POLITECNICO DI TORINO

Corso di Laurea in Ingegneria Informatica

Tesi di Laurea Magistrale

Automazione e creazione di ambienti in Cloud attraverso l'uso di Container Docker

Relatore
prof. Fulvio Risso

Candidato
Luca Vacchetta

Tutor aziendale
Manager di Blue Reply srl
dott. Guido Vicino

Ottobre 2017

†

Questo mio importante traguardo lo voglio principalmente dedicare a mia madre, che sicuramente da lassù ha sempre fatto il tifo per me.

Ringraziamenti

Un pensiero ed una dedica a tutti coloro i quali mi sono stati vicino in questi anni di studi:

- a mio padre e alla mia famiglia, che mi hanno sempre supportato, assecondando i miei desideri ed interessi - non solo in campo universitario - e contribuendo in maniera essenziale, sia moralmente che concretamente, alla loro realizzazione;
- ai miei amici storici compagni di mille uscite e bevute tra cui: mio fratello Alex, Dani, Robi (il commercialista), Calca, Fabri;
- agli amici amanti della musica e dei balli occitani con i quali ho partecipato ad un sacco di concerti in tutto il Piemonte e non solo, nonché compagni di svariate camminate per le valli cuneesi, vale a dire un grazie va a: Alessandra, Manuel, Lorenzo, Simone, Valeria, Anna, Robi;
- ai compagni ed amici di studi conosciuti qui al Politecnico: Simone, Luca, Gianluca, Nicola, Simon, Thomas;
- alle compagne di ballo in quel di Torino: Veronica, Emma e Miriam;
- ai colleghi di lavoro di Blue sempre pronti ad aiutarmi nei momenti più critici della tesi, quindi un grosso grazie va a: Guido, Marco, Fabio, Davide Sarais, Angelo, Davide Massano, Davide Circhirillo, Nicolò, Dario e Thomas.

Indice

Elenco delle figure v

1 Introduzione 1

1.1 Perché JMeter in multicloud 1

1.2 Stato dell'arte e obiettivi attesi 2

1.2.1 Al giorno è possibile deployare JMeter in multicloud? 2

1.2.2 Obiettivo della Tesi 2

2 Docker 5

2.1 Confronto tra Containers e Virtual machines 6

2.1.1 Containers 7

2.1.2 Virtual machine 7

2.2 Componenti principali 8

2.2.1 Docker Engine 8

2.2.2 Docker Client 9

2.2.3 Docker Image e Docker Container 10

2.3 Dockerfile 11

2.4 Network 12

3 Kubernetes 14

3.0.1 Perché si è scelto di gestire i containers? 15

3.1 Elementi principali 16

3.1.1 Master 17

3.1.2 Node 19

3.2 Concetti fondamentali . . . 22
3.2.1 Pod . . . 22
3.2.2 Perché mettere più containers all'interno di un Pod? . . . 22
3.2.3 Deployment . . . 24
3.2.4 Service . . . 24
3.2.5 HPA (Horizontal Pod Autoscaling) . . . 25
3.2.6 Rolling update . . . 26

4 Terraform 31
4.1 Caratteristiche principali . . . 31

5 Architettura di JMeter 33
5.1 Overview . . . 33
5.1.1 Premessa . . . 33
5.2 Progettazione . . . 34

6 Implementazione di JMeter 38
6.1 Overview su ETCD . . . 38
6.2 JMeter master . . . 39
6.2.1 Dockerfile . . . 39
6.2.2 Alias di JMeter . . . 42
6.2.3 Script di entrypoint . . . 43
6.3 JMeter slave . . . 44
6.3.1 Dockerfile . . . 44
6.3.2 Script di entrypoint . . . 46
6.4 Conclusioni . . . 51

7 Kubernetes in multicloud 53
7.1 Configurazione . . . 53
7.1.1 Master su Softlayer . . . 54
7.1.2 Slave su Softlayer . . . 60
7.1.3 Slave su AWS . . . 61

8 Deploy di JMeter in multi-cloud 66

8.1 Elementi fondamentali 66

8.1.1 Secret 67

8.1.2 Volume 67

8.1.3 Deployment di JMeter master 69

8.1.4 Deploy di JMeter slave su Softlayer 71

8.1.5 Deploy di JMeter slave su AWS 73

9 Conclusioni 74

9.1 Vantaggi 74

9.1.1 Terraform 74

9.1.2 Kubernetes 76

9.1.3 Docker 78

9.2 Caso d'uso 78

9.2.1 Setup 78

9.2.2 Creazione del cluster di Kubernetes 80

9.2.3 Deploy di JMeter su Kubernetes 81

Bibliografia 84

Elenco delle figure

1.1 Schema d'uso del lavoro di Tesi 4

2.1 Schema di principio di Docker (fonte: docker.io). 6
2.2 Schema di principio dei Containers (fonte: docker.io). 7
2.3 Schema di principio delle VMs (fonte: docker.io). 8
2.4 Archittetura di Docker 9
2.5 Interazione tra containers ed immagini (fonte: docs.docker.com). . . 10
2.6 Passaggio dal Dockerfile all'immagine. 12

3.1 Schema dell'uso di docker in Kubernetes(fonte: Kubernetes). 16
3.2 Funzionamento di Flannel (fonte: chunqi.li). 18
3.3 Schema logico di comunicazione in Kubernetes. 18
3.4 Funzionamento del kube-proxy. 20
3.5 Proxy-mode in userspace (fonte: kubernetes.io). 20
3.6 Proxy-mode con iptables (fonte: kubernetes.io). 21
3.7 Esempio di Pod con più containers al suo interno(fonte: Kubernetes). 23
3.8 Stato iniziale del cluster 27
3.9 Richiesta di rilascio di una nuova versione 27
3.10 Sostituzione primo Pod 28
3.11 Sostituzione secondo Pod 28
3.12 Sostituzione terzo Pod 29
3.13 Sostituzione quarto Pod 29
3.14 Stato finale del cluster 30

5.1 Funzionamento logico di JMeter 34
5.2 Creazione cluster ETCD 34
5.3 Lancio del primo slave 35
5.4 Lancio del secondo slave 35
5.5 Lancio del terzo slave 36
5.6 Rimozione del primo slave 36
5.7 Rimozione del secondo slave 37
5.8 Rimozione del terzo slave 37

6.1 Tempistiche di JMeter nel caso in cui venga lanciato prima lo slave che il master 49
6.2 Schema di funzionamento della chiusura di un Pod 51

7.1 Grafo delle risorse creato da Terraform 54
7.2 Schema logico etichettamento nodi 58

8.1 Schema di principio 66

9.1 Screenshot che indica il successo riguardante la creazione delle risorse 81

Capitolo 1

Introduzione

Il lavoro di Tesi si concentra principalmente su due aspetti:

- Lo studio delle nuove tecnologie (Docker, Kubernetes e Terraform) con il fine di apprendere appieno i vantaggi derivati dal loro utilizzo e prendere confidenza con esse.
- A questo punto una volta appreso il potenziale di queste tecnologie, si passa alla parte pratica suddivisa in questi macroblocchi:
  - Creazione dei vari servers virtuali su differenti cloud providers, usando Terraform.
  - Installazione e configurazione di Kubernetes sui servers specificati al punto precedente, anche questa fase viene automatizzata mediante Terraform.
  - Progettazione ed implementazione dei containers di JMeter (master e slave), utilizzando la tecnologia Docker.

In particolare si è deciso di implementare e deployare i containers di JMeter in multicloud.

1.1 Perché JMeter in multicloud

Genericamente il multicloud può essere utilizzato per risolvere una problematica che si pone al giorno d'oggi, vale a dire: dato che spesso le aziende hanno la necessità

di esporre su Internet servizi di e-commerce/internet-banking, ad esempio, che devono essere raggiungibili a livello globale, affinché di tali servizi abbiano successo è fondamentale che il tempo di risposta sia il più breve possibile, quindi dato che non tutti i cloud provider hanno datacenter in tutte le nazioni (ad esempio: AWS non ha un datacenter in Francia, IBM non ne ha uno dietro il grande firewall cinese) per offrire un buon servizio bisogna avvalersi di più cloud provider.
Durante il lavoro di Tesi si è deciso di deployare JMeter (che è un'applicazione avente il compito di effettuare dei test di carico o stress-test a degli applicativi, non necessariamente applicazioni web, con il fine di misurarne le performance) in multicloud, appunto per ricavarne delle misure che siano le più precise possibile. Questo perché, intuitivamente, se le componenti che eseguono i tests li localizziamo nel luogo in cui gli utenti risiedono nella realtà, si otterranno, ovviamente, delle misure più realistiche, cioè il tempo di latenza sarà più attendibile.
Un altro vantaggio nell'utilizzare cloud providers diversi risiede nel fatto che i tests vengono eseguiti da servers con indirizzi IP, che ragionevolmente, sono completamente diversi. Per cui si ottengono delle misure ancora più realistiche, in quanto spesso i servers a cui sono sottoposti gli stress-test sono posizionati dietro un load balancer che il più delle volte indirizza il traffico ad un server anziché un altro solamente per affinità dell'indirizzo IP.

1.2 Stato dell'arte e obiettivi attesi

1.2.1 Al giorno è possibile deployare JMeter in multicloud?

Ovviamente sì, solamente che questa operazione deve essere effettuata manualmente, per cui prima di avere l'ambiente pronto per l'utilizzo sono necessarie diverse ore di lavoro. Inoltre non è scalabile in quanto nel momento in cui si vogliano aumentare il numero di slave di JMeter queste repliche vanno installate e configurate manualmente, quindi il tempo di deploy aumenta in modo proporzionale al numero di repliche desiderate.

1.2.2 Obiettivo della Tesi

L'obiettivo della Tesi è appunto quello di velocizzare questa operazione, infatti come verrà poi riportato nella conclusione si passerà ad ottenere il sistema completamente funzionante in pochi minuti.
Inoltre dato che le fasi di:

- Creazione servers.
- Installazione e configurazione di Kubernetes.
- Deploy di JMeter sul cluster di Kubernetes.

sono pressoché automatizzate (come schematizzato nella figura 1.1) si eviterà di cadere in banali errori di distrazione, che però non sono così banali da scovare.

Infine anche la sua gestione sarà molto semplificata, in quanto, digitando semplicemente dei comandi, saranno possibili le segunti operazioni:

- Monitorare quanti slave di JMeter sono presenti ed in quale cloud provider.
- Aggiungere/rimuovere slave di JMeter da qualsivoglia cloud provider.
- Lanciare gli stress-test su tutti gli slave presenti, senza preoccuparsi di recuperare gli indirizzi IP degli slave stessi.

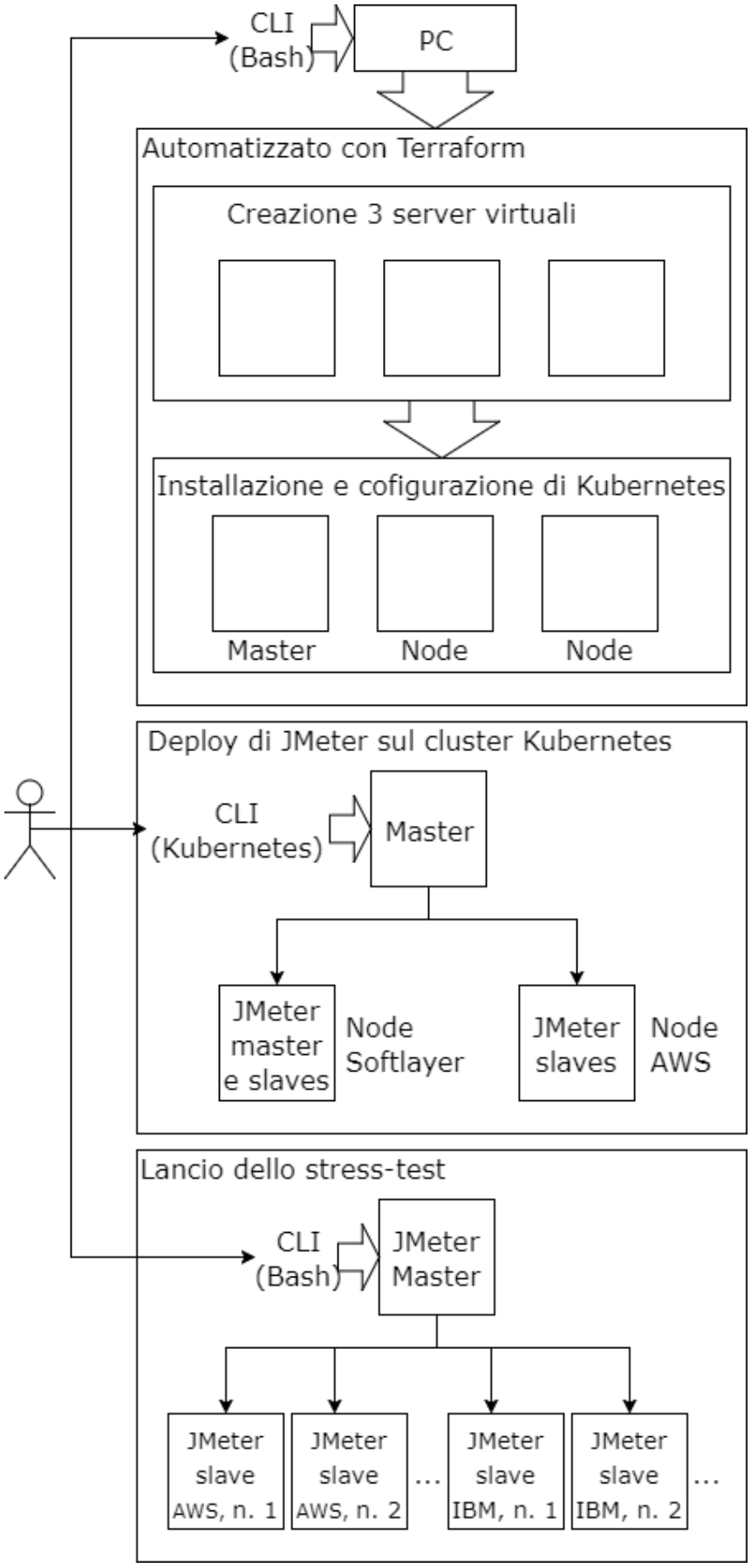

Figura 1.1. Schema d'uso del lavoro di Tesi

Capitolo 2

Docker

Docker è la piattaforma leader mondiale per la creazione di container.
In particolare gli sviluppatori utilizzano Docker per eliminare il problema "ma sul mio pc funziona", invece in ambiente di produzione no. Le aziende lo usano invece per creare software agile, vale a dire consegnare nuove versioni di applicazioni sia per Linux che per Windows Server in modo veloce e sicuro.
Più precisamente Docker è basato sui containers, infatti, fa da "wrapper" attorno a Linux Containers (per questo a volte è chiamato LXC).
In pratica Linux containers crea un'ulteriore strato di virtualizzazione al di sopra del sistema operativo. In breve, i sistemi operativi sono virtualizzati in modo tale da ottimizzare l'utilizzo delle risorse, questo fa sì che nella maggior parte del tempo l'hardware resti libero, fanno eccezione soltanto alcuni momenti in cui sono presenti dei picchi dovuti all'utilizzo di applicazioni.
Come risultato al giorno d'oggi si possono lanciare molti sistemi operativi virtualizzati su di uno stesso host. E con il passare del tempo questi sistemi operativi hanno al loro interno un numero sempre più elevato di funzionalità, pertanto nella maggior parte dei casi molte di queste funzionalità sono inutili per l'applicazione che ci gira su di esso; quindi cresce sempre più la necessità di eliminare dal sistema operativo queste funzionalità affinché l'applicazione desiderata risulti più "leggera".
Qui entrano in gioco i containers in quanto la loro filosofia è appunto quella di installare SOLAMENTE ciò che è realmente necessario per l'applicazione o il processo desiderato.
In sostanza Docker mette a disposizione un wrapper "git like" che permette in modo semplice la creazione dei Linux Containers ed il loro monitoraggio più user friendly.
Questi sono alcuni degli aspetti fondamentali di Docker:

- Leggera: in quanto più containers girano sulla stessa macchina condividendo il

kernel del sistema operativo, di conseguenza il loro avvio è più veloce perché richiede mena CPU e RAM. Inoltre le immagini hanno la forma di un file-system stratificato per cui i file comuni vengono condivisi minimizzando l'occupazione del disco e di conseguenza il tempo di download di un'immagine è più veloce (perché vengono scaricati solo gli strati che non si hanno localmente).

- Indipendente: in quanto pur variando il sistema operativo sottostante i container sono sempre gli stessi, perché si basano su degli standard.
- Sicura: in quanto i containers sono isolati sia l'uno dall'altro e sia dall'infrastruttura sottostante; per cui delle problematiche su di un container non si ripercuotono sugli altri container e il resto della macchina.

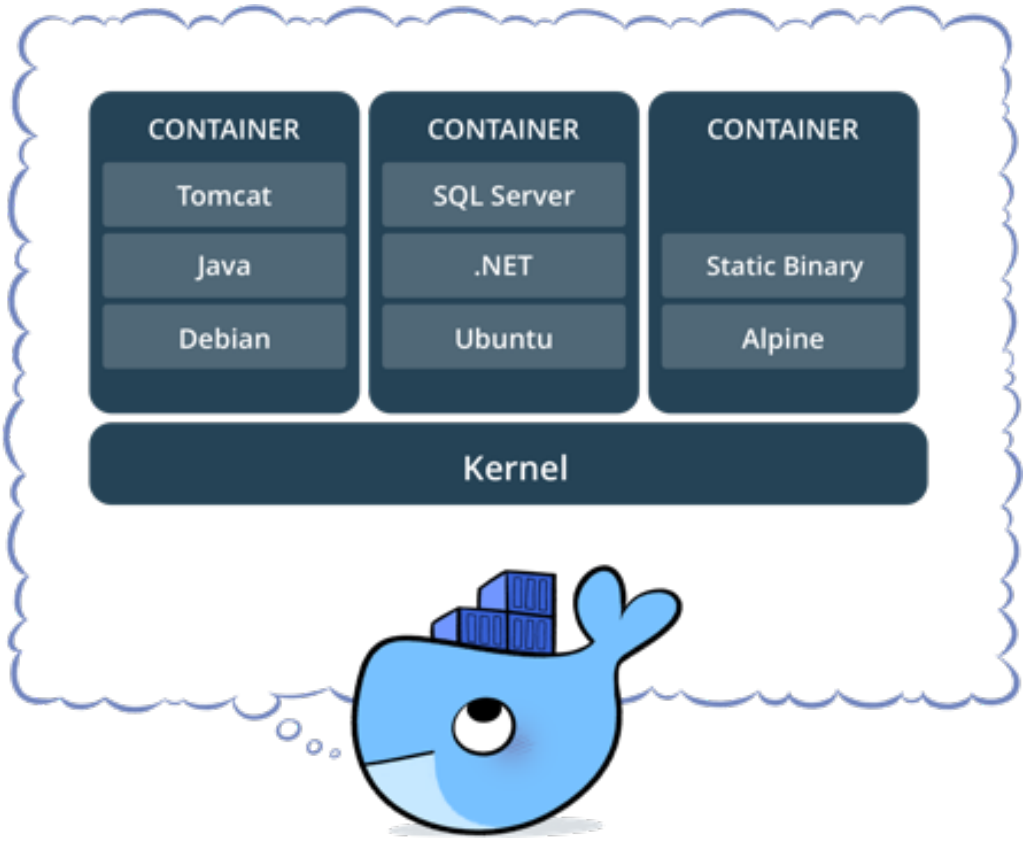

Figura 2.1. Schema di principio di Docker (fonte: docker.io).

2.1 Confronto tra Containers e Virtual machines

Sia i containers che le virtual machines hanno pressoché gli stessi benefici per quanto riguarda l'isolamento delle applicazioni, ma lavorano a differenti livelli di virtualizzazione in quanto i containers virtualizzano il sistema operativo, invece le virtual

machines virtualizzano l'hardware. Questo fa sì che i containers siano maggiormente portabili ed efficienti.

2.1.1 Containers

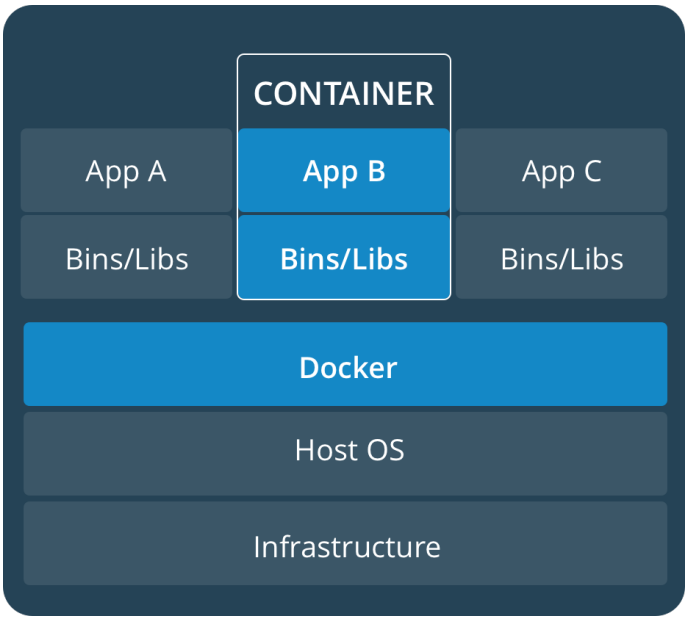

Figura 2.2. Schema di principio dei Containers (fonte: docker.io).

Il container è un'astrazione effettuata a livello applicativo che in sostanza va a mettere insieme sia il codice che le sue dipendenze in una "scatola".
Inoltre più containers possono girare sulla stessa macchina condividendo il kernel del sistema operativo, per cui ogni container girerà come un processo isolato in user space.
Un altro vantaggio rispetto alle virtual machine è il fatto che i container occupano meno spazio (tipicamente le loro immagini hanno una dimensione pari a qualche decina o poche centinaia di MBs), per cui il loro avvio è pressoché istantaneo.

2.1.2 Virtual machine

Come già detto le virtual machines (VMs) virtualizzano l'hardware per cui grazie al loro uso è possibile creare alcuni servers avendo a disposizione un solo server fisico. L'elemento chiave per permettere tutto ciò è l'hypervisor che in sostanza dà la possibilità di lanciare più VMs su una singola macchina.
Infine ogni VM ha al suo interno una copia completa del sistema operativo, una o più applicazioni, nonché tutte le librerie e i binari necessari, appunto per questo le

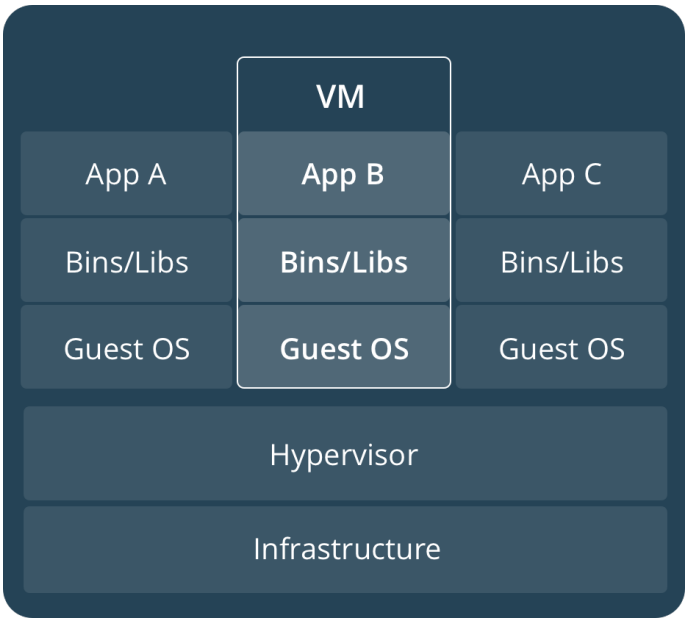

Figura 2.3. Schema di principio delle VMs (fonte: docker.io).

VM possono occupare fino ad alcune decine di GBs di spazio, per cui, ovviamente, il loro avvio è più lento rispetto ai containers.

2.2 Componenti principali

Docker si compone principalmente di questi quattro componenti:

- Docker Engine.
- Docker Client.
- Docker Image.
- Docker Container.

Inoltre l'architettura di massima è indicata in figura 2.4

2.2.1 Docker Engine

É il core della piattaforma ed in pratica standardizza la gestione dei containers. Grazie ad esso è possibile sviluppare un'applicazione su di una piattaforma (ad esempio una workstation) per poi spostarla su di un'altra piattaforma (ad esempio server di produzione) senza alcuna fatica, a patto che entrambi eseguano Docker

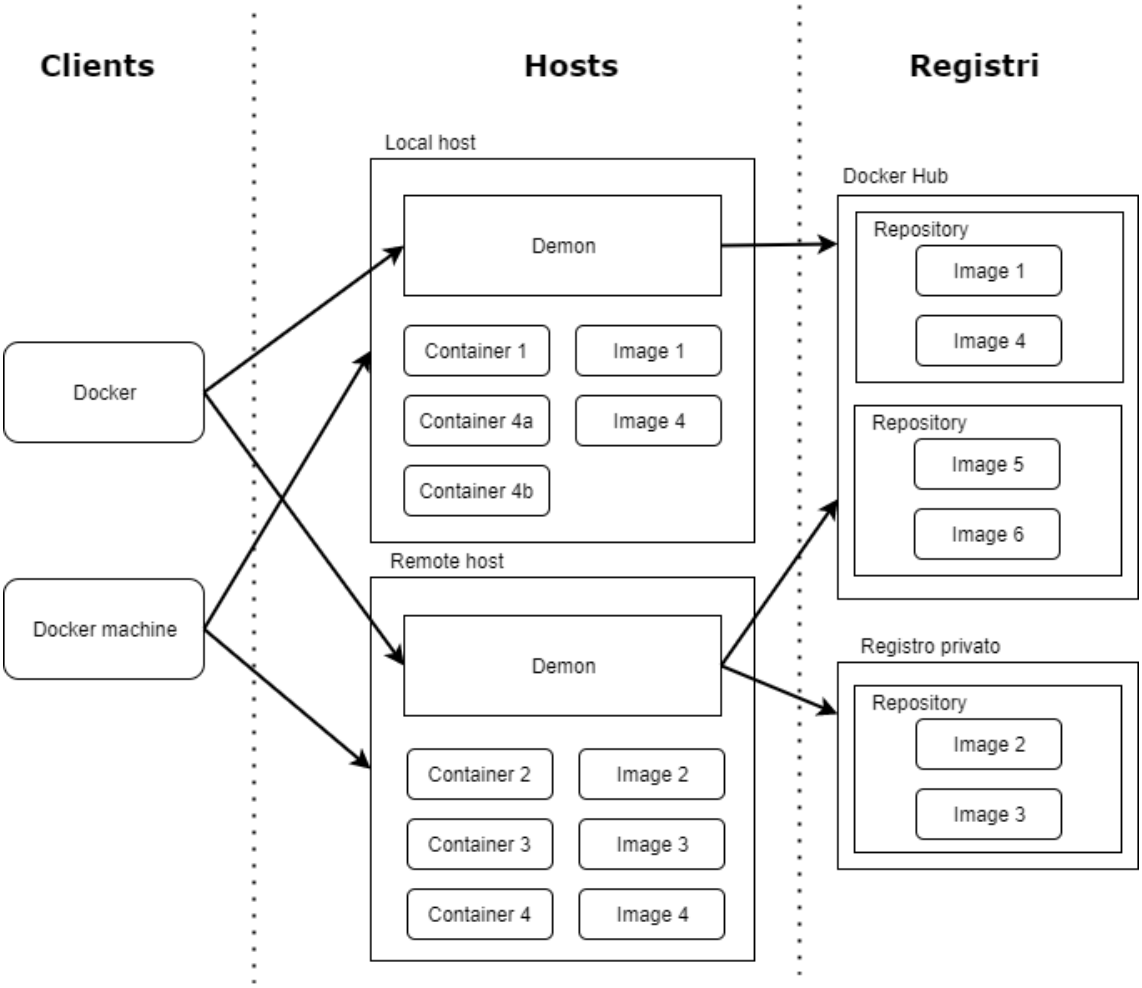

Figura 2.4. Architettura di Docker

Engine.
In sostanza il Docker Engine è un demone che gira sulla macchina che deve ospitare i containers ed ha il compito di fornire l'accesso a tutte le funzionalità e servizi offerti da Docker.
Al giorno d'oggi esistono due versioni del Docker Engine:

- Docker EE (Enterprise Edition), a pagamento.
- Docker CE (Community Edition), gratis.

2.2.2 Docker Client

Fa da interfaccia per le API esposte dal Docker Engine, ed in sostanza mette a disposizione una serie di comandi (CLI) con cui accedere alle funzionalità offerte.
Inoltre ogni comando inizia con la parola chiave:

- docker

Ad esempio il comando:

- docker build -t my_container .

Crea l'immagine di un nuovo container partendo da un Dockerfile presente nelle directory in cui viene lanciato quel comando dandogli come tag (nome) my_container.

2.2.3 Docker Image e Docker Container

Un'immagine è un insieme di file e parametri che definisce e configura un'applicazione da usare a runtime. Inoltre essa ha le seguenti due caratteristiche:

- Non ha uno stato.
- É immutabile.

Un container, invece, è un'istanza in esecuzione di un'immagine il cui file system è costituito da uno strato R/W sovrapposto agli strati immutabili dell'immagine (come mostrato in figura 2.5)

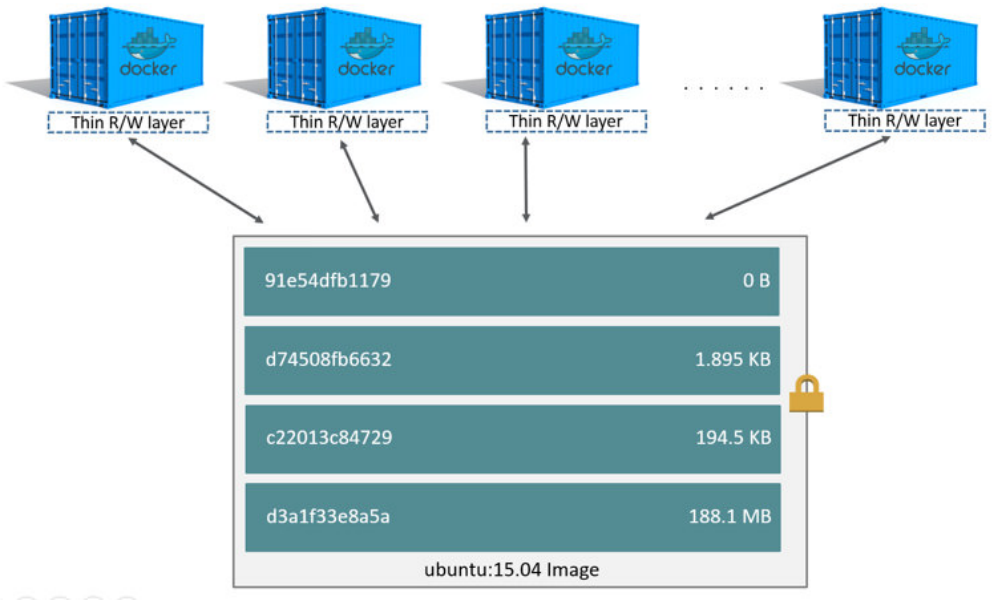

Figura 2.5. Interazione tra containers ed immagini (fonte: docs.docker.com).

2.3 Dockerfile

É possibile costruire una nuova immagine descrivendone il suo contenuto in un Dockerfile, che in sostanza è un file di testo avente al suo interno una serie di istruzioni che via via costruiscono l'immagine.
Le istruzioni non sono case sensitive, però è buona norma utilizzare il maiuscolo per le istruzioni cosicché da distinguerle dai parametri.
Le principali istruzioni sono le seguenti:

- **FROM**: che è obbligatoria e DEVE essere la prima istruzione presente nel Dockerfile, e va a definire l'immagine di base su cui si va a costruire la propria.
- **RUN**: esegue nel Docker Engine un comando di shell aggiungendo un nuovo layer all'immagine base, questo è utile per l'installazione di software e package aggiuntivi necessari per l'esecuzione dell'applicazione desiderata. La shell di default è:
  - /bin/sh per Linux.
  - cmd/S /C per Windows.
- **COPY**: permette di copiare dei files locali sull'immagine nella posizione specificata.
- **ENV**: dà la possibilità di definire delle variabili d'ambiente utilizzabili all'interno del container.
- **VOLUME**: crea un punto di mount per il path specificato, dichiarandolo esterno al container, appartenente all'host oppure ad un altro container.
- **CMD**: permette di eseguire un comando di shell a runtime, cioè una volta che il container viene lanciato (se è presente questa istruzione deve essere unica, anche perchè nel caso in cui ce ne fosse più di una verrebbe eseguita soltanto l'ultima).
  Lo scopo principale di questa istruzione è quella di fornire un'istruzione di default che il container dovrà eseguire, inoltre il container terminerà la sua esecuzione non appena il processo utilizzato per servire l'istruzione di default terminerà.
- **ENTRYPOINT**: ha lo stesso scopo di CMD, cioè definisce il programma da eseguire all'avvio del container, la differenza principale consiste nel fatto che un comando passato da CLI con *docker run* può sovrascrivere CMD ma non l'ENTRYPOINT.

- **EXPOSE**: dichiara quali porte possono essere raggiungibili dall'esterno, ad esempio se il container ospita un server web è opportuno lasciare raggiungibili le porte: 80 e 443.

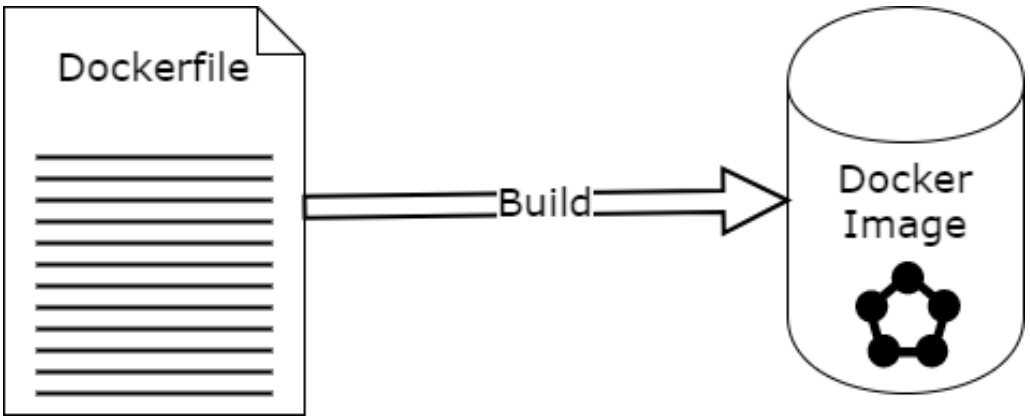

Figura 2.6. Passaggio dal Dockerfile all'immagine.

2.4 Network

Docker offre tre reti di default:

- Bridge:
  La rete bridge rappresenta l'interfaccia docker0 presente sugli host che ospitano Docker.
  Di default i containers appartengono a questa rete, ed inoltre tali containers saranno in grado di comunicare tra loro utilizzando l'indirizzo IP a loro assegnato.

- None:
  In sostanza i containers appartenenti a questa rete non avranno nessuna interfaccia di rete se non quella di loopback.

- Host:
  I container aggiunti alla rete host vedranno l'intero stack di rete visto dal Docker host.

Inoltre c'è anche la possibilità di creare nuove reti a piacimento. Per far ciò Docker mette a disposizione alcuni drivers di rete:

- Bridge network.

- Overlay network.
- MACVLAN network.
- Oppure è anche possibile creare nuovi network plugin per avere una piena personalizzazione della rete.

In breve con l'utilizzo di questi drivers, c'è la possibilità di creare nuove reti a cui è possibile aggangiare un insieme ben determinato di containers, ottenendo quindi un maggiore isolamento rispetto alla totalità dei containers presenti.

Capitolo 3

Kubernetes

Kubernetes è una piattaforma open-source che automatizza il deployment, lo scalamento e la gestione delle operazioni da effettuare sui containers all'interno di un cluster di hosts, fornendo quindi un'infrastruttura centralizzata sui containers.
Pertanto con Kubernetes, si è in grado di effettuare in modo efficiente e veloce le seguenti esigenze degli utenti:

- Deployare applicazioni.
- Scalare orizzontalmente il numero di repliche delle applicazioni.
- Aggiornare le applicazioni senza momenti di downtime.
- Limitare l'uso delle hardware, in modo tale da utilizzare solo quanto richiesto.

Per cui l'obiettivo di Kubernetes è quello di creare un ecosistema di componenti e tools che riducano il peso delle applicazioni sia in cloud pubblici che privati.
Quindi Kubernetes è:

- Portabile: cioè si può istanziare in ambienti:
  - Pubblici.
  - Privati.
  - Ibridi.
  - Multi-cloud.
- Estendibile: cioè è sempre possibile modificare la dimensione del cluster di Kubernetes. Inoltre è modulare, ad esempio è possibile scegliere qualsivoglia componente per gestire la comunicazione (a livello di rete) tra i vari nodi.

- Self-healing: cioè ha dei meccanismi per:
  - L'auto collocamento dei containers nel cluster.
  - L'auto restart dei container che per qualche motivo sono andati in crash.
  - L'auto replication cioè è in grado di mantenere il numero di repliche desiderate sul cluster.
  - L'auto scaling cioè in base alle politiche desiderate è in grado di aumentare o diminuire il numero di repliche di una data applicazione, con il fine di ottimizzare l'utilizzo delle risorse.

I principali aspetti che portato al successo Kubernetes sono due:

- Aspetto economico: con l'avvento del Cloud computing si presenta il problema di ottimizzare le risorse, in quanto, generalmente, il carico sulle macchine non è costante quindi se si allocano le risorse in modo fisso si rischia di avere delle risorse non utilizzate (ma pagate) oppure di non riuscire a rispondere a tutte le richieste in altri periodi, per cui Kubernetes grazie al meccanismo di auto-scaling permette di avere più o meno risorse dedicate ad un certo servizio in base al carico di lavoro corrente.
- Aspetto pratico: al giorno d'oggi i servizi devono essere disponibili 24 ore su 24 e 7 giorni su 7 e gli sviluppatori hanno la necessità di deployare nuove release periodicamente (anche più volte al giorno) quindi grazie al meccanismo del rolling update tutto ciò diventa facilmente realizzabile senza momenti di downtime.

In sostanza Kubernetes funziona in modo centralizzato, cioè è composto da un nodo master che coordina e impartisce gli ordini agli altri nodi chiamati semplicemente "node".

3.0.1 Perché si è scelto di gestire i containers?

Nella vecchia maniera era necessario installare le applicazioni direttamente sul sistema operativo (come mostrato in figura 3.1), questo aveva il grosso svantaggio di avere l'applicazione, cioè i files: eseguibili, di configurazione e le librerie compatibili con il sistema operativo sottostante. Una possibile soluzione può essere quella di creare una virtual machine (VM) per ogni applicazione, ma lo svantaggio delle VMs sta nel fatto che sono più pesanti e non portabili.

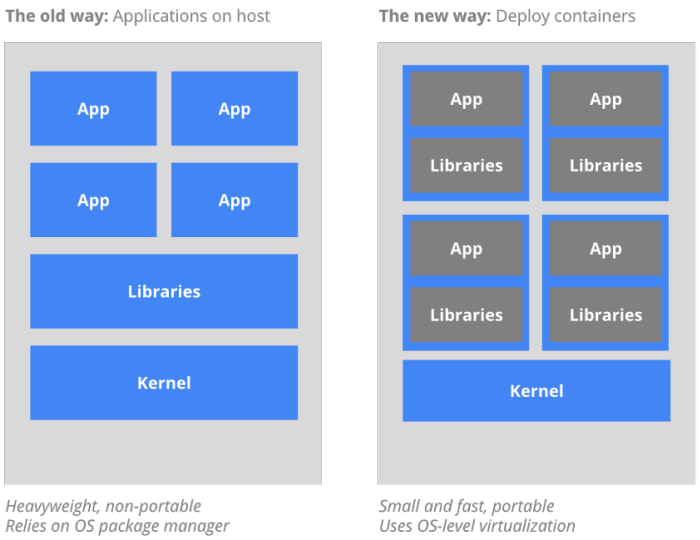

Figura 3.1. Schema dell'uso di docker in Kubernetes(fonte: Kubernetes).

Con i containers invece si vanno a deployare delle immagini che sono virtualizzate a livello di sistema operativo invece che a livello hardware. Inoltre i containers sono isolati gli uni dagli altri e dall'host che gli ospita, vale a dire che hanno un loro file system, hanno a disposizione una ben definita capacità computazionale e non hanno visibilità sugli altri processi presenti sull'host. E grazie al fatto che disaccoppiano l'infrastruttura sottostante e il file system dell'host sono portabili su diversi sistemi operativi.
Infine dato che i containers sono più leggeri e veloci, è opportuno creare un container per ogni applicazione, cosicché si ha una relazione uno-a-uno tra applicazione e immagine, in modo tale da sfruttare appieno i vantaggi offerti dai containers. Ad esempio i containers sono molto più trasparenti delle VMs per cui sono più facili da gestire e monitorare. Pertanto se si ha un'applicazione per container, monitorare i containers equivale a monitorare le applicazioni.

3.1 Elementi principali

In Kubernetes esistono due tipi di nodi:

- Master.
- Node.

3.1.1 Master

Il nodo master è unico ed è il nodo principale del cluster, cioè ha il compito di impartire gli ordini agli altri nodi ed è anche il nodo che si interfaccia con l'utente, ossia l'utente inoltra le sue richieste solamente al master.
A sua volta il master ha al suo interno i seguenti elementi:

- **Kube controller-manager**:
  è un demone che gira nel nodo master, ed ha il compito di mantenere il cluster nello stato desiderato, ad esempio svolge la funzione di replication controller, cioè mantiene per un determinato pod il numero di repliche desiderato. Pertanto ha la necessità di comunicare con i vari nodi, e per far ciò si avvale del kube-apiserver; che a sua volta usa etcd[1], per ricavare le informazioni sulla configurazione di rete degli altri nodi, vale a dire i loro indirizzi IP e la configurazione dei tunnel usati nella comunicazione verso i containers.

- **Kube-scheduler**:
  è il modulo che decide dove lanciare (su che nodo) un determinato pod, in pratica è il responsabile del bilanciamento dei pod sul cluster. Di default il suo comportamento è il seguente:

  – Filtra le macchine in modo tale da scegliere solamente su quelle che hanno le risorse necessarie.

  – Lancia il pod su quella macchina che dà la maggior distribuzione del carico possibile.

- **Flannel**:
  è il modulo che permette la comunicazione dei pod che girano su nodi diversi. In sostanza va a creare una rete di overlay sul data center, mediante la creazione di tunnel IP, cosicché da non dover intervenire in nessun modo sugli apparati di rete (switches e routers), come mostrato nella figura 3.2.
  In pratica questa rete di overlay va a creare un maglia completa di tunnel tra tutti i nodi presenti sul cluster, in modo tale da non aver nessun collo di bottiglia, cioè il traffico non è obbligato a passare da un nodo "speciale" che ha il compito di gestire la comunicazione, come si può vedere nella figura 3.3.

[1] Database distribuito

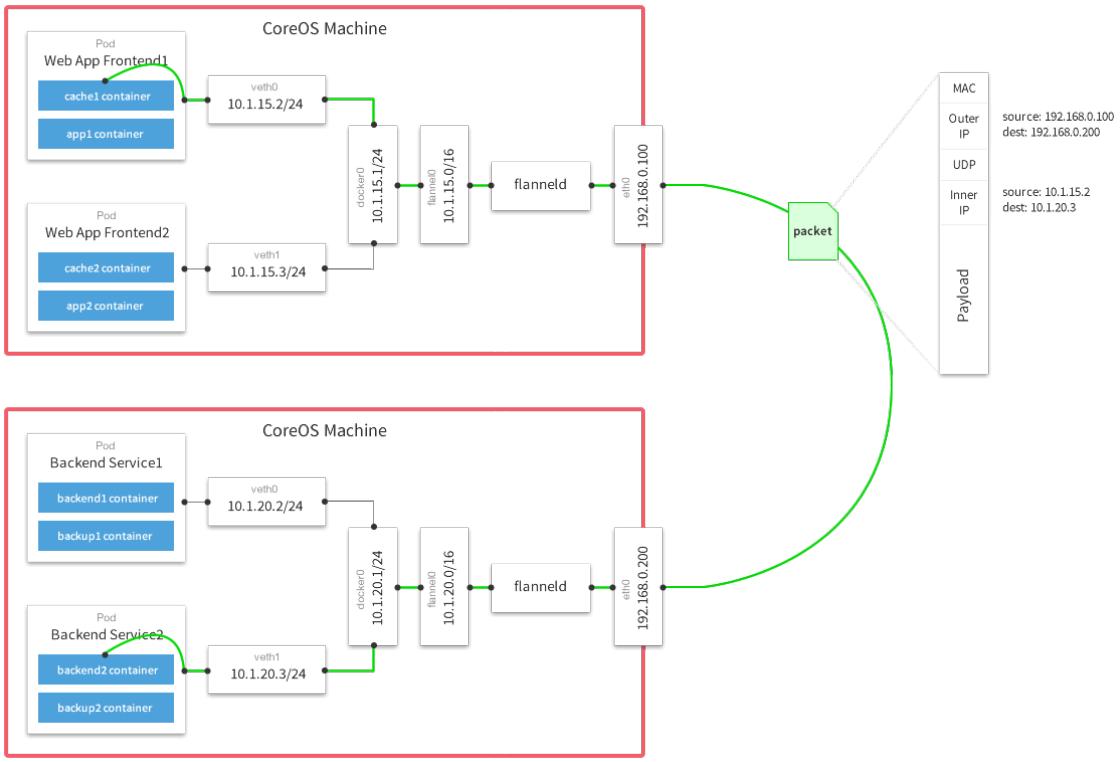

Figura 3.2. Funzionamento di Flannel (fonte: chunqi.li).

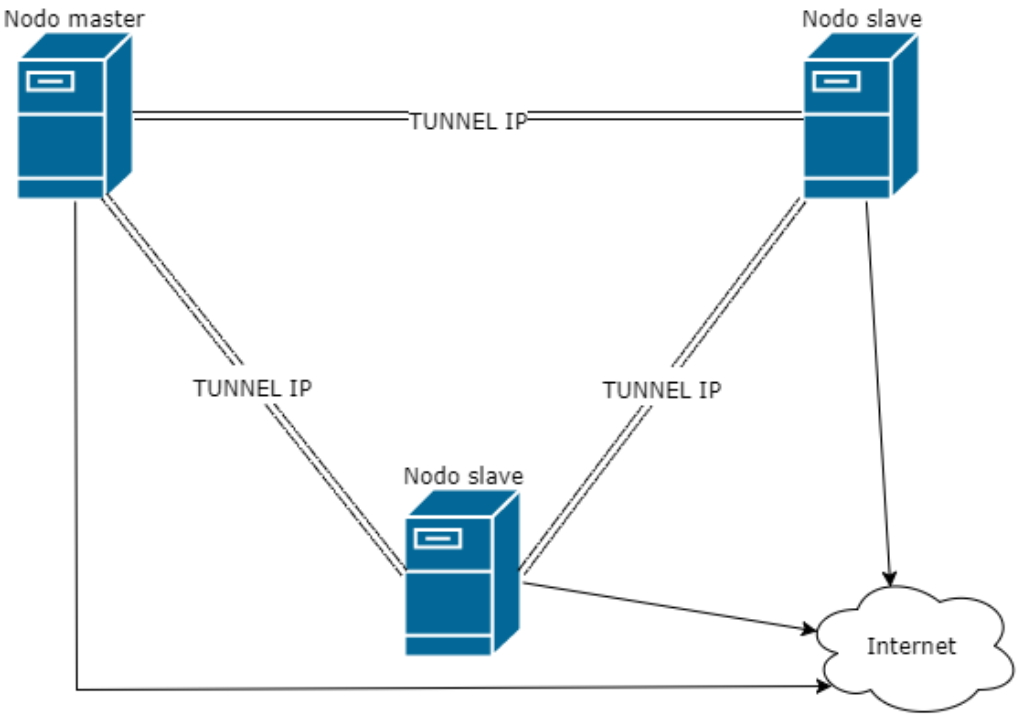

Figura 3.3. Schema logico di comunicazione in Kubernetes.

3.1.2 Node

Il node (anche chiamato minion) è la macchina che esegue i task richiesti; un node può essere sia una Virtual Machine che una macchina fisica. Inoltre ogni node ha una serie di servizi necessari affinché sia in grado di far girare i containers ed interagire con il master, tra cui:

- **Kubelet**:
  è il "node agent", vale a dire il modulo che comunica con il master, per cui una volta ricevuto il comando di lanciare un determinato pod su quel nodo, si assicura che i containers descritti nella specifica del pod siano attivi.

- **Flannel**:
  stesse funzioni del modulo che gira nel master.

- **Kube-proxy**:
  responsabile del traffico diretto ai pod proveniente dall'esterno e viceversa, in sostanza all'atto di creazione di un nuovo pod il kube-proxy inserisce in quel nodo una nuova regola all'interno dell'iptables che indirizza tutto il traffico che soddisfa determinate regole (basate principalmente sull'ip sorgente e porta di destinazione) ad un pod specifico, come mostrato nella figura 3.4.

Ma come funziona il Kube-proxy?

In ogni nodo appartenente al cluster di Kubernetes è attivo il kube-proxy. Il kube-proxy è il responsabile dell'implementazione del virtual IP utilizzato dai services. Negli anni ha subito un'evoluzione, cioè nella versione 1.0 il kube proxy girava puramente in userspace, successivamente dalla versione 1.1 il kube-proxy va ad interagire con l'iptables.
Più in dettaglio:

Proxy-mode: userspace

In questa modalità il kube-proxy è in contatto con il master, più precisamente con l'apiserver del master, il quale gli comunica quali servizi sono da aggiungere o rimuovere (come mostrato in figura 3.5). Per ogni servizio apre una porta (scelta a caso dal master) sul nodo. Dopodiché inserisce una nuova regola nell'iptables che in sostanza va indirizzare tutto il traffico diretto all'indirizzo IP e porta del servizio al kube-proxy, il quale a sua volta inoltrerà questo traffico al Pod opportuno, di default

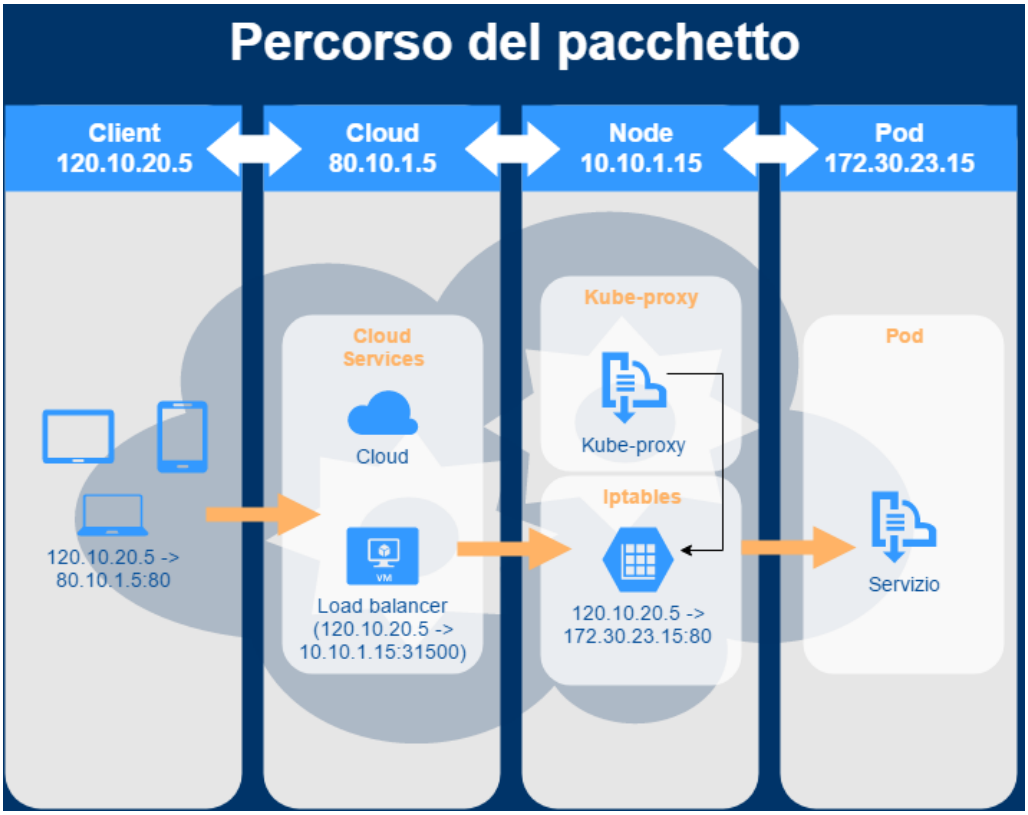

Figura 3.4. Funzionamento del kube-proxy.

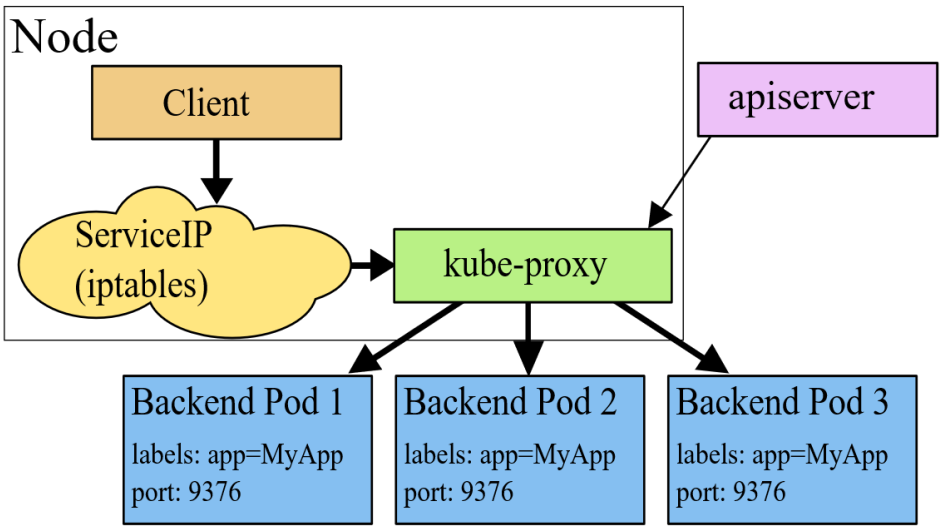

Figura 3.5. Proxy-mode in userspace (fonte: kubernetes.io).

il kube-proxy sceglie il Pod a cui inviare il traffico in round robin, ma volendo può essere selezionata un'altra politica ad esempio quella che si basa sull'affinità dell'indirizzo IP del client.

In questo caso si ha lo svantaggio che tutto il traffico passa in userspace (per cui non

è veloce come soluzione), ha invece il vantaggio che se per qualche motivo il Pod non risponde il kube-proxy può ritentare di inviare il pacchetto ad un altro Pod.

Proxy-mode: iptables

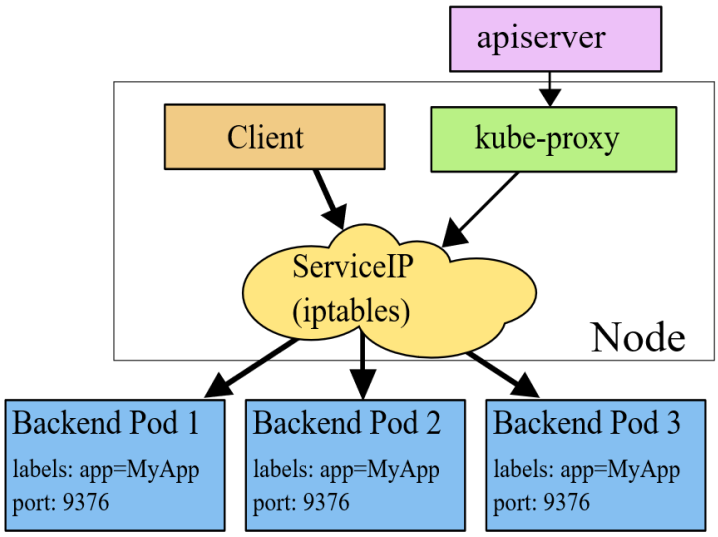

Figura 3.6. Proxy-mode con iptables (fonte: kubernetes.io).

Anche in questo caso il kube-proxy è in contatto con l'apiserver del master il quale gli comunica quali servizi sono da aggiungere o rimuovere dal cluster.
Per ogni service va a configurare una nuova regola nell'iptables che va ad indirizzare tutto il traffico diretto all'indirizzo IP del service (che di fatto è virtuale) ad un insieme di Pod di quel service (come mostrato in figura 3.6).
Inoltre configura per ogni Endpoint una regola che vada ad inviare il traffico proveniente da uno specifico Endpoint al Pod, di default questo Pod è scelto in modo random. Dato che è possibile che ci sia un numero elevato di Endpoints, di default le regole inserite nell'iptables riguardano solamente una parte dell'indirizzo IP sorgente (cioè si suppone che le richieste arrivino da indirizzi IP uniformemente distribuiti) così da bilanciare il carico sui vari Pod senza eccedere nelle dimensioni della tabella iptables.

Esempio di possibili regole da inserire nell'iptables:

IP sorgente	Pod destinazione
(IPsrc && 0.0.0.3) == 0	Frontend001 (IP 172.30.23.1)
(IPsrc && 0.0.0.3) == 1	Frontend002 (IP 172.30.23.2)
(IPsrc && 0.0.0.3) == 2	Frontend003 (IP 172.30.23.3)
(IPsrc && 0.0.0.3) == 3	Frontend004 (IP 172.30.23.4)

Il principale vantaggio di utilizzare questa soluzione (che è utilizzata di default dalla versione 1.2 in poi) sta nel fatto che il traffico non passa più in userspace (cioè non passa dal kube-proxy) per cui l'inoltro ai Pod è più veloce. Dall'altro lato si ha lo svantaggio che il kube-proxy non può ritentare di inviare il pacchetto ad un altro Pod qualora il primo Pod non risponda, questo perché non è più il kube-proxy l'incaricato di inoltrare i pacchetti ai Pods.

3.2 Concetti fondamentali

3.2.1 Pod

Un Pod è l'oggetto più semplice e piccolo che Kubernetes può modellare, creare e gestire. In sostanza un Pod rappresenta un processo attivo sul cluster.
Un Pod incapsula al suo interno un container (o, in alcuni casi, più containers), risorse di stoccaggio, ed ha un indirizzo IP (unico all'interno del cluster) inoltre è anche a conoscenza di tutte le impostazioni necessarie per il corretto funzionamento dei containers.
Il container (o i containers) all'interno dei Pods nella maggior parte dei casi usano Docker come piattaforma di gestione, ma possono comunque supportare altri tipi di containers.
I Pods possono essere utilizzati in diversi modi all'interno di un cluster di Kubernetes, ad esempio:

- Pods con al loro interno un singolo container. Questo è il modello maggiormente utilizzato dagli utilizzatori di Kubernetes, in questo caso il Pod fa da wrapper al singolo container, questo è necessario perché Kubernetes non gestisce direttamente i containers ma solamente i Pods.
- Pods con al loro interno più containers. Un Pod può incapsulare un'applicazione composta da più containers fortemente correlati tra di loro che hanno la necessità di condividere le risorse. In questo caso il Pod fa da wrapper a questi containers in modo tale da gestirli come un'unica entità.

3.2.2 Perché mettere più containers all'interno di un Pod?

Innanzitutto i containers all'interno di un Pod sono automaticamente allocati e schedulati su di una stessa macchina (fisica o virtuale) appartenente al cluster. I containers possono quindi condividere risorse e dipendenze, comunicare gli uni con

gli altri, e coordinare quando e come vogliono la loro terminazione.
NB:
raggruppare più containers all'interno di uno stesso Pod è consigliabile solo per casi particolari, ovvero solo quando questi sono fortemente correlati tra di loro.
Ad esempio, quando un web-server ha la necessità di recuperare dei dati che sono continuamente aggiornati da un altro processo, come descritto nel diagramma in figura 3.7.

Dato che i containers all'interno di un Pod condividono le risorse, hanno anche lo

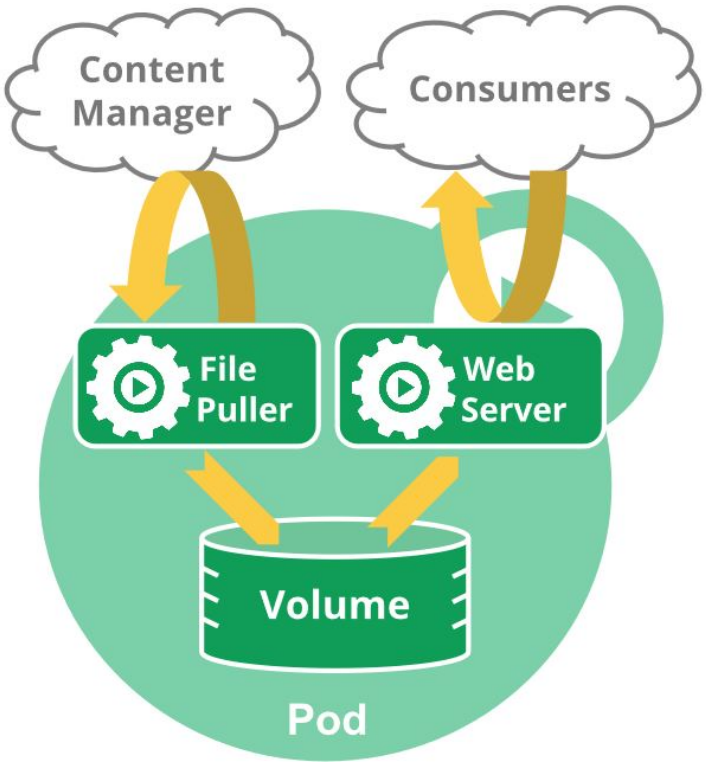

Figura 3.7. Esempio di Pod con più containers al suo interno(fonte: Kubernetes).

stesso indirizzo IP e le stesse porte a disposizione. Per cui i container all'interno dello stesso Pod possono comunicare tra di loro usando localhost, invece quando devono comunicare con entità esterne al Pod è indispensabile che vadano a coordinare l'accesso alle risorse di rete.
È anche possibile avere dei volumi all'interno di un Pod, questo permette ai containers (appartenti al Pod) di condividere dati. Inoltre i volumi all'interno di un Pod permettono lo stoccaggio persistente di dati, cioè sopravvivono nel caso in cui un container abbia bisogno di un riavvio. Però non sopravvivono al Pod, cioè il ciclo di vita di un volume all'interno di un Pod è legata alla vita del Pod stesso. Per avere dei dati salvati in modo persistente e che abbiano una durata indipendente dalla vita dei Pod è necessario utilizzare dei volumes esterni al Pod.

3.2.3 Deployment

Un deployment fornisce un servizio di aggiornamento automatico per i Pod e ReplicaSets descritti all'interno dell'oggetto Deployment.
Per cui è necessario descrivere lo stato desiderato all'interno dell'oggetto Deployment, dopodiché ci penserà il Deployment controller a mantenere all'interno del cluster lo stato descritto per l'insieme di Pod o ReplicaSets in questione.

3.2.4 Service

Un service è un astrazione che va raggruppare logicamente un insieme di Pod, e va definirne le policies di accesso ad essi. Il service per raggruppare i Pods, per comodità, può avvalersi delle Label Selector, in questo modo tutti i Pod con una determinata Label faranno parte di quel service.
L'introduzione dei services è stata necessaria in quanto i Pod non hanno una vita certa e per di più ogni Pod ha un indirizzo IP diverso.
Questo è un problema ad esempio nel caso in cui, un insieme di Pods (che svolgono la funzione di back end) forniscono un servizio ad altri Pod (che hanno la funzione di frontend), perché in questo caso i Pods di frontend devono essere a conoscenza degli indirizzi IP dei Pod di backend, che in linea di principio possono essere tanti a svolgere la stessa funzione, inoltre questi indirizzi non sono stabili, ad esempio perché:

- si decide di cambiare il numero di repliche dei Pod.
- si effettuano delle roll-update.
- per qualche motivo un Pod va in crash, per cui il ReplicationController provvede a crearne un altro (che può avere un indirizzo IP diverso).

Quindi in questo caso l'utilizzo del service fa sì che tutti i Pod di backend (come quelli di frontend) siano raggiungibili da un unico indirizzo IP, facilitando di molto la loro comunicazione.
Con l'utilizzo dei services è anche possibile esporre all'esterno del cluster determinati Pod. Per far ciò Kubernetes offre due modalità di funzionamento:

Nodeport

Settando il campo "type" del service a "NodePort" il master di Kubernetes alloca automaticamente una porta (appartenente al range: 30000-32767) in ogni nodo del

cluster in modo tale che vada a fare da proxy a quel service. Cosicché il servizio sia disponibile all'esterno al seguente indirizzo:

<NodeIP>:nodePort

In questo modo è possibile definire un proprio servizio di load balancing del servizio in questione.

LoadBalancer

Settando il campo "type" del service a "LoadBalancer" si fa in modo che il proprio servizio vada ad utilizzare un Load Balancer offerto dal cloud provider per esporre i propri Pods all'esterno.

3.2.5 HPA (Horizontal Pod Autoscaling)

Grazie all'utilizzo dell'HPA (Horizontal Pod Autoscaling) Kubernetes è automaticamente in grado di scalare il numero di Pods presenti all'interno di un Replication controller, Deployment o Replica Set osservando principalmente l'utilizzo della CPU (ma è comunque possibile definire altre metriche). L'HPA si compone di due elementi:

- Kubernetes API resource
- Controller

In sostanza all'interno della resource è definita tutta la conoscenza che il controller deve avere (ad esempio la soglia di utilizzo della CPU oltre la quale creare un'altra replica del Pod). Quindi il controller non deve far altro che regolare periodicamente il numero di repliche presenti all'interno di un deployment in base alle specifiche descritte nella resource.
In dettaglio:
L'HPA è implementato come un loop di controllo che in ogni periodo (di default lungo 30 secondi) esegue le seguenti operazioni:

- Il controller ottiene i dati che riguardano l'utilizzo delle risorse di tutti quegli elementi che si avvalgono dell'HorizontalPodAutoscaler (HPA); questi dati li può recuperare in due modi:
  - Accesso diretto tramite Heapster.

  - Accesso da remoto con un client tramite REST.

- Per le metriche riguardanti i Pod (ad esempio l'uso della CPU), il controller processa le metriche ottenute in precedenza. In pratica va a confrontarle con quelle definite nell'HorizontalPodAutoscaler, producendo come risultato il numero di repliche desiderate.

- Per le metriche custom il controller ha un funzionamento simile a quello descritto nel punto precedente, soltanto che le metriche specificate nell'HPA devono essere dei numeri, cioè non possono essere percentuali.

- Infine l'autoscaler accede al corrispondente replication controller, deployment o replica set con il fine di impostare il corretto numero di repliche necessarie.

3.2.6 Rolling update

È il meccanismo offerto da Kubernetes agli sviluppatori per effettuare il rilascio di nuove versioni senza momenti di downtime.
Kubernetes segue il seguente procedimento per effettuare le rolling update:

- Crea un nuovo replication controller con la configurazione aggiornata.

- Aggiunge nuove repliche aggiornate al cluster ed elimina quelle vecchie, e di default lo scostamento massimo di numero di repliche è pari ad 1; ad esempio se si desidera aggiornare un deploy che ha al suo interno 4 repliche di un determinato Pod nella fase di transizione potranno esserne presenti al più 5 Pod, come mostrato nelle figure seguenti (3.8 - 3.14).

- Infine la vecchia immagine della repliche non è eliminata dal cluster ma resta immagazzinata su disco, questo per permettere di tornare ad una versione precedente in breve tempo. Infatti tutte queste operazioni possono essere eseguite dallo sviluppatore, o in generale dall'utente di Kubernetes, con pochi comandi.

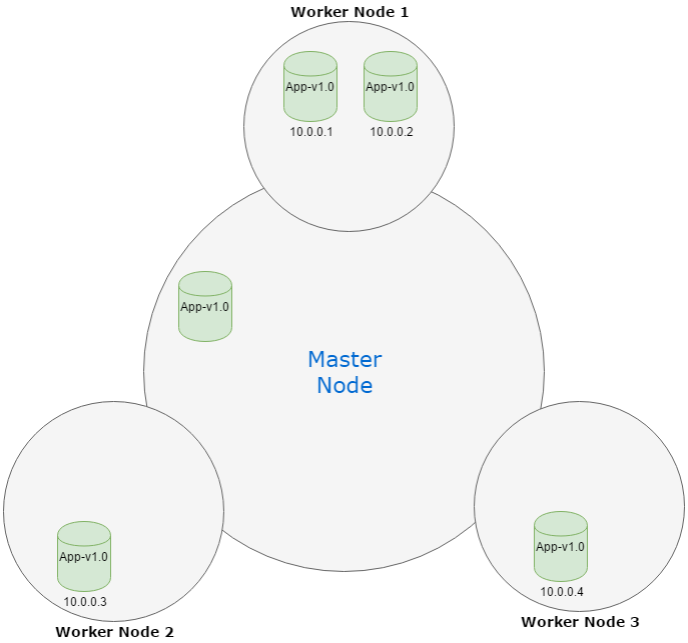

Figura 3.8. Stato iniziale del cluster

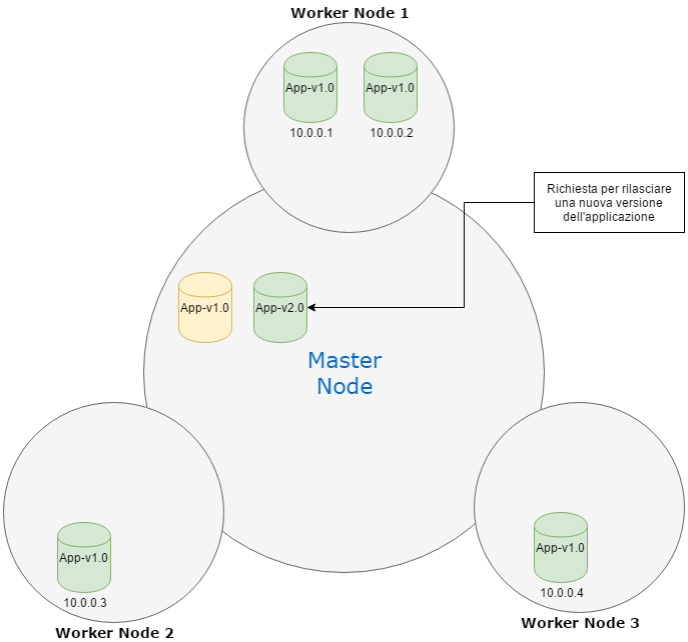

Figura 3.9. Richiesta di rilascio di una nuova versione

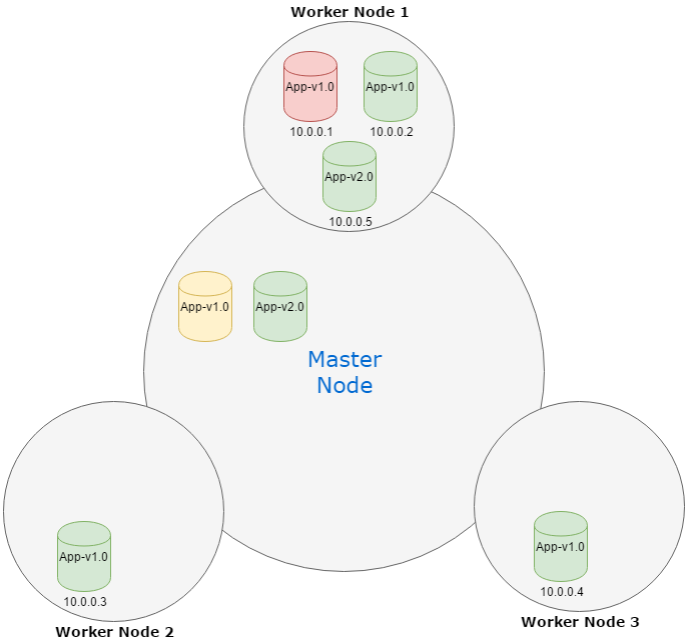

Figura 3.10. Sostituzione primo Pod

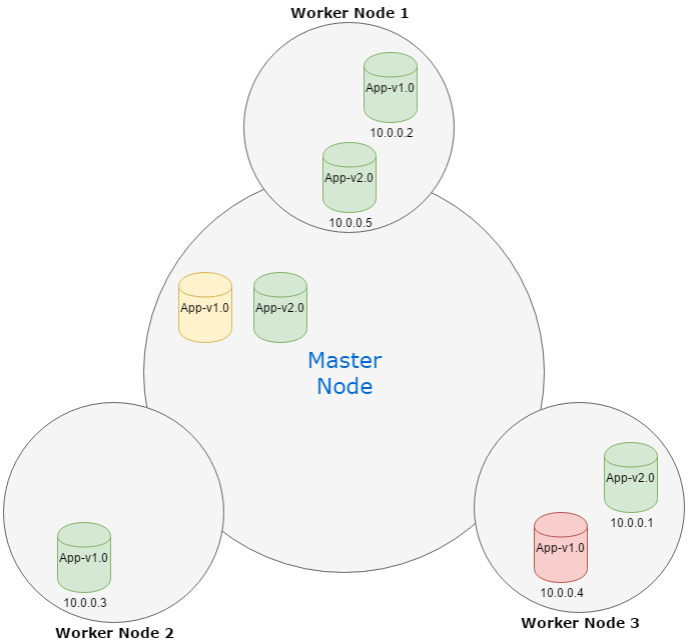

Figura 3.11. Sostituzione secondo Pod

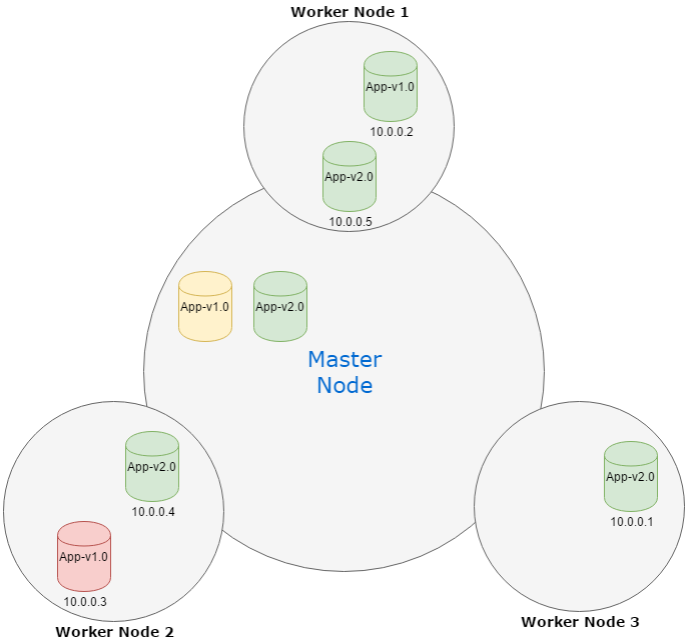

Figura 3.12. Sostituzione terzo Pod

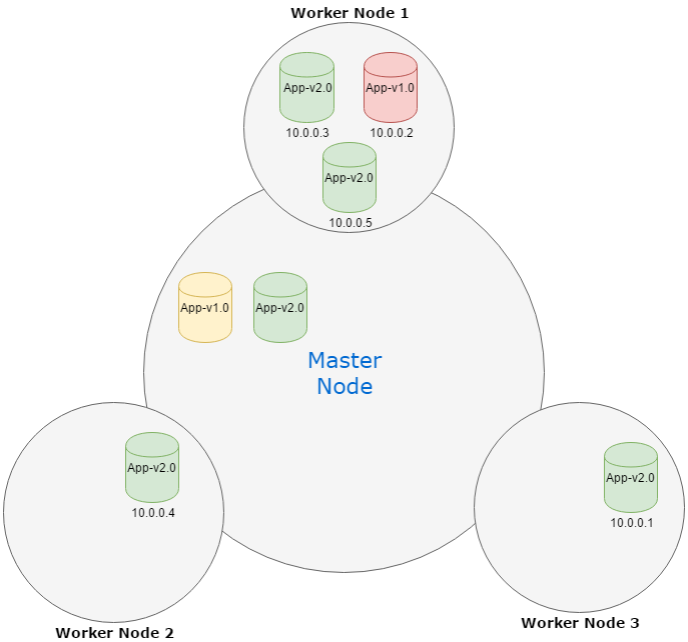

Figura 3.13. Sostituzione quarto Pod

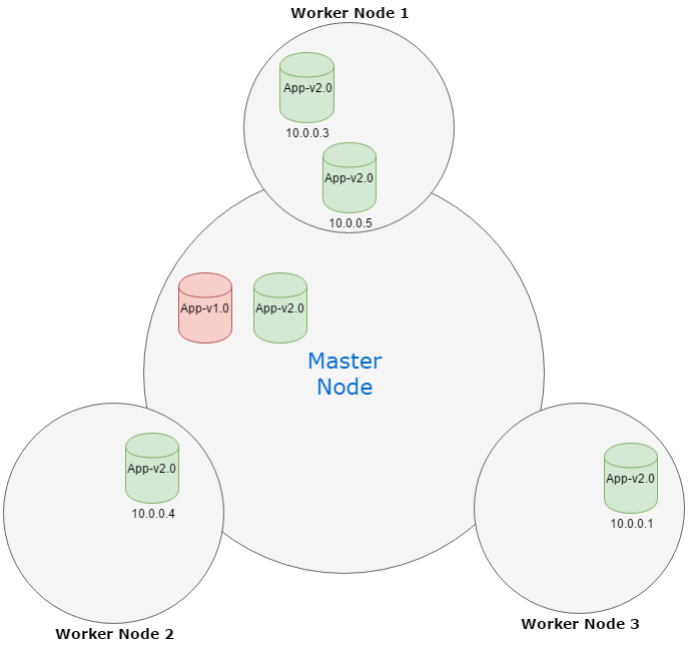

Figura 3.14. Stato finale del cluster

Capitolo 4

Terraform

Terraform è un tool usato per creare, modificare ed aggiornare la propria infrastruttura in modo sicuro ed efficiente. Infatti Terraform è in grado di gestire indifferentemente vari cloud provider oppure custom hardware.
Per infrastruttura si intende praticamente qualsiasi cosa, cioè con Terraform si possono gestire sia componenti di basso livello come nodi di compute, storage e networking sia componenti di alto livello come funzionalità di SaaS (Software as Service), entries del DNS, ecc.
Per far ciò è necessario scrivere degli appositi files di configurazione in cui è possibile descrivere in modo dettagliato sia una singola applicazione che l'intero datacenter. In seguito Terraform genera un piano di esecuzione che descrive quali operazioni andrà ad effettuare per raggiungere lo stato desiderato, successivamente lo esegue ottenendo l'infrastruttura descritta nei files.
Nel caso in cui si voglia soltanto modificare alcune parti dell'infrastruttura esistente, Terraform è in grado di modificare solo il necessario per cui andrà ad eseguire un piano di esecuzione incrementale.

4.1 Caratteristiche principali

I punti di forza offerti da Terraform sono:

- Infrastructure as Code (IaC):
  L'infrastruttura è descritta usando una sintassi di alto livello, questo consente di avere facilmente una visione completa del proprio datacenter per cui diventa semplice modificare, condividere e riusare la propria infrastruttura.

- Piano di esecuzione:
  Terraform è solito pianificare gli step che andrà ad eseguire, cioè genera un piano di esecuzione. Per cui prima di applicare le modifiche è possibile visionare questo piano evitando "sorprese" al tempo di creazione.

- Grafo delle risorse:
  Terraform crea un grafo delle risorse per cui è in grado di parallelizzare la creazione/modifica di risorse non dipendenti tra loro; questo è appunto il motivo per cui Terraform crea le risorse nel modo più efficiente possibile.

- Automazione delle modifiche:
  Grazie al piano di esecuzione ed al grafo delle risorse l'utente finale sa perfettamente quali modifiche verranno effettuate ed in quale ordine, quindi dato che l'intervento umano nella modifica vera e propria dell'infrastruttura è pressoché irrilevante la possibilità di cadere in errori di distrazione è notevolmente ridotta.

Nel corso della tesi Terraform è stato utilizzato principalmente per rendere facilmente replicabile la creazione di un cluster Kubernetes in multicloud (Softlayer e AWS), cosicché possa essere creato a richiesta (velocemente) dopodiché una volta esauriti i compiti da effettuare su Kubernetes si possa, altrettanto velocemente, eliminare l'instrastruttura creata in precendenza (onde evitare uno spreco di risorse).

Capitolo 5

Architettura di JMeter

5.1 Overview

JMeter è un'applicazione puramente Java ed open source che ha il compito di effettuare dei test di carico o stress test a degli applicativi (non necessariamente applicazioni web) con il fine di misurarne le performance.
Principalmente JMeter è composto da due componenti (vedi figura 5.1):

- Master:
  - È la componente che si interfaccia con l'utente, cioè offre la possibilità di lanciare i test e di recepire i risultati.
  - In sostanza fa da client verso gli slaves indicandogli quali test devono eseguire
- Slave:
  - È la componente che esegue materialmente i test.
  - In genere sono presenti più nodi slave, che prendono gli ordini da un unico master.

5.1.1 Premessa

Dato che si tratta dell'implementazione dei container di JMeter, si è deciso di aggiungere una feature a JMeter "classico". In sostanza, appunto per sfruttare il concetto di astrazione offerto Docker & Kubernetes, il master viene automaticamente a conoscenza degli indirizzi IP degli slave cosicché l'utente finale non si debba preoccupare

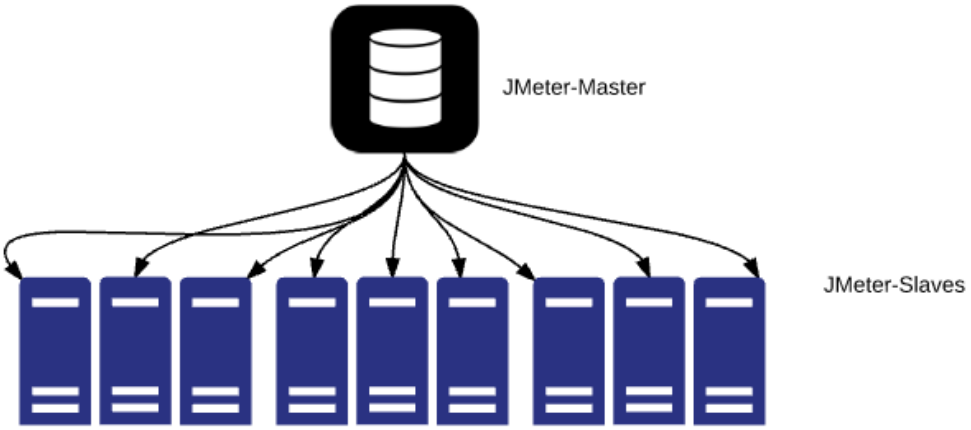

Figura 5.1. Funzionamento logico di JMeter

di recuperare tali indirizzi, dato che in Kubernetes è facile scalare orizzontalmente il numero di repliche degli slaves.

5.2 Progettazione

Affinché il master venga a conoscenza degli indirizzi IP degli slave è necessario, che questi indirizzi gli vengano comunicati. Per far ciò mi avvalgo di un database distribuito, in questo caso ho deciso utilizzare ETCD nel seguente modo:

1. Inizialmente il master è solo e crea un nuovo cluster ETCD (come indicato in figura 5.2):

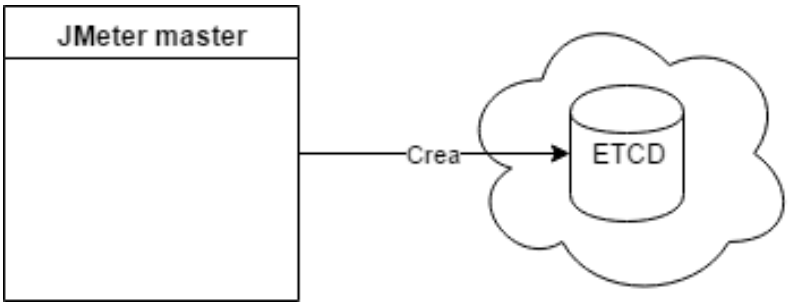

Figura 5.2. Creazione cluster ETCD

2. Successivamente vengono lanciate varie repliche dello slave (come indicato nelle figure 5.3 5.4 5.5), che al loro avvio eseguono le seguenti operazioni:

   (a) Si registrano al cluster ETCD creato dal master, per far ciò recuperano il nome del cluster tramite l'uso di un file condiviso.

   (b) Inseriscono in questo database il loro indirizzo IP.

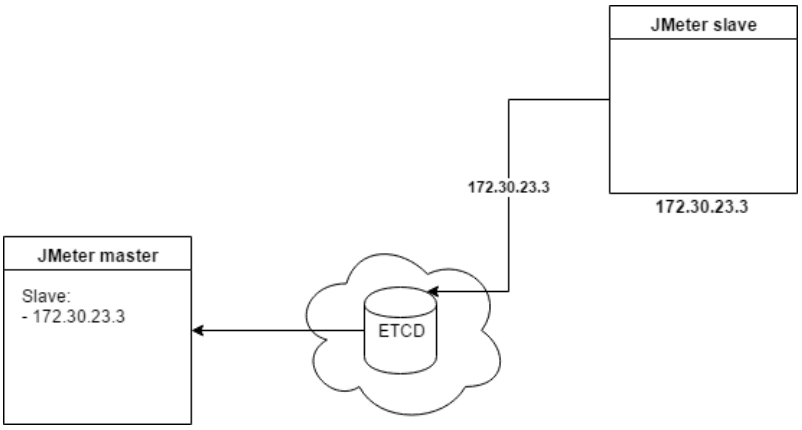

Figura 5.3. Lancio del primo slave

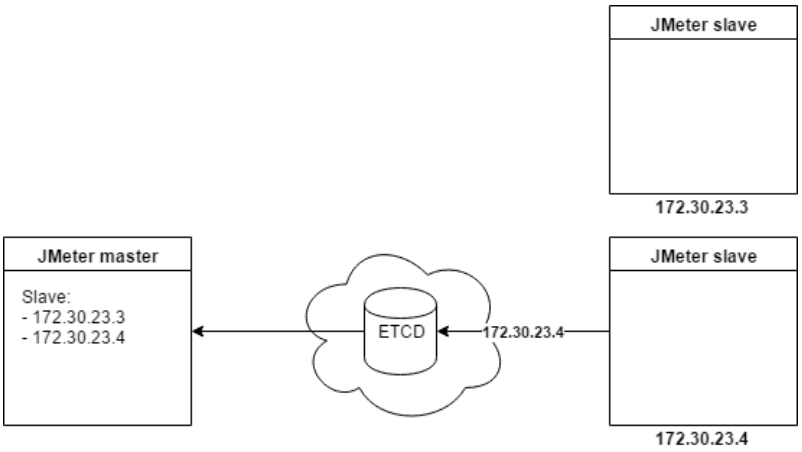

Figura 5.4. Lancio del secondo slave

3. A questo punto il database è popolato ed il master è a conoscenza di tutti gli slave disponibili, per cui può agevolmente lanciare i tests desiderati su tutti gli slave.

4. Una volta conclusi i tests per non consumare risorse e denaro è opportuno rimuovere del tutto il deployment di JMeter slave o per lo meno ridurne il numero di repliche. Quindi dato che la situazione sul cluster cambia è necessario che il database resti comunque aggiornato.

Figura 5.5. Lancio del terzo slave

Per far ciò, il container dello slave appena prima di essere rimosso elimina dal database la entry precedentemente inserita (cioè quella con il suo indirizzo IP), come indicato nelle figure 5.6 5.7 5.8.

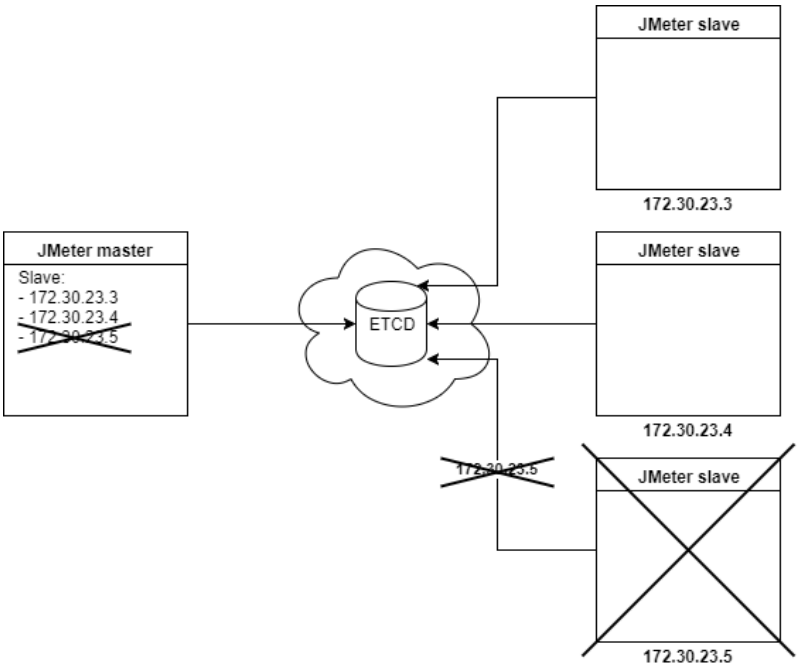

Figura 5.6. Rimozione del primo slave

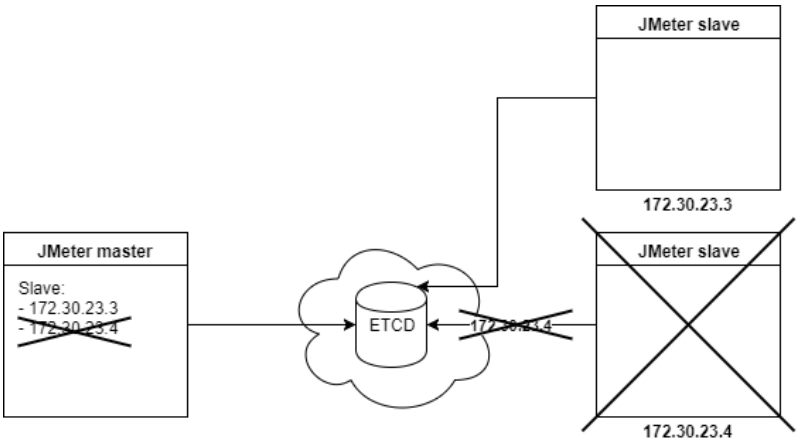

Figura 5.7. Rimozione del secondo slave

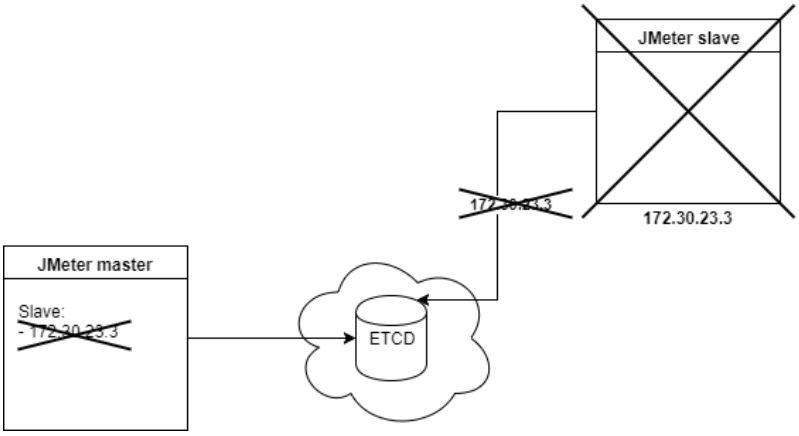

Figura 5.8. Rimozione del terzo slave

Capitolo 6

Implementazione di JMeter

6.1 Overview su ETCD

ETCD è un database distribuito che permette principalmente lo scambio di coppie chiave-valore, e affinché funzioni correttamente è necessario che tutti i membri di un cluster ETCD si conoscano tra loro.
Appunto per questo ETCD offre varie modalità di funzionamento (o configurazione):

- Static:
  gli indirizzi IP dei membri vengono inseriti manualmente al lancio del database. Per cui questo approccio è consigliato quando già a priori si conoscono tutti i membri del cluster.
- In altri casi non è possibile conoscere a priori gli indirizzi IP dei membri. Questo è comune quando si utilizzano i cloud providers oppure quando la rete usa un servizio DHCP. In questi casi la configurazione statica non è utilizzabile, per cui è necessario un servizio di discovery che è presente in due modalità:
  - ETCD Discovery: in questa modalità viene utilizzato un web service (disponibile all'indirizzo: https://discovery.etcd.io/) sia per creare un nuovo cluster, sia per agganciarsi ad un cluster esistente.
  - DNS Discovery: è consigliato utilizzare questo approccio quando si ha un servizio di DNS in cui poter memorizzare gli SRV records, che in sostanza sono i record utilizzati da ETCD per recuperare gli indirizzi IP dei membri.

Alla luce di ciò la configurazione più appropriata per il caso d'uso in questione è quella di utilizzare ETCD nella modalità ETCD discovery.

6.2 JMeter master

Step necessari per la creazione dell'immagine Docker di JMeter master:

- Creare il Dockerfile che specifica di quali parti è composta l'immagine.
- Creare gli script necessari per la configurazione ed installazione di JMeter.

6.2.1 Dockerfile

```
FROM ubuntu

MAINTAINER Luca Vacchetta


# Install useful commands for jmeter

RUN export http_proxy=http://proxy.reply.it:8080 && \

        apt-get clean && \

        apt-get update && \

        apt-get -qy --allow-unauthenticated install \

                        wget \

                        default-jre-headless \

                        telnet \

                        iputils-ping \

                        unzip


# Install jmeter

RUN   mkdir /jmeter \

        && cd /jmeter/ \

        && wget https://archive.apache.org/dist/jmeter/binaries/apache-jmeter-

            2.13.tgz \

        && tar -xzf apache-jmeter-2.13.tgz \

        && rm apache-jmeter-2.13.tgz \

        && mkdir /jmeter-plugins \

        && cd /jmeter-plugins/ \

        && wget https://jmeter-plugins.org/downloads/file/JMeterPlugins-ExtrasLibs

            -1.4.0.zip \
```

```
        && unzip -o JMeterPlugins-ExtrasLibs-1.4.0.zip -d /jmeter/apache-jmeter-
                2.13/

# Set Jmeter Home
ENV JMETER_HOME /jmeter/apache-jmeter-2.13/

# Add Jmeter to the Path
ENV PATH $JMETER_HOME/bin:$PATH

# Copy the entrypoint script in /run/ folder and add execution privilege
COPY launch_etcd.sh /run/
RUN chmod +x /run/launch_etcd.sh

# Copy shell script with the content of new command
# and add execution privilege
COPY jmeter_to_all_slaves.sh /run/
RUN chmod +x /run/jmeter_to_all_slaves.sh

# Install etcd
RUN export http_proxy=http://proxy.reply.it:8080 \
        && apt-get -qy --allow-unauthenticated install curl net-tools \
        && curl -L  https://github.com/coreos/etcd/releases/download/v2.1.0-rc.0/
                etcd-v2.1.0-rc.0-linux-amd64.tar.gz -o etcd-v2.1.0-rc.0-linux-
                amd64.tar.gz \
        && tar xzvf etcd-v2.1.0-rc.0-linux-amd64.tar.gz \
        && cd etcd-v2.1.0-rc.0-linux-amd64 \
        && mv etcd /usr/local/bin \
        && mv etcdctl /usr/local/bin

ENTRYPOINT ["/run/launch_etcd.sh"]

# Port to be exposed from the container for JMeter Master
EXPOSE 60000 2379 2380
```

Le scelte che hanno portato alla creazione di questo Dockerfile sono state le seguenti:

- Come immagine di partenza ho scelto Ubuntu, e non essendo specificata la versione si sottintende l'ultima versione di Ubuntu "containerizzata" presente sul docker hub pubblico.
- Prima di procedere con l'installazione di jmeter è stato necessario:
  - Configurare il proxy, dato che in questo caso l'immagine è stata creata su un server sotto la rete Reply.
  - Dato che la versione di Ubuntu containerizzata è ridotta "all'osso" è necessario installare i componenti indispensabili per il download e configurazione di JMeter, per cui si procede con il download e l'installazione di:
    * wget
    * default-jre-headless
    * telnet
    * iputils-ping
    * xterm
    * unzip
- Installazione di jmeter ed aggiunta alla variabile di sistema PATH la cartella contenente i binari di JMeter.
- Copia del bash script "`launch_etcd.sh`" (che verrà poi utilizzato come entrypoint del container) con l'aggiunta dei diritti di esecuzione nella cartella /run/ del container stesso.
- Copia dello script "`jmeter_to_all_slaves.sh`" (che verrà poi utilizzato come alias del comando jmeter), con, ovviamente, l'aggiunta dei diritti di esecuzione, nella cartella /run/ del container.
- Download ed installazione di ETCD con seguente spostamento dei binari etcd ed etcdctl nella cartella /usr/local/bin/ (inserita nella variabile PATH di sistema).
- Settaggio dell'entrypoint allo script: "`launch_etcd.sh`"
- Esposizione del master alla porte:
  - 60000: utilizzata da JMeter.
  - 2379 e 2380: utilizzate da ETCD.

6.2.2 Alias di JMeter

Bash script:

```
#!/bin/sh

# With awk script retrieve all IPs addresses of slaves,
# and insert them in an string
line_to_replace="remote_hosts="‘etcdctl ls | awk ’{print $0","}’ | awk ’{ gsub("/",
                 ""); print $1 }’ |  tr -d ’\n\r’ | awk ’{gsub(/,$/,""); print}’‘

# With another awk script replace the in jmeter configuration in order
# to insert all IPs addresses of slaves
awk -v newline="$line_to_replace" ’{ if (NR == 232) print newline; else print $0}’
                $JMETER_HOME/bin/jmeter.properties > /tmp/jmeter.properties
mv /tmp/jmeter.properties $JMETER_HOME/bin/jmeter.properties

# Now launch the real jmeter commmand
command="/jmeter/apache-jmeter-2.13/bin/jmeter"
$command $@
```

Questo script ha il compito di sostituire il comando jmeter, affinché vengano sfruttate le informazioni presenti sul database ETCD. Per cui prima di lanciare il comando jmeter “ufficiale” vengono eseguite le seguenti operazioni:

- Si recuperano tutte le entries (in particolare tutte le chiavi) presenti nel database ETCD, che rappresentano gli indirizzi IP degli slaves.
- Queste entries vengono parsificate tramite uno script awk in modo tale da ottenere tutti gli indirizzi IP su una stringa separati da una virgola.
- Viene rimpiazzata la riga 232 nel file di configurazione di Jmeter, che contiene la lista dei remote hosts cioè degli slaves, con la stringa appena ottenuta.

In questo modo lanciando JMeter con il parametro –r viene eseguito il test desiderato su tutti gli slaves specificati in quella riga. Ad esempio il comando:

- jmeter –n –t testcase.jmx –r

lancerà il test “testcase.jmx” su tutti gli slave agganciati allo stesso cluster ETCD del master.

6.2.3 Script di entrypoint

Bash script:

```
#!/bin/bash

# Definition of some useful environment variables
export LOCALIP=‘ifconfig eth0 2>/dev/null|awk ’/inet addr:/ {print $2}’|sed ’s/
                addr://’‘
export NAME_MASTER=master-$LOCALIP

# Creation of new etcd cluster
curl -w "\n" ’https://discovery.etcd.io/new?size=1’ > /mnt/clusterDiscoveryID

# Add the jmeter master to the cluster
etcd --name $NAME_MASTER --initial-advertise-peer-urls http://$LOCALIP:2380
     --listen-peer-urls http://$LOCALIP:2380 --listen-client-urls http://$LOCALIP:
     2379,http://127.0.0.1:2379 --advertise-client-urls http://$LOCALIP:2379
     --discovery ‘cat /mnt/clusterDiscoveryID‘ &

# Launch bash in order to have an interactive shell on master
exec bash --init-file <(echo "trap \"rm -f /mnt/clusterDiscoveryID\" SIGTERM EXIT;
                              alias jmeter=’/run/jmeter_to_all_slaves.sh’")
```

Questo script viene lanciato all'avvio del container (Pod) JMeter master ed in sostanza svolge le seguenti operazioni:

- Creazione di una variabile globale LOCALIP contenente l'indirizzo IP del container stesso. Per far ciò questa variabile è costruita utilizzando ifconfig ed un semplice script awk per estrarre il solo indirizzo IP dall'output di ifconfig.
- Creazione di un'altra variabile globale contente il nome di questo nodo all'interno del cluster ETCD. Dato che questo nome deve essere univoco all'interno del cluster ha al suo interno l'indirizzo IP estratto precedentemente.
- Creazione di un nuovo cluster ETCD, passando come parametro: size=1 al Web service discovery.etcd.io, questo indica che si vuol creare un nuovo cluster

con un solo membro al suo interno, perché inizialmente solo il master è attivo. Successivamente la url a cui è disponibile il servizio di ETCD discovery di quel cluster viene memorizzata in un file all'interno di una cartella comune ai container master e slave.

- Lancio del database ETCD in background.
- Infine per avere la modalità interattiva sul master, tramite il comando exec viene lanciata una nuova bash avente nel file di inizializzazione due comandi utili per i seguenti scopi:
  - Eliminare dalla cartella comune il file contenente il nome del cluster ETCD al momento della rimozione del container; questo per evitare che ulteriori repliche dello slave vadano ad agganciarsi ad un master che in realtà non esiste più.
  - Aggiungere un alias al comando jmeter affinché venga utilizzato lo script descritto precedentemente.

6.3 JMeter slave

Gli step necessari per la creazione di JMeter slave sono analoghi a quelli del master, vale a dire:

- Creare il Dockerfile che specifica di quali parti è composta l'immagine.
- Creare lo script di entrypoint necessario per la configurazione ed installazione di JMeter.

6.3.1 Dockerfile

```
FROM ubuntu

MAINTAINER Luca Vacchetta


# Set the http proxy

#RUN export http_proxy=http://proxy.reply.it:8080


# Install useful commands for jmeter

RUN export http_proxy=http://proxy.reply.it:8080 && \
```

```
        apt-get clean && \
        apt-get update && \
        apt-get -qy --allow-unauthenticated install \
                        wget \
                        default-jre-headless \
                        telnet \
                        iputils-ping \
                        unzip

# Install jmeter
RUN   mkdir /jmeter \
        && cd /jmeter/ \
        && wget https://archive.apache.org/dist/jmeter/binaries/apache-jmeter-
            2.13.tgz \
        && tar -xzf apache-jmeter-2.13.tgz \
        && rm apache-jmeter-2.13.tgz \
        && mkdir /jmeter-plugins \
        && cd /jmeter-plugins/ \
        && wget https://jmeter-plugins.org/downloads/file/JMeterPlugins-
            ExtrasLibs-1.4.0.zip \
        && unzip -o JMeterPlugins-ExtrasLibs-1.4.0.zip -d /jmeter/
                    apache-jmeter-2.13/

# Set Jmeter Home
ENV JMETER_HOME /jmeter/apache-jmeter-2.13/

# Add Jmeter to the Path
ENV PATH $JMETER_HOME/bin:$PATH

# Install etcd
RUN export http_proxy=http://proxy.reply.it:8080 \
        && apt-get -qy --allow-unauthenticated install curl net-tools \
        && curl -L  https://github.com/coreos/etcd/releases/download/v2.1.0-rc.0/
```

```
                etcd-v2.1.0-rc.0-linux-amd64.tar.gz -o etcd-v2.1.0-rc.0-linux
                -amd64.tar.gz \
        && tar xzvf etcd-v2.1.0-rc.0-linux-amd64.tar.gz \
        && cd etcd-v2.1.0-rc.0-linux-amd64 \
        && mv etcd /usr/local/bin \
        && mv etcdctl /usr/local/bin

# Ports to be exposed from the container for JMeter Slaves/Server
EXPOSE 1099 50000 2379 2380

# Copy the script file for the entrypoint
COPY launch_etcd_and_jmeter_signal_handler.sh /run/

# Add execute mode to script bash
RUN chmod +x /run/launch_etcd_and_jmeter_signal_handler.sh

# Run jmeter slave
ENTRYPOINT ["/run/launch_etcd_and_jmeter_signal_handler.sh"]
```

Si capisce immediatamente che il Dockerfile è molto simile a quello del master infatti le uniche cose che variano sono le seguenti:

- Non viene installato il pacchetto xterm in quanto l'interfaccia grafica non è necessaria nello slave.
- Le porte di ascolto sono diverse, in quanto lo slave sta in ascolto su porte diverse rispetto al master.
- Infine ovviamente l'entrypoint corrisponde ad uno script diverso rispetto a quello passato dal master.

6.3.2 Script di entrypoint

Bash script:

```
#!/usr/bin/env bash
set -x
```

```
pid=0

#Sig TERM handler
term_handler(){
  if [ $pid -ne 0 ]; then
    etcdctl rmdir $LOCALIP
    kill -SIGTERM "$pid"
    wait "$pid"
  fi
  exit 143; # 128 + 15 -- SIGTERM
}

# setup handlers
# on callback, kill the last background process, which is `tail -f /dev/null`
# and execute the specified handler
trap 'term_handler' SIGTERM

#Definition of some useful environment variables
export LOCALIP=`ifconfig eth0 2>/dev/null|awk '/inet addr:/ {print $2}'|sed
               's/addr://'`
export NAME_SLAVE=slave-$LOCALIP

#Look for the master to activate and insert a clusterID in mnt folder
#(in sostanza fino a quando quella cartella e' vuota si aspetta)
while [ -z "$(ls -A /mnt/)" ]
do
  sleep 1
done

#useful sleep for synchonization with etcd peer node
sleep 5

#Add this node to etcd cluster
```

```
etcd --name $NAME_SLAVE --initial-advertise-peer-urls http://$LOCALIP:2380
     --listen-peer-urls http://$LOCALIP:2380 --listen-client-urls http://$LOCALIP:
     2379,http://127.0.0.1:2379 --advertise-client-urls http://$LOCALIP:2379
     --discovery ‘cat /mnt/clusterDiscoveryID‘ &

#Sleep useful for synchronization
sleep 3
etcdctl mk $LOCALIP OK
$JMETER_HOME/bin/jmeter-server -Dserver.rmi.localport=50000 -Dserver_port=1099
                               -H proxy.reply.it -P 8080 &

pid="$!"

wait $pid
```

Questo script viene lanciato all'avvio del container ed in pratica ha il compito di avviare il server di JMeter affinché sia in grado di rispondere alle richieste. Però prima di avviare JMeter lo slave deve comunicare al master, tramite l'uso di ETCD, il suo indirizzo IP. Per cui sono necessari i seguenti step:

- Estrazione del solo indirizzo IP e salvataggio di esso in una variabile globale (LOCALIP).
- Successivamente è presente un loop di attesa, che non attende altro che il master crei un cluster ETCD.
  NB: questo loop è utile soltanto nel caso in cui l'utente erroneamente lanci prima lo slave che il master, per cui in quel caso lo slave non saprebbe a che cluster ETCD agganciarsi.
- Dopodiché è necessario inserire una sleep di 5 secondi (la durata dello sleep è stata ricavata in seguito a prove pratiche e misure sul campo) utile per la sincronizzazione con il master.
  NB: anche in questo caso la sleep è utile soltanto nel caso in cui l'utente lanci prima lo slave che il master, per cui è necessario dare del tempo al master affinché si avvii in modo tale da riuscire a ricevere le richieste.
- Avvio di ETCD mediante l'aggancio al cluster ETCD creato dal master.

- Sleep di 3 secondi necessaria per dare il tempo allo slave di avviare correttamente ETCD (anche questo tempo è ricavato da prove pratiche e misure sul campo).
- Inserimento nel database di una nuova coppia chiave-valore contenente l'indirizzo IP dello slave stesso come chiave.

Nella figura 6.1 è riportato il diagramma delle tempistiche appena descritte.
A questo punto sono ancora necessari due step:

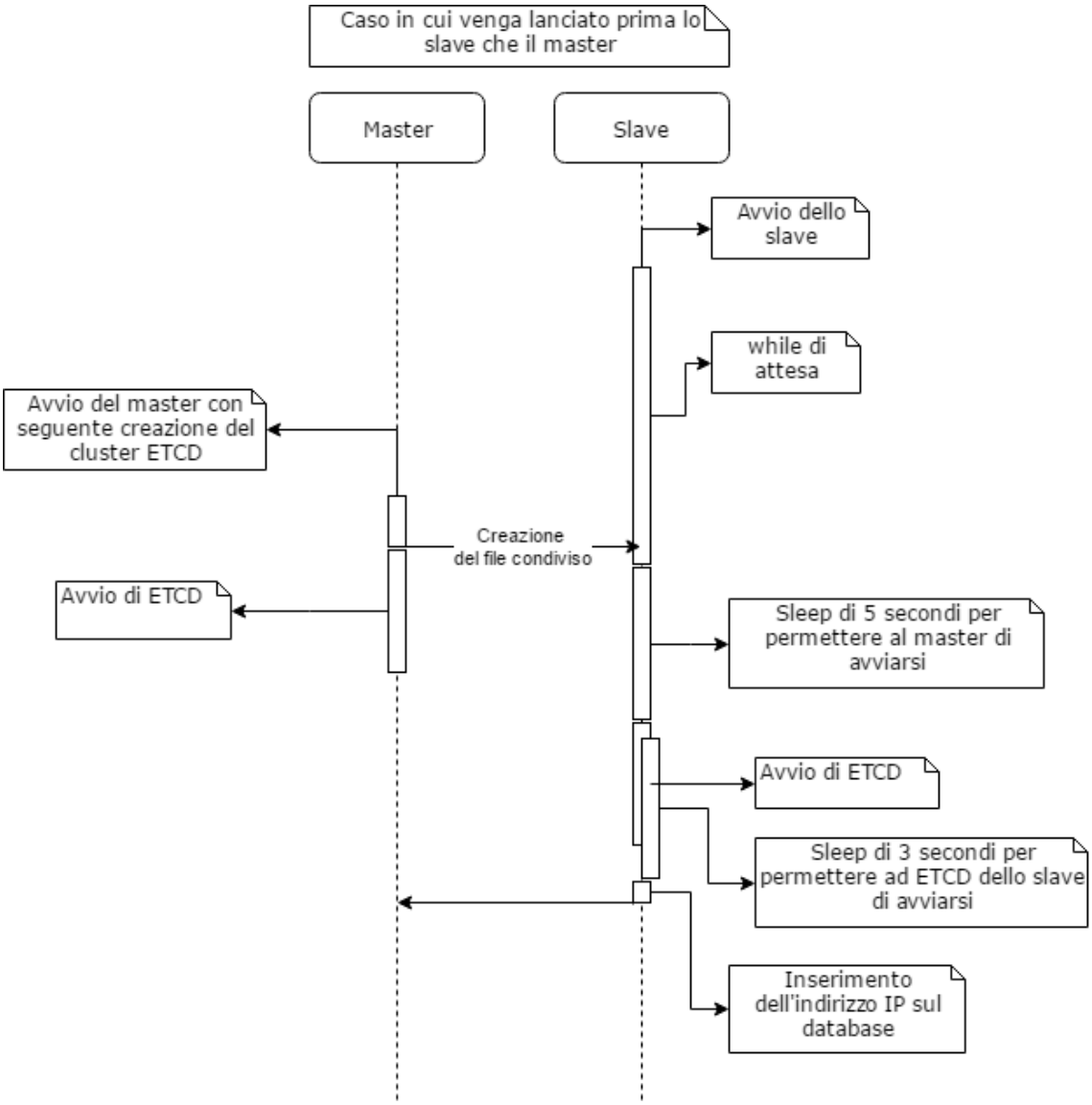

Figura 6.1. Tempistiche di JMeter nel caso in cui venga lanciato prima lo slave che il master

- Avviare il server di JMeter affinché sia in grado di rispondere alle richieste del master.

- Registrare il comando che rimuove dal database ETCD la entry key-value (precedentemente inserita) da eseguire solamente al termine del pod, per far ciò è stato necessario seguire i seguenti passi:

  – Dato che il server è lanciato in background è necessario creare un'attesa infinita, per cui è indispensabile attendere che il processo legato al server di JMeter termini, perché altrimenti il container uscirebbe immediatamente (cioè farebbe una exit) senza servire alcuna richiesta dal master.

  – Definire una funzione `term_handler()` che deve essere richiamata ogni qualvolta si riceva un segnale di terminazione (**SIG TERM**). In sostanza questa funzione ha il compito di eliminare la entry contenente l'indirizzo IP del pod stesso dal database ETCD ed infine di terminare cioè fare una exit con il codice 143 che equivale al codice del **SIGNAL TERMINATION**.

  – Infine affinché il signal handler appena definito venga richiamato è necessario registrare la chiamata a tale funzione mediante l'uso di una trap che intercetti il segnale **SIGNAL TERMINATION**.
  NB:
  Il fatto che si debba intercettare il SIGNAL TERMINATION è una caratteristica tipica dei container, perché a differenza della consueta versione di Ubuntu, ai containers non è permesso utilizzare i run level (che in sostanza vanno ad aggiungere un ulteriore strato di virtualizzazione), per cui, per poter eseguire delle operazioni di clean-up al termine dell'esecuzione del Pod, Docker & Kubernetes mettono a disposizione questo meccanismo:

    * Al momento in cui Kubernetes riceve il comando di interruzione di un Pod, il demone di Kubernetes invia a tutti i containers appartenenti a quel Pod un SIG TERM, dopodiché avvia un countdown.
    * A questo punto i vari containers appartenenti al Pod ricevono il SIG TERM che hanno possibilità di intercettare, per cui hanno il tempo necessario per svolgere le funzioni di clean-up.
    * Dopodiché se al termine del countdown (della durata di 30 secondi), il Pod è ancora attivo, cioè i containers appartenenti ai Pod dopo aver eseguito le operazioni di clean-up non si sono disattivati, il demone di Kubernetes provvederà ad inviare un ulteriore segnale, ma questa volta invierà a tutti i containers del Pod il SIG KILL che non può

essere mascherato, come viene mostrato in figura 6.2; per cui verrà forzata la chiusura dei vari containers.

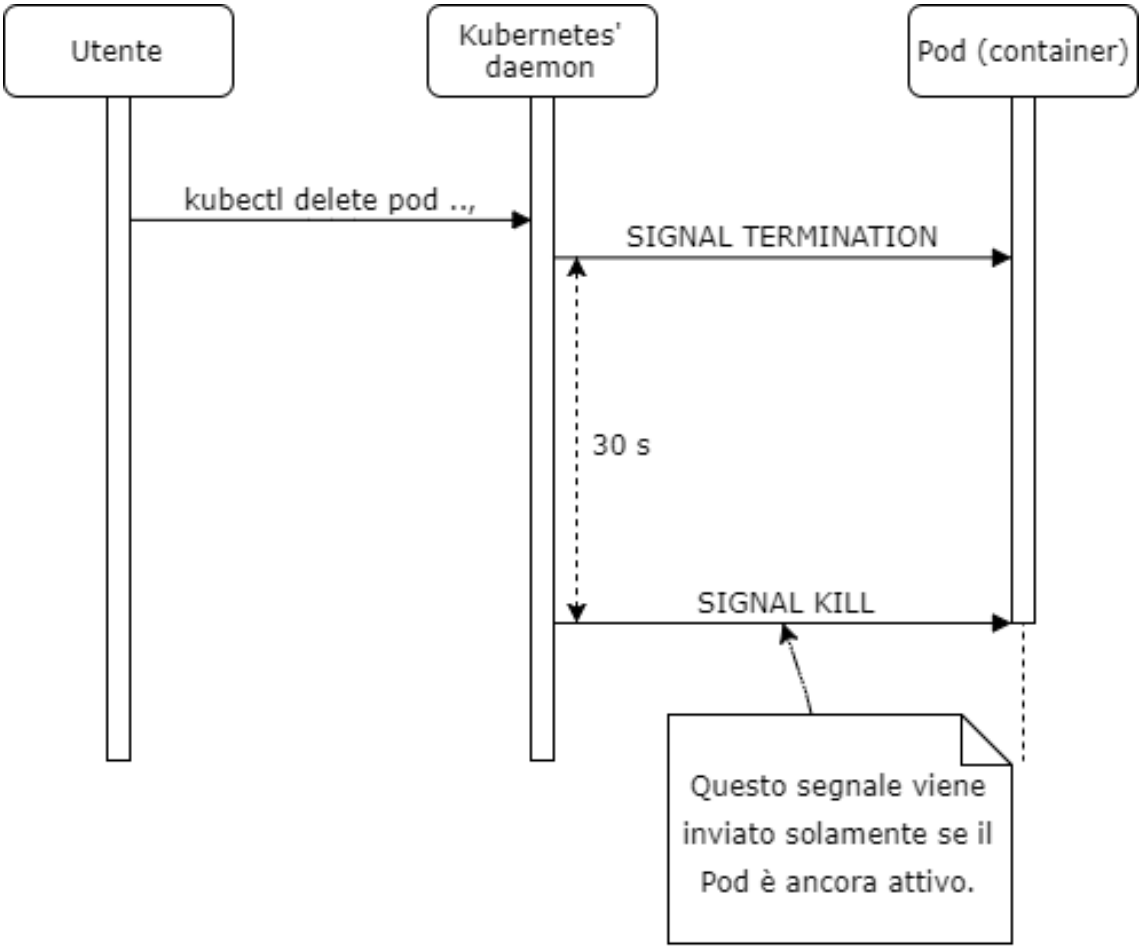

Figura 6.2. Schema di funzionamento della chiusura di un Pod

6.4 Conclusioni

Infine per creare le immagini a partire dai files appena descritti e renderle utilizzabili da Kubernetes è necessario seguire i seguenti passi:

- Creare le immagini con l'apposito comando Docker, ad esempio:
  - docker build —tag=jm-master .

  in sostanza con questo comando si va a creare l'immagine di JMeter master e questa immagine viene memorizzata in locale. In particolare questo comando va lanciato dalla stessa cartella in cui è presente il Dockerfile di JMeter master

(altrimenti al posto del '.' finale va specificato il path in cui è presente il suddetto Dockerfile).

- A questo punto è necessario caricare queste immagini sul Docker Hub, per cui se non si utilizza un Docker Hub personale è necessario registrarsi sul Docker Hub offerto dalla piattaforma Docker.
- Successivamente per poter caricare queste immagini sul Docker Hub bisogna aggiungerci un tag alle immagini che sia coerente con il repository sul Docker Hub appena creato.
  Per far ciò è sufficiente lanciare il seguente comando:
  - docker tag local-image:tagname repository-name:tagname
- Infine bisogna effettuare l'upload delle immagini sul repository remoto, per cui basta seguire questi due step:
  - Loggarsi al proprio repository remoto, con il seguente comando:
    - docker login
  - Effettuare l'upload delle immagini, in questo modo:
    - docker push <reponame>:<tagname>

Capitolo 7

Kubernetes in multicloud

Per la creazione di un unico cluster Kubernetes in multi-cloud, in questo caso sui cloud providers:

- Softlayer.
- Amazon Web Services (AWS).

Si è deciso di utilizzare Terraform con il fine di rendere questa operazione automatizzabile e facilmente replicabile.
Per cui è stato necessario seguire questi passaggi:

- Creare un account Softlayer e AWS entrambi abilitati a creare delle macchine tramite API.
- Creare il file di configurazione di Terraform necessario per descrivere l'infrastruttura desiderata.

7.1 Configurazione

Affinché le macchine create rispettino le dipendenze stabilite dal file di configurazione Terraform va a creare un grafo delle risorse, che in questo caso corrisponde in modo logico a quello riportato in figura 7.1.
Pertanto per chiarezza espositiva ho deciso di commentare successivamente le risorse create in modo separato, vale a dire che ho diviso il file di configurazione in questi tre macroblocchi, che corrispondono alle seguenti tre macchine che si vanno a creare:

Figura 7.1. Grafo delle risorse creato da Terraform

- Un virtual server contenente il nodo master di Kubernetes (Softlayer).
- Un virtual server contenente un nodo worker di Kubernetes (Softlayer).
- Un virtual server contenente un nodo worker di Kubernetes (AWS).

7.1.1 Master su Softlayer

```
provider "softlayer" {
  username = "YOUR_USERNAME"
  api_key  = "YOUR_API_KEY"
}

resource "softlayer_ssh_key" "my_key" {
    label = "${var.user_string}"
    public_key = "${file("~/.ssh/id_rsa.pub")}"
}

resource "softlayer_virtual_guest" "master" {
    hostname          = "kube-master-${var.user_string}"
```

```
domain            = "bluereply.it"
os_reference_code = "CENTOS_7_64"
datacenter        = "${var.datacenter}"
cores             = 1
memory            = 1024
hourly_billing    = true
local_disk        = true

ssh_key_ids = [
    "${softlayer_ssh_key.my_key.id}"
]

connection {
    private_key = "${file("~/.ssh/id_rsa")}"
}

provisioner "file" {
  source      = "conf/virt7-docker-common-release.repo"
  destination = "/etc/yum.repos.d/virt7-docker-common-release.repo"
}

provisioner "file" {
  source      = "setup_master.sh"
  destination = "/tmp/setup_master.sh"
}

provisioner "file" {
  source      = "script-on-master.sh"
  destination = "/run/script-on-master.sh"
}

provisioner "remote-exec" {
  inline = [
```

```
      "chmod +x /tmp/setup_master.sh",
      "chmod +x /run/script-on-master.sh",
      "/tmp/setup_master.sh ${self.ipv4_address}",
    ]
  }

}
```

Commenti:

- Innanzitutto è necessario inserire le proprie credenziali nel campo provider.
- Affinché il file di configurazione e lo script di setup sia trasmesso in modo sicuro è indispensabile seguire questi step:
  - Creare una chiave ssh con il seguente comando:
    - ssh-keygen -t rsa
  - Creare la risorsa softlayer_ssh_key (sempre mediante Terraform) che in sostanza trasmette a Softlayer la parte pubblica della chiave appena creata, che verrà usata in ricezione dal cloud provider per autenticare il mittente.
- Per quanto riguarda il virtual server sono state fatte le seguenti scelte:
  - Centos 7 come sistema operativo.
  - Il datacenter su cui creare questa macchina è inserito tramite una variabile di terraform (definita nel file vars.tf).
  - Un core.
  - Un GB di RAM.
  - Tariffazione su base oraria.
  - Passaggio di uno script (tramite provisioner 'file') che verrà eseguito all'avvio del server, che in pratica va ad installare Kubernetes configurandolo come master.
  - Per ultimo è stato passato uno script che verrà richiamato al termine della creazione delle tre macchine.

Script finale

Premesse

1. Dato che un possibile modo per poter assegnare un Pod ad uno specifico nodo è quello di forzare il Pod ad essere lanciato solo su quei nodi che abbiano una label (etichetta) particolare con un preciso valore (come indicato nella figura 7.2).
   É necessario eseguire la seguente operazione:

   - Assegnare ad ogni nodo worker presente sul cluster la label chiamata, ad esempio, cloud-provider con uno di questi valori:
     - Softlayer
     - AWS

     a seconda se il nodo appartenga al cloud di IBM oppure di Amazon.

   NB:
   questa fase di "etichettamento" va fatta manualmente solamente perché, al momento, la feature che permette di inserire delle label ad un nodo in fase di creazione del nodo stesso è presente solanto nella versione alpha di Kubernetes (cioè non stabile).

2. L'altro problema riguarda il fatto che l'indirizzo pubblico dei servers di Amazon non è settato su nessuna interfaccia presente sui servers, questo probabilmente perché i servers di Amazon sono dietro un Reverse Proxy. Quindi il modulo Flanneld di Kubernetes, che si occupa del networking, crea il tunnel mettendo come endpoint del server di Amazon un indirizzo privato, per cui la comunicazione tra pod non funziona correttamente.
   Per ovviare a questo problema è necessario modificare manualmente l'endpoint di quel tunnel mettendoci il corretto IP pubblico.

Soluzione

Per avere una soluzione automatizzata a questi problemi si possono inserire i vari comandi da effettuare in uno script che dovrà essere eseguito al termine della creazione delle tre macchine sul nodo master di Kubernetes, appunto per questo in fase di creazione del nodo master gli viene passato il seguente script:

```
#!/bin/bash


MASTER_IP_SOFTLAYER=$1
```

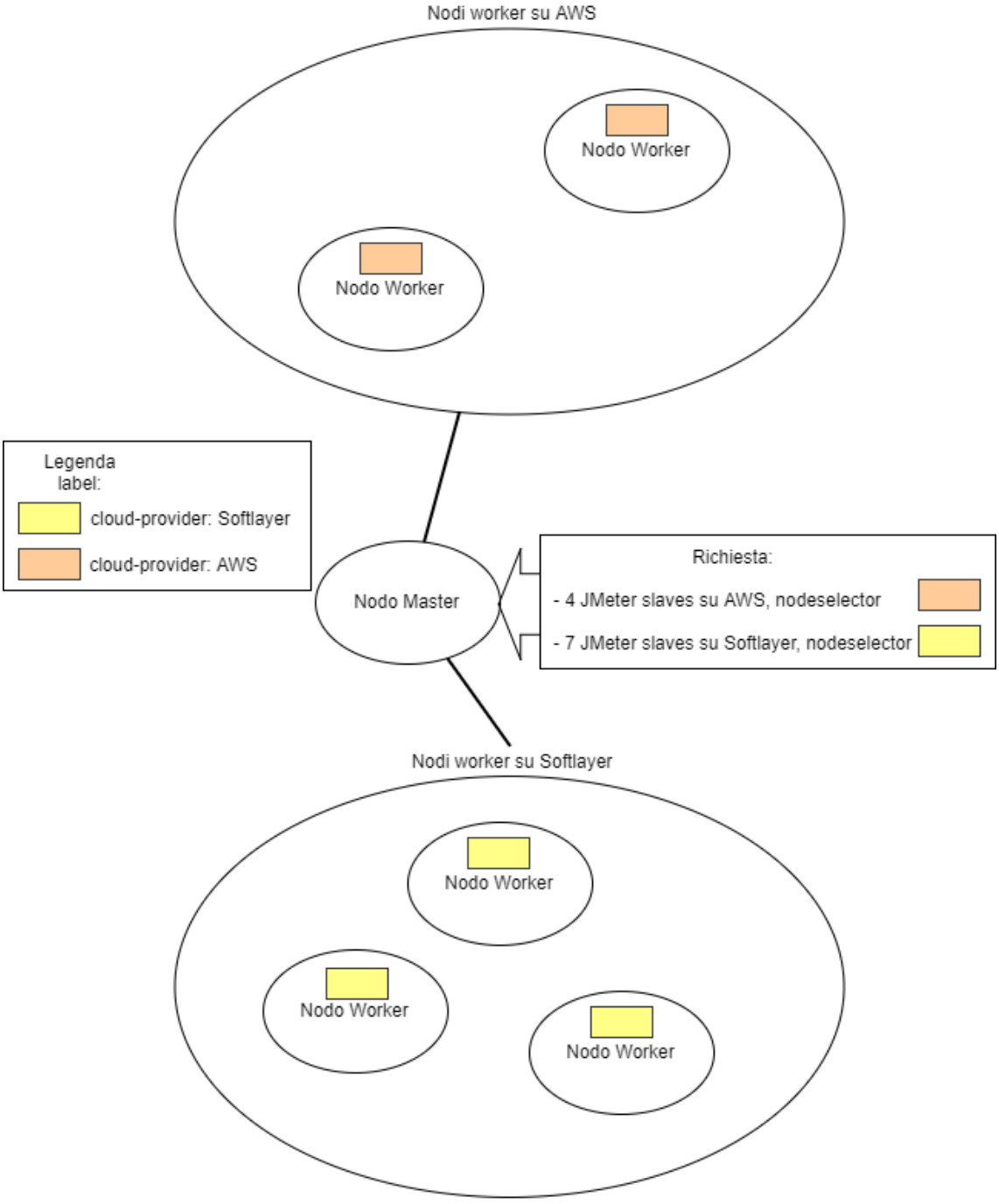

Figura 7.2. Schema logico etichettamento nodi

```
WORKER_IP_SOFTLAYER=$2

WORKER_IP_AWS=$3


# add labels to worker nodes

kubectl label nodes $WORKER_IP_SOFTLAYER cloud-provider=Softlayer

kubectl label nodes $WORKER_IP_AWS cloud-provider=AWS
```

```
# now replace the endpoint of the tunnel with aws server because probably Amazon
# use a reverse proxy, so the endpoint of the tunnel is with a private IP
etcdctl ls kube-centos/network/subnets > output_etcd

for i in $(seq 1 3); do
  awk 'BEGIN{FS="/"} NR=='$i' {print $5}' output_etcd >> output_filtrato
done

for i in $(seq 1 3); do
  subnet=`awk 'NR=='$i' {print $0}' output_filtrato`
  etcdctl get kube-centos/network/subnets/$subnet > output_etcdctl
  publicIP=`awk -F "\"" '{print $4}' output_etcdctl`
  if [ "$publicIP" != "$MASTER_IP_SOFTLAYER" ] &&
     [ "$publicIP" != "$WORKER_IP_SOFTLAYER" ]
  then
    echo "cambio IP tunnel "$publicIP", "$subnet
    firstPart=`awk -F \${publicIP} '{print $1}' output_etcdctl`
    secondPart=`awk -F \${publicIP} '{print $2}' output_etcdctl`
    entryUpdated=$firstPart$WORKER_IP_AWS$secondPart

    echo $entryUpdated > newEntry
    etcdctl set kube-centos/network/subnets/$subnet "`cat newEntry`"
  else
    echo "ok"
  fi
done

rm -f newEntry
rm -f output_etcdctl
rm -f output_filtrato
rm -f output_etcd
```

```
# now it need to restart flanneld service on all servers
systemctl restart flanneld
```

In dettaglio questo script svolge le seguenti operazioni:

- Aggiunge un etichetta che indica l'appartenenza ad uno specifico cloud-provider ad ogni nodo worker presente sul cluster.
- Cambia l'endpoint del tunnel diretto verso il nodo di Amazon mettendoci l'indirizzo IP pubblico di quel nodo.
- Infine riavvia Flanneld (cioè il servizio che offre il networking in Kubernetes).

7.1.2 Slave su Softlayer

```
resource "softlayer_virtual_guest" "worker" {
    count             = "${var.worker_count}"
    hostname          = "kube-worker-${var.user_string}-${count.index}"
    domain            = "bluereply.it"
    os_reference_code = "CENTOS_7_64"
    datacenter        = "${var.datacenter}"
    cores             = 1
    memory            = 1024
    local_disk        = true

    ssh_key_ids = [
        "${softlayer_ssh_key.my_key.id}"
    ]

    connection {
        private_key = "${file("~/.ssh/id_rsa")}"
    }

    provisioner "file" {
      source      = "conf/virt7-docker-common-release.repo"
      destination = "/etc/yum.repos.d/virt7-docker-common-release.repo"
```

```
    }

    provisioner "file" {
      source      = "setup_worker.sh"
      destination = "/tmp/setup_worker.sh"
    }

    provisioner "remote-exec" {
      inline = [
        "chmod +x /tmp/setup_worker.sh",
        "/tmp/setup_worker.sh ${softlayer_virtual_guest.master.ipv4_address}
                              ${self.ipv4_address}",
      ]
    }
}
```

Commenti:

- Analogamente a prima è stato scelto un virtual server con capacità di processing ridotte (dato viene utilizzato soltanto come prova).
- Il campo count indica quante repliche si vogliono avere dello stesso server, in questo caso il numero repliche viene letto da una variabile definito nel file vars.tf.
- Infine allo script che verrà eseguito all'avvio del server gli viene passato come parametro l'indirizzo IP pubblico del nodo master, questo perché, per una corretta installazione, è necessario che il nodo worker conosca anticipatamente l'indirizzo IP del master.

7.1.3 Slave su AWS

```
provider "aws" {
  access_key = "YOUR_ACCESS_KEY"
  secret_key = "YOUR_SECRET_KEY"
  region     = "us-east-1"
```

```
}

# Our default security group to access
# the instances over SSH
resource "aws_security_group" "ssh" {
  name        = "ssh_security"
  description = "Security group of AWS machine"

# Permit access from anywhere
  ingress {
    from_port   = 0
    to_port     = 0
    protocol    = -1
    cidr_blocks = ["0.0.0.0/0"]
  }

  # outbound internet access
  egress {
    from_port   = 0
    to_port     = 0
    protocol    = "-1"
    cidr_blocks = ["0.0.0.0/0"]
  }
}

resource "aws_key_pair" "deployer" {
  key_name   = "key-aws-ssh"
  public_key = "${file("~/.ssh/id_rsa.pub")}"
}

resource "aws_instance" "worker" {
    ami = "${var.aws_ami}" # questo codice specifica CENTOS_7_64
                           # del datacenter us-east-1
```

```
instance_type = "t2.micro"
key_name      = "${aws_key_pair.deployer.id}"

# Our Security group to allow SSH access
vpc_security_group_ids = ["${aws_security_group.ssh.id}"]

provisioner "file" {
  source      = "conf/virt7-docker-common-release.repo"
  destination = "/tmp/virt7-docker-common-release.repo"

  connection {
    user = "centos"
    private_key = "${file("id_rsa.pem")}"
    agent = false
    timeout = "1m"
  }
}

provisioner "file" {
  source      = "setup_worker.sh"
  destination = "/tmp/setup_worker.sh"

  connection {
    user = "centos"
    private_key = "${file("id_rsa.pem")}"
    agent = false
    timeout = "1m"
  }
}

provisioner "remote-exec" {
  connection {
    user = "centos"
```

```
      private_key = "${file("id_rsa.pem")}"
      agent = false
      timeout = "1m"
    }

    inline = [
      "sudo mv /tmp/virt7-docker-common-release.repo /etc/yum.repos.d/
             virt7-docker-common-release.repo",
      "chmod +x /tmp/setup_worker.sh",
      "sudo /tmp/setup_worker.sh ${softlayer_virtual_guest.master.ipv4_address}
                                 ${self.public_ip}"
    ]
  }

  provisioner "local-exec" {
    command = "./local-script.sh ${softlayer_virtual_guest.master.ipv4_address}
               ${softlayer_virtual_guest.worker.ipv4_address} ${self.public_ip}"
  }

}
```

Commenti:

- In questo caso le principali differenze rispetto allo slave creato su Softlayer sono di sintassi, ad esempio:
  - Il sistema operativo viene specificato tramite una stringa che indica un preciso SO in un preciso datacenter, anche in questo caso la stringa viene letta da una variabile definita nel file vars.tf.
  - Le caratteristiche della macchina da creare sono indicate nel campo "instance_type", in questo caso è stato il tipo: "t2.micro", che in sostanza signigica:
    - Server virtuale.
    - Un core.
    - Un GB di RAM.

- Dopodiché, come in precedenza, gli viene passato uno script che riceve come parametro l'indirizzo IP del master con il fine di installare Kubernetes e configurarlo come worker.
- Infine al termine della creazione di questa macchina (che è anche l'ultima macchina che verrà creata) viene eseguito in locale uno script che svolge le seguenti funzioni:
  - Richiama e fa eseguire lo script, descritto in precedenza, sul nodo master di Kubernetes.
  - Riavvia il servizio flanneld nell'altro nodo worker di Kubernetes.

Capitolo 8

Deploy di JMeter in multi-cloud

8.1 Elementi fondamentali

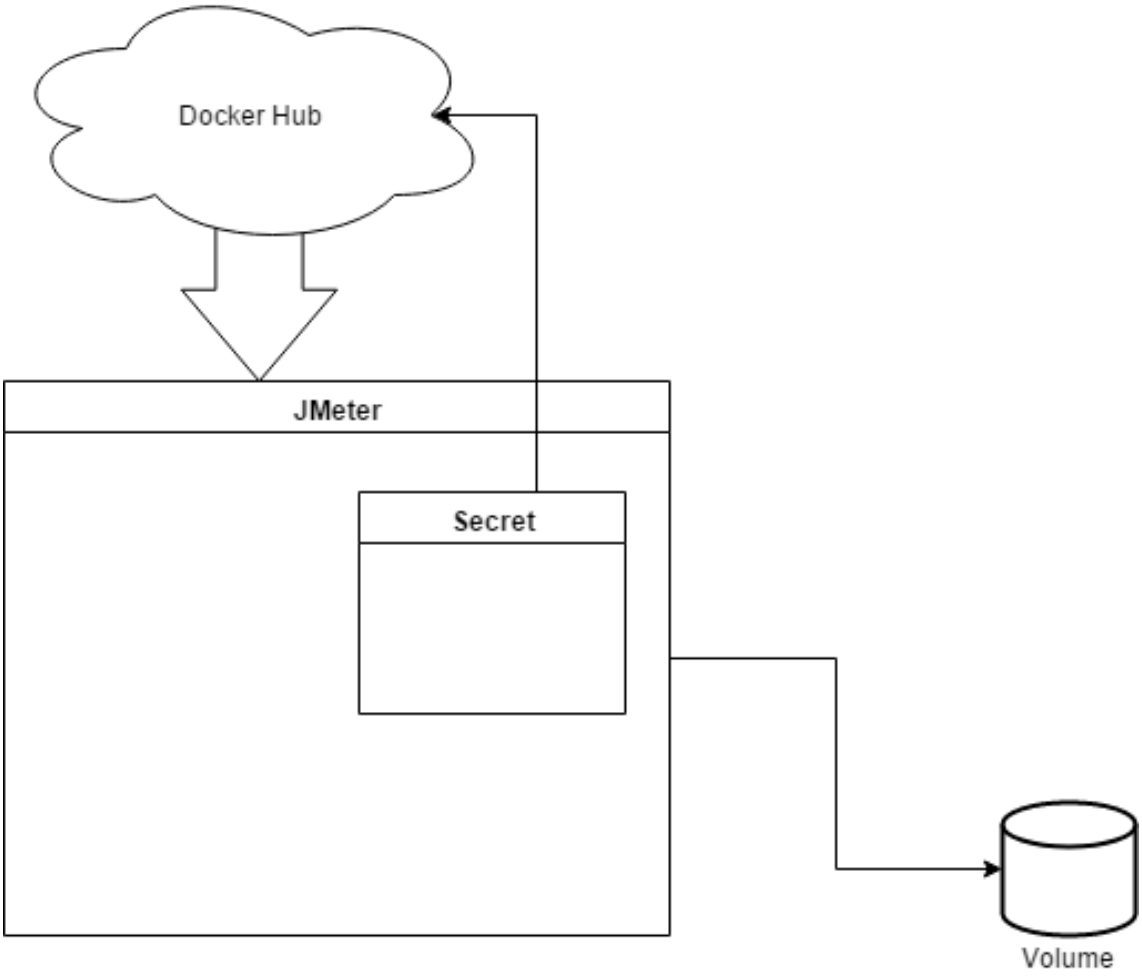

Figura 8.1. Schema di principio

Da come mostrato nella figura 8.1 si capisce che sono necessari almeno tre elementi affinché sia il master che gli slaves di JMeter siano deployati su Kubernetes:

- Secret
- Volume
- Deployment

NB:
sia il Secret che il Volume sono risorse condivise tra master e slave.

8.1.1 Secret

Il secret è una componente che offre la possibilità di memorizzare in modo sicuro dei dati sensibili in modo da renderli facilmente usufruibili, come ad esempio delle credenziali. Infatti in questo caso si va a creare un secret che contiene le credenziali al repository privato sul Docker Hub, cosicché Kubernetes abbia i permessi di scaricare le immagini desiderate da quel repository.
Per creare questo secret è stato necessario lanciare il seguente comando:

- kubectl create secret docker-registry regsecret —docker-server=<your-registry-server> —docker-username=<your-name> —docker-password=<your-pwd> —docker-email=<your-email>

8.1.2 Volume

In generale i volumi vengono utilizzati per memorizzare delle informazioni in modo permanente, cioè hanno una durata temporale che è indipendente dalla vita del pod, ed anche per consentire la condivisione di dati/informazioni tra pod.
In questo caso il volume è utilizzato unicamente per il secondo motivo, vale a dire lo scambio di informazioni tra pod diversi. Infatti affinché il database ETCD (in modalità ETCD discovery) funzioni correttamente è necessario che tutti i membri del cluster ETCD conoscano la url del servizio di discovery di quel cluster.
Per far ciò è stato necessario deployare su kubernetes due componenti distinte:

- PersistentVolume: è la componente che alloca fisicamente l'area di immagazzinamento sul cluster. In questo caso ho deciso di creare un volume di tipo

HostPath, dato che non si aveva a disposizione un NFS (Network File System). In sostanza un volume di tipo HostPath è in grado di condividere le informazioni solo tra i pod che sono presenti sullo stesso nodo in cui è presente il volume (nonostante ciò per il mio ambiente di prova va benissimo questo tipo di volume siccome il cluster è di dimensioni ridotte).
Quindi per creare questo volume è stato necessario deployare su Kubernetes un Persistent Volume che si basa sul seguente file di configurazione:

```
    kind: PersistentVolume
apiVersion: v1
metadata:
  name: jmeter-pv-storage
  labels:
    type: local
spec:
  capacity:
    storage: 200Mi
  accessModes:
    - ReadWriteOnce
  hostPath:
    path: "/mnt/jmeter-volume"
```

- PersistentVolumeClaim: è la componente che richiede l'accesso al volume perché in Kubernetes i pod non accedono direttamente ai volumi ma ci accedono tramite l'uso di claim (cioè richieste/reclami di accesso) per motivi di sincronizzazione.
  Quindi per creare il PersistentVolumeClaim è necessario deployare sul cluster il seguente file di configurazione:

```
    kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: jmeter-pv-claim
spec:
  accessModes:
    - ReadWriteOnce
```

```
  resources:
    requests:
      storage: 100Mi
```

8.1.3 Deployment di JMeter master

Nel file di Deployment viene specificato come e cosa si vuole creare sul cluster di Kubernetes. Nel caso in questione è stato necessario editare il seguente file:

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: jm-master
  labels:
    app: jm-master
spec:
  replicas: 1
  selector:
    matchLabels:
      app: jm-master
  template:
    metadata:
      labels:
        app: jm-master
    spec:
      volumes:
      - name: jmeter-pv-storage
        persistentVolumeClaim:
          claimName: jmeter-pv-claim
      containers:
      - name: jm-master
        image: tesijmeterkubernetes/jmeter:master
        ports:
        - containerPort: 60000
```

```
      name: jmeter
    - containerPort: 2379
      name: etcd1
    - containerPort: 2380
      name: etcd2
    stdin: true
    tty: true
    volumeMounts:
    - mountPath: "/mnt/"
      name: jmeter-pv-storage
  imagePullSecrets:
  - name: regsecret
```

Commenti:

I principali elementi di cui si compone il file di deployment sono i seguenti:

- Name:
  rappresenta il nome del deployment, ed inoltre gli ho aggiunto anche una etichetta (Label) cosicché da facilitare il monitoraggio e la gestione del deployment in Kubernetes.
- All'interno delle specifiche del deployment il campo più interessante è:
  - Replicas:
    questo è il campo in cui specifico il numero di repliche desiderate del pod sottostante, per cui trattandosi del master, si mette logicamente questo valore pari ad 1.
- Infine segue la parte più corposa che descrive il pod, in cui sono indicati:
  - Volume:
    vale a dire il volume fisicamente a disposizione ed il claim utilizzato per accederci. Entrambi i nomi inseriti in questa sezione devono essere coerenti con i PersistentVolume e PersistentVolumeClaim precedentemente creati.
  - Le caratteristiche del container, cioè:
    - Name:
      questo campo è facoltativo ma per questioni di ordine e facilità di gestione è possibile dare un nome ad ogni container presente nel pod.

        * Image:
          indica l'immagine di partenza, vale a dire quella del master caricata sul repository di Docker Hub.
        * Port:
          è una lista di porte alle quale è consentita la comunicazione con quel pod.
        * VolumeMounts:
          indica dove montare i volumi disponibili cioè per ogni volume presente si indica a quale path montarlo all'interno del container.
        * due flag (stdin e tty) per permettere la modalità interattiva su quel pod.
    - ImagePullSecrets:
      questo elemento permette di agganciarsi ad un particolare tipo di secret, vale a dire quello che contiene le credenziali per l'accesso al docker hub (creato in precedenza); cosicché al momento del download dell'immagine Kubernetes sia in grado di autenticarsi ed ottenere i permessi per effettuare il pull dell'immagine desiderata.

NB:
In questo caso, trattandosi del master di JMeter, non è interessante sapere in quale datacenter verrà lanciato; per cui questa scelta viene delegata al kube-scheduler.

8.1.4 Deploy di JMeter slave su Softlayer

Deployment file:

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: jm-slave-softlayer
  labels:
    app: jm-slave-softlayer
spec:
  replicas: 1
  selector:
    matchLabels:
      app: jm-slave-softlayer
```

```
template:
  metadata:
    labels:
      app: jm-slave-softlayer
  spec:
    volumes:
    - name: jmeter-pv-storage
      persistentVolumeClaim:
        claimName: jmeter-pv-claim
    containers:
    - name: jm-slave-softlayer
      image: tesijmeterkubernetes/jmeter:slave
      ports:
      - containerPort: 1099
        name: port1
      - containerPort: 50000
        name: port2
      - containerPort: 2379
        name: etcd1
      - containerPort: 2380
        name: etcd2
      stdin: true
      tty: true
      volumeMounts:
      - mountPath: "/mnt/"
        name: jmeter-pv-storage
    nodeSelector:
      cloud-provider: Softlayer
    imagePullSecrets:
    - name: regsecret
```

Commenti:

Il file di deployment è alquanto simile a quello del master, vale a dire che anch'esso si aggancia allo stesso volume (appunto per ottenere la condivisione dei dati). Le uniche differenze presenti sono abbastanza ovvie e riguardano:

- L'immagine di partenza che ovviamente è quella di JMeter slave.
- Le porte usate da JMeter che in questo caso sono due (una che fa da server ed una che fa da client).
- Infine la differenza principale riguarda l'aggiunta del campo "nodeSelector" nel sezione di descrizione del Pod con al suo interno la label che necessariamente deve essere presente sul nodo worker in cui il pod verrà lanciato, vale a dire:
  - cloud-provider: Softlayer

  In questo modo si ha la possibilità di gestire quanti slaves lanciare su un determinato cloud-provider.

8.1.5 Deploy di JMeter slave su AWS

In questo caso il file di deployment è completamente identico a quello indicato nel punto precedente, le uniche differenze riguardano i nomi e le labels; in quanto:

- Il nome del Deployment e del container sarà:
  - jm-slave-aws
- Infine la label che deve essere presente sul nodo che lancerà il Pod deve essere:
  - cloud-provider: AWS

Capitolo 9

Conclusioni

9.1 Vantaggi

I vantaggi principali derivati dal lavoro di Tesi sono misurabili in termini di semplicità (vale a dire: numero di operazioni necessarie) per svolgere le seguenti funzioni:

- Deploy di Kubernetes in un ambiente multicloud.
- Deploy di JMeter.
- Lancio di stresstest con JMeter.
- Rimozione di JMeter.
- Rimozione dell'infrastruttura creata.

9.1.1 Terraform

Con l'uso di Terraform si è reso pressoché automatica la creazione di un cluster Kubernetes in multicloud.
Infatti il confronto tra il numero di operazioni richieste nell'usare Terraform piuttosto che non usarlo è eloquente:

- Con Terraform:
  È necessario un solo comando.
- Senza Terraform:
  Sono necessari i seguenti passaggi:

- Creazione manuale dei servers fisici o virtuali mediante l'utilizzo dell'interfaccia grafica messa a disposizione dai cloud providers.
- Installazione e configurazione di Kubernetes sui vari servers, anche questa operazione va effettuata manulmente.

Nella tabella che segue sono riportate le tempistiche necessarie a Terraform per eseguire le operazioni specificate. Purtroppo non è possibile effettuare un confronto realistico riguardante le tempistiche delle stesse operazioni ma svolte manualmente, perché quest'ultime dipendono dalla velocità dell'operatore, comunque si può dire che, grossolanamente, richiedono almeno due ore di lavoro.

Operazione	**Tempistiche**
Creazione server master (di Kubernetes) su Softlayer	h 0:03:28 ± 0:00:03
Installazione di Kubernetes e configurazione come master node	h 0:01:50 ± 0:00:01
Creazione node (di Kubernetes) su Softlayer	h 0:03:27 ± 0:00:03
Installazione di Kubernetes e configurazione come "worker" node	h 0:01:49 ± 0:00:01
Creazione node (di Kubernetes) su AWS	h 0:01:05 ± 0:00:01
Installazione di Kubernetes e configurazione come "worker" node	h 0:02:05 ± 0:00:04
Esecuzione dello script necessario per la riconfigurazione dei tunnel IP e l'etichettamento dei nodi	h 0:00:07 ± 0:00:01
Totale	**h 0:13:51 ± 0:00:14**

Commenti:
L'incertezza che affligge le tempistiche sono dovute principalmente a due motivi:

- Incertezza di misura, questa è presente in quanto l'unico modo per rilevare queste tempistiche è farlo manualmente per cui l'errore di misura può incidere in modo più o meno significativo.
- La seconda causa riguarda il contesto, ad esempio sono necessari i tempi di download per l'installazione di Kubernetes e ovviamente questi tempi non sono fissi, oppure a seconda dello stato del datacenter possono variare le tempistiche riguardanti la creazione dei servers.

In modo analogo si risparmia un buon numero di operazioni anche nel momento in cui si voglia eliminare l'infrastruttura creata, in quanto:

- Con Terraform:
  È necessario un solo comando.
- Senza Terraform:
  È necessario cancellare un server alla volta, e questa operazione va fatta manualmente avvalendosi della dashboard messa a disposizione dai vari cloud providers.

Successivamente vengono riportate le tempistiche riguardanti l'eliminazione dell'infrastruttura (anche in questo caso l'incertezza è dovuta dai medisimi motivi specificati nel punto precedente).

Operazione	**Tempistiche**
Eliminazione completa dell'infrastruttura	h 0:01:46 ± 0:00:03

9.1.2 Kubernetes

Con l'uso di Kubernetes si va semplificare il deploy di JMeter in multicloud, ed anche la variazione del numero di repliche degli slaves.
Infatti per quanto riguarda il deploy sono necessarie le seguenti operazioni:

- Con Kubernetes:
  - 5 comandi che creano rispettivamente:
    - Un PersistentVolume.
    - Un PersistentVolumeClaim.
    - Un master di JMeter.
    - Uno o più slaves di JMeter su Softlayer.
    - Uno o più slaves di JMeter su AWS.
  - Oppure un solo comando nel caso in cui si metta in un unico file le specifiche di tutte le risorse sopra elencate.
- Senza Kubernetes:
  È necessario installare il master di JMeter in locale o sul cloud (come meglio si desidera), dopodiché gli slaves vanno installati e configurati uno ad uno in servers fisici o virtuali dislocati in differenti cloud providers.

Per quanto riguarda la variazione del numero di repliche degli slaves il discorso è simile, cioè:

- Con Kubernetes:
  È necessario un solo comando.

- Senza Kubernetes:
  È necessario ripetere l'operazione di installazione e configurazione dello slave sui servers disponibili affinché si raggiunga il numero di repliche desiderato.

Anche in questo caso si sono analizzate le tempistiche necessarie per svolgere le operazioni appena descritte.

Operazione	**Tempistiche**
Creazione del container di JMeter master	h 0:00:37 ± 0:00:01
Creazione del container di JMeter slave su Softlayer	h 0:00:56 ± 0:00:01
Creazione del container di JMeter slave su AWS	h 0:00:45 ± 0:00:02
Crazione di un secondo slave su Softlayer	h 0:00:04 ± 0:00:01

Commenti:
Similmente a quanto avveniva per Terraform anche queste tempistiche non sono confrontabili in modo ragionevole alle analoghe svolte nel modo "tradizionale".
Detto ciò si può intuire che le misure sono affette dalle stesse cause di incertezza che hanno afflitto anche le misure precedenti.
Più curiosa invece è la differenza che riguarda il tempo di deploy dello slave su Softlayer con quello su AWS, questa differenza è dovuta dal fatto che nelle misure effettuate il master di JMeter è stato posizionato su nodo di Amazon, per cui dato che le immagini di JMeter master e slave hanno degli strati comuni per il deploy di JMeter slave su AWS è stato necessario scaricare soltanto gli strati mancanti, questo ha comportato una differenza di circa 10 sec rispetto al deploy della stessa immagine sul nodo di Softlayer.
Come ultima osservazione si può notare che il tempo di deploy di un secondo slave sul nodo di Softlayer (ma sarebbe potuto benissimo essere su AWS) è notevolmente più basso, questo perché quel nodo ha già disponibile su disco l'immagine dello slave, per cui deve soltanto eseguire l'operazione di deploy.

9.1.3 Docker

Con l'utilizzo di Docker si ha il vantaggio che si può personalizzare, per cui anche aggiungere delle funzionalità ad un prodotto.
Infatti a JMeter è stata aggiunta la feature che permette al master di venire a conoscenza automaticamente degli indirizzi IP dei vari slaves presenti sul cluster.
Per cui questa nuova funzionalità, ha semplificato il lancio dei tests su tutti gli slaves di JMeter disponibili sui vari cloud providers, in quanto:

- Con Docker (& Kubernetes):
  Basta aggiungere il parametro '-r' al comando 'jmeter', oppure basta cliccare l'apposita icona presente nell'interfaccia grafica di JMeter.
- Senza Docker (& Kubernetes):
  È necessario seguire i seguenti passi:
  - Ricavare gli indirizzi IP di tutti gli slaves disponibili.
  - Inserire questi indirizzi IP nel file jmeter.properties del master, alla riga in cui vengono specificati i vari remote hosts disponibili.
  - Infine lanciare il comando 'jmeter' con il parametro '-r', oppure cliccare sull'apposita icona presente nell'interfaccia grafica di JMeter.

9.2 Caso d'uso

Supponiamo di voler fare un test di carico su di un sito di e-commerce che ha la maggior parte dei clienti residenti negli USA.
Avendo già a disposizione il file .jmx contenente il testcase in questione, non resta altro che creare l'intera infrastruttura necessaria per il lancio del test.
Pertanto sono necessarie le seguenti fasi:

- Setup.
- Creazione del cluster Kubernetes.
- Lancio di JMeter sul cluster.

9.2.1 Setup

La fase di setup riguarda il compimento di queste tre operazioni:

- Creazione di un'utenza su Softlayer.
- Creazione di un'utenza Amazon.
- Installazione di Terraform in locale.

Per quanto riguarda la creazione delle utenze vanno date ad entrambe i permessi di creare delle risorse via API.
In seguito verranno descritti i passi per l'installazione di Terraform, nel caso in cui in locale si usi il sistema operativo Ubuntu:

- Download del binario di Terraform per Linux dal sito: https://www.terraform.io/downloads.html
- Decomprimere lo zip con il comando:
  - unzip terraform_0.10.2_linux_amd64
- Spostare l'eseguibile in una cartella presente nella variabile globale PATH, ad esempio con il comando:
  - mv terraform /usr/local/bin
- Download del plugin di Softlayer per Linux, necessario per la corretta interazione con Terraform, dal sito: https://github.com/softlayer/terraform-provider-softlayer/releases
- Rinominare il binario appena scaricato con il nome terraform-provider-softlayer, digitando il seguente comando:
  - mv terraform-provider-softlayer_linux_amd64 terraform-provider-softlayer
- Aggiunta dei diritti di esecuzione a tale binario:
  - chmod +x terraform-provider-softlayer
- Spostare il plugin in una cartella presente nella variabile globale PATH:
  - mv terraform-provider-softlayer /usr/local/bin
- Creare il file ~/.terraformrc con al suo interno il seguente contenuto:
  providers {
  softlayer = "/usr/local/bin/terraform-provider-softlayer"
  }

A questo punto Terraform è pronto per essere utilizzato.
In conclusione del setup è necessario scaricare il lavoro di Tesi disponibile su Github, per cui digitare il seguente comando:

- git clone https://github.com/LucaVacchetta/Tesi.git

9.2.2 Creazione del cluster di Kubernetes

Per creare tre macchine con sopra installato 'Kubernetes' in multicloud, basta seguire i seguenti passi:

- Spostarsi all'interno della cartella Kubernetes presente nel progetto git scaricato in precedenza.
- Inizializzare i plugin di Terraform mancanti, cioè in questo caso verranno eseguite queste due operazioni:
  - Scaricato e configurato il plugin di AWS.
  - Configurato il plugin di Softlayer precedentemente scaricato.

  Per far ciò basta digitare il seguente comando:
  - terraform init
- Inserire all'interno del file 'kubernetes.tf' le credenziali per l'accesso tramite API a Softlayer e AWS.
- Creare una chiave ssh, necessaria per il trasferimento sicuro dei files di setup ai cloud providers, con il seguente comando:
  - ssh-keygen -t rsa
- Ricavare dalla chiave ssh appena creata l'analoga in formato PEM, in modo che sia compatibile con i requisiti richiesti da AWS[1], per far ciò è necessario digitare i seguenti comandi:
  - openssl rsa -in /.ssh/id_rsa -outform pem >id_rsa.pem
  - chmod 700 id_rsa.pem
- Con il fine di avere una preview di quante risorse verranno create, digitare il seguente comando:

[1]La chiave privata in AWS deve avere il formato PEM e deve essere in sola lettura

  - terraform plan

- Creare effettivamente le macchine digitando il comando:

  - terraform apply

Al termine dell'operazione apparirà una schermata simile a quella mostrata in figura 9.1.

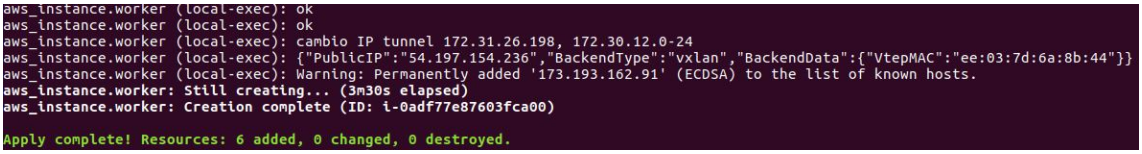

Figura 9.1. Screenshot che indica il successo riguardante la creazione delle risorse

9.2.3 Deploy di JMeter su Kubernetes

A questo punto non resta che effettuare il deploy di JMeter (master e slaves) sul cluster di Kubernetes, quindi è necessario seguire i passi sotto indicati:

- Accedere alla console in cui gira il nodo master di Kubernetes, con il comando:

  - ssh root@<IP address of master node>

- Scaricare nuovamente il progetto git contenente il lavoro di Tesi, digitando il comando:

  - git clone https://github.com/LucaVacchetta/Tesi.git

- Spostarsi nella cartella 'Deploy JMeter in multicloud' presente nel progetto git.

- Creare il volume lanciando i seguenti comandi (dalla cartella che contiene i files deployment del Volume):

  - kubectl create -f persistentVolume.yaml
  - kubectl create -f persistentVolumeClaim.yaml

- Creare il master lanciando il comando (dalla cartella in cui è presente il file deployment del master):

  - kubectl create -f deployment.yaml

NB:
Dato che si sta utilizzando un volume di tipo hostPath (solo perché non si ha a disposizione un NFS) è **NECESSARIO** copiare la cartella /mnt/jmeter-volume presente sul nodo in cui è stato creato il master su tutti i nodi "worker" del cluster affinché gli slave presenti sui nodi diversi da quello del master possano accedere al file comune.

- Creare lo slave di Softlayer lanciando il comando (dalla cartella che contiene il file deployment dello slave di "etichettato" Softlayer):
  - kubectl create -f deployment.yaml
- Creare lo slave di AWS lanciando il comando (dalla cartella che contiene il file deployment dello slave di "etichettato" AWS):
  - kubectl create -f deployment.yaml
- In seguito se si desidera aumentare il numero di slave presenti, è necessario lanciare il comando:
  - kubectl scale deployment jm-slave-softlayer —replicas=3
    che in pratica impone a Kubernetes di avere un numero di repliche di quel Pod pari al numero specificato nel parametro " *–replicas*"
- A questo punto per poter utilizzare il proprio testcase all'interno di questa infrastruttura, è necessario copiare il testcase in questione all'interno della cartella /mnt/jmeter-volume presente sul nodo in cui gira il pod del master di JMeter.
- Dopodichè è necessario entrare nella console del master di JMeter, e lo si fa con questo comando:
  - kubectl attach <nome del pod di JMeter master[2] >-i -t
- Infine per lanciare il test desiderato su tutti gli slaves presenti nel cluster è indispensabile lanciare il comando jmeter con il seguente parametro: -r vale a dire il comando sarà nella forma:
  - jmeter -n -r -t /mnt/testcase.jmx

[2]Per venire a conoscenza del nome del pod di JMeter master è necessario digitare il seguente comando: kubectl get pod

- Al termine dell'esecuzione, il test di carico produrrà un file CSV contenente il risultato del testcase, per cui sarà possibile scaricare tale file ed analizzarlo in separata sede.

Infine se non si ha più bisogno dell'intera infrastruttura è possibile eliminarla onde evitare uno spreco di risorse. Per far ciò è necessario semplicemente seguire i seguenti step:

- Tornare nella console locale, cioè sul pc con cui si sono create le macchine con Terraform.
- Entrare da terminale nella cartella 'Kubernetes' presente nel progetto github.
- Dopodiché con il fine di verificare quali risorse verranno distrutte, digitare il seguente comando:
  - terraform plan -destroy
- Per ultimo eliminare tali risorse, digitando:
  - terraform destroy

Bibliografia

[1] Il tool Docker e relativa documentazione, https://www.docker.com/

[2] Il tool Kubernetes e relativa documentazione, https://kubernetes.io/

[3] Documentazione per stesura files di configurazione in Kubernetes, https://kubernetes.io/docs/resources-reference/v1.5/

[4] Il progetto Heapster, https://github.com/kubernetes/heapster

[5] Il progetto Flannel, https://github.com/coreos/flannel

[6] Il tool terraform e relativa documentazione, https://www.terraform.io/

[7] Estensione di Terraform per Softlayer, https://github.com/softlayer/terraform-provider-softlayer/tree/master/docs/resources

[8] Il tool JMeter e relativa documentazione, http://jmeter.apache.org/

[9] Documentazione relativa ai segnali in Unix, https://www.tutorialspoint.com/unix/unix-signals-traps.htm