

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA
CAMPUS DI CESENA

DIPARTIMENTO DI INFORMATICA – SCIENZA E INGEGNERIA
Corso di Laurea in Ingegneria e Scienze Informatiche

**CONTROLLO E SCALABILITÀ
AUTOMATIZZATI IN CLUSTER
KUBERNETES**

Elaborato in:
Systems Integration

Relatore:
Prof. Ghini Vittorio

Presentata da:
Baiardi Martina

Sessione II
Anno Accademico 2019-2020

Introduzione

Le moderne applicazioni che vogliono adattarsi al meglio nel distribuito sono costituite da container.

Kubernetes (K8s) è un software che fornisce orchestrazione di container lavorando su cluster di macchine che interagiscono tra loro. Sviluppato inizialmente da Google e poi reso open-source nel 2014 proprio per aiutare gli sviluppatori a far funzionare correttamente questa tipologia di sistemi complessi: K8s permette di dispiegare applicazioni containerizzate, facilitandone la configurazione, e di gestire dinamicamente carichi di lavoro variabili nel tempo.

L'obiettivo di questa tesi è perciò quello di fornire agli studenti del Corso di Ingegneria e Scienze Informatiche uno strumento per fare pratica con queste infrastrutture software direttamente sulle proprie macchine locali, virtualizzando completamente tutta l'infrastruttura su un singolo host mediante più macchine virtuali.

La tesi è strutturata come segue:

- Nel primo capitolo sono introdotti i principali software che stanno alla base della ricerca e i motivi per i quali sono interessanti in uno scenario odierno.
- Nel secondo capitolo sono descritti dettagliatamente gli strumenti utilizzati maggiormente all'interno del documento, nonchè:

- Kubernetes, software che permette di dispiegare applicazioni containerizzate su cluster di macchine connessi tra loro.
 - Docker, software che consente di creare e gestire container di applicazioni, necessario per permettere a Kubernetes di gestire i container.
 - Kubeadm, software utilizzato per dispiegare Kubernetes sulle macchine virtuali.
 - Helm, software che consente di installare e condividere applicazioni su Kubernetes.
- Nel terzo capitolo vengono descritti gli obiettivi che si vogliono raggiungere all'interno della tesi e viene suddiviso concettualmente il suo sviluppo.
 - Nel quarto capitolo viene effettuata un'installazione guidata delle macchine virtuali per generare un Cluster Kubernetes composto da un Master Node e due Worker Nodes, per poi procedere con una spiegazione dettagliata su come poter dispiegare un'applicazione al suo interno.
 - Nel quinto capitolo, per espandere le conoscenze sui servizi offerti da Kubernetes, vengono studiate le possibilità di scaling (aumentare o diminuire automaticamente le risorse) del cluster.

Viene approfondito il componente di Kubernetes chiamato Horizontal Pod Autoscaler, che si occupa di scalare il numero di repliche di una certa applicazione, scoprendo che funziona grazie all'interrogazione di metriche che tengono traccia di valori relativi all'applicazione.

Per sperimentare metriche differenti, sono stati approfonditi due strumenti:

1. Metrics Server, API Server che si occupa di collezionare le metriche da un componente chiamato cAdvisor, installato di default all'interno del cluster, che permette di osservare valori relativi a memoria e CPU utilizzati.

2. Prometheus, software open-source che permette di raccogliere metriche di molteplici tipi dal cluster. Prometheus funziona grazie ai suoi Exporters che si occupano di osservare oggetti all'interno del cluster ed estrarne metriche.

Questi vengono poi testati entrambi all'interno del cluster per poter apprezzare il funzionamento dell'Horizontal Pod Autoscaler.

- Nel sesto capitolo viene valutata la principale problematica del dispiegamento di un cluster in locale: l'accesso dall'esterno.

Per poter esporre verso l'esterno un'applicazione tipicamente si utilizza un Ingress. Questo, però, richiede la presenza di un Load Balancer, ossia un agente che osservi il traffico in ingresso al cluster e decida a quale dei nodi contenenti le repliche del servizio richiesto inoltrare la richiesta.

È stato quindi introdotto MetalLB, progetto open-source pensato proprio per risolvere questa problematica installandolo all'interno del cluster, permettendo così da poter integrare una risorsa ingress all'interno del nostro cluster.

Indice

Introduzione	i
1 Scenario	1
2 Strumenti utilizzati	3
2.1 Kubernetes	3
2.1.1 API Object	3
2.1.2 Architettura	4
2.1.3 Addons	7
2.2 Docker	8
2.3 Kubectl	10
2.3.1 Requisiti di installazione	10
2.3.2 Kubectl create	12
2.3.3 Kubectl apply	15
2.4 Helm	17
3 Obiettivi della tesi	19
3.1 Obiettivi	19
4 Creazione del Cluster	21
4.1 Creazione del Master Node	21
4.2 Creazione dei Worker Nodes	25
4.3 Primo Deployment su Cluster	31

5	Scaling del Cluster	39
5.1	Horizontal Pod Autoscaler	40
5.1.1	Le unità di misura di Kubernetes	48
5.2	Vertical Pod Autoscaler	48
5.3	Cluster Autoscaler	49
5.4	Metrics Server	49
5.4.1	Integrazione dell'Horizontal Pod Autoscaler con il Metrics Server	53
5.5	Prometheus	59
5.5.1	Prometheus Operator	61
5.5.2	Prometheus Adapter	62
5.5.3	Kube Prometheus	63
5.5.4	Integrazione Kube Prometheus Stack all'interno del Cluster	64
6	Accesso al Cluster	83
6.1	Ingress	83
6.1.1	Ingress Class	85
6.1.2	Ingress Controller	86
6.2	Integrazione di un Ingress all'interno del Cluster	88
6.2.1	Installazione MetalLB	89
6.2.2	Installazione ingress-nginx	94
6.2.3	Accesso al Cluster dall'esterno	94
	Conclusioni e Progetti Futuri	97
	Bibliografia	99

Capitolo 1

Scenario

Al giorno d'oggi dispiegare applicazioni su internet richiede di valutare molte incognite, tra cui la disponibilità sono rilevanti la disponibilità e la distribuzione del servizio; questo permette l'accesso, da parte di un utente, da qualsiasi parte del mondo e con buona reattività. Inoltre, sta prendendo popolarità preferire il dispiegamento delle applicazioni tramite container piuttosto che l'esecuzione diretta su macchine fisiche e/o virtuali, in modo che i server siano in grado di eseguire, in ambienti isolati, più applicazioni contemporaneamente (anche replicate) per aumentare la loro capacità di servire gli utenti.

I container rappresentano una soluzione di virtualizzazione più leggera rispetto alle macchine virtuali, in quanto condividono tutti lo stesso sistema operativo. Essi dispongono di una porzione riservata di File System, CPU, memoria, Process Identifier (PID) e altro ancora e, siccome sono disaccoppiati dall'infrastruttura sottostante, si prestano ad essere portabili tra diverse distribuzioni e differenti cloud.

Kubernetes (o **K8s**) è una piattaforma sviluppata inizialmente da Google e poi resa open-source nel 2014 proprio per aiutare gli sviluppatori a far funzionare correttamente questa tipologia di sistemi: permette di gestire

carichi di lavoro, applicazioni containerizzate ed in grado di facilitare sia la configurazione che l'automazione.

K8s raggruppa i container che compongono gli applicativi in unità logiche, in modo da semplificare la gestione e la visibilità dei servizi. Per questo motivo è necessario che sia in esecuzione in un ambiente in cui è presente un software che gestisce container con cui comunicare, come ad esempio **Docker**.

Docker è un progetto open-source che consente di dispiegare applicazioni containerizzate sopra un sistema operativo Linux. Esso sfrutta le funzionalità del kernel e l'isolamento delle risorse per definire ciò che l'applicazione, situata all'interno del container, può vedere del sistema operativo.

Ci piacerebbe, con questo lavoro, mettere a disposizione degli studenti del corso di Ingegneria e Scienze Informatiche gli strumenti necessari per poter implementare Kubernetes all'interno delle proprie macchine. In particolare, vorremmo fornire le operazioni per riprodurre un cluster, anche di piccole dimensioni, su un insieme di macchine virtuali sui computer personali; in questo modo, gli studenti, potranno testarne e apprezzarne le qualità.

Capitolo 2

Strumenti utilizzati

Sono molti e differenti gli strumenti utilizzati all'interno di questa ricerca, alcuni direttamente introdotti dagli stessi sviluppatori del progetto principale e altri aggiunti da utenti che non fanno parte del team Kubernetes.

2.1 Kubernetes

Kubernetes è un software sviluppato per poter dispiegare applicazioni containerizzate in cluster di server, in modo da garantire scalabilità e maggiore fruizione delle risorse da parte delle applicazioni.

2.1.1 API Object

Gli oggetti di Kubernetes[1] sono concetti di istanze contenute all'interno di un cluster K8s e vengono utilizzati per implementarne le features. Sono presenti tante tipologie di oggetti, ognuna specifica per l'obiettivo che si vuole raggiungere all'interno del cluster.

Alcuni degli oggetti principali su cui si basa un cluster kubernetes sono:

1. **Pod**. Strutture che contengono uno o più container che eseguono specifiche applicazioni. Questi container lavorano come se si trovassero su uno stesso host virtuale e condividono:

- (a) Memoria (o Volume)
- (b) Indirizzo IP
- (c) Porte di interfaccia (API)
- (d) PID
- (e) IPC (Inter-Process Communications), meccanismi di comunicazione tra processi differenti.

2. **Deployment.** Raggruppamenti logici per le repliche di uno stesso Pod.
3. **Services.** Metodo di accesso ai pods che include una politica di bilanciamento di carico nel caso in cui si presentino più repliche di uno stesso pod.

2.1.2 Architettura

Cluster

Quando viene dipiegato Kubernetes si ottiene un Cluster, insieme di macchine che eseguono al loro interno container di applicazioni.

Nodes

I Nodes non sono altro che le macchine che compongono il cluster Kubernetes, le quali si occupano di contenere i pods. Esistono due tipologie di nodi in un cluster:

1. **Master Node (o Control-Plane).** Contiene le componenti che consentono di gestire il cluster, ad esempio permettendo di: accettare richieste dai client, effettuare scheduling dei container ed effettuare cicli di controllo per mantenere il cluster funzionante. Ciascun master node contiene al suo interno:
 - (a) **API Server.** Fornisce API di tipo REST che consentono di effettuare operazioni di Create, Read, Update and Delete (CRUD)

sugli oggetti del cluster. L'API Server è un servizio stateless che si occupa di controllare lo stato del cluster stesso e per farlo si avvale del componente ETCD. L'API Server contiene a sua volta un proxy che permette di collegare un utente dall'esterno del cluster con servizi altrimenti non raggiungibili (effettuando inoltre bilanciamento di carico).

- (b) **ETCD.** Database open-source distribuito di tipo chiave-valore che contiene tutte le informazioni sul cluster. Può essere configurato come componente del Master Node oppure come componente esterno, in ogni caso solamente l'API Server può interagire con esso.
- (c) **Scheduler.** Organizza i pods all'interno dei nodi. Controlla quale nodo è in grado di gestire al meglio il pod che si vuole dispiegare (in base alle risorse che ha ancora disponibili) e glielo assegna. Le informazioni sulle risorse utilizzate da ciascun Worker Node del cluster sono contenute all'interno del database ETCD.
- (d) **Control Manager.** Si occupa di gestire i container, si suddivide in:
 - i. *Kube-Controller Manager*, che si occupa di controllare l'esecuzione dei Worker Nodes e informa il Master nel caso in cui ce ne sia uno che abbia fallito o non sia più disponibile. Si suddivide a sua volta in:
 - A. Node Controller, si preoccupa di controllare i nodi.
 - B. Replication Controller, si preoccupa di controllare il numero di repliche dei pods.
 - C. Endpoint Controller, si occupa dell'unione tra services e pods (costituendo oggetti Kubernetes chiamati Endpoint).
 - D. Service Account and Token Controllers, crea gli account di default e i token per accedere alle API.

- ii. *Cloud-Controller Manager*, che si occupa di monitorare l'interazione con una eventuale infrastruttura cloud sottostante, quindi gestisce Volumi Cloud e Load Balancing. Si compone di:
 - A. Node Controller, controlla il cloud provider per stabilire se un nodo che ha smesso di rispondere viene eliminato.
 - B. Route Controller, crea un instradamento all'interno dell'infrastruttura cloud.
 - C. Volume Controller, crea e collega i volumi dati all'interno del cloud provider.
 - D. Service Controller, crea, aggiorna ed elimina i Load Balancer.

2. **Worker Node.** Si occupa di contenere pods ed eseguire container. Ciascun Worker Node esegue:

- (a) **Kubelet.** Agente che si occupa di comunicare con il Master Node attraverso l'API Server e assicura che i pods all'interno del nodo stiano lavorando correttamente. Kubelet ragiona con una lista di informazioni chiamata PODSPEC, la quale non è altro che file in formato YAML o JSON che contiene una descrizione dettagliata dei pod. Nel caso in cui un pod non funzioni correttamente, Kubelet cercherà di riavviarlo (eventualmente anche su un altro nodo).
- (b) **Kube-Proxy.** Componente che si occupa di effettuare proxy delle comunicazioni UDP, TCP e SCTP (non gestisce HTTP) ed effettua bilanciamento di carico. Il kube proxy è utilizzato per raggiungere i servizi situati all'interno di un nodo.
- (c) **Container Runtime.** Software che gestisce l'esecuzione di container, come ad esempio Docker.

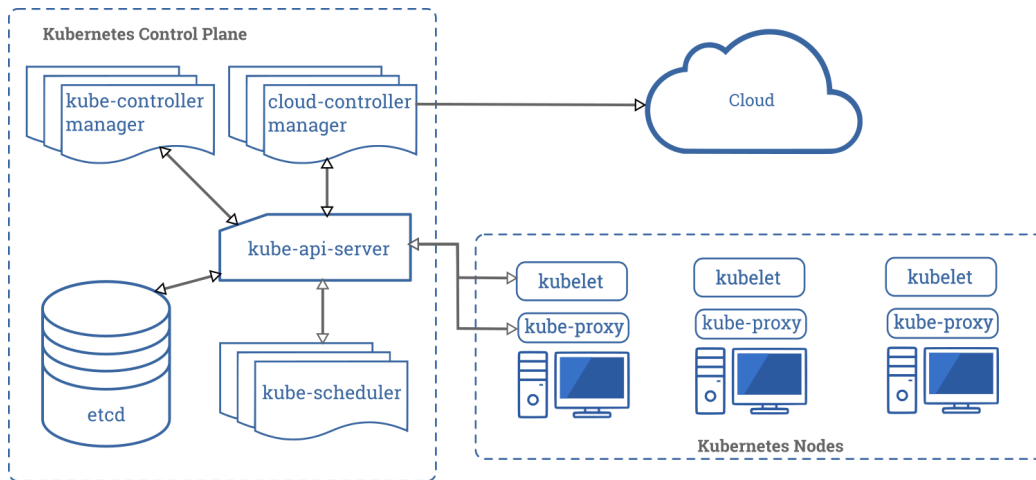


Figura 2.1: Rappresentazione grafica di un Cluster Kubernetes e dei suoi componenti.

Non esiste una regola che dica che possa esistere un solo nodo Master all'interno di un cluster: Kubernetes fornisce un'architettura completamente personalizzabile.

È anche possibile creare un cluster dove sia i Master che i Worker Nodes possano fornire servizi dispiegati; tipicamente questo si può fare effettuando l'operazione di taint.

Nota: Di default, tramite dispiegamento di un cluster con applicativo Kubeadm, il Master Node non può esporre servizi. In un'applicazione reale non sempre è vantaggioso farlo, in quanto aumenta il rischio di introdurre delle latenze se il server è già occupato a gestire delle richieste dei client. Inoltre, questo è un possibile rischio per la sicurezza del cluster siccome esso contiene tutte le informazioni per mantenere in funzione i Worker e i Deployment al suo interno.

2.1.3 Addons

Le Addons estendono le funzionalità di Kubernetes:

1. **Networking and Network Policy.** Plugin necessari per adattare il cluster alla configurazione di rete dei vari nodi permettendo, tra le altre cose, di far comunicare i pod tra loro. La documentazione offre una lista[2] relativa ai network plugins ufficialmente supportati.
2. **Service Discovery.** È necessario integrare un DNS Server all'interno del cluster poichè questo permette di ottenerne gli indirizzi che sono stati assegnati ad un determinato pod/servizio e alle sue repliche. CoreDNS è il servizio che viene installato di default da Kubeadm.
3. **Dashboard.** Add-on che permettere agli utenti di fare troubleshooting del cluster (e delle applicazioni al suo interno) tramite interfaccia web.
4. **Monitoring.** Add-on che permette di raccogliere serie temporali di metriche sugli oggetti del cluster e fornisce un'interfaccia che permette di navigare tra i dati. Le metriche possono essere raccolte da servizi differenti, tra cui il Metrics Server e Prometheus.

2.2 Docker

Poichè Kubernetes lavora a stretto contatto con i container prima di procedere con l'installazione di Kubeadm è necessario installare Docker all'interno della macchina.

Docker, in realtà, rappresenta solamente la scelta di default di una lista di Container Runtime Interface (CRI) che è possibile utilizzare, elencate nella documentazione ufficiale; le principali sono: Docker, **Containerd** e **CRI-O**.

Questi software vengono rilevati in automatico durante l'installazione da parte di Kubeadm e in base a quello presente sulla macchina vengono aggiunte le dipendenze necessarie per la collaborazione con Kubernetes.

Nota: se viene installato più di un CRI sulla macchina, allora è necessario esplicitare quale si vuole che venga utilizzato, perchè altrimenti kubeadm terminerà con errore.

Nella documentazione Kubernetes[3] viene spiegato come installare la CRI scelta. Di seguito viene riportato come installare Docker all'interno della macchina.

```
# Set up the repository:
# Install packages to allow apt to use a repository over HTTPS
apt-get update && apt-get install -y apt-transport-https
↪ ca-certificates curl software-properties-common gnupg2

# Add Docker's official GPG key:
curl -fsSL https://download.docker.com/linux/ubuntu/gpg |
↪ apt-key add -

# Add the Docker apt repository:
add-apt-repository "deb [arch=amd64]
↪ https://download.docker.com/linux/ubuntu $(lsb_release -cs)
↪ stable"

# Install Docker CE
apt-get update && apt-get install -y containerd.io=1.2.13-2

docker-ce=5:19.03.11~3-0~ubuntu-$(lsb_release -cs)
↪ docker-ce-cli=5:19.03.11~3-0~ubuntu-$(lsb_release -cs)

# Set up the Docker daemon
cat > /etc/docker/daemon.json <<EOF
{
  "exec-opts": ["native.cgroupdriver=systemd"],
  "log-driver": "json-file",
  "log-opts": {
    "max-size": "100m"
  }
}
```



```
    },  
    "storage-driver": "overlay2"  
  }  
EOF
```

```
mkdir -p /etc/systemd/system/docker.service.d
```

```
# Restart Docker  
systemctl daemon-reload  
systemctl restart docker  
sudo systemctl enable docker
```

2.3 Kubeadm

Per costruire il cluster, cioè l'insieme di macchine che compongono il sistema K8s, si è utilizzato lo strumento Kubeadm. Questo software fornisce i comandi necessari per implementare un cluster funzionante dalle funzionalità minime: per come è stato progettato effettua solamente una configurazione delle macchine e non il loro approvvigionamento, compito dell'utente che lo vuole costruire.

2.3.1 Requisiti di installazione

Kubeadm necessita però di requisiti per installare Kubernetes sulle macchine, ben descritti sulla guida ufficiale [4]. Vengono riportati i principali:

1. **Sistema Operativo.** Può essere installato solamente su ambiente Linux
2. **RAM.** Il sistema necessita di almeno 2GB di memoria per macchina.
3. **CPU.** Richiede che le macchine abbiano a loro disposizione almeno due CPU.

4. **Rete.** Le macchine che compongono il cluster devono essere tutte nella stessa rete e devono poter comunicare tra loro. È inoltre necessario che i nodi possano vedere e trattare come traffico in uscita i pacchetti scambiati all'interno della rete di tipo Bridged. Per farlo configuriamo le iptables:

```
# Open /etc/ufw/sysctl.conf file
sudo nano /etc/ufw/sysctl.conf

# Add the following lines on the bottom of the file and
→ save
net/bridge/bridge-nf-call-ip6tables = 1
net/bridge/bridge-nf-call-iptables = 1
net/bridge/bridge-nf-call-arptables = 1

# reboot the machine
sudo reboot
```

Infine, sono necessarie le seguenti dipendenze per far funzionare kubeadm:

```
apt-get install ebtables ethtool
```

5. **Nome macchina.** Ogni macchina del cluster su cui viene utilizzato kubeadm deve avere nome univoco.
6. **MAC Address.** Ogni macchina del cluster deve avere indirizzo MAC dell'interfaccia di rete univoco. È possibile visualizzare il MAC Address dell'interfaccia di rete digitando il comando:

```
ifconfig
```

7. **Product UUID.** Per ogni macchina del cluster l'Universally Unique Identifier (UUID) deve essere univoco. È possibile consultarne il valore con il comando:

```
sudo cat /sys/class/dmi/id/product_uuid
```

8. **Swap della memoria disabilitato.** Non è consentita la possibilità di utilizzare memoria virtuale su disco. È possibile disabilitare lo swap della memoria digitando il comando:

```
swapoff -a
```

Ma, per mantenere questa impostazione anche al reboot della macchina, è necessario effettuare le seguenti operazioni:

```
# Open /etc/fstab file  
# Then comment the last line of this file and save  
sudo nano /vi/fstab
```

Kubeadm fornisce due comandi essenziali che permettono di implementare il cluster: *kubeadm init* e *kubeadm join*.

2.3.2 Kubeadm init

Kubeadm `init`[5] è il comando che viene eseguito per creare il primo componente del cluster: il nodo Master. Nel dettaglio, le operazioni che si susseguono sono le seguenti:

1. **Pre-flight check:** fase in cui viene controllato che il sistema su cui si vuole installare il control-plane di kubernetes possieda tutti i requisiti necessari.

2. **Self-Signed CA:** viene generato un certificato che permette di riconoscere le identità delle componenti del cluster. È possibile metterne uno personalizzato attraverso i flag del comando `init`.
3. **Kubeconfig files:** all'interno della directory `/etc/kubernetes/` vengono salvati tutti i file di configurazione necessari al controller-manager e allo scheduler per connettersi all'API Server. Viene inoltre generato un file per l'amministrazione chiamato `admin.conf`.
4. **Static Pod Manifest:** vengono settati i manifest dei pod che il kubelet deve istanziare quando viene avviato; questi file sono salvati nella directory `/etc/kubernetes/manifests`.
5. **Taint-Node:** viene applicata la label sul nodo contenete il control plane in modo che di default esso non contenga carico aggiuntivo.
6. **Token:** viene generato il token necessario ai nodi per registrarsi presso il control-plane (operazione di `join`).
7. **Join Configuration:** vengono effettuate tutte le operazioni necessarie per permettere ad altri nodi di far parte del cluster (Bootstrap Tokens and TLS Bootstrap mechanism).
8. **DNS e kube-proxy:** vengono installati il DNS server (`coreDNS`) e il `kube-proxy`.

È possibile personalizzare l'installazione del nodo master con i seguenti flag:

- **--api-advertise-address:** permette di specificare con quale indirizzo l'API Server comunica con le altre componenti del cluster, il quale è lo stesso utilizzato all'interno del comando `kubeadm join`. Se non settato (oppure settato a `0.0.0.0`) esso prende come IP quello dell'interfaccia di default.
- **--apiserver-bind-port:** porta su cui comunica l'API Server, di default la `6443`.

- **--apiserver-cert-extra-sans**: hostname aggiuntivi o indirizzi IP che vengono considerati come Subject Alternate Name all'interno del certificato dell'API Server.

```
--apiserver-cert-extra-sans= kubernetes.example.com,  
↪ kube.example.com, 10.100.245.1
```

- **--cert-dir**: directory su cui vengono salvati i certificati, la default è `/etc/kubernetes/pki`
- **--certificate-key**: chiave utilizzata per criptare i certificati all'interno dei Secrets.
- **--config**: file di configurazione di kubeadm che viene specificato per personalizzare configurazioni aggiuntive ai flag.
- **--control-plane-endpoint**: permette di specificare un IP oppure un nome DNS per accedere al control plane.
- **--node-name**: specifica il nome del nodo.
- **--kubernetes-version**: versione di kubernetes utilizzata all'interno del cluster. Di default viene presa l'ultima disponibile.
- **--pod-network-cidr**: è possibile specificare una sottorete che permetta di assegnare indirizzi IP differenti ai pods all'interno dei nodi rispetto alla rete dell'API Server.
- **--service-cidr**: permette di personalizzare la sottorete che viene utilizzata per i servizi, di default settata a `10.96.0.0/12`. Attenzione, se si sceglie di personalizzare questa configurazione allora è necessario aggiornare il file `/etc/systemd/system/kubelet.service.d/10-kubeadm.conf` in modo che sia coerente con la modifica, altrimenti il dns non funzionerà correttamente.

- **--service-dns-domain**: dominio utilizzato per risolvere i servizi, di default settato a cluster.local. Anche in questo caso è necessario aggiornare il file `/etc/systemd/system/kubelet.service.d/10-kubeadm.conf` oppure il DNS non funzionerà correttamente.
- **--skip-preflight-checks**: utilizzato per saltare i passaggi in cui kubeadm controlla che il sistema in cui lo si sta installando abbia tutti i requisiti necessari.
- **--token**: di default viene generato in automatico un token con il comando `kubeadm init` per poter associare i nodi al cluster, ma è possibile personalizzarlo con questo flag.
- **--token-ttl**: specifica un tempo di validità per il token. se `ttl` viene settato a zero, come avviene di default, allora il token sarà sempre valido.

2.3.3 Kubeadm join

Kubeadm join[6] è l'operazione che viene utilizzata per inserire un Worker Node in un cluster inizializzato da kubeadm; per farlo bisogna stabilire un rapporto di fiducia bidirezionale. Le fasi del comando sono:

1. **Discovery**. Questa fase si divide in due metodi principali:
 - (a) Il primo è utilizzare un token condiviso insieme all'indirizzo IP dell'API Server. La forma utilizzata è:

```
kubeadm join --discovery-token abcdef.1234567890abcdef  
↪ 1.2.3.4:6443
```

 - (b) Il secondo è fornire un file (un sottoinsieme del file `kubeconfig` standard) locale o scaricato tramite un URL HTTPS. La forma utilizzata è:

```
kubeadm join --discovery-file path/to/file.conf
```

oppure

```
kubeadm join --discovery-file https://url/file.conf
```

2. **Bootstrap TLS.** Gestito anch'esso tramite token condiviso, viene utilizzato per autenticarsi temporaneamente presso il Master Kubernetes e scambiarsi una coppia di chiavi. Per configurazione predefinita il Master Kubernetes approva automaticamente queste richieste in arrivo.

Opzioni specifiche:

- **--config:** Opzioni estese specificate in un file di configurazione specifico di kubeadm.
- **--skip-preflight-checks:** Per impostazione predefinita, kubeadm esegue una serie di controlli preliminari per convalidare il sistema prima di apportare qualsiasi modifica. Gli utenti avanzati possono utilizzare questo flag per ignorarli, se necessario.
- **--discovery-file:** Un percorso file locale o URL HTTPS. Il file specificato viene utilizzato per trovare sia la posizione del server API a cui unirsi sia un root CA Bundle da utilizzare quando si parla con quel server.
- **--discovery-token:** Il token di rilevamento viene utilizzato insieme all'indirizzo del server API per scaricare e verificare le informazioni sul cluster. La parte più critica delle informazioni sul cluster è il root CA Bundle, utilizzato per verificare l'identità del server durante le successive connessioni TLS.
- **--tls-bootstrap-token:** Utilizzato per l'autenticazione al server API ai fini del bootstrap TLS.

- **--token**: Spesso lo stesso token viene utilizzato per entrambi `--discovery-token` e `--tls-bootstrap-token`. Questa opzione specifica lo stesso token per entrambi. Altri flag sovrascrivono questo flag se presente.

2.4 Helm

HELM è un'applicazione che aiuta nel dispiegamento di applicazioni su Kubernetes attraverso gli Helm Charts. Si può considerare come un Docker Hub ma per applicazioni Kubernetes al posto di immagini di Containers.

È possibile installarlo[7] dal sito ufficiale per poi procedere alla sua configurazione [8] prima di utilizzarlo.

Nel lavoro è stato utilizzato per installare la maggior parte delle componenti aggiuntive del cluster.

Una volta installata un'applicazione con helm, attraverso il comando:

```
helm install <nome> nome_applicazione
```

è possibile disinstallarla con:

```
helm delete <nome>
```

Allo stesso modo è possibile personalizzare l'installazione di un componente, per esempio specificando il namespace in cui deve essere posizionato all'interno del cluster con il flag `-n` (sono presenti tutti i flag sul sito ufficiale del software). È possibile tenere traccia degli elementi installati con helm attraverso il comando:

```
helm list -n <namespace>
```

mentre per visualizzarne le informazioni basta usare

```
helm status <nome> -n <namespace>.
```

e per aggiornare le dipendenze:

```
helm repo update
```

Capitolo 3

Obiettivi della tesi

In questo capitolo vengono descritti gli obiettivi che si vogliono raggiungere all'interno della tesi, proponendo una suddivisione logica del lavoro che viene poi svolto.

3.1 Obiettivi

L'obiettivo è quello di dispiegare l'architettura di Kubernetes su macchine virtuali locali in modo da permettere agli studenti di Ingegneria e Scienze Informatiche di testare sul loro pc tutte le funzionalità di un cluster, senza la necessità di appoggiarsi ad un cloud provider.

Sostanzialmente, il lavoro si divide in due sezioni principali:

1. Fase di **dispiegamento del cluster**: Si installa il cluster sulle macchine virtuali e si analizzano i comandi che è possibile utilizzare per interagire con Kubernetes. Questa parte comprende anche della sperimentazione per approfondire come vengono distribuiti gli oggetti all'interno del cluster.
2. Fase di **scaling del cluster**: Una volta ottenuta un'architettura funzionante, si analizzano quali sono le possibilità di automazione nel caso in cui aumenti/diminuisca il carico e l'utente non riesca a rendersene

conto e agire in fretta. Kubernetes fornisce degli strumenti built-in di scaling per questa particolare funzionalità e lo scopo della ricerca è quello di integrarli all'interno del cluster, valutandone pregi e difetti.

È possibile suddividere questa fase di scaling del cluster in:

- (a) Incremento e decremento del numero di repliche dei pods interni ad un cluster sulla base della valutazione di metriche.
 - (b) Incremento e decremento delle risorse che possono essere utilizzate da ciascuno dei pods.
 - (c) Incremento e decremento dei nodi del cluster, per aumentare le risorse che ha a disposizione.
3. Fase di **accesso al cluster**: Per poter esporre i servizi all'interno del cluster è necessario integrare un componente che ne fornisca un punto d'accesso, chiamato Ingress. Vengono valutate le problematiche che si incontrano per poter integrare queste risorse e infine si crea un punto di accesso al cluster, per testarne le funzionalità.

Nei capitoli successivi, verranno trattate le due fasi nel dettaglio.

Capitolo 4

Creazione del Cluster

Nel seguente capitolo viene effettuata un'installazione guidata delle macchine virtuali per generare un Cluster Kubernetes composto da un Master Node e due Worker Nodes.

Viene infine dispiegato un Deployment sul cluster appena creato, effettuando considerazioni su come Kubernetes agisce sui nodi e come questi interagiscano con Docker.

4.1 Creazione del Master Node

Procediamo ora con l'installazione della macchina che avrà come compito quello di essere il Master Node del cluster, nonché il nodo che contiene l'API Server di Kubernetes.

Non sono necessarie caratteristiche particolari rispetto ai Worker Nodes, ma bisogna soddisfare i requisiti precedentemente elencati. Nell'implementazione abbiamo utilizzato VirtualBox come software di virtualizzazione, ma questo non rappresenta un vincolo da rispettare.

Una volta installata una macchina consigliamo di installare le VirtualBox Guest-Additions, in quanto permettono alla macchina di essere più reattiva e abilitano la possibilità di avere appunti condivisi in modo da poter fare copia-incolla tra macchina host e macchina guest. Ogni Hypervisor presenta le sue

estensioni, pertanto è consigliato, a prescindere della scelta, di installarle; noi faremo riferimento solamente a quelle su VirtualBox. Per installare le VBoxGuestAddition è necessario effettuare i seguenti passaggi:

```
# Una volta avviata la macchina virtuale, nella barra degli
# strumenti selezionare Dispositivi > Inserisci l'immagine del
↪ CD
# delle Guest Additions... . Questo creerà sul desktop un
# collegamento al cd montato.
# Entrare nella cartella del disco e aprire un terminale al suo
# interno
# Installare le dipendenze necessarie per installare le Guest
# Additions con il comando
sudo apt-get install -y dkms build-essential
↪ linux-headers-generic linux-headers-$(uname -r)

# Passare ad utente root della macchina con il comando
sudo su

# Eseguire lo script di installazione con il comando
./VBoxLinuxAdditions.run

# Una volta terminata l'installazione, riavviare la macchina
↪ con
# il comando
reboot
```

A questo punto è necessario installare una CRI che interagisca con Kubernetes per farlo funzionare; come spiegato nel Capitolo 3.2, nella nostra implementazione si è scelto di installare Docker. A questo punto possiamo installare Kubeadm con i seguenti comandi:

```
# Install HTTPS and curl support, useful to download packages
apt-get update && apt-get install -y apt-transport-https

# Download Kubernetes repository keys and add them inside apt
curl -s https://packages.cloud.google.com/apt/doc/apt-key.gpg |
  → apt-key add -

# Add Kubernetes dependency inside apt configuration
cat <<EOF >/etc/apt/sources.list.d/kubernetes.list
    deb http://apt.kubernetes.io/ kubernetes-xenial main
EOF

# Install kubeadm, kubectl and kubelet
apt-get update && apt-get install -y kubelet kubeadm kubectl

# Start the service
sudo systemctl enable kubelet && sudo systemctl start kubelet
```

Possiamo infine configurare il nostro primo nodo per usare Kubernetes, digitando il comando:

```
kubeadm init --pod-network-cidr=192.168.0.0/16
```

Sarà richiesto un po' di tempo per completare la configurazione perché Kubeadm deve scaricare tutte le immagini dei container che popolano il Control-Plane.

```

Your Kubernetes control-plane has initialized successfully!

To start using your cluster, you need to run the following as a regular user:

mkdir -p $HOME/.kube
sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
sudo chown $(id -u):$(id -g) $HOME/.kube/config

You should now deploy a pod network to the cluster.
Run "kubectl apply -f [podnetwork].yaml" with one of the options listed at:
https://kubernetes.io/docs/concepts/cluster-administration/addons/

Then you can join any number of worker nodes by running the following on each as root:

kubeadm join 10.0.2.15:6443 --token nraztp.shzsm9yvslpvpaW3 \
--discovery-token-ca-cert-hash sha256:7a2133fb4bd188d289b0e9de58e64de55e5cb2727d0fa6a98c04
2211c1ea7411

```

Figura 4.1: Risultato dell'operazione kubeadm init.

L'ultima riga della Figura 4.1 esplicita qual è il comando di join necessario per collegare altre macchine (Worker Nodes) al cluster.

Per configurare kubectl, comando che permette di interagire con gli oggetti di Kubernetes, è necessario seguire i seguenti passaggi:

```

mkdir -p $HOME/.kube
sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
sudo chown $(id -u):$(id -g) $HOME/.kube/config

```

Infine, resta da installare una Container Network Interface (CNI), add-on necessaria per la comunicazione all'interno dei cluster.

Nota: se questo passaggio venisse saltato, il cluster, interamente dispiegato, non sarebbe mai disponibile perché il DNS mancherebbe di configurazioni essenziali.

Nel nostro cluster si è scelto di integrare **Calico**, perché unico Network Plugin testato direttamente dagli sviluppatori Kubernetes:

```

kubectl apply -f
↪ https://docs.projectcalico.org/v3.14/manifests/calico.yaml

```

Per verificare che l'installazione del primo nodo del cluster sia andata a buon fine l'output del seguente comando deve dare un risultato come quello rappresentato dalla Figura 4.2:

```
kubectl get pods {all-namespaces}
```

```
root@kubeadml:/media/sf_DirectoryCondivisa# kubectl get pods --all-namespaces
NAMESPACE      NAME                                                    READY   STATUS    RESTARTS   AGE
kube-system    calico-kube-controllers-789f6df884-dmcdg             1/1     Running  0          30m
kube-system    calico-node-jgbzm                                     1/1     Running  0          30m
kube-system    calico-node-t72jp                                     1/1     Running  0          30m
kube-system    coredns-66bff467f8-bjlch                             1/1     Running  0          62m
kube-system    coredns-66bff467f8-qvgpz                             1/1     Running  0          62m
kube-system    etcd-kubeadml                                         1/1     Running  1          62m
kube-system    kube-apiserver-kubeadml                               1/1     Running  1          62m
kube-system    kube-controller-manager-kubeadml                     1/1     Running  1          62m
kube-system    kube-proxy-2ng6p                                      1/1     Running  0          43m
kube-system    kube-proxy-s7hf4                                      1/1     Running  1          62m
kube-system    kube-scheduler-kubeadml                              1/1     Running  1          62m
```

Figura 4.2: Pods dispiegati all'interno del Control-Plane.

Con il comando indicato si chiede a Kubernetes di elencare tutti i pods contenuti all'interno del cluster all'interno di ogni **Namespace**. Questo rappresenta lo spazio dei nomi all'interno del quale è presente una applicazione che si vuole mantenere concettualmente separata dal resto del cluster. I namespace sono una modalità per dividere le risorse di un cluster tra più utenti: è una buona abitudine quella di incapsulare ogni applicazione dentro ad un namespace separato.

4.2 Creazione dei Worker Nodes

Per installare i nodi Worker è necessario effettuare gli stessi passaggi di configurazione fatti per il nodo Master. Non è possibile duplicare la macchina virtuale del Master perché altrimenti non avremmo Nome Macchina e UUID univoci (richiesti da Kubeadm), di conseguenza bisogna installare una

macchina nuova da zero.

Nota: Non bisogna effettuare l'operazione `kubeadm init` nei nodi successivi al primo perché questo causerebbe la generazione di un nuovo nodo Master.

Prima di procedere con l'operazione di `join`, dobbiamo inserire le macchine sotto la stessa rete virtuale in modo che possano comunicare tra loro. Utilizzando un hypervisor questa operazione è molto semplice, basta generare una rete NAT privata e associarla a tutte le macchine che si vuole facciano parte del cluster. I seguenti passaggi fanno riferimento a VirtualBox:

Per prima cosa andare in `File > Preferenze`:

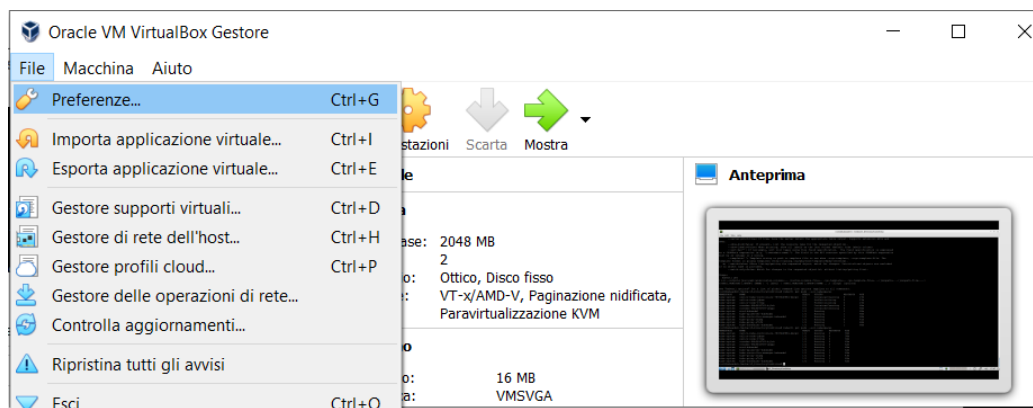


Figura 4.3: Preferenze VirtualBox

Al suo interno, ci spostiamo nella scheda `Rete` e ne creiamo una nuova:

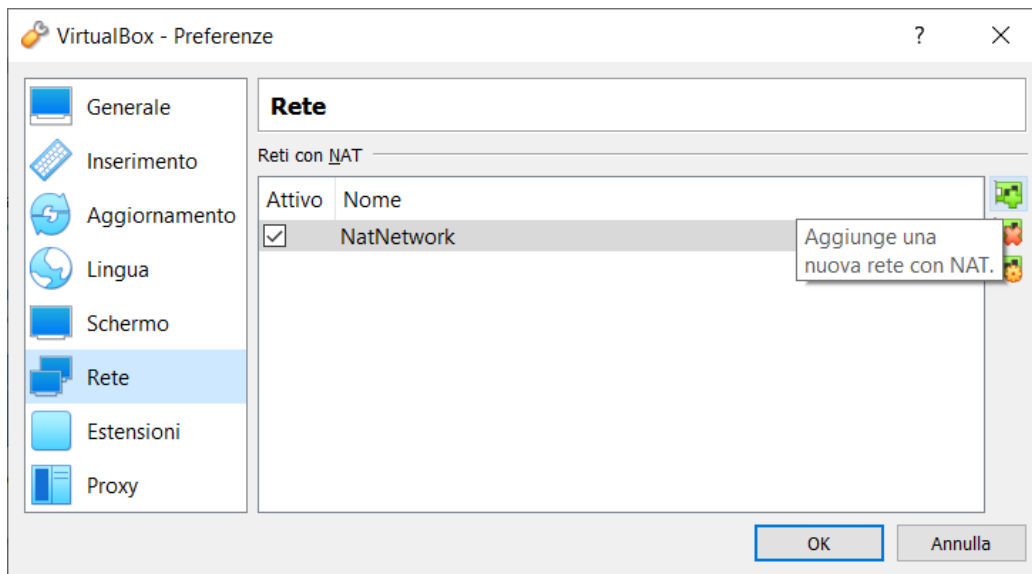


Figura 4.4: Creazione di una Rete NAT

A questo punto configuriamo la rete di TUTTI i nodi del cluster, nelle impostazioni di ciascuna delle macchine:

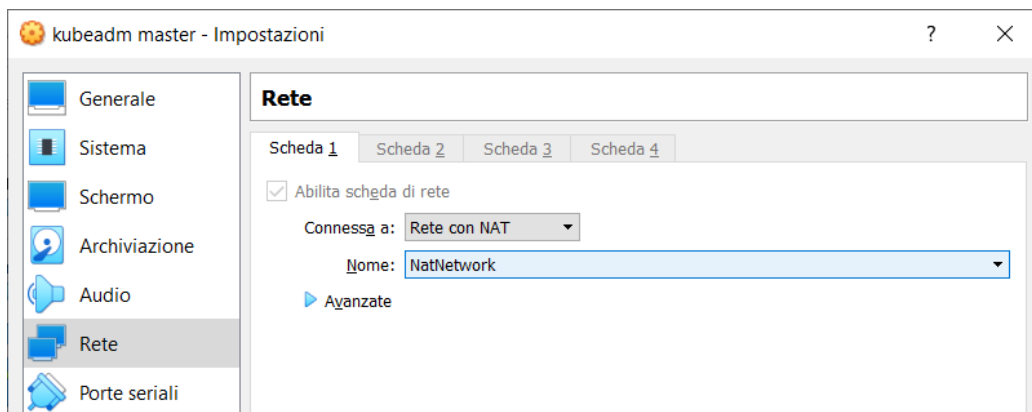


Figura 4.5: Configurazione di rete delle VM VirtualBox

In questo modo ora le due macchine appartengono alla stessa rete virtuale e sono in grado di vedersi a vicenda.

Eventualmente, se si presenta la necessità, è possibile configurare una directory condivisa tra le macchine guest e la macchina host di VirtualBox per condividere file.

Per farlo, a macchine virtuali spente, bisogna andare nelle impostazioni di VirtualBox e selezionare la cartella del proprio File System (dell'host) che si decide di condividere.

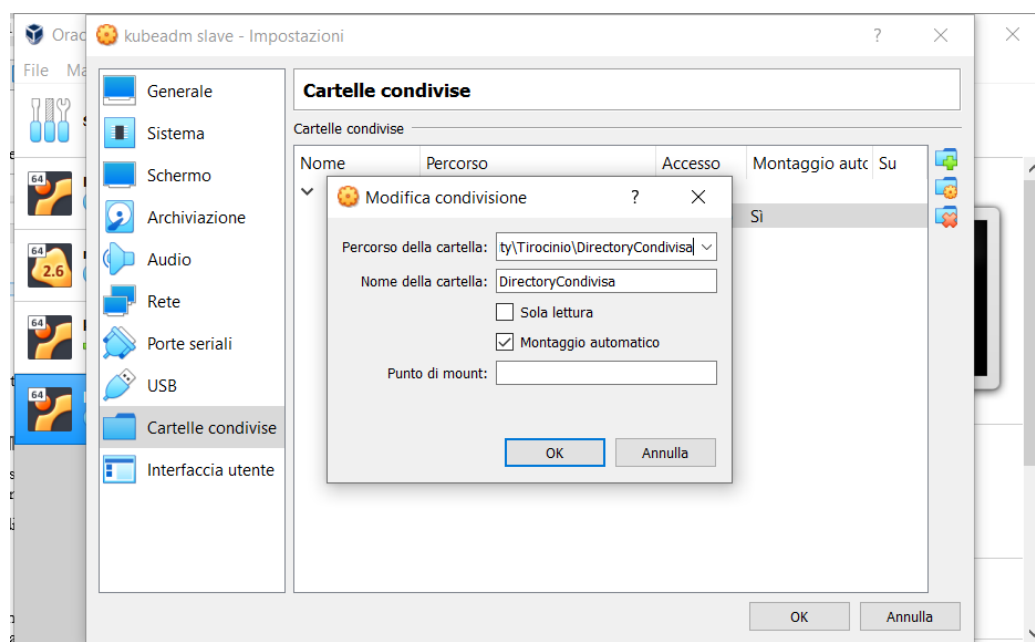


Figura 4.6: Configurazione VirtualBox per aggiungere una cartella condivisa.

Questa cartella verrà vista dalla macchina virtuale come un disco, ma ancora non è possibile accedervi dall'interno delle macchine virtuali.

A questo punto è necessario avviare la macchina virtuale e aggiungere il proprio utente che si sta utilizzando al gruppo di vboxsf (VirtualBox Shared Folder) con il comando:

```
sudo adduser <user> vboxsf
```

Una volta terminato con successo è possibile riavviare la macchina e avere accesso completo alla directory condivisa.

Attenzione! La directory condivisa non vede una copia dei file rispetto alla macchina fisica ma gli STESSI. Di conseguenza è pericoloso dare troppo accesso a tali cartelle per questioni di sicurezza in quanto si è in grado di manomettere i file su disco della macchina FISICA. Per effettuare questa operazione infatti si consiglia sempre di utilizzare una cartella vuota su cui non è possibile avere accesso a informazioni sensibili da parte della macchina virtuale.

È possibile, dopo le precedenti considerazioni, effettuare l'operazione di join, semplicemente eseguendo il comando esposto da kubeadm init sulle macchine da configurare come Worker Nodes.

```
root@kubeadm1:/media/sf_directorycondivisa# kubeadm join 10.0.2.15:6443 --token hmvw/s.3z88vom2wbbcliu --discovery-token-ca-cert-hash sha256:cb3d1e1ff3
399e6d291eb23668e14
K8IS 17:00:29.781023 2413 join.go:346] [preflight] WARNING: JoinControlPlane.controlPlane settings will be ignored when control-plane flag is not set.
[preflight] Running pre-flight checks
[preflight] Reading configuration from the cluster...
[preflight] FYI: You can look at this config file with 'kubectl -n kube-system get cm kubeadm-config -oyaml'
[kubelet-start] Downloading configuration for the kubelet from the "kubelet-config-1.18" ConfigMap in the kube-system namespace
[kubelet-start] Writing kubelet configuration to file "/var/lib/kubelet/config.yaml"
[kubelet-start] Writing kubelet environment file with flags to file "/var/lib/kubelet/kubeadm-flags.env"
[kubelet-start] Starting the kubelet
[kubelet-start] Waiting for the kubelet to perform the TLS Bootstrap...

This node has joined the cluster:
* Certificate signing request was sent to apiservert and a response was received.
* The Kubelet was informed of the new secure connection details.

Run 'kubectl get nodes' on the control-plane to see this node join the cluster.
```

Figura 4.7: Risultato dell'operazione kubeadm join su un Worker Node

Per verificare che l'integrazione del nodo sia avvenuta con successo è possibile eseguire sul Master il comando:

```
kubectl get nodes
```

che visualizza una lista di nodi che fanno parte del cluster e il loro stato. Per avere certezza che la configurazione sia avvenuta con successo i nodi devono avere tutti STATUS=Ready.

```
root@kubeadm1:/media/sf_DirectoryCondivisa# kubectl get nodes
NAME          STATUS    ROLES    AGE   VERSION
kubeadm1     Ready    master   50m   v1.18.2
kubeadm2     Ready    <none>   30m   v1.18.2
```

Figura 4.8: Risultato del comando `kubectl get nodes` dopo aver introdotto un nodo al cluster

Per poter utilizzare `kubectl` all'interno dei Worker Nodes è necessario fare una configurazione aggiuntiva.

Nota: questa configurazione non è obbligatoria in quanto è possibile effettuare tutte le operazioni desiderate dal Control-Plane, ma se si ha la necessità di farle da un nodo Worker allora diventa necessaria.

```
# Copiare il file /etc/kubernetes/admin.conf presente sul
↪ master
# node in ognuno dei worker nodes del cluster.
# Una volta copiato è necessario linkare il file ad una
↪ variabile
# d'ambiente chiamata $KUBECONFIG; dopo quest'ultima
# configurazione il worker node è configurato all'uso di
↪ kubectl.
export KUBECONFIG=/etc/kubernetes/admin.conf
```

Esiste la possibilità di configurare un cluster in modo che tutti i suoi nodi, compreso il Master, possano dispiegare al loro interno i pods. In questa implementazione del cluster si è scelto di lasciare i concetti di Nodo Master e Nodo Slave separati, quindi non vengono dispiegati pods sul nodo Master.

Anche per questo motivo abbiamo scelto di configurare un cluster che avesse due nodi Worker, in modo che si possa apprezzare al meglio il lavoro dei componenti di Kubernetes per distribuire i pods su più nodi.

4.3 Primo Deployment su Cluster

Una volta correttamente configurato, il cluster è pronto per dispiegare applicazioni. L'applicazione proposta è semplice ma permette di comprendere come il cluster viene utilizzato: restituisce la stringa "Hello World!" a chiunque effettui una richiesta. Il seguente file, dal nome **hello-application.yaml**, contiene una descrizione di cosa viene configurato da kubectl all'interno del cluster:

- **kind**: che tipo di oggetto deve essere dispiegato all'interno del cluster; in questo caso un Deployment.
- **name**: il Deployment che verrà dispiegato avrà nome "hello-world"
- **replicas**: il numero di pods che verranno dispiegati di questo Deployment, cioè 2
- **image**: l'immagine di container che verrà utilizzata per l'applicazione
- **containerPort**: i container esporranno la porta 8080
- **protocol**: protocollo con cui viene esposta la porta indicata in containerPort.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: hello-world
spec:
  selector:
    matchLabels:
      run: load-balancer-example
  replicas: 2
  template:
    metadata:
```

```
labels:
  run: load-balancer-example
spec:
  containers:
  - name: hello-world
    image: gcr.io/google-samples/node-hello:1.0
    ports:
    - containerPort: 8080
      protocol: TCP
```

L'immagine del container viene offerta da Google come esempio, infatti viene descritto l'indirizzo a cui è reperibile senza avere la necessità di pre-compilarla all'interno del registry locale.

Il file .yaml stesso viene fornito da k8s.io, quindi non necessita di essere in locale in quanto può essere reperito direttamente dal loro server tramite url con il seguente comando sul nodo master:

```
kubectl apply -f
↪ https://k8s.io/examples/service/access/hello-application.yaml
```

```
root@kubeadml:/home/anitvam# kubectl apply -f https://k8s.io/examples/service/ac
cess/hello-application.yaml
deployment.apps/hello-world created
```

Figura 4.9: Output ottenuto da kubectl dopo aver dispiegato un Deployment

Se si vogliono visualizzare gli attuali deployment attivi nel cluster, è possibile farlo con il comando:

```
kubectl get deployment
```

```
root@kubeadm1:/home/anitvam# kubectl get deployment
NAME          READY   UP-TO-DATE   AVAILABLE   AGE
hello-world   2/2     2             2           4m12s
```

Figura 4.10: Output ottenuto con il comando `kubectl get deployment`

Una cosa interessante da notare è che all'interno del Master Node non si ha l'immagine del container appena dispiegato, perché questa viene scaricata solamente dai Worker Node su cui viene dispiegata l'applicazione. Utilizzando Docker, con il comando:

```
docker images | grep hello
```

eseguito sul Master non avremo risultati, mentre sui Worker è presente l'immagine descritta precedentemente nel file `.yaml`.

```
root@kubeadm1:/home/anitvam# docker images | grep hello
root@kubeadm1:/home/anitvam#
```

Figura 4.11: Output della ricerca dell'immagine del container sul nodo Master

```
root@kubeadm2:/home/anitvam# docker images | grep hello
gcr.io/google-samples/node-hello 1.0          4c7ea8709739    4 years ago
o                                     644MB
```

Figura 4.12: Output della ricerca dell'immagine del container sul nodo Worker

Digitando il comando:

```
kubectl get pods -o wide
```

riusciamo a mettere in evidenza il fatto che sono presenti due pods in stato di **Running**. È ora interessante notare che questi pods hanno indirizzo IP appartenente ad una rete virtuale diversa rispetto a quella degli host, cioè la 192.168.x.x, che corrisponde a quella richiesta nel momento in cui è stato creato il cluster con il flag `--pod-network-cidr`; questo dettaglio è importante in quanto abbiamo configurato una separazione di indirizzi virtuali per i servizi e i pods.

```
root@kubeadm1:/home/anitvam# kubectl get pods -o wide
NAME                READY   STATUS    RESTARTS   AGE   IP              NODE       NOMINATED NODE   READINESS GATES
hello-world-86d6c6f84d-vvftk  1/1     Running   0           13m   192.168.236.195 kubeadm2   <none>           <none>
hello-world-86d6c6f84d-xb927  1/1     Running   0           13m   192.168.236.255 kubeadm2   <none>           <none>
```

Figura 4.13: Pods in esecuzione sul cluster e dettagli sui loro indirizzi IP

Ora, utilizzando Docker, andiamo ad analizzare i container che eseguono questa applicazione sul nodo in cui si trovano, con il comando:

```
docker ps | grep hello
```

```
root@kubeadm2:/home/anitvam# docker ps | grep hello
7358b7f7b672         4c7ea8709739        "/bin/sh -c 'node se..." 19 minutes ago         Up 19 minutes
                                k8s_hello-world_hello-world-86d6c6f84d-vvftk_default_32dfcd15-e41e-4e50-8793
-6e0680943bcc_0
0ebf1c4a5bb5         4c7ea8709739        "/bin/sh -c 'node se..." 19 minutes ago         Up 19 minutes
                                k8s_hello-world_hello-world-86d6c6f84d-xb927_default_9339723d-dc2c-448b-921a
-a0ca836d1ec1_0
c294a71f6b65         k8s.gcr.io/pause:3.2 "/pause"                  19 minutes ago         Up 19 minutes
                                k8s_POD_hello-world-86d6c6f84d-vvftk_default_32dfcd15-e41e-4e50-8793-6e06809
43bcc_0
81cca0f82125         k8s.gcr.io/pause:3.2 "/pause"                  19 minutes ago         Up 19 minutes
                                k8s_POD_hello-world-86d6c6f84d-xb927_default_9339723d-dc2c-448b-921a-a0ca836
d1ec1_0
root@kubeadm2:/home/anitvam#
```

Figura 4.14: Container in esecuzione sul nodo in cui sono stati dispiegati i miei pods

L'output del nodo presenta però quattro container attualmente attivi relativi all'applicazione dispiegata. Questo avviene poichè un pod, come precedentemente descritto, non è altro che un host logico che presenta un'interfaccia di rete e altre caratteristiche tutte condivise dai molteplici container

che possono essere al suo interno. Grazie a questo output siamo in grado di evidenziare il fatto che i nostri due pods attualmente dispiegati contengono ciascuno due container: uno con l'immagine dell'applicazione hello-world e un altro con l'immagine k8s.io/gcr/pause:3.2. Il compito di quest'ultimo container è quello di inizializzare il pod e generare il "contenitore" che poi conterrà i container desiderati. Con il comando:

```
kubectl describe deployments hello-world
```

è possibile visualizzare tutte le caratteristiche relative al Deployment hello-world.

```
root@kubeadml:/home/anitvam# kubectl describe deployments hello-world
Name:                hello-world
Namespace:           default
CreationTimestamp:   Wed, 27 May 2020 11:17:23 +0200
Labels:              <none>
Annotations:         deployment.kubernetes.io/revision: 1
Selector:            run=load-balancer-example
Replicas:            2 desired | 2 updated | 2 total | 2 available | 0 unavailable
StrategyType:        RollingUpdate
MinReadySeconds:     0
RollingUpdateStrategy: 25% max unavailable, 25% max surge
Pod Template:
  Labels:  run=load-balancer-example
  Containers:
    hello-world:
      Image:   gcr.io/google-samples/node-hello:1.0
      Port:   8080/TCP
      Host Port:  0/TCP
      Environment: <none>
      Mounts:     <none>
      Volumes:    <none>
Conditions:
  Type           Status  Reason
  ----           -
  Available      True    MinimumReplicasAvailable
  Progressing    True    NewReplicaSetAvailable
OldReplicaSets: <none>
NewReplicaSet:  hello-world-86d6c6f84d (2/2 replicas created)
Events:
  Type           Reason             Age             From              Message
  ----           -
  Normal         ScalingReplicaSet  33m            deployment-controller  Scaled up replica set hello-world-86d6c6f84d to 2
root@kubeadml:/home/anitvam#
```

Figura 4.15: Descrizione dettagliata del Deployment hello-world, ottenuta con il comando `kubectl describe`

A questo punto, per esporre l'applicazione verso l'esterno possiamo eseguire il seguente comando:

```
kubectl expose deployment hello-world --type=NodePort  
↪ --name=my-service
```

```
root@kubeadml:/home/anitvam# kubectl expose deployment hello-world --type=NodePort --name=my-service  
service/my-service exposed  
root@kubeadml:/home/anitvam#
```

Figura 4.16: Output ottenuto digitando il comando `kubectl expose`

Attenzione! Anche se appartenenti ad una rete logica diversa, si ha la possibilità di comunicare direttamente con i pods dispiegati attraverso il loro indirizzo `192.168.x.x`, ma questa è una comunicazione diretta che in una applicazione reale non porta nessun vantaggio nell'utilizzo di Kubernetes, senza considerare il fatto che questi indirizzi al di fuori del cluster non sono visibili in nessun modo. Il vantaggio di esporre un Service permette di mettere in gioco il meccanismo di Load Balancing interno al Cluster stesso, nel senso che nel momento in cui non si possiedono non solo più pods ma anche più host su cui sono dispiegati, esso automaticamente permette di ridirezionare la richiesta a quello con meno carico. Questo approccio permette di vedere la vera potenzialità dello strumento in un'applicazione reale.

Per visualizzare le informazioni relative al servizio appena abilitato possiamo eseguire il comando:

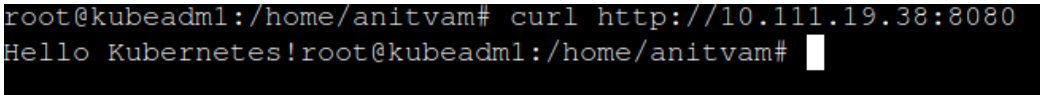
```
kubectl get services my-service
```

```
root@kubeadml:/home/anitvam# kubectl get services my-service  
NAME          TYPE        CLUSTER-IP    EXTERNAL-IP    PORT(S)          AGE  
my-service    NodePort    10.111.19.38  <none>         8080:31239/TCP  52s  
root@kubeadml:/home/anitvam#
```

Figura 4.17: Servizi disponibili all'interno del cluster e IP a loro relativo

In questo modo si ottiene l'indirizzo che è possibile utilizzare per accedere al servizio. Con il seguente comando possiamo effettuare, da un nodo appartenente al cluster, una richiesta alla nostra applicazione:

```
curl http://10.111.19.38:8080
```



```
root@kubeadml:/home/anitvam# curl http://10.111.19.38:8080
Hello Kubernetes!root@kubeadml:/home/anitvam#
```

Figura 4.18: Richiesta al servizio my-service e relativo output

Il campo `--type` di un Service permette di definire come si vuole che quel servizio sia esposto e prevede tre valori:

1. **ClusterIP**. Fornisce un indirizzo IP univoco all'interno del cluster per individuare un servizio. Questo indirizzo non può essere utilizzato dall'esterno del cluster.

Pro: comodo per fare velocemente del debugging, infatti può essere usato solo all'interno del cluster kubernetes.

Contro: questo metodo richiede di eseguire kubectl come utente autenticato, di conseguenza in produzione è sconsigliato in quanto può esporre criticità di sicurezza.

2. **NodePort**. Rappresenta il metodo base per instradare del traffico verso un servizio kubernetes. Questo metodo richiede di aprire una porta specifica nel nodo e tutto il traffico che viene instradato al suo interno viene gestito da quel servizio. Se non viene specificata una porta nel file yaml con cui viene creato tale servizio, kubectl ne sceglie una casuale (come regola generale è bene lasciare che sia kubernetes a scegliere la porta).

Pro: accesso veloce ad un servizio.

Contro: è possibile specificare un solo servizio per porta.

3. **LoadBalancer.** É in grado di gestire molteplici richieste e indirizzi per le risorse all'interno del cluster. Rappresenta la scelta migliore per gestire il traffico in ingresso al cluster.

Pro: riesce a gestire molteplici richieste e riesce a bilanciare efficientemente il traffico.

Contro: non è disponibile di default per tutti i tipi di infrastruttura e a volte diventa complesso da configurare.

Capitolo 5

Scaling del Cluster

Si vuole cercare di creare un cluster Kubernetes che aumenti/diminuisca di dimensione in base al carico di richieste degli utenti. Questa funzionalità, in una applicazione reale, permette di automatizzare la grande problematica del dispiegamento delle applicazioni online.

Ad esempio, se si vuole fornire un certo servizio agli utenti di internet, si vuole fare in modo che questi possano accedervi in qualsiasi momento e senza latenze. Se per caso, in un certo periodo non prevedibile della giornata, si ha un picco di richieste a tale servizio, è possibile che l'infrastruttura non riesca a servire così tanti accessi per mancanza di risorse e quindi la prestazione calerebbe a picco, con la possibilità anche di incorrere in veri e propri "down di servizio".

È comodo avere un software che riesca, automaticamente, a rendersi conto che le risorse disponibili sono poche (oppure analogamente che le richieste sono aumentate) e agire di conseguenza per aumentare le prestazioni del cluster: questo è quello di cui si occupano gli oggetti che effettuano autoscaling all'interno di Kubernetes.

L'autoscaling, all'interno di Kubernetes, si suddivide in tre tipologie:

1. **Horizontal Pod Autoscaler.** Componente che si occupa di scalare automaticamente il numero di repliche di un pod.
2. **Vertical Pod Autoscaler.** Componente che si occupa di modificare le richieste e i limiti nell'uso delle risorse di ogni replica di pod.
3. **Cluster Autoscaler.** Componente che permette di aumentare/ diminuire il numero di nodi all'interno di un cluster.

5.1 Horizontal Pod Autoscaler

L'Horizontal Pod Autoscaler (HPA) è implementato come loop di controllo con periodo di default di 15s, personalizzabile, e si basa sul controllo di metriche. L'algoritmo fondamentale di questo componente è molto semplice:

$$x = N * (c / t)$$

dove:

- **x** rappresenta il numero di repliche desiderato
- **N** rappresenta il numero attuale di repliche esistenti
- **c** rappresenta l'attuale valore della metrica a cui si fa riferimento
- **t** rappresenta il valore di target settato nell'HPA, che deve essere mantenuto superiore al valore corrente della metrica.

Questo calcolo viene effettuato per ogni metrica che viene settata nell'Horizontal Pod Autoscaler per un Deployment.

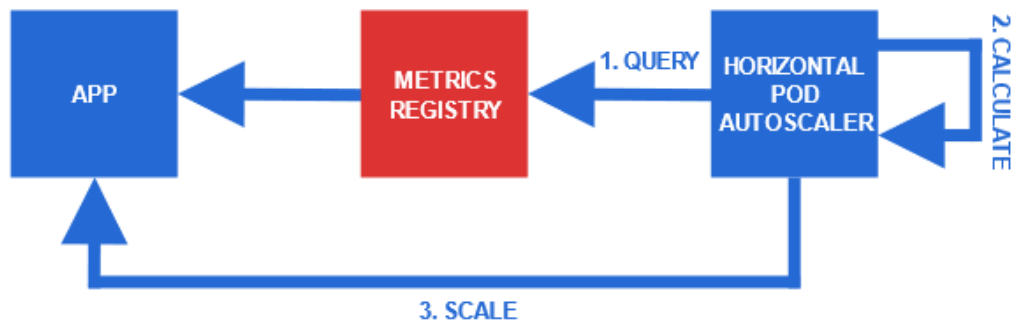


Figura 5.1: Rappresentazione grafica del loop di controllo dell'Horizontal Pod Autoscaler.

Come rappresentato nella Figura 5.1, il funzionamento dell'Horizontal Pod Autoscaler si basa sull'esistenza del componente **Metrics Registry**, il quale si occupa di fornire delle API alle quali l'HPA può fare richiesta. Esistono fondamentalmente tre tipologie di metriche:

1. **Resource Metrics API**, riguardanti risorse di pods/nodi, solitamente di tipo Memoria e CPU. Queste vengono esposte dall'API `metrics.k8s.io`.
2. **Custom Metrics API**, basate sugli oggetti Kubernetes. Queste vengono esposte dall'API `custom.metrics.k8s.io`.
3. **External Metrics API**, non associate ad oggetti Kubernetes. Queste vengono esposte dall'API `external.metrics.k8s.io`.



Figura 5.2: Rappresentazione dettagliata delle tipologie di API esposte dal Metrics Registry.

Per poter usufruire delle metriche bisogna aggiungere al cluster un componente chiamato **Metrics API Server**, che si occupa di fornire le API all'HPA, e di un **Metrics Collector**, che si occupa di collezionare le metriche dalle sorgenti.

Per quanto riguarda le metriche di tipo Resource, il collector più utilizzato si chiama **Container Advisor** (cAdvisor), eseguito di default come parte del kubelet su ogni worker node Kubernetes, mentre per quanto riguarda l'API Server la scelta più gettonata è quella di utilizzare il **Metrics Server** che, successore di **Heapster** (marcato come deprecato nelle ultime versioni di Kubernetes), rappresenta il principale aggregatore di metriche all'interno del cluster.

Parlando invece di Custom e External Metrics, la scelta ricade su software open-source sviluppati esternamente rispetto al progetto Kubernetes, i quali sono: **Prometheus**, **Datadog** e **Google Stackdriver**. Per quanto riguarda gli API Server, ognuno dei collector propone il proprio.

Esistono due versioni dell'HPA:

- **API version autoscaling/V2beta1**: Questa versione dell'API permette di autoscalare i pods in base a metriche che considerano l'utilizzo

di memoria o di CPU.

- **API Version autoscaling/V2beta2:** Questa versione permette di autoscalare i pods sempre su metriche basate su utilizzo di CPU e di memoria ma in aggiunta considera anche le Custom Metrics esposte da un API Server specializzato.

I comandi per gestire l'HPA sono i seguenti:

1. *kubectl autoscale*, comando diretto che permette di scalare un oggetto Kubernetes, tipicamente un Deployment
2. *kubectl get hpa*, permette di visualizzare una lista contenente tutti gli oggetti HPA del cluster. Bisogna considerare che questo comando visualizza solamente quelli che sono stati dispiegati sul namespace default, se si vogliono visualizzare quelli contenuti in namespace differenti bisogna specificarli con il flag -n.
3. *kubectl describe hpa hpa_name*, permette di avere una vista dettagliata dell'HPA specificato e dello stato in cui si trova.

Aumentando il numero di pods all'interno del cluster si ha il vantaggio di poter aumentare il carico di lavoro che può essere supportato dalla nostra applicazione, ma allo stesso tempo si rischia di esaurire le risorse dei nodi, creando una situazione in cui alcuni dei pods dispiegati dall'HPA non riescono ad eseguire. Kubernetes tiene traccia della capacità di ciascuno dei nodi e delle risorse richieste da ciascuno dei pods, in questo modo può aumentare il numero di oggetti tenendo conto delle capacità fisiche delle macchine.

Template di risorsa HPA:

```

{{- if .Values.autoscaling }}
kind: HorizontalPodAutoscaler
apiVersion: autoscaling/v2beta1
metadata:
  name: {{ .Release.Name }}
  namespace: {{ .Release.Namespace }}
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: {{ .Release.Name }}
  minReplicas: {{ .Values.autoscaling.min }}
  maxReplicas: {{ .Values.autoscaling.max }}
  metrics:
  - type: Pods
    pods:
      metricName: {{ .Values.autoscaling.metricName }}
      targetAverageValue: {{ .Values.autoscaling.target }}
{{- end }}

```

Codice 1: Esempio generico di un file YAML per la generazione di un oggetto Horizontal Pod Autoscaler

Gli oggetti Horizontal Pod Autoscaler contengono al loro interno definizioni diverse in base alla tipologia della metrica che si sta considerando per effettuare la valutazione, rispettivamente:

1. **Type: Resource**, indica valori esposti dall'API metrics.k8s.io:

```

metrics:
- type: Resource
  resource:

```

```
name: cpu
target:
  type: Utilization
  averageUtilization: 50
```

Permette di specificare i valori delle risorse come valori diretti o come percentuali. Utilizzando **target.type** con valore **AverageValue** e assegnando il valore a **target.averageValue** si può specificare un valore diretto; utilizzando **target.type** con valore **Utilization** e settando il valore del campo **target.averageUtilization** si può specificare un valore in percentuale, come riportato dall'esempio.

2. **Type: Pods**, indica metriche esposte dall'API custom.k8s.io e tratta valori che vengono calcolati facendo la media tra tutti i pods dispiegati:

```
type: Pods
pods:
  metric:
    name: packets-per-second
  target:
    type: AverageValue
    averageValue: 1k
```

Questo tipo di oggetto supporta solamente risorse specificate come valori interi con **target.type = AverageValue**.

3. **Type: Object**, indica metriche sempre esposte dall'API custom.metrics.k8s.io ma che descrivono oggetti differenti rispetto ai pods. Tipicamente queste tipologie di metriche si occupano di descrivere un oggetto senza prelevarne informazioni:

```
type: Object
object:
  metric:
    name: requests-per-second
  describedObject:
    apiVersion: networking.k8s.io/v1beta1
    kind: Ingress
    name: main-route
  target:
    type: Value
    value: 2k
```

Sono supportati due valori di **target.type**: **Value** e **AverageValue**. Value specifica un valore diretto che viene confrontato con il valore della metrica stessa, mentre AverageValue divide il valore ottenuto dalla metrica per il numero di pods dispiegati di quel deployment.

Nota: Nel caso in cui si vogliono utilizzare metriche multiple all'interno degli oggetti Horizontal Pod Autoscaler, verranno calcolate le repliche necessarie per soddisfare ciascuno dei vincoli inseriti, ma verrà scelta quella con il numero più elevato.

Per descrivere le metriche desiderate in maniera più precisa, l'Horizontal Pod Autoscaler fornisce la possibilità di inserire anche delle **labels**. Queste possono essere specificate solamente quando viene utilizzato **type: Object** e vengono specificate con **selector**.

Ad esempio:

```
type: Object
object:
```

```
metric:  
  name: http_requests  
  selector: {matchLabels: {verb: GET}}
```

Nel caso in cui si vogliono utilizzare metriche di tipo `EXTERNAL` all'interno del cluster (esposte dall'opportuna API `external.metrics.k8s.io`) queste devono essere specificate attraverso `type: External`.

Ad esempio:

```
type: External  
external:  
  metric:  
    name: queue_messages_ready  
    selector: "queue=worker_tasks"  
  target:  
    type: AverageValue  
    averageValue: 30
```

Anche con `type: External` è possibile specificare una label, come per `Object`.

Nota: La documentazione di Kubernetes consiglia, dove possibile, l'utilizzo di metriche di tipo `External`, favorendo quelle di tipo `Custom`, in quanto gli amministratori del cluster hanno la garanzia che siano sicure, mentre quelle `External` potrebbero potenzialmente accedere a qualsiasi metrica del cluster e gli amministratori dovrebbero porci maggiore attenzione nell'usarle, soprattutto in produzione.

5.1.1 Le unità di misura di Kubernetes

Tutte le metriche che vengono specificate negli HPA sono specificate utilizzando una notazione conosciuta in kubernetes come **QUANTITY**. Esempio: 10500m, in notazione decimale rappresenta il valore 10,5ms.

5.2 Vertical Pod Autoscaler

Il **Vertical Pod Autoscaler**[9] (**VPA**) permette di incrementare o diminuire le richieste in termini di CPU e memoria dei pods, in modo da non sfruttare in modo eccessivo le risorse a disposizione del nodo del cluster su cui sono dispiegati.

Il dispiegamento di un VPA presenta tre componenti principali:

1. **Recommender**. Si occupa di monitorare l'utilizzo delle risorse e i valori massimi messi a disposizione dalla macchina.
2. **Updater**. Si occupa di eliminare i pods che necessitano di essere aggiornati con nuovi limiti sulle risorse.

Nota: Attualmente Kubernetes non fornisce la possibilità di AGGIORNARE la configurazione di un pod in esecuzione, ma elimina quello che contiene la versione vecchia per sostituirlo con un nuovo dispiegamento del pod.

3. **Admission Controller**. Si occupa di sovrascrivere le richieste delle risorse per un pod al momento della creazione.

Dalla documentazione del VPA[9] attualmente non è possibile utilizzarlo se non insieme al Metrics Server, che espone metriche relative a utilizzo di CPU e di memoria. È inoltre in fase di sperimentazione l'introduzione della possibilità di effettuare un aggiornamento dei pods durante la loro esecuzione al posto di sostituirli con nuove versioni aggiornate.

5.3 Cluster Autoscaler

Il **Cluster Autoscaler**[10] effettua un loop tra due task principali:

1. Viene controllata la presenza di pods che non possono essere schedulati, se presenti viene calcolato se l'aggiunta di un nuovo nodo al cluster porterebbe la possibilità di sbloccare il pod. In tal caso, se possibile, si occupa di aggiungere un nodo al pool dei nodi.
2. Viene calcolato se è possibile riunire tutti i pods dispiegati su un numero inferiore di nodi, in tal caso i pods verranno rischedulati in un altro nodo e quello in eccesso verrà eliminato.

Il Cluster Autoscaler è uno strumento molto potente che però non può essere implementato nel nostro cluster composto di macchine virtuali, in quanto, ad oggi, necessita supporto da parte di un cloud provider che gli dia la possibilità di generare ed eliminare macchine a piacere.

5.4 Metrics Server

Il **Metrics Server**[11] è uno strumento che permette l'integrazione della scalabilità automatizzata in Kubernetes attraverso l'uso di risorse come HPA e VPA. Una volta installato, permette di essere consultato attraverso l'API `v1beta1.metrics.k8s.io`.

Per vedere l'elenco completo delle API esposte all'interno del cluster basta digitare il comando:

```
kubectl get APIService
```

Dove gli `APIService` sono gli oggetti che definiscono le API e quello che devono fare. Nel caso del Metrics Server, l'API esposta inoltra le richieste al Service dal nome `metrics-server` del namespace `kube-system`, come visibile nella Figura 5.3.


```
root@kubeadml:/home/anitvam# kubectl describe APIService v1beta1.metrics.k8s.io
Name:          v1beta1.metrics.k8s.io
Namespace:
Labels:        <none>
Annotations:   API Version:  apiregistration.k8s.io/v1
Kind:          APIService
Metadata:
  Creation Timestamp:  2020-07-21T08:09:58Z
  Resource Version:    362926
  Self Link:           /apis/apiregistration.k8s.io/v1/apiservices/v1beta1.metrics.k8s.io
  UID:                 8ca075d5-b025-4906-8ae7-ef2c791fdeaa
Spec:
  Group:             metrics.k8s.io
  Group Priority Minimum:  100
  Insecure Skip TLS Verify: true
  Service:
    Name:             metrics-server
    Namespace:        kube-system
    Port:              443
    Version:           v1beta1
    Version Priority:   100
Status:
  Conditions:
    Last Transition Time:  2020-08-17T07:53:37Z
    Message:               all checks passed
    Reason:                 Passed
    Status:                 True
    Type:                   Available
Events:                   <none>
root@kubeadml:/home/anitvam#
```

Figura 5.3: Dettagli riguardanti l'oggetto APIService esposto con l'installazione del Metrics Server

Per rilevare le risorse utilizzate, il Metrics Server utilizza il collector chiamato cAdvisor, che di default è installato in tutti i nodi da parte del kubelet.

Le risorse osservate dal Metrics Server sono:

1. **CPU**, indicata come utilizzo medio in core per un periodo di tempo. Il kubelet sceglie la finestra di tempo per il calcolo dell'utilizzo.
2. **Memoria**, indicata come *working set* in byte. Idealmente il "working set" è la quantità di memoria in uso che non può essere liberata in momenti critici, tuttavia il calcolo del set di lavoro varia in base al sistema operativo host e generalmente fa un uso intensivo dell'euristica per produrre una stima. La metrica include in genere anche una memoria cache, poiché il sistema operativo host non può sempre recuperare tali pagine.

I suoi punti di forza sono:

- Un singolo deployment che funziona sulla maggior parte dei cluster
- Supporto fino a 5.000 nodi
- Efficienza delle risorse: Metrics Server utilizza 0,5 m di core di CPU e 4 MB di memoria per nodo

È tuttavia sconsigliato l'impiego di Metrics Server in caso di:

- Cluster non Kubernetes
- Necessità di metriche accurate sull'utilizzo delle risorse
- Scalabilità automatica basata su altre risorse all'infuori di CPU e memoria

Per altri casi d'uso si consiglia l'impiego di altre soluzioni di monitoraggio come **Prometheus**.

Un esempio di HPA a cui viene delegato l'incarico di autoscalare il deployment “dep_name” sulla base dell'utilizzo della CPU è riportato nel file **hpa.yaml**:

```
kind: HorizontalPodAutoscaler
apiVersion: autoscaling/v2beta1
metadata:
  name: hpa_name
spec:
  maxReplicas: 10
  minReplicas: 1
  scaleTargetRef:
    apiVersion: app/v1
    kind: Deployment
```

```
    name: dep_name
  metrics:
  - type: Resource
    resource:
      name: cpu
      targetAverageUtilization: 60
```

Di seguito, viene invece riportato un esempio, nel file **hpa.yaml**, in cui come metrica viene utilizzata la memoria:

```
kind: HorizontalPodAutoscaler
apiVersion: autoscaling/v2beta1
metadata:
  name: hpa_name
spec:
  maxReplicas: 10
  minReplicas: 1
  scaleTargetRef:
    apiVersion: app/v1
    kind: Deployment
    name: dep_name
  metrics:
  - type: Resource
    resource:
      name: memory
      targetAverageValue: 20M
```

5.4.1 Integrazione dell'Horizontal Pod Autoscaler con il Metrics Server

L'obiettivo è quello di riuscire a scalare automaticamente il cluster sulla base di una metrica di tipo Resource desiderata. È quindi necessario integrare all'interno del cluster un **Metrics Server**.

Installazione

Per prima cosa bisogna verificare che il Metrics Server non sia presente di default all'interno del cluster (Kubernetes fornito dai cloud provider lo integra automaticamente di default e Minikube lo possiede già come plugin aggiuntivo), semplicemente digitando il comando:

```
kubectl top nodes
```

oppure:

```
kubectl top pods
```

Questi due comandi permettono di interagire con il Metrics Server e visualizzare lo stato rispettivamente di nodi e pods per quanto riguarda memoria e CPU utilizzata.

Se tale comando restituisce errore significa che il Metrics Server non è presente e bisogna procedere con l'installazione.

Per prima cosa abbiamo quindi scaricato il file `.yaml` dell'ultima release su github del Metrics Server[12].

Prima di dispiegarlo però sono state rese necessarie delle modifiche per supportare la rete utilizzata dalle macchine virtuali, aggiungendo i seguenti flag nella sezione `args` del Deployment:

```
- --kubelet-insecure-tls  
-  
↪ --kubelet-preferred-address-types=InternalIP,ExternalIP,Hostname
```

Nota: Applicando questi flag vengono ignorati i controlli di sicurezza del Metrics Server; bisogna quindi valutare bene questa implicazione prima di utilizzare l'approccio in produzione.

Questa operazione non è necessaria se si sta utilizzando un cluster su infrastruttura Cloud.

Dopo queste considerazioni è ora possibile integrare il Metrics Server all'interno del cluster, utilizzando il comando:

```
kubectl apply -f ./metrics_server.yaml
```

È possibile verificare il corretto funzionamento, digitando nuovamente il comando:

```
kubectl top nodes
```

E ora si dovrebbero visualizzare le i valori delle metriche dei nodi del cluster, come da Figura 5.4.

Nota: Oltre al tempo richiesto per scaricare le immagini necessarie a far funzionare il Metrics Server, necessita un discreto tempo per potersi sincronizzare e ottenere i valori del cluster.

```
root@kubeadm1:/home/anitvam# kubectl top nodes
NAME          CPU(cores)   CPU%   MEMORY(bytes)  MEMORY%
kubeadm1      525m         26%   1543Mi         81%
kubeadm2      156m         15%   934Mi          49%
root@kubeadm1:/home/anitvam# kubectl top pods
NAME          CPU(cores)   MEMORY(bytes)
hello-world-86d6c6f84d-h77nc  0m           8Mi
hello-world-86d6c6f84d-hh88r  0m           16Mi
root@kubeadm1:/home/anitvam#
```

Figura 5.4: Visualizzazione delle metriche del metrics-server relative ai nodi ed ai pods del cluster.

Utilizzo

Per testare il funzionamento dell'Horizontal Pod Autoscaler con il Metrics Server è stata utilizzata la guida della documentazione ufficiale Kubernetes[13], la quale permette in pochi semplici passaggi di vedere il funzionamento dell'HPA.

Dispieghiamo ora un'applicazione sul cluster:

```
kubectl apply -f
↪ https://k8s.io/examples/application/php-apache.yaml
```

La quale presenta le seguenti caratteristiche:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: php-apache
spec:
  selector:
    matchLabels:
```

```
    run: php-apache
replicas: 1
template:
  metadata:
    labels:
      run: php-apache
  spec:
    containers:
    - name: php-apache
      image: k8s.gcr.io/hpa-example
      ports:
      - containerPort: 80
      resources:
        limits:
          cpu: 500m
        requests:
          cpu: 200m
---
apiVersion: v1
kind: Service
metadata:
  name: php-apache
  labels:
    run: php-apache
spec:
  ports:
  - port: 80
  selector:
    run: php-apache
```

E infine generiamo un HPA con il comando:

```
kubectl autoscale deployment php-apache --cpu-percent=50  
↪ --min=1 --max=10
```

Questo oggetto va a tenere traccia dei valori di utilizzo della CPU all'interno dei pods del deployment chiamato php-apache.

Nota: Il pod, nel momento della creazione, hanno settato un valore di utilizzo massimo della CPU di 500m e ne richiedono 200m.

È possibile verificare il funzionamento dell'HPA nel momento in cui si visualizza un valore diverso da <undefined> prima del valore selezionato per scalare il cluster, come da Figura 5.5.

```
root@kubeadm1:/home/anitvam# kubectl get hpa  
NAME          REFERENCE          TARGETS  MINPODS  MAXPODS  REPLICAS  AGE  
php-apache    Deployment/php-apache  0%/50%   1         10        1          8m41s  
root@kubeadm1:/home/anitvam# █
```

Figura 5.5: Dimostrazione che l'oggetto Horizontal Pod Autoscaler è riuscito a raccogliere i valori dagli oggetti che gli sono stati assegnati.

Creando ora un traffico intenso verso il deployment, digitando in una shell vuota:

```
kubectl run -it --rm load-generator --image=busybox /bin/sh
```

e in seguito, dopo aver premuto invio:

```
while true; do wget -q -O- http://php-apache; done
```

è possibile apprezzare che il componente è in grado di vedere il cambiamento nel tempo delle metriche, le quali aumentano di valore; è interessante

osservare anche che l'HPA inizia a dispiegare altre repliche dei pods come da Figura 5.7.

```
root@kubeadm1:/home/anitvam/Downloads# kubectl get hpa
NAME                REFERENCE                TARGETS  MINPODS  MAXPODS  REPLICAS  AGE
php-apache          Deployment/php-apache     247%/50%  1         10        4          11m
root@kubeadm1:/home/anitvam/Downloads#
```

Figura 5.6: Dimostrazione dell'aumento di carico per il Deployment php-apache

```
root@kubeadm1:/home/anitvam/Downloads# kubectl get deployments
NAME                READY  UP-TO-DATE  AVAILABLE  AGE
hello-world         2/2    2            2           55d
php-apache          3/5    5            3           23m
root@kubeadm1:/home/anitvam/Downloads#
```

Figura 5.7: Effettivo aumento nel numero di repliche di php-apache

Dopo un po' di tempo da quando si smette di effettuare le richieste (semplicemente digitando CTRL + C + Q nella bash con la busy-box) l'HPA riesce ad accorgersi che il traffico è calato e di conseguenza riduce il numero di repliche.

```
root@kubeadm1:/home/anitvam# kubectl get hpa
NAME                REFERENCE                TARGETS  MINPODS  MAXPODS  REPLICAS  AGE
php-apache          Deployment/php-apache     0%/50%   1         10        2          20m
root@kubeadm1:/home/anitvam# kubectl get pods
NAME                READY  STATUS      RESTARTS  AGE
hello-world-86d6c6f84d-h77nc  1/1    Running     1          53d
hello-world-86d6c6f84d-hh88r  1/1    Running     1          53d
php-apache-5c4f475bf5-7rvtq   1/1    Running     0          32m
php-apache-5c4f475bf5-rss44   1/1    Terminating 0          9m38s
root@kubeadm1:/home/anitvam#
```

Figura 5.8: Riduzione del numero di repliche del Deployment su cui è attivo l'HPA

Il comando inline, utilizzato per implementare l'HPA, presenta delle limitazioni. Esso infatti non fornisce la possibilità di scalare su più metriche e

non consente di utilizzare quelle di tipo custom: in questi casi si è obbligati a scrivere un file `.yaml` che contenga tutti i dettagli relativi all'HPA e poi dispiegarlo con il comando:

```
kubectl apply -f <file_dir>
```

5.5 Prometheus

Prometheus[14] è uno strumento open-source scritto in Golang che permette di monitorare i carichi di lavoro di un'applicazione offrendo un database temporale per l'interazione con i dati. Prometheus rappresenta una valida scelta (e anche parecchio utilizzata) per monitorare un cluster Kubernetes.

Il valore aggiunto di Prometheus, rispetto all'utilizzo del Metrics Server, è che esso fornisce la possibilità di esporre metriche, anche personalizzate, su molti più ambiti rispetto la memoria e la CPU utilizzata.

Prometheus è un sistema pull-based: invia richieste HTTP chiamate **scrape**, basandosi sulle configurazioni definite nel file di deployment, e la risposta a queste scrape viene memorizzata insieme alla metrica di risposta. La memorizzazione avviene su un database situato sul server di Prometheus che consente di gestire un grande flusso di dati; con un singolo server è possibile gestire simultaneamente migliaia di macchine.

L'unico modo per poter accedere alle informazioni su memoria, spazio su disco, utilizzo della CPU e larghezza di banda è quello di utilizzare un **Node Exporter** (cAdvisor è uno di questi). Gli exporters sono software che vengono affiancati all'applicazione e hanno il compito di:

- Accettare richieste HTTP da Prometheus.
- Assicurarsi che le informazioni siano in un formato supportato.
- Richiedere le informazioni al Prometheus server.

Il Prometheus Adapter è lo strumento che permette di fruire delle metriche raccolte da Prometheus, esponendole attraverso API che possono essere lette dall'HPA.

Una volta che il sistema ha collezionato le informazioni, è possibile accedervi utilizzando il PromQL Query Language, graficamente attraverso interfacce come Grafana oppure utilizzando notifiche attraverso l'Alertmanager.

Il Prometheus Server mette a disposizione una Dashboard sulla quale è possibile effettuare query con il linguaggio PromQL. I risultati di queste query possono essere anche rappresentati attraverso grafici. Grafana rappresenta una interfaccia alternativa esterna al Prometheus Server che fornisce sempre possibilità di effettuare rappresentazioni grafiche dei dati.

Prometheus fornisce quattro tipologie di metriche:

1. **Counter**, metrica il cui valore può solo aumentare o ripartire da zero (numero di richieste esaudite, task completati, errori, ...)
2. **Gauge**, singolo valore numerico che può aumentare o diminuire (utilizzo di memoria, temperature, ...)
3. **Histogram**, campionatura delle durate (tempi di risposta/ di richiesta) che vengono messe in raccolte configurabili. Queste metriche forniscono anche una somma dei valori osservati.
4. **Summary**, come un Histogram ma inoltre permette di calcolare quantili (valori statistici che dividono il dominio in intervalli di stessa probabilità) configurabili sopra finestre a tempo scorrevole.

Prometheus però non è stato sviluppato direttamente per l'utilizzo all'interno di un cluster Kubernetes ma nasce come applicativo per monitorare applicazioni di qualsiasi natura. Diverse sono le possibilità per integrare Prometheus all'interno di un cluster in modo ottimizzato, ma la modalità più semplice e diffusa è quella che utilizza un software chiamato **Prometheus Operator**.

5.5.1 Prometheus Operator

Prometheus Operator[15] è uno strumento che permette di integrare Prometheus all'interno di un cluster Kubernetes pre-esistente. La sua notorietà sta proprio nel fatto che permette di installare un oggetto complesso come Prometheus all'interno di un cluster adattandosi in modo completo e automatico alla configurazione e versione del cluster utilizzato.

Questo software si basa sull'idea di separare le istanze di Prometheus da quelle di configurazione (cioè di quali entità si stanno osservando). Per aiutarsi nel separare i concetti, introduce nuove **Custom Resource Definition (CRD)** all'interno del cluster, tra cui *Prometheus* e *ServiceMonitor*. Le CRD non sono altro che API Object Kubernetes aggiuntivi che il Control-Plane integra tra quelli che gestisce.

Di default, Prometheus non osserva le metriche dei servizi esposti da un utente, ma solamente gli elementi del Control-Plane e quelli di se stesso; proprio per questo motivo è necessario utilizzare le CRD.

Le CRD introdotte da Prometheus Operator sono:

1. **Prometheus**, risorsa che permette di configurare l'istanza di Prometheus attiva nel cluster. Permette anche di definire con quali label verranno individuati i Service/Pod Monitor per permettere al Prometheus Server di raccogliere le giuste metriche.
2. **AlertManager**
3. **ThanosRules**
4. **ServiceMonitor**, importante perchè permette di definire quali, dei Service dispiegati, devono essere controllati da Prometheus per memorizzarne le metriche esposte.

5. **PodMonitor**, definisce quali, dei Pods attivi, devono essere monitorati da Prometheus, per memorizzarne delle metriche.

6. PrometheusRule

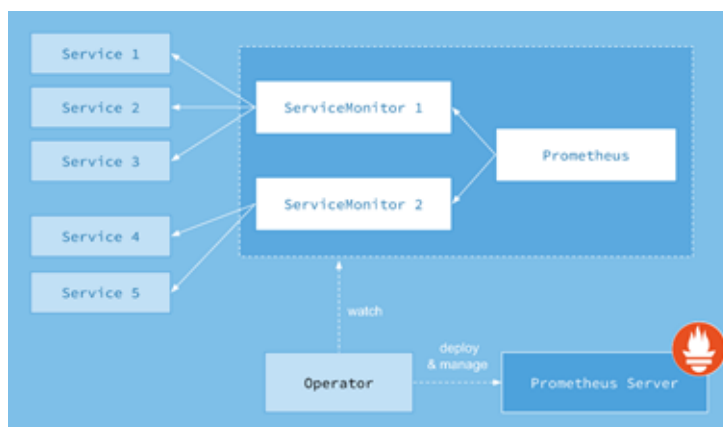


Figura 5.9: Rappresentazione grafica di come Prometheus Raccoglie le Metriche dai Service Kubernetes.

5.5.2 Prometheus Adapter

Per poter utilizzare Prometheus è necessario integrare il Prometheus Adapter[16], nonché il componente che permette di gestire le richieste effettuate sulle API dall'HPA (non fa altro che inoltrare le richieste al Service k8s che deve esaudirle, nel nostro caso il Prometheus Server). Le API che possono essere esposte dagli Adapter sono di tipo `external.metrics.k8s.io` e `custom.metrics.k8s.io`, quella effettivamente utilizzata dipende dal Collector che viene installato.

Logicamente possiamo identificare le componenti di Prometheus come rappresentato nella Figura 5.10.

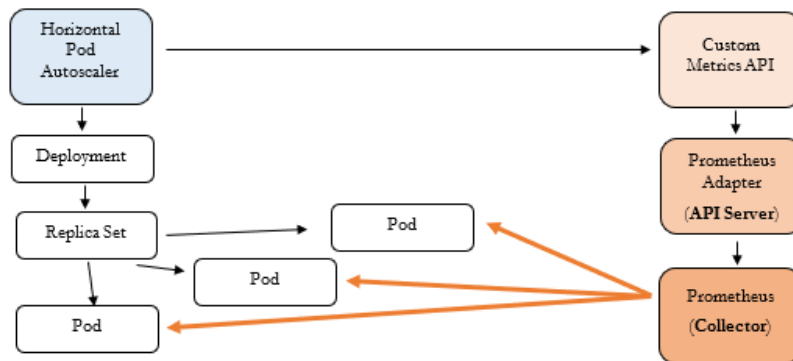


Figura 5.10: Schema che sintetizza le comunicazioni che avvengono tra i vari componenti di Prometheus per effettuare monitoring sul cluster

5.5.3 Kube Prometheus

Kube Prometheus[17] è un repository che permette di includere all'interno del cluster tutto lo stack di applicazioni necessarie per ottenere un Prometheus Server funzionante.

I suoi componenti sono:

- **Prometheus Operator**, utilizzato per integrare Prometheus all'interno di un cluster a prescindere dall'architettura sottostante.
- **Prometheus**
- **Alertmanager**, utilizzato per generare notifiche, tramite la Dashboard di Prometheus, su determinate condizioni delle metriche che si vogliono monitorare.
- **Prometheus node-exporter**, componente scritto in GO che permette di prelevare metriche Hardware e a livello di Sistema Operativo sui *NIX kernels.
- **Prometheus Adapter**, per poter esporre le API riguardanti le metriche.

- **kube-state-metrics**, servizio che si occupa di monitorare il Kubernetes API Server e generare metriche in base allo stato dei suoi oggetti.
- **Grafana**, dashboard interattiva tramite interfaccia web che permette di visualizzare grafici sulle metriche.

Da un punto di vista generale, è possibile pensare a Kube Prometheus come un pacchetto software che automaticamente integra al suo interno Prometheus Operator e tutti i componenti aggiuntivi necessari per effettuare monitoring all'interno di un cluster Kubernetes.

5.5.4 Integrazione Kube Prometheus Stack all'interno del Cluster

All'interno del cluster vogliamo installare tutta la struttura necessaria per poter visualizzare le metriche dei nostri oggetti, pertanto procediamo con l'integrazione del Kube Prometheus all'interno del Cluster.

Installazione

L'installazione può essere semplificata sfruttando il repository di Prometheus Community[18], il quale contiene una serie di pacchetti pronti per l'installazione attraverso l'utilizzo di helm. Il problema principale di helm è che questo non sfrutta lo stesso codice sorgente dei progetti ai quali fa riferimento, ma solamente alcune porzioni che poi rende automatizzate per l'installazione.

Per quanto riguarda il Kube Prometheus, infatti, helm propone un repository chiamato Kube Prometheus Stack[19] che esplicita il fatto che non vengono installate esattamente le stesse dipendenze che installerebbe il progetto Kube Prometheus ma solamente una parte di esse.

Le dipendenze installate da Kube Prometheus Stack, oltre al Prometheus Operator, sono le seguenti:

- `stable/kube-state-metrics`
- `stable/prometheus-node-exporter`
- `grafana/grafana`

Queste, a loro volta, sono altri repository helm che integrano all'interno del cluster tali componenti.

Prima di procedere con questa installazione bisogna accertarsi che non sia installata un'altra istanza di Prometheus o del Metrics Server, in tal caso è necessario disinstallarli prima di procedere con questa configurazione con il comando:

```
helm install prometheus
↪ prometheus-community/kube-prometheus-stack
  --set prometheus.prometheusSpec.
    ↪ serviceMonitorSelectorNilUsesHelmValues=false
  -n monitoring
```

Una volta eseguito, si dovrebbe ottenere in output un risultato simile alla Figura 5.11. Il flag `serviceMonitorSelectorNilUsesHelmValues = false` permette di configurare il Prometheus Server in modo che non sia obbligatorio specificare una label per poter applicare un Service Monitor alla nostra applicazione.


```

NAME: prometheus
LAST DEPLOYED: Mon Sep 14 10:25:16 2020
NAMESPACE: monitoring
STATUS: deployed
REVISION: 1
NOTES:
kube-prometheus-stack has been installed. Check its status by running:
  kubectl --namespace monitoring get pods -l "release=prometheus"

```

Figura 5.11: Risultato dell'esecuzione del comando di installazione di Kube Prometheus Stack

Viene ora riportato l'elenco effettivo dei componenti installati nel cluster:

```

root@K8s-master:/home/anitvam# kubectl get pods -n monitoring -o wide
NAME                                READY   STATUS    RESTARTS   AGE   IP              NODE
alertmanager-prometheus-kube-prometheus-alertmanager-0  2/2     Running   0           15m   192.168.194.119  k8s-worker1
prometheus-grafana-78684d5c95-zpckk  2/2     Running   0           16m   192.168.126.53   k8s-worker2
prometheus-kube-prometheus-operator-79d8949876-xkk9z    2/2     Running   0           16m   192.168.126.52   k8s-worker2
prometheus-kube-state-metrics-95d956569-7xlmr           1/1     Running   0           16m   192.168.194.118  k8s-worker1
prometheus-prometheus-kube-prometheus-prometheus-0     3/3     Running   1           15m   192.168.194.121  k8s-worker1
prometheus-prometheus-node-exporter-4jtqg               1/1     Running   0           16m   10.0.2.15        k8s-worker1
prometheus-prometheus-node-exporter-f4wkc               1/1     Running   0           16m   10.0.2.5         k8s-master
prometheus-prometheus-node-exporter-j59gp               1/1     Running   0           16m   10.0.2.4         k8s-worker2
root@K8s-master:/home/anitvam#

```

Figura 5.12: Pods dispiegati dopo l'installazione del repository prometheus-community/kube-prometheus-stack

Di conseguenza, i pacchetti non installati restano il **Prometheus Adapter** e il **Prometheus Alert Manager**. Il secondo non è un componente essenziale per il funzionamento dello stack di controllo, ma il primo invece lo è, perchè consente di far vedere a Kubernetes le API esposte da Prometheus su custom.metrics.k8s.io.

Procediamo ora con l'installazione del Prometheus Adapter seguendo la guida presente sul repository del progetto[16]:

```

helm install prom-adapter stable/prometheus-adapter --set
↪ prometheus.url=
↪ "http://prometheus-operator-prometheus.monitoring.svc" ,
↪ prometheus.port="9090" --set rbac.create="true" --namespace
↪ monitoring

```

```

NAME: prom-adapter
LAST DEPLOYED: Mon Sep 14 10:43:18 2020
NAMESPACE: monitoring
STATUS: deployed
REVISION: 1
TEST SUITE: None
NOTES:
prom-adapter-prometheus-adapter has been deployed.
in a few minutes you should be able to list metrics using the following command(s):

kubectl get --raw /apis/custom.metrics.k8s.io/v1beta1

```

Figura 5.13: Risultato atteso dopo aver eseguito l'installazione del Prometheus Adapter.

È di fondamentale importanza settare correttamente i flag **prometheus.url** e **prometheus.port**, necessari per far riconoscere all'adapter la posizione all'interno del cluster del Prometheus Server. Con **prometheus.url** si specifica il nome DNS relativo al Service che fa da entry al Prometheus Server, informazioni che si possono facilmente ottenere elencando i servizi esposti nel namespace in cui sono stati esposti i componenti di Prometheus, come visibile in Figura 5.14:

```
kubectl get svc -n monitoring
```

```

root@K8s-master:/home/anitvam# kubectl get svc -n monitoring
NAME                                TYPE          CLUSTER-IP      EXTERNAL-IP      PORT(S)
alertmanager-operated              ClusterIP      None             <none>            9093/TCP,9094/TCP,9094/UDP
prom-adapter-prometheus-adapter     ClusterIP      10.109.25.36    <none>            443/TCP
prometheus-operated                ClusterIP      None             <none>            9090/TCP
prometheus-operator-alertmanager    ClusterIP      10.108.54.220   <none>            9093/TCP
prometheus-operator-grafana         ClusterIP      10.102.80.215   <none>            80/TCP
prometheus-operator-kube-state-metrics ClusterIP      10.98.147.171   <none>            8080/TCP
prometheus-operator-operator         ClusterIP      10.106.100.69   <none>            8080/TCP,443/TCP
prometheus-operator-prometheus      ClusterIP      10.107.95.24    <none>            9090/TCP
prometheus-operator-prometheus-node-exporter ClusterIP      10.109.223.226   <none>            9100/TCP

```

Figura 5.14: Elenco di tutti i servizi esposti all'interno del cluster relativi a Prometheus.

Nel nostro caso il Prometheus API Server, dal nome *prometheus-operator-prometheus* si trova all'indirizzo 10.107.95.24 e sulla porta 9090, quindi possiamo facilmente ricavare il suo nome DNS che sarà:

```
prometheus-operator-prometheus.monitoring.svc.cluster.local.
```

Per avere la conferma che il Prometheus Adapter sia stato installato correttamente e sia funzionante, basta controllare che il suo pod dispiegato sia in stato di esecuzione:

```
kubectl get pods -n monitoring
```

```
root@K8s-master:/home/anitvam# kubectl get pods -n monitoring -o wide
NAME                                READY   STATUS    RESTARTS   AGE   IP              NODE
alertmanager-prometheus-kube-prometheus-alertmanager-0  2/2    Running   0           19m   192.168.194.119  k8s-worker1
prom-adapter-prometheus-adapter-ffd998497-tdx9t          1/1    Running   0           84s   192.168.126.54   k8s-worker2
prometheus-grafana-78684d5c95-zpckk                     2/2    Running   0           19m   192.168.126.53   k8s-worker2
prometheus-kube-prometheus-operator-79d8949876-xkk9z    2/2    Running   0           19m   192.168.126.52   k8s-worker2
prometheus-kube-state-metrics-95d956569-7xlmr           1/1    Running   0           19m   192.168.194.118  k8s-worker1
prometheus-prometheus-kube-prometheus-prometheus-0     3/3    Running   1           18m   192.168.194.121  k8s-worker1
prometheus-prometheus-node-exporter-4jtqg               1/1    Running   0           19m   10.0.2.15        k8s-worker1
prometheus-prometheus-node-exporter-f4wkc               1/1    Running   0           19m   10.0.2.5         k8s-master
prometheus-prometheus-node-exporter-j59gp              1/1    Running   0           19m   10.0.2.4         k8s-worker2
root@K8s-master:/home/anitvam#
```

Figura 5.15: Elenco di tutti i pods esposti all'interno del cluster relativi a Prometheus.

Una volta aver verificato che i pods siano in esecuzione si può eseguire il comando che viene restituito (si veda Figura 5.13) una volta che il prometheus adapter è stato installato:

```
kubectl get --raw /apis/custom.metrics.k8s.io/v1beta1
```

Questo permette di restituire una lista con tutte le API esposte dal Prometheus Adapter che vengono restituite in un formato quasi incomprensibile come in Figura 5.16.

```

File Edit Tabs Help
ed:true,"kind":"MetricValueList","verbs":["get"]},{"name":"namespaces/kube_pod_container_status_waiting","singularName":"","namespaced":false,"kind":"MetricValueList","verbs":["get"]},{"name":"namespaces/kube_replicaset_metadata_generation","singularName":"","namespaced":false,"kind":"MetricValueList","verbs":["get"]},{"name":"statefulsets/apps/kube_statefulset_status_update_revision","singularName":"","namespaced":true,"kind":"MetricValueList","verbs":["get"]},{"name":"jobs.batch/prometheus_sd_updates","singularName":"","namespaced":true,"kind":"MetricValueList","verbs":["get"]},{"name":"namespaces/cpu_usage","singularName":"","namespaced":false,"kind":"MetricValueList","verbs":["get"]},{"name":"services/node_socketat_UDPLITE_inuse","singularName":"","namespaced":true,"kind":"MetricValueList","verbs":["get"]},{"name":"services/serveaccount_valid_tokens","singularName":"","namespaced":true,"kind":"MetricValueList","verbs":["get"]},{"name":"namespaces/coredns_dns_requests","singularName":"","namespaced":false,"kind":"MetricValueList","verbs":["get"]},{"name":"jobs.batch/prometheus_tsdm_compaction_chunk_range_seconds_sum","singularName":"","namespaced":true,"kind":"MetricValueList","verbs":["get"]},{"name":"pods/node_memory_inactive_bytes","singularName":"","namespaced":true,"kind":"MetricValueList","verbs":["get"]},{"name":"services/storage_operation_duration_seconds_count","singularName":"","namespaced":true,"kind":"MetricValueList","verbs":["get"]},{"name":"namespaces/node_memory_usage_bytes","singularName":"","namespaced":false,"kind":"MetricValueList","verbs":["get"]},{"name":"jobs.batch/node_vmstat_pgfault","singularName":"","namespaced":true,"kind":"MetricValueList","verbs":["get"]},{"name":"jobs.batch/grafana_ldap_users_sync_execution_time_count","singularName":"","namespaced":true,"kind":"MetricValueList","verbs":["get"]},{"name":"services/go_gc_duration_seconds_sum","singularName":"","namespaced":true,"kind":"MetricValueList","verbs":["get"]},{"name":"services/node_cooling_device_cur_state","singularName":"","namespaced":true,"kind":"MetricValueList","verbs":["get"]},{"name":"pods/alertmanager_silences_gc_duration_seconds_count","singularName":"","namespaced":true,"kind":"MetricValueList","verbs":["get"]},{"name":"services/prometheus_tsdm_wal_corruptions","singularName":"","namespaced":true,"kind":"MetricValueList","verbs":["get"]},{"name":"jobs.batch/prober_probe","singularName":"","namespaced":true,"kind":"MetricValueList","verbs":["get"]},{"name":"namespaces/memory_rss","singularName":"","namespaced":false,"kind":"MetricValueList","verbs":["get"]},{"name":"namespaces/prometheus_sd_kubernetes_workqueue_latency_seconds_count","singularName":"","namespaced":false,"kind":"MetricValueList","verbs":["get"]},{"name":"services/node_memory_MemFree_bytes","singularName":"","namespaced":true,"kind":"MetricValueList","verbs":["get"]},{"name":"jobs.batch/node_getstat_top_OutSegs","singularName":"","namespaced":true,"kind":"MetricValueList","verbs":["get"]},{"name":"services/kube_dsemonset_status_current_number_scheduled","singularName":"","namespaced":true,"kind":"MetricValueList","verbs":["get"]},{"name":"pods/node_memory_inactive_file_bytes","singularName":"","namespaced":true,"kind":"MetricValueList","verbs":["get"]},{"name":"services/node_cpu","singularName":"","namespaced":true,"kind":"MetricValueList","verbs":["get"]},{"name":"jobs.batch/node_socketat_UDPLITE_inuse","singularName":"","namespaced":true,"kind":"MetricValueList","verbs":["get"]},{"name":"jobs.batch/alertmanager_dispatcher_aggregation_groups","singularName":"","namespaced":true,"kind":"MetricValueList","verbs":["get"]},{"name":"services/kube_certificateSigningrequest_conditions","singularName":"","namespaced":true,"kind":"MetricValueList","verbs":["get"]},{"name":"services/instance_device/node_disk_io_time_seconds_stats","singularName":"","namespaced":true,"kind":"MetricValueList","verbs":["get"]},{"name":"kubernetes/kubelet_runtime_operations","singularName":"","namespaced":false,"kind":"MetricValueList","verbs":["get"]},{"name":"jobs.batch/node_network_receive_errs","singularName":"","namespaced":true,"kind":"MetricValueList","verbs":["get"]}]]]
root@K8s-master:/home/anitvam#

```

Figura 5.16: Visualizzazione API ottenute dal Prometheus Adapter di default.

Per migliorarne la visualizzazione si consiglia di utilizzare un tool chiamato *jq*, il quale formatta il testo che gli viene passato in input in json, ottenendo una lista più leggibile rispetto la precedente, come da Figura 5.17.

```
File Edit Tabs Help
{
  "singularName": "",
  "namespaced": true,
  "kind": "MetricValueList",
  "verbs": [
    "get"
  ]
},
{
  "name": "services/grafana_page_response_status",
  "singularName": "",
  "namespaced": true,
  "kind": "MetricValueList",
  "verbs": [
    "get"
  ]
},
{
  "name": "namespaces/po_annotata_allot_bytes",
  "singularName": "",
  "namespaced": false,
  "kind": "MetricValueList",
  "verbs": [
    "get"
  ]
},
{
  "name": "jobs.batch/coredns_health_request_duration_seconds_count",
  "singularName": "",
  "namespaced": true,
  "kind": "MetricValueList",
  "verbs": [
    "get"
  ]
},
{
  "name": "jobs.batch/apiserver_request_terminations",
  "singularName": "",
  "namespaced": true,
  "kind": "MetricValueList",
  "verbs": [
    "get"
  ]
}
]
root@k8s-master:~/home/anitvan#
```

Figura 5.17: Visualizzazione API ottenute dal Prometheus Adapter con parser jq.

È possibile visualizzare tale comando anche in un secondo momento rispetto all'installazione utilizzando il seguente comando helm e ottenendo un risultato come in Figura 5.18:

```
helm status prom-adapter -n monitoring
```

```
root@K8s-master:/home/anitvam# helm status prom-adapter -n monitoring
NAME: prom-adapter
LAST DEPLOYED: Thu Sep  3 23:48:50 2020
NAMESPACE: monitoring
STATUS: deployed
REVISION: 1
TEST SUITE: None
NOTES:
prom-adapter-prometheus-adapter has been deployed.
In a few minutes you should be able to list metrics using the following command(s):

  kubectl get --raw /apis/custom.metrics.k8s.io/v1beta1
root@K8s-master:/home/anitvam#
```

Figura 5.18: Output restituito da helm status.

Troubleshooting

Se nell'eseguire la richiesta delle API il risultato ottenuto equivale a quello in Figura 5.19, allora significa che il Prometheus Server non è attivo in modo corretto: il problema può essere derivato da un nodo che non è entrato in funzione oppure dai pods che non sono riusciti ad eseguire in modo corretto.

```
root@K8s-master:/home/anitvam# kubectl get --raw /apis/custom.metrics.k8s.io/v1beta1 | jq .
Error from server (ServiceUnavailable): the server is currently unable to handle the request
root@K8s-master:/home/anitvam#
```

Figura 5.19: Errore tipico che si presenta quando ancora il Prometheus Server non è attivo.

Se invece si ottiene un output come in Figura 5.20, allora il problema può essere derivato dal fatto che il pod del Prometheus Adapter non è ancora entrato in funzione (il che avviene solitamente entro una manciata di minuti). Se il problema persiste può essere dovuto da una configurazione sbagliata del Prometheus Adapter, avvenuta durante la sua installazione. In questo caso c'è la possibilità che i parametri che sono stati passati ad helm per installarlo fossero sbagliati; bisogna procedere con una disinstallazione della versione sbagliata con il comando:

```
helm delete prom-adapter -n monitoring
```

per poi reinstallarlo con i parametri corretti.

```
root@K8s-master:/home/anitvam# kubectl get --raw /apis/custom.metrics.k8s.io/v1beta1 | jq .
{
  "kind": "APIResourceList",
  "apiVersion": "v1",
  "groupVersion": "custom.metrics.k8s.io/v1beta1",
  "resources": []
}
root@K8s-master:/home/anitvam#
```

Figura 5.20: Errore tipico che si presenta quando l'installazione del Prometheus Adapter non è stata correttamente configurata.

Utilizzo

Per poter ora testare le potenzialità di Prometheus, si è scelto di implementare l'esempio proposto[20] dal team del Prometheus Adapter con qualche piccolo accorgimento.

Per prima cosa aggiungiamo al cluster il deployment di un'applicazione che espone una metrica: il numero di accessi che vengono effettuati (metrica di tipo gauge).

Creando un file chiamato **sample-app-depoy.yaml**, inseriamo al suo interno il seguente codice YAML:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: sample-app
  labels:
    app: sample-app
spec:
  replicas: 1
  selector:
```

```
matchLabels:
  app: sample-app
template:
  metadata:
    labels:
      app: sample-app
  spec:
    containers:
      - image: luxas/autoscale-demo:v0.1.2
        name: metrics-provider
        ports:
          - name: http
            containerPort: 8080
```

Per poi dispiegarlo all'interno del cluster con il comando:

```
kubectl apply -f sample-app-deploy.yaml -n sample-app
```

Esponiamo ora il servizio associato a questo Deployment in modo da potervi accedere, eseguendo il comando:

```
kubectl expose deploy sample-app -n sample-app
↪ --type=ClusterIP --name sample-app
```

Per riuscire ad effettuare una richiesta al servizio appena esposto, si ha prima la necessità di visualizzare indirizzo IP e porta su cui espone il servizio con il comando:

```
kubectl get svc -n sample-app
```

```
root@K8s-master:/home/anitvam# kubectl get svc -n sample-app
NAME          TYPE          CLUSTER-IP    EXTERNAL-IP    PORT(S)
sample-app    ClusterIP     10.109.238.11 <none>         8080/TCP
root@K8s-master:/home/anitvam#
```

Figura 5.21: Servizi esposti nel namespace sample-app.

Nel nostro caso, visto che l'indirizzo IP associato al servizio appena esposto è il 10.109.238.11 sulla porta 8080, effettuiamo una richiesta al path /metrics per vedere la metrica esposta dall'applicazione:

```
curl 10.109.238.11:8080/metrics
```

```
root@K8s-master:/home/anitvam# curl 10.109.238.11:8080/metrics
# HELP http_requests_total The amount of requests served by the server in total
# TYPE http_requests_total counter
http_requests_total 9696
```

Figura 5.22: Richiesta effettuata alla metrica di sample-app. In risposta ottengo la metrica con la sintassi da permettere a Prometheus di identificarla.

Per poter esporre tale metrica a Prometheus però, non basta avere una sintassi corretta esposta dall'applicazione, ma serve un oggetto ServiceMonitor che permetta al Prometheus Server di riconoscere che si sta esponendo una metrica che deve essere rilevata su tale servizio, specificando l'URL su cui è possibile trovarla e la porta alla quale il servizio accetta le richieste.

Creiamo il file **sample-app-service-monitor.yaml** con il seguente contenuto:

```
kind: ServiceMonitor
apiVersion: monitoring.coreos.com/v1
metadata:
  name: sample-app
```

```
labels:
  app: sample-app
  namespace: monitoring
spec:
  namespaceSelector:
    matchNames:
      - sample-app
  selector:
    matchLabels:
      app: sample-app
  endpoints:
    - interval: 15s
      path: /metrics
      targetPort: 8080
```

Il contenuto è leggermente modificato rispetto a quello proposto dal documento a cui si sta facendo riferimento, in particolare per quanto riguarda i seguenti punti:

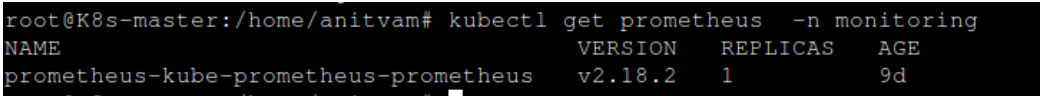
- La nostra configurazione di Prometheus riesce a visualizzare solamente gli oggetti ServiceMonitor presenti sullo stesso namespace degli oggetti Prometheus, il cui nome è monitoring.
- Sono state aggiunte maggiori descrizioni sulle caratteristiche relative a dove il Prometheus Server può trovare la metrica desiderata. Nel nostro caso si troverà alla porta 8080 e al path/metrics.

Sarebbe stato necessario aggiungere la label **release: prometheus** al Service Monitor, perchè il Prometheus Server, se non configurato diversamente, va alla ricerca di oggetti ServiceMonitor sia all'interno del suo stesso namespace, sia tramite label da lui scelta, specificata all'interno della risorsa Prometheus di configurazione. Questo non è stato necessario nel nostro caso perchè, quando abbiamo effettuato l'installazione di Kube Prometheus,

è stato settato appositamente il flag `serviceMonitorSelectorNilUsesHelmValues=false` che permette, come descritto nella documentazione[21], di intercettare tutti i `ServiceMonitor` presenti nello stesso namespace del Prometheus Server in modo automatico, senza la necessità di trovare una corrispondenza con una eventuale label.

È possibile visualizzare dove si trovino queste label all'interno del file di configurazione di Prometheus, ossia in una sezione nominata **serviceMonitorSelectors**. Questa risorsa è una sola ed è visualizzabile con il seguente comando:

```
kubectl get prometheus -n monitoring
```



```
root@K8s-master:/home/anitvam# kubectl get prometheus -n monitoring
NAME                                VERSION  REPLICAS  AGE
prometheus-kube-prometheus-prometheus  v2.18.2    1          9d
```

Figura 5.23: Risorse di tipo Prometheus presenti all'interno del cluster nel namespace monitoring.

Andiamo ora a controllare dove si trova la label appena descritta all'interno di questo file; di default l'output risulta essere come quello in Figura 5.24:

```
kubectl edit prometheus prometheus-kube-prometheus-prometheus
↵ -n monitoring
```

dove `prometheus-kube-prometheus-prometheus` è il nome dell'oggetto di tipo Prometheus.

```
root@K8s-master:/home/anitvam# kubectl get prometheus -n monitoring
NAME                                VERSION  REPLICAS  AGE
prometheus-kube-prometheus-prometheus  v2.18.2    1          9d
```

Figura 5.24: Estratto del file di configurazione di Prometheus contenente la sezione dove deve essere specificata la label per selezionare i Service Monitor.

Aggiungiamo ora il ServiceMonitor all'interno del nostro cluster in modo che la metrica della nostra applicazione sia letta dal Prometheus Server:

```
kubectl apply -f sample-app-service-monitor.yaml
```

Per avere la certezza che Prometheus sia riuscito a prelevare la nostra metrica permettendo in questo modo al Prometheus Adapter di esporre l'API in modo da poterla utilizzare proviamo ad effettuare una ricerca su tutte le API disponibili per vedere se la troviamo, comparando come da Figura 5.25.

```
kubectl get --raw /apis/custom.metrics.k8s.io/v1beta1 | jq . |
↪ grep http_requests
```

```

root@K8s-master:/home/anitvam# kubectl get --raw /apis/custom.metrics.k8s.io/v1beta1 | jq . |
grep http_requests
  "name": "namespaces/kubelet_http_requests",
  "name": "nodes/kubelet_http_requests",
  "name": "jobs.batch/kubelet_http_requests",
  "name": "namespaces/kubelet_http_requests_duration_seconds_sum",
  "name": "services/http_requests",
  "name": "namespaces/alertmanager_http_requests_in_flight",
  "name": "jobs.batch/prometheus_http_requests",
  "name": "pods/alertmanager_http_requests_in_flight",
  "name": "pods/http_requests",
  "name": "services/kubelet_http_requests",
  "name": "jobs.batch/kubelet_http_requests_duration_seconds_count",
  "name": "services/alertmanager_http_requests_in_flight",
  "name": "pods/prometheus_http_requests",
  "name": "services/prometheus_http_requests",
  "name": "jobs.batch/kubelet_http_requests_duration_seconds_bucket",
  "name": "services/kubelet_http_requests_duration_seconds_sum",
  "name": "nodes/kubelet_http_requests_duration_seconds_count",
  "name": "jobs.batch/alertmanager_http_requests_in_flight",
  "name": "namespaces/http_requests",
  "name": "jobs.batch/http_requests",
  "name": "jobs.batch/kubelet_http_requests_duration_seconds_sum",
  "name": "namespaces/prometheus_http_requests",
  "name": "nodes/kubelet_http_requests_duration_seconds_bucket",
  "name": "nodes/kubelet_http_requests_duration_seconds_sum",
  "name": "namespaces/kubelet_http_requests_duration_seconds_count",
  "name": "services/kubelet_http_requests_duration_seconds_bucket",
  "name": "namespaces/kubelet_http_requests_duration_seconds_bucket",
  "name": "services/kubelet_http_requests_duration_seconds_count",

```

Figura 5.25: Filtro sulle metriche esposte da Prometheus Operator in cui compare anche la metrica esposta dalla nostra applicazione.

Una volta che si ha a disposizione la metrica da parte del Prometheus Operator è ora possibile utilizzarla a piacere all'interno di un oggetto HPA.

Aggiungiamo un oggetto HPA che scala in base all'uso di questa metrica; chiamiamo il file **sample-app-hpa.yaml**:

```

kind: HorizontalPodAutoscaler
apiVersion: autoscaling/v2beta1
metadata:
  name: sample-app
  namespace: sample-app
spec:
  scaleTargetRef:
    # point the HPA at the sample application

```

```
# you created above
apiVersion: apps/v1
kind: Deployment
name: sample-app
# autoscale between 1 and 10 replicas
minReplicas: 1
maxReplicas: 10
metrics:
# use a "Pods" metric, which takes the average of the
# given metric across all pods controlled by the autoscaling
↪ target
- type: Pods
  pods:
    # use the metric that you used above: pods/http_requests
    metricName: http_requests
    # target 500 milli-requests per second,
    # which is 1 request every two seconds
    targetAverageValue: 500m
```

Con questa configurazione, diciamo al file HPA che ciascuna delle repliche dei pods dispiegati per questa applicazione gestisca 500 milli-richieste al secondo, ossia una richiesta ogni due secondi: l'HPA si preoccuperà di cercare di mantenere il numero di richieste inferiore a questo valore; integriamo ora il file all'interno del cluster con il seguente comando:

```
kubectl apply -f sample-app-hpa.yaml
```

Se ora andiamo a visualizzare il valore attuale registrato dall'HPA dovremmo vedere che si riesce a calcolare il valore della metrica come in Figura 5.26; otteniamo in questo modo la garanzia che l'oggetto abbia accesso alla metrica. Se il valore fosse rimasto `<undefined>`, allora sarebbe significato che l'HPA ha avuto problemi nel reperirla:

```
kubectl get hpa -n sample-app
```

```
root@K8s-master:/home/anitvam# kubectl get hpa sample-app -n sample-app
NAME          REFERENCE          TARGETS   MINPODS   MAXPODS   REPLICAS
sample-app    Deployment/sample-app  66m/500m   1         10        1
```

Figura 5.26: Visualizzazione del valore attuale della metrica riscontrato dall'Horizontal Pod Autoscaler.

A questo punto possiamo testare l'effettivo funzionamento dello scaling sulla metrica dell'HPA. Generiamo del carico da un'altra bash con il comando:

```
while true; do wget -q -O- http://10.109.238.11:8080; done
```

Come output otterremo una situazione come quella rappresentata in Figura 5.27, ossia i pods dell'applicazione iniziano a rispondere e, passato un lasso di tempo, aumenta il numero di pods che mi risponde, siccome stiamo generando un elevato flusso di richieste.

```
Hello! My name is sample-app-7cfb596f98-h7gqj. I have served 617 requests so far.
Hello! My name is sample-app-7cfb596f98-vjzq2. I have served 63 requests so far.
Hello! My name is sample-app-7cfb596f98-grf2f. I have served 70 requests so far.
Hello! My name is sample-app-7cfb596f98-vjzq2. I have served 64 requests so far.
Hello! My name is sample-app-7cfb596f98-vn4rp. I have served 100 requests so far.
Hello! My name is sample-app-7cfb596f98-7gpnh. I have served 2565 requests so far.
Hello! My name is sample-app-7cfb596f98-grf2f. I have served 71 requests so far.
Hello! My name is sample-app-7cfb596f98-vn4rp. I have served 101 requests so far.
Hello! My name is sample-app-7cfb596f98-7gpnh. I have served 2566 requests so far.
Hello! My name is sample-app-7cfb596f98-h78qj. I have served 818 requests so far.
Hello! My name is sample-app-7cfb596f98-7gpnh. I have served 2567 requests so far.
Hello! My name is sample-app-7cfb596f98-psnf9. I have served 913 requests so far.
```

Figura 5.27: I pods dell'applicazione iniziano ad aumentare di numero a fronte di un forte carico di richieste.

In questo modo stiamo indirettamente notando che l'HPA sta funzionando correttamente, intercettando l'aumento di traffico e riuscendo ad aumen-

tare dinamicamente il numero di pods per mantenere l'applicazione sotto i requisiti richiesti.

Smettendo infine di effettuare traffico notiamo che l'HPA diminuisce di valore e dopo un po' di tempo cala anche il numero di pods dispiegati.

```
root@K8s-master:/home/anitvam# kubectl get hpa sample-app -n sample-app
NAME          REFERENCE          TARGETS          MINPODS          MAXPODS          REPLICAS
sample-app    Deployment/sample-app  1305m/500m      1                10               1
root@K8s-master:/home/anitvam# kubectl get hpa sample-app -n sample-app
NAME          REFERENCE          TARGETS          MINPODS          MAXPODS          REPLICAS
sample-app    Deployment/sample-app  6966m/500m      1                10               3
root@K8s-master:/home/anitvam# kubectl get hpa sample-app -n sample-app
NAME          REFERENCE          TARGETS          MINPODS          MAXPODS          REPLICAS
sample-app    Deployment/sample-app  66m/500m        1                10               10
root@K8s-master:/home/anitvam# kubectl get hpa sample-app -n sample-app
NAME          REFERENCE          TARGETS          MINPODS          MAXPODS          REPLICAS
sample-app    Deployment/sample-app  66m/500m        1                10               10
root@K8s-master:/home/anitvam# kubectl get hpa sample-app -n sample-app
NAME          REFERENCE          TARGETS          MINPODS          MAXPODS          REPLICAS
sample-app    Deployment/sample-app  62m/500m        1                10               2
```

Figura 5.28: Andamento nel tempo della valutazione dell'Horizontal Pod Autoscaler: prima se ne accorge, poi incrementa il valore delle repliche e nel momento in cui il flusso di richieste cala, riduce il numero di metriche.


```
root@K8s-master:/home/anitvam# kubectl get pods -n sample-app
NAME                                READY   STATUS    RESTARTS
sample-app-7cfb596f98-7gpnh        1/1     Running   0
sample-app-7cfb596f98-87dhn        1/1     Running   0
sample-app-7cfb596f98-grf2f        1/1     Running   0
sample-app-7cfb596f98-h78qj        1/1     Running   0
sample-app-7cfb596f98-pbwrr        1/1     Running   0
sample-app-7cfb596f98-plztp        1/1     Running   0
sample-app-7cfb596f98-psnf9        1/1     Running   0
sample-app-7cfb596f98-r6nlj        1/1     Running   0
sample-app-7cfb596f98-vjzq2        1/1     Running   0
sample-app-7cfb596f98-vn4rp        1/1     Running   0
root@K8s-master:/home/anitvam# kubectl get pods -n sample-app
NAME                                READY   STATUS    RESTARTS
sample-app-7cfb596f98-7gpnh        1/1     Running   0
```

Figura 5.29: Confronto della presenza dei pods all'interno del cluster. Il primo elenco fa riferimento a quando il flusso all'applicazione era elevato, mentre il secondo fa riferimento a quando il ciclo infinito di richieste è stato interrotto.

Capitolo 6

Accesso al Cluster

Sono state fin'ora presentate varie metodologie per aumentare il numero di pods all'interno di un Cluster in modo da poter rispondere ad un aumento non previsto delle richieste ad un servizio, ma non è mai stato considerato come poter accedere dall'esterno.

Con Kubernetes, quando si espone un servizio, tipicamente il suo indirizzo è visibile solamente dall'interno del Cluster stesso, a meno che quest'ultimo non sia dispiegato su infrastruttura Cloud che offre un Load Balancing esterno per accedervi. Con **Load Balancer** si intende un componente che è in grado di associare molteplici richieste a molteplici indirizzi cercando di distribuire il carico nella maniera più efficiente possibile.

Uno strumento che può essere utilizzato per accedere al Cluster è un oggetto **Ingress**.

6.1 Ingress

Gli Ingress [22] sono API object che consentono di effettuare l'accesso al cluster (tipicamente HTTP/S) dall'esterno, fornendo anche **Load Balan-**

cing, SSL e Hosting Virtuale.

Un esempio di risorsa Ingress:

```
apiVersion: networking.k8s.io/v1beta1
kind: Ingress
metadata:
  name: test-ingress
  annotations:
    nginx.ingress.kubernetes.io/rewrite-target: /
spec:
  rules:
  - http:
      paths:
      - path: /testpath
        pathType: Prefix
        backend:
          serviceName: test
          servicePort: 80
```

Ogni regola Ingress, contenuta nel campo **rules**, specifica:

1. **Host**, (opzionale) a cui vengono riferite le regole dell'Ingress.
2. **Path**, ad ognuno dei quali viene associato un backend con `serviceName` e `servicePort`.
3. **Backend**, combinazione di `serviceName` e `portName` a cui viene girata la richiesta in ingresso sul path specificato.

Tipologie di path (specificate in **pathType**):

1. **Implementation specific**, (default) dove il confronto tra path viene rimandato alla IngressClass di riferimento.

2. **Exact**, confronta il percorso URL in modo che coincida perfettamente con il contenuto di path (in case sensitivity).
3. **Prefix**, confronta il prefisso del percorso URL separato da / (in case sensitivity). Un esempio di confronto può essere il seguente: con il path specificato /foo/bar combacia /foo/bar/baz, ma non /foo/barbaz (Le substrings combaciano sempre). In caso di corrispondenze multiple con più paths specificati viene data la precedenza all'ingress con la sequenza più lunga che combacia.

Per utilizzare le risorse ingress ho bisogno di aggiungere al cluster il componente **Ingress Controller**.

Tipologie di ingress:

1. SINGLE SERVICE INGRESS, ad un punto di ingresso viene associato un solo servizio.
2. SIMPLE FANOUT, ad un punto di ingresso vengono associati più servizi, riconoscibili tramite il path.
3. NAME BASED VIRTUAL HOSTING, ad un punto di ingresso vengono associati più servizi, riconoscibili tramite l'hostname.

Inoltre è possibile integrare TLS all'interno degli ingress definendo una risorsa **Secret** che definisce chiavi e certificati.

6.1.1 Ingress Class

Le Ingress Class [23] sono risorse aggiuntive che per ogni Ingress definiscono a quale Ingress Controller fa riferimento (oltre ad altre configurazioni). Un esempio di Ingress Class è il seguente:

```
apiVersion: networking.k8s.io/v1beta1
kind: IngressClass
metadata:
```

```
name: external-lb
spec:
  controller: example.com/ingress-controller
  parameters:
    apiGroup: k8s.example.com/v1alpha
    kind: IngressParameters
    name: external-lb
```

Le Ingress Class possono essere marcate come **Default** all'interno del cluster con il flag `ingressclass.kubernetes.io/is-default-class` settato a true. Se vengono definite default più di una Ingress Class allora è necessario specificare in ognuna a quale ogni Ingress fa riferimento (con il campo `ingressClassName`).

6.1.2 Ingress Controller

A differenza di tutti gli altri controllers, installati automaticamente dal componente kube-controller-manager all'interno del cluster, gli Ingress Controller[24] devono essere aggiunti manualmente e sono NECESSARI se si vuole utilizzare una risorsa Ingress.

Attualmente, i controllers supportati e mantenuti dal progetto Kubernetes sono **GCE** e **ingress-nginx**, ma ne sono disponibili molti in base alle necessità, tra cui:

1. Application Gateway Ingress Controller (**AGIC**) è un Ingress Controller che permette di accedere dall'esterno ad un cluster AKS (Azure Kubernetes Service) utilizzando il nativo Azure Application Gateway (L7 LoadBalancer).
2. **Ambassador** API Gateway è un Envoy based Ingress Controller con supporto sia open-source che professionale.

3. **AWS ALB Ingress Controller** abilita gli Ingress utilizzando l'AWS Application Load Balancer.
4. **Contour** è un Ingress Controller basato su Envoy creato e supportato da VMWare.
5. **HAProxy Ingress** Ingress Controller altamente personalizzabile sviluppato dalla Community.
6. HAProxy Technologies fornisce una versione professionale del **HAProxy Ingress Controller**.
7. NGINX, Inc. fornisce il **NGINX Ingress Controller** per Kubernetes.

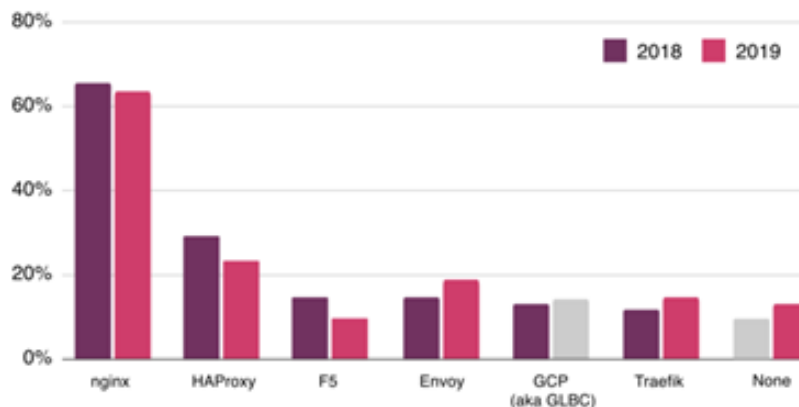


Figura 6.1: Grafico che riporta la classifica, aggiornata al 2019, degli Ingress Controller più utilizzati. Fonte CNCF Survey 2019[25].

Dalla classifica della Figura 6.1 emerge che la categoria nginx primeggia tra gli Ingress Controller scelti, questo perchè **ingress-nginx** è considerato il più semplice da utilizzare e il progetto è mantenuto direttamente dal Team di Kubernetes.

In realtà non è l'unico Ingress Controller che si basa su nginx, in quanto la NGINX Inc stessa ha prodotto il proprio NGINX Ingress Controller, sia in versione open-source che versione commerciale.

Al secondo posto troviamo gli HAProxy-Based Ingress Controllers, in quanto offrono ottime prestazioni e elevata personalizzazione delle configurazioni.

6.2 Integrazione di un Ingress all'interno del Cluster

Nel momento in cui si vuole utilizzare un Ingress è necessario dispiegare precedentemente un Ingress Controller. La problematica principale[26] che si pone per l'installazione di un Ingress Controller all'interno di un cluster non integrato in una infrastruttura Cloud è la mancanza di un Load Balancer.

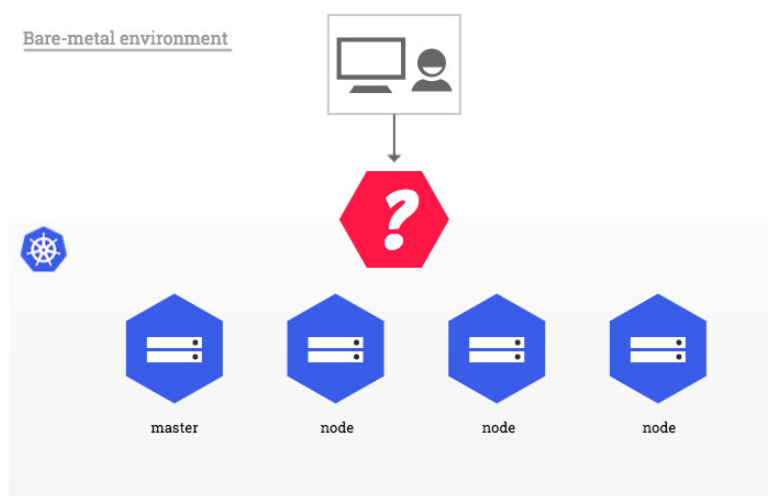


Figura 6.2: Rappresentazione grafica della problematica principale di un cluster bare-metal: Non esiste un componente (LoadBalancer) che permette di fare da tramite tra un utente che vuole effettuare una richiesta al cluster e i nodi del cluster stesso.

È stata introdotta una soluzione software per questo tipo di problema: **MetalLB**, software che assegna ad uno dei nodi del cluster la responsabilità

di svolgere il ruolo di Load Balancer.

Una volta applicato ad un cluster un Load Balancer è possibile integrare un Ingress Controller.

6.2.1 Installazione MetalLB

MetalLB[27] è un software che fornisce un'implementazione di un Load-Balancer di rete per un cluster Kubernetes che non viene eseguito su strutture cloud.

È particolarmente interessante la **Modalità Layer 2** di questo software, dove viene assegnata ad uno dei nodi (battezzato **Nodo Leader**) la possibilità di pubblicare un servizio in una rete locale (a livello di rete sembra che quella macchina abbia più indirizzi assegnati alla sua interfaccia). Il vantaggio di questa modalità è che non è richiesto un hardware specifico affinché funzioni, basta una qualsiasi rete ethernet.

Il nodo che viene scelto nella modalità Layer 2 si occupa di accettare tutte le richieste in arrivo per un tale IP/Servizio e, con l'aiuto del kube-proxy, distribuisce il traffico a tutti i pod disponibili.

Viene inoltre fornito un **meccanismo di failover** che, in automatico, si occupa di sostituire il nodo leader con un altro nel caso in cui questo non riesca a rispondere alle richieste. Il nodo secondario viene scelto da una *memberlist* di nodi del cluster che possono prendere il suo posto.

Questa modalità però presenta anche due importanti problematiche:

1. **Single-Node Bottlenecking.** Siccome solamente uno dei nodi riceve tutto il traffico in ingresso la larghezza di banda dell'Ingress è limitata dalla larghezza di banda del nodo stesso.
2. **Slow Failover.** Nel momento in cui entra in funzione il meccanismo di failover MetalLB invia pacchetti chiamati *gratuitos/unsolicited layer*

2 *packets* ai vari client per notificare che l'indirizzo MAC associato all'indirizzo IP del servizio che stanno richiedendo sta cambiando. La maggior parte dei sistemi operativi gestisce questo tipo di pacchetti in modo corretto e aggiorna velocemente la propria cache, mentre per alcuni sistemi questi pacchetti non vengono proprio gestiti oppure hanno delle implementazioni con problemi che comportano un ritardo nell'aggiornamento della cache del client.

Tutte le moderne versioni dei sistemi operativi principali (Linux, MacOS e Windows) implementano queste funzionalità correttamente, ma versioni più vecchie e/o sistemi operativi meno gettonati possono generare problemi.

Un'altra funzionalità esposta da MetalLB è la BGP mode (dove BGP sta per Border Gateway Protocol).

I requisiti per l'installazione[28] di MetalLB sono i seguenti:

- Un cluster Kubernetes, di versione superiore alle 1.13.0, che non ha già installato al suo interno un Load Balancer.
- Una configurazione di rete del cluster che può coesistere con MetalLB.
- Degli indirizzi IPv4 che MetalLB possa utilizzare.
- In base alla modalità per cui si vuole utilizzare MetalLB, potrebbe essere necessario uno o più router in grado di utilizzare BGP (MetalLB GBP Mode).

Se si sta utilizzando il kube-proxy in modalità **IPVS** (IP Virtual Server, che consente di effettuare bilanciamento di carico a livello di trasporto), dalla versione 1.14.2 di Kubernetes è necessario abilitare la strict ARP mode.

Nota: se si sta utilizzando kube-router come service-proxy, strict arp mode è abilitata di default quindi non è necessario abilitarla.

Per abilitare questa modalità basta eseguire il comando:

```
kubect1 edit configmap -n kube-system kube-proxy
```

e settare al suo interno le seguenti configurazioni:

```
apiVersion: kubeproxy.config.k8s.io/v1alpha1
kind: KubeProxyConfiguration
mode: "ipvs"
ipvs:
  strictARP: true
```

Infine, per installare MetalLB basta eseguire:

```
kubect1 apply -f https://raw.githubusercontent.com/metallb/
↳ metallb/v0.9.3/manifests/namespace.yaml
kubect1 apply -f https://raw.githubusercontent.com/metallb/
↳ metallb/v0.9.3/manifests/metallb.yaml
```

Se si tratta della prima installazione allora è necessario eseguire anche questo comando che imposta le chiavi segrete:

```
kubect1 create secret generic -n metallb-system memberlist
↳ --from-literal=secretkey="$(openssl rand -base64 128)"
```

Con questi comandi viene dispiegato MetalLB nel cluster, sotto il namespace **metallb-system**. Una volta entrati in funzione, si dovrebbe avere una situazione come in Figura6.3:

```
kubect1 get pods -n metallb-system
```

```
root@k8s-master:/home/anitvam# kubectl get pods -n metallb-system -o wide
NAME                                READY   STATUS    RESTARTS   AGE   IP              NODE
controller-fb659dc8-tk7zv           1/1    Running   0           23h   192.168.194.123 k8s-worker1
speaker-bv57p                       1/1    Running   0           23h   10.0.2.4         k8s-worker2
speaker-q6197                       1/1    Running   0           3h50m 10.0.2.5         k8s-master
speaker-vc2tw                       1/1    Running   0           23h   10.0.2.15        k8s-worker1
root@k8s-master:/home/anitvam#
```

Figura 6.3: Pods dispiegati nel momento in cui si integra MetalLB all'interno del Cluster.

Le componenti dispiegate da MetalLB, come visibile in Figura 6.3 sono:

- Il **metallb-system/controller**: componente che si occupa dell'assegnamento degli indirizzi IP.
- Il **metallb-system/speaker**: componente che si occupa di rendere accessibile il servizio sul protocollo scelto.
- **Service accounts** per il controller e lo speaker insieme a permessi RBAC (Role-Based Access Control) che questi componenti necessitano per funzionare.

Questa installazione però non include il file di configurazione[29], necessario per far funzionare MetalLB. Finché non viene aggiunto questo file, anche se MetalLB è funzionante, resta in fase idle, cioè aspetta per entrare effettivamente in azione.

La configurazione comprende la creazione e il dispiegamento di una **ConfigMap** nello stesso namespace in cui è dispiegato il deployment di MetalLB (nonché metallb-system).

Viene fornito un file di esempio che dà la possibilità di configurare MetalLB: <https://raw.githubusercontent.com/google/metallb/v0.9.3/manifests/example-config.yaml>

La configurazione di MetalLB in Layer 2 mode è la più semplice, perché funziona rispondendo direttamente a richieste ARP all'interno di una rete locale per dare i MAC address delle macchine ai client:

```
apiVersion: v1
kind: ConfigMap
metadata:
  namespace: metallb-system
  name: config
data:
  config: |
    address-pools:
    - name: default
      protocol: layer2
      addresses:
      - 10.0.2.240/28
```

In questo file deve essere esplicitata la modalità in cui si vuole che MetalLB entri in funzione e il pool di indirizzi che si vogliono assegnare per dare accesso al cluster.

Nota: Sono stati scelti indirizzi della stessa rete delle macchine che compongono il cluster in modo da potervi accedere dall'esterno del cluster nella stessa rete virtuale, ma questi indirizzi possono essere arbitrari.

Inserendo la nostra configurazione all'interno di un file, chiamato ad esempio **config.yaml**, possiamo direttamente procedere con la dispiegazione con il semplice comando:

```
Kubectl apply -f config.yaml
```

Il pool di indirizzi che viene assegnato a MetalLB deve essere riservato, non può essere quello appartenente alla rete dei nodi o quelli posseduti da un server DHCP.

6.2.2 Installazione ingress-nginx

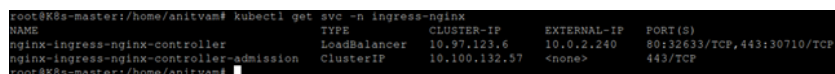
Una volta effettuata l'installazione di MetalLB si può procedere con l'installazione dell'ingress-nginx con i comandi:

```
helm repo add ingress-nginx
↪ https://kubernetes.github.io/ingress-nginx
helm install nginx ingress-nginx/ingress-nginx -n ingress-nginx
```

6.2.3 Accesso al Cluster dall'esterno

Una volta installato MetalLB e ingress-nginx all'interno del cluster, si dovrebbe avere una situazione come in Figura 6.4, digitando il comando sottostante:

```
kubectl get svc -n ingress-nginx
```



```
root@K8s-master:/home/anitvam# kubectl get svc -n ingress-nginx
NAME                                TYPE                CLUSTER-IP      EXTERNAL-IP      PORT(S)
nginx-ingress-nginx-controller      LoadBalancer        10.97.123.6     10.0.2.240       80:32633/TCP,443:30710/TCP
nginx-ingress-nginx-controller-admission ClusterIP            10.100.132.57  <none>           443/TCP
```

Figura 6.4: Servizi abilitati nel namespace ingress-nginx del Cluster. Da notare che per la prima volta è presente un indirizzo di tipo External.

dove il servizio di Ingress è attivo con l'external-ip dal valore 10.0.2.240, il quale rappresenterà il punto di accesso al nostro cluster dall'esterno.

Aggiungiamo ora un file di risorsa Ingress che ci permetta di definire un percorso valido per raggiungere uno dei nostri servizi esposti all'interno del cluster, chiamandolo **ingress.yaml**:

```
apiVersion: networking.k8s.io/v1
kind: Ingress
```

```
metadata:
  name: minimal-ingress
  annotations:
    nginx.ingress.kubernetes.io/rewrite-target: /
spec:
  rules:
  - http:
      paths:
      - path: /testpath
        pathType: Prefix
        backend:
          service:
            name: php-apache
            port:
              number: 80
```

Questo file semplicemente collega il percorso `/testpath` al servizio chiamato `php-apache`, precedentemente esposto nel nostro cluster. Quest'ultimo è un server web che risponde "OK!" ad ogni richiesta che riceve.

Ora dispiogliamo l'ingress con il comando:

```
kubectl apply -f ingress.yaml
```

Per testare che tutto funzioni, accendiamo una macchina virtuale esterna al cluster, cioè che non contiene nessuna delle sue configurazioni, ma la inseriamo nella sua stessa rete (rete virtuale messa a disposizione dall'hypervisor). Da questa macchina proviamo ora a connetterci all'indirizzo esposto dall'ingress, cioè `10.0.2.240` e al percorso specificato nel file di configurazione:

```
curl 10.0.2.240/testpath
```

Il risultato è quello aspettato: il servizio è effettivamente disponibile esternamente al cluster.

Conclusioni e Progetti Futuri

La tesi aveva come obiettivo elencare le informazioni necessarie per poter dispiegare un Cluster Kubernetes sui computer personali degli studenti del Corso di Ingegneria e Scienze Informatiche per conoscere e poter utilizzare Kubernetes anche con funzionalità estese.

Si è di fornire, passo dopo passo, tutte le nozioni per permettere di comprendere i software con cui si stava interagendo e indirizzare un eventuale utente a capire quali scelte effettuare per raggiungere il suo scopo.

Gli argomenti citati all'interno del documento sono tanti e anche complessi, per questo di alcuni si è cercato di dare solamente nozioni più generali sul loro funzionamento piuttosto che i dettagli; questo è stato fatto per poi espandere le conoscenze avendo già un punto di partenza consapevole della direzione che si sta prendendo e che si vuole prendere.

Sono possibili molteplici espansioni per quanto riguarda il cluster Kubernetes generato, tra cui:

- Integrare un VPA nel Cluster. Nel documento infatti si è approfondito in maggior parte il componente HPA, pertanto sarebbe interessante poter apprezzare i vantaggi di aumentare e diminuire in modo dinamico le risorse che un pod dispiegato può utilizzare.
- Approfondimento su ulteriori Ingress Controller. Quelli citati nel documento rappresentano solamente una minima parte di quello che in realtà è possibile utilizzare. Sicuramente sono presenti software con

funzionalità aggiuntive rispetto a quello da noi scelto che potrebbero essere studiati e integrati.

- Integrare un Cluster Autoscaler nel Cluster. Anche se ad oggi non possibile, non è escluso che ulteriori release del progetto Cluster Autoscaler non possano essere integrate in locale, magari attraverso il supporto di tool aggiuntivi come è stato fatto per gli Ingress Controller.
- Approfondimento su Prometheus. Sono state riportate le nozioni necessarie per utilizzare questo strumento, ma esistono molti altri aspetti che non sono stati toccati, come ad esempio gli Exporters, che sarebbe interessante implementare e personalizzare a piacere.
- MetalLB in Modalità BGP. Questa utilissima funzionalità di MetalLB è stata solamente citata, a vantaggio della Modalità Layer 2.
- Evoluzione nelle modalità di accesso al cluster. Nella nostra implementazione l'accesso al cluster è stato effettuato dalla stessa rete privata delle macchine virtuali; questa astrazione rappresenta certamente una limitazione, sarebbe interessante poter accedere ad un cluster anche da macchina host delle macchine virtuali guest sulle quali è situato il cluster, con opportune configurazioni di rete.
- Utilizzare gli Ingress per creare un hostname per il cluster. Nel nostro cluster, l'accesso è effettuato tramite indirizzo IP dell'Ingress Controller che è esposto verso l'esterno del cluster; con opportune configurazioni sarebbe possibile anche generare un proprio hostname assegnato al cluster.
- Sviluppo di applicazioni inter-cluster. Implementare un sistema che permetta di dispiegare repliche di una applicazione su cluster differenti, magari situati anche in reti differenti, che comunichino tra loro come se si trovassero logicamente sullo stesso.

Bibliografia

- [1] Kubernetes. Kubernetes api objects documentation, 2020. URL <https://kubernetes.io/docs/reference/generated/kubernetes-api/v1.18/>.
- [2] Kubernetes. Kubernetes network plugins, 2020. URL <https://kubernetes.io/docs/concepts/cluster-administration/networking/#how-to-implement-the-kubernetes-networking-model>.
- [3] Kubernetes. Kubernetes cri installation documentation, 2020. URL <https://kubernetes.io/docs/setup/production-environment/container-runtimes/>.
- [4] Kubernetes. Kubeadm installation guide, 2020. URL <https://kubernetes.io/docs/setup/production-environment/tools/kubeadm/install-kubeadm/>.
- [5] Kubernetes. Kubeadm init command documentation, 2020. URL <https://kubernetes.io/docs/reference/setup-tools/kubeadm/kubeadm-init>.
- [6] Kubernetes. Kubeadm join command documentation, 2020. URL <https://kubernetes.io/docs/reference/setup-tools/kubeadm/kubeadm-join/>.
- [7] Helm. Helm installation guide, 2020. URL <https://helm.sh/docs/intro/install/>.

-
- [8] Helm. Helm configuration guide, 2020. URL <https://helm.sh/docs/intro/quickstart/>.
 - [9] Kubernetes. Kubernetes vertical pod autoscaler documentation, 2020. URL <https://github.com/kubernetes/autoscaler/tree/master/vertical-pod-autoscaler>.
 - [10] Kubernetes. Kubernetes cluster autoscaler documentation, 2020. URL <https://github.com/kubernetes/autoscaler/tree/master/cluster-autoscaler#deployment>.
 - [11] Kubernetes. Kubernetes metrics server, 2020. URL <https://github.com/kubernetes-sigs/metrics-server>.
 - [12] Kubernetes. Kubernetes metrics server releases, 2020. URL <https://github.com/kubernetes-sigs/metrics-server/releases>.
 - [13] Kubernetes. Kubernetes metrics server official tutorial, 2020. URL <https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale-walkthrough/>.
 - [14] Prometheus. Prometheus official documentation, 2020. URL <https://prometheus.io/>.
 - [15] CoreOS. Prometheus operator official documentation, 2020. URL <https://coreos.com/operators/prometheus/docs/latest/user-guides/getting-started.html>.
 - [16] DirectXMan12. Prometheus adapter repository, 2020. URL <https://github.com/DirectXMan12/k8s-prometheus-adapter>.
 - [17] CoreOS. Kube prometheus official repository, 2020. URL <https://github.com/prometheus-operator/kube-prometheus>.
 - [18] Prmetheus Community. Prometheus community helm repository, 2020. URL <https://github.com/prometheus-community/helm-charts>.

-
- [19] Prometheus Community. Kube prometheus stack helm repository, 2020. URL <https://github.com/prometheus-operator/prometheus-operator>.
- [20] DirectXMan12. Prometheus adapter walkthrough, 2020. URL <https://github.com/DirectXMan12/k8s-prometheus-adapter/blob/master/docs/walkthrough.md>.
- [21] Helm Community. Prometheus operator label configuration, 2020. URL <https://coreos.com/operators/prometheus/docs/latest/user-guides/getting-started.html>.
- [22] Kubernetes. Kubernetes ingress api object reference, 2020. URL <https://kubernetes.io/docs/concepts/services-networking/ingress/#ingress-class>.
- [23] Kubernetes. Kubernetes ingress class reference, 2020. URL <https://kubernetes.io/docs/concepts/services-networking/ingress/#ingress-class>.
- [24] Kubernetes. Kubernetes ingress controller reference, 2020. URL <https://kubernetes.io/docs/concepts/services-networking/ingress-controllers/>.
- [25] Cloud Native Computer Foundation. Cncf survey 2019, 2020. URL https://www.cncf.io/wp-content/uploads/2020/03/CNCF_Survey_Report.pdf.
- [26] NGINX Inc. Deploy an ingress controller in a bare-metal cluster, 2020. URL <https://kubernetes.github.io/ingress-nginx/deploy/baremetal/>.
- [27] MetalLB. Metallb documentation, 2020. URL <https://metallb.universe.tf/concepts/>.

- [28] MetalLB. Metallb installation guide, 2020. URL <https://metallb.universe.tf/installation/>.
- [29] MetalLB. Metallb configuration guide, 2020. URL <https://metallb.universe.tf/configuration/>.
- [30] CoreOS. Prometheus operator official repository, 2020. URL <https://github.com/prometheus-operator/prometheus-operator>.