

Esame di Virtualizzazione

1. Descrizione del progetto

1.1 Descrizione ed obiettivo

L'obiettivo di questo progetto è quello di creare un cluster kubernetes production ready, scalabile orizzontalmente e con high availability (HA). Questo progetto si occupa solamente di dare una soluzione pronta all'uso per la creazione del solo cluster kubernetes, senza quindi occuparsi di ulteriori aspetti sistemistici quali configurazione del Firewall, gestione della DMZ, gestione dei certificati SSL, eccetera. Il cluster che si vuole creare è un cluster che risolva il problema dato da un scenario molto diffuso in ambito aziendale ovvero la gestione in container di un sistema client server sviluppata a micro servizi che sfrutti al meglio le funzionalità che kubernetes offre mantenendo comunque semplice la gestione del cluster.

L'applicativo in questione avrà 3 micro servizi:

- applicazione client
- applicazione server
- database

Essendo lo scopo di questo progetto un'architettura high available, saranno presenti più nodi control plane, più worker node e almeno 2 load balancer per accedere ai control plane ed ai worker node.

Per lo sviluppo di questo progetto sono state usate solamente tecnologie Open Source

1.2 Contesto - Scenario Real-World

Si immagina che l'azienda A abbia sviluppato un'applicazione web utilizzando il pattern monolitico, una situazione piuttosto diffusa, considerando che lo sviluppo orientato ai microservizi è una tendenza relativamente recente. L'applicazione in questione era inizialmente ospitata su un singolo server VPS.

Con il passare del tempo, l'applicazione si è arricchita di funzionalità moderne, come le comunicazioni in tempo reale, e ha visto crescere in modo esponenziale il numero di utenti in breve tempo, grazie ad alcune intuizioni vincenti da parte dell'azienda. Quello appena descritto sembrerebbe uno scenario ideale, se non fosse per il fatto che la crescita continua degli utenti ha finito per sovraccaricare il server VPS, che non riesce più a reggere il carico. L'azienda decide quindi di acquistare un server più potente, operazione che però comporta diverse ore, giorni o addirittura settimane di migrazione, durante le quali il vecchio server offre un servizio lento o addirittura assente.

Dopo la migrazione, eseguita con successo, può comunque verificarsi un guasto sul nuovo server, oppure un errore umano può compromettere l'ambiente di produzione, portando persino alla perdita dei dati contenuti nel database.

I principali problemi di scenari come questo sono tre:

- Il singolo punto di fallimento: se la macchina di produzione ha un guasto, il servizio diventa indisponibile.
- La non scalabilità: al crescere degli utenti, l'unica soluzione è acquistare server sempre più potenti, rischiando però disservizi tra una migrazione e l'altra.

- Il classico problema “it worked on my machine”: non è garantito che l’ambiente di sviluppo sia identico a quello di produzione, causando possibili malfunzionamenti.

Una soluzione a tutti questi problemi è offerta dalla containerizzazione (con tecnologie come containerd e Docker) e da Kubernetes, che consente la gestione di applicazioni containerizzate. Grazie alla containerizzazione, è possibile evitare il problema del “it worked on my machine”, sviluppando e testando il software localmente sulla stessa immagine usata in produzione.

Kubernetes permette invece di superare i primi due problemi: consente la creazione di ambienti high availability (senza singoli punti di fallimento) e offre un’ottima scalabilità, soprattutto orizzontale, rendendo molto semplice l’aggiunta di nuovi server al cluster di produzione distribuendo il carico di lavoro.

1.3 Challenge affrontate nel progetto

1. Lo sviluppo di applicazioni orientati ai micro servizi
2. La containerization dei vari applicativi con relativa gestione in cloud tramite un docker hub
3. La creazione di un cluster kubernetes con high availability
4. L'horizontal pod autoscaling dei micro servizi
5. La gestione di un DB Mysql con high availability (quindi la gestione dello storage condiviso in più nodi in modo da garantire le funzionalità anche in caso di spegnimento di un nodo)
6. Rendere Client e Server disponibile per gli utenti tramite distinti nomi DNS.

1.4 Descrizione dell'applicazione containerizzata

L'applicazione containerizzata consiste in:

- applicativo client: Applicazione lato client sviluppata con [React JS](#), utilizzo di [Nginx](#) come HTTP server.
- applicativo server: Applicazione server sviluppata con [Node JS](#) utilizzando il framework [Express JS](#).
- database: Database [Mysql](#), utilizzo di [Percona operator per Mysql](#) per rendere Mysql High Available e distribuito su più nodi.
- load balancer: Utilizzo del load balancer [HAProxy](#) insieme all'applicativo [Keepalived](#) per gestire la ridondanza sui load balancer
- strumento di virtualizzazione: Per creare il cluster e renderlo il più simile possibile ad un ambiente di produzione si fa uso di macchine linux ubuntu server create tramite il software [LXD](#)

1.5 Strumenti per il testing

Per testare la web app sarà sufficiente testare il corretto funzionamento collegandosi dal browser all'URL del client (default client.local, importante aggiungere l'host nel file [/etc/hosts](#), facendolo puntare al Virtual IP del load balancer).

Per testare l'horizontal pod autoscaling verrà utilizzato il software [Locust](#) perfetto per load e stress testing. Per testare invece la ridondanza tra i server e quindi il corretto funzionamento dell'high availability cluster basterà spegnere e riaccendere i nodi del cluster in momenti pseudo-randomici.

Per testare il Database è inoltre presente un Deploy dell'immagine di [PHP My Admin](#) (usando phpmyadmin.local come ingress).

2. Infrastruttura

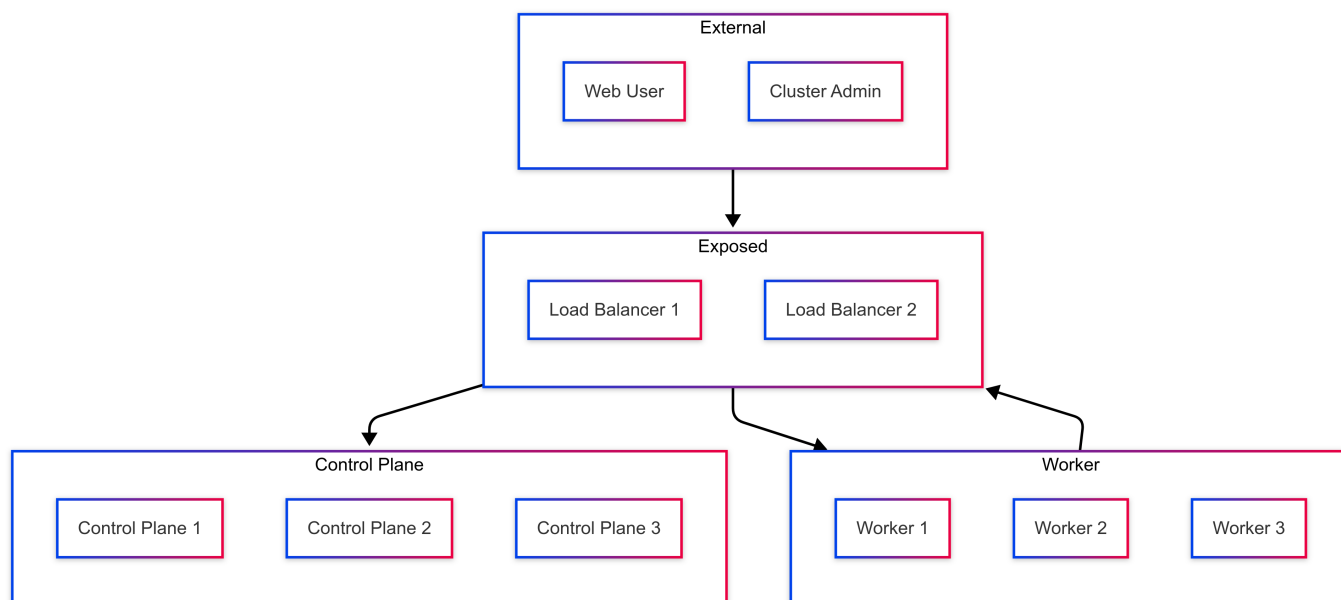
2.1 Descrizione del Cluster

Per questo progetto è stato scelto di realizzare un cluster kubernetes con high availability quindi con ridondanza di macchine server e con la totale assenza di single point of failure.

Il cluster è quindi composto da:

- 3 Control Plane: come descritto nella documentazione di Kubernetes, servono almeno 3 nodi control plane per creare un HA cluster. Per avere quorum nel sistema di consenso (etcd) e garantire la disponibilità anche in caso di guasto di un nodo.
- 3 Worker Node: sono stati scelti 3 nodi worker per garantire risorse sufficienti all'esecuzione dei carichi applicativi e per rispettare i requisiti minimi del Percona Operator, che richiede almeno 3 repliche per il corretto funzionamento del cluster database in modalità HA.
- 2 Load Balancer: vengono utilizzati per fornire un punto di accesso unificato al cluster, distribuendo il traffico verso i 3 nodi control plane e bilanciando gli accessi agli ingress delle web app tra i nodi worker. La presenza di due istanze evita un single point of failure; tramite Keepalived viene gestito un IP virtuale condiviso, che garantisce continuità di accesso al cluster anche in caso di guasto di uno dei due load balancer.

2.2 Diagramma del cluster



3. Implementazione

3.1 Sviluppo di applicazioni orientate ai micro servizi

Per utilizzare al meglio kubernetes e le sue funzionalità è preferibile sviluppare applicazione orientate ai micro servizi, per ottenere questo risultato ho creato 2 progetti distinti per client e server (in modo da poterli scalare singolarmente).

Inoltre, sempre per poter sfruttare al meglio lo scaling e la possibilità di distribuire i pod di una stesso micro servizio su nodi diversi, ho sviluppato sia l'applicativo client che l'applicativo stateless per non avere problemi di session in memoria o di storage condivisi (delegandoli a MySQL).

Per quanto riguarda MySQL, invece, ho scelto di utilizzare il Percona Operator per MySQL che gestisce automaticamente la creazione di un cluster database altamente disponibile, con replica sincrona e failover automatico, facilitando così la gestione e la resilienza del database all'interno dell'ambiente Kubernetes.

3.2 Containerizzazione dei micro servizi

Ogni micro servizio ha la sua immagine creata tramite DockerFile che esegue la build del micro servizio e poi lo espone tramite una porta.

Queste immagini vengono successivamente pushate sul docker hub (un docker hub pubblico in questo caso) per poi essere pullate da kubernetes.

Entrando nella cartella di ogni progetto è possibile trovare il DockerFile contenente le istruzioni per creare l'immagine del container.

Applicativo Client: L'applicativo client è un applicativo realizzato tramite react JS, e servito tramite nginx, il suo docker file parte da un'immagine di alpine linux con node js preinstallato per eseguire il building dell'applicazione, utilizza successivamente un'ulteriore immagine di alpine linux con all'interno nginx già pronto per l'uso per esporre il server http client all'esterno.

Applicativo Server: L'applicativo server è invece un web server http realizzato tramite node JS e la libreria express JS, il suo docker file parte da un'immagine di alpine linux con node js preinstallato, ne installa le dipendenze e fa partire l'applicativo

3.3 Creazione del cluster kubernetes con High Availability

Come CNI del cluster è stato utilizzato [Flannel](#).

Per quanto riguarda la containerizzazione, è stato invece utilizzato [Containerd](#), lo stesso runtime su cui si basa anche Docker.

Per creare un cluster Kubernetes ad alta disponibilità (high availability) è necessario avere almeno tre nodi control plane. Questo permette al control plane di continuare a funzionare anche in caso di guasto di uno dei nodi. Ciò è possibile grazie al sistema di consenso su cui si basa la sincronizzazione dei nodi, ovvero etcd, che richiede il raggiungimento di un quorum (la maggioranza) per garantire consistenza e disponibilità. Con tre nodi, il cluster può tollerare la perdita di uno di essi, mantenendo il controllo operativo senza rischio di inconsistenze o interruzioni.

I load balancer vengono configurati per distribuire il traffico verso i tre API server presenti sui nodi control plane. Inoltre, si occupano del bilanciamento delle richieste HTTP e HTTPS (sulle porte 80 e 443), inoltrandole ai servizi NGINX Ingress presenti sui nodi worker.

3.4 Horizontal pod autoscaling

Viene utilizzato l'Horizontal Pod Autoscaling (HPA) per monitorare dinamicamente il carico di lavoro sull'applicazione server e adattare automaticamente il numero di pod in esecuzione. In questo modo, il sistema è in grado di scalare orizzontalmente le risorse in base al volume delle richieste ricevute, garantendo prestazioni ottimali anche in caso di picchi di traffico e ottimizzando l'utilizzo delle risorse del cluster. Per rendere possibile ciò è stato inoltre installato il [metric server](#).

3.5 High Availability con Mysql

Per garantire HA con MySQL ho utilizzato il Percona Operator per MySQL, una soluzione che permette la creazione di un cluster MySQL distribuito e resiliente. Il Percona Operator automatizza la creazione e il failover del database, assicurando che i dati siano sempre sincronizzati tra i nodi e che il sistema continui a funzionare anche in caso di guasto di uno o più nodi. Per lo storage persistente ho scelto [Rancher](#) come storage provisioner, che fornisce volumi dinamici.

3.6 Creazione degli ingress

Per permettere l'accesso alle applicazioni client e server tramite domini personalizzati, ho configurato gli Ingress nel cluster Kubernetes. Utilizzando il controller [NGINX Ingress](#), ho creato regole che instradano il traffico verso i rispettivi servizi in base ai domini `client.local` e `server.local`.

4 Simulazione

4.1 Setup macchine virtuali

```
[mattia@mattia-B0D-WXX9]--[~/Projects/kubernetes]
[~ master]->> lxc start k8s-lb-2 k8s-master-1 k8s-master-2 k8s-master-3 k8s-worker-1 k8s-worker-2 k8s-lb-1 k8s-worker-3
[mattia@mattia-B0D-WXX9]--[~/Projects/kubernetes]
[~ master]->> lxc list
```

NAME	STATE	IPV4	IPV6	TYPE	SNAPSHOTS
base-ubuntu	STOPPED			VIRTUAL-MACHINE	0
k8s-lb-1	RUNNING	10.196.35.25 (enp5s0)	fd42:5ae:7c59:1137:216:3eff:fe27:c6ed (enp5s0)	VIRTUAL-MACHINE	0
k8s-lb-2	RUNNING	10.196.35.30 (enp5s0) 10.196.35.26 (enp5s0)	fd42:5ae:7c59:1137:216:3eff:fe4e:ef03 (enp5s0)	VIRTUAL-MACHINE	0
k8s-master-1	RUNNING	10.196.35.20 (enp5s0)	fd42:5ae:7c59:1137:216:3eff:fe95:4b18 (enp5s0)	VIRTUAL-MACHINE	0
k8s-master-2	RUNNING	10.196.35.21 (enp5s0)	fd42:5ae:7c59:1137:216:3eff:fea3:5171 (enp5s0)	VIRTUAL-MACHINE	0
k8s-master-3	RUNNING	10.196.35.22 (enp5s0)	fd42:5ae:7c59:1137:216:3eff:fe4e:9b2b (enp5s0)	VIRTUAL-MACHINE	0
k8s-worker-1	RUNNING	10.196.35.23 (enp5s0)	fd42:5ae:7c59:1137:216:3eff:fe46:863a (enp5s0)	VIRTUAL-MACHINE	0
k8s-worker-2	RUNNING	10.196.35.24 (enp5s0)	fd42:5ae:7c59:1137:216:3eff:fe79:c953 (enp5s0)	VIRTUAL-MACHINE	0
k8s-worker-3	RUNNING	10.196.35.27 (enp5s0)	fd42:5ae:7c59:1137:216:3eff:fe20:c25a (enp5s0)	VIRTUAL-MACHINE	0

Per eseguire questa simulazione verranno impiegate 8 macchine virtuali così distribuite:

- 2 load balance (`k8s-lb-1`, `k8s-lb-2`)
- 3 kubernetes control plane (`k8s-master-1`, `k8s-master-2`, `k8s-master-3`)
- 3 kubernetes worker (`k8s-worker-1`, `k8s-worker-2`, `k8s-worker-3`)

Inoltre ad ognuna delle macchine virtuali è stato assegnato un indirizzo IP fisso (che corrisponde a quello mostrato in figura).

Su ogni macchina è stato configurato un server ssh e sono stati installati i programmi necessari per il cluster kubernetes quali:

- kubeadm, kubectl, kubelet
- containerd (per la gestione dei container)

Si può inoltre notare che il nodo `k8s-lb-1` ha 2 indirizzi IP configurati sulla stessa interfaccia di rete in quanto è la macchina che di default contiene l'indirizzo IP locale assegnatogli da keepalived ovvero l'indirizzo IP `10.196.35.30`.

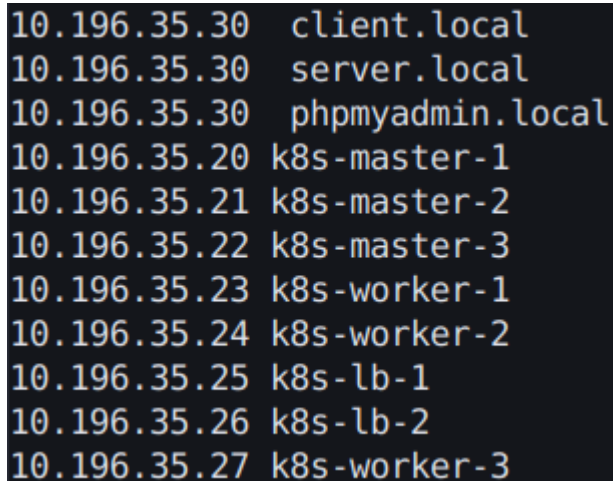
Questo indirizzo IP sarà di particolare importanza in quanto è quello indicato come `control-plane-endpoint` al momento della creazione del cluster, il nodo `k8s-lb-1` ogni volta che gli arriva una richiesta sulla port 6443 (porta di default del kubernetes api server) la reindirizzerà ad uno dei nodi control plane tramite un server haproxy appositamente configurato (vedere la cartella `load_balancer` per accedere a degli esempi dei file di configurazione di keepalived e di haproxy usati su `lb-1` e `lb-2`).

Per quanto riguarda il CNI è stato usato Flannel, in quanto è un plugin semplice da configurare e ben integrato con Kubernetes, che non richiede configurazioni di rete complesse.

Il cluster è stato creato tramite il seguente comando eseguito sul nodo `k8s-master-1` (usando come `pod-network-cidr` quello di default di Flannel)

```
kubeadm init --control-plane-endpoint "10.196.35.30:6443" \
--upload-certs \
--pod-network-cidr=10.244.0.0/16
```

Per rendere più semplice l'interazione con il cluster ed il successivo testing degli Ingress di kubernetes ho assegnato ad ogni host un nome usando, in ambiente debian, il file `/etc/hosts` come mostrato nella seguente immagine.



```
10.196.35.30 client.local
10.196.35.30 server.local
10.196.35.30 phpmyadmin.local
10.196.35.20 k8s-master-1
10.196.35.21 k8s-master-2
10.196.35.22 k8s-master-3
10.196.35.23 k8s-worker-1
10.196.35.24 k8s-worker-2
10.196.35.25 k8s-lb-1
10.196.35.26 k8s-lb-2
10.196.35.27 k8s-worker-3
```

4.2 Setup del cluster

Una volta creato il cluster, è stato subito aggiunto Flannel seguendo la documentazione ufficiale di Github ([Deploying Flannel with kubectl](#)).

Successivamente alla creazione del cluster è stato utilizzato il comando `kubeadm join` (fornito dall'output del comando `kubeadm init`) sui vari nodi per aggiungere control plane e worker node.

Per permettere il funzionamento degli Ingress e dell'horizontal pod autoscaler ho applicato i necessari manifesti .yaml reperibili dalle documentazioni ufficiali dei servizi necessari ([metric server](#), [nginx ingress](#)) (situati nella cartella `k8s/cluster-only`).

Per quanto riguarda il database viene utilizzato il Percona Operator per Mysql, bisogna quindi applicare correttamente i relativi manifesti .yaml reperibile dalla [guida ufficiale](#) (situati nella cartella `k8s/percona-operator`, i manifesti verranno applicati sotto il namespace `percona-operator`).

Infine rimangono da applicare i manifesti dei servizi client, server e phpMyAdmin (non possiede HA è solo a scopo di test) situati nella cartella `k8s/default`

La seguente immagine mostra gli output dei comandi una volta che il cluster è stato correttamente

configurato

```
[mattia@mattia-B00-WXX9]~$ kubectl get nodes
NAME                STATUS    AGE       VERSION
k8s-master-1        Ready    12d       v1.33.1
k8s-master-2        Ready    12d       v1.33.1
k8s-master-3        Ready    12d       v1.33.1
k8s-worker-1        Ready    12d       v1.33.1
k8s-worker-2        Ready    12d       v1.33.1
k8s-worker-3        Ready    2d3h      v1.33.1

[mattia@mattia-B00-WXX9]~$ kubectl get pods -o wide
NAME                READY    STATUS    RESTARTS   AGE    IP                NODE                NOMINATED NODE    READINESS GATES
client-85cbdd96d7-4nkk  1/1      Running   3 (17m ago)  152m   10.244.4.159      k8s-worker-1        <none>             <none>
client-85cbdd96d7-cxftx  1/1      Running   3 (17m ago)  152m   10.244.3.78       k8s-worker-3        <none>             <none>
client-85cbdd96d7-ms52s  1/1      Running   3 (17m ago)  152m   10.244.5.135      k8s-worker-2        <none>             <none>
phpmyadmin-78c4cbc65b-kwfn  1/1      Running   5 (17m ago)  2d2h   10.244.3.80       k8s-worker-3        <none>             <none>
server-789bb85d7-9x4v2    1/1      Running   0           36s    10.244.4.168      k8s-worker-1        <none>             <none>
server-789bb85d7-b2gkm    1/1      Running   0           2m57s  10.244.4.167      k8s-worker-1        <none>             <none>
server-789bb85d7-fzmc9    1/1      Running   0           110s   10.244.3.83       k8s-worker-3        <none>             <none>

[mattia@mattia-B00-WXX9]~$ kubectl get pods -o wide -n ingress-nginx
NAME                READY    STATUS    RESTARTS   AGE    IP                NODE                NOMINATED NODE    READINESS GATES
ingress-nginx-controller-78bdd77b9-5zlv  1/1      Running   3 (17m ago)  145m   10.244.4.162      k8s-worker-1        <none>             <none>
ingress-nginx-controller-78bdd77b9-qt6nc  1/1      Running   3 (17m ago)  145m   10.244.3.76       k8s-worker-3        <none>             <none>
ingress-nginx-controller-78bdd77b9-zzmqf  1/1      Running   3 (17m ago)  145m   10.244.5.138      k8s-worker-2        <none>             <none>

[mattia@mattia-B00-WXX9]~$ kubectl get pods -o wide -n percona-operator
NAME                READY    STATUS    RESTARTS   AGE    IP                NODE                NOMINATED NODE    READINESS GATES
cluster1-haproxy-0  2/2      Running   11 (18m ago)  2d2h   10.244.3.75       k8s-worker-3        <none>             <none>
cluster1-haproxy-1  2/2      Running   6 (18m ago)  171m   10.244.5.134      k8s-worker-2        <none>             <none>
cluster1-haproxy-2  2/2      Running   8 (18m ago)  2d2h   10.244.4.164      k8s-worker-1        <none>             <none>
cluster1-pxc-0      3/3      Running   14 (16m ago)  178m   10.244.5.137      k8s-worker-2        <none>             <none>
cluster1-pxc-1      3/3      Running   17 (14m ago)  2d2h   10.244.3.79       k8s-worker-3        <none>             <none>
cluster1-pxc-2      3/3      Running   18 (14m ago)  2d2h   10.244.4.158      k8s-worker-1        <none>             <none>
percona-xtradb-cluster-operator-df7b749b5-mgw5s  1/1      Running   4 (18m ago)  2d2h   10.244.4.165      k8s-worker-1        <none>             <none>
```

4.3 Descrizione dei manifesti

Il sistema è composto da tre principali componenti deployati in Kubernetes: client, server, e phpMyAdmin, ciascuno definito tramite Deployment, Service e Ingress. Il client è una web app con 3 repliche e policy anti-affinità per distribuirne i pod tra i nodi. È accessibile tramite l'host client.local. Il server, anch'esso con 3 repliche e anti-affinità, espone un'API sulla porta 3000, configurata tramite ConfigMap e Secret, e include probe di liveness e readiness. È bilanciato dinamicamente tramite un HorizontalPodAutoscaler (HPA) tra 3 e 9 repliche, in base all'utilizzo di CPU e memoria. L'interfaccia phpMyAdmin consente l'accesso al database e viene esposta su phpmyadmin.local. La configurazione dell'ambiente e delle credenziali è separata tramite oggetti ConfigMap e Secret, garantendo modularità e sicurezza. Tutti i servizi interni usano ClusterIP, e l'accesso esterno è gestito da un Ingress NGINX.

Per far funzionare correttamente gli ingress NGINX i 2 load balancer faranno un load balancing tra i 3 worker node servendo alla porta 80 del load balancer le porte esposte dal service dell'ingress, quindi i servizi web saranno accessibili tramite i load balancer e quindi tramite l'IP virtuale sopra descritto.

4.4 Test del cluster

Per testare il cluster ci sono diversi modi, il migliore è accedere all'url <http://client.local>, creare un account ed accedere alla dashboard, questo permette di controllare il corretto funzionamento di Ingress, client, server e database, ricoprendo completamente lo use case di questa applicazione.

Per semplicità in questa simulazione testeremo la correttezza del cluster tramite un route del server che accede al Database, questo ci permette di testare Ingress, server e database che ricoprono una buona parte degli elementi da testare e rendere facile e veloce avere un riscontro visivo del funzionamento corretto del cluster.

Testiamo quindi che il cluster funzioni

```
[mattia@mattia-B00-WXX9]~$ curl -s http://server.local/ready
Ready!
```

4.5 Test HA sul load balancer

Ora testeremo il corretto funzionamento dell'high availability su varie parti del cluster spegnendo a runtime alcune macchine virtuali.

Il primo test lo faremo sui load balancer: spegneremo il load balancer 1 che è quello che attualmente detiene

l'IP virtuale e vedremo se il load balancer 2 prenderà correttamente l'ip virtuale.

```
[mattia@mattia-B0D-WXX9]--[~/Projects/kubernetes] (bas
[mattia@mattia-B0D-WXX9]-->> lxc stop k8s-lb-1
[mattia@mattia-B0D-WXX9]--[~/Projects/kubernetes] (bas
[mattia@mattia-B0D-WXX9]-->> lxc list
```

NAME	STATE	IPV4	IPV6	TYPE	SNAPSHOTS
base-ubuntu	STOPPED			VIRTUAL-MACHINE	0
k8s-lb-1	STOPPED			VIRTUAL-MACHINE	0
k8s-lb-2	RUNNING	10.196.35.30 (enp5s0) 10.196.35.26 (enp5s0)	fd42:5ae:7c59:1137:216:3eff:fe4e:ef03 (enp5s0)	VIRTUAL-MACHINE	0
k8s-master-1	RUNNING	10.244.0.1 (cni0) 10.244.0.0 (flannel.1) 10.196.35.20 (enp5s0)	fd42:5ae:7c59:1137:216:3eff:fe95:4b18 (enp5s0)	VIRTUAL-MACHINE	0
k8s-master-2	RUNNING	10.244.1.0 (flannel.1) 10.196.35.21 (enp5s0)	fd42:5ae:7c59:1137:216:3eff:fea3:5171 (enp5s0)	VIRTUAL-MACHINE	0
k8s-master-3	RUNNING	10.244.2.0 (flannel.1) 10.196.35.22 (enp5s0)	fd42:5ae:7c59:1137:216:3eff:fe9e:9b2b (enp5s0)	VIRTUAL-MACHINE	0
k8s-worker-1	RUNNING	10.244.4.1 (cni0) 10.244.4.0 (flannel.1) 10.196.35.23 (enp5s0)	fd42:5ae:7c59:1137:216:3eff:fe96:863a (enp5s0)	VIRTUAL-MACHINE	0
k8s-worker-2	RUNNING	10.244.5.1 (cni0) 10.244.5.0 (flannel.1) 10.196.35.24 (enp5s0)	fd42:5ae:7c59:1137:216:3eff:fe79:c953 (enp5s0)	VIRTUAL-MACHINE	0
k8s-worker-3	RUNNING	10.244.3.1 (cni0) 10.244.3.0 (flannel.1) 10.196.35.27 (enp5s0)	fd42:5ae:7c59:1137:216:3eff:fe20:c25a (enp5s0)	VIRTUAL-MACHINE	0

```
[mattia@mattia-B0D-WXX9]--[~/Projects/kubernetes] (bas
[mattia@mattia-B0D-WXX9]-->> curl server.local/ready
Ready
```

Come dimostrato dall'immagine il load balancer 2 detiene correttamente l'ip virtuale 10.196.35.30 ed il cluster continua a funzionare correttamente.

4.6 Tets HA sul control plane

Ora proveremo a spegnere il nodo master 1 ovvero uno dei control plane e testeremo che il cluster continui a funzionare correttamente in quando l'etcd server dovrebbe continuare ad avere il quorum di 2 nodi su 3

votanti.

```
[mattia@mattia-B0D-WXX9]--[~/Projects/kubernetes] (base)
[mattia@mattia-B0D-WXX9]--> lxc list
```

NAME	STATE	IPV4	IPV6	TYPE	SNAPSHOTS
base-ubuntu	STOPPED			VIRTUAL-MACHINE	0
k8s-lb-1	STOPPED			VIRTUAL-MACHINE	0
k8s-lb-2	RUNNING	10.196.35.30 (enp5s0) 10.196.35.26 (enp5s0)	fd42:5ae:7c59:1137:216:3eff:fe4e:ef03 (enp5s0)	VIRTUAL-MACHINE	0
k8s-master-1	STOPPED			VIRTUAL-MACHINE	0
k8s-master-2	RUNNING	10.244.1.1 (cni0) 10.244.1.0 (flannel.1) 10.196.35.21 (enp5s0)	fd42:5ae:7c59:1137:216:3eff:fea3:5171 (enp5s0)	VIRTUAL-MACHINE	0
k8s-master-3	RUNNING	10.244.2.1 (cni0) 10.244.2.0 (flannel.1) 10.196.35.22 (enp5s0)	fd42:5ae:7c59:1137:216:3eff:fe4e:9b2b (enp5s0)	VIRTUAL-MACHINE	0
k8s-worker-1	RUNNING	10.244.4.1 (cni0) 10.244.4.0 (flannel.1) 10.196.35.23 (enp5s0)	fd42:5ae:7c59:1137:216:3eff:fe4e:863a (enp5s0)	VIRTUAL-MACHINE	0
k8s-worker-2	RUNNING	10.244.5.1 (cni0) 10.244.5.0 (flannel.1) 10.196.35.24 (enp5s0)	fd42:5ae:7c59:1137:216:3eff:fe79:c953 (enp5s0)	VIRTUAL-MACHINE	0
k8s-worker-3	RUNNING	10.244.3.1 (cni0) 10.244.3.0 (flannel.1) 10.196.35.27 (enp5s0)	fd42:5ae:7c59:1137:216:3eff:fe20:c25a (enp5s0)	VIRTUAL-MACHINE	0

```
[mattia@mattia-B0D-WXX9]--[~/Projects/kubernetes] (base)
[mattia@mattia-B0D-WXX9]--> curl server.local/ready
Ready
[mattia@mattia-B0D-WXX9]--[~/Projects/kubernetes] (base)
[mattia@mattia-B0D-WXX9]--> kubectl get nodes
```

NAME	STATUS	ROLES	AGE	VERSION
k8s-master-1	NotReady	control-plane	12d	v1.33.1
k8s-master-2	Ready	control-plane	12d	v1.33.1
k8s-master-3	Ready	control-plane	12d	v1.33.1
k8s-worker-1	Ready	<none>	12d	v1.33.1
k8s-worker-2	Ready	<none>	12d	v1.33.1
k8s-worker-3	Ready	<none>	12d	v1.33.1

4.7 Test HA sul un worker node

Ora proveremo a spegnere il nodo worker 1 ovvero un nodo worker e testeremo che il cluster continui a funzionare correttamente in quanto Percona XtraDB Cluster che utilizza il galera cluster (libreria di replicazione sincrona multimaster) continua ad avere il quorum di 2 nodi con Percona XtraDB Cluster attivi su

3.

```
[mattia@mattia-B0D-WXX9]--[~/Projects/kubernetes] (base 3.1)
[.] master ● ~6] ->> lxc stop k8s-worker-1
[mattia@mattia-B0D-WXX9]--[~/Projects/kubernetes] (base 3.1)
[.] master ● ~6] ->> lxc list
```

NAME	STATE	IPV4	IPV6	TYPE	SNAPSHOTS
base-ubuntu	STOPPED			VIRTUAL-MACHINE	0
k8s-lb-1	STOPPED			VIRTUAL-MACHINE	0
k8s-lb-2	RUNNING	10.196.35.30 (enp5s0) 10.196.35.26 (enp5s0)	fd42:5ae:7c59:1137:216:3eff:fe4e:ef03 (enp5s0)	VIRTUAL-MACHINE	0
k8s-master-1	STOPPED			VIRTUAL-MACHINE	0
k8s-master-2	RUNNING	10.244.1.1 (cni0) 10.244.1.0 (flannel.1) 10.196.35.21 (enp5s0)	fd42:5ae:7c59:1137:216:3eff:fea3:5171 (enp5s0)	VIRTUAL-MACHINE	0
k8s-master-3	RUNNING	10.244.2.1 (cni0) 10.244.2.0 (flannel.1) 10.196.35.22 (enp5s0)	fd42:5ae:7c59:1137:216:3eff:fe4e:9b2b (enp5s0)	VIRTUAL-MACHINE	0
k8s-worker-1	STOPPED			VIRTUAL-MACHINE	0
k8s-worker-2	RUNNING	10.244.5.1 (cni0) 10.244.5.0 (flannel.1) 10.196.35.24 (enp5s0)	fd42:5ae:7c59:1137:216:3eff:fe79:c953 (enp5s0)	VIRTUAL-MACHINE	0
k8s-worker-3	RUNNING	10.244.3.1 (cni0) 10.244.3.0 (flannel.1) 10.196.35.27 (enp5s0)	fd42:5ae:7c59:1137:216:3eff:fe20:c25a (enp5s0)	VIRTUAL-MACHINE	0

```
[mattia@mattia-B0D-WXX9]--[~/Projects/kubernetes] (base 3.1)
[.] master ● ~6] ->> curl server.local/ready
Ready
```

4.8 Test HA sul secondo control plane

Ora proveremo a spegnere il nodo master 3 ovvero un altro dei control plane, in questo caso il cluster resterà attivo ma in stato degradato in quanto l'etcd perde il quorum e non potrà più accettare modifiche allo stato del cluster ma i workload esistenti continueranno a girare.

```
[mattia@mattia-B0D-WXX9]--[~/Projects/kubernetes] (base 3.1)
[.] master ● ~6] ->> lxc stop k8s-master-3
[mattia@mattia-B0D-WXX9]--[~/Projects/kubernetes] (base 3.1)
[.] master ● ~6] ->> lxc list
```

NAME	STATE	IPV4	IPV6	TYPE	SNAPSHOTS
base-ubuntu	STOPPED			VIRTUAL-MACHINE	0
k8s-lb-1	STOPPED			VIRTUAL-MACHINE	0
k8s-lb-2	RUNNING	10.196.35.30 (enp5s0) 10.196.35.26 (enp5s0)	fd42:5ae:7c59:1137:216:3eff:fe4e:ef03 (enp5s0)	VIRTUAL-MACHINE	0
k8s-master-1	STOPPED			VIRTUAL-MACHINE	0
k8s-master-2	RUNNING	10.244.1.1 (cni0) 10.244.1.0 (flannel.1) 10.196.35.21 (enp5s0)	fd42:5ae:7c59:1137:216:3eff:fea3:5171 (enp5s0)	VIRTUAL-MACHINE	0
k8s-master-3	STOPPED			VIRTUAL-MACHINE	0
k8s-worker-1	STOPPED			VIRTUAL-MACHINE	0
k8s-worker-2	RUNNING	10.244.5.1 (cni0) 10.244.5.0 (flannel.1) 10.196.35.24 (enp5s0)	fd42:5ae:7c59:1137:216:3eff:fe79:c953 (enp5s0)	VIRTUAL-MACHINE	0
k8s-worker-3	RUNNING	10.244.3.1 (cni0) 10.244.3.0 (flannel.1) 10.196.35.27 (enp5s0)	fd42:5ae:7c59:1137:216:3eff:fe20:c25a (enp5s0)	VIRTUAL-MACHINE	0

```
[mattia@mattia-B0D-WXX9]--[~/Projects/kubernetes] (base 3.1)
[.] master ● ~6] ->> curl server.local/ready
Ready
```

4.9 Test horizontal pod autoscaling

La prossima simulazione è volta a testare il corretto funzionamento dell'horizontal pod autoscaler, in questo cluster l'hpa è applicato solamente sul server a scopo dimostrativo, inizialmente possiamo notare che se nessuno fa chiamate al server i suoi livelli di CPU e memoria sono bassi e quindi il numero di pod è al minimo

Every 2.0s: kubectl get hpa

mattia-B0D-WXX9: Mon Jun 9 11:33:05 2025

NAME	REFERENCE	TARGETS	MINPODS	MAXPODS	REPLICAS	AGE
server-hpa	Deployment/server	cpu: 1%/70%, memory: 48%/80%	3	9	3	7d17h

Iniziamo il testing facendo partire Locust e utilizzando la route stress sul server che aspetta 2 secondi e poi risponde.

Start new load test

Number of users (peak concurrency) *

300

Ramp up (users started/second) *

50

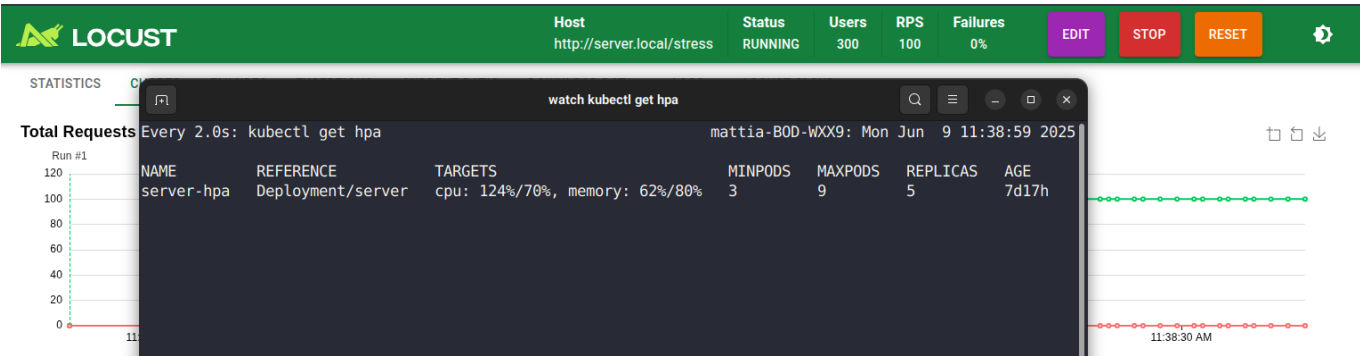
Host

http://server.local/stress

Advanced options

START

Possiamo subito notare che dopo un po' di tempo in cui locust è in esecuzione subito il numero di POD arriva a 5.



Continuiamo quindi il test aumento il numero di utenti attivi e quindi di richieste simultanee su locust

Start new load test

Number of users (peak concurrency) *

1000

Ramp up (users started/second) *

100

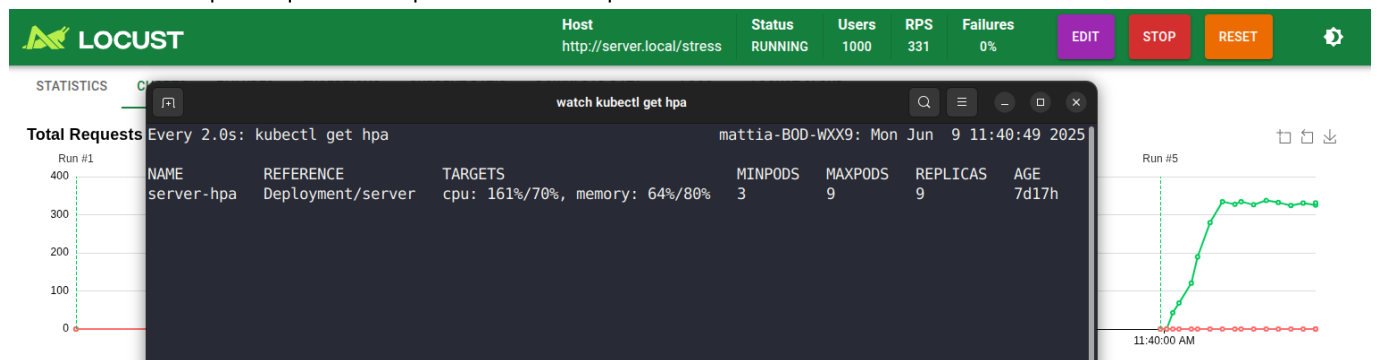
Host

http://server.local/stress

Advanced options

START

Notiamo che dopo un po' di tempo il numero di pod che kubernetes schedula sale subito al massimo



Una volta stoppato locust possiamo poi notare che lentamente il numero di pod cala fino a tornare al minimo di 3.

Abbiamo quindi verificato il corretto funzionamento dello scaling basato sulle risorse