ALMA MATER STUDIORUM – UNIVERSITA' DI BOLOGNA
CAMPUS DI CESENA
SCUOLA DI SCIENZE

CORSO DI LAUREA IN INGEGNERIA E SCIENZE INFORMATICHE

TITOLO DELLA TESI:

Valutazione degli strumenti di virtualizzazione per costituire un servizio di macchine virtuali personalizzate a supporto della didattica

Tesi in Sistemi Operativi

Relatore
Vittorio Ghini

Presentato da
Federico Naldini

Sessione II Appello
Anno Accademico 2016/2017

Indice

Introduzione

Durante gli ultimi anni le tecniche di progettazione software si sono evolute molto rapidamente: nuovi modelli e architetture, che fanno la loro forza nel creare sistemi e applicazioni distribuiti su un numero sempre maggiore di macchine connesse tramite Internet, stanno soppiantando le vecchie soluzioni basate su una progettazione molto compatta e poco distribuita.
Tra gli elementi che hanno portato a questo cambio di rotta, si colloca la virtualizzazione, insieme di tecniche che consentono di mandare in esecuzione più sistemi operativi in contemporanea su uno stesso hardware fisico, mediante un'attenta condivisione di risorse; questa possibilità di avere più macchine operanti in maniera autonoma, senza dover fornire risorse fisiche a ciascuna, ha incentivato una progettazione orientata alla modularizzazione dei sistemi, fondamentale per tecnologie come i *web service* e il *cloud computing*.
Tuttavia le tecniche di virtualizzazione classiche sono diventate inefficienti molto rapidamente, dato che i nuovi pattern di progettazione, come ad esempio le architetture a servizi, richiedevano un tale livello di scalabilità per cui la quantità di risorse fisiche richieste dalla singola macchina virtuale risultava eccessiva per i compiti da svolgere.
Per riuscire a integrare i vantaggi forniti dalla virtualizzazione ai nuovi modelli di progettazione, è nato il concetto di *container*.
Questa tesi si pone come obbiettivo quello di delineare una panoramica sul mondo dei *containers* e sulle principali applicazioni di questa nuova forma di virtualizzazione, inoltre di descrivere minuziosamente lo studio di fattibilità, iniziato durante un'esperienza di tirocinio formativo nei primi mesi del 2017 e poi proseguito in tesi, su una soluzione a *container* per le macchine di laboratorio del corso di laurea di "Ingegneria e Scienze Informatiche".
Dapprima si tratterà brevemente della virtualizzazione e dell'impatto che essa ha avuto sull'evoluzione delle tecniche di progettazione software, esplorando poi il concetto di *container*, delle sue potenzialità e limiti e di quali siano le differenze con le classiche macchine virtuali.
Successivamente si introdurrà lo studio di fattibilità, delineandone in primo luogo le esigenze che hanno portato al suo svolgimento, partendo da una descrizione del sistema attuale in uso presso i laboratori del corso di laurea, per poi arrivare a delineare i requisiti di una soluzione alternativa che integrasse la tecnologia a *containers* e i conseguenti vantaggi.
Si descriveranno poi gli strumenti utilizzati durante lo studio, partendo da Docker, progetto open source per lo sviluppo e distribuzione di applicazioni in *containers*, di cui verranno presentate in modo approfondito architettura e funzionamento; successivamente verranno trattate le soluzione alternative a Docker: LXC, LXD e Windows-containers; inoltre si farà una panoramica sui software utilizzati per

integrare funzionalità alle applicazioni sviluppate su *containers* durante lo studio di fattibilità; infine si darà una rapida occhiata agli strumenti per l'integrazione tra *containers* e *cloud computing*, presentando le caratteristiche principali di architettura e funzionamento delle due piattaforme utilizzate durante lo studio(ProxMox e OpenStack).
Dopo aver concluso questa sezione, verrà trattato nel dettaglio lo svolgimento dello studio di fattibilità: il filo conduttore sarà dato dal percorso seguito per giungere agli obbiettivi fissati; al fine di agevolarne la trattazione, il percorso verrà diviso in tappe che comprenderanno le maggiori problematiche affrontate, gli strumenti impiegati, i tentativi effettuati e infine le soluzioni formulate; la prima parte di questo percorso verterà sulla realizzazione di un ambiente Desktop Linux con la tecnologia a *containers*, una seconda sulle alternative a Docker nel mondo dei *containers* e infine l'ultima tratterà delle prove realizzate in ambienti di virtualizzazione distribuiti, come ProxMox e OpenStack.
Infine nell'ultima sezione si descriveranno in maniera rapida i risultati ottenuti nel corso dello studio, analizzando successi e fallimenti; si rifletterà inoltre su dove indirizzare futuri approfondimenti che abbiano come partenza i risultati ottenuti nello svolgimento dello studio.

Capitolo 1

Scenario Applicativo

Il seguente capitolo si pone come scopo di analizzare l'evoluzione che hanno subito le tecniche di virtualizzazione, per arrivare a definire il concetto di *container* [1] e le sue potenzialità nell'ambito della progettazione software.
Fatto ciò, saranno trattate le esigenze che hanno spinto a valutare una soluzione a *containers* per la costruzione di un ambiente desktop da utilizzare durante le lezioni di laboratorio del corso "Sistemi Operativi", esaminando in seguito l'insieme di requisiti e obbiettivi posti per ottenere un sistema con prestazioni migliori di quello attualmente in uso.

1.1 Panoramica sui *containers*

Durante la storia dell'informatica, la parte software di un sistema ha assunto un ruolo sempre più fondamentale, sono ormai lontani i tempi in cui il software era solo un accessorio, talvolta valutato come superfluo, dell'hardware.
Oggi la componente software ha raggiunto un livello di potenza espressiva tale da poter perfino simulare un insieme di risorse hardware che forniscono le stesse funzionalità delle loro controparti reali, l'insieme delle tecniche volte a rendere possibile ciò è detto *virtualizzazione* [2].
La storia della *virtualizzazione* [3] è molto più lunga di quanto si possa pensare: i primi studi sono cominciati infatti durante gli anni sessanta del secolo scorso, studi con lo scopo di rendere più agevole la gestione di risorse su sistemi complessi; i vantaggi ottenuti sono stati tali da portare a uno sviluppo molto rapido di queste tecnologie, che nel giro di una ventina d'anni sono arrivate a poter consentire il funzionamento di interi sistemi operativi su insiemi di risorse astratte.
Agli inizi degli anni duemila, i moduli necessari alla *virtualizzazione*, detti *Hypervisor* [4], hanno iniziato a essere integrati nei maggiori sistemi operativi, aprendo così la strada a sistemi basati interamente su macchine virtuali, ovvero sistemi composti da diverse macchine virtuali a cui l'*Hypervisor* fornisce un insieme di risorse virtuali, le cui corrispondenti reali sono condivise con altre macchine virtuali.
I vantaggi portati dalle potenzialità della *virtualizzazione* hanno avuto un impatto considerevole sull'evoluzione delle tecnologie impiegate in ambito informatico: innanzitutto la possibilità di mettere in esecuzione più macchine virtuali sullo stesso insieme di risorse fisiche ha portato a una riduzione di quantità di hardware impiegato in un sistema a favore della sua potenza, hardware con una potenza maggiore consente di mettere in esecuzione più macchine rispetto a più hardware di qualità scadente riducendo inoltre le dimensioni fisiche; altro grande vantaggio

consiste nell' isolamento: ogni macchina virtuale viene messa in esecuzione in un contesto isolato dalle altre, in modo che qualsiasi problematica rimanga confinata alla macchina virtuale che l'ha generata, senza intaccare l'esecuzione delle altre macchine virtuali.
Ogni macchina virtuale, infine, non deve per forza disporre di tutte le risorse dell'hardware che utilizza, ma può avere accesso solo ad alcune componenti tra quelle disponibili, in modo da fornire a ciascuna macchina le risorse strettamente necessarie al suo funzionamento.
Negli ultimi anni tuttavia le soluzioni a macchine virtuali e più in generale derivate dalla virtualizzazione classica stanno iniziando a diventare meno adeguate, causa i cambiamenti subiti dall'industria software: se prima le applicazioni erano "monolitiche", ovvero composte da un insieme di file sorgenti, generati in seguito a un lungo ciclo di sviluppo, compilati e installati sulla singola macchina; ora si tende ad avere applicazioni dinamiche, le cui singole funzionalità sono scalabili su più nodi di un sistema distribuito; questi nodi si occupano dell'esecuzione di solo una parte dell'applicazione e comunicano tra di loro tramite rete internet, facendo sì che il carico affidato al singolo nodo sia minore dell'esecuzione dell'applicazione in toto e migliorando la portabilità del sistema, dovendo riprogrammare parti decisamente minori di sistema per garantire una compatibilità maggiore.
In questo nuovo contesto nasce il concetto di *container* come alternativa alle classiche forme di virtualizzazione, per integrare i vantaggi di un sistema virtualizzato con quelli di uno distribuito.
L'idea base è molto semplice: inserire l'applicazione in una "scatola" che le fornisca il suo contesto di esecuzione, predisponendo solamente le risorse necessarie, sia software che hardware, in modo da ridurre al minimo l'impatto sull'ambiente circostante, inoltre la "scatola" garantisce un isolamento all'applicazione molto simile a quello messo a disposizione dalle macchine virtuali.
Il funzionamento dei *container* è basato su una virtualizzazione a livello di sistema operativo, tecnica che prevede multipli *user-space* [5] isolati tra loro facenti riferimento a uno stesso *kernel-space* [6], in pratica ogni *container* viene eseguito in un diverso *user-space*, condividendo con gli altri *containers* le funzionalità offerte dal *kernel-space*: questa soluzione permette così di fornire un contesto isolato a ogni applicazione, facilmente sostituibile in caso di guasto, e di ridurre al minimo, a volte addirittura annullare, l'*overhead* dovuto alla virtualizzazione/emulazione di risorse.
Analizzando gli schemi di design di *container* e macchine virtuali basate su *Hypervisor*(figura 1.1), è possibile notare come nello schema di *Hypervisor* siano presenti due livelli aggiuntivi(*Hypervisor* e GuestOS), assenti invece nello schema *container*, questi due livelli sono la principale causa di *overhead* delle macchine virtuali: l'esecuzione di più sistemi operativi comporta un impiego notevole di risorse, seppur virtuali, che peggiora notevolmente le prestazioni di un sistema, inoltre nei vari sistemi operativi sono presenti tutta una serie di funzionalità che risultano inutili per l'esecuzione dell'applicazione, ma sono comunque istanziate in quanto parte del sistema operativo. Dall'altro lato, lo schema di design dei *container*s appare molto più snello, grazie alla condivisione del *kernel space* del sistema operativo reale: questo fa sì che ogni *container* acceda solamente alle risorse di cui ha bisogno, senza dover istanziare sistemi operativi non necessari.

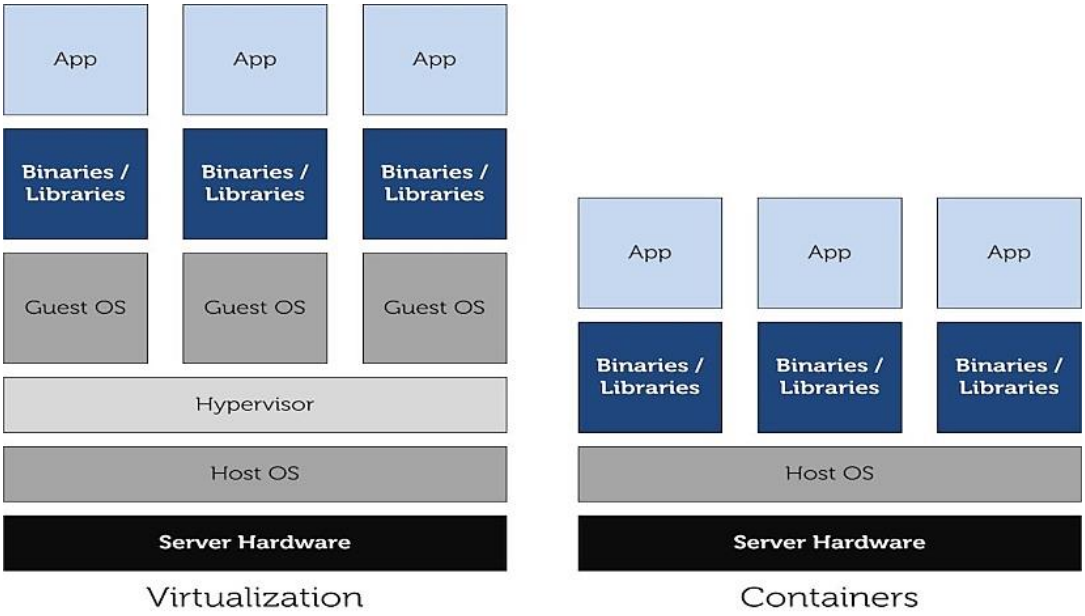

Figura 1.1 Architettura macchina virtuale e architettura containers (fonte: http://searchservervirtualization.techtarget.com/definition/container-based-virtualization-operating-system-level-virtualization)

E' però molto importante sottolineare che i *containers* non si pongono come alternativa alle forme di virtualizzazione più classica, ma piuttosto come loro integrazione, questo poiché la virtualizzazione a livello di sistema operativo rimane indissolubilmente legata al kernel su cui viene progettata: ad esempio un *container* creato utilizzando le primitive di un kernel Linux non potrà mai essere messo in esecuzione in ambiente Windows, se non utilizzando strumenti di virtualizzazione classica per creare un kernel Linux.
Per quanto riguarda la realizzazione pratica dei *containers*, le piattaforme che implementano questa virtualizzazione sono aumentate in maniera considerevole durante gli ultimi anni: in ambiente Linux sono presenti i *containers* LXC [7] e LXD [8], che sfruttano le funzionalità di isolamento del kernel Linux quali *cgroups* [9] e *namespaces* [10]; su *containers* LXC si basa Docker [11], piattaforma open source per automatizzare lo sviluppo di applicazioni inserite in *containers* [12] ; anche Microsoft negli ultimi anni ha sviluppato i propri *containers*, il cui supporto è integrato nel kernel di Windows 10 e sulle nuove versioni di Windows Server.

1.1 Esigenze che hanno portato allo studio di fattibilità

La ricerca svolta sui *containers* è partita da uno studio di fattibilità svolto nelle prime fasi come tirocinio, successivamente divenuto argomento di tesi.
La proposta è stata fatta dai laboratori DISI della sede del corso di laurea di "Ingegneria e Scienze Informatiche", il cui obbiettivo è fornire supporto sempre alla didattica del corso di laurea, migliorando continuamente gli strumenti a disposizione di insegnanti e studenti.

Proprio nel miglioramento degli strumenti di supporto alla didattica si collocano le esigenze che hanno portato all'idea di sviluppare un sistema a *containers* per la gestione delle macchine di laboratorio nel corso di Sistemi Operativi.
La soluzione attualmente in uso è intuitiva e lineare, infatti prevede un insieme di macchine virtuali (una per ogni studente iscritto al corso) che vengono memorizzate su un server e scaricate di volta in volta sulla macchina fisica in cui lo studente accede con le proprie credenziali; al termine della sessione occorre caricare sul server le macchine virtuali aggiornate, in modo da conservare il lavoro svolto.
Questo sistema, per quanto possa essere flessibile e funzionale, ha alcuni punti critici: innanzitutto la dipendenza da un server esterno per lo storage delle macchine virtuali, necessaria per non vincolare lo studente a una singola postazione, comporta che, in caso di irraggiungibilità del server, sia impossibile accedere alle macchine virtuali salvate e conseguentemente svolgere le normali attività, inoltre rende necessario un backup costante del server, in modo da tutelarsi contro guasti gravi del sistema.
Sempre per quanto riguarda la dipendenza da un server esterno, il download e upload delle macchine virtuali costituiscono colli di bottiglia notevoli, considerato innanzitutto le considerevoli dimensioni di una macchina virtuale, in secondo luogo il numero degli studenti frequentanti il corso, che si aggira attorno agli 80-100; ciò comporta che il download di una macchina virtuale, in contemporanea da 100 studenti, possa durare anche 15 – 20 minuti, tempo destinato a crescere se si considerano i tempi di installazione e avvio della macchina.
L'insieme di queste problematiche hanno spinto i tecnici di laboratorio a cercare soluzioni alternative, caratterizzate da un flusso di dati tra macchine di laboratorio e server centralizzato che cercasse di eliminare o quantomeno limitare trasmissioni di ingenti quantità di dati; altro obbiettivo era la semplificazione dei strumenti di ripristino dei guasti, cercando di evitare backup eccessivamente pesanti; infine risultava utile creare un sistema in cui i singoli *host* potessero funzionare autonomamente anche in caso di guasto del server centrale, ma questo requisito non appariva facilmente realizzabile.

1.3 Descrizione di obbiettivi, requisiti e contesto dello studio.

Sin dai primi incontri per la definizione dello studio, le richieste avanzate dai tecnici erano molto chiare: come principale obbiettivo si poneva lo studio e la comprensione dei *containers*, focalizzandosi in particolare sulla piattaforma di sviluppo Docker, sia in ambito teorico sia soprattutto in ambito pratico, visto che nessuno dei due era già stato oggetto di particolari studi o ricerche.
In secondo luogo si puntava a realizzare un sistema alternativo alle macchine virtuali per il corso di Laboratorio di Sistemi Operativi, occorreva cioè realizzare un ambiente desktop Linux con tecnologia Docker, in modo da risolvere, almeno teoricamente, i problemi dell'attuale soluzione in uso; al fine di rendere utilizzabile

questo ambiente, occorreva però scontrarsi con determinate problematiche: innanzitutto capire se fosse possibile costruire una soluzione portabile su diversi sistemi operativi, rendendola compatibile almeno con Windows e le principali distribuzioni Linux con il minor numero di modifiche e aggiustamenti possibile.
Successivamente, pensando alla costruzione dell'ambiente vero e proprio, si riteneva necessario fornirlo di alcune applicazioni fondamentali per il corso in questione, come ad esempio un compilatore *gcc* o un browser, implementate ricorrendo anche a più *containers* se necessario; parallelamente si sarebbe valutato dove mettere in esecuzione i *containers*: se sfruttare le risorse offerte dalle tante macchine di laboratorio, mantenendo comunque i problemi di collo di bottiglia durante la fase di download e avvio dei *container*, oppure centralizzare l'esecuzione su un unico server, a discapito però di una maggiore dipendenza dalla stabilità di rete; inoltre, per ridurre al minimo l'influenza dell'ambiente esterno, si rendeva utile lo sviluppo di un repository privato da mettere in esecuzione su un server e/o cluster fisicamente collocato all'interno delle strutture informatiche del campus di Cesena , dove poter memorizzare versioni intonse dei *containers* sviluppati, al fine di poterne distribuire copie che ogni studente avrebbe modificato a proprio piacimento, potendone compromettere il funzionamento senza bisogno di grandi moli di lavoro per il ripristino, dato che sarebbe stato sufficiente fornire una nuova copia del *container* salvato.
Una volta terminato il processo di costruzione, risultava fondamentale comprendere se era possibile sfruttare la memorizzazione permanente per i dati creati, modificati o cancellati dentro un *container*: una delle feature principali presentate dall'approccio a *container*s era infatti la perdita di tutte le modifiche effettuate sul *container* alla sua chiusura, rendendo da un lato molto leggere operazioni come la modifica dei file e processi di sistema, ma dall'altro decisamente scomodo il non poter tener traccia delle operazioni effettuate; di conseguenza bisognava progettare una soluzione che permettesse al *container* in questione di salvare i file e le modifiche apportate dagli studenti durante il corso delle lezioni.
In caso di successo nella creazione di un ambiente con tutte queste caratteristiche, sarebbe seguita una fase di test in laboratorio, al fine di verificare prestazioni di server e *host*, in modo da comprendere definitivamente dove mettere in esecuzione i *containers* così creati.
Successivamente risultava opportuno quantificare il miglioramento ottenuto rispetto alle vecchie macchine virtuali, stimando le tempistiche di avvio, backup e rispristino da errori lievi e gravi.
In caso di fallimento invece, la nuova direzione dello studio sarebbe stata l'integrazione di *containers* e macchine virtuali, cercando così nuove piattaforme che potessero risultare più adatte rispetto a Docker per lo scopo fissato in origine.

Capitolo 2

Strumenti utilizzati

Nel corso di questo capitolo saranno trattati gli strumenti utilizzati durante l'esperienza, soffermandosi con particolare attenzione sulle tecnologie direttamente collegate alla virtualizzazione a *containers*.
Si traccerà un'ampia panoramica su Docker e sugli strumenti ad esso collegati, entrando nello specifico per quanto riguarda architetture e moduli, inoltre verrà analizzato l'insieme delle tecnologie impiegate per arricchire e/o migliorare le funzionalità dei *container* realizzati. Verranno poi brevemente esaminate le alternative a Docker, come ad esempio LXC; infine saranno presentate alcuni strumenti per il supporto di *containers* su *cloud computing*.

2.1 Sistema Docker per l'implementazione di *containers*.

Il progetto open source Docker viene rilasciato nel 2013 da una compagnia chiamata dotCloud, che lavorava su software di tipo PAAS (Platform as a service) [13] per il *cloud computing*.
Docker si pone come obbiettivo di automatizzare lo sviluppo di applicazioni all'interno di *containers* software, sfruttando tutti i vantaggi di una virtualizzazione a livello di sistema operativo; per riuscire a fare ciò, si avvale delle funzionalità di isolamento presenti nel kernel Linux, come ad esempio *cgroups*, utilizzato per la gestione di risorse fisiche, e *namespaces*, che invece garantiscono un isolamento a livello di processo.
La figura 2.1 illustra le modalità di accesso per Docker al kernel Linux: all'inizio Docker accedeva alle risorse in maniera indiretta, appoggiandosi su altre tecnologie come *container* LXC o librerie di virtualizzazione esterne(*libvirt*); a partire dalla versione 0.9 però, rilasciata a meno di un anno dal lancio, è stata aggiunta la libreria *libcontainer* [14], per avere un accesso diretto alle funzionalità del kernel Linux.

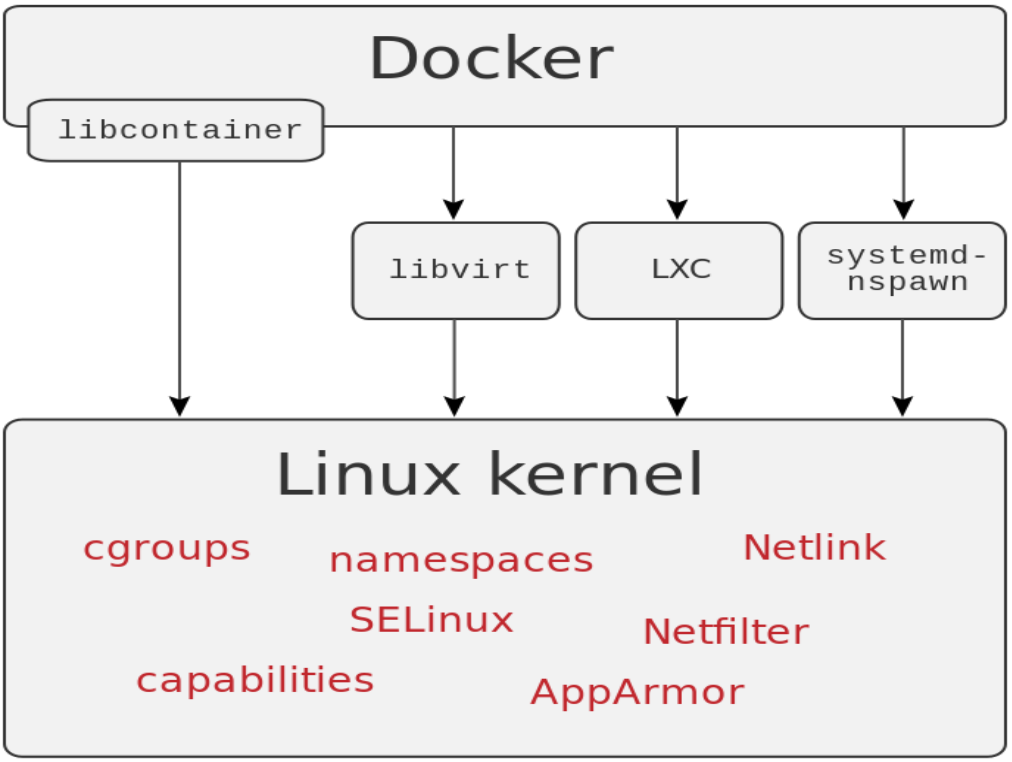

Figura 2.1 Modalità di accesso al kernel Linux (fonte: https://it.wikipedia.org/wiki/Docker)

Per implementare efficacemente tutti i vantaggi di un approccio a *container* e aggiungerne di nuovi, Docker ha messo in campo strategie di notevole ingegno: ad esempio viene impiegato UnionFS [15], un *filesystem* Linux che permette di simulare l'unione di più *filesystem*, mantenendo la semantica Unix; ciò consente di sovrapporre in modo trasparente file e directory di file system separati. Grazie a UnionFS è possibile documentare più facilmente le azioni eseguite su un *container*, in quanto ogni azione viene associata alle modifiche che ha effettuato su uno o più layer del file system e di conseguenza basta memorizzare solo cambiamenti dei vari layer per mantenere memoria delle azioni compiute; grazie a ciò la dimensione delle immagini risulta molto più contenuta (massimo 1 - 2 GB per le immagini più grandi) rispetto alle immagini utilizzate da piattaforme concorrenti.
Siccome i *containers* Docker sono così leggeri, uno stesso server o macchina virtuale può mandare in esecuzione molti più *containers* Docker rispetto che macchine virtuali o *containers* di altro tipo, un'analisi del 2016 ha infatti registrato che il caso d'uso di un sistema di media complessità prevede che ci siano circa 5 *containers* Docker in esecuzione per ciascun *host* coinvolto [16].
Tutte queste caratteristiche, combinate alle qualità dell'approccio a *containers*, rendono Docker una piattaforma molto valida per la creazione di ampli sistemi distribuiti, infatti con Docker si possono sfruttare singoli *host* per mettere in esecuzione un numero molto grande di applicazioni e processi in un ambiente isolato; ciò consente di semplificare non solo la gestione globale del sistema, permettendo di aggiungere e togliere nodi senza praticamente alcuno sforzo, ma anche la distribuzione del carico di lavoro sull'insieme del sistema risulta più agile nella gestione complessiva.

Visto questi enormi vantaggi, Docker viene integrato in moltissime infrastrutture differenti, come Amazon Web Service, Google Cloud Platform o Microsoft Azure, ci sono poi tutta una serie di software che dispongono di una compatibilità con *containers* Docker, ad esempio Nanobox, applicazione con uno scopo simile a Docker, o OpenShift, alternativa di Red Hat a Docker per lo sviluppo e manutenzione di applicazioni in *containers*.

2.1.1 Architettura Sistemi Docker.

Così come i *containers* offrono un approccio alla programmazione orientato alla modularizzazione e alla scalabilità, così l'architettura del sistema Docker si presenta sotto vari moduli, di cui alcuni solamente accessori, in modo da consentire l'installazione di ciò di cui si necessita a seconda delle diverse esigenze.
Il più importante tra tutti i moduli del sistema è *Docker Engine* [17] che si occupa di tutto ciò che riguarda il ciclo di vita dei *container*, come la creazione e messa in esecuzione; questo rende quindi *Docker Engine* il cuore del sistema Docker, su cui tutti gli altri moduli si appoggiano per la gestione dei *container*, tuttavia questo modulo non è compatto, ma può essere suddiviso in tre componenti ben distinte.
La prima e più importante è il *Docker Daemon* [18], un demone eseguito in background che si occupa della maggior parte del lavoro, gestendo sotto comando tutti gli oggetti del sistema; complementare a questo è il *Docker Client* [19], componente col compito di fornire un insieme di comandi all'utente per poter istruire il demone sul da farsi.
La comunicazione tra *Client* e *Daemon* è molto flessibile per quanto riguarda le modalità: è supportato infatti sia la comunicazione locale tramite socket Unix, nel caso *Client* e *Daemon* siano in esecuzione sulla stessa macchina, oppure tramite protocollo http nel caso di comunicazione remota; a prescindere dal tipo di comunicazione, vengono usate delle API proprietarie, dette REST API, per gestire la semantica dei messaggi scambiati tra i due.
Ultima componente non meno fondamentale è *Docker Registry* [20], registro che si occupa della memorizzazione delle *Docker Images* [21], il cui funzionamento sarà approfondito in seguito.
Docker mette a disposizione un registro pubblico, chiamato *Docker Hub*, in cui ognuno può postare le proprie *Docker Images* e scaricare le *Docker Images* di altri utenti; ogni *Docker Daemon* è impostato di default per cercare le immagini dei *containers* che non possiede localmente su questa piattaforma, ma è possibile anche creare un proprio repository privato e configurare *Docker Daemon* perché acceda a quello di default.
La figura 2.2 riassume egregiamente i moduli di *Docker Engine* e le loro iterazioni: si può notare come Docker Client comunichi istruzioni di *build*, *pull* e *run* a *Docker Daemon* che provvede a creare e a mettere in esecuzione i *containers* richiesti, compiendo operazioni di download da *Docker Registry* qualora le immagini necessarie non siano presenti sull'*host* di esecuzione.

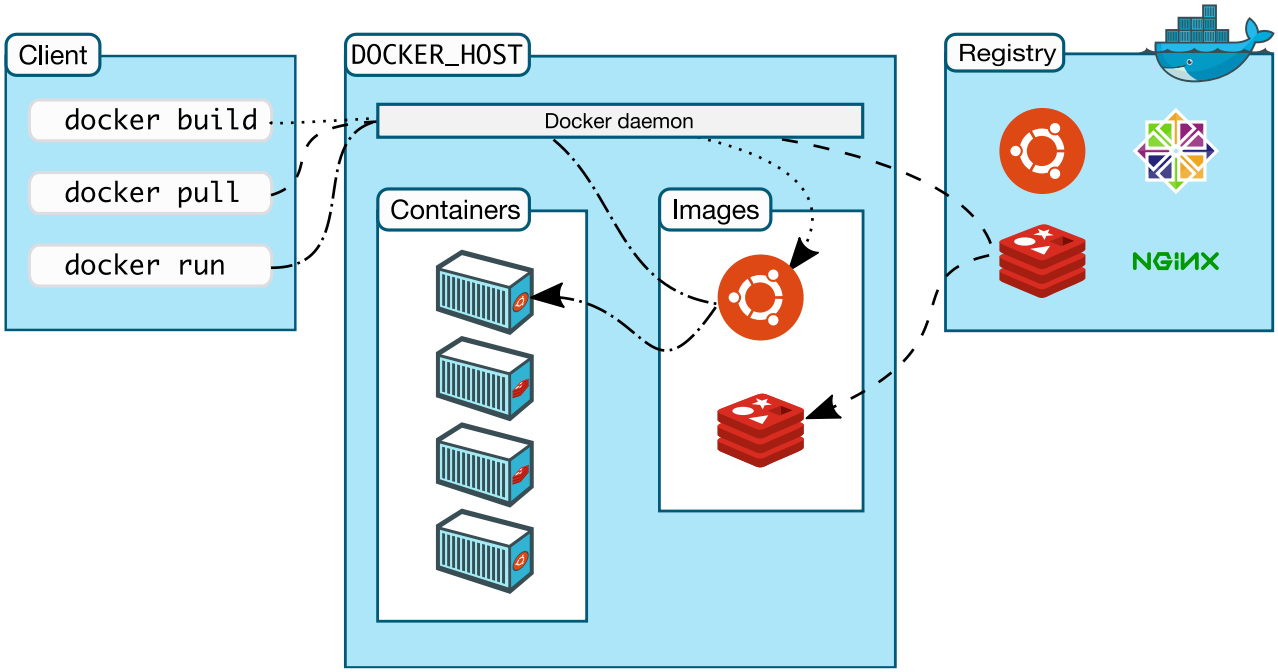

Figura 2.2 Architettura Docker Engine (fonte: https://docs.docker.com/engine/docker-overview/#docker-engine)

Come detto in precedenza, *Docker Engine* costituisce una base su cui si appoggiano alcuni moduli aggiuntivi come ad esempio *Docker Machine* o *Docker Compose*.
Docker Machine rappresenta una prima soluzione per garantire la compatibilità di *container* Docker con sistemi operativi senza kernel Linux: semplicemente, sfruttando uno strumento di virtualizzazione esterno come *VirtualBox* [21], si crea un ambiente virtuale Linux essenziale su cui poi viene installato *Docker Engine*; a costo di un piccolo *overhead* si riesce a lavorare con Docker a prescindere dal sistema operativo.
Docker Machine [22] sta diventando tuttavia obsoleto: sono nati infatti alcune applicazioni native, come ad esempio *Docker for Windows*, per i principali sistemi operativi che risultano più efficienti; questo relega *Docker Machine* a un ruolo di supporto per la compatibilità con sistemi obsoleti, su cui le nuove applicazioni non sono compatibili, oppure a simulatore per sistemi scalati su singolo *host*, potendo virtualizzare diversi *host* con installato *Docker Engine*.
Docker Compose [23] è invece uno strumento di supporto per la gestione di applicazioni scalate su un piccolo numero di *containers*, sfruttando un apposito file di configurazione in formato YML [24] è possibile indicare numero e tipologia di *containers* da mandare in esecuzione, definendo riferimenti e modalità di comunicazione.

2.1.2 Docker Swarm Mode.

Nel paragrafo precedente si è trattato del funzionamento della piattaforma Docker per quanto riguarda applicazioni distribuite su un solo o comunque piccolo numero di *containers*, in esecuzione preferibilmente sullo stesso *host*.
Tuttavia se le modalità di funzionamento di Docker si limitassero a questo, la piattaforma in questione avrebbe avuto molto meno successo, causa la poca efficienza nel *cloud computing*.
Per ottenere performance ottimali anche nell'ambito sopracitato, Docker mette a disposizione una modalità di funzionamento alternativa, chiamata *swarm mode*.
Docker swarm mode [25] si compone di tutto un insieme di tecniche per gestire cluster di *containers* Docker distribuiti su rete, sfruttando un insieme di comandi e API, inclusi nel *Docker Engine*, che rendono trasparente al programmatore la parte di configurazione e manutenzione dei vari cluster.
Le differenze di progettazione e funzionamento della *swarm mode* rispetto alla modalità classica sono molto marcate: scopo del *container* non è più ospitare l'applicazione nella sua interezza, ma offrire un servizio che, combinato con altri servizi di nature differenti in esecuzione su *containers* differenti, possa fornire le stesse funzionalità dell'applicazione richiesta.
Unità base della *Docker swarm mode* è il nodo [26], ovvero un'istanza di *Docker Engine* appositamente configurata per funzionare in *swarm mode* e collegata a un cluster di altri nodi, è inoltre fondamentale che quest'istanza possa accedere alla rete e comunicare attraverso di essa; ogni nodo poi ha una propria natura: può essere definito come Manager qualora si voglia assegnargli il compito di coordinare gli altri nodi del cluster, dividendo tra essi i carichi di lavoro e coordinandosi con altri Manager; tra tutti i Manager è presente un nodo Leader, che ha il compito di coordinare tra loro i vari Manager.
Se invece si vuole che il nodo si occupi solo di eseguire i servizi richiesti, allora conviene assegnarli la natura di Worker, così facendo le risorse del nodo verranno interamente impiegate per la realizzazione di uno o più servizi.
Andando più nello specifico, un nodo di natura Manager comunica con altri Manager per assicurarsi che lo stato e le prestazioni del cluster non subiscano bruschi peggioramenti, tuttavia molto spesso si occupa anche di parte del carico di lavoro complessivo, utilizzando le proprie risorse per mettere in esecuzione uno o più servizi; dall'altro lato un nodo di natura Worker impiega le sue risorse interamente per soddisfare il carico di lavoro complessivo, inoltre ha in esecuzione un task di natura agente per notificare il suo stato al Manager a cui è associato.
Passando poi al funzionamento dell'insieme dei nodi nel cluster, occorre specificare che la gestione del sistema, o Orchestration che dir si voglia, avviene tramite specifiche API per etichettare i ruoli dei singoli nodi, uno scheduler che si occupa di creare i vari task e di gestirli a seconda dello stato del sistema e infine un dispatcher, il cui compito è mandare in esecuzione sui nodi i task secondo le istruzioni dello scheduler e sorvegliare l'andamento del sistema tramite una serie di controlli periodici.
Avendo trattato del rapporto tra nodi e cluster, si rende necessario parlare di come vengono gestite le problematiche all'interno di un cluster: sono presenti infatti

diversi meccanismi di difesa contro i possibili problemi che possono sorgere. Qualora sorga un errore o un blocco, la prima cosa che viene fatta è controllare la natura del nodo su cui è sorto: se è un Worker, i suoi processi vengono interrotti, il nodo viene posto momentaneamente offline e il suo carico di lavoro viene ridistribuito sulla rete; nel caso invece si tratti di un nodo Manager vengono eseguite le stesse operazioni sopracitate, inoltre si riassegnano i Worker che facevano capo al nodo in questione a un altro Manager, infine nel caso si trattasse del nodo Leader, un Manager sarebbe eletto come nuovo leader; una volta eseguita la procedura di trasferimento, il sistema continua col suo normale funzionamento, a meno che non si trovi senza più nodi Manager, in quel caso l'intero cluster registra uno stato di errore e si disattiva.
Come già detto in precedenza ogni nodo si occupa di mettere in esecuzione uno o più servizi, traducibili come insiemi di task da eseguire, solitamente a un task corrisponde un *container* e una serie di comandi per realizzare i compiti necessari alla costituzione del servizio. Le opzioni disponibili per questi servizi sono direttamente collegati alle possibilità di personalizzazione dei *containers* che li compongono; in generale per ogni *container* è possibile specificare un limite alle risorse utilizzabili, le porte disponibili per socket di comunicazione, politiche sui tempi di aggiornamento e il numero di repliche che il task deve avere all'interno del sistema.
Per quanto riguarda la distribuzione dei task nel sistema, è buona prassi isolare atomicamente i task e dedicare a ciascuno un *container* separato, in modo da ridurre al minimo le interferenze, inoltre è opportuno che ogni servizio o insieme di task abbia più repliche messe in esecuzione su nodi differenti del sistema; per quanto questa soluzione vada a complicare il funzionamento del cluster, aggiungendo un ulteriore grado di sincronizzazione tra servizi replicati, la resistenza agli errori ne guadagna enormemente, poiché un servizio i cui task riscontrino errori può essere terminato senza perdere i progressi, in quanto questi sono replicati su altre macchine in rete.
Al fine di gestire meglio la distribuzione dei vari task, il cluster si dota di uno scheduler, il cui compito è di assegnare task al nodo più idoneo alla sua esecuzione; nel caso però nessun nodo possa eseguire il task, allora il servizio proprietario passa in stato "*Pending*", stato che può essere attribuito anche a mano qualora si voglia limitare il carico di lavoro del sistema.
Docker Swarm Mode ha ben due modalità alternative per la gestione dei servizi e conseguentemente dei task ad essi collegati [27]: la prima è chiamata *Replicated Service*, mentre la seconda *Global Service*.
Replicated Service prevede che si definisca un numero di repliche del servizio da mantenere in esecuzione in contemporanea, qualora il numero di repliche scendesse sotto quello desiderato, lo scheduler creerebbe una nuova istanza di quel servizio e il dispatcher metterebbe in esecuzione i task corrispondenti sul nodo più appropriato.
Global Service invece mette in esecuzione un'istanza del servizio per ogni nodo del cluster, nel caso si aggiunga un nodo alla rete, lo scheduler provvederà a mettere in esecuzione un'istanza del servizio su di esso.
La figura 2.3 rappresenta un esempio di un semplice cluster su cui sono in esecuzione un servizio di tipo *Replicated* e uno di tipo *Global*: come è possibile

notare il numero di repliche per il servizio *Replicated* è stato impostato a 3 e di conseguenza compare su tre nodi di categoria *Worker*, mentre il servizio *Globa*l compare su ogni nodo del cluster, *Manager* compreso.

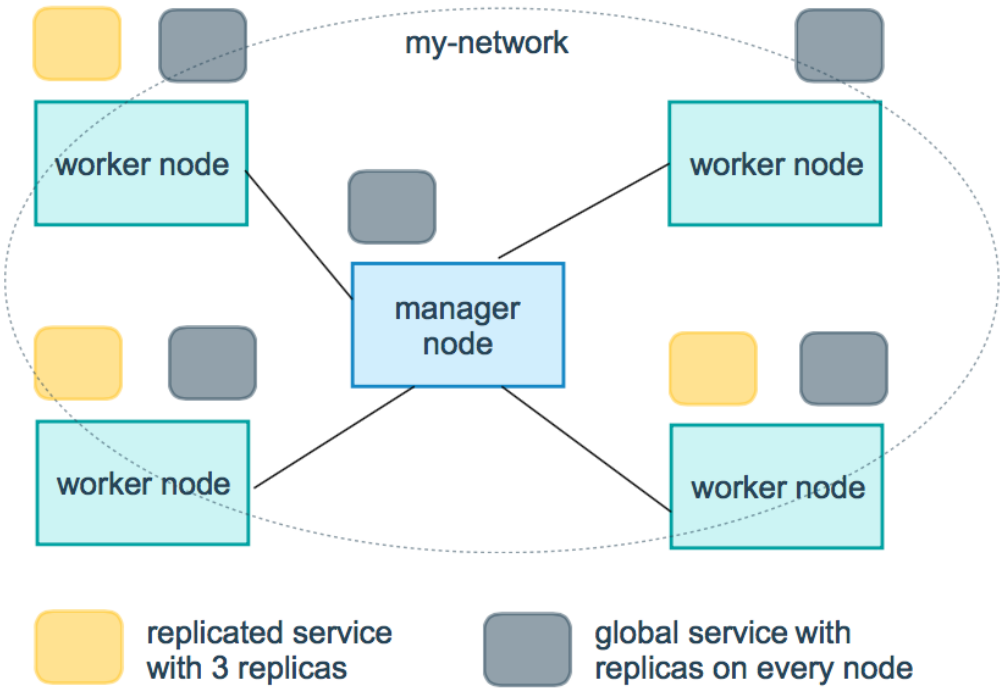

Figura 2.3: Esempio di Cluster Swarm con servizi di natura differente (fonte https://docs.docker.com/engine/swarm/how-swarm-mode-works/services/#replicated-and-global-services).

2.1.3 *Docker Images*.

Tra tutti i gli oggetti di Docker, chiamati *Docker Objects*, senza dubbio le più importanti sono le *Docker Images*, che costituiscono infatti la base del meccanismo di funzionamento dei *containers* Docker.
Le *Docker Images* [28] sono sostanzialmente templates che contengono le istruzioni necessarie per creare un *container* Docker, questi templates sono gli oggetti protagonisti della maggior parte delle iterazioni all'interno dei moduli di *Docker Engine*: per mandare in esecuzione un'applicazione, è necessario che sull'*host* sia presente l'immagine corrispondente, altrimenti occorre scaricarla dall'apposito registro.
Al fine di regolamentare l'utilizzo delle immagini nel modo più efficiente possibile, Docker persegue alcune severe regole per la loro gestione: innanzitutto ogni immagine può essere acceduta solo in lettura, in modo da prevenire modifiche che potrebbero alterare la natura del *container* da essa creato; in secondo luogo qualora

si desiderasse aggiungere o modificare le funzionalità dell'immagine, si renderebbe necessario creare un nuovo template che comprenda anche le funzionalità del precedente; infine ogni immagine deve essere documentata adeguatamente nella sua struttura mediante un apposito file chiamato *Dockerfile*.
I *Dockerfile* [29] esprimono, mediante una sintassi apposita, la serie di comandi e operazioni necessaria per costruire una *Docker Image* partendo da un'immagine di riferimento, ogni *Dockerfile* è collegato a una e una sola immagine e può essere interpretato mediante un apposito comando da un *Docker Daemon* che provvederà a costruire il template richiesto.
Ogni *Dockerfile* può essere diviso in tre parti distinte, facilmente distinguibili tra loro.
Nella prima è possibile trovare l'immagine di partenza del template che verrà costruito, preceduta dalla parola chiave FROM; fatta eccezione per alcune immagini "primitive", cioè che realizzano le componenti di un semplice user space, ogni altra immagine si riferirà sempre a un altro template che realizza parte delle funzionalità dell'applicazione per cui è stata progettata: ad esempio se si volesse creare il template di un applicazione che necessita di più componenti, come uno stack Lamp, allora si dovrebbe cercare un'immagine di partenza che realizzi una buona parte dei componenti, in questo caso un'immagine che realizzi un server Apache e/o un database MySQL.
Nella seconda sezione si trova l'insieme dei comandi da eseguire sull'immagine di partenza per ottenere il risultato richiesto: ogni comando è preceduto dalla parola chiave RUN ed è espresso nel linguaggio *bash* della distribuzione Linux che si sta utilizzando per quel template.
Infine l'ultima sezione è dedicata ai comandi che devono essere eseguiti all'avvio del *container*, è possibile specificare un solo insieme di comandi preceduti dalla parola CMD; come per le istruzioni nella sezione precedente, anche qui è necessario utilizzare il linguaggio *bash*.
È importante sottolineare che sono presenti alcuni comandi che possono essere specificati in multiple sezioni, poiché gestiscono processi come la copia di file, variabili d'ambiente e porte esposte per la comunicazione.
Per costruire il template specificato nel *Dockerfile* corrispondente, il Docker Engine esegue le seguenti operazioni: viene controllato se in memoria è presente l'immagine di partenza e se non è presente, viene scaricata dal repository di default, poi si crea un *container* con il template di partenza in esecuzione, si esegue il primo comando della seconda sezione, una volta terminato si effettua un *commit* dell'immagine e il *container* in questione viene distrutto; per tutti i successivi comandi della seconda sezione si ripete il processo, utilizzando come immagine di partenza il *commit* effettuato al passo precedente, infine generato un ultimo *container* in cui vengono eseguite le istruzioni della terza sezione, poi il template risulta pronto.

2.2 Strumenti di supporto al lavoro con *containers* Docker.

Nella sottosezione 2.1 si è trattato ampliamente delle modalità di funzionamento dei *containers* Docker, descrivendo accuratamente architetture e moduli, tuttavia se le potenzialità di Docker si limitassero solo a quanto descritto nella precedente sezione, il successo della piattaforma sarebbe stato molto minore.
Il concetto di *container* Docker, come presentato sopra, presenta limiti ben visibili, come ad esempio l'impossibilità di tenere traccia delle modifiche effettuate a meno di creare una nuova immagine, oppure l'impossibilità di creare e gestire dati persistenti, o ancora l'isolamento da feature può diventare un peso, qualora impedisca di accedere/comunicare con ambienti degli altri *containers*.
Per riuscire a risolvere parte di queste problematiche, sono stati utilizzati una serie di strumenti, alcuni sviluppati da software house esterne, altri invece proprietari di Docker, il cui impiego aumenta notevolmente l'efficienza e le possibilità delle applicazioni realizzate.
Nel corso di questa sottosezione verranno analizzate i principali strumenti utilizzati durante il corso dello studio; benché questi strumenti di supporto siano molto numerosi, si è scelto di trattare quelli più inerenti al lavoro svolto, in modo da facilitare la comprensione delle fasi successive.

2.2.1 Visualizzazione Remota: VNC e XRDP.

Il primo problema sorto dopo lo studio e il test dei primi semplici *container* è stato come dotare un *container* di interfaccia grafica: tutte le prime immagini testate riproducevano solamente un terminale testuale e sulle guide presenti sui siti ufficiali non era presente nessuna sezione riguardante componenti native Docker per visualizzare interfacce grafiche.
Non essendoci vere e proprie linee guida, si è scelto di cercare tra immagini presenti sul repository ufficiale Docker, DockerHub, quelle che implementassero interfacce utente grafiche: i principali risultati funzionanti erano immagini che utilizzavano tecnologie di *Virtual Network Computing* [30].
I *Virtual Network Computing* (*VNC* abbreviato) sono software di controllo remoto per gestire tramite interfaccia grafica un *host* collegato su rete, consentono infatti di riprodurre su schermo l'ambiente grafico della macchina remota e di interagirvi inviando input con i classici dispositivi come mouse e tastiera.
Il sistema si compone di due software comunicanti tra loro, ovvero *VNC server* e *VNC client*: *VNC server* è il software che va installato sulla macchina da pilotare, mentre *VNC client*, chiamato anche *Viewer*, viene eseguito sulla macchina da cui si intende manovrare il sistema.
La comunicazione *Client-Server* avviene tramite un protocollo specifico chiamato *RBF* (Remote Framebuffer protocol) [31], i messaggi scambiati tra client e server hanno contenuti differenti a seconda della direzione della comunicazione: il client notifica al server gli input registrandoli e poi inviandoli come eventi, il server invece

invia al client comunicazione contenenti primitive grafiche su come gestire lo schermo (ad esempio: “colora il pixel in posizione X Y del colore 123”).
La comunicazione *VNC* sfrutta solitamente le porte dalla 5900 in poi per riprodurre diversi tipi di oggetti a seconda del numero della porta, ad esempio una comunicazione passante per la porta 5900 indica la riproduzione di un ambiente desktop; può essere inoltre utilizzata la porta 5800 qualora si voglia accedere tramite interfaccia web.
Il protocollo *RBF*, e di conseguenza *VNC*, ha però due grandi problematiche: in primo luogo la comunicazione *RBF* non è una comunicazione sicura, poiché, per quanto sia presente un sistema di autenticazione con password sui server *VNC* e le password vengono criptate, non è presente un sistema di crittografia a *doppia chiave*, di conseguenza la chiave crittografica è una sola e viene scambiata durante la comunicazione, cosa che la rende vulnerabile a uno sniffing sulla rete.
L’altra grande problematica di *RBF* è costituita dalla sua scarsa efficienza: il basarsi su un comando diretto dei pixel rende molto complicati e pesanti i messaggi da scambiare, soprattutto se paragonati con altri protocolli basati su primitive grafiche di più alto livello (ad esempio: “Apri una finestra”).
La prima problematica non aveva un impatto particolarmente pesante sul sistema che si intendeva realizzare, mentre la seconda poteva essere un fattore con un notevole peso, di conseguenza si è cercato un altro strumento per la riproduzione remota.
Lo strumento selezionato è stato *XRDP* [32], reimplementazione in ambiente Unix del protocollo *RDP* [33] di Windows.
RDP, o “*Remote Desktop Protocol*” è un protocollo di rete, proprietario di Microsoft, che permette la connessione remota da un computer all’altro in maniera grafica. La principale differenza con *VNC* sta nel fatto che *RDP* utilizza una comunicazione basata su primitive grafiche e comandi di più alto livello, al contrario delle istruzioni pixel per pixel di *VNC*; questo comporta che, a scapito di una minore compatibilità per via dell’utilizzo di primitive che potrebbero non essere interpretate da ogni macchina, la comunicazione tramite *RDP* sia più veloce e leggera rispetto alla controparte *VNC*.
Durante l’esperienza, entrambe le soluzioni sono state utilizzate, tuttavia per ragioni che verranno chiarite meglio nei capitoli che tratteranno delle immagini sviluppate, si è scelto di utilizzare *XRDP* per progettare la visualizzazione grafica dell’ambiente Linux da realizzare nel corso dello studio.

2.2.2 Networking

Nel momento in cui si è pensato di utilizzare tecniche di visualizzazione remota, sono sorti alcuni dubbi riguardante la natura dell’isolamento che contraddistingue i *container*: fino a che punto l’ambiente di esecuzione del *container* rimane isolato? In che modo si può costruire la comunicazione tra due distinti *containers*?

Per regolamentare la comunicazione tra *containers* in rete, Docker mette a disposizione una serie di strumenti accessori, composti principalmente da driver per il networking.
Di default Docker fornisce due driver di rete chiamati *Bridge* e *Overlay* [34], lasciando anche la possibilità all'utente di aggiungere driver di rete personalizzati; questi driver servono poi a gestire le reti interne a *Docker Engine*, principale mezzo di comunicazione per i *containers*, sia fra di loro che verso il mondo esterno.
Ogni *container*, al momento della creazione, viene registrato su una rete di default, chiamata *Bridge*; la rete in questione si occupa di fornire un indirizzo IP ai *container*s e di svolgere i normali compiti di una *LAN* (Local Area Network), connessa comunque con il mondo esterno grazie alla scheda di rete dell'*host* fisico su cui sono in esecuzione i *containers*.
La figura 2.4 riassume rapidamente lo stato di rete di un *container* creato in maniera standard: questo viene infatti registrato sulla rete *Bridge* (Docker0 nell'immagine) e gli viene attribuito un indirizzo IP, che servirà a eventuali altri *containers*, collegati sempre su Bridge, per comunicare con lui; qualora invece il *container* in questione voglia inviare dati a un *host* esterno alla rete bridge, le comunicazioni vengono reindirizzate sulla scheda di rete dell'*host* fisico.

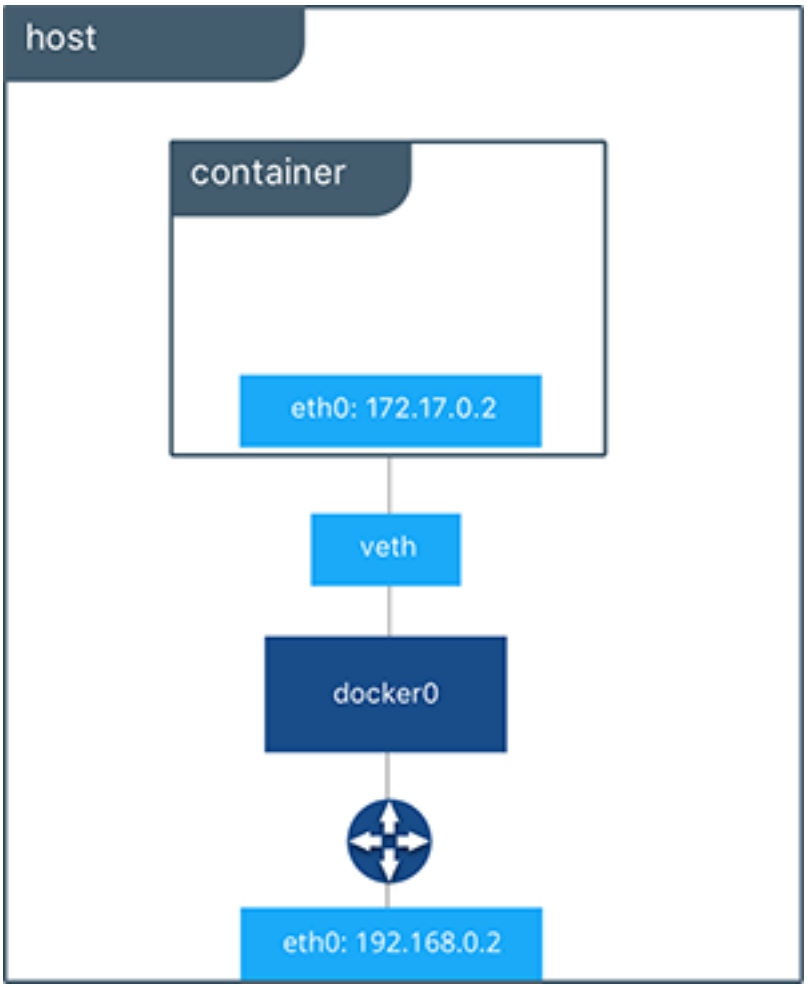

Figura 2.4: Struttura di rete di un container (fonte: https://docs.docker.com/engine/tutorials/networkingcontainers/)

Come detto in precedenza, è possibile però anche creare le proprie reti personalizzate e registrarvi *container*; i due driver *Bridge* e *Overlay* servono appunto per specificare la natura della rete che si andrà a creare: una rete che utilizzi il driver *Bridge* è una rete locale a un singolo *host*, mentre quelle che utilizzano *Overlay* possono includere più istanze di *Docker Engine*.
Una volta scelto il driver più appropriato e impartiti i giusti comandi a *Docker Engine*, la rete sarà pronta per registrare *containers* e gestire le loro comunicazioni.
La possibilità di gestire più reti su cui registrare i *containers* è alla base delle politiche di isolamento di Docker: due *container* (figura 2.5), registrati su due reti differenti, non possono comunicare tra di loro, qualunque tentativo di accesso esterno alla rete viene bloccato dalle politiche di *nat*, che impediscono qualunque comunicazione non sia stata iniziata da un *host* all'interno della rete locale.

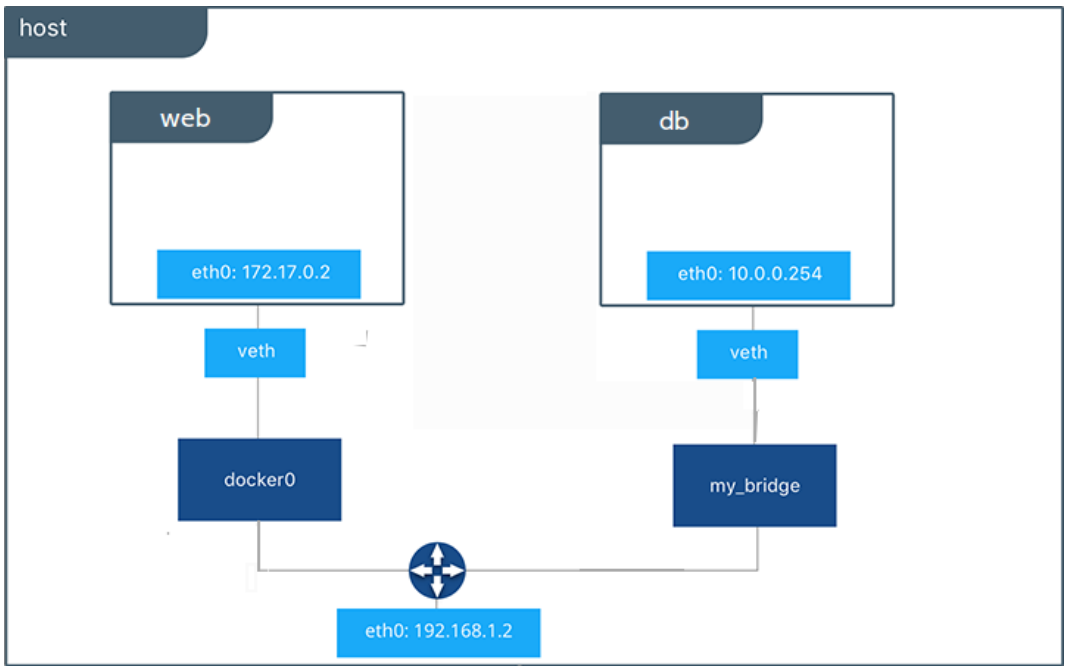

Figura 2.5: Container registrati su network differenti (fonte: https://docs.docker.com/engine/tutorials/networkingcontainers/)

L'impossibilità di comunicare tra reti differenti può diventare un limite non indifferente in determinate situazioni, per questo motivo Docker ha scelto di implementare la possibilità di registrare un *containers* su multiple reti: qualora si voglia comunicare con altri nodi su una rete differente da quella su cui si è attualmente registrati, basta semplicemente che si registri questo *container* sulla rete desiderata, che provvederà ad assegnargli un indirizzo IP, che si andrà ad affiancare ad eventuali altri indirizzi di altre reti(figura 2.6).

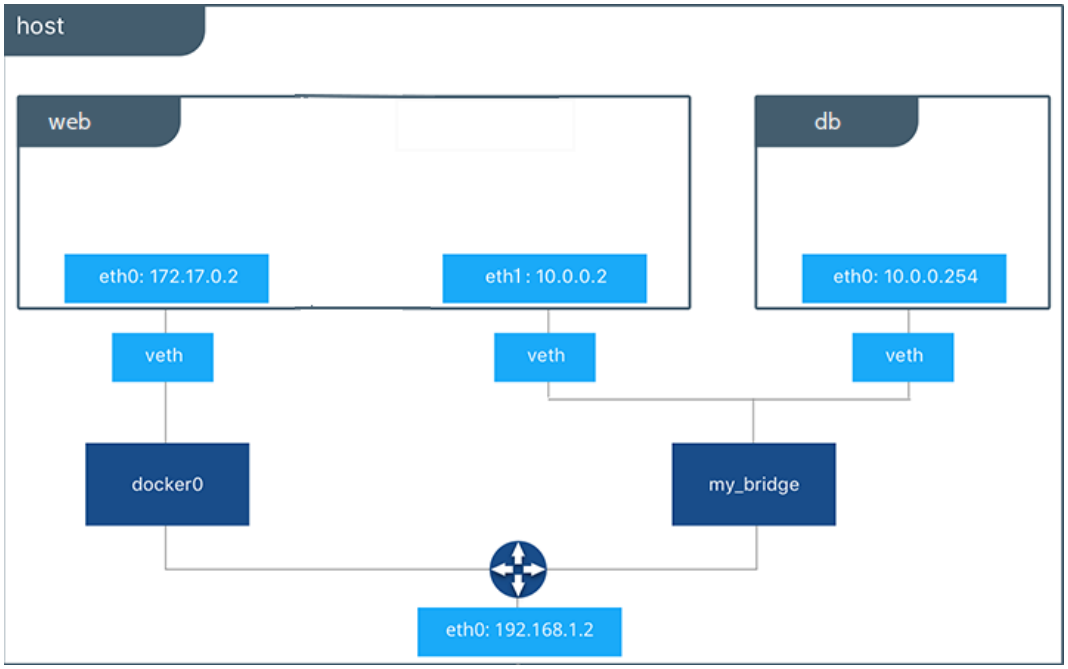

Figura 2.6: Container registrato su più reti (fonte: https://docs.docker.com/engine/tutorials/networkingcontainers/)

2.2.3 Storage di dati persistenti

Ultimo problema fondamentale sorto in fase di analisi riguardava i dati persistenti: una delle peculiarità dei *containers* era infatti l'assenza di memorizzazione delle modifiche effettuate e dei dati prodotti una volta terminata l'esecuzione; per quanto ciò rendesse i *container* dei perfetti ambienti per lo sviluppo e testing delle applicazioni, sarebbe risultato molto difficile che potessero risultare idonei per l'esecuzione di programmi e servizi, visto la grande quantità di dati che si deve memorizzare per il loro corretto funzionamento.
Di default è possibile memorizzare dati all'interno di un *container* [35], grazie a un apposito *layer* presente all'interno del *container* stesso; questa soluzione tuttavia comporta non pochi svantaggi: innanzitutto non garantisce la persistenza dei dati, che vengono cancellati nel momento in cui il *container* termina la sua esecuzione; inoltre il *layer* di scrittura del *container* è strettamente legato all'*host* su cui è in esecuzione, di conseguenza non risulta facile muovere i dati su altri supporti; infine è necessario un driver specifico, chiamato *storage driver*, per gestire il *filesystem* durante i processi di scrittura e lettura dal *layer* dedicato; questo driver fornisce un ulteriore *filesystem* di tipo *Union*, sfruttando il kernel Linux, causando un'ulteriore utilizzo di risorse, a cui segue un peggioramento delle performance del sistema.
Docker mette a disposizione tre diverse soluzioni per risolvere il problema dei dati persistenti: *Docker volumes* [36], *Bind Mounts* [37] e *tmpfs volumes* [38].
Per quanto riguarda la prospettiva dall'interno del *container*, tra le tre opzioni non c'è alcuna differenza, tutte e tre vengono infatti gestite dal *filesystem* come un *mount* di una directory o un file a cui poter accedere; per riuscire a comprendere le

differenze tra i tre diversi metodi di memorizzazione occorre infatti pensare a dove risiederanno i dati memorizzati sull'*host* del *Docker Engine* (figura 2.7).

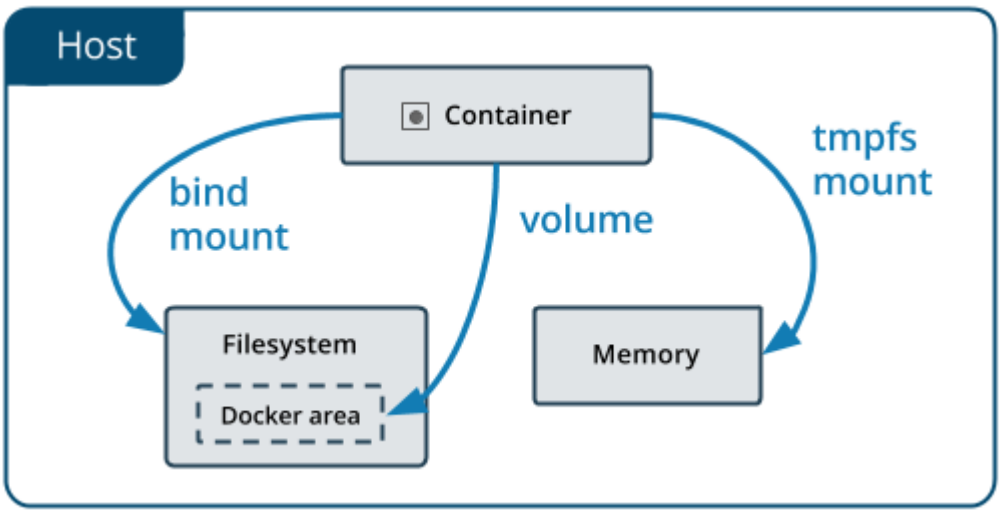

Figura 2.7: Tipologie di storage persistente possibili (fonte: https://docs.docker.com/engine/admin/volumes/tmpfs/)

Se si sceglie di creare un volume di categoria *Docker Volumes*, allora si disporrà di una struttura di memorizzazione salvata in un'apposita directory all'interno dell'area del *filesystem* dell'*host* gestita esclusivamente da Docker; così facendo ogni altro processo non Docker è impossibilitato ad accedere a questi dati.
Nel momento in cui si vuole utilizzare il volume, basta eseguire la procedura di *mount* al momento della creazione del *container*, utilizzando alcuni semplici comandi; ogni volume può essere utilizzato in contemporanea da più *containers* e anche qualora il volume non risultasse impiegato, rimarrebbe salvato nella directory dedita alla memorizzazione dei volumi.
I *Docker Volumes* poi supportano appositi driver, detti *Volume Drivers*, per lo storage su *host* remoto o su un provider *cloud*, espandendo così le possibilità di memorizzazione.
Una seconda possibilità è costituita da i *Bind Mounts*: disponibili sin dalle prime versioni di Docker, offrono un approccio diverso allo storage permanente rispetto ai *Docker Volumes*. Quando si usa un *Bind Mount*s infatti, un file o directory del *filesystem* dell'*host* vengono aggiunti, tramite un'operazione di *mount*, al *container*; ciò fornisce la possibilità di un accesso più veloce e dinamico ai dati prodotti dal *container*, infatti la directory o file utilizzati nel processo di *mount* sono disponibili anche ad ogni altro processo in esecuzione sull'*host*, al contrario di quanto accadeva con i Docker *Volumes*.
La libertà concessa da un *Bind Mount* è però un'arma a doppio taglio: innanzitutto causa una forte dipendenza dalla struttura del *filesystem* dell'*host* sottostante, in secondo luogo compromette la proprietà dell'isolamento del *container*: processi pensati per essere eseguiti in un ambiente sicuro e isolato sono in grado ora di accedere a parte del *filesystem* sottostante e di, potenzialmente, interagire con file e directory di sistema, modificandoli a piacimento.

L'ultimo tipo di storage per i dati è chiamato *tmpfs volumes* e il suo utilizzo diverge profondamente dalle due alternative precedenti: il suo impiego è infatti per la gestione e condivisione di dati non persistenti, salvati sulla memoria RAM dell'*host*; le tipologie di dati gestite con questo supporto sono solitamente informazioni sensibili, che si ritiene utile cancellare una volta terminata l'esecuzione.
Tra le tre alternative, la più caldamente consigliata è quella basata sui *Docker Volumes*, poiché consente una gestione dei dati più idonea ai principi della progettazione a *containers*; qualora però si ritenga utile rilassare alcune di queste regole, al fine di garantire magari uno scambio di file più agevole, *allora Bind Mount* risulta la scelta più valida; infine nel caso si abbia a che fare con dati sensibili che occorre eliminare una volta terminato il loro impiego, i volumi di tipo *tmpfs* risultano i più idonei allo scopo.

2.3 Altre implementazioni per i *containers*

Per quanto Docker sia il principale riferimento nell'ambito della virtualizzazione a *containers*, sono presenti molte alternative, nate sia prima che dopo l'avvento di quest'ultimo, che realizzano implementazioni differenti rispetto a Docker, ma altrettanto valide in molte situazioni.
Queste alternative sono state studiate nel momento in cui la parte principale dello studio di fattibilità si è conclusa, con particolare interesse per quanto riguardava *containers* basati su kernel Windows, in un secondo momento invece sono stati valutati sistemi basati su kernel Linux come Docker.

2.3.1 *Containers* basati su kernel Windows.

Pensando a un'integrazione tra la virtualizzazione a *containers* e il sistema operativo Windows è molto facile correre con la fantasia: la possibilità di integrare un ambiente leggero, rapido e autosufficiente con uno dei più diffusi sistemi operativi al mondo e avere così a disposizione applicazioni quasi autosufficienti, combinata con l'ampia vastità di software progettati per Windows, avrebbe da sola risolto la maggior parte dei problemi di portabilità dei software su ogni sistema operativo, tuttavia la realtà è ben diversa, almeno per il momento.
La collaborazione tra Docker e Microsoft comincia nel 2014, ma è nel settembre 2016, con il rilascio di Windows Server 2016 che i *containers* basati su kernel Windows sbarcano sul mercato.
Le caratteristiche di Windows containers rimangono pressoché le stesse dei *containers* Docker, seppur con alcune differenze: per prima cosa, l'architettura della piattaforma Windows ha come base alcune primitive per il supporto ai *containers*, comprese nel virtualizzatore *Hyper-V* del kernel Windows, invece di

Docker Engine; in secondo luogo, i *containers* Windows possono essere di due tipi (figura 2.8) a seconda del grado di isolamento che si desidera.
I *Windows Server Containers* [39] sono la tipologia meno isolata: infatti, come accade su *Docker Engine*, tutti i *containers* hanno accesso allo stesso Kernel Windows; questo fa sì che l'*host* possa vedere i processi eseguiti all'interno dei diversi *container*; dall'altra parte invece gli *Hyper-V Containers* [39] forniscono un isolamento assoluto, grazie alla virtualizzazione di un kernel minimale non condiviso con nessun altro *container*.

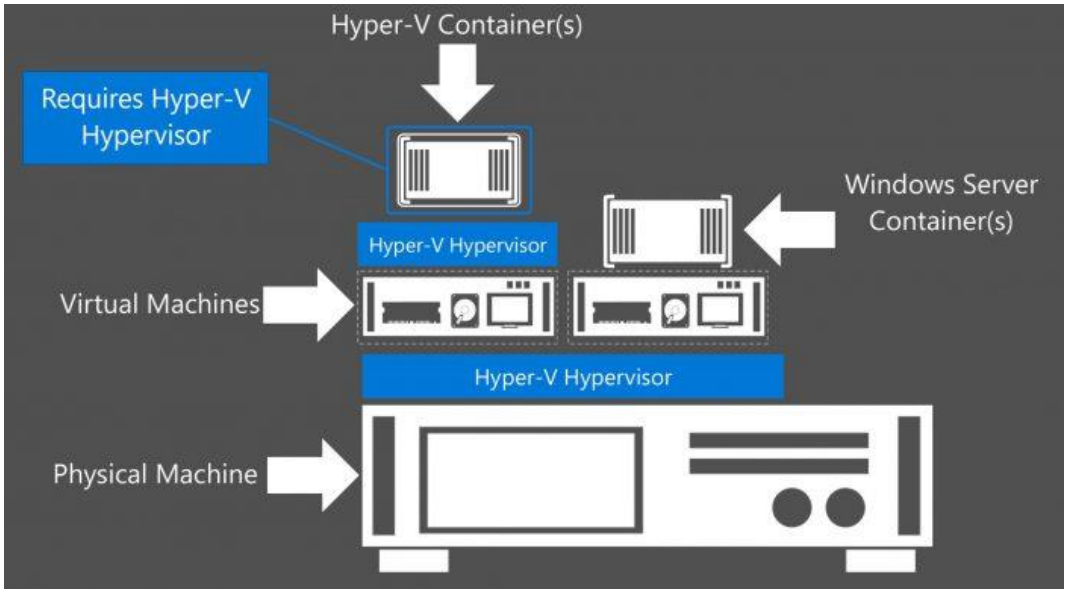

Figura 2.8: Modello di containers Windows (fonte: https://www.windowserver.it/2015/11/windows-server-2016-introduzione-ai-container/)

Analogamente a quanto accade su Docker, anche per Windows containers è presente un sistema di gestione basato su immagini; come ogni immagine Docker si basa su un template di un sistema operativo (ad esempio: Ubuntu), così ogni immagine Windows è costruita su un template di Windows Server; a differenza di quanto accade per Docker, dove sono presenti diversi template di partenza a seconda della distribuzione Linux che si preferisce utilizzare, per un *container* Windows sono presenti solo due immagini di partenza, ovvero *Windows Server Core* e *Nano Server* [40].
Nano Server è costituito da un'installazione minimale di Windows, comprendente gli strumenti necessari per garantire la compatibilità con un numero limitato di applicazioni; a scapito di una minor sicurezza e compatibilità si ottengono immagini più leggere.
Windows Server Core è invece un'installazione completa di Windows, permette quindi di eseguire qualunque software sia compatibile con un sistema operativo Windows; ne consegue che le immagini sviluppate con questa base risulteranno molto più pesanti rispetto a quelle sviluppate con *Nano Server*.

2.3.2 *Containers* LXC e LXD.

La principale alternativa, in ambiente Linux, a Docker è stata LXD: nel momento in cui lo studio di fattibilità, che utilizzava Docker come piattaforma di sviluppo, ha prodotto il suo risultato, di cui si tratterà nel capitolo successivo, si sono valutate altre piattaforme, sempre basate su *containers*, per riuscire a sviluppare un'applicazione che avesse tutte le caratteristiche richieste nel capitolo 2; la piattaforma più idonea escludendo Docker era LXD.
Prima di trattare del funzionamento dei *containers* LXD è necessario soffermarsi su LXC, punto di partenza sia di LXD che di Docker.
LXC nasce come ambiente di virtualizzazione a *containers* sviluppato appositamente per un kernel Linux, risultando però molto simile a ad altri sistemi di virtualizzazione già esistenti da tempo sia su Linux che su altri sistemi operativi, come ad esempio le *jail* di FreeBSD [41] o i *containers* [42] di Solaris.
Analogamente a Docker, LXC utilizza le funzionalità *cgroups* e *namespaces* per gestire risorse e isolamento, senza bisogno di applicare patch al kernel, come accade invece su altri sistemi come OpenVZ.
LXD e Docker possono essere visti come estensioni di LXC in direzioni differenti: mentre Docker si orienta verso l'applicazione come soggetto del container, LXD si pone come obbiettivo di utilizzare la virtualizzazione a livello di sistema operativo per riprodurre intere distribuzioni Linux.
Praticamente ogni feature aggiunta da Docker a LXC è stata implementata in maniera simile anche su LXD: la scalabilità, la sicurezza, la gestione di rete e di memorizzazione persistente dei dati sono tutte caratteristiche realizzate efficacemente anche su LXD, utilizzando però tecniche differenti rispetto a Docker.
Come detto in precedenza, il rapporto tra Docker e LXD non è di concorrenza, bensì di complementarietà: mentre Docker fornisce un contesto per l'esecuzione di singole applicazioni o servizi, LXD si occupa di costruire immagini che mettano in esecuzione veri e propri sistemi operativi Linux, andando così a competere con le forme più classiche di virtualizzazione.
Se però dal punto di vista dell'architettura e funzionamento LXD e Docker condividono molti aspetti, lo stesso non si può dire per quanto riguarda le politiche di gestione delle immagini: mentre su Docker si incentiva l'utilizzo di appositi file, detti *Dockerfile*, su cui descrivere la struttura dell'applicazione che si intende generare, LXD incentiva l'utilizzo di *snapshots* (ovvero nuove immagini generate memorizzando lo stato di un container in esecuzione, oltre al template di partenza) e di *commit*(ovvero il salvataggio delle operazioni svolte sull'immagine di partenza) per tenere traccia delle operazioni svolte sul *container*.
Snapshot e *commit* permettono di risolvere il problema dei dati persistenti su LXD, ogni dato generato o modificato viene salvato una volta avviata una di queste due operazioni; la combinazione di queste due tecniche permette poi una migrazione di *containers* tra *host* connessi in rete senza bisogno di fermare i processi in esecuzione su di essi.
Per quanto riguarda l'ambito di applicazione, LXD, come Docker, trova ampio utilizzo nel mondo del *cloud* computing, soprattutto nella piattaforma OpenStack.

2.4 Strumenti per il *cloud computing*

Come esposto precedentemente, il modello di virtualizzazione a container ben si adatta al mondo del *cloud computing*: sono infatti numerose le piattaforme che integrano al loro interno un supporto diretto a *containers* di varia natura.
Durante questa sezione saranno trattate due sistemi, ProxMox [42] e OpenStack [43], per la gestione di cluster di computer: il primo è una piattaforma open source per la virtualizzazione che comprende un numero limitato di *host*; il secondo invece è un vero e proprio progetto di *cloud computing*.
L'idea dietro l'utilizzo delle due diverse piattaforme era la seguente: prima si sarebbe studiato e testato ProxMox e una volta consci di come funziona un ambiente distribuito di piccole dimensioni, si sarebbe esplorato OpenStack in tutte le sue possibilità; allo stesso modo questa sezione tratterà prima di ProxMox, in quanto sistema più approfondito tra i due e in secondo luogo di OpenStack, del quale saranno analizzate l'architettura e le funzionalità.

2.4.1 ProxMox

Può sembrare strano che tra tutte le piattaforme *cloud* disponibili, si sia scelto proprio ProxMox: quest'ultimo infatti non è una vera e propria piattaforma *cloud*, ma bensì un ambiente orientato alla virtualizzazione in ogni sua forma.
Le motivazioni che hanno spinto a questa scelta sono molto semplici: siccome occorreva fare esperienza con i concetti e il funzionamento di piattaforme *cloud*, si è scelto di utilizzare ProxMox come punto di partenza, come "demo" di un sistema più complesso.
Lo studio dei concetti, delle funzionalità e l'esperienza maturata nell'ambiente ProxMox sarebbero state una buona base per il successivo approccio a vere piattaforme *cloud* come OpenStack.
ProxMox VE nasce come progetto open source, sviluppato dall'azienda austriaca ProxMox Server Solution GmbH.
Il punto forte di ProxMox VE sta nel supporto sia della virtualizzazione classica sia ai *containers*; ciò è reso possibile dalla presenza di un kernel Linux che dispone sia di un modulo *KVM*, sia delle componenti necessarie per l'esecuzione di *containers* LXC; *KVM* [43] interagendo *QEMU* [44], un ulteriore modulo del kernel, riesce a gestire macchine virtuali con installato quasi qualunque sistema operativo, mentre LXC mette a disposizione un ambiente di esecuzione per i *containers* nativi Linux che, come detto in precedenza, costituiscono la base delle implementazioni di maggior successo per quanto riguarda la virtualizzazione a *containers*.
ProxMox è basato su una distribuzione Debian a 64 bit, su cui vengono installate le componenti necessarie al funzionamento del software in questione; una volta terminata l'installazione l'intero sistema è controllabile mediante interfaccia web,

che comunica direttamente con un *layer* di gestione col compito di esporre tutte le funzionalità necessarie per creare e gestire un'infrastruttura virtuale.
La figura 2.9 mostra una delle schermate principali dell'interfaccia web di ProxMox: partendo da sinistra è possibile notare un elenco dei nodi del sistema e di quante e quali macchine virtuali, identificate dall'icona di uno schermo, e/o *containers*, rappresentati invece come cubi, sono presenti sul nodo e quanti tra questi sono attivi(le icone che presentano una piccola freccia verde alla loro destra); la parte di destra, più grande e sostanziosa, è invece impiegata per le funzioni più svariate, dalla creazione di nuovi container e macchine virtuali alla gestione dei singoli nodi della rete; nell'esempio in questione viene impiegata per visualizzare lo stato relativo a un nodo della rete(pve-2-51), fornendo informazioni come l'utilizzo attuale di RAM e CPU.

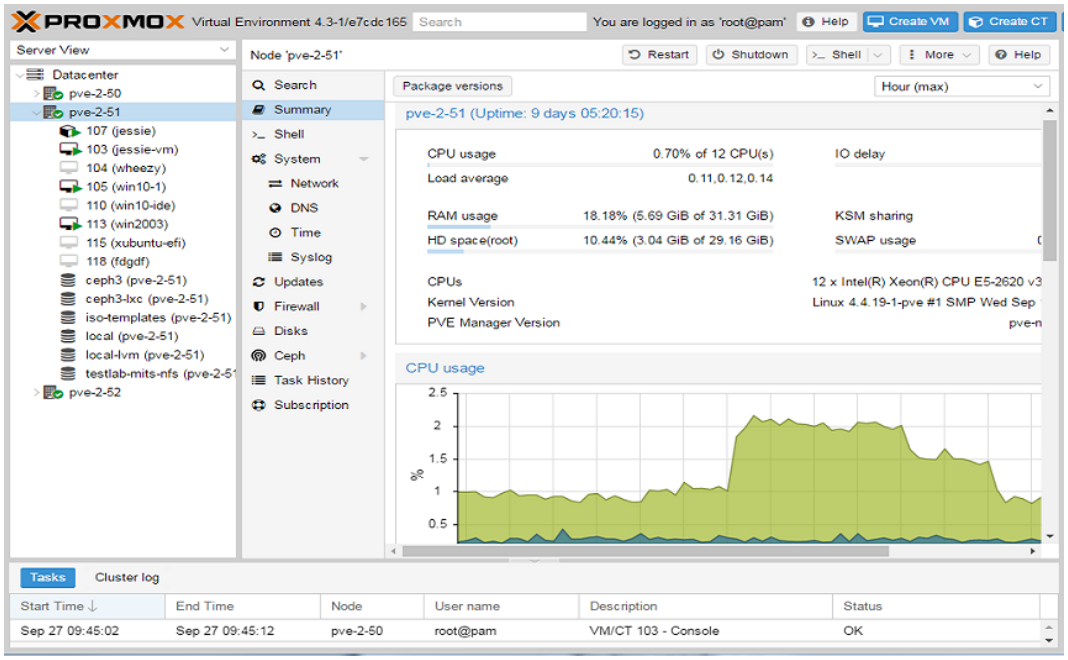

Figura 2.9: Schermata dell'ambiente di gestione ProxMox (fonte: http://www.corsinvest.it/proxmox/).

Un' altra feature fondamentale di ProxMox è la possibilità di gestire un cluster di nodi: è infatti implementata una modalità di funzionamento come *Cluster HA* (High Availabilility) [47], ovvero una configurazione, basata su un'architettura *Master – Slave* [48], il cui scopo finale è connettere i nodi in maniera tale da renderli un unico grande calcolatore a livello logico.
Con l'implementazione di un cluster di questa natura, si può bilanciare il carico di lavoro su tutti gli *host* disponibili, inoltre è garantito il supporto alla cosiddetta "*Live Migration*".
Live Migration [48] è un insieme di tecniche che permettono di muovere una macchina virtuale o container da un nodo all'altro di una rete "al volo", ovvero lo spostamento avviene in maniera trasparente rispetto ai processi del container o macchina virtuale, senza quindi comprometterne il funzionamento in alcun modo. Ciò rende ProxMox molto resistente ai malfunzionamenti: grazie al *Live Migration* è infatti possibili muovere le macchine virtuali e/o *containers* da nodi della rete che subiscono errori e successivamente ridistribuirli sulla rete, l'unica condizione

necessaria per una compatibilità totale con la migrazione in tempo reale è la presenza di uno storage condiviso, su cui far risiedere i dischi delle macchine virtuali.
Per quanto riguarda le configurazioni di rete, viene impiegato un modello chiamato "*bridge*", in poche parole tutte le macchine virtuali condividono lo stesso *bridge*, come se tutte le schede di rete fossero collegate a uno stesso switch; per quanto riguarda le comunicazioni esterne, ogni bridge è collegato con la scheda di rete dell'*host* fisico su cui il *container* o macchina virtuale è in esecuzione.
Parlando di storage, la flessibilità concessa dal sistema è molto grande: immagini di macchine virtuali e container possono essere memorizzate su singolo nodo oppure su storage condiviso, è inoltre supportato DRBD (Distribuited Replicated Block Device) [49], software specifico per la replica di device con blocchi distribuiti su diversi *host*.

2.4.2 OpenStack

Se l'esperienza con ProxMox era solamente un tutorial, OpenStack era la piattaforma *cloud* ideale per integrare il paradigma a container con tutte funzionalità di un sistema puramente orientato al *cloud* computing.
OpenStack è un sistema operativo modulare nato nel 2010 da una collaborazione tra NASA e RackSpace Cloud; scopo di questa collaborazione era creare un sistema modulare, in grado di fornire servizi e processi secondo il modello IAAS (Infrastructure as a Service).
Il successo commerciale di OpenStack è stato subito grande, grazie ad alcune scelte in fase di progettazione che si sono rivelate vincenti: da un lato la natura open source del progetto e la sua scrittura in linguaggio Python hanno attratto e mantenuto vivo l'interesse e il supporto di un'ampia comunità; dall'altro lato le funzionalità del progetto hanno attirato le attenzioni di grossi nomi dell'industria, come HP, Dell o Cisco.
Per quanto riguarda le finalità della piattaforma OpenStack, l'immagine 2.10 riassume egregiamente le principali componenti esposte agli utilizzatori: lo scheletro del sistema si compone di tre grosse funzionalità, ovvero Compute, Storage e Networking, gestite da un apposito data center e pronte per essere accedute da parte degli utenti con un'interfaccia web oppure mediante specifiche API di programmazione.

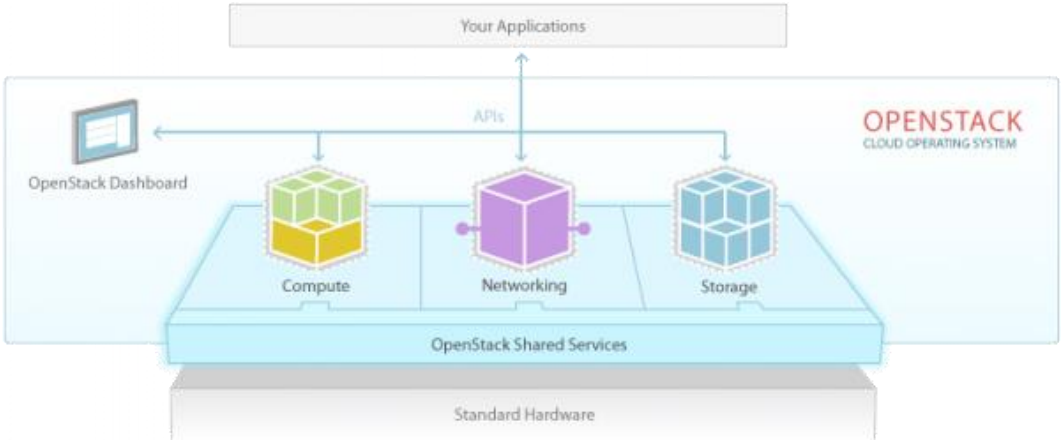

Figura 2.10: Ossatura della piattaforma OpenStack (fonte: http://www.html.it/articoli/openstack-i-moduli-e-larchitettura/)

Al fine di fornire le funzionalità delle tre componenti descritte sopra, l'architettura di OpenStack si compone di diversi moduli [52] interconnessi tra loro.
Il primo e più importante prende il nome di Nova(Compute) [52] e il suo compito è di controllare l'intera piattaforma, gestendo le diverse istanze di macchine virtuali e di comunicazione interna; i suoi compiti vengono realizzati mediante una serie di servizi specializzati, coordinati per collaborare. Tra questi servizi hanno particolare rilievo *nova-schedule*, che si occupa di allocare le risorse sulle macchine fisiche, e *nova-compute*, col compito di comunicare col l'*hypervisor* installato sulle macchine fisiche e istruirlo su come gestire le varie fasi delle macchine virtuali in esecuzione.
Secondo modulo del sistema è Neutron [52] (Networking), specializzato nelle comunicazioni di rete.
Neutron è il modulo più recente tra quelli presenti in OpenStack, in precedenza la gestione della rete era affidata a un servizio, chiamato *nova-network*, che però presentava diversi limiti, soprattutto per quanto riguarda le configurazioni avanzate di rete. Con l'introduzione di un modulo specifico, questo genere di problematiche è stato completamente risolto, visto che Neutron si occupa di gestire le comunicazioni di rete sia verso l'esterno sia verso l'interno del sistema.
Swift (Object Storage) [52] invece è il modulo che si occupa di memorizzazione dei dati: si compone infatti di un sistema distribuito di storage, pensato per l'alta affidabilità. Questo modulo dà i migliori risultati in spazi di memorizzazione molto ampli da dedicare a archiviazione dei dati, backup e altre operazioni connesse.
Per gestire problematiche di sicurezza, è messo in campo un modulo apposito chiamato Keystone (Identity) [52], che accentra su di sé tutti i processi di autenticazione e autorizzazione per i vari moduli e processi del sistema OpenStack.
La gestione delle immagini virtuali è affidata al modulo Glance (Image Service) [52], progetto a sé stante integrato su OpenStack. Glance a sua volta si compone di un database, solitamente MySQL, un servizio di categorizzazione delle immagini e un set di API per l'interazione.
Horizon (Dashboard) [52] è invece l'interfaccia web del sistema, con cui l'utente dialoga. Questo modulo è il più personalizzabile tra tutti quelli presenti su OpenStack, infatti è possibile sostituire interamente l'interfaccia di default con una completamente personalizzata dall'utente, purché integri le specifiche API di OpenStack.

Ultimo modulo del sistema è Cinder (Block Storage) [52]. Questo modulo si occupa di ciò che riguarda i volumi a blocchi per il salvataggio dei dati; analogamente a Neutron, è di recente introduzione, un tempo infatti era utilizzato un servizio di Nova, chiamato *nova-volume*, per svolgere i suoi compiti.
Terminata la descrizione di ogni modulo, è il momento di tracciare una visione d'insieme del sistema, andando a mostrare come le varie funzionalità dei singoli moduli interagiscono tra loro.
Nell'immagine 2.11 è possibile avere una rappresentazione grafica dello schema strutturale di OpenStack, è importante sottolineare che nell'immagine sono rappresentate le funzionalità espresse dai moduli, indicate nel paragrafo precedente tra parentesi, e non i nomi dei moduli.

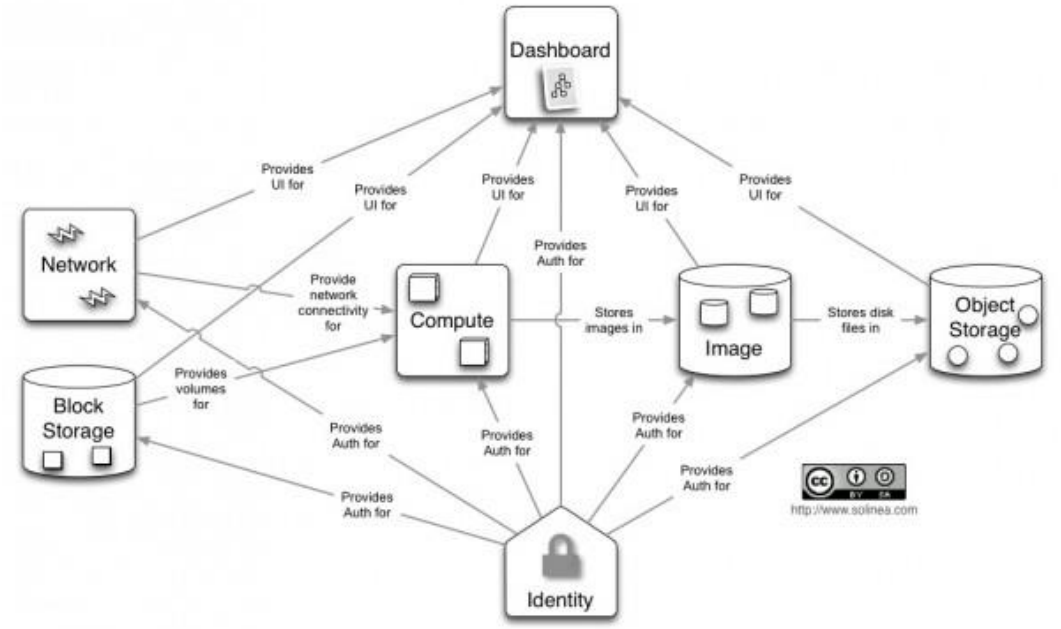

Figura 2.11: Schema delle interazioni tra le diverse funzionalità di OpenStack (fonte: http://www.html.it/articoli/openstack-i-moduli-e-larchitettura/)

Come si può notare dall'immagine, due sole componenti sono collegate a tutte le altre, ovvero *Dashboard* e *Identity*.
La ragione dietro a questa scelta risulta evidente: *Dashboard* si occupa di fare da ponte tra l'utente e il resto del sistema, di conseguenza deve poter scambiare dati con tutti gli altri moduli per poterli utilizzare o visualizzare il loro stato; dall'altra parte invece *Identity* si occupa di autenticazione e autorizzazione, funzionalità fondamentale per ogni comunicazione del sistema.
Per quanto riguarda la parte destra del grafo in immagine invece, essa è relativa alla gestione delle immagini virtuali. Tutti i dati relativi sono collocati su *Image* che utilizza *Object Storage* per la vera memorizzazione dei file.
Nel cuore del grafo si trova *Compute*, le cui funzioni di gestione e coordinamento di macchine virtuali costituiscono la base dell'infrastruttura.
Infine la parte sinistra del grafo si compone di *Networking* e *Block Storage*:
Il primo gestisce le comunicazioni di rete interne al sistema e verso il mondo esterno, mentre il secondo si occupa di fornire dei volumi per *Compute*.

Capitolo 3

Valutazione funzionale e sperimentale degli strumenti di virtualizzazione.

I capitoli precedenti avevano l'obbiettivo di fornire al lettore gli strumenti per riuscire a comprendere al meglio il contenuto di questa parte.
Ciò che verrà trattato durante questo capitolo sarà infatti il lato pratico dell'esperienza: se durante la sezione dedicata agli strumenti utilizzati si è delineata una panoramica dei suddetti approfondendo il funzionamento generale, qui si tratterà di come lo strumento è stato impiegato.
Per mantenere una coerenza con la presentazione di una problematica e la sua risoluzione, gli eventi saranno riportati in ordine temporale, cioè si descriveranno le esperienze fatte in ordine cronologico, tralasciando qualvolta qualche dettaglio nella presentazione di determinati test o eventi al fine di riprenderlo in un secondo momento.
Infine si terrà l'esposizione dei fatti più oggettiva possibile, rimandando considerazioni su eventuali alternative e sviluppi futuri in un capitolo a sé stante.

3.1 Avvicinamento ai *containers*

La prima parte dello studio di fattibilità consisteva nell'apprendimento dei concetti fondamentali della virtualizzazione a *containers*.
Essendo la prima esperienza col concetto di virtualizzazione a *containers*, si è deciso di impiegare le prime ore nell'apprendere e familiarizzare con queste nuove tecnologie; grazie alla chiara documentazione di Docker, lo studio è stato piuttosto breve e semplice.
Una volta terminato l'apprendimento delle basi dell'architettura Docker, si è scelto di procedere con un approccio *bottom-up* allo studio, alternando teoria e pratica, in modo non solo da fissare meglio i concetti appresi, ma anche di variare la routine dello studio.
Il primo problema affrontato in assoluto è stata l'installazione di Docker in ambiente Windows; cercando sulla documentazione Docker, la scelta consigliata era l'installazione dell'applicazione nativa *Docker for Windows* [53], che offriva tutto il necessario all'esecuzione di Docker su sistema operativo Microsoft.
Docker for Windows presentava però alcune problematiche: innanzitutto i primi tentavi di installazione erano falliti, causa una versione non aggiornata di Windows 10, in secondo luogo la presenza dell'applicazione avrebbe reso inutilizzabile VirtualBox, software fondamentale per le macchine di laboratorio su cui sarebbe

stato poi installato Docker nell'ipotesi di riuscita dello studio, di conseguenza si è cercata un'alternativa che non presentasse questi problemi.
Il software scelto è stato *Docker ToolBox* [54], ovvero un kit di strumenti che comprendeva *Docker CLI, Docker Machine e Docker Compose*, più che sufficienti per eseguire Docker con le funzionalità di base; il funzionamento di *ToolBox* è molto semplice: sfruttando un virtualizzatore di terze parti (VirtualBox di Oracle è quello di default, ma si può cambiare liberamente) *Docker Machine* crea una macchina virtuale Linux minimale, su cui poi vengono installati il modulo *Engine* e *Compose*.
L'immagine 3.1 mostra la schermata che si presenta all'utente una volta avviato *ToolBox*: è presente un *prompt dei comandi* nel quale inserire le istruzioni da inviare al *Docker Engine*, inoltre vengono riportati il nome e l'indirizzo IP della macchina virtuale a cui si è attualmente collegati.

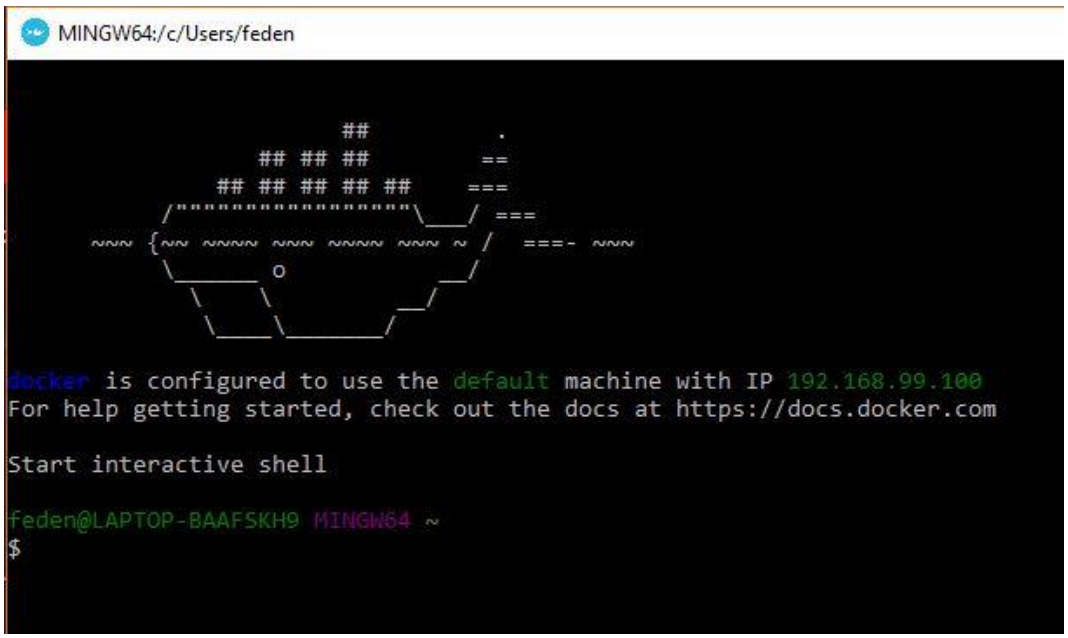

Figura 3.1: Terminale collegato alla macchina virtuale creata da Docker Machine

3.1.1 Test di immagini Docker

Una volta installato Docker, sono cominciati i primi test per comprendere al meglio l'aspetto pratico del lavoro con i *containers*.
Seguendo un'apposita guida [55] sono state testate una serie di immagini di prova fornite da Docker.
La prima tra tutte è stata "Hello-World", una semplice immagine la cui esecuzione stampava semplicemente a video la frase "Hello from Docker",
descriveva brevemente le operazioni e interazioni tra moduli che avevano portato alla comparsa di quella scritta e infine consigliava di provare l'immagine "Ubuntu".
L'immagine successiva è stata proprio "Ubuntu", che metteva a disposizione un terminale che agiva su una basilare installazione del medesimo sistema operativo.
Su quest'immagine si sono potute fare prove decisamente più serie che sulla

precedente, ad esempio sono stati testati i tempi di creazione del container e la complessità delle operazioni di recupero necessarie per ripristinarne il corretto funzionamento: durante il corso di uno di questi test, si è rimossa interamente la cartella bin e tutto il suo contenuto da un *container* creato dall'immagine "Ubuntu", eliminando così alcuni comandi fondamentali per il sistema operativo, tuttavia è bastato semplicemente mettere in esecuzione una nuova istanza dell'immagine per rimpiazzare la precedente senza nessuna procedura aggiuntiva.
Terminati alcuni semplici test come quelli descritti sopra, lo studio è proseguito verso i *Dockerfile* e la generazione di immagini personalizzate, qui si è costruita un'implementazione personalizzata su *container* di Cowsay, un semplice software che stampa a video una frase inserita da tastiera.
La personalizzazione in questione consisteva nel generare, mediante l'utilizzo di un software chiamato Fortune, una frase random che Cowsay avrebbe poi stampato automaticamente a video; questo risultato è stato ottenuto aggiungendo alcune semplici istruzioni all'immagine base di Cowsay, già presente su DockerHub, con cui veniva installato Fortune e si modificava comando di lancio di Cowsay richiamando prima il nuovo software installato.

3.1.2 Studio della *swarm mode* di Docker

Argomento successivo nei tutorial presenti sul sito di Docker era l'introduzione alla modalità *swarm*.
Nonostante le funzionalità della *swarm-mode* potessero sembrare poco pertinenti per ciò che si intendeva realizzare nello studio, si è deciso comunque di continuare a seguire i tutorial; questo per continuare a familiarizzare con l'utilizzo della piattaforma Docker e per avere una visione d'insieme su tutte le modalità di funzionamento offerte da quest'ultima.
Il tutorial in questione [56] consisteva nel realizzare un sondaggio utilizzando vari servizi: si componeva quindi di diverse parti tra cui le interfacce web per votare e accedere alle statistiche del sondaggio, diverse istanze di database *sql* e non per memorizzare i voti e infine un framework .NET per la gestione e l'esecuzione dei vari servizi web.
La procedura per realizzare questa semplice applicazione distribuita è stata la seguente: tramite *Docker Machine* si sono create più macchine virtuali, dopodiché su una di queste è stata avviata la *swarm mode*, questa istruzione ha creato un codice univoco per permettere a altri *Docker Engine* di unirsi allo *swarm* come nodi; una volta registrate sullo *swarm* tutte le macchine in questione, è stato il momento di creare i vari servizi compilando un file in formato YML presente sulla guida. Analogamente a quanto accade per i *Dockerfile*, questo file YML conteneva la descrizione di tutti i servizi da mandare in esecuzione, specificando per ciascuno porte impiegate, repliche e eventuali limitazioni hardware.
Una volta terminata la compilazione, l'applicazione e i relativi servizi sono stati distribuiti sulla rete e messi in esecuzione; si sono effettuate quindi alcune prove per testare il corretto funzionamento dell'applicazione così realizzata.

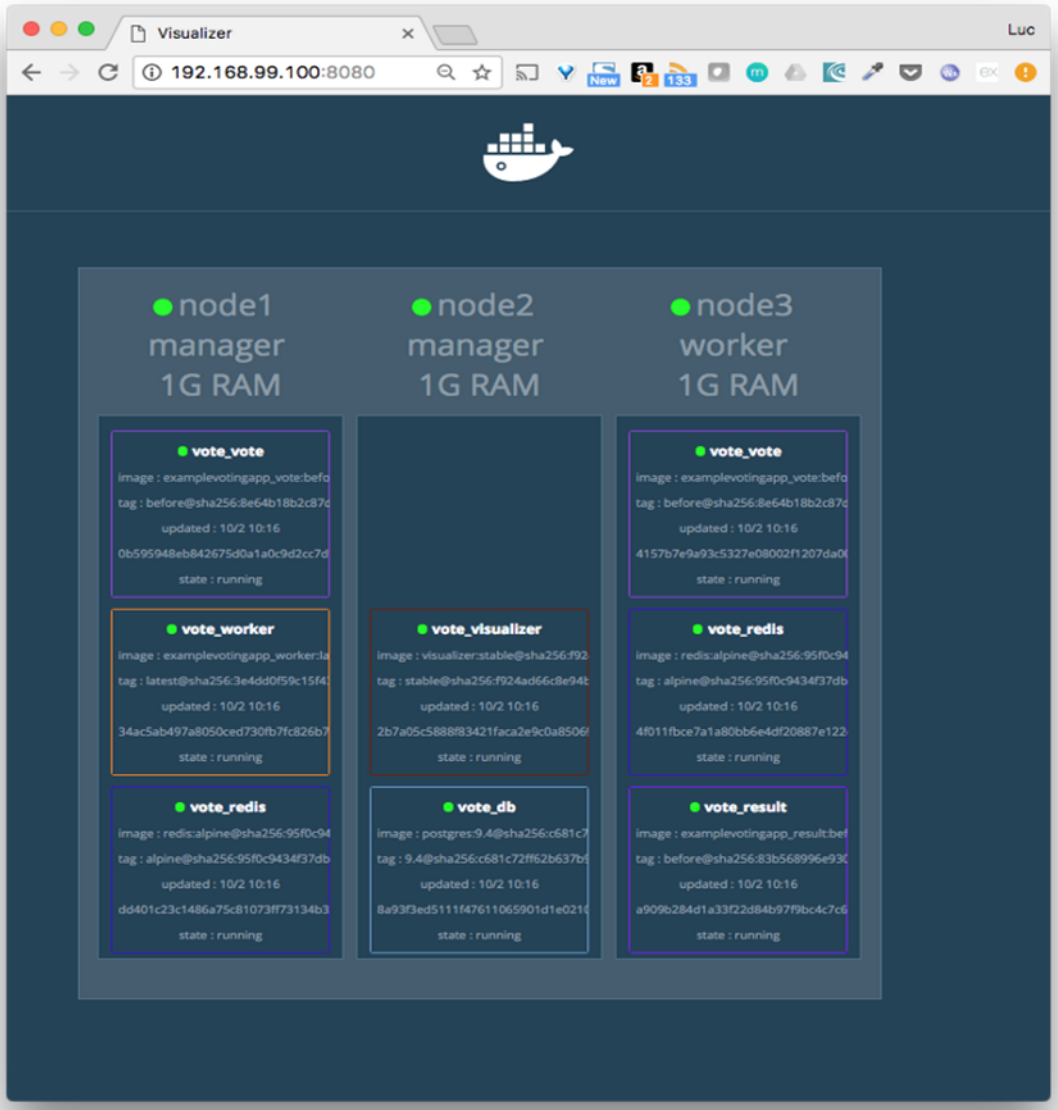

Figura 3.2: Schermata del servizio Visualizer dell'applicazione generata (fonte: https://medium.com/lucjuggery/deploy-the-voting-apps-stack-on-a-docker-swarm-4390fd5eee4)

Tra tutti i servizi integrati nell'applicazione era presente una funzionalità per fornire una rappresentazione grafica dello stato dello *swarm* tramite una pagina web; tale schermata è rappresentata nell'immagine 3.2.
Com'è possibile constatare dalla figura 3.2, il servizio fornisce un'efficiente visualizzazione di tutti i nodi dello *swarm*, indicando per ciascuno lo stato e i servizi in esecuzione su di esso; per ogni servizio poi sono indicati alcuni dati come il nome dell'immagine da cui è stato generato il *container* o lo stato attuale del servizio.
Una volta terminati i test su questa semplice applicazione distribuita, l'approfondimento della *swarm mode* è stato messo da parte, poiché si riteneva che la modalità di funzionamento classica di Docker fosse più idonea per gli obbiettivi dello studio.

3.2 Realizzazione di un ambiente desktop Linux.

Terminate le prime prove con l'ambiente Docker, si è iniziato a lavorare per raggiungere il principale obbiettivo pratico posto in fase di analisi: la costruzione di un'immagine che realizzasse un ambiente desktop Linux analogo a quello ottenuto utilizzando le più classiche forme di virtualizzazione.
L'approccio con cui si è affrontato il problema ha seguito la filosofia del "Divide et impera": dato come punto di partenza ciò che si era appreso sui *container* e le immagini utilizzate precedentemente, quali erano i problemi da risolvere per giungere al risultato finale e come potevano essere separati e affrontati nella maniera più atomica possibile?
La presenza di "Ubuntu" su DockerHub, cioè un'immagine che realizzava un'installazione base del sistema operativo Ubuntu, era un eccellente inizio; bastava semplicemente integrare su quest'immagine un'interfaccia grafica, installare alcuni software, configurare eventualmente la connessione di rete e infine trovare un modo per rendere i dati e le modifiche create permanenti.
Detto ciò, la prima cosa fatta è stata cercare se sul repository ufficiale Docker erano presenti immagini che realizzassero un ambiente Linux con un'interfaccia grafica, per comprendere in che maniera venivano risolti i problemi sopradescritti. La ricerca ha avuto fortuna: sono state trovate un paio di immagini con le caratteristiche richieste che però presentavano alcune problematiche che saranno oggetto della sottosezione successiva.

3.2.1 Visualizzazione di un ambiente grafico

Il primo problema affrontato è stata la visualizzazione dell'ambiente grafico, in quanto, nonostante la mancanza di documentazione ufficiale da parte di Docker, erano presenti già diverse immagini che implementavano interfacce utenti funzionanti.
Le immagini in questione si basavano quasi tutte sul protocollo *VNC*, utilizzando quindi la riproduzione remota per accedere alla componente video del *container*; le soluzioni messe in piedi da queste immagini erano molto differenti tra di loro, ognuna infatti utilizzava un diverso ambiente grafico (Gnome, Unity ecc..) e anche le modalità di connessione variavano tra interfacce web e *client* dedicati.
Tuttavia era presente un problema comune a tutte queste immagini: nel momento di riprodurre software con interfacce grafiche particolarmente pesanti, come ad esempio il browser web Firefox, il *container* andava momentaneamente in crash e l'applicazione in questione veniva terminata.
Visto che testare la compatibilità di queste immagini con ogni software necessario per l'ambiente desktop era un'operazione dispendiosa e poco professionale, si è cercato, con l'aiuto dei tecnici di laboratorio, un approccio diverso al problema, cercando di identificare le cause del malfunzionamento.
Considerando che tutte le immagini sopra testate utilizzavano direttamente o meno tecnologie di riproduzione remota basata sul protocollo *VNC*, si è preso in considerazione questa come la causa dei crash all'avvio di determinate

applicazioni; così si è pensato di provare un diverso protocollo di riproduzione remota, per vedere se la causa dei problemi era effettivamente *VNC*.
Sotto suggerimento dei tecnici di laboratorio, si è scelto *XRDP*, che avrebbe garantito una comunicazione più leggera e efficiente tra *client* e *container*, sul repository ufficiale Docker però non era presente nessun'immagine che realizzasse un ambiente desktop Linux e utilizzasse *XRDP*, bisognava quindi partire da zero.
Il punto di partenza era costituito dall'immagine "Ubuntu", su cui andava installato un ambiente grafico, ma quale dei tanti? Cercando risposte in rete, è stata reperita una guida [57] per configurare *XRDP* su *Ubuntu 16.04*, utilizzando *Mate* come interfaccia grafica.
Siccome la versione di Ubuntu realizzata dal container coincideva con quella della guida, si è scelto di seguire la guida e utilizzare *Mate* come ambiente grafico.
La parte di installazione di *Mate* e *XRDP* è andata a buon fine, di fatti per installarli è stato sufficiente il *package manager*, già presente nel sistema Ubuntu generato dalla corrispettiva immagine, tuttavia i primi tentativi di connessione andavano a vuoto poiché il *client* non riusciva a stabilire nessuna connessione con l'indirizzo IP del *container*.
Analizzando i file di *log* di *XRDP* all'interno del *container*, risultava poi che nessun tentativo di connessione era stato effettuato nei confronti del servizio. La soluzione al problema era molto semplice: per dimenticanza non era stato incluso nel *Dockerfile* un comando per esporre la porta utilizzata dal protocollo *XRDP*, di conseguenza ogni comunicazione aperta dal *client* venina terminata ancora prima di raggiungere il *container*.
Aggiungendo una semplice riga al *Dockerfile* dell'immagine che si stava costruendo, il problema si è risolto: quando il *client* avviava la connessione al *container*, appariva una schermata di login e una volta inserite le credenziali compariva la schermata desktop Mate, che però subiva un crash nel giro di pochi secondi.
Nuovamente si è cercato nei file di *log* una risposta, questa volta la ricerca ha avuto una maggior fortuna: sono emersi diversi messaggi di errore di avvertimento che riportavano gli eventi che avevano portato al crash.
Cercando in rete soluzioni per i problemi segnalati da quei messaggi, ne è emerso che una possibile causa del problema poteva essere la *Ubuntu 16.04*, ancora non completamente compatibile con *XRDP*.
Nonostante la suddetta ragione non convincesse completamente, si è deciso di realizzare una nuova immagine partendo dal template di *Ubuntu 14.04*, dato che era presente sul repository ufficiale.
Strano a dirsi, la riproduzione remota sui *containers* realizzati su questa nuova immagine funzionava molto bene, la sessione infatti rimaneva stabile e fluida per tutta la sua durata e anche i crash presenti su *VNC* qui erano assenti.
Discutendo coi tecnici di laboratorio, si è valutato che utilizzare *Ubuntu 14.04* rispetto al *16.04* non avrebbe compromesso eccessivamente il risultato finale desiderato, di conseguenza si è chiuso il problema della visualizzazione remota e si è passati oltre.

3.2.2 Gestione delle sessioni

Resa funzionante la riproduzione remota, restava un grande interrogativo: per salvare software installati, occorreva fare un *commit* del *container* mentre questo era in esecuzione, generando così una nuova immagine, di conseguenza come si sarebbe comportato *XRDP*? Sarebbero sorti problemi nell'avvio del servizio nei *containers* creati dalla nuova immagine?
Per dare una risposta a questi interrogativi si è optato per un approccio pratico: sono state svolte infatti alcune prove di *commit* e generazione di *containers* dall'immagine così ottenuta; questi test hanno confermato i timori sopracitati: il servizio *XRDP* nei nuovi *containers* non solo era inaccessibile da *client*, ma addirittura non rispondeva a comandi di *stop, reset* o *restart.*
Con l'aiuto dei tecnici di laboratorio, sono stati ispezionati i file collegati al servizio *XRDP*: ne è emerso che lo script che ne regolava i comandi presentava porzioni di codice malfunzionante che impedivano il corretto funzionamento del servizio. Nello specifico, la procedura di *stop*, e di conseguenza quella di *restart e reset* che includevano una chiamata alla funzione sopracitata, non eliminava il processo col compito di marcare lo stato del servizio come attivo.
Di conseguenza nel momento in cui o l'utente o il sistema operativo terminavano o riavviavano *XRDP*, invocando direttamente o meno la procedura di *stop*, il servizio rimaneva in stato attivo nonostante tutti i processi a esso dedicati fossero stati arrestati, fatta eccezione per quello che ne marcava lo stato come attivo; al momento di un nuovo avvio quindi il servizio sarebbe risultato come già in esecuzione, perciò non sarebbero stati creati i processi necessari per il suo funzionamento.
Per riuscire a risolvere questo problema, si è individuato prima il nome del processo responsabile dello stato e in secondo luogo è stato modificato lo script di *XRDP*, in modo da assicurassi che il processo responsabile del malfunzionamento fosse eliminato; una volta eseguite queste due operazioni, *XRDP* è diventato trasparente nel funzionamento rispetto alle operazioni di *commit*, ovvero una nuova immagine ottenuta dal *commit* di una precedente era in grado di ricreare i processi collegati a *XRDP* senza problemi.
Tuttavia una volta sistemata il malfunzionamento in questione, una nuova problematica è apparsa: dopo pochi minuti dalla messa in esecuzione dei *containers*, generati dai *commit* sopracitati, questi smettevano di funzionare, diventando non responsivi a input di ogni sorta, persino da terminale risultava impossibile inviare alcun tipo di comando.
La prima ipotesi è stata che la quantità di RAM allocata per i *containers* venisse completamente saturata, così si è provato ad avviare uno dei *containers* in questione e a lanciare periodicamente un comando che misurasse la quantità di RAM disponibile; ne è risultato che la memoria in questione calava a vista d'occhio, saturando completamente tutto lo spazio disponibile, di circa 1 Gigabyte, nel giro di sette secondi.
Mentre si cercava la causa di questo proliferare di processi, per coincidenza è stata pubblicata su DockerHub un'immagine per realizzare un ambiente desktop Mate sopra un'installazione di *Ubuntu 16.04* riprodotto in remoto con *XRDP* [58]; subito si è testato il suo funzionamento e si è potuto constatare che i *containers* generati

da quest'immagine non presentavano nessuna delle problematiche che affliggevano invece le immagini sviluppate fino ad ora nel corso dello studio.
Andando ad analizzare attentamente il *Dockerfile* dell'immagine funzionante ne è emerso che veniva impiegato un servizio chiamato *Supervisor* [59] per gestire *XRDP* ed evitare quindi tutte le problematiche legate agli errori di programmazione presenti nei file del servizio.
Dopo aver testato la resistenza e l'affidabilità dell'immagine tramite l'esecuzione di applicazioni graficamente pesanti e *commit* in seguito all'installazione di pacchetti, l'immagine trovata su DockerHub è stata scelta come base per il raggiungimento dell'obbiettivo dello studio di fattibilità.

3.2.3 Memorizzazione permanente e storage condivisi

Una volta realizzato un prototipo funzionante dell'immagine obbiettivo dello studio, è stato il momento di pensare a come rendere il sistema nel suo complesso il più portabile possibile.
Il problema era il seguente: posto che per mettere in esecuzione l'immagine si necessitava solamente di un *Docker Engine* installato su ogni macchina di laboratorio, era possibile separare i dati dell'utente dal container sottostante in modo da non vincolare lo studente sempre alla stessa macchina su cui aveva iniziato a lavorare?
Per riuscire a dare una risposta, si è cercato nella documentazione ufficiale Docker, apprendendo i concetti espressi nella sezione 2.2.3: tra le varie tipologie di volumi messi a disposizione, i *bind mounts* sembravano essere i più idonei per il sistema che si stava andando a costruire, poiché garantivano una maggior flessibilità rispetto alle altre soluzioni, anche se a scapito di una minor sicurezza.
La ragione principale della scelta dei *bind mounts* stava nella possibilità di eseguire operazioni di *mount* di qualunque directory del *filesystem* dell'*host*, comprese in teoria directory remote, tuttavia inizialmente l'impiego di *bind mounts* non aveva prodotto risultati buoni, dato che i primi tentativi di *mount* fallivano a prescindere dalla directory utilizzata.
Per identificare le ragioni dietro a questa problematica, si sono percorse diverse strade, nessuna delle quali ha portato a grossi passi avanti, finché non si è deciso di ripetere tutti i test in un'installazione pulita di *Docker Engine* su un sistema operativo Linux: al contrario di quanto accadeva in ambiente Windows, le operazioni di *mount* avevano successo e le modifiche all'interno della directory in questione avvenivano in tempo reale. A quel punto le ragioni del malfunzionamento erano chiare, la macchina virtuale su cui veniva messo in esecuzione *Docker Engine*, presente su Windows e non su Linux, non aveva accesso alle cartelle del *filesystem* inferiore che si tentavano di utilizzare, di conseguenza ogni operazione di *mount* con una cartella del *filesystem* Windows sarebbe fallita; occorreva dunque eseguire un'ulteriore operazione di *mount* per rendere disponibile le directory richieste all'interno del *filesystem* della macchina virtuale.

Una volta configurato l'ulteriore *mount* richiesto, le funzionalità di storage persistente hanno cominciato a funzionare anche su Windows. Successivamente si è pensato di utilizzare lo storage condiviso del campus di Cesena, che può ospitare una cartella per ciascuno studente, come supporto permanente dove salvare i dati.
L'idea per mettere assieme tutto era grossomodo questa: creare una cartella per ogni studente che frequentasse il laboratorio di Sistemi Operativi, al momento del login su macchina di laboratorio, lo studente avrebbe avviato uno script per aggiungere la sua directory remota e eseguirne il *mount* sulla macchina virtuale su cui in seguito sarebbe stato messo in esecuzione *Docker Engine*; una volta terminata l'operazione, l'utente avrebbe avviato *Docker Engine* e creato un nuovo *container* specificando l'operazione di *Bind Mount* della cartella precedentemente selezionata.

3.2.4 Creazione di un repository locale per immagini

L'ultimo passo per la conclusione dello studio di fattibilità consisteva nel creare un repository locale su cui salvare le immagini generate per il corso di Sistemi Operativi.
Il ruolo di un repository locale risultava fondamentale, dal momento che per ogni applicazione sarebbero state presenti diverse immagini: questo perché le tecniche di memorizzazione permanente funzionavano molto bene per i file o cartelle di dimensione ridotte, ma risultano inefficienti per tenere traccia di software installati, in quanto si sarebbe dovuto effettuare un *mount* sulla directory radice del *filesystem* del *container*, operazione computazionalmente molto pesante e di conseguenza non consentita dal *Docker Engine*.
Non potendo utilizzare le tecniche di memorizzazione persistente offerte da Docker, per tenere traccia dell'aggiunta o rimozione di software si doveva necessariamente usare il *commit* dei *containers* in esecuzione per generare nuove immagini, perciò, per mettere ordine alla generazione di quest'ultime, si riteneva necessario avere un repository locale, dove andare a caricare i principali template per renderli disponibili a tutti gli studenti del corso.
Questa soluzione comportava notevoli vantaggi, ad esempio si sarebbero potute preparare un certo numero di immagini comprendenti un numero crescente di software e funzionalità: ad ogni lezione di laboratorio il professore avrebbe mostrato il processo di installazione di uno o più software che gli studenti avrebbero di seguito provato a replicare nei loro *containers*; qualora fosse sorto un qualunque errore, causato dallo studente o meno, durante la procedura di installazione, la lezione sarebbe potuta procedere senza dover aspettare che lo studente ripetesse il processo di installazione, visto che sarebbe bastato scaricare un' immagine dal repository che comprendesse già il software in questione.
Date queste premesse, si è cominciata l'implementazione di questo repository, fortunatamente però gli sviluppatori di Docker avevano realizzato un semplice software [60] che metteva in esecuzione un repository locale, su cui potevano essere caricate e scaricate immagini.

Installato questo software, sono state fatte un paio di prove di upload e download di immagini sia in locale che da *host* remoto, tutte si sono concluse tutte con successo.

3.2.5 Considerazioni sul risultato ottenuto

Alla luce del lavoro descritto nelle precedenti sezioni, l'obbiettivo pratico dello studio era raggiunto, era giunto dunque il momento di riflettere sui risultati ottenuti e di analizzare se la soluzione costruita rispettava le caratteristiche pensate in fase di progettazione.
Si è fissato dunque un incontro per discutere della cosa con il professor Ghini.
Dopo una breve presentazione, il professore ha iniziato a porre domande sul funzionamento del sistema e su cosa potesse o non potesse fare; la discussione è andata avanti verificando il risultato ottenuto rispetto ai vincoli posti in fase di progettazione.
Da questa discussione è emerso che la possibilità di installare software a piacimento era un fattore molto più determinate di quanto si fosse ritenuto durante lo sviluppo: sebbene la soluzione di una serie di immagini con funzionalità incrementali potesse in parte arginare questa problematica, ciò non era sufficiente a concedere agli studenti la libertà di amministrazione sul proprio ambiente desktop di cui dovevano disporre.
Alla luce di ciò, sono state valutate possibili alternative che potessero fornire un supporto migliore per quanto riguardava la gestione delle modifiche su immagini, senza tuttavia giungere ad alcun risultato.
Per queste ragioni, la piattaforma Docker è stata ritenuta non idonea per sostituire le macchine virtuali impiegate nel corso di Sistemi Operativi.

3.3 Sperimentazione di *containers* basati su kernel Windows

Nonostante il risultato negativo ottenuto dalla piattaforma Docker, la virtualizzazione a *container* era lungi dall'aver finito le possibili applicazioni: durante lo studio infatti il tirocinante aveva avuto modo di conoscere altri sistemi basati sulla virtualizzazione a *container* diversi da Docker.
Il più interessante era costituito dai Windows containers: come già detto nella sezione 2.3.1, l'integrazione delle caratteristiche dei *containers* con i software presenti su Windows avrebbe potuto portare notevoli vantaggi nella portabilità dei software.
Alla prova dei fatti però, la piattaforma Windows containers si è rivelata molto più goffa di quanto previsto, causa il suo essere ancora acerba.
Innanzitutto per utilizzare Windows containers, occorre disporre di un'installazione di Windows Server 2016 oppure di Windows 10; a differenza di Docker qualunque

compatibilità con sistemi differenti è assente (non esiste nessun corrispondente di *Docker Machine*).
Ma la cosa che ha fatto desistere da sviluppi successivi ai primi semplici test sono state le politiche di gestione delle immagini.
La gestione delle immagini è molto simile a quella già vista su piattaforma Docker: analogamente a quanto avviene lì, anche per Windows containers sono presenti alcune immagini su cui basare tutte le altre, ma mentre su Docker è disponibile un'ampia gamma di basi, la cui dimensione si aggira sui 200Mb, Windows containers basa le sue immagini solamente su due distinti template di partenza, chiamati Server Core e Nano Server, delle dimensioni rispettivamente di 8GB e 800Mb.
Su queste immagini, già molto pesanti da sole, si sarebbero poi installati i software da mettere in esecuzione sui *containers*, incrementando ulteriormente dimensioni già considerevoli.
Oltre a ciò, a sfavore di questa piattaforma si aggiunge uno scarso supporto delle communities online per colpa della loro poca diffusione e compatibilità con sistemi diversi da Windows.

3.4 Impiego di *containers* con tecnologia alternativa a Docker

Chiusa la breve parentesi su Windows, si è tornati a lavorare con *containers* Linux, cercando di trovare alternative a Docker che implementassero un sistema migliore di salvataggio dei software installati dall'utente nei singoli *containers*.
Tra tutte le possibili soluzioni, si è scelto di utilizzare LXD per due ragioni ben definite: la prima è il supporto da parte della piattaforma *cloud* OpenStack, la seconda è la natura di alternativa alle macchine virtuali.
Difatti, se LXD si poneva come alternativa alle macchine virtuali nell'ambito di interi sistemi operativi e non solo di applicazioni, come invece faceva Docker, allora necessariamente doveva avere una progettazione senza dubbio migliore nello storage permanente non solo dei dati dell'utente, ma anche dei software installati da quest'ultimo.
L'approccio nei confronti della piattaforma è stato lo stesso tenuto con Docker, di conseguenza sono stati installati su distribuzione Linux, visto l'assente compatibilità con Windows i pochi software necessari al funzionamento di LXD e sono iniziati prove e test per comprendere meglio il funzionamento della nuova piattaforma.
Andando avanti con lo studio, era sempre più evidente la somiglianza tra LXD e Docker sia nella struttura che nell'utilizzo, l'unica differenza notevole stava nel tipo di containers realizzati dalle due piattaforme.
Questa somiglianza ha facilitato molto lo sviluppo e il testing di immagini su LXD, basti pensare che per realizzare sulla nuova piattaforma l'ambiente desktop creato in precedenza su Docker, con decine di ore di lavoro, è bastata poco più di mezz'ora.

Purtroppo le somiglianze tra le due piattaforme non sono state solo un fattore positivo: difatti al momento dello studio della memorizzazione di dati permanenti, si è scoperto che le tecniche messe in campo da LXD erano le stesse di Docker, con la sola differenza che si incentivava l'utilizzo di *commit* dei *containers* in esecuzione.
Di conseguenza il verdetto espresso su Docker veniva calato anche su LXD, tuttavia la piattaforma avrebbe potuto avere un impiego nei futuri studi su OpenStack, quindi si è scelto di accantonare momentaneamente LXD in favore delle piattaforme *cloud.*

3.5 Piattaforme *cloud* per l'integrazione di *containers*

L'ultimo argomento approfondito come appendice allo studio sui *containers* sono state le piattaforme *cloud*, su richiesta dei tecnici di laboratorio si è approfondito prima ProxMox, per cominciare a prendere confidenza con sistemi distribuiti, e successivamente OpenStack, piattaforma completa in ambito *cloud*.
Per evitare di eseguire un lavoro frettoloso, si è scelto di anteporre la qualità del lavoro alla quantità: anche qualora non si fosse arrivati a trattare a fondo ogni argomento, l'importante sarebbe stato svolgere uno studio di qualità sulle piattaforme trattate.
Questa parte dello studio è stata anche quella che ha richiesto le maggiori risorse hardware: mentre fino a questo momento era bastata una singola macchina su cui erano installate in *dual boot* Windows e Ubuntu, per utilizzare le piattaforme *cloud* in questione sono servite dalle tre alle sei macchine collegate in contemporanea, per alcune di queste poi servivano requisiti hardware molto specifici, come la presenza di due hard disk di stesse dimensioni oppure di due schede di rete.

3.5.1 Installazione e testing di ProxMox

Sin dal principio, ProxMox è stato sempre molto intuitivo nell'utilizzo anche per un utente inesperto.
Per la prima installazione è stata utilizzata una macchina con specifiche hardware nella media (una CPU Intel i-3, 4GB di RAM ddr3, 500GB di hard disk meccanico), tuttavia sufficienti a garantire una fluida esecuzione del software.
Avviata l'installazione, utilizzando un supporto usb sul quale era stata appositamente masterizzata un'immagine Debian contenente i software necessari a ProxMox, si sono configurati i vari parametri del sistema, come ad esempio l'indirizzo IP della macchina, l'accesso alla rete, le partizioni del disco, in poche parole nulla di diverso rispetto all'installazione di un ordinario sistema operativo.
Terminata l'installazione, è stata avviata la macchina in questione e sono cominciate le prime prove: all'inizio è stato testato l'accesso alla macchina via interfaccia web, che è risultato perfettamente funzionante, successivamente sono

state scaricate diverse immagini di vari sistemi operativi e *containers* per poi trasferirle su sistema ProxMox e verificarne la configurazione e funzionamento.
Inizialmente però, risultava impossibile creare qualunque sorta di macchina virtuale, mentre al contrario era possibile eseguire *containers* senza nessun problema; questo perché, come si è scoperto dai codici di errore riportati a video, il modulo *KVM*, presente nella macchina fisica, non risultava funzionante. Di conseguenza è stato ispezionato il *bios* della macchina sopracitata, scoprendo che effettivamente il modulo *KVM* era disabilitato.
Dopo aver attivato il modulo in questione, le macchine virtuali hanno iniziato a funzionare a dovere, a questo punto l'attenzione è stata concentrata sulle opzioni presenti durante la creazione di queste ultime o di *containers*.
Il processo di creazione di una nuova macchina o *container* permette infatti di specificare le caratteristiche di ogni componente hardware necessario: per le CPU si può indicare potenza massima e numero di *core*, per la RAM la quantità rispetto a quella disponibile sulla macchina fisica, si possono definire inoltre la dimensione e il *filesystem* del disco virtuale e infine la tipologia della scheda di rete.
Al termine del processo di creazione, la macchina in questione viene messa in esecuzione, per accedervi basta semplicemente cliccare su un apposito tasto nell'interfaccia, il quale aprirà un client per la riproduzione remota tramite protocollo *VNC* direttamente collegato con la macchina appena avviata; per la gestione della macchina sono disponibili tutta una serie di comandi impartibili tramite interfaccia web sulla singola macchina virtuale o *container*.
Terminata questa prima parte, il tirocinante si è proseguito con la costruzione di un piccolo cluster ProxMox: seguendo una guida trovata in rete [61], è stato configurato un secondo nodo, collegato al primo, andando così ad aumentare la disponibilità di risorse presenti nel sistema; l'unica operazione extra da compiere rispetto a una normale installazione è stata modificare il file "\etc*hosts*" in entrambi i nodi, scrivendo al suo interno gli indirizzi IP delle macchine fisiche in questione.
A questo punto mancava solo da implementare la *Live Migration* sul cluster appena costruito.
Per quanto riguarda i *containers*, nel momento in cui era stato aggiunto il secondo nodo al cluster la *Live Migration* era diventata disponibile come opzione, al contrario le macchine virtuali rimanevano ancora escluse da questa possibilità, causa l'assenza di uno storage condiviso all'interno del cluster.
Di conseguenza è stato aggiunto un terzo nodo al cluster, avente però un ruolo diverso rispetto alle altre macchine: se le altre due avevano il ruolo di nodo, questa macchina fungeva solamente da storage condiviso per il cluster, senza quindi aggiungere ulteriori risorse di RAM o CPU, ma solo capacità di storage.
Il processo di creazione di questo nodo è stato leggermente differente dai due precedenti, la base di partenza infatti era un'installazione di Ubuntu Server pulita, su cui si è creata un'apposita directory e su questa sono stati concessi ai nodi del cluster, identificandoli con i loro indirizzi IP, permessi di lettura e scrittura; dopodiché semplicemente si è registrato la macchina in questione come storage tramite l'interfaccia web del sistema, rendendola così disponibile come supporto di memorizzazione per ogni nodo presente sul cluster.
Una volta aggiunto questo storage condiviso, la *Live Migration* era finalmente disponibile e funzionante anche per le macchine virtuali.

Sono effettuati diversi test sulle funzionalità della *Live Migration*: nei primi semplicemente si provava il suo funzionamento spostando da un nodo all'altro macchine virtuali e *containers* di diversa natura; nei test successivi invece si andava a verificare le interazioni della *Live Migration* con applicazioni in esecuzione.
Il test più rilevante tra tutti è stato condotto su una macchina virtuale Linux, su cui è stato lanciato un comando di *ping* verso un server e poi avviata la procedura di *Live Migration* su un altro nodo della rete, l'efficienza del processo sarebbe stata misurata dal numero di pacchetti persi durante l'operazione di spostamento; nonostante questo test sia stato ripetuto più volte, in nessuna occasione è stato registrato neanche un pacchetto perso, di conseguenza si può affermare che l'efficienza della *Live Migration* su ProxMox sia consona alle aspettative.

3.5.2 Installazione di OpenStack

Terminata l'esperienza con ProxMox, i tempi erano maturi per iniziare le ricerche su OpenStack.
Sin dalle prime fasi degli studi, è stato evidente che OpenStack era molto più complesso di tutte le piattaforme precedentemente utilizzate: già dalle ricerche su come creare un cluster OpenStack, i pochi risultati rinvenuti richiedevano conoscenze di concetti collegati all'ambiente *cloud* molto vaste; tra questi la meno complessa era una guida presente sulla documentazione ufficiale Ubuntu.
La guida in questione [62] poneva due possibili opzioni: un'installazione demo, utilizzando *containers* LXD, e un'installazione completa di tutta la piattaforma, sfruttando un apposito software chiamato Autopilot, che avrebbe automatizzato il più possibile il processo di installazione; dopo un'attenta valutazione, si è scelto di tentare l'installazione completa.
Avendo scelto l'opzione più complessa, si è cominciato col reperire l'hardware richiesto, occorrevano infatti un minimo di sei macchine fisiche, di cui almeno tre dovevano avere almeno due dischi fisici e una due schede di rete; oltre a questo serviva uno switch dedicato connesso alla rete internet.
Una volta preparato l'hardware, si è dato il via al processo di creazione del cluster OpenStack.
Primo passo consisteva nell'installazione di *Ubuntu Server 16.04* su una delle macchine del sistema e nella configurazione su di esso di MAAS [63], uno strumento per l'automazione dei server fisici; una volta terminata la sua installazione, MAAS è stato propriamente configurato, specificando credenziali di login, *dns*, *gateway* e parametri di una sottorete virtuale per fornire le funzionalità DHCP ai nodi registrati sulla piattaforma.
Successivamente è iniziata la fase di registrazione dell'hardware su MAAS e qui sono cominciati i problemi: innanzitutto su ogni macchina andava impostato il *PXE boot*, successivamente ogni computer veniva acceso manualmente e cominciava la registrazione su server MAAS, che avrebbe poi portato a rendere la macchina in questione avviabile da *boot* remoto.
Nella fase di registrazione sorgeva un problema considerevole, ovvero per avviare le macchine da posizione remota occorreva un apposito standard Ethernet, tuttavia

mentre i computer supportavano tutti *Wake on Lan* [64], uno degli standard più diffusi, MAAS metteva a disposizione tutta una serie di protocolli differenti, di cui nessuno era però compatibile con le macchine in questione.
Dopo un considerevole numero di tentativi per trovare una risposta a questo problema, si è deciso di accantonare momentaneamente il problema, poiché tra le varie opzioni di *boot* remoto di MAAS era presente anche un'alternativa manuale e, visto che la guida non specificava che protocollo utilizzare, si è scelto di utilizzare quella come soluzione temporanea per terminare la procedura in questione.
Una volta registrate tutte le macchine, si poteva finalmente partire con l'installazione vera e propria, tramite il pacchetto *Conjure-Up*, punto di partenza di Autopilot, il software che avrebbe fatto il grosso del lavoro.
Dopo aver iniziato la configurazione dell'ambiente con *Conjure-Up*, il sistema registrava un freeze subito dopo aver inserito l'indirizzo IP e la chiave API del server MAAS.
Vano è stato ogni tentativo di trovare una soluzione: nulla era presente nei file di log delle varie macchine e neppure in rete risultavano problemi analoghi, è stato più volte riavviato il processo, anche eliminando ogni dato e ricominciano l'intera installazione da capo, ma nonostante tutto questo, nessun cambiamento è avvenuto, l'installazione ha continuato a bloccarsi nello stesso punto.
Alla luce di questa nuova problematica si è discusso ampiamente della cosa e ne è emerso un errore di valutazione da parte di tutti nella complessità della piattaforma OpenStack e si è valutato di concludere lì lo studio, lasciando a un nuovo studio o ricerca il compito di portare avanti l'installazione della piattaforma OpenStack.

Capitolo 4

Conclusioni e sviluppi futuri

In questo ultimo capitolo sono raccolte le considerazioni sull'esperienza vissuta nel suo complesso, comprendente quindi studio sui *containers*, realizzazione ambiente desktop Linux e infine esperienza con le piattaforme *cloud*.
Per iniziare si traccerà un'analisi di pregi e difetti successivamente si tratterà dei possibili sviluppi futuri delle ricerche, indicando quali piattaforme tra quelle trattate hanno, alla luce di ciò che si è appreso, maggiori possibilità di evoluzione e in quali direzioni.

4.1 Analisi dell'esperienza

Dal punto di vista degli obbiettivi fissati in fase di analisi e progettazione, lo studio di fattibilità è stato condotto nella giusta maniera, il suo scopo infatti non era sostituire le macchine virtuali, ma valutare se una soluzione alternativa basata su *containers* potesse eventualmente sostituirle.
La differenza tra i due concetti è sottile: in un caso si parla di cercare un'alternativa a prescindere dalla soluzione impiegata, dall'altra parte invece si pone come vincolo l'utilizzo di una tecnologia specifica e si valuta se la soluzione è peggiore o migliore di quella attuale.
Di conseguenza, la risposta negativa allo studio di fattibilità assume un altro significato, ovvero non rappresenta un fallimento, ma al contrario un nuovo punto di partenza per futuri sviluppi.
Guardando il lavoro svolto col senno di poi si può notare un importante errore concettuale, che è sfuggito in fase di progettazione, causa la mancata conoscenza del sistema con cui si sarebbe andati a lavorare: Docker infatti è una piattaforma che fornisce un contesto statico per l'esecuzione di applicazioni, non prevede la possibilità di cambiare dinamicamente il contesto stesso.
Un'applicazione ha tempi e modalità di sviluppo completamente diversi da un sistema operativo, di conseguenza le modalità pensate da Docker per la gestione delle immagini si sposano molto bene con lo sviluppo di applicazioni o servizi, per ogni versione dell'applicazione infatti si può predisporre un'immagine diversa non modificabile, in modo da garantire il funzionamento pensato dallo sviluppatore.
Un sistema operativo invece richiede di poter essere personalizzato da un utente molto più di una normale applicazione, installando software e modificando configurazioni.
Alla luce di ciò si può affermare che Docker durante lo studio di fattibilità sia stato usato in maniera impropria, cioè utilizzato per uno scopo diverso da quello per cui era stato pensato, perciò risulta naturale che la soluzione costruita presentasse limiti così marcati rispetto ad alcune caratteristiche di fondamentale importanza, per quanto su molti altri aspetti migliorasse oggettivamente il sistema a macchine virtuali.

Per quanto riguarda le altre parti dell'esperienza, ovvero lo studio delle altre implementazioni di *containers* e le piattaforme *cloud*, il bilancio dell'esperienza rimane positivo, sebbene anche qui il riscontro avuto con l'utilizzo pratico differisse parecchio dalle potenzialità teoriche.
Windows containers si è rivelata una piattaforma ancora acerba, con ancora molte potenzialità inespresse, al contrario LXD ha rivelato di avere solide basi, tuttavia entrambe hanno ancora un grosso limite, ovvero la compatibilità con più sistemi operativi, che vincola non poco i loro ambiti di applicazione.
Il lato *cloud* è probabilmente la parte dell'esperienza con risultati pratici meno importanti, su ProxMox non ci si è soffermati troppo, mentre per quanto riguarda OpenStack non si ha avuto modo di testare nessuna delle funzionalità presenti, causa il fallimento dell'installazione; l'unico risultato concreto ottenuto in questa sezione può considerarsi la teoria appresa in ambito di servizi *cloud* e architetture distribuite.

4.2 Sviluppi futuri

Guardando al futuro, i *containers* sembrano acquisire sempre più importanza nell'industria informatica, analogamente cresce il numero delle possibili integrazioni di questa tecnologia.
La prosecuzione più naturale e immediata agli studi svolti consiste nel riprendere la piattaforma OpenStack e terminarne con successo la procedura di installazione, riuscendo così a esplorare nella pratica tutte le possibilità presenti su carta.
Altra direzione di interesse sta nei *Windows containers*: nei prossimi anni infatti potrebbero esserci evoluzioni interessanti nella piattaforma, tali da avvicinarsi alle ipotesi di compatibilità quasi assoluta teorizzate durante il corso dell'esperienza.
Ma la piattaforma in cui si nutrono più aspettative è Docker.
Nel giro di soli quattro anni di vita, Docker si è imposta come leader nello sviluppo di applicazioni su *containers*, sbaragliando ogni forma di concorrenza, grazie a una tecnologia intuitiva nell'utilizzo, progressiva nell'apprendimento e alla portata di veramente tutti, dallo sviluppatore esperto al novellino alla sua prima applicazione.
Docker evolve con una tale rapidità da lasciare a bocca aperta: si pensi che durante il corso dell'esperienza svolta, si è tenuta l'edizione annuale della conferenza chiamata DockerCon a Austin, Texas, che ha rimescolato parecchio le carte in tavola sulle direzioni future della piattaforma.
La quantità di novità presentate a questa conferenza è stata tale da riscrivere una buona parte della documentazione Docker: ad esempio le guide iniziali utilizzate per l'apprendimento dei concetti base sono state riscritte completamente, poiché erano stati aggiunti nuovi strumenti per migliorare l'apprendimento del concetto di *container* e della sua implementazione Docker.
Di DockerCon vengono tenute due edizioni all'anno e in ciascuna di esse vengono presentate sostanziose novità: *swarm mode* e *Windows containers* sono solo due delle tante tecnologie presentate negli anni a DockerCon.
Oltre al continuo aggiornamento e miglioramento delle funzioni da parte di sviluppatori ufficiali, Docker può contare inoltre su una *community* molto attiva che

produce a sua volta nuove funzioni e/o integrazioni di tecnologie con Docker e viceversa.

Bibliografia

[1]. https://www.docker.com/what-container.
[2]. https://www.vmware.com/solutions/virtualization.html
[3]. https://www.slideshare.net/PraveenHanchinal/virtualizationthe-*cloud*-enabler-by-inspiregroups/7-Timeline_of_Virtualization_1970s_virtual
[4]. http://searchservervirtualization.techtarget.com/definition/hypervisor
[5]. http://www.linfo.org/user_space.html
[6]. http://www.linfo.org/kernel_space.html
[7]. https://linuxcontainers.org/it/lxc/introduction/
[8]. https://linuxcontainers.org/it/lxd/introduction/
[9]. https://en.wikipedia.org/wiki/Cgroups
[10]. https://en.wikipedia.org/wiki/Linux_namespaces
[11]. https://www.docker.com/what-docker
[12]. https://docs.microsoft.com/en-us/virtualization/windowscontainers/about/
[13]. https://italy.emc.com/corporate/glossary/platform-as-a-service.htm
[14]. https://blog.docker.com/2014/03/docker-0-9-introducing-execution-drivers-and-libcontainer/
[15]. http://unionfs.*filesystem*s.org/
[16]. https://www.datadoghq.com/docker-adoption/
[17]. https://docs.docker.com/engine/docker-overview/#docker-engine
[18]. https://docs.docker.com/engine/docker-overview/#the-docker-daemon
[19]. https://docs.docker.com/engine/docker-overview/#the-docker-client
[20]. https://docs.docker.com/engine/docker-overview/#docker-registries
[21]. https://www.ilsoftware.it/download.asp?tag=virtualbox_996
[22]. https://docs.docker.com/machine/overview/
[23]. https://docs.docker.com/compose/overview/
[24]. http://yaml.org/
[25]. https://docs.docker.com/engine/swarm/
[26]. https://docs.docker.com/engine/swarm/key-concepts/#what-is-a-node
[27]. https://docs.docker.com/engine/swarm/key-concepts/#services-and-tasks
[28]. https://docs.docker.com/engine/docker-overview/#docker-objects
[29]. https://docs.docker.com/engine/reference/builder/
[30]. https://en.wikipedia.org/wiki/Virtual_Network_Computing
[31]. https://docs.kde.org/trunk5/it/kdenetwork/krdc/what-is-RFB.html
[32]. https://wiki.ubuntu-it.org/InternetRete/DesktopRemoto/Xrdp
[33]. https://support.microsoft.com/it-it/help/186607/understanding-the-remote-desktop-protocol-rdp
[34]. https://docs.docker.com/engine/tutorials/networkingcontainers/
[35]. https://docs.docker.com/engine/admin/volumes/
[36]. https://docs.docker.com/engine/admin/volumes/volumes/
[37]. https://docs.docker.com/engine/admin/volumes/bind-*mount*s/
[38]. https://docs.docker.com/engine/admin/volumes/tmpfs/
[39]. https://docs.microsoft.com/en-us/virtualization/windowscontainers/manage-containers/hyperv-container
[40]. https://docs.docker.com/docker-for-windows/

[41]. https://blog.docker.com/2016/09/dockerforws2016/
[42]. https://www.freebsd.org/doc/it_IT.ISO8859-15/books/handbook/jails.html
[43]. http://www.oracle.com/technetwork/server-storage/solaris/containers-169727.html
[44]. http://www.corsinvest.it/proxmox/
[45]. https://www.openstack.org/software/
[46]. https://www.linux-kvm.org/page/Main_Page
[47]. https://it.wikipedia.org/wiki/QEMU
[48]. http://searchdisasterrecovery.techtarget.com/definition/high-availability-cluster-HA-cluster
[49]. https://it.wikipedia.org/wiki/Architettura_master-slave
[50]. https://pve.proxmox.com/wiki/Backup_-_Restore_-_Live_Migration
[51]. https://en.wikipedia.org/wiki/Distributed_Replicated_Block_Device
[52]. http://www.html.it/articoli/openstack-i-moduli-e-larchitettura/
[53]. https://docs.docker.com/docker-for-windows/
[54]. https://docs.docker.com/toolbox/overview/
[55]. https://docs.docker.com/get-started/
[56]. https://medium.com/lucjuggery/deploy-the-voting-apps-stack-on-a-docker-swarm-4390fd5eee4
[57]. http://c-nergy.be/blog/?p=8952
[58]. https://hub.docker.com/r/rigormortiz/ubuntu-xrdp-mate/
[59]. http://supervisord.org/
[60]. https://blog.docker.com/2013/07/how-to-use-your-own-registry/
[61]. https://www.howtoforge.com/tutorial/how-to-configure-a-proxmox-ve-4-multi-node-cluster/
[62]. https://www.ubuntu.com/download/*cloud*/autopilot
[63]. https://maas.io/
[64]. https://en.wikipedia.org/wiki/Wake-on-LAN