseguita una diversa istanza del kernel non condivisa con gli altri container Hyper-V.
Il tipo di container utilizzato ha implicazioni anche sul costo della licenza Windows Server.

1.1.1 Container vs. Virtual Machine

In questo caso introduciamo come esempio facendo riferimento al modello architetturale di Docker.

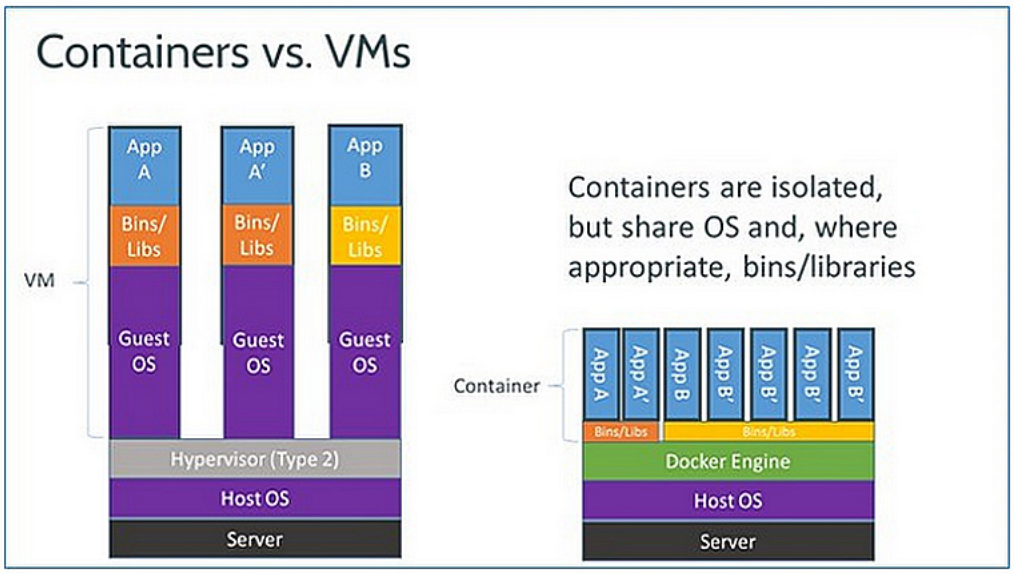

Figura 1.3: Container vs. Virtual Machine..

Come è possibile vedere nella figura 1.3, i vantaggi principali dei container si riassumono in un minore consumo di memoria e spazio disco. I container usano meno memoria poiché tutti i container in un ambiente ospitante condividono lo stesso kernel. I container usano meno spazio su disco poiché le immagini dei container possono essere condivise da container che sono in esecuzione sullo stesso host.
Dal punto di vista della sicurezza, il consenso generale sembra essere a favore delle macchine virtuali. Infatti le virtual machine appaiono più sicure e più isolate rispetto ai container, ma visto il rapido sviluppo si presume che anche questo cambierà con la progressiva maturazione della tecnologia container.

1.2 Docker

Docker è un progetto open source con una azienda di supporto che fornisce assistenza e altri servizi a pagamento per la versione docker-EE. La forza di Docker, e una delle ragioni della sua diffusione, è la sua facilità d'uso. Docker

è un insieme di tecnologie che cooperano tra loro al fine di standardizzare il formato dei container e avere un insieme di tool standard e funzionanti che permettono di avere la piena portabilità e poter eseguire i container su diverse piattaforme e host. Le componenti principali di Docker sono:

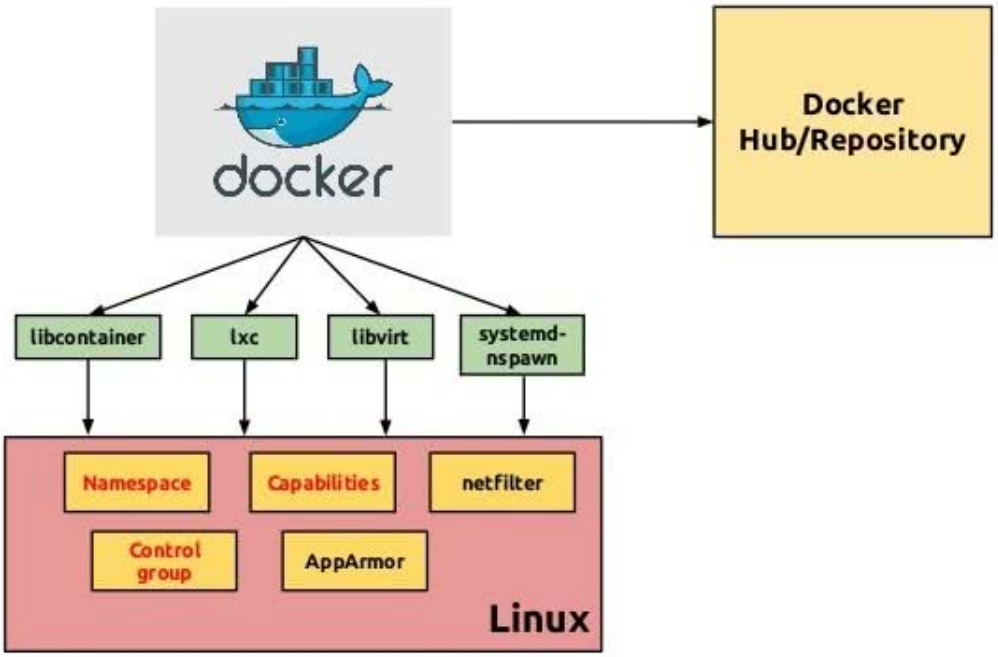

Figura 1.4: I registry di Docker.

- il Docker engine, che è un demone che risiede sul server e accetta comandi da un client (sia da riga di comando sia da chiamate alle API). Esso comunica con il Kernel Linux attraverso la libreria libcontainer (che viene installata insieme a docker).

- il registry, che è una piattaforma che risiede nel cloud e che fornisce un'area dove caricare, scaricare e condividere le immagini dei vari container. Quella ufficiale è https://hub.docker.com. Supponiamo di voler utilizzare un container con una determinata immagine. Grazie a queste

due componenti il cliente chiede al server di prendere quella immagine e “iniettarla” in un container. Il server, se non possiede ancora l’immagine richiesta, contatta il registry. Se esiste nel cloud la scarica e viene creata una copia nella cache locale. Dopo la inietta in un container. Vedi figura: 1.5 1.6.

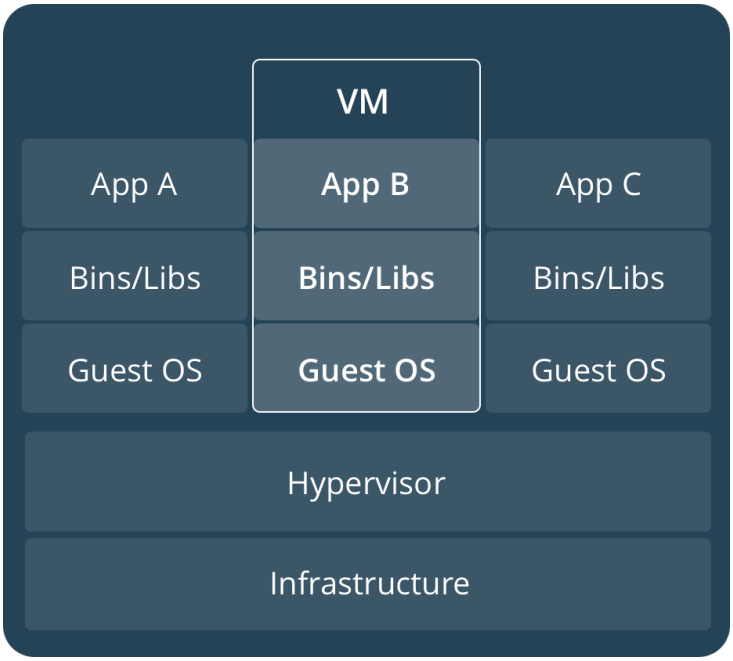

Figura 1.5: Solution stack classico.

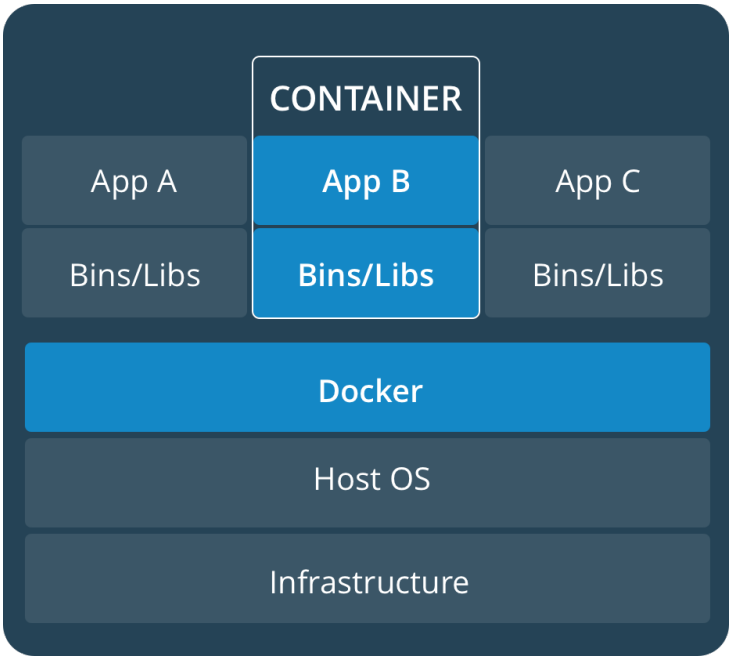

Figura 1.6: Solution stack di Docker.

Docker implementa una architettura client–server. Il client è un semplice strumento a linea di comando che invia comandi al server utilizzando servizi REST: questo implica che chiunque potrebbe costruire client differenti. Il server è un daemon Linux in grado di costruire immagini ed eseguire container sulla base di una immagine preesistente.
Vedi figura: 1.4 1.7

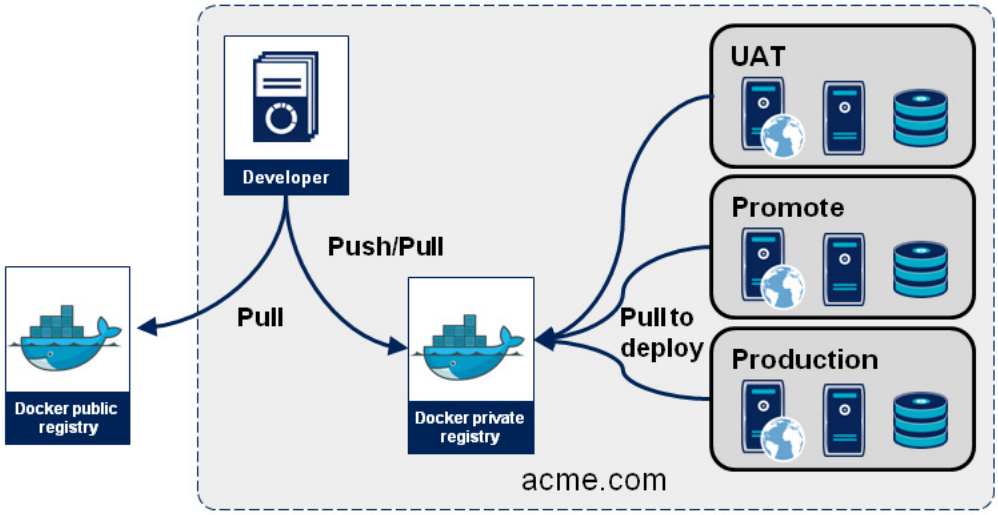

Figura 1.7: I registry di Docker.

Cenni sulla struttura del FS

Union File System: Una caratteristica davvero interessante di Docker è il modo in cui viene gestito il file system. Docker utilizza un'implementazione di **Union File System** per comporre il file system che apparirà visibile al container. Gli Union File System (ad esempio UnionFS o **aufs**) consentono di unire diversi file system e montarli in un unico punto di mount.
Un'immagine Docker è in realtà un insieme di file organizzati in directory: in definitiva, è un file system. Un'immagine docker può essere costruita sulla base di un'immagine già esistente semplicemente aggiungendo e/o modificando alcuni file ad esempio facendo commit di un'immagine già esistente oppure agguingendo strati con il dockerfile. E questo può essere ripetuto per un numero illimitato di volte. Un'immagine può essere ad esempio costituita da n strati (vedi figura: 1.1): questo concetto è conosciuto anche come "immagini stratificate" (**layered**). Nell'esempio precedente, quando l'immagine Docker va in esecuzione, gli n file system vengono uniti insieme (merge) in modalità sola lettura e ad essi viene sovrapposto un ulteriore strato in

modalità lettura–scrittura writable layer (vedi fig: 1.1).

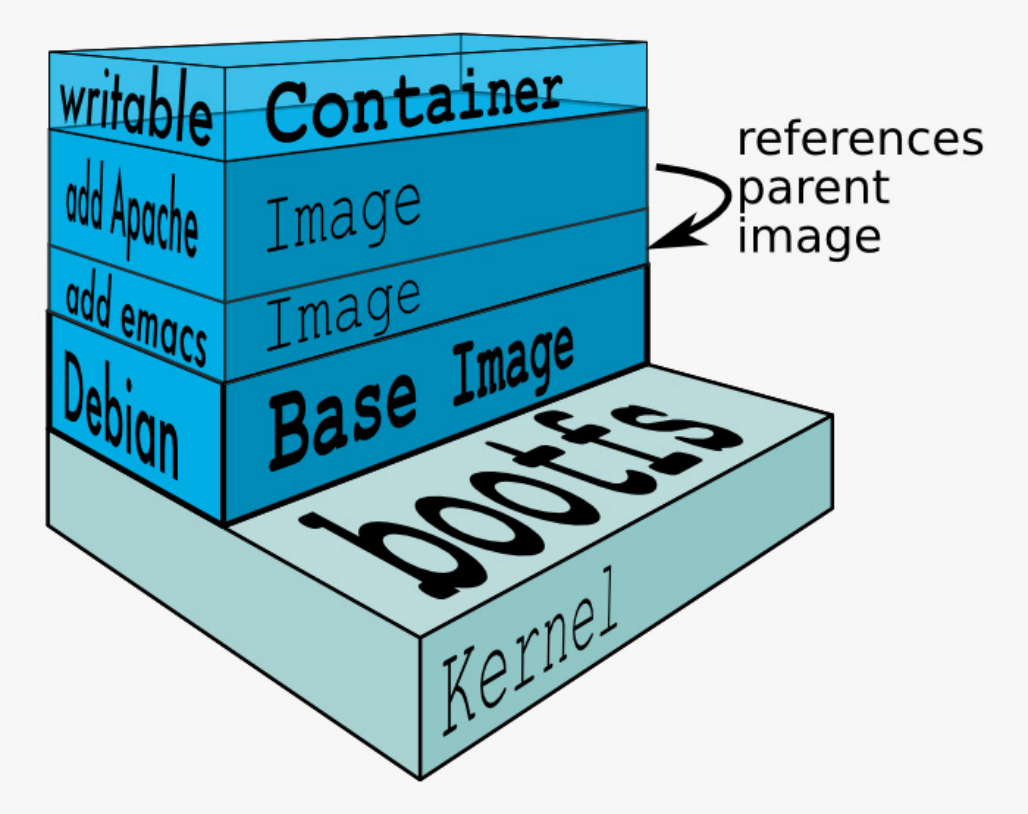

Figura 1.8: Un file system stratificato (layered file system).

Se due container condividono parte degli strati dell'immagine, questi layer non dovranno essere replicati nel file system host, poiché sono montati in modalità sola lettura. Questo può generare significativi risparmi specie se un'azienda si dà una certa disclipina riguardo al modo in cui possono essere create le immagini: potrebbe essere il caso, ad esempio, di stabilire che tutte le immagini debbano derivare da una base comune.
In questo caso, siccome la base comune sarà probabilmente lo strato che prende la maggior parte dello spazio disco e gli altri layer dovranno aggiungere installazioni o configurazioni minori, tale semplice linea guida potrebbe far risparmiare grandi quantità di spazio disco. Come in figura (1.8) l'immagi-

ne di base condivisa fra vari container potrebbe essere quella del layer che contiene l'immagine Debian.

1.3 Istallare Docker

Per installare Docker su qualsiasi altro sistema *unix o Windos, seguire la guida ufficiale di Docker reperibile al link https://docs.docker.com/engine/installation/. In questa sezione vi mostrerò la procedura per l'installazione dello strumento Docker CE (community edition) su una macchina Ubuntu e come verificare se la procedura sia andata a buon fine. Allo scopo di non riscontrare problemi futuri è necessario verificare che la distribuzione Docker appena installata sia una versione maggiore o uguale 1.13.
Poichè nelle precedenti versioni non sono embedded in docker alcuni tool di automazione e la piattaforma di container scheduling Swarm di cui parleremo più tardi.

Prerequisiti minimi

Per installare Docker CE è necessario avere installato una delle seguenti distribuzioni ubuntu a 64bit in caso contrario guarda il link della documentazione ufficiale:

- Yakkety 16.10
- Xenial 16.04
- Trusty 14.04

1. Step

```
$ which curl

#  If curl isn't installed, install it after updating your manager:
```

```
   $ sudo apt-get update

   $ sudo apt-get install curl

#  Get the latest Docker package.

   $ curl -fsSL https://get.docker.com/ | sh

#  The system prompts you for your sudo password. Then, it downloads and installs Do

# Note: If your company is behind a filtering proxy, you may find that the apt-key c
# during installation. To work around this, add the key directly using the following

   $ curl -fsSL https://get.docker.com/gpg | sudo apt-key add -
```

IN ADDITION
In aggiunta se si vuole si può considerare l'opzione di aggiungere il programma docker alla lista dei programmi con privilegi elevati in modo da poterlo eseguire in modalità non root o senza il comando `sudo` ogni volta:

```
   $ sudo usermod -aG docker <name_of_the_user_to_add>
```

Per esempio se si vuole aggiungere l'utente corrente il comando andrà così formattato:

```
   $  sudo usermod -aG docker ‘whoami‘
```

Ricordati che le modifiche diventeranno effettive solo dopo il logout!

PACKAGE EXTRA RACCOMANDATI
L'installazione di questi package extra è necessaria se si vuole montare **??**aufs sulla nostra macchina linux per visualizzare Docker sul file system.

```
$ sudo apt-get update

$ sudo apt-get install curl \
    linux-image-extra-$(uname -r) \
    linux-image-extra-virtual
```

DOPO L'INSTALLAZIONE:
procedo con la verifica dell'installazione.
procedere con esercizio di base.
Verifica che il servizio docker sia in esecuzione sulla tua maccchina ad ogni modo per sicurezza eseguire la seguente sequenza di comandi per verificare lo stato del servizio e riavviarlo:

```
$ sudo service docker status

#next command restart all docker service

$ sudo service docker restart
```

In caso arrivati a questo punto per qualche motivo l'istallazione non sia andata a buon fine si potrebbe provare con questa sequenza di comandi:
1. Set up del repository
Aggiunta di Docker CE repository in Ubuntu. Tramite il seguente comando `lsb_release -cs` stampo il nome della mia versione corrente di Ubuntu, esempio "xenial or trusty".

```
sudo apt-get -y install \
apt-transport-https \
ca-certificates \
curl

curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key add -

sudo add-apt-repository \
"deb [arch=amd64] https://download.docker.com/linux/ubuntu \
$(lsb_release -cs) \
stable"

sudo apt-get update
\begin{verbatim}
2. Get Docker CE

Install the latest version of Docker CE on Ubuntu:


sudo apt-get -y install docker-ce
```

*Tratto direttamente dalla documentazione originale (causa recenti aggiornamenti della piattaforma è possibile che alcuni comandi per alcune procedure siano variati nel frattempo invito sempre a verificare sulla documentazione).

Verifica dell'installazione:

Ora dovremmo essere in grado di lanciare il comando `docker run hello-world` e dovremmo ottenere un output tipo questo:

```
$ docker run hello-world

Hello from Docker!
This message shows that your installation appears to be working correctly.

To generate this message, Docker took the following steps:
...(snipped)...
```

Iinfine per motivi di compatibilità e per evitare problemi con l'esecuzione di Swarm in Docker sucessivamente, accertiamoci che la versione sia uguale o superiore alla 13.0 eseguendo il comando `docker -version` e se vogliamo maggiori informazioni possiamo digitare il comando `docker info`.

```
$ docker --version
Docker version 17.05.0-ce-rc1, build 2878a85
```

Arrivati a questo punto si è pronti per iniziare a sperimentare passando alla sezione dove creeremo di persona le nostre immagini.

1.4 Creare un container con Docker

Inizialmente controlliamo di aver installato docker e che sia la versione giusta come descritto del capitolo precedente.
1. Step:
Utilizziamo ora alcuni comandi già usati in precedenza:

```
1 $ docker run hello-world
2
3 $ docker --version
Docker version 17.05.0-ce-rc1, build 2878a85
```

Ora analizziamo brevemente cosa è successo quando abbiamo lanciato sulla nostra macchina per la prima volta un container docker. Se non ci siamo accorti di averlo fatto è tutto normale, anche perchè non si direbbe che in pochi secondi dal lancio del comando fino allo scaricaricamento dell'immagine dal docker registry alla sua esecuzione sia servito così poco tempo e sia stato così facile. Il comando che abbiamo lanciato è stato quello alla riga 2 (`docker run hello-world`). Questo comando una volta lanciato dice a Docker di eseguire un l'immagine di un container taggata con il nome di "hello-world", questo ha fatto si che il sistema in modo automatico sia andato nel repo locale delle immagini e abbia cercato la presenza di un'immagine con un tag di riferimento "hello-world", ma dato che era la prima volta che la usavamo il sistema non l'ha trovata quindi si è connesso al docker-hub e ha cercato l'immagine da noi richiesta. Una volta trovata la scarica in locale sul nostro host e la eseguita restituendonci a video la banale scritta ciao mondo e qualcos'altro.
Esempio di output:

```
test-VirtualBox test \# docker run hello-world
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
78445dd45222: Already exists
Digest: sha256:c5515758d4c5e1e838e9cd307f6c6a0d620b5e07e6f927b07d05f6d12a1ac8d7
Status: Downloaded newer image for hello-world:latest

Hello from Docker!
This message shows that your installation appears to be working correctly.

To generate this message, Docker took the following steps:
1. The Docker client contacted the Docker daemon.
2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
```

```
3. The Docker daemon created a new container from that image which runs the
executable that produces the output you are currently reading.
4. The Docker daemon streamed that output to the Docker client, which sent it
to your terminal.

To try something more ambitious, you can run an Ubuntu container with:
\$ docker run -it ubuntu bash

Share images, automate workflows, and more with a free Docker ID:
https://cloud.docker.com/

For more examples and ideas, visit:
https://docs.docker.com/engine/userguide/
```

Tramite il comando (`docker images`) possiamo visualizzare le immagini salvate sulla memoria locale del nostro host:

```
test-VirtualBox test \#
docker images
```

REPOSITORY	TAG	IMAGE ID	CREATED
<none>	<none>	8ff43252d7f2	7 weeks ago
<none>	<none>	b7a9f1663f67	7 weeks ago
<none>	<none>	7ee09435238b	7 weeks ago
<none>	<none>	8126e3300de2	7 weeks ago
ubuntu	16.04	0ef2e08ed3fa	10 weeks ago
hello-world	latest	48b5124b2768	3 months ago

Tramite il comando (`docker rmi <REPO:TAG | IMAGE ID> ...`) possiamo eleiminare le immagini salvate sulla memoria locale del nostro calcolatore.

```
test-VirtualBox test \#
docker rmi 8ff43252d7f2 b7a9f1663f67 7ee09435238b 8126e3300de2

Deleted: sha256:8ff43252d7f2b4f84785f139e6135df037e731505fd70fc64d00248cbca15bad
Deleted: sha256:b656184f7e21b4dfc6de17aa8df42a0fe01a37df82acd0bbd491dbc529f860b1
.
.
.
Deleted: sha256:7ee09435238ba560fe614d699c33a3532e3ada996ae6c1d9b2d42869525ae3b6
Deleted: sha256:818754db35602595492ef843a644ec16b985b0ab18582b543edc1b4129e0cde8
Deleted: sha256:f91bca33aeae5b04e4235f84459dc933b02268862077ff51d2b2ba4f392be940
==>Error response from daemon: conflict: unable to delete 8126e3300de2 (cannot be fo

test-VirtualBox test \# docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
<none>              <none>              8126e3300de2        7 weeks ago         70.8
ubuntu              16.04               0ef2e08ed3fa        10 weeks ago        130
hello-world         latest              48b5124b2768        3 months ago        1.84
```

Nel caso la rimozione dell'immagine non vada a buon fine come nel caso (8126e3300de2) basta aggiunger il flag (`docker rmi -force <REPO:TAG | IMAGE ID> ...`) Ecco un breve riassunto dei principali comandi più usati nella gestione di un container.
Una volta installato bisogna far partire il servizio del docker. Su Ubuntu il comando è il seguente:
`service docker start`

Vediamo adesso una serie di comandi (per semplicità solo su Ubuntu, per le altre distribuzioni o sistemi operativi basta consultare la documentazione su https://docs.docker.com/) da utilizzare.

- Per verificare se docker è stato installato correttamente possiamo lanciare il classico "Hello World": `docker run hello-world`

- Per stoppare il servizio di docker:
  `service docker stop`

- Per cercare un'immagine in Docker Hub:
  `docker search <nome_immagine>`

- Per scaricare un'immagine da Docker Hub:
  `docker pull <nome_immagine>`

- Per scaricare un'immagine da un repository di Docker Hub:
  `docker pull <nome_repositorynome_immagine>`

  Nei due comandi precedenti si vuole scaricare un'immagine ma non abbiamo indicato il tag, così viene presa l'ultima versione presente dell'immagine. L'uso dei tag viene altamente consigliato perché sono utili soprattutto per l'isolamento e il packaging.

- Il comando con indicazione anche del tag diventa per esempio:
  `docker pull <nome_repositorynome_immagine:tag>`
  `docker pull ventus85/tomcat:8.0.32`

- Per visualizzare l'elenco delle immagini presenti sulla macchina host:
  `docker images`

- Per avviare un nuovo container interattivo che usa la relativa immagine e avvia un terminale all'interno del nuovo container:
  `docker run parametri -name <nome_container> <nome_immagine:tag>`

- Per esempio se vogliamo avviare il container per tomcat e vogliamo che utilizzi una determinata porta è necessario fare un forwarding. Per fare la mappatura delle porte si usa il parametro "p" mentre per mappare una cartella il parametro è "v".
  `docker run -it -d -name tomcat -p 8080:8080 -v homemyApptomcat:usrlocaltomcatlo`
  `ventus85tomcat:8.0.32`

- Quando viene avviato un container esso eseguirà un eventuale entry point (ovvero un file chiamato dockerentrypoint.sh). Per avere l'elenco dei container presenti (se aggiungiamo l'opzione -a mostra solo quelli attivi):
  `docker ps -a`
  L'elenco dei contanier contiene:

  1. l'id del container
  2. il nome del container
  3. il nome dell'immagine
  4. il comando per avviarlo
  5. quando è stato creato
  6. lo stato (per esempio se è "up")
  7. le eventuali porte mappate

- Se invece vogliamo eseguire un comando su un container che vogliamo eseguire:
  `docker exec [OPTIONS] <nome_container> comando [ARG...]`

- Se per esempio vogliamo aprire una shell per poter usare il container con Oracle:
  `docker exec -it oracle bash`

- Per rimuovere un container:
  `docker rm <id_container>`

- Per rimuovere l'immagine:
  `docker rmi nome_immagine`

- Per stampare un file di log del container:
  `docker logs <nome_container>`

- Per avere le informazioni del container (vengono restituite in formato json):
  `docker inspect <nome_container>`

- Per fare un commit (ovviamente in locale) sulle eventuali modifiche effettuate sul container:
  `docker commit nome_container nome_immagine`

  Questa operazione è utile nel caso si voglia creare una nuova immagine con, per esempio, delle impostazioni diverse rispetto alla precedente.

1.4.1 Che cos'è il Dockerfile

Oltre a usare il comando `commit` per personalizzare i nostri container è possibile eseguire questa operazione anche attraverso l'ausilio di un altro documento. Esso è un file chiamato "Dockerfile" senza alcune estensione e deve contenere i comandi cesseri a costruire una nuova immagine Docker. La sintassi del Dockerfile è scritta in Ruby più i classici comandi bash Vedi esempio sottostante. Riferimento alla guida ufficiale al link Dockerfile reference.

```
FROM node:alpine

# Set the environment variables
ENV PORT=9111


COPY ./node-ex2 ./build

RUN \
        ls -alh ./build/views &&\
        npm install ./build

#EXPOSE ${PORT}
EXPOSE 9111

RUN echo -e "Expose port ${PORT}\n"

WORKDIR ./build

CMD ["node", "./app.js"]
```

Una volta creato il nostro Dockerfile (oppure dopo averne modificato uno già esistente), il passo successivo sarà quello di creare il container tramite il comando **docker build** nel seguente modo:

```
docker build -t marco94/service:fib_1.0
```

* Il nome del repository matteo94 è stato scelto a caso era tra i nick-name disponibili al momento della creazione.
Link del repo sul Docker Hub.

Come scrivere in semplicità un Dockerfile

Le cose da tener presente quando si scrive un Dockerfile non sono molte, capite queste si è pronti per scrivere il proprio file personalizzato. La sintassi di base è pressoche identica a quella della bash. I commenti si fanno con il carattere

```
# Questa è una riga di commento
```

Il continuo di un istruzione si fa con il back slash

```
RUN \
echo " come è andata oggi a scuola:"\
$RISPOSTA;
```

La prima riga, che non sia un commento all'interno del file dovrà necessariamente essere l'instruzione **FROM**.

Esempio:

```
FROM scratch
ADD ubuntu-xenial-core-cloudimg-amd64-root.tar.gz /
.
.
.
# overwrite this with 'CMD []' in a dependent Dockerfile
CMD ["/bin/bash"]
```

Mentre per tutte le altre righe non esistono altri limiti particolari, tranne che, i comandi di compilazione non vengono eseguiti in questa fase di build. Questo per quanto riguarda quello che ho riscontrato io facendo delle prove. Provare per credere Dockerfile esercizio sulla successione di Fibonacci.

Capitolo 2

Container orchestration

Il progetto Docker è nato con l'obiettivo di semplificare la creazione e la gestione di container. Docker svolge questi compiti in modo molto efficace e va riconosciuto al team che l'ha creato la capacità di mantenere l'ambito di questo progetto chiaro e limitato. Docker è un mattone per costruire dei sistemi cloud basati su container.

Costruire un ambiente cloud basato su container implica di tenere presenti alcune aree principali:

- container scheduling
- networking avanzato
- sicurezza
- monitoraggio
- quote e misurazioni

Vediamo di seguito questi diversi aspetti. **Container scheduling**:
Il container scheduling (più o meno "calendarizzazione dei container") è la possibilità di definire e attivare un ambiente potenzialmente composto di diversi container. Come già detto, è un concetto simile a quello dell'orchestrazione negli ambienti cloud basati su Virtual Machine.

Attraverso il container scheduling si dovrebbe riuscire a definire la corretta sequenza di attivazione dei container, la dimensione del cluster di container che è necessario creare e la qualità del servizio dei container: per esempio, un'app potrebbe richiedere un minimo garantito di memoria e di accessi al disco.
Tool avanzati per l'orchestrazione devono garantire anche che un determinato ambiente rifletta lo stato desiderato dopo essere stato creato: significa che questi strumenti di orchestrazione possono anche svolgere un certo grado di monitoraggio ed effettuare, almeno in parte, una autoriparazione nel caso lo stato dell'ambiente non si allinei con quello stabilito dalla configurazione.
Un'altra importante caratteristica comune a molti scheduler per container è la capacità di definire un livello di astrazione intorno a un cluster di container identici, in maniera tale che essi siano visti come servizio, vale a dire un indirizzo IP e una porta. Questo risultato è ottenuto facendo impostare allo scheduler un load balancer davanti al cluster di container accompagnato dalle regole di indirizzamento necessarie.
In questo ambito, alcuni degli strumenti disponibili sono Swarm e e Docker Compose, Apache Mesos, Kubernetes. Li vediamo di seguito.

2.1 Swarm embedded in Docker

Swarm è la proposta sviluppata dalla startup Docker che si adatta facilmente ad ambienti in cui già si lavora con container Docker. La soluzione è ideale per operazioni di testing o per il deploy di applicazioni su ristretta scala . "Swarm" è caratterizzato da una serie di componenti incaricati di svolgere specifici compiti, nel dettaglio:

- **Manager.** Si occupano di distribuire le task tra i vari cluster. Un manager si occupa di orchestrare i nodi worker che costituiscono lo swarm.

- **Worker.** Eseguono i container che sono stati assegnati da un manager. Servizi. Un'interfaccia per un particolare set di container docker in esecuzione.
- **Task.** Singoli container che eseguono immagini e comandi richiesti da un particolare servizio.
- **Key-value store.** Servizi (etcd, Consul, Zookeeper) che si occupano di archiviare lo stato dello swarm e fornire visibilità al servizio.

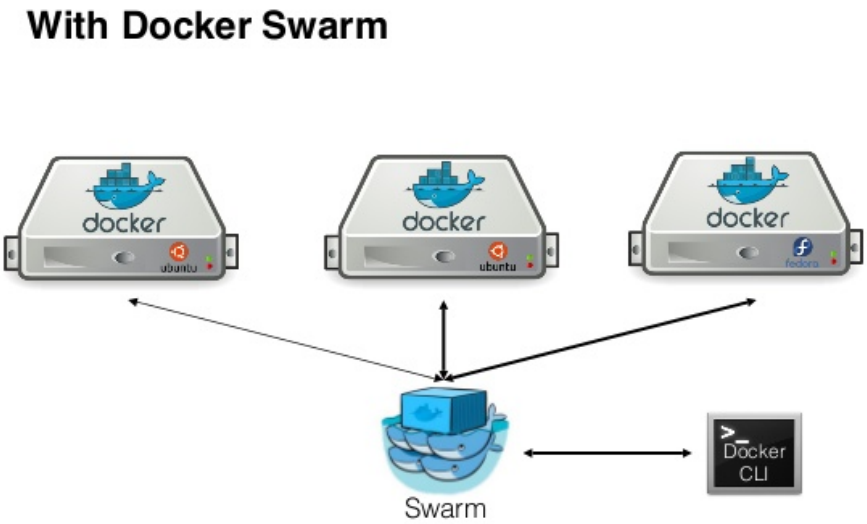

Figura 2.1: docker swarm architecture source

Capitolo 3

Introduzione su Swarm e concetti di base

3.1 Che cos'è Swarm?

Swarm è un cluster manager presente in Docker engine dalla versione 1.12 e successive all'interno dello SwarmKit. Come funziona, i sistemi in cui eseguiamo Docker candidati a partecipare ad un cluster di macchine (ad uno "swarm") potranno farlo solo se in modalità Swarm utilizzando i comandi es. (`docker swarm <command_name>` mediante la tipica sintassi per richiamare tools interni a docker). Una macchina per poter partecipare a uno Swarm("sciame") ha due possibilità inizializzarne uno nuovo oppure unirsi ad uno già esistente.
Swarm quindi non è altro che un insieme di macchine Docker engine "nodi" interconnessi dove si possono distribuire dei servizi. Per effettuare la gestione sono messe a disposizione (Docker engine CLI) e (API) che includono al loro interno comandi per effettuare la gestione dei nodi come ad es.(add, remove, init nodes), e per gestire i servizi in esecuzione su di essi.
Quindi possiamo dire che quando usiamo Docker al di fuori della modalità Swarm, i nostri "contenitori"(container) verranno eseguiti con i comandi normali di Docker.

Quando invece eseguiamo Docker in modalità uno Swarm i "contenitori" di immagini saranno gestiti come servizi. Questo significa che io posso eseguire un servizio Swarm o un container("stanalone Docker container") partendo da una medesima istanza "immagine" di partenza.
Portiamo particolarmente attenzione a cosa si intende per "nodo" di uno Swarm, potrebbe essere il chiarimento che stiamo cercando alle domande inizialmente poste.

3.2 Che cos'è un nodo?

Un **nodo** può essere inteso come una singola macchina su cui è in esecuzione un'istanza di Docker engine ogni nodo può far parte di uno "swarm". Ogni macchina la si può immaginare come un "nodo Docker". Si possono eseguire uno o più nodi sopra un singolo computer fisico o all'interno di cloud server, ma un tipico risultato di una semplice distribuzione con Swarm, potrebbe corrisponde a più nodi Docker distribuiti su diversi computer fisici o su cloud in comunicazione tra loro tramite una rete di interconnessione. Tale rete mette in comunicazione i vari nodi di uno swarm sfruttando il paradigma master-worker.
La distribuzione delle applicazioni con Swarm, la si effettua definendo un task servizio e assegnandolo ad un nodo in particolare quello avente funzionalità di manager. Il nodo manager (**manager node**) a questo punto ha il compito di occuparsi della distribuzione delle unità di lavoro chiamate **tasks** ai nodi lavoratori (**worker nodes**).
I nodi manager hanno anche il compito di occuparsi dell'orchestrazione e della gestione delle funzionalità richieste allo "swarm" per lo svolgimento del proprio compito e il mantenimento dello stato. Tra i vari nodi aventi il ruolo di manager ne esiste solo uno eletto come Leader incaricato di condurre l'orchestrazione dei tasks.
I nodi lavoratori ricevono ed eseguono i tasks(compiti) spediti dal nodo manager. Di default anche il nodo manager oltre ad eseguire le ordinarie ope-

razioni di gestione dello sciame esegue anch'esso i servizi come se fosse un semplice "worker nodes". Successivamente questa configurazione può essere modificata in modo che esegua esclusivamente con funzionalità di manager. Un "agente" che non è altro che un servizio in esecuzione in background nello swarm, che per tutti i nodi lavoratori a cui sono stati assegnati dei tasks produce un report informativo in cui tiene traccia dello stato. I worker node notificano continuamente al nodo manager lo stato dei tasks a loro assegnati così che il manager possa operare per mantenere lo stato desiderato per tutti i nodi del sistema questa operazione è detta **load balancing**. Per maggiori chiarimenti vedere documentazione ufficiale QUI.

3.3 Servizi e tasks

Un **servizio** in esecuzione su un nodo "worker" è definito come **task**. Questi rappresentano l'unità minima di lavoro svolta da ogni nodo del sistema. I **tasks** consentono di soddisfare richieste multiple degli utenti che utilizzano quel determinato servizio. Quando si crea un servizio, la prima cosa che si deve fare è specificare da quale immagine di base si vuole partire a creare il container Docker e quali comandi lanciare al suo interno.
Dopo aver creato un nuovo servizio o avendone uno già precedentemente generato la fase di replicazione e distribuzione è a carico del nodo Manager, che serve i vari nodi Worker con i tasks in modo da mantenere lo stato della distribuzione del servizio come specificato. Nel caso di servizi GLOBALI, lo swarm eseguirà il task di quel servizio su di un qualsiasi nodo disponibile nel cluster. Quindi riassumendo: un task al suo interno contiene un container Docker più i comandi necessari all'esecuzione di quest'ultimo che rappresenta l'unità minima di lavoro atomica e indivisibile. Il manager scala i compiti ai nodi worker in accordo alle politiche di scheduling definite dall'amministratore di sistema. Un task una volta assegnato ad un nodo, non può essere spostato in un altro nodo può solamente eseguire con successo oppure fallire. Per maggiori chiarimenti vedere documentazione ufficiale QUI.

3.4 Load balancing

Il nodo manager, in quel momento candidato alla gestione dello sciame, usa il traffico in ingresso per bilanciare la distribuzione del servizio reso accessibile dall'esterno. Il manager, inoltre, se non specificata può assegnare automaticamente una porta di rete ad ogni servizio aggiunto o in alternativa è possibile specificare una porta sulla quale esporre il servizio, in tal caso è sufficiente in fase di creazione del servizio assegnare mediante l'utilizzo di un flag passato al comando una porta non usata tra quelle disponibili. Nel caso noi lasciassimo la gestione delle porte al nodo manager, sarà lui a provvedere all'assegnamento sciegliendo tra una di queste nel range dalla 30000 alla 32767.
Componenti esterni, per lo scheduling del carico possono anch'essi accedere al servizio tramite la sua interfaccia pubblica resa accessibile all'indirizzo e sulla porta esposta per ogni nodo del cluster, indipendentemente dal fatto che il nodo in quel momento stia eseguendo il servizio o meno. Tutti i nodi dello sciame che permettono connessioni in ingresso eseguono un istanza di un task.
Inoltre uno Swarm ha un componente DNS interno che assegna automaticamente ad ogni servizio nello swarm un proprio nome simbolico interno per distinguere tra loro i vari tasks presenti nello swarm. Il sistema interno di bilanciamento del carico distribuisce le richieste dei vari servizi all'interno del cluster sui vari nodi in base alle richieste utilizzando il nome simbolico DNS del servizio.

Capitolo 4

Esempio con Docker Swarm

La prova svolta aveva lo scopo di utilizzare lo strumento di container scheduling Docker Swarm, per osservarne e studiarne il funzionamento in un tipico scenario applicativo. Ad esempio, in questo caso, ho realizzato un semplice servizio web incapsulato mediante la tecnologia dei container offerta da Docker. Questo è stato successivamente scalato su un cluster di macchine virtuali connesse e gestite tramite lo strumento di orchestrazione Swarm. Sempre mediante più VM sono stati emulati dei Client, necessari per effettuare le richieste GET che emulavano il traffico in entrata al Cloud Server. Il cluster di macchine server anch'esse virtualizzate con delle VM come detto in precedenza invece ospitavano il servizi. I servizi sono gestiti e incapsulati come tasks mediante la tecnologia dei container. Cercherò ora di fare in modo chiaro e conciso un breve riassunto di questa sperimentazione pratica. In questo documento non saranno appositamente affrontati tutti i dettagli dei passaggi e le varie fasi necessarie alla preparazione delle macchine. È importante tenere a mente che per la riuscita della dimostrazione sono elementi strettamente fondamentali. Per semplicità suddividerò l'esperimento in fasi:

1. Inizialmente, è stato necessario incorporare l'immagine di un web-server all'interno di un container Docker in modo da ottenere con una semplice operazione di DockerFile scripting il nostro container che incapsula al suo interno web server e i comandi necessari successivamente per la

sua esecuzione. Il container da me costruito quindi si comporrà al suo interno dei seguenti layer: la base di partenza è un'immagine Linux Alpine minimale. Il sever sul quale girerà il nostro applicativo server-side è node.js sul quale farò girare un'applicazione java script app.js che rappresenterà in definitiva il servizio che renderemo disponibile all'esterno. Per comodità ho deciso di renderlo raggiungibile sulle porte 9111:9111.

2. Successivamente, è stato necessario creare un cluster di VM con Virtual Box per simulare la piattaforma di hosting. Una volta configurate e settate le comunicazioni di rete delle varie macchine, ho provveduto all'installazione di Docker » v.1.12 su ognuna di queste. Le VM da me create per l'occasione erano 4 macchine aventi: due 1Gb di RAM e con l'utilizzazione di 1 core della cpu, mentre, le restanti due avevano 512MB di RAM e 1 core a disposizione. Interconnesse tra di loro mediante interfaccia di rete bridge-virtualBox. Il SO di base installato su ogni nodo è la versione mate di Linux Mint 18.0.

3. Creazione e inizializzazione.
   Installato e configurato il sistema è ora possibile creare lo Swarm su cui andremo a distribuire il servizio. Per chi non lo conoscesse Swarm è la soluzione integrata di Docker per la gestione di un cluster in grado di eseguire container o come li definisce Swarm stesso, servizi; un servizio è formato da uno o più container, espone una porta, ed è in grado di scalare facilmente in termini di numero di container. L'intero strato di bilanciamento detto network, nella sua configurazione più semplice, è interamente gestito da Docker Swarm stesso e permette di astrarre il servizio dai container reali che lo fanno girare. Swarm richiede almeno 2 nodi (un manager ed un worker) per funzionare, anche se per motivi di ridondanza è sempre meglio avere almeno un paio di worker e due o più manager.

Con il comando:
`docker swarm init -advertise-addr <IP of my server leader>`
Creo e inizializzo il mio primo nodo master e con semplicità.

```
$ docker swarm init --advertise-addr 10.0.0.3
Swarm initialized:\
current node (jkr2vnv7ntyj0p0zgmdj5t17c) is now a manager.

To add a worker to this swarm, run the following command:

docker swarm join \
--token SWMTKN-1-06hpb763v16zbsplkmbw9yp3uyt5wwti5 \
10.0.0.3:2377

To add a manager to this swarm, run 'docker swarm join-token manager' and follo
```

Il comando ci fornisce anche in output già i comandi per fare join di nuovi worker al cluster. In questo modo aggiungerò due nodi worker e un ulteriore nodo master in modo da avere tutte e 4 le macchine attive e pronte all'uso.
Il nostro cluster è quindi composto da:

- node1, node2 in funzione di manager.
- node2 e node3 come worker.

Alcuni comandi messi a disposizione per verificare lo stato e l'attuale distribuzione del carico di lavoro sono:
`docker info`
`docker ps`

```
docker node ls
docker node ps <nome del servizio>
```

4. Distribuzione del servizio:
   Seppur possa sembrare strano, avere più manager che worker. Inoltre è da notare che per i cluster Swarm i manager fungono anche da worker. Il che vuol dire: attualmente abbiamo un cluster in grado di supportare 4 container in parallelo per l'esecuzione di un singolo servizio (potrebbero essere anche di più, ma si andrebbe a non avere più parallelismo). L'unica differenza, dunque, tra un worker ed un manager è che sui manager è possibile eseguire i comandi di gestione del cluster e di deploy dei servizi. Su quale worker poi i servizi andranno in esecuzione è a discrezione di Swarm a noi non è dato a priori sapere quale criterio di scheduling lo sciame userà per sciegliere i nodi su cui verranno sclati i tasks.
   Quale modo migliore di testare un cluster Swarm se non creare un servizio che eroghi qualcosa? A questo punto useremo come base di partenza il servizio che abbiamo creato inizialmente nella prima fase useremo il nostro servizio web. Il comando da usare per creare un servizio:
   `docker service create -name myServiceName -replicas 3 -p 9111:9111 node:app0.1`
   I parametri sono molto semplicemente i seguenti:
   `-name`: Il nome che vogliamo dare al servizio. Nel nostro caso lo abbiamo chiamato 'myServiceName'
   `-replicas`: Il numero di repliche (container) che vogliamo deployare. Partiamo con 3
   `-p`: Il mapping delle porte. Node parte di default erogando la porta 9111, noi vogliamo accedervi usando la porta 9111, faremo quindi questo mapping 9111:9111 per maggior chiarezza da ora in poi usero le porte standard per un web server 8080:80
   `node:app0.1`: Il nome del container su cui si basa il servizio

Come conseguenza al comando avremo quindi generato 3 container Linux Alpine (su che nodi al momento non lo sappiamo) che erogano il servizio '`node:app0.1`' sulla porta 8080. Questo significa che contattando la porta 8080 del nostro cluster andremo ad utilizzare uno di questi container. Le logiche di bilanciamento, al momento, sono tutte in mano a Docker Swarm.

5. Scaliamo il servizio:
   I servizi in Docker Swarm possono essere scalati molto velocemente come vedremo nell'esempio sottostante.
   In pratica abbiamo detto a Docker di scalare il servizio `node:app0.1` fino ad avere 20 repliche, in poche parole abbiamo aggiunto 17 tasks ai 3 già presenti. Tutta la loro configurazione di rete e il bilanciamento della network viene fatta in automatico da docker:
   `docker node ls` tutti i servizi in esecuzione
   `docker node ps node:app0.1` vediamo ogni singolo tasks di quel servizio su quale nodo del cluster è in esecuzione con il rispettivo stato.

6. Contattiamo il server: Infine aprendo il browser e contattando il server web all'indirizzo e la porta su cui esso è in ascolto ci risponderà con un oggetto json dove è contenuto un id univoco per l'oggetto e l'Id del container che ha eseguito la richiesta. Quindi infine per vedere quali container e come rispondevano alle richieste è stato sufficiente fare uno script bash che tramite un ciclo for e sfruttando il comando bash curl effettua un numero r di richieste salvando gli oggetti json ricevuti in risposta in un file sucessivamente ispezionabile.

```
#!bin/bash
docker node ps node:app0.1 >> result;
for i in ‘seq 1 20‘;
do
  curl -i -X GET "http://localhost:9111/guid" >> result;
  #echo -e "\n" >> result;
done;
exit 0;
```

concetti di base di swarm servizio ecc. https://docs.docker.com/glossary/

4.1 Deploy service

Originariamente per effettuare il deploy di servizi si richiedeva necessariamente l'installazione Docker e Swarm come due componenti software distinti su ogni macchina candidata a far parte di un determinato gruppo di lavoro. Poichè questi due sistemi software svolgono compiti differenti ed originariamente erano distribuiti separatamente. Questo portava ad inevitabili complicazioni nella fase di start-up dovute principalmente all'overhead solo della fase di configurazione, rendendo eccessivamente onerosa questa fase preparatoria del lavoro. Ma grazie alla notorietà acquisita da queste tecnologie e dalla continua espansione della cerchia di sviluppatori e utilizzatori, ha fatto si che guidati dalla necessità si sviluppassero dei tools aggiuntivi per migliorare, velocizzare e semplificare l'utilizzo di questi strumenti. Attualmente alcuni di essi fanno a loro volta parte degli strumenti interni offerti da Docker. Ad esempio tool come docker-compose o docker-machine entrambi per automatizzare le fasi di build delle VM e il deploy di servizi e applicazioni. Per ulteriori informazioni in maggior dettaglio consultare la documentazione ufficiale Compose e Machine

4.2 Apache MESOS

Mesos

Mesos è un progetto open source ospitato dalla Apache Foundation. Lo scopo di Mesos sta nel fornire "un sistema kernel distribuito". Mesos tenta di trasportare i principi del kernel Linux sulla temporizzazione dei processi e applicarli a un cluster o a un intero datacenter.
Il progetto Mesos è cominciato nel 2012 ed è stato recentemente sottoposto a un processo di reingegnerizzazione per renderlo in grado di gestire i container Docker per maggiori info visita pagina ufficiale http://mesos.apache.org.
Mesos utilizza ZooKeeper per implementare i concetti dei servizi.
Mesos eccelle sulla portabilità ed è possibile creare dei cluster Mesos che si

estendono su diversi datacenter e su differenti fornitori di servizi cloud.

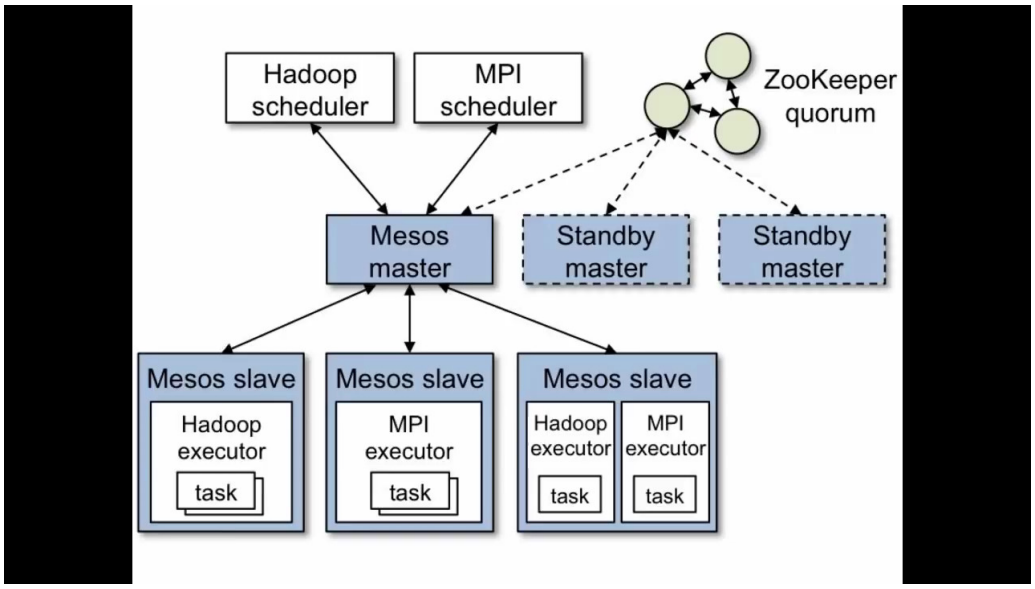

Figura 4.1: Schema architetturale di Apache MESOS.

4.2.1 Instroduzione Mesosphere

Qualcosa in più Mesos on Mesosphere e Marathon

Mesos è un cluster manager che mette a disposizione di un framework delle risorse computazionali. Marathon è un framework specializzato nell'esecuzione di applicazioni, inclusi i container, su cluster Mesos. Entrambi sono stati pensati per gestire decine di migliaia di nodi e sono quindi adatti a lavorare su larga scala. Anche in questo caso gli strumenti e le API adoperate si rivelano differenti da quelli delle altre soluzioni obbligando gli utilizzatori a ripartire sostanzialmente da zero. Passiamo ora all'elenco degli elementi che costituiscono un cluster Mesos:

- **Master.** Zookeeper gestisce un quantitativo minimo di nodi master pari a 3 e garantisce alta disponibilità affidandosi ad un quorum tra questi nodi.

- **Slave.** Nodi che si occupano di eseguire le task inviate dal framework.
- **Framework.** E' l'elemento che si occupa di gestire i vari workload ed al quale Mesos offre risorse computazionali.

A sua volta Marathon offre:

- **Service discovery.** Visibilità attraverso varie opzioni (servizio DNS dedicato etc.).
- **Load balancing.** Via HAProxy.
- **Constraint Management.** Per controllare in quale parte del cluster siano eseguiti determinati workload, per mantenere a disposizione un determinato set di risorse per questi ultimi ed altre mansioni.
- **Applicazioni.** I servizi in esecuzione – container o altri workload.
- **REST API:** effettua il deploy, la modifica e l'eliminazione dei workload.

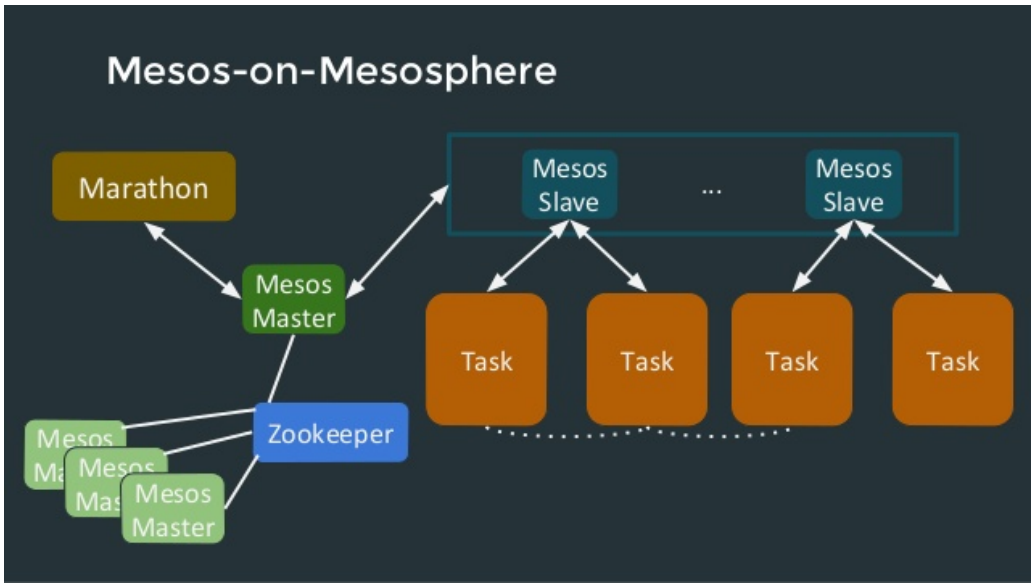

Figura 4.2: Mesos on Mesosphere architecture source

4.3 kubernetes

Kubernetes

E' il progetto open source sviluppato da Google che adotta un approccio totalmente differente da Docker. Kubernetes è consigliato per cluster di medie-grandi dimensioni ed applicazioni complesse. Rispetto a Swarm, la curva di apprendimento è molto più ripida – ma coloro che ne apprenderanno il funzionamento saranno ripagati dalla flessibilità / modularità della soluzione e dalla possibilità di gestire anche deploy su larga scala. Un cluster Kubernetes è formato da:

- **Master.** Si occupa di gestire le chiamate delle API, assegnare workload e supervisionare la configurazione degli stati.

- **Minion.** I server che eseguono workload o altri elementi non localizzati nel Master.

- **Pod.** Unità computazionali costituite da uno o più container il cui deploy è stato effettuato sul medesimo host. Si occupano di eseguire task ed hanno un singolo IP.

- **Servizi.** Front end e load balancer dei pod, mettono a disposizione un floating IP per accedere ai pod che eseguono il servizio. Replication controller. Si occupano di supervisionare un determinato numero di copie dei pod richiesti.

- **Label.** Tag utilizzati per identificare pod, replication controller e servizi.

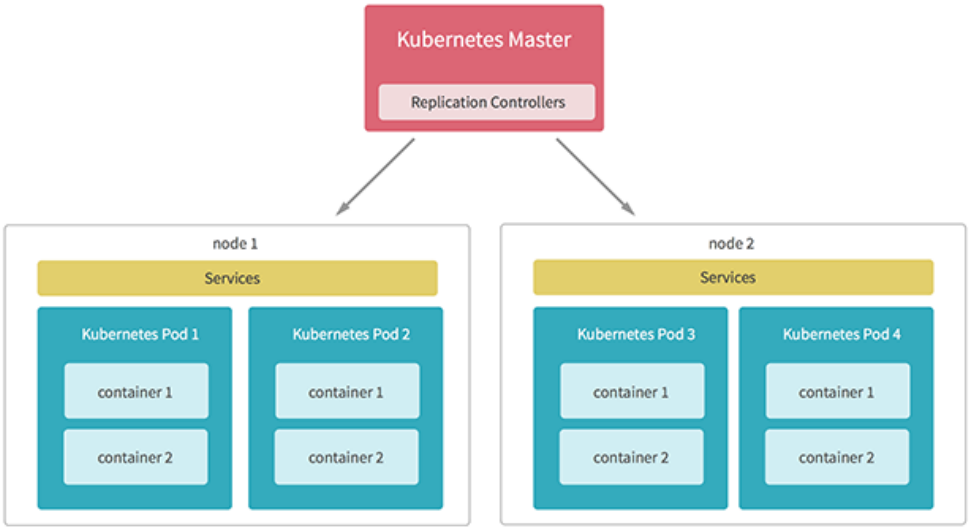

Figura 4.3: kubernetes architecture source

Bibliografia

Documenti di riferimento:

$-Dockervirtualizationadminguide.$ (reperoibile in doc/Docker-for-Virtualization-Admin-eBook.pdf)
$-Nistdefinizionedicloudcomputing.$ (reperibile in/doc/nist-traduzione.pdf)

Siti di riferimento:

- Primi passi con i registry di Docker
$-https://it.wikipedia.org/wiki/Docker$
$-http://goo.gl/hLLCoc$
$-https://it.wikipedia.org/wiki/Comunicazione_tra_processi$
$-https://it.wikipedia.org/wiki/LXC$
$-https://it.wikipedia.org/wiki/Cloud_computing$
$-http://www.cloudtalk.it/container-cosa-sono-come-funzionano$
$-http://www.cwi.it/data-center/virtualizzazione/container-cosa-funzionano-virtualizzazione-86516$

Altri link utili/interessanti:

https://it.wikipedia.org/wiki/Docker
https://en.wikipedia.org/wiki/Docker_(software)
https://www.zoomingin.net/installiamo-docker-su-ubuntu/
http://www.html.it/articoli/docker-i-nostri-servizi-in-un-container-a-prova-di-bomba/
http://www.html.it/articoli/vagrant-virtualizzazione-e-provisioning-senza-sforzo/

https://www.docker.com/what-docker
https://www.docker.com/
https://www.miamammausalinux.org/2017/01/giochiamo-con-docker/
https://github.com/tianon/docker-brew-ubuntu-core/blob/8ec739cf49ac7fb7517d9ba97b0421
http://mesos.apache.org/gettingstarted/ http://mesos.apache.org/documentation/latest/arc
http://mesos.apache.org/documentation/latest/app-framework-development-guide/ https://www.digitalocean.com/community/tutorials/an-introduction-to-mesosphere https://www.digitalocean.com/community/tutorials/how-to-configure-a-production-ready-mesosphere-cluster-on-ubuntu-14-04

Video tutorial di riferimento:

$-https://www.youtube.com/watch?v = VeiUjkiqo9E\#t = 60$

Thank you Meshua Galassi.