UNIVERSITA' DEGLI STUDI DI
NAPOLI FEDERICO II

Scuola Politecnica e delle Scienze di Base
Corso di Laurea in Ingegneria Informatica

Elaborato finale in **Reti di Calcolatori**

# *Virtualizzazione basata su container mediante la tecnologia Docker*

Anno Accademico 2016/2017

Candidato:

**Vincenzo Mangiacapra**

**matr. N46002199**

*Alla mia famiglia.*
*A Ilaria.*

# Indice

# Introduzione

Con il termine **Virtualizzazione** s'intende una procedura informatica volta alla definizione di versioni virtuali di risorse fisiche (hardware) o software, come ad esempio banchi di memoria RAM, un disco rigido, schede di rete, periferiche, sistemi operativi ed applicazioni con il fine di renderne l'utilizzo pari a quello delle risorse reali. Una macchina virtuale individua un applicativo che è in grado di realizzare un ambiente caratterizzato da risorse analoghe a quelle disponibili in una macchina fisica corrispondente. Se si considera un sistema operativo installato su una macchina virtuale, si ha la possibilità di installare nuovi programmi, periferiche e navigare in rete come se si stesse interagendo con una macchina reale.

La virtualizzazione di un sistema operativo richiede un grande sforzo computazionale da parte della macchina fisica su cui è fatta girare la macchina virtuale, per questo motivo nel passato era molto costoso utilizzare tecniche di virtualizzazione. Oggi, grazie soprattutto alla diminuzione dei costi e all'accessibilità verso calcolatori abbastanza potenti, la virtualizzazione trova largo impiego grazie soprattutto ai tanti vantaggi che ne conseguono. Tra questi troviamo la portabilità del sistema, il fatto che è possibile evitare dual boot, la possibilità di testare applicativi in fase embrionale in ambiente protetto ecc.

L'obiettivo di questo elaborato è di fornire una prima **differenza** tra le due grandi metodologie di virtualizzazione: quella basata su **Hypervisor** e quella basata su **Container** analizzandone vantaggi e svantaggi per entrambe. Si analizzerà in particolare

la seconda tecnica, la container-based, utilizzando gli strumenti che il sistema operativo Linux mette a disposizione tramite gli **LXC (Linux Container)** ed infine il focus del discorso si sposterà su **Docker**, una piattaforma open source che fornisce strumenti per la creazione e la gestione dei container, attualmente molto utilizzata nel settore del cloud computing.

# Capitolo 1: I Container

## 1.1 Differenza tra virtualizzazione Hypervisor-based e Container-based

Come già accennato nell'introduzione, esistono due tecniche di virtualizzazione: Hypervisor-based e Container-based.

La prima tecnica permette la virtualizzazione di un'intera macchina fisica ed è quindi una tecnica di virtualizzazione hardware. Il virtual macchine monitor (Hypervisor) si pone tra l'hardware della macchina fisica su cui è installato e le macchine virtuali che lo utilizzano. Esso opera in maniera trasparente senza pesare con la propria attività sul funzionamento e sulle prestazione dei vari sistemi operativi, svolge attività di controllo al di sopra di ogni sistema, l' hypervisor può controllare ed interrompere attività pericolose, questo fa si che tale tecnica sia molto utilizzata nelle fasi di collaudo di un nuovo software.

L'hypervisor può allocare le risorse dinamicamente quando e dove sono necessarie, riducendo il tempo di messa in opera di nuovi sistemi.

I **vantaggi** che spingono a utilizzare questa tecnica sono:

- Migliore utilizzo di risorse (una singola macchina fisica è divisa in più VM)
- Scalabilità

Questa tecnica d'altra parte soffre di molte **limitazioni**: ogni macchina virtuale ha bisogno di un'allocazione di CPU, memoria RAM, memoria di massa e di un intero sistema operativo dedicato. Le risorse come la CPU, la RAM e lo storage sono quelle della macchina fisica dove l'hypervisor è ospitato per cui il numero di VM perfettamente funzionanti è per forza limitato. Inoltre se sono presenti molte VM diventa complicato gestire molti OS. Infatti, più VM sono presenti, più risorse vengono consumate. Ergo, se

ho bisogno di molte VM su un'unica macchina fisica, quest' ultima deve essere molto potente.

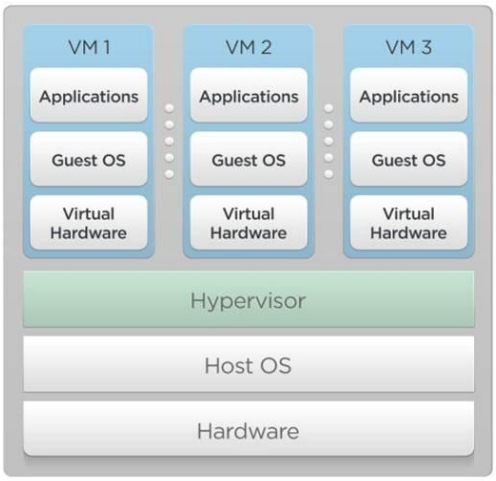



**Figura 1 - Schema di sistema Hypervisor-based [1]**

Per ovviare a questi problemi che limitano l'uso delle VM in molti ambiti, è stata sviluppata un'altra tecnica di virtualizzazione, quella Container-based.

Questa tecnica è, a differenza della precedente, a livello di sistema operativo ed usa il kernel dell' OS installato sull'host per far girare più istanze ospiti. Queste istanze user-space sono completamente indipendenti tra loro, esse vengono dette Container.

Ogni Container come abbiamo detto è indipendente dall'altro, infatti ognuno ha un suo file system, i suoi processi, la propria memoria, i propri dispositivi e la propria interfaccia di rete.

I principali vantaggi di questa tecnica sono:

- Elevata leggerezza computazionale dei container rispetto ad un'intera VM.
- I Container non hanno bisogno di un OS dedicato
- Minor utilizzo di risorse dell'host (CPU, RAM, storage)
- Su ogni host è possibile avere molti più container che VM.
- Maggiore portabilità: i container una volta creati, possono girare su qualsiasi host.

Come già accennato in precedenza, la virtualizzazione basata su container lavora a livello di sistema operativo, questo vuol dire che ogni container utilizza lo stesso kernel, ossia quello dell'host. Per questo motivo tale tecnica viene vista come "meno isolante" rispetto alla virtualizzazione hypervisor-based. Dal punto di vista dell'utente però ogni container

viene visto come sistema completamente indipendente grazie al namespace kernel messo a disposizione dal kernel di Linux che permette a diversi processi di avere una visione eterogenea del sistema.

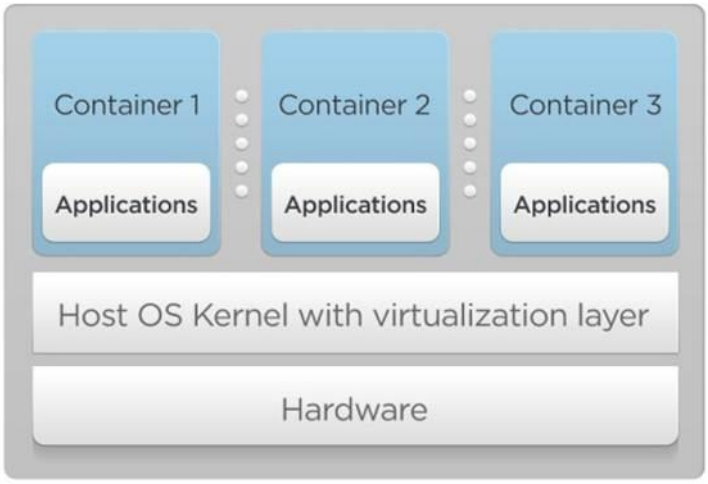



**Figura 2 - Struttura di un sistema Container-based [1]**

## 1.2 LinuX Container (LXC)

LXC è la metodologia con cui Linux esegue la virtualizzazione container-based. È quindi una tecnica di virtualizzazione di Linux a livello di sistema operativo. Questa tecnica esegue diversi sistemi Linux, ossia i containers, completamente isolati tra loro su un singolo host di controllo, detto LXC host. LXC fornisce ad ogni container un ambiente isolato con la propria CPU, la propria RAM, il proprio storage e le proprie periferiche.

Le caratteristiche di LXC [2] che permettono la creazione e la gestione dei container sono:

**Kernel namespaces (ipc, uts, mount, pid, network e user)**: spazi di nomi che permettono un'astrazione tale da rendere ogni container indipendente. In pratica i kernel namespaces forniscono ad ogni container un proprio spazio di PID, IPC ed una propria interfaccia di rete.

**Cgroups**: un meccanismo di aggregazione e partizionamento di un set di tasks e di tutti i loro futuri tasks figli in gruppi gerarchici con comportamenti specializzati. Questo meccanismo permette di gestire i limiti e priorità di utilizzo delle risorse senza dover utilizzare una macchina virtuale.

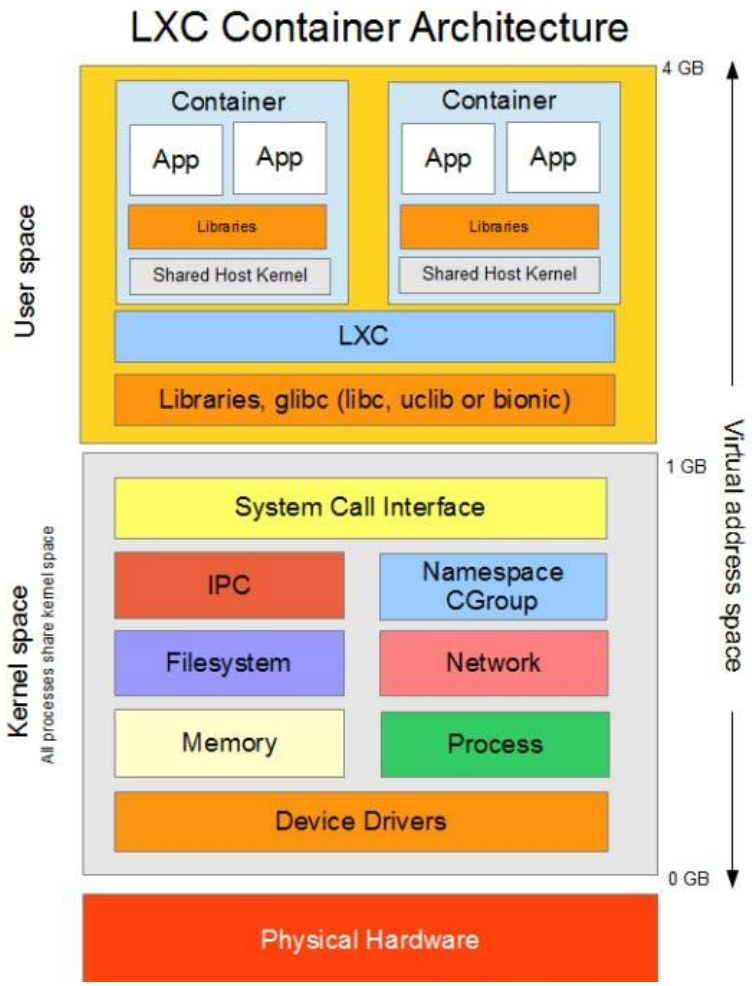



**Figura 3 - Architettura di un container LXC[3]**

### 1.2.1 Kernel Namespaces

Il compito dei namespaces è quello di fornire un'astrazione di una risorsa globale del sistema e far si che ogni processo interno al namespace la veda isolata rispetto a processi di altri namespaces. L'utilizzo dei namespaces è diffuso in molti settori dell'informatica. Questi trovano largo impiego anche nella virtualizzazione container-based, infatti sono utilizzati da LXC. L'utilizzo dei namespaces consente ad ogni container di utilizzare le risorse in maniera isolata dagli altri container, in pratica è come se ognuno avesse le proprie risorse a disposizione indipendentemente dagli altri.

Linux implementa 6 tipi di namespaces [4]:

**PID Namespace**

Isolano lo spazio dei PID (Process ID). In altre parole, processi che si trovano in diversi PID Namespace possono avere lo stesso PID. Uno dei maggiori vantaggi di questi namespace è che permettono ai container di essere migrati attraverso gli host mantenendo gli stessi PID per i processi interni al container. Inoltre, i PID namespaces permettono ad ogni container di avere il loro processo init (PID 1) ossia "il precursore di tutti i processi", che gestisce le attività di inizializzazione del sistema e riunisce i processi orfani al momento della loro terminazione. Per una particolare istanza di PID namespace , ogni processo è dotato di due PID, uno interno al namespace ed uno esterno. I PID namespace permettono anche un processo di nidificazione, i processi verranno identificati attraverso il percorso dei PID a partire dalla radice, ossia dal PID relativo al namespace più esterno.

**IPC Namespace**

Isolano alcune risorse IPC (Inter-Process Communication), ovvero gli oggetti del sistema che regolano le interazioni tra i processi come le code di messaggi e i semafori. Ogni istanza di IPC namespace avrà le sue chiavi per l'identificazione degli IPC ed ovviamente IPC di diversi namespace possono avere le stesse chiavi di identificazione.

**UTS Namespace**

Isolano i due identificatori di sistema (nodename e domainname) che vengono restituiti dalla *system call* `uname()`. Gli UTS namespaces permettono ad ogni container di avere i propri hostname e domain name. Questo può risultare particolarmente utile per script ed applicazioni che basano i loro indirizzamenti su questi nomi.

**Network Namespace**

Forniscono l'isolamento delle risorse del sistema associate al networking. Quindi, ogni network namespace possiede i propri device di rete, indirizzi IP, tabelle di routing IP, numeri di porta ecc. Questi namespace rendono i container particolarmente utili anche sotto il profilo dell'utilizzo nelle reti: ogni container può avere il suo dispositivo di rete

virtuale e le sue applicazioni di rete. In questo modo le regole di routing già presenti nel sistema host possono indirizzare i pacchetti al dispositivo di rete associato ad uno specifico container. Ad esempio, è possibile avere multipli web server "containerizzati" sullo stesso sistema host con ogni server collegato alla porta tcp 80 all'interno del suo network namespace.

**User Namespace**

Permettono di isolare l'utente e gli spazi di group ID. In altre parole, un processo di un utente ed i group ID possono risultare diversi all'interno e all'esterno di un user namespace. Lo scenario più interessante è quello in cui un processo può avere un user ID senza privilegi all'esterno dell' user namespace e contemporaneamente avere un user ID con priorità massima all'interno del namespace. Questo significa che il processo ha pieni privilegi di root per operazioni interne al namespace, ma non ha alcun privilegio particolare per operazioni al di fuori dell'ambiente del namespace.

**Mount Namespace**

Isolano il set di mount points del file system visto da un gruppo di processi, ovvero, i processi che si trovano in Mount Namespace diversi possono avere diverse viste della gerarchia del file system. Con l'introduzione dei Mount Namespace, le system call `mount`() e `unmount`() cessano di operare su un set di mount points visibile a tutti i processi del sistema, ma iniziano ad effettuare operazioni che vanno ad influenzare soltanto il Mount Namespace associato al processo chiamante.

### 1.2.2 CGroups

I Control Groups sono un meccanismo di aggregazione e partizionamento di task e loro futuri figli in una gerarchia di gruppi con comportamenti specializzati. È possibile limitare il quantitativo di risorse che ogni CGroup può utilizzare, per questo vengono detti gruppi di controllo.[5]

Un CGroup quindi è un insieme di processi che condividono gli stessi criteri. Questi

gruppi possono essere gerarchici, cioè i processi figli possono ereditare le limitazioni che i processi padre avevano acquisito essendo membri di un gruppo di controllo. Il kernel di Linux fornisce l'accesso a più controllori o sottosistemi attraverso l'interfaccia di cgroups. Tale interfaccia permette solo il raggruppamento dei processi mentre la gestione delle risorse e dei gruppi è affidata ai sottosistemi che compongono tale interfaccia. L'utilizzo dei CGroups è molto utile perché permette di gestire in maniera impeccabile le risorse per processi ben definiti. Ad esempio su di un server in cui girano una web application e un database si possono limitare le risorse usate dall'applicativo affinché non venga compromesso il funzionamento del db.

CGroup può anche essere utilizzato per isolare o dare comandi specifici a gruppi di processi, di fatti esistono 2 tipi di sottosistemi: uno dedito alla gestione delle risorse ed uno che gestisce comandi speciali per gruppi di tasks.

Di seguito possiamo osservare graficamente come è possibile definire per ogni gruppo l'allocazione e la limitazione delle risorse che ogni gruppo può utilizzare, realizzando cosi la "resource isolation". Il tutto viene gestito operando a basso livello, operando direttamente con il kernel di Linux, quindi fornendo uno scarso livello di astrazione.

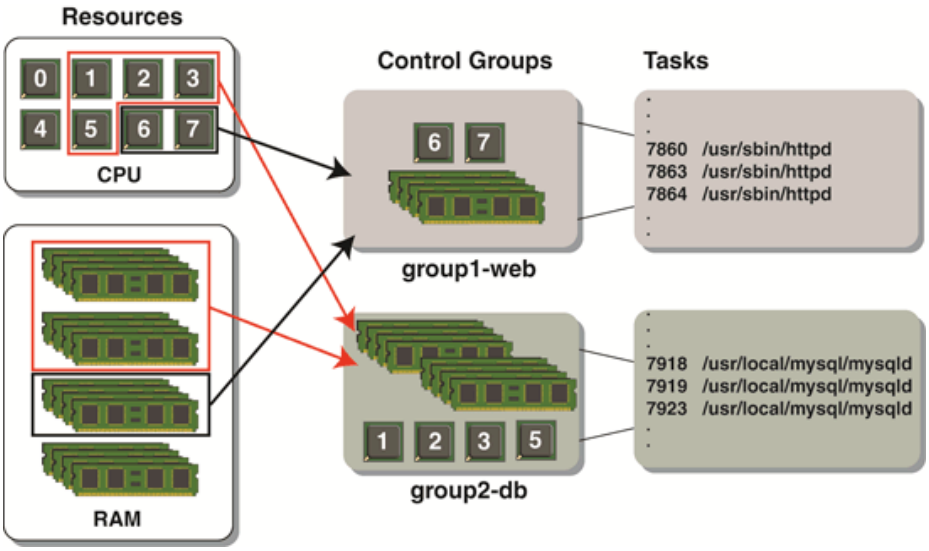



**Figura 4 - Allocazione di risorse per due CGroups [6]**

## 1.3 Software per la gestione dei container

Nei precedenti paragrafi abbiamo introdotto il concetto di container e la tecnica di virtualizzazione basata su questa tecnologia: OS-oriented. Abbiamo inoltre osservato come Linux gestisce i container attraverso LXC utilizzando i Kernel Namespace e i CGroups. Come abbiamo visto, tale meccanismo prevede operazioni a basso livello con un limitato livello di astrazione che rendono complicate le interazioni con l'utente.

Con il tempo sono nati software e strumenti che operando ad alto livello rendono più semplice e diretta la gestione dei container in Linux.

Nel seguito daremo una panoramica su quelli più utilizzati come Kubernetes di Google e Apache Mesos dando ampia osservazione al software principe per la gestione dei container: Docker.

### 1.3.1 Kubernetes


Kubernetes [7] , dal greco kubernetes, che significa "timoniere" o "pilota", è un software open source nato dall'esperienza decennale di Google con i container in Linux.

È un software per l'automazione della distribuzione, scalabilità e gestione di applicazioni container-based.

Kubernetes introduce il concetto di pod: una serie di container che devono essere collocati insieme nello stesso host che cooperano al fine di realizzare un applicativo. Kubernetes fornisce anche un meccanismo di replicazione detto replication controller che ha il compito di garantire che tutti i pod su di un host non abbiano comportamenti indesiderati. Se un pod è danneggiato o non esegue come dovrebbe, kubernetes lo elimina e lo sostituisce.

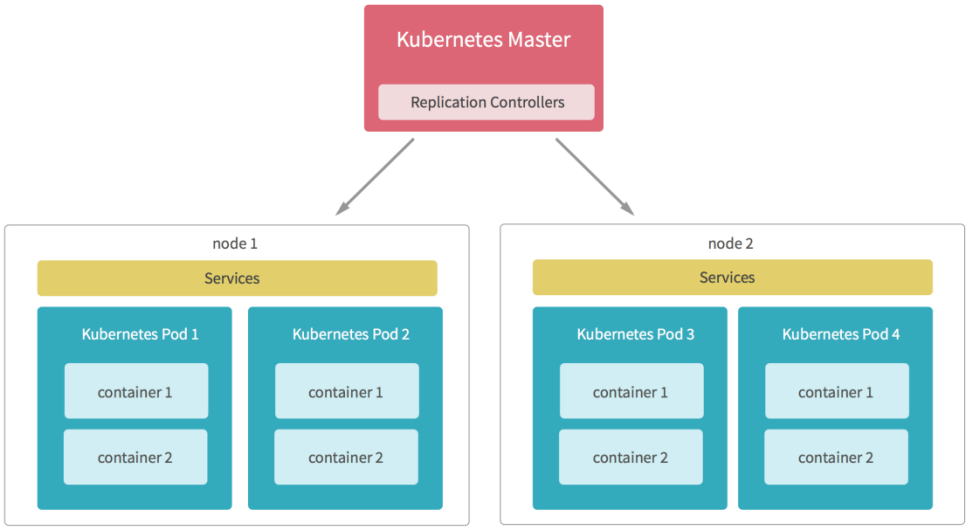



**Figura 5 - Architettura di un sistema sviluppato con Kubernetes[8]**

### 1.3.2 Apache Mesos

Mesos è un progetto della Apache Foundation nell'università della California. L'obbiettivo di Mesos è quello di fornire un kernel distribuito, tentando di esportare i principi del kernel di Linux sulla temporizzazione dei processi ed applicarli ad un cluster o all'interno di un data center. Esso utilizza Zookeper (un software open source sempre di Apache per la sincronizzazione dei sistemi distribuiti) per implementare i servizi. Mesos è utilizzato per far si che più macchine fisiche (cluster) vengano unificate in un unico container, per questo è molto utilizzato nei sistemi distribuiti, di fatti è un cluster-manager. Il vantaggio di Mesos è che permette di eseguire moderne applicazioni di data processing contemporaneamente.

Attualmente Mesos sta subendo un intervento di reingegnerizzazione per permettergli di adoperare molte delle funzionalità di Kubernetes e Docker.

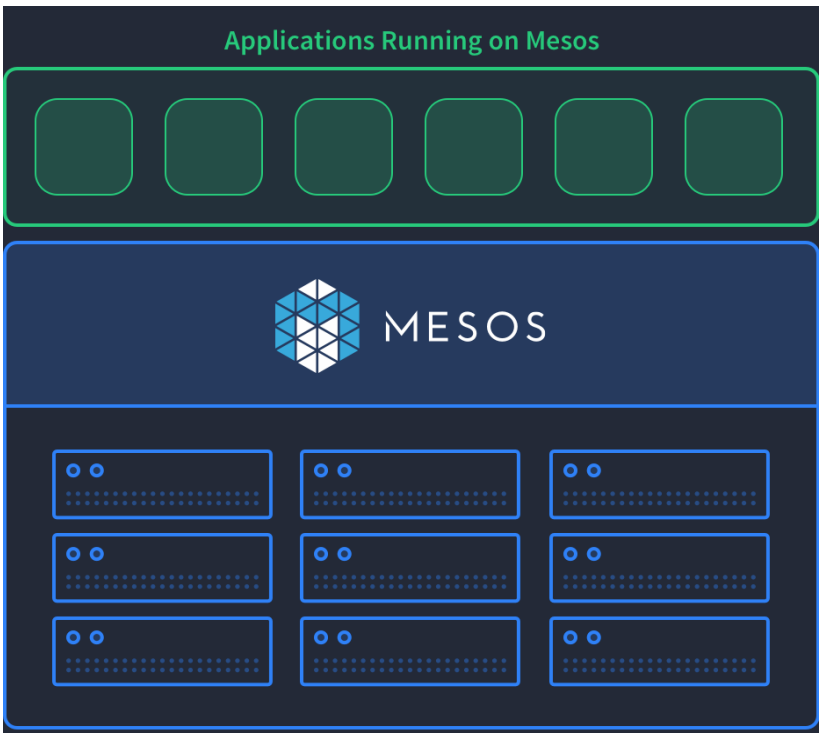



**Figura 6 – Architettura di un sistema sviluppato con MESOS[9]**

Come possiamo notare dalla figura, in basso troviamo una serie di macchine fisiche che MESOS raggruppa in un unico container. Quando utilizziamo un' applicazione attraverso MESOS, questo gestisce il cluster permettendone l'esecuzione.

### 1.3.3 Docker

Docker è un software open source che riesce ad automatizzare e semplificare la gestione dei container, rendendo più immediata la realizzazione di applicazioni che sfruttano questa

tecnologia. Esso fornisce un livello di astrazione addizionale rispetto alla virtualizzazione a livello di sistema operativo offerta da LXC. Docker fa comunque uso delle caratteristiche di isolamento già citate del kernel Linux come i kernel namespace e CGroups, in più fornisce un particolare file system che consente a container indipendenti di essere eseguiti in una stessa istanza Linux.

### 1.3.4 Interazione tra Docker, Kubernetes e Mesos.

I software che abbiamo descritto in precedenza sono molto spesso utilizzati insieme per creare un sistema più vasto per applicazioni complesse e distribuite, di fatto Docker viene utilizzato per la creazione e la gestione dei container, Mesos mette insieme tutti i container e Kubernetes fa da framework  e da controller al sistema, gestendo il problema della portabilità dell'applicativo e della replicazione. Di seguito possiamo osservare graficamente tale infrastruttura.

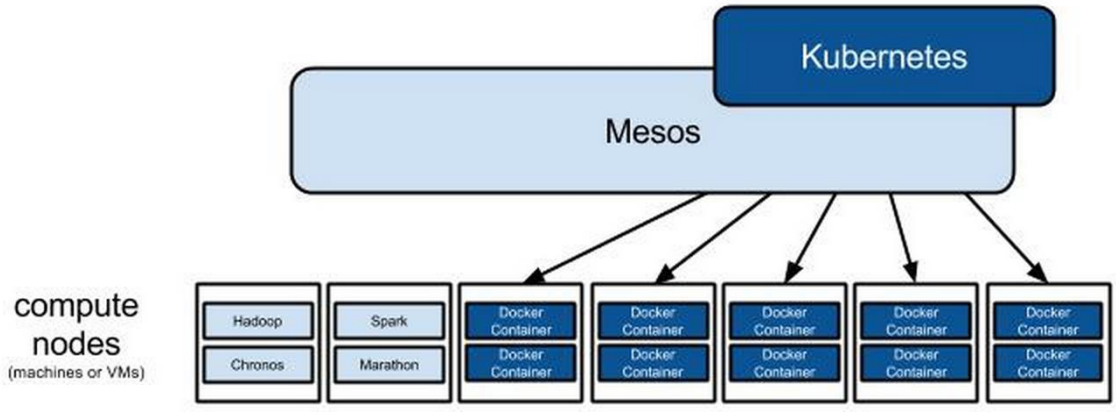



**Figura 7 - interazione dei 3 software**

Per avere un idea ancora più chiara dell'interazione tra questi software è possibile citare Solomon Hykes, fondatore e CEO di Docker:

"If a Docker application is a Lego brick, Kubernetes would be like a kit for building the Millennium Falcon and the Mesos cluster would be like a whole Star Wars universe made of Legos." [9]

# Capitolo 2: Docker

## 2.1 Funzionamento e caratteristiche principali

Come già accennato nel precedente paragrafo, Docker [10] è un software open source che si basa principalmente sul concetto di LinuX Container (LXC) ma che ne aumenta l'accessibilità introducendo API di alto livello. Esso fornisce un processo di virtualizzazione che esegue processi UNIX in isolamento andando a semplificare quella che è la distribuzione automatizzata di software in ambiente sicuro.

Docker utilizza come LXC, il concept dei container standard che comprende oltre al software, anche tutte le dipendenze, file binari, file di configurazione e le librerie utili all'esecuzione di quest'ultimo. Può essere eseguito su qualsiasi sistema Linux a 64 bit che supporta il meccanismo dei CGroups. La comodità dei container è che essi possono essere implementati su di un pc portatile, su di un server o su una workstation con un elevata portabilità, possono essere implementati anche in ambiente distribuito come nel cloud computing, preservando comunque l'ambiente che li ospita. Questo li rende adatti a molti impieghi: distribuzione continua, implementazioni web, ecc.

Solomon Hykes, CEO di docker, spiega che la virtualizzazione leggera fornita dai container è dovuta al fatto che ognuno di essi ha il proprio file system e sono isolati a livello di processo. Infatti è proprio questo lo scopo dei container: utilizzare lo stesso kernel dell'host ma in modo isolato l'uno dall' altro grazie ai namespace e alla limitazione nell'utilizzo delle risorse fornite dai CGroups, il tutto gestito mediante un API di alto livello con cui è possibile effettuare operazioni sui container: creazione, avvio, arresto ecc.

Alcune delle caratteristiche principali di Docker sono:

**Isolamento del file system** : Ogni container ha un proprio file system indipendente dagli altri container. Un processo di un container A non può accedere ad una directory del container B.

**Isolamento delle risorse** : Le risorse del sistema host come la CPU e la RAM possono essere assegnate in proporzioni diverse ad ogni container utilizzando i CGroups.

**Isolamento della rete** : Ogni container ha la propria scheda di rete virtuale, il proprio pool di indirizzi IP e le proprie porte TCP e UDP. Ciò consente a processi di container diversi di comunicare con l'esterno utilizzando la stessa porta e indirizzi differenti pur condividendo lo stesso host.

**Copy-On-Write** : I file system di ogni container vengono creati ed aggiornati mediante copy on write: un meccanismo che prevede la copia ad ogni eventuale scrittura. Questo rende i container particolarmente utili negli aggiornamenti di applicazioni distribuite garantendo una elevata velocità ed evitando un dispendio di memoria.

**Logging** : Gli stream standard di ogni container(stdin, stderr, stdout), quindi tutti i flussi di input, output o messaggi di errore, vengono salvati in log così da poter monitorare l'attività di ogni container.

**Gestione dei cambiamenti** : Tutti i cambiamenti effettuati al file system di un container possono essere salvati in un' immagine che potrebbe permettere in un futuro di ricreare il container di partenza con tutte le modifiche già attuate.

**Shell Interattiva** : Docker può creare uno pseudo terminale e collegarlo allo stream input di ogni container, in modo da interagire in maniera rapida direttamente con questi ultimi: un terminale "usa e getta"

## 2.2 Architettura

Dopo aver presentato il software e le sue caratteristiche, è necessario introdurre i concetti fondamentali che contraddistinguono l'architettura di Docker. Bisogna quindi capire come Docker sfrutti il kernel di Linux per la creazione e la gestione dei container.

Docker [11] ha una struttura client-server classica che ne permette il funzionamento. Il lato client è rappresentato dall'interfaccia a linea di comando o da API remote che tramite la rete permettono di controllare e monitorare i container in esecuzione e quelli presenti sull'host. Il lato server invece è rappresentato dal processo Daemon di Docker, ossia un processo eseguito in background, non visibile all'utente, che viene eseguito sull'host ed ha il compito di esaudire ogni richiesta del client nella gestione dei container, esso può essere sulla stessa macchina in cui è presente il client o anche su un host remoto.

Per generare un container, il daemon utilizza le cosiddette immagini. Un' immagine è un modello di sola lettura che viene utilizzato per la creazione di un container. È costituita da una vista di una distribuzione Linux come Ubuntu o Fedora ed un insieme di applicazioni o ambienti di runtime come Apache o Java. Gli utenti che usano questo software possono creare le proprie immagini a piacimento oppure utilizzarne altre già esistenti presenti nell'archivio online di Docker: il Docker HUB che costituisce il più grande registro pubblico per immagini.

Un registro Docker è una repository in cui è possibile caricare o scaricare immagini. L'utilizzo dell'HUB è molto comodo nel momento in cui sono presenti immagini caricate da tutta la Docker community ma spesso i team di lavoro preferiscono utilizzare registri aziendali proprietari accessibili solo ai membri del gruppo.

Partendo da un immagine è possibile allora creare un container Docker, ossia una directory isolata che contiene tutto il necessario di cui l'applicazione contenuta al suo interno ha bisogno per eseguire in maniera corretta. Sfruttando quindi il kernel del sistema operativo presente sull'host, il container che a sua volta è basato su un immagine Docker con un sistema operativo, può eseguire l'applicazione per cui è stato creato, il tutto però

senza dipendenze con il tipo di host su cui Docker è installato. Questa proprietà rende i container portabili ed installabili su qualsiasi macchina senza il rischio di errori di configurazione.

Dalla seguente immagine è possibile capire l'interazione tra client e daemon di Docker ed intuire il percorso di creazione ed esecuzione di un container.

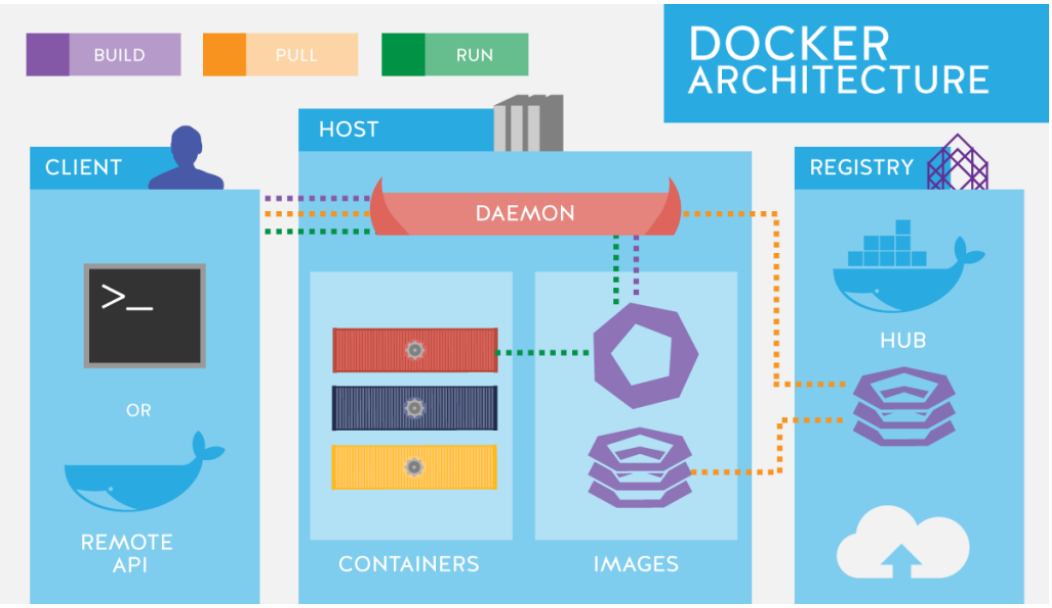



**Figura 8 – Architettura di docker [11]**

Nel seguito andremo ad analizzare in maniera più specifica le componenti che formano il software.

### 2.2.1 Immagini stratificate e UFS(Union File System)

Come abbiamo già accennato in precedenza, le immagini Docker sono delle viste di sola lettura necessarie alla creazione di un container. Volendo fare un paragone object-oriented è come se le immagini fossero le classi e i container oggetti, istanze delle classi.

Le immagini di Docker [12] sono dette stratificate: esse sono composte da più layer, ognuno dei quali rappresenta una porzione del file system dell'immagine stessa che aggiorna o sostituisce lo strato precedente, gli stessi layer sono immagini. Ad esempio un primo layer di un immagine può essere composto da Xubuntu mentre il layer sovrastante può essere composto dal web server Apache.

Quando viene creato un container, si aggiunge un ulteriore layer all'immagine: uno strato per la lettura-scrittura che consente la modifica di quest'ultima. Docker nella creazione dei container utilizza l'Union File System, un meccanismo che consente di vedere i vari file system stratificati come un unico grande FS. È proprio questa la caratteristica che rende Docker e la virtualizzazione basata su container una scelta saggia per la costituzione di applicazioni distribuite. Quando un container e quindi un immagine deve essere aggiornata, basta semplicemente aggiungere il nuovo layer mantenendo intatta la struttura sottostante, di conseguenza in applicazioni distribuite questo si traduce in un minor quantitativo di dati da trasmettere/ricevere ad ogni aggiornamento. Inoltre se due container condividono gran parte degli strati di un immagine, questi verranno creati a partire dalla stessa vista ma aggiungendo i layer che necessitano entrambi.

La stratificazione e l'union file system rendono i container molto più leggeri di un sistema virtualizzato in maniera classica.

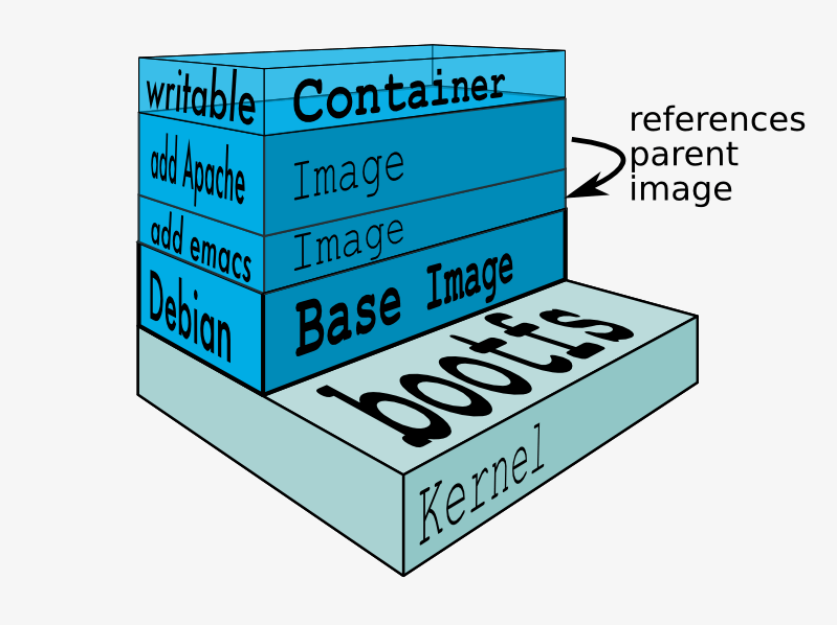



**Figura 9**
**Layers di un immagine Docker : è possibile notare l'immagine base, strati aggiuntivi come Emacs ed Apache ed infine lo strato "scrivibile" che viene aggiunto alla creazione del container.**

### 2.2.2 Costruzione di un Immagine e Dockerfile

Ogni volta che si vuole costruire un'immagine Docker, quest'ultimo esegue un insieme di istruzioni ben precise.

Cosi come per costruire un software scritto in C si fa uso del makefile per le direttive di compilazione e di linking, Docker fa uso di un file analogo, il Dockerfile, che contiene tutte le istruzioni utili alla costruzione dell'immagine desiderata.

Prima di vedere come è strutturato un dockerfile, osserviamo quali sono le best practices, [12] ossia delle linee guida che l'azienda fornisce per la scrittura dei dockerfile:

**Container Effimeri** : i container prodotti con l'immagine generata dal dockerfile devono esse effimeri nel senso che essi possono essere distrutti, ricostruiti e sostituiti utilizzando il minimo sforzo di configurazione. Inoltre quando un container termina non dovrebbero esserci conseguenze sul sistema su cui è implementato.

**Utilizzo di un file .dockignore** : è buona norma scrivere il dockerfile in una directory vuota per poter inserire nell'immagine che ne verrà solo i file strettamente necessari. In altri casi è possibile usare un file .dockignore che possa indicare quali file o directory, docker non dove considerare al momento della build.

**Evitare di installare pacchetti non necessari** : al fine di risparmiare storage, risorse e complessità computazionale è doveroso non installare pacchetti che non sono necessari al fine per cui l'immagine è ideata.

**Ogni container deve avere un compito preciso** : "smistare" un' applicazione in più container e creare una sorta di separazione degli interessi aiuta a gestire la portabilità del sistema oltre a garantire un miglior riutilizzo dei suddetti container. Il principio di mantenere alta coesione e basso accoppiamento, continua a valere per lo sviluppo di software tramite container, cosi come nel classico sviluppo dettato dall'ingegneria del software.

**Minimizzare il numero di strati** : utilizzare un gran numero di strati rende si il container completo, ma anche difficile da manutenere , bisogna quindi stare attenti e studiare a priori il numero di layer che l'immagine dovrà contenere.

Ora che sappiamo quali sono le buone norme da tenere a mente per la stesura di un dockerfile, osserviamo la struttura di quest'ultimo:

**FROM** – specifica quale è l'immagine base da cui partire.

**RUN** – specifica il comando da eseguire, ogni comando RUN eseguirà la sua istruzione sull'ultimo strato dell'immagine, ossia quello scrivibile che viene aggiunto alla creazione del container. È possibile aggregare insieme più comandi attraverso l'utilizzo dell'operatore "&&".

**ADD** – Copia i file necessari alla creazione dell'immagine dal file system dell'host o dalla rete.

**CMD** – definisce il comando predefinito da eseguire una volta che il container è stato creato. Tale comando, che può essere sovrascritto a runtime, non viene eseguito durante il processo di costruzione dell'immagine, inoltre è possibile definirne solo uno in ogni dockerfile.

Una volta scritto il dockerfile, è possibile eseguire il comando docker build dall'interfaccia a linea di comando di Docker nella stessa directory di quest'ultimo per poter creare l'immagine, come avviene più o meno con il comando make ed il makefile.

### 2.2.3 Docker Swarm

Docker swarm [14] è uno strumento di clustering e raggruppamento, che il software mette a disposizione degli utenti, per gestire insiemi di container su diversi host e considerarli come un unico sistema virtuale.

Il clustering è un fattore molto importante per l'utilizzo dei container, infatti considerare un sistema formato da molti di essi può aiutare a risolvere problemi di interruzione dei servizi grazie alla ridondanza creata proprio dalla cooperazione dei container.

L'utente può gestire lo swarm (sciame) attraverso uno swarm manager che dirige e schedula i container che ne fanno parte. Lo swarm manager permette di creare una prima istanza del gestore più delle copie che potrebbero servire nel caso in cui il primario cessi di funzionare.

Un altro vantaggio nell'utilizzo dello swarm è che l'engine di Docker, auto bilancia le

risorse utilizzate dai container che ne fanno parte, facendo in modo che gli host non vadano sotto sforzo compromettendo il regolare funzionamento del sistema.

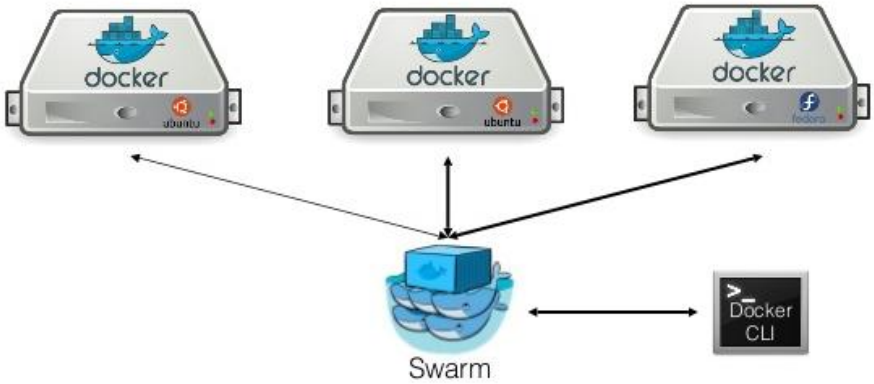



**Figura 10 - Struttura organizzativa e gestione di un Docker Swarm**

# Capitolo 3: Docker in pratica:esempio e comandi di gestione

Dopo aver analizzato il software open source Docker in teoria, è tempo di "toccare con mano" l'ambiente e provare a gestire le immagini ed i container che possono essere costruiti a partire da quest'ultime.

Analizzeremo prima i comandi di gestione per le immagini ed i container, per poi passare ad un esempio pratico in cui viene utilizzata la virtualizzazione container-based per realizzare un web server Apache.

Per prima cosa partiamo dall'installazione su Ubuntu di Docker.

## 3.1 Installazione

Docker è installabile su qualsiasi tipo di sistema operativo, quindi sia su Windows, su Mac e su Linux. Nei primi 2 casi, essendo comunque un software ideato per Linux, Docker crea una macchina virtuale con una distro di Linux per girare.

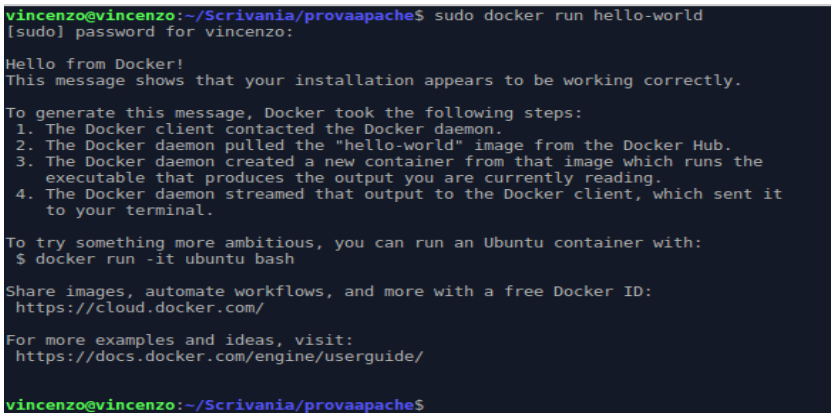
Per installare Docker su Ubuntu [15] o qualsiasi altra distro di Linux (io ad esempio ho utilizzato Xubuntu), bisogna prima scaricare la repository di Docker per poi, dal terminale utilizzare questo comando per l'installazione vera e propria:

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

a questo punto, per verificare la corretta installazione di Docker, è possibile creare il primo container, il più semplice possibile, che scrive a schermo "hello world" tramite il comando:

```
sudo docker run hello-world
```

Se il risultato del comando è il seguente, allora l'installazione è andata a buon fine.




## 3.2 Comandi per la gestione delle immagini

Ogni volta che vogliamo costruire un container dobbiamo prima partire dall'immagine su cui vogliamo costruirlo. Come visto in precedenza le immagini possono essere costruite partendo da un' immagine base (un sistema operativo ad esempio) ed aggiungendo layer aggiuntivi come Apache o altre applicazioni. Il tutto può essere fatto in due modi: si può scegliere di utilizzare un Dockerfile ed utilizzare il comando **DOCKER BUILD** per costruire un' immagine seguendo le istruzioni di quest'ultimo oppure usare direttamente il comando **RUN** per creare immediatamente un container partendo da un immagine scaricata dal Docker hub in automatico.

Nel primo caso il comando **DOCKER BUILD** [15], che può essere eseguito nella stessa directory dove è presente il dockerfile, è cosi strutturato:

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

al posto di options è possible usare tag come "--m" che permette di scegliere il limite di memoria da assegnare ad un container creato a partire da questa immagine.

I tag path e url vanno usati se il comando viene eseguito in una directory diversa da quella del dockerfile: il primo prevede che il dockerfile sia presente sulla stessa macchina dove è eseguito il comando, in questo caso occorre inserire il percorso del dockerfile. L'url del dockerfile và inserito nel momento in cui si vuole costruire una immagine a partire da un dockerfile presente su un altro host.

Il comando **DOCKER RUN** [16], invece, permette di creare ed avviare un container a partire da un' immagine costruita in precedenza oppure, se non presente sull'host, scaricata dal docker hub.

```
$ docker run [OPTIONS] IMAGE[:TAG|@DIGEST] [COMMAND] [ARG...]
```

Al posto di options è possible utilizzare alcuni tag tra cui "--name" che seguito da una stringa dà la possibilità di assegnare un nome al container, in alternativa docker assegna un nome a caso. Un'altra opzione interessante è il tag "-d", *detached*, che permette di creare un container che eseguirà in background.

Il flag IMAGE permette di scegliere l'immagine da cui partire per la creazione del container utilizzando il nome o l' ID fornito da Docker. Il tag invece riguarda la versione dell'immagine scelta che di default è *latest*. COMMAND identifica il comando che il container deve eseguire una volta partito.

Altri comandi per la gestione delle immagini sono **DOCKER IMAGES** che permette di osservare quelle presenti sull'hard disk con tag, ID e dimensione.

Con il comando **DOCKER IMAGE RM** è possibile rimuovere un immagine dall'host usando il suo nome o parte dell'ID.

```
vincenzo@vincenzo:~/Scrivania/provaapache$ sudo docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
immagineapache      latest              01597ec222a5        2 hours ago         177 MB
httpd               2.4                 278cd55ca6c5        2 weeks ago         177 MB
debian              wheezy              69e388a5985c        2 weeks ago         85.3 MB
ubuntu              latest              0ef2e08ed3fa        6 weeks ago         130 MB
hello-world         latest              48b5124b2768        2 months ago        1.84 kB
vincenzo@vincenzo:~/Scrivania/provaapache$ sudo docker image rm 0ef2e08
Untagged: ubuntu:latest
Untagged: ubuntu@sha256:dd7808d8792c9841d0b460122f1acf0a2dd1f56404f8d1e56298048885e45535
Deleted: sha256:0ef2e08ed3fabfc44002ccb846c4f2416a2135affc3ce39538834059606f32dd
Deleted: sha256:0d58a35162057295d273c5fb8b7e26124a31588cdadad125f4bce63b638dddb5
Deleted: sha256:cb7f997e049c07cdd872b8354052c8084999937645f6164912c4126015df036cc
Deleted: sha256:fcb4581c4f016b2e9761f8f69239433e1e123d6f5234ca9c30c33eba698487cc
Deleted: sha256:b53cd3273b78f7f9e7059231fe0a7ed52e0f8e3657363eb015c61b2a6942af87
Deleted: sha256:745f5be9952c1a22dd4225ed6c8d7b760fe0d3583efd52f91992463b53f7aea3
vincenzo@vincenzo:~/Scrivania/provaapache$
```

## 3.3 Comandi per la gestione dei container

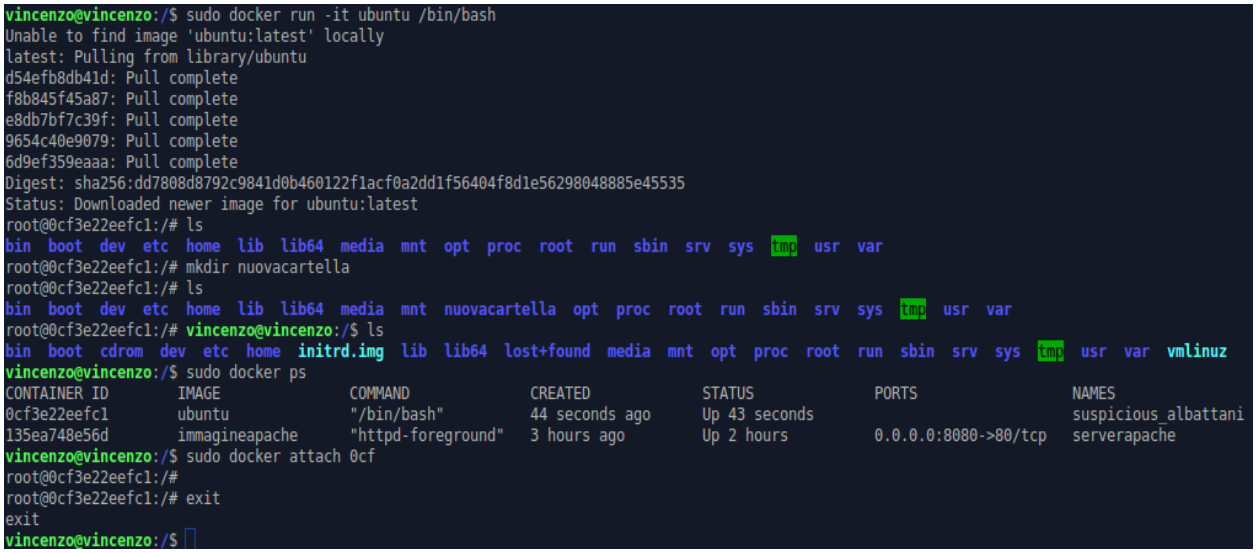
Come abbiamo visto nel paragrafo precedente, una volta costruita un' immagine, è possibile, tramite il comando **DOCKER RUN**, poter creare un container.

Una volta creato il container esso può essere avviato, stoppato, rimosso e monitorato.

Per avviare un container già esistente è possibile usare il comando **DOCKER START**

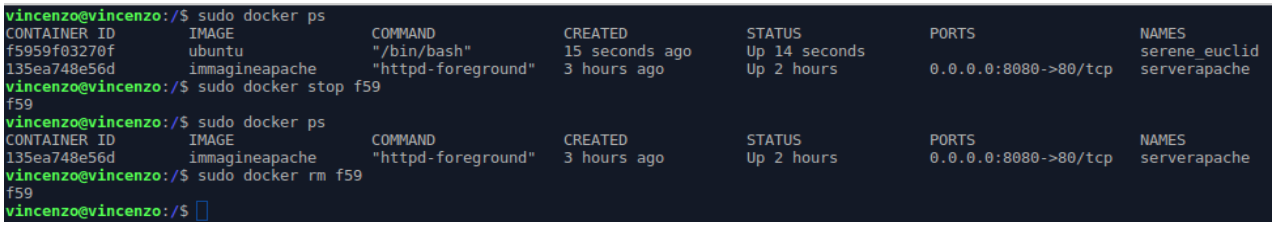
**`docker start [OPTIONS] CONTAINER [CONTAINER...]`**

Una volta che il container è in esecuzione, ad esempio se ne usiamo uno che è basato su Ubuntu e ci permette di usare la shell, possiamo uscire dalla modalità interattiva senza che esso termini utilizzando la combinazione di tasti **CTRL + Q + P**. Per poi rientrare in modalità interattiva è possibile usare il comando **DOCKER ATTACH** seguito dal nome o dall'ID del container in questione.




Dall' immagine, oltre a visualizzare i comandi descritti in precedenza, è possibile notare la struttura a strati dell'immagine Ubuntu e l'isolamento dei file system del container con Ubuntu e del FS dell'host.

Un container in esecuzione può essere interrotto senza essere eliminato attraverso il comando **DOCKER STOP** oppure completamente rimosso attraverso il comando **DOCKER RM**, eventualmente seguito dall'opzione "-f " se il container è in esecuzione.




Come detto in precedenza, è possibile attivare un container in modalità background attraverso l'opzione *detatch*. In questi casi può essere molto utile il comando **DOCKER LOGS** che permette di tener traccia di qualsiasi operato di un container.

```
vincenzo@vincenzo:/$ sudo docker run -d --name forzanapoli hello-world
af82b4c672891c58ed906060c7236666067d770efb2cfd4e03f753382569d60e
vincenzo@vincenzo:/$ sudo docker logs forzanapoli

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/

vincenzo@vincenzo:/$
```

Per conoscere i container in esecuzione è possibile utilizzare il comando **DOCKER PS** oppure **DOCKER PS –a** che mostra tutti i container, anche quelli che non stanno eseguendo.

```
vincenzo@vincenzo:/$ sudo docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              PORTS                  NAMES
135ea748e56d        immagineapache      "httpd-foreground"  3 hours ago         Up 2 hours          0.0.0.0:8080->80/tcp   serverapache
vincenzo@vincenzo:/$ sudo docker ps -a
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS                      PORTS                  NAMES
af82b4c67289        hello-world         "/hello"            44 seconds ago      Exited (0) 43 seconds ago                          forzanapoli
9266d37c524d        ubuntu              "/bin/bash"         3 minutes ago       Exited (0) 3 minutes ago                           gracious_aryabhata
c2cf9f0221fe        hello-world         "/hello"            4 minutes ago       Exited (0) 4 minutes ago                           ciaomondo
0cf3e22eefc1        ubuntu              "/bin/bash"         6 minutes ago       Exited (0) 5 minutes ago                           suspicious_albattani
135ea748e56d        immagineapache      "httpd-foreground"  3 hours ago         Up 2 hours                  0.0.0.0:8080->80/tcp   serverapache
vincenzo@vincenzo:/$
```

Un altro comando molto importante che permette di monitorare le risorse utilizzate dai container in esecuzione è il comando **DOCKER STATS** seguito dal nome o da parte dell'ID del container che si vuole analizzare.

```
CONTAINER           CPU %               MEM USAGE / LIMIT     MEM %               NET I/O             BLOCK I/O           PIDS
serverapache        0.08%               7.812 MiB / 3.184 GiB   0.24%               30.8 kB / 25.7 kB   3.15 MB / 0 B       0
```

Per altri comandi di gestione di container e immagini è possibile utilizzare il comando **DOCKER HELP** all'interno del terminale oppure far riferimento alla documentazione di Docker all'indirizzo https://docs.docker.com.

## 3.4 Esempio pratico : realizzazione di un webserver

Ora che abbiamo analizzato nel dettaglio il funzionamento di Docker, osservando i comandi principali per la gestione delle immagini e dei container, possiamo provare a

costruire un container basato sul web server Apache [17] per riuscire a percepire ancora meglio la potenza e la semplicità che Docker fornisce nella creazione di applicazioni distribuite.

Per prima cosa partiamo dal dockerfile che utilizzeremo per creare l'immagine su cui si baserà il nostro container:

***FROM httpd:2.4***

***COPY ./public-html/ /usr/local/apache2/htdocs/***

In pratica, con questo dockerfile costruiamo un' immagine basata sulla versione 2.4 dell'immagine httpd (Apache) che, se non è presente sull'hard disk verrà scaricata dal docker hub. Inoltre, nella stessa directory del dockerfile è presente una cartella *public-html* al cui interno sono presenti tutte le pagine .html (nel nostro caso solo un file index.html). Il dockerfile quindi, oltre a fornire l'immagine base, comunica che il primo comando da effettuare è una copia della directory public-html nel path */usr/local/apache2/htdocs/*.

Per costruire l'immagine utilizziamo il comando **DOCKER BUILD** così formato:

```
sudo docker build –t immagineapache .
```

che eseguita nello stessa directory del dockerfile non fa altro che eseguire le istruzioni presenti in esso creando un' immagine di nome immagine apache. L'opzione –t serve a creare un nuovo tag per l'immagine.

Una volta costruita l'immagine è possibile, tramite il comando **DOCKER RUN,** creare un container a partire da quest'ultima.

```
sudo docker run –dit --name serverapache –p 8080:80 immagineapache
```

Il risultato di questo comando è la creazione e l'avvio di un container serverapache basato sull'immagine prima costruita. I flag –i e –t fanno si che i canali standard del container vengano rediretti, dando cosi l'impressione di essere fisicamente all'interno dello stesso, mentre il flag –d serve ad eseguire il container in background. Come è possibile notare, i tre flag sono sinteticamente scritti con –dit.

Il flag –p serve ad impostare un numero di porta tcp che consenta l'accesso al server, in questo caso è stata scelta la porta 8080 che reindirizza alla porta tcp 80 designata per l'http.

Il risultato delle operazioni di build e run è il seguente.

```
vincenzo@vincenzo:~/Scrivania/provaapache$ sudo docker build -t immagineapache .
Sending build context to Docker daemon 4.608 kB
Step 1/2 : FROM httpd:2.4
 ---> 278cd55ca6c5
Step 2/2 : COPY ./public-html/ /usr/local/apache2/htdocs/
 ---> 40a5618b8413
Removing intermediate container d5d2f40bc765
Successfully built 40a5618b8413
vincenzo@vincenzo:~/Scrivania/provaapache$ sudo docker run -dit --name serverapache -p 8080:80 immagineapache
528b695ba55faf49a594f645914a06217c57cc947c30e6723cfc6d43a8d8689b
vincenzo@vincenzo:~/Scrivania/provaapache$
```

Sapendo che l'indirizzo dell'host su cui Docker è presente è 192.168.0.5, configurando il NAT del router ed aprendo la porta tcp 8080 per l'indirizzo precedente è possibile attraverso l'indirizzo IP pubblico della mia rete domestica, accedere al web server appena creato. Infatti, provando con uno smartphone connesso alla rete 4G, ad effettuare l'accesso all'indirizzo 151.77.76.136:8080, che rappresenta l'indirizzo IP pubblico seguito dalla porta tcp del web server, il risultato è il seguente.

# Capitolo 4: Le reti in Docker e Cloud Computing

## 4.1 Reti di container

In precedenza abbiamo parlato di una delle caratteristiche fondamentali che contraddistingue la virtualizzazione container-based: l'isolamento. Grazie ai namespace kernel è possibile realizzare container e quindi sistemi che sono indipendenti l'uno dall'altro. È possibile però fare in modo che container diversi possano comunicare tra loro attraverso delle reti ad hoc.

Quando il Docker engine viene installato sull'host, esso crea automaticamente tre diverse reti: la rete bridge, la rete host e la rete none.[18]

La **rete bridge** rappresenta l'interfaccia *docker0* che viene creata all'installazione di Docker. È la modalità di rete predefinita per ogni container se non viene specificata una particolare rete di appartenenza. Tutti i container collegati ad essa sono NATtati attraverso l'interfaccia *docker0* ed è come se fossero tutti connessi ad una stessa rete LAN.

La **rete host** rappresenta invece lo stack di rete della macchina fisica su cui è installato il Docker engine, infatti tutti i container connessi a quest'ultima condividono lo stack di rete con l'host.

La **rete none,** invece, è una finta rete in cui tutti i container collegati ad essa sono sprovvisti di interfaccia di rete.

Per poter osservare la lista delle reti presenti in docker è possibile usare il comando

```
docker network ls
```

Il cui risultato è il seguente:

```
vincenzo@vincenzo:~/Scrivania/provaapache$ docker network ls
NETWORK ID          NAME                DRIVER              SCOPE
02d1d1e99975        bridge              bridge              local
b1a263708589        host                host                local
7c3c2ddaf7c6        miarete             bridge              local
728ff1f28825        miarete2            bridge              local
37d2ed1e581a        none                null                local
vincenzo@vincenzo:~/Scrivania/provaapache$
```

Un container può essere connesso ad una qualsiasi rete attraverso il tag --net seguito dal nome della rete nel comando **DOCKER RUN**, ad esempio

**`sudo docker run -it --name c1 -net miarete ubuntu /bin/bash`**

In questo caso stiamo connettendo il container c1 basato su Ubuntu alla rete "miarete"

Per conoscere le informazioni di una particolare rete, indirizzi IP, container collegati ad essa ecc. si usa il comando **DOCKER NETWORK INSPECT**. Ad esempio se voglio conoscere le informazioni della rete "miarete":

```
vincenzo@vincenzo:~/Scrivania/provaapache$ docker network inspect miarete
[
    {
        "Name": "miarete",
        "Id": "7c3c2ddaf7c64c81d850fedbba20aae1e60ad98bea0cafb74b232717e57ee477",
        "Created": "2017-04-12T12:30:32.768533241+02:00",
        "Scope": "local",
        "Driver": "bridge",
        "EnableIPv6": false,
        "IPAM": {
            "Driver": "default",
            "Options": {},
            "Config": [
                {
                    "Subnet": "172.18.0.0/16",
                    "Gateway": "172.18.0.1"
                }
            ]
        },
        "Internal": false,
        "Attachable": false,
        "Containers": {
            "3f87b9e95e56d074250b14f770c1a8462e78aa066bb1d049cefe9940f267c16b": {
                "Name": "MDL",
                "EndpointID": "ade8b8de7a2e4d6d83594093aa19ad7b185e89f53859e5495d58cd32cdea3d4f",
                "MacAddress": "02:42:ac:12:00:02",
                "IPv4Address": "172.18.0.2/16",
                "IPv6Address": ""
            },
            "7a7bab4845049e7688f6e9f959e9e53fa4b28d6a01c9cd0c94e5e008655a467a": {
                "Name": "FRESA",
                "EndpointID": "4a26abc6980691503bbd46c4c0d83c33a346b168c0bb80577f6e607e46f67806",
                "MacAddress": "02:42:ac:12:00:03",
                "IPv4Address": "172.18.0.3/16",
                "IPv6Address": ""
            }
        },
        "Options": {},
        "Labels": {}
    }
]
vincenzo@vincenzo:~/Scrivania/provaapache$
```

In particolare, vengono mostrate informazioni riguardanti il tipo di rete, la subnet mask, il gateway e tutti i container ad essa collegati con i rispettivi indirizzi IP e MAC virtuali.

Gli indirizzi IP vengono assegnati in maniera automatica ai container in base alla disponibilità ma restano statici.

In questa rete ad esempio, essendo una rete bridge, i container che ne fanno parte possono comunicare tra loro come se fossero in una semplice rete LAN a patto che conoscano i loro indirizzi IP.

Come abbiamo avuto modo di vedere già nell'esempio precedente riguardante il web server, un container, oltre ad avere un suo indirizzo IP possiede anche una mappatura di porte che possono essere definite attraverso il flag –p seguito dal numero di porta( TCP o UDP ) o dal range di porte che esso dovrà avere.

Per definire una sola porta basta usare ad esempio `..-p 8080:80/TCP..` in questo caso definiamo una porta 8080 TCP che reindirizza alla porta 80. Se invece vogliamo assegnare un range di porte al container usiamo il flag `..-p 10000:20000/UDP..`

per dichiarare che quel container ha mappate l'insieme di porte UDP che vanno dalla 10000 alla 20000.

La mappatura delle porte dei container è visibile attraverso il comando **DOCKER PS**.

## 4.2 Reti User-defined

Per isolare meglio i container che ne fanno parte è possibile creare una rete diversa da quelle che Docker fornisce di default, in particolare si parla di reti definite dall'utente. Tali reti possono essere basate su alcuni driver di rete che Docker mette a disposizione, infatti, possono essere create reti con driver bridge o driver overlay. Oltre a questi due driver è possibile anche scriverne uno proprio in modo tale da personalizzare ancora di più le funzionalità e la gestione della rete.

Una volta creata una propria rete è possibile collegarci tutti i container che si vogliono, inoltre, un container può appartenere anche a più di una rete. Questo comporta che due container appartenenti a reti diverse non possono comunicare tra loro, infatti i container possono entrare in contatto solo se facenti parte dello stesso network.

Il Docker engine gestisce un server DNS integrato che fornisce ai container connessi alle reti user-defined un servizio di ricerca automatica, infatti, le richieste di risoluzione dei domini vengono tutte gestite in primis dal DNS integrato in Docker. Se quest'ultimo non è in grado di risolvere un dominio, la richiesta viene inoltrata ad un qualsiasi DNS esterno, in questo caso il compito del DNS integrato è quello di risolvere il dominio del container che ha inoltrato la richiesta per permettergli di ricevere l'indirizzo legato ad un particolare nome.

### 4.2.1 Bridge driver

Una rete bridge è il tipo di network più semplice da creare ed utilizzare. Per realizzarne una con il nome "miarete" è suffciente utilizzare il comando

```
sudo docker network create --driver=bridge miarete
```

Di conseguenza, per avere informazioni su tale rete e quindi conoscere l'indirizzo del

gateway, la subnet mask e le informazioni riguardo i container ad essa collegati basta usare il comando **DOCKER NETWORK INSPECT:**

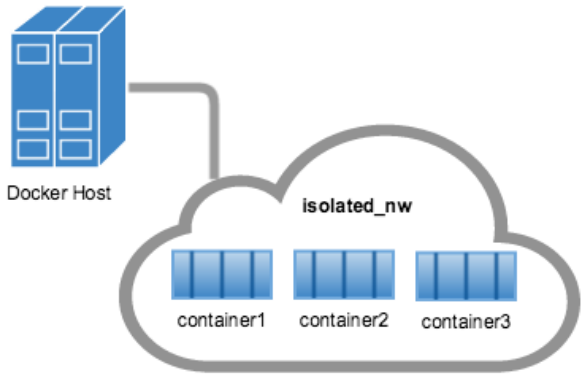
```
sudo docker network inspect miarete
```

Tutte le reti definite dall'utente attraverso il comando **CREATE** precedente, utilizzano il driver bridge che Docker fornisce di default e, inoltre, sono tutte simili, in termini di struttura, alla rete bridge predefinita ossia la *docker0*.

Dopo che la rete è stata creata, è possibile creare un qualsiasi container e collegarlo ad essa attraverso il tag "--net" seguito dal nome della rete, ad esempio:

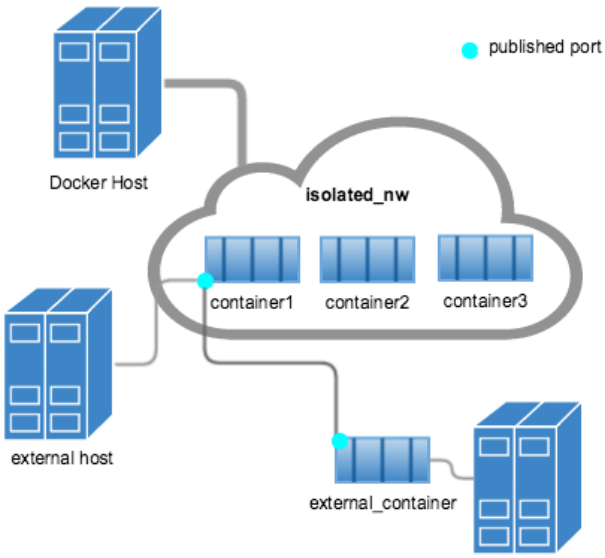
```
sudo docker run --net=miarete...
```

L'unico limite che una rete user-defined, basata sul driver bridge, impone è che tutti i container che fanno parte di essa devono risiedere sulla stessa macchina fisica(host). Una volta che un container viene connesso alla rete può immediatamente comunicare con gli altri connessi ad essa, inoltre è la rete stessa che isola i container che ne fanno parte rispetto alle reti esterne. Dalla seguente immagine possiamo comprendere ancora meglio gli aspetti descritti finora.[19]




Per esporre e rendere pubblici i container di una rete bridge user-defined, è necessario mappare delle porte per ognuno di essi così che sia possibile accedervi anche da remoto.

Dalla seguente immagine notiamo come è possibile, a partire dalla rete riportata nella precedente figura, permettere a container remoti, o semplicemente ad host remoti, di accedere al nostro container attraverso una porta pubblica, cosa che abbiamo già avuto modo di testare attraverso l'esempio del web server del capitolo 3.

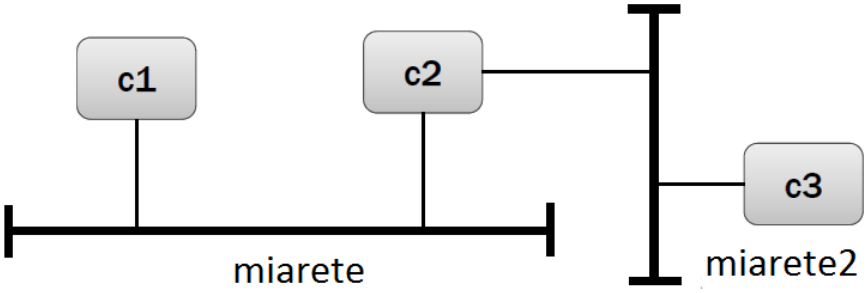



Un altro comando molto comodo è **DOCKER CONNECT**, esso infatti permette di collegare un container ad una qualsiasi rete anche dopo la sua creazione.

```
sudo docker connect miarete cont1
```

In questo modo, il container "cont1", già esistente, verrà collegato alla rete "miarete" e gli verrà assegnato un indirizzo IP.

Di seguito verrà riportato un esempio in cui sono presenti 3 container: c1, c2 e c3 e 2 reti bridge definite da me: "miarete" e "miarete2". I container c1 e c2 sono collegati alla rete "miarete" mentre il container c3 è collegato alla rete "miarete2". In seguito tramite il comando CONNECT, il container c2 verrà connesso anche alla rete "miarete2" creando una struttura di rete del genere:




I 3 container sono tutti basati su Ubuntu ed al loro interno sono stati installati i pacchetti utili all'esecuzione del comando *ping*, cosi da poter mostrare come, una volta connesso ad entrambe le reti, il container c2 possa comunicare sia con c1 che con c3.

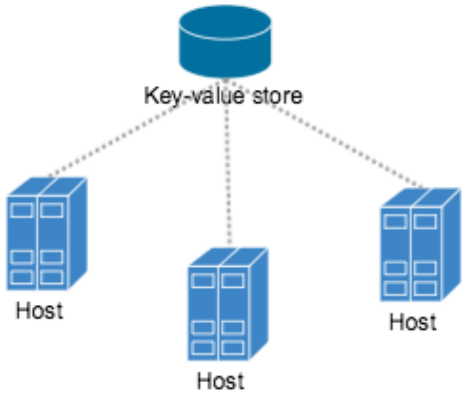
Nella pagina seguente è mostrato l'intero esempio svolto al terminale. Come già accennato, nel primo ping eseguito da c2 verso c3, i pacchetti non giungono a destinazione mentre nel secondo caso sì in quanto c2 viene connesso anche a miarete2.

```
vincenzo@vincenzo:~/Scrivania$ sudo docker network create --driver=bridge miarete
3df61c1458b9f339c530d86fcdad5b494dcb9fc9911f3605c1c208970e407637
vincenzo@vincenzo:~/Scrivania$ sudo docker network connect miarete c1
vincenzo@vincenzo:~/Scrivania$ sudo docker network connect miarete c2
vincenzo@vincenzo:~/Scrivania$ sudo docker network create --driver=bridge miarete2
df549cf87ffc535302acb13801020456eedecb95b7c9e845de0394341a685c0f
vincenzo@vincenzo:~/Scrivania$ sudo docker network connect miarete2 c3
vincenzo@vincenzo:~/Scrivania$ sudo docker attach c2
root@4633ec36dd11:/#
root@4633ec36dd11:/# ping c1
PING c1 (172.18.0.2) 56(84) bytes of data.
64 bytes from c1.miarete (172.18.0.2): icmp_seq=1 ttl=64 time=0.136 ms
64 bytes from c1.miarete (172.18.0.2): icmp_seq=2 ttl=64 time=0.087 ms
64 bytes from c1.miarete (172.18.0.2): icmp_seq=3 ttl=64 time=0.083 ms
^C
--- c1 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 1998ms
rtt min/avg/max/mdev = 0.083/0.102/0.136/0.024 ms
root@4633ec36dd11:/# ping c3
ping: unknown host c3
root@4633ec36dd11:/# ^Pvincenzo@vincenzo:~/Scrivania$
vincenzo@vincenzo:~/Scrivania$ sudo docker network connect miarete2 c2
vincenzo@vincenzo:~/Scrivania$ sudo docker attach c2
root@4633ec36dd11:/#
root@4633ec36dd11:/# ping c3
PING c3 (172.19.0.2) 56(84) bytes of data.
64 bytes from c3.miarete2 (172.19.0.2): icmp_seq=1 ttl=64 time=0.150 ms
64 bytes from c3.miarete2 (172.19.0.2): icmp_seq=2 ttl=64 time=0.094 ms
64 bytes from c3.miarete2 (172.19.0.2): icmp_seq=3 ttl=64 time=0.093 ms
64 bytes from c3.miarete2 (172.19.0.2): icmp_seq=4 ttl=64 time=0.098 ms
^C
--- c3 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 2997ms
rtt min/avg/max/mdev = 0.093/0.108/0.150/0.027 ms
root@4633ec36dd11:/# ping c1
PING c1 (172.18.0.2) 56(84) bytes of data.
64 bytes from c1.miarete (172.18.0.2): icmp_seq=1 ttl=64 time=0.097 ms
64 bytes from c1.miarete (172.18.0.2): icmp_seq=2 ttl=64 time=0.098 ms
64 bytes from c1.miarete (172.18.0.2): icmp_seq=3 ttl=64 time=0.096 ms
64 bytes from c1.miarete (172.18.0.2): icmp_seq=4 ttl=64 time=0.098 ms
64 bytes from c1.miarete (172.18.0.2): icmp_seq=5 ttl=64 time=0.100 ms
^C^C
--- c1 ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 3997ms
rtt min/avg/max/mdev = 0.096/0.097/0.100/0.012 ms
root@4633ec36dd11:/#
```

### 4.2.2 Overlay driver

Il driver per reti overlay permette di creare reti multi host nel senso che è possibile mettere in comunicazione diretta diversi host e di conseguenza anche tutti i container presenti al loro interno. [25]

Una rete basata su driver overlay necessita di un *key-value store service*, come zookeper, installato e configurato a priori per poter funzionare correttamente. Una rete overlay di primo livello si presenta come nell' immagine a destra.

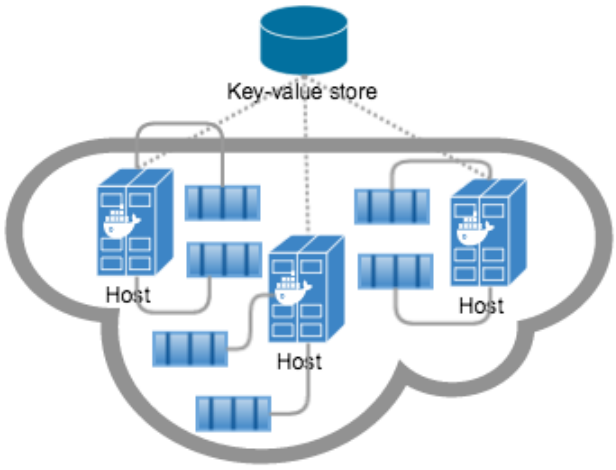



A partire da questo punto, su ogni host va installato ed avviato il Docker engine. Una volta eseguito Docker su ogni macchina è necessario aprire le porte 4789 TCP e 7946 TCP/UDP per ogni host.

A questo punto è possibile creare la rete overlay attraverso l'uso di Docker swarm che riunisce tutti gli host inquadrandoli come uno solo. La rete viene creata eseguendo su ogni host il comando

```
sudo docker network create –d overlay
```

Una volta costituita la rete, i container creati sull'host A potranno comunicare liberamente con i container dell'host B e dell'host C. La forma di una rete overlay finale è mostrata nella figura sottostante.



## 4.3 Cloud Computing

Il cloud computing [20] ( in italiano “nuvola informatica”) è un paradigma che permette l’erogazione di risorse informatiche, come l’elaborazione o l’archiviazione dei dati, caratterizzato dalla richiesta on-demand attraverso internet di particolari risorse già esistenti e configurabili. Tale paradigma è costituito di 5 caratteristiche essenziali che sono:

- **On-demand self-service**: Un utente che accede al cloud può dotarsi di funzionalità di calcolo o di una qualsiasi risorsa, come ad esempio l’orario del server o l’archiviazione in rete, di cui ha bisogno senza alcun intervento umano o interazione diretta con il fornitore di tali servizi, essi infatti sono lì, pronti ad essere usati.

- **Ampio accesso alla Rete**: I servizi sono tutti in rete e resi accessibili attraverso i meccanismi standard che ne promuovono l’uso da tutti i tipi di terminali in maniera adattiva. Le funzionalità quindi possono essere utilizzate da diversi tipi di client come un semplice PC, una workstation, tablet o smartphone.

- **Condivisione delle risorse**: Le risorse di calcolo del provider sono riunite per servire più consumatori contemporaneamente, utilizzando un modello *multi-tenant*, attraverso l’allocazione di diverse risorse fisiche e virtuali in maniera dinamica che vengono assegnate e riassegnate in base alle richieste dei consumatori. Ogni client utilizza le funzionalità in totale indipendenza, avendo la sensazione di essere l’unico a farlo. Il consumatore non può gestire direttamente le risorse ne tantomeno conosce la posizione esatta di esse, può però utilizzarle in base ad un’ area geografica. Le funzionalità infatti di uno stesso fornitore possono variare per ogni Paese o Stato.

- **Elasticità rapida**: Le funzionalità possono essere rilasciate in modo veloce, spesso anche automatico in modo di adattarsi al meglio alla domanda. Al client, le risorse appaiono illimitate, inoltre esso deve avere la percezione di potersene appropriare e di poterle utilizzare in qualsiasi momento senza problemi.

- **Servizio Misurato**: I sistemi cloud controllano ed ottimizzano automaticamente la gestione delle risorse in base al loro richiesta grazie all'utilizzo di unità di misura ad un certo livello di astrazione come ad esempio larghezza di banda, numero di utenti attivi, archiviazione ecc.

Dopo aver analizzato le principali caratteristiche che un sistema di cloud computing deve soddisfare, analizziamo ora le tre tipologie fondamentali di servizi di cloud computing:

- **Software as a Service (Saas)** : permette l'utilizzo di funzionalità e programmi, forniti da un provider, che sono installati su un server remoto attraverso un' interfaccia web. Alcuni esempi possono essere Dropbox, le Google apps come Gmail o Maps, GitHub o Wikipedia.

- **Platform as a Service (PaaS)** : Una o più funzionalità viene eseguita su una piattaforma software, ossia l'insieme dei programmi, librerie e risorse che una applicazione ha bisogno per essere eseguita. Tale struttura è tipica di alcune piattaforme utilizzate per lo sviluppo di altri software come Google App Engine, Microsoft Azure o Bluemix di IBM.

- **Infrastructure as a Service (IaaS)**: Come un SaaS, permette l'utilizzo di software in remoto inoltre, aggiunge la condivisione di hardware come risorse per l'archiviazione e il backup. La caratteristica fondamentale di questo servizio è che tali risorse hardware vengono allocate nel momento in cui la piattaforma ne ha bisogno, quindi al momento di una richiesta. Un importante servizio IaaS è Amazon Web Services.

Nell'immagine seguente è possibile osservare le differenze tra i tre tipi di servizi finora riportati e notare in ognuno di essi quali parti del sistema può gestire l'utente e quali parti gestisce invece il provider.

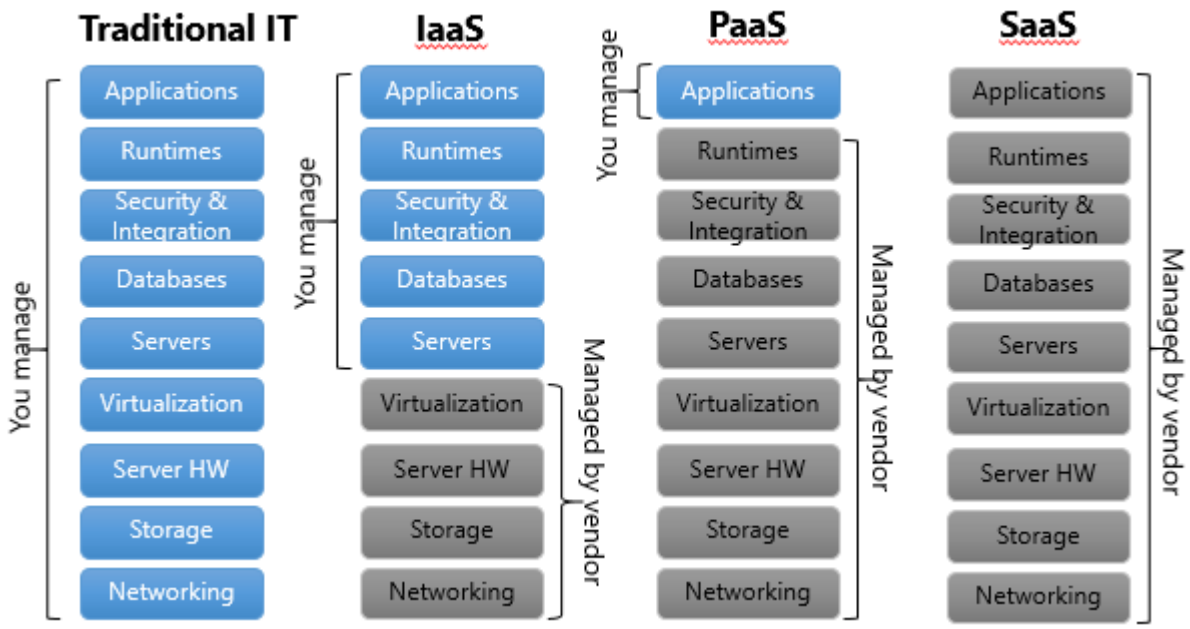



**Figura 11 - Differenze tra i vari servizi di Cloud Computing[21]**

### 4.3.1 Utilizzo di Docker nel Cloud Computing

Dalle analisi che abbiamo sviluppato nel paragrafo precedente, appare evidente che un sistema di cloud computing sia interamente basato sulla virtualizzazione. Come sappiamo, esistono due tipi di virtualizzazione: quella hypervisor-based e quella container-based. Alla nascita dei primi sistemi basati sul cloud veniva utilizzato il primo tipo di virtualizzazione per far si che su un unico server fossero presenti più applicativi. Nasceva però il problema di dover gestire un enorme overhead dovuto alla presenza di molte macchine virtuali ognuna contenete un diverso sistema operativo e tutta una serie di processi più o meno utili. Inoltre tali sistemi erano molto lenti nell'avvio a causa del boot che ogni virtual machine doveva eseguire per essere completamente operativa.

Dopo anni di utilizzo della virtualizzazione hypervisor-based, e di macchine virtuali di "riserva" che entravano in uso per ovviare ai problemi di quelle esistenti, si è giunti alla conclusione che l'unico metodo per rendere efficiente e performante un sistema di cloud computing era quello di utilizzare una virtualizzazione a livello di sistema operativo.

L’utilizzo dei container infatti agevola molto il sistema dal punto di vista della velocità e dell’utilizzo delle risorse. I container, come sappiamo, condividono il kernel dell’OS installato sull’host che li ospita e garantiscono un elevato livello di isolamento.

Dal punto di vista dell’avvio del sistema notiamo un incredibile miglioramento, infatti, un utente che utilizza un sistema cloud basato su container ha come l’impressione che i servizi che richiede on-demand siano presenti sulla sua macchina.

Nell’immagine sottostante possiamo notare la differenza nella velocità di risposta ad una richiesta di servizio in un sistema di cloud computing con i due tipi di virtualizzazione studiati.

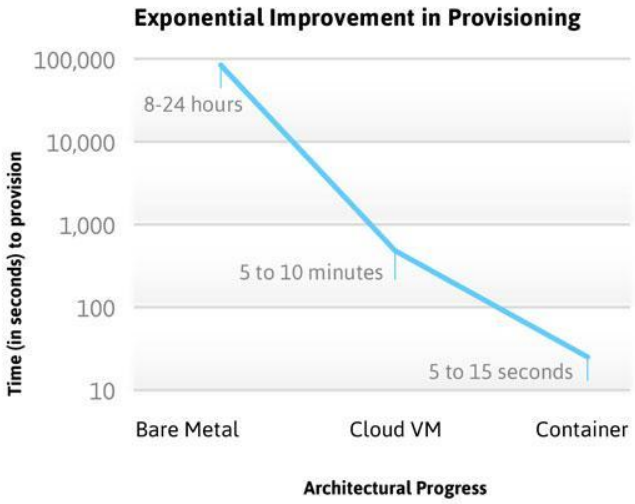



**Figura 12 – Confronto, in termini di velocità di risposta, tra sistemi cloud gestiti con VM e container[22]**

Come possiamo notare, si passa dai 5-10 minuti impiegati per rispondere ad una richiesta fatta ad un sistema basato su VM ai 5-15 secondi di un sistema container-based: un miglioramento notevole.

È stata Google con la sua Google Cloud Platform a dare una notevole importanza all’utilizzo dei container ed in particolare di Docker nei sistemi di cloud computing. GCP è un PaaS interamente basato e gestito da container, con tutti i vantaggi che ne derivano. Esso è un insieme di servizi che consentono di creare semplici siti web, applicazioni in cloud o sistemi molto

più complessi.

La GCP utilizza il Google Container Engine (GKE) per la gestione dei container Docker.

Esso è [23] un sistema di gestione ed orchestrazione per container Docker, un grande cluster manager basato su Kubernetes che, come abbiamo visto in precedenza, è un sistema di gestione open source di Google.

È possibile controllare GKE attraverso l'interfaccia a riga di comando (gcloud) o la console di Google Cloud Platform.

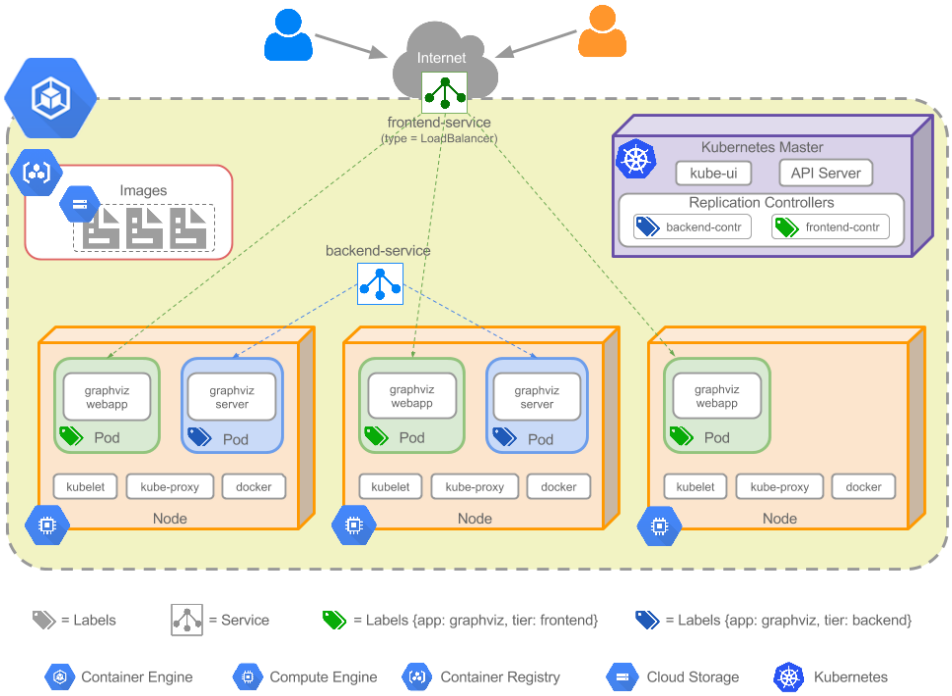



**Figura 13 - Struttura di un sistema Cloud gestito dal GKE[24]**

Kubernetes gestitsce i pod (insieme di container sullo stesso host) e i replication controllers. GKE permette invece la gestione di tutti i nodi e quindi dei container che ne fanno parte.

# Conclusioni

Lo scopo di questo elaborato è stato fornire una panoramica su quella che è la virtualizzazione a livello di sistema operativo: la virtualizzazione basata su container. Come abbiamo potuto vedere, questo tipo di virtualizzazione risulta essere molto più vantaggioso rispetto alla classica virtualizzazione con macchine virtuali in quanto non c'è più la necessità di astrarre un intero sistema. La leggerezza ed il minor overhead introdotto dalla virtualizzazione OS-oriented hanno spinto il mercato software ed in particolare quello del cloud computing ad utilizzare questa tecnologia per lo sviluppo di complessi sistemi. Le prime applicazioni basate su cloud, utilizzando la virtualizzazione hypervisor-based, risultavano lente nel rispondere e poco efficienti in termini di utilizzo delle risorse fisiche, oggi invece l'utilizzo dei container ha reso il cloud computing il paradigma prediletto da ogni software house per la creazione e la distribuzione dei propri sistemi ed applicazioni.

Oltre ad evidenziare gli enormi vantaggi che la virtualizzazione a livello di sistema operativo ha introdotto, si è voluto introdurre ed analizzare il software principe per la gestione dei container: Docker. Come abbiamo avuto modo di osservare, Docker prende tutto ciò che il kernel Linux mette a disposizione per la gestione dei container, come i kernel namespace e i Cgroups, ma ne semplifica la gestione grazie ad API di alto livello.

Grazie alla semplicità di utilizzo ed il largo impiego che il cloud sta avendo, è molto semplice ipotizzare l'espansione che Docker riuscirà ad avere negli anni a seguire.

# Bibliografia


[1] http://www.cloudvortex.com/cloud-hosting/web-applications/parallels/parallels-cloud-server

[2] https://linuxcontainers.org/it/lxc/introduction/

[3] http://www.hackthesec.co.in/2016/02/how-to-install-and-setup-lxc-linux-containers.html

[4] Namespaces in operation, part 1: namespaces overview By Michael Kerrisk January 4, 2013
https://lwn.net/Articles/531114/

[5] https://linuxaria.com/article/introduction-to-cgroups-the-linux-conrol-group?lang=it

[6]http://www.oracle.com/technetwork/articles/servers-storage-admin/resource-controllers-linux-1506602.html

[7] https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/

[8] https://meteorhacks.com/learn-kubernetes-the-future-of-the-cloud/

[9] Containers, DevOps, Apache Mesos and Cloud Reshaping how we develop and delivery software - Marcelo Sousa Ancelmo
http://events.linuxfoundation.org/sites/events/files/slides/cloud_devops_mesos.pdf

[10] Docker: Automated and Consistent Software Deployments by Abel Avram on Mar 27, 2013
https://www.infoq.com/news/2013/03/Docker

[11] API-Driven DevOps: Spotlight on Docker by Vassili van der Mersch on March 15, 2016
http://nordicapis.com/api-driven-devops-spotlight-on-docker/

[12] Docker Basics Webinar Q&A: Understanding Union Filesystems, Storage and Volumes, By Chris Hines October 1, 2015
https://blog.docker.com/2015/10/docker-basics-webinar-qa/
[13]Best practices for writing Dockerfiles
https://docs.docker.com/engine/userguide/eng-image/dockerfile_best-practices/
[14] Docker Swarm by Margaret Rouse
http://searchitoperations.techtarget.com/definition/Docker-Swarm
[15] https://docs.docker.com/engine/reference/commandline/build/#description
[16] https://docs.docker.com/engine/reference/run/
[17] https://hub.docker.com/_/httpd/
[18] Prof. Roberto Canonico,Cenni alle tecnologie di virtualizzazione basate su container. A cura di: Alessandro Amirante
http://wpage.unina.it/rcanonic/didattica/cdn/lucidi/CDN-L09-Virtualization-Containers.pdf
[19] Docker container networking
https://docs.docker.com/engine/userguide/networking/
[20] Peter Mell, Timothy Grance, The NIST Definition of Cloud Computing. NIST, Special Publication 800-145, Settembre 2011.
http://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-145.pdf
[21] http://www.mazikglobal.com/blog/cloud-computing-stack-saas-paas-iaas/
[22] Containers—Not Virtual Machines—Are the Future Cloud
Jun 17, 2013 By David Strauss
http://www.linuxjournal.com/content/containers%E2%80%94not-virtual-machines%E2%80%94are-future-cloud?page=0,0
[23] Google Container Engine (GKE) by Margaret Rouse
http://searchitoperations.techtarget.com/definition/Google-Container-Engine-GKE
[24]http://omerio.com/2016/01/02/getting-started-with-kubernetes-on-google-container-engine/
[25]http://docs.master.dockerproject.org/engine/userguide/networking/dockernetworks/