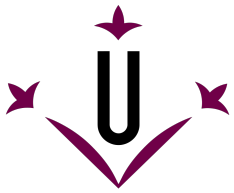

Universitat de Lleida

Microserveis, Docker i Kubernetes

Computació Distribuïda - Grau en Enginyeria Informàtica

Pablo Fraile Alonso

19 d'abril de 2022

Índex

Índex de figures

1 Microserveis

1.1 Què són els microserveis?

Segons IBM [2], els microserveis són una arquitectura de software, on una aplicació es troba composada per més aplicacions/serveis petits, dèbilment acoblats entre ells i desplegables independentment.
Aquests serveis tenen la seva pròpia pila de tecnologies i dades i es comuniquen amb altres microserveis a partir d'APIs REST, event streaming i brokers.

1.2 Per què van aparèixer els microserveis?

Segons OpusSoftwareTeam [10], tres causes han motivat l'aparició d'arquitectures basades en microserveis:

1. **La facilitat de dotar d'elasticitat dinàmica a l'aplicació**, ja que és una arquitectura on és molt fàcil ampliar/disminuir la quantitat de microserveis actius en funció de la demanda.

   A més, la facilitat d'assignar i alliberar recursos a partir del cloud computing [1] ha fet que la combinació entre el cloud i els microserveis sigui una molt bona opció per crear un producte escalable, on es redueix el cost de deployment (ja que únicament es paga pels recursos emprats).

2. **L'exigència actual de les empreses digitals de millorar constantment els seus sistemes**. En una aplicació monolítica gran pot ser molt frustrant recompilar tot el projecte i les seves dependències, a més, realitzar les proves funcionals de l'aplicació pot ser exhaustiu.

   Els microserveis permeten desglossar el problema i actualitzar i compilar només una funcionalitat canviada sense haver de treballar amb la resta de l'aplicació.

[1] Mitjançant màquines virtuals o containers

3. **La tolerància a errors**. En un sistema monolític, l'error d'una certa funcionalitat [2] generalment provoca que l'aplicació sencera s'aturi.

   En els microserveis, l'error d'una certa funcionalitat o microservei fa que aquesta deixi de funcionar, però no afecta a l'aplicació en conjunt, cosa que minimitza l'impacte d'error global.

[2] Com podria ser una excepció no capturada

2 Docker

Tal i com s'ha comentat a la secció 1, els microserveis haurien d'estar en un ambient aïllat i comunicar-se entre ells mitjançant algun protocol, com per exemple http, i seguint una estil arquitectural com REST.
Una forma d'aconseguir aquest aïllament entre serveis és a partir de **Docker**.

2.1 Què és Docker?

Segons Microsoft [6], Docker és un projecte de codi obert per automatitzar el desplegament d'aplicacions com a contenidors portàtils i autosuficients que es poden executar al núvol o en local.

En aquesta definició ens apareix un nou concepte: **contenidors**, per tant, per definir docker primer hem de definir què són els contenidors i com funcionen.

2.2 Què són els contenidors?

Segons Microsoft [7], un contenidor (container) és un paquet estàndard de programari, que agrupa el codi d'una aplicació, juntament amb els fitxers, configuració i dependències necessàries per a què l'aplicació s'executi.
Això permet als desenvolupadors i professionals de IT desplegar aplicacions independentment de l'entorn emprat.

Per tant, podem veure docker com una eina que ens permet crear contenidors del nostre projecte que puguin executar-se en tota màquina que tingui docker instal·lat (independentment dels paquets i versions que es tinguin localment al dispositiu).

2.3 Containers vs Virtual Machines

Després de la definició de la secció 2.2 al lector li pot sorgir el següent dubte: quina diferència hi ha entre un contenidor i una màquina virtual?

Una màquina virtual també ens permet aïllar el programari i les seves dependències del host que l'executa a més, normalment també es pot empa-

quetar i moure una màquina virtual d'un host a un altre.

Per saber les diferències entre les màquines virtuals i els contenidors, s'ha d'entendre com funcionen i els pros/contres del seu funcionament.

2.3.1 Arquitectura de les Màquines Virtuals

L'arquitectura de les màquines virtuals es pot apreciar en la figura 1. Es poden veure els següents components:

- El hardware del dispositiu.
- La capa del hypervisor.
- La màquina virtual (sistema operatiu + app).

La capa que fa possible la virtualització del sistema operatiu és el hypervisor. Aquest tradueix les instruccions del sistema operatiu de la VM al sistema operatiu del host [3].

Un cop carregada la capa del hypervisor, es continuarà amb tot el sistema operatiu que es vol virtualitzar. Finalment, s'executarà les aplicacions desitjades.

Aquesta arquitectura però té un cost d'inici molt gran ja que s'ha d'arrancar el hypervisor, el sistema operatiu i finalment les aplicacions.
Per tant, si es vol desplegar la imatge d'una màquina virtual en un altre dispositiu, aquest tardarà varis minuts en disposar de la capa d'aplicacions.

Una possible solució a aquest problema seria que cada màquina virtual contingués més d'una aplicació a executar, però llavors es tindria el problema de que no es pot escalar individualment cada component de la capa d'aplicació.

[3] Realment, no és la única tasca del hypervisor. Per més informació, consultar l'explicació del hypervisor KVM de redhat.

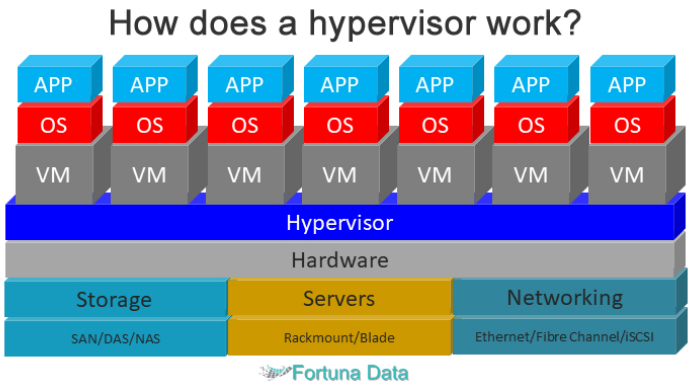

Figura 1: Data Storage UK: Arquitectura de les Màquines Virtuals

2.3.2 Arquitectura de Docker

Docker, en canvi, empra l'arquitectura mostrada a la figura 2 (la qual ja no té una capa de hypervisor, sinó que carrega el seu propi sistema operatiu) i, a continuació, executa un programa (docker engine) que suporta els diferents containers.

A la figura també es pot observar que **un container és únicament un conjunt de llibreries i programari necessaris per executar l'aplicació.**

A més, docker està utilitzant el sistema operatiu del host concretament el kernel i, per tant, no es tradueixen les instruccions d'un container al host, sinò que aquestes ja han de ser compatibles amb el sistema operatiu.

Això implica que si s'està emprant un sistema operatiu linux, els contenidors que s'executin al docker engine únicament podran ser executats si es troben pensats per a un sistema operatiu linux [4].

Aquesta arquitectura però, ens aporta grans beneficis a l'hora d'escalar les nostres aplicacions, ja que ara si tenim varies màquines corrent una instància del docker engine, podem executar un container amb un temps mínim, ja que únicament s'haurà d'executar la capa d'aplicacions i llibreries.

[4]De fet, va ser per aquest motiu que Windows va començar a crear el WSL (Windows Subsystem for Linux). Més informació aquí.

A més, com cada component/servei es troba encapsulat en un container, podem escalar individualment cada un d'aquests, creant containers d'un servei en funció de la demanda.

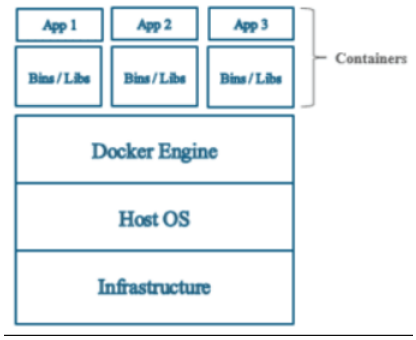

Figura 2: GeekFlare: Arquitectura de Docker

2.4 Docker images vs Docker containers

A la secció 2.3.2 s'ha vist com és l'arquitectura de docker envers les màquines virtuals però no s'ha explicat com es poden moure/transportar aquests "containers" per tal de que es puguin executar en qualsevol màquina que tingui el docker engine instal·lat i executant-se.

És aquí on apareixen el concepte de **docker images** i **docker containers** que, tot i que semblin conceptes molt similars, tenen objectius molt diferents.

- Una imatge de docker (docker image) és un "model" que únicament accepta lectura i que conté empaquetat totes les aplicacions i dependències d'una component/servei.
- Un container (docker container) és una instància d'una imatge, és a dir, és l'execució d'una imatge.

Aquestes definicions es veuen molt més clares si s'observen les figures 3 i 4 on es pot veure que una imatge és **immutable** i que conté totes les necessitats de l'aplicació mentre que un container afegeix una capa **mutable** on

es produeix l'execució de l'aplicació.

Aquest model implica que, en cas de que modifiquem la part mutable d'un container i es vulgui guardar aquests canvis, docker pot empaquetar aquestes capes com a una nova imatge immutable.

Cal recalcar que, a partir d'una imatge, es poden crear N containers diferents ja que les parts immutables seran les mateixes però la part mutable variarà en funció del container creat.

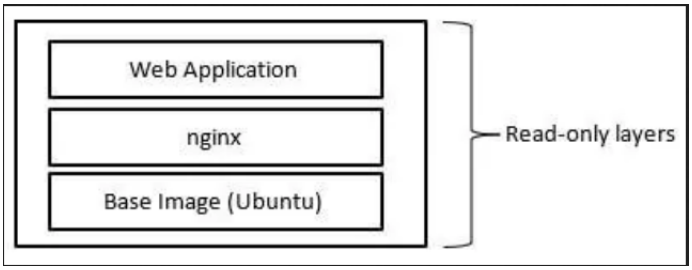

Figura 3: Docker images layers

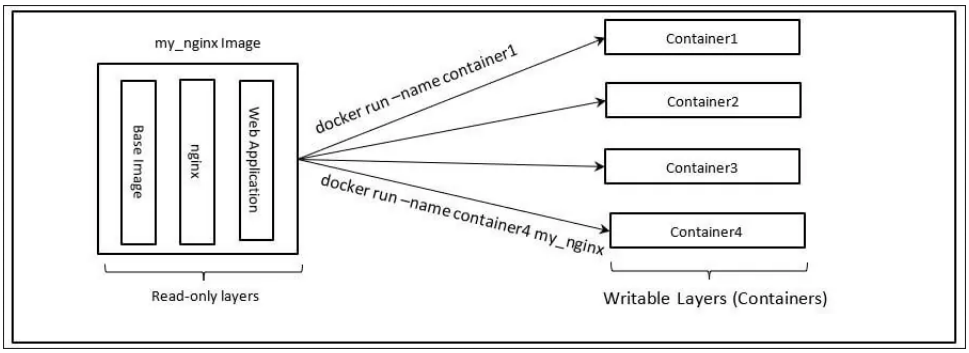

Figura 4: Docker containers layers

2.5 Crear una Docker image a partir d'un projecte

Tot i que Docker proporcioni comandes per tal de crear una imatge a partir d'un projecte, això acostuma a ser una opció molt tediosa i poc pràctica.

Docker també pot produir docker images automàticament llegint les instruccions d'un fitxer anomenat **Dockerfile**.

Un Dockerfile és un document de text que conté totes les comandes per tal de crear una imatge. Els usuaris poden crear l'imatge únicament executant:

```
docker build -t [nom_imatge] [directori_dockerfile]
```

Per més informació sobre la sintaxis del Dockerfile i les comandes per tal d'executar accions més complexes, es pot llegir el manual oficial de Dockerfile.

2.6 Docker compose

2.6.1 Execució de diversos contenidors sense docker compose

En les anteriors seccions s'ha vist com executar un contenidor i com aquest es troba en un context aïllat de la resta de contenidors i del sistema operatiu del host.
En producció però, les aplicacions web o arquitectures basades en microserveis no tenen únicament un servei que parla amb l'exterior, sinó que es comuniquen entre ells per tal d'elaborar una resposta. Per tant, **han de poder parlar entre ells a partir d'una xarxa**.

Amb docker, cada container té la seva pròpia interfície dins de la seva pròpia xarxa. Per tant, per defecte, un container no pot parlar amb un altre. Igualment, hi ha l'opció de crear una xarxa i, al crear els contenidors, assignar-los aquella xarxa per tenir el mateix adreçament:

```
Creació de la xarxa:
    $ docker network create mongo-network

Creació del container de la base de dades:
    $ docker run -d \
                 -p 27018:27018 \
                 --net mongo-network
                 --name mongo-container
                 mongo

Creació del container del servidor:
```

```
$ docker run -d \
             -p 443:443 \
             --net mongo-network \
             --name express-container \
             mongo-express
```

Com podem veure, amb aquestes comandes s'han creat dos containers (un per a la base de dades i l'altre per al servidor), que es troben a la mateixa xarxa i poden parlar entre sí.

Aquesta solució però, és bastant llarga d'escriure i, per tant, és fàcil equivocar-se. A més, faltaria configurar les diferents variables d'entorn tant per la base de dades (contrasenya, nom d'usuari, etc) com per al servidor (si s'està emprant un entorn de producció, development, stagging, etc) que encara farien aquesta comanda més verbosa.

2.6.2 Execució de diversos contenidors amb docker compose

La solució a les comandes verboses anomenades anteriorment és docker-compose. Aquesta eina ens permet executar i configurar múltiples contenidors a partir d'un arxiu de text anomenat **docker-compose.yml**.

Docker compose crea tots els contenidors de dintre del mateix fitxer a la mateixa xarxa i ofereix una sintaxis fàcil i entenedora per afegir els paràmetres de configuració a cada container. Un exemple de com seria el docker-compose.yml dels contenidors vists anteriorment (secció 2.6.1) seria aquest:

```
version: '3'
services:
    mongo-container:
        image: mongo
        ports:
          - 27018:27018
    express-container:
        image: mongo-express
        ports:
          - 443:443
```

2.6.3 Conclusions de docker compose

Docker compose s'ha afegit ja que es una eina molt emprada en el món dels contenidors, tot i que no és més que un *wrapper*[5] de docker, és a dir, abstreu la complexitat de certes comandes i evita "barallarse" amb les xarxes de docker, però no afegeix cap més funcionalitat.

Si es volgués afegir més funcionalitats com, per exemple, creació de contenidors depenent de les peticions, balanceig de càrrega, etc, s'haurien d'emprar altres eines com, per exemple, Kubernetes (secció 3) o Docker Swarm.

[5]En català "wrapper" s'anomena embolcall

3 Kubernetes

3.1 Necessitat de Kubernetes

La tendència d'emprar arquitectures basades en microserveis (secció 1), fa que les aplicacions distribuïdes siguin formades per molts contenidors. Tal i com s'ha vist a la secció 2.6.2, docker compose ens permet "lligar" els contenidors que tenen relació entre si (cosa que va molt bé en local), però no ens permet escalar centenars de containers.

Quan arribem a un context de producció, hem de ser capaços d'organitzar i gestionar tots aquests containers per què s'autoescalin en funció de la demanda i estiguin preparats per reaccionar davant de fallades.

La solució a aquest problema la proporcionen programaris anomenats *container orchestration tools*, com ho són Kubernetes o Docker Swarm. En concret, en el nostre cas d'ús, ens centrarem en explicar **Kubernetes**.

Kubernetes ens soluciona el problema d'organització de containers, aconseguint:

- Alta disponibilitat dels contenidors, és a dir, no hi ha temps d'inactivitat del servei.
- Escalabilitat per obtenir un alt rendiment.
- *Disaster recovery*: Si falla un servidor, es capaç de fer backup i restore de les dades dels containers que es trobaven en aquell servidor.

A més, Kubernetes ens permet administrar aplicacions formades per múltiples contenidors en diferents *environments* (independentment si es troba en servidors físics, en el cloud o en màquines virtuals).

3.2 Components de Kubernetes

Abans d'explicar l'arquitectura de kubernetes, és necessari entendre les definicions dels principals components emprats per aquesta tecnologia:

- **Node**: És com s'anomena a un servidor físic, virtual machine, etc.

- **Pod**: És l'unitat més petita de Kubernetes. **Representa l'abstracció d'un container.**

  Al ser representat com una abstracció, no és necessari que el clúster de kubernetes treballi amb docker, sinó que únicament es treballa amb l'abstracció (és a dir, únicament s'accedeix a la capa de kubernetes).

  Normalment un pod acostuma a ser un container [6]. Un pod és efímer, és a dir, es pot morir/destruir en qualsevol instant de temps.

- **Service**: Cada pod té assignada una IP. Per tant, si un Pod mor i es crea un altre per reemplaçar-lo, tindrà assigna una nova IP, cosa que involucraria avisar a tots els altres pods que la ip d'aquell servei ha variat.

  La component service apareix per solucionar aquest problema, ja que prové una ip permanent per cada "servei" creat.
  Si el pod que implementava aquell service mor, el service continua tenint la mateixa ip i, per tant, el nou pod que continuarà el servei tindrà la mateixa ip.

- **Ingress**: Servidor DNS. Tradueix la DNS del client a l'ip local del service de kubernetes.

- **ConfigMap**: Component que guarda les configuracions/variables d'entorn (URLs de la base dades, configuració del port del servidor, etc).

  Per les configuracions que no s'haurien de guardar en text pla (com són contrassenyes, etc) tenim la component **Secret** que guarda les dades encriptades.

- **Volumes**: Ajunten l'emmagatzement lògic (que pot ser físic al propi clúster, al cloud, etc) amb un pod. Gràcies a aquesta component, si un pod es reinicia o es crea un altre que el substitueix, es pot recuperar l'informació.

[6] Tot i que en cas de molt acoplament entre els diferents containers es pot afegir dos containers en un pod).

- **Replicacions (Deployment)**: Crea N pods al mateix service (on N-1 pods actuen com a rèplica) i el service actua com a ip permanent i load balancer entre els N pods.

  Normalment es treballa amb deployments, no directament amb pods, ja que així es poden crear tantes rèpliques de pods com es demani.

- **Replicacions (StatefulSet)**: És similar a deployment però per a les bases de dades. Comprova que no hi hagin condicions de carrera en les escriptures, etc.

  Normalment és difícil de crear i mantenir, és per això que s'acostuma a configurar les base dades fora del clúster de Kubernetes.

Un exemple on es veu la configuració de dos nodes que empren tots els components és la figura 5.

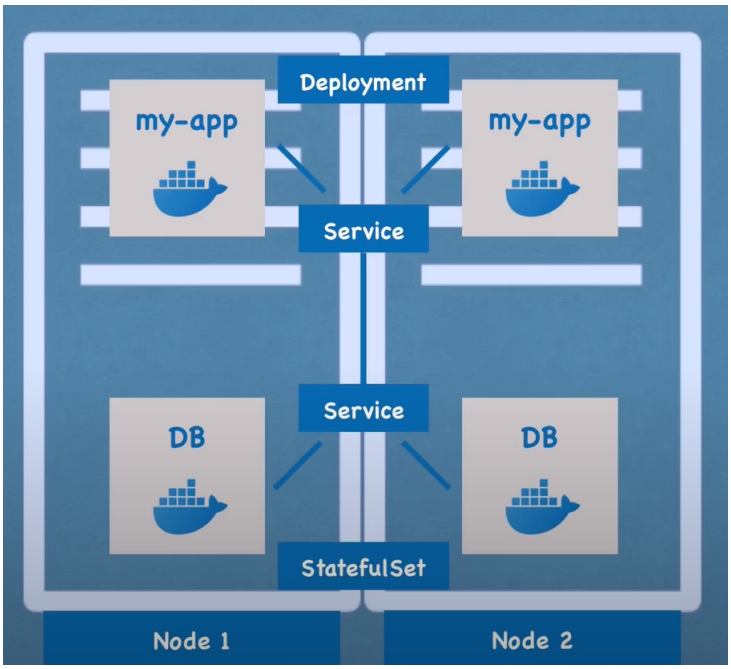

Figura 5: Configuració de nodes kubernetes emprant totes les components

3.3 Master-Slave Architecture

L'arquitectura de kubernetes consisteix en afegir dues capes més de les que tenia docker. Les capes d'un node de kubernetes són:

- Container runtime: Capa que inclou la tecnologia de contenidors. Normalment és Docker, rkt, etc.
- Kubelet: Interacciona amb el container i amb el propi node. És a dir, inicia un pod amb un container a dins.
- Kube Proxy: És la capa encarregada de la interacció entre els diferents nodes.

Un exemple de les diferents capes dels nodes es veu a la figura 6.

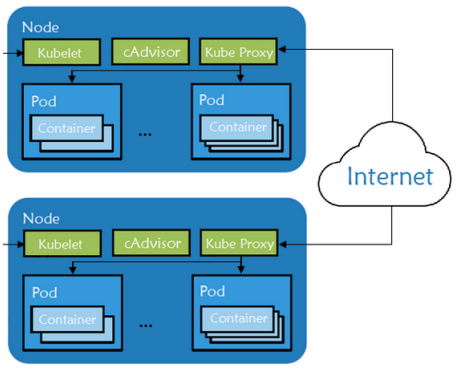

Figura 6: Nodes de kubernetes amb les capes Kubelet, Kube Proxy i Container Runtime

Amb únicament aquestes capes però, ens sorgeixen els següents dubtes:

1. A quin node hauria d'anar cada pod per tal de que la càrrega es divideixi de forma equitativa?
2. Què passa quan mor un pod? Què passa si s'ha de reiniciar un pod? Quin node serà l'encarregat de crear un nou pod i executar la tasca?
3. Com fem join d'un nou node al clúster? Com s'assabenten els altres nodes?

És per això, que Kubernetes fa la distinció entre dos tipus de nodes:

- Nodes Master: Els encarregats de gestionar el clúster. És a dir, són amb els quals accedim des de el client i amb els quals distribuïm la càrrega de pods entre els diferents nodes slaves. Tenen una arquitectura diferent a l'anomenada anteriorment.
- Nodes Slave: Són els que executen els pods que indica el node master. Tenen l'arquitectura anomenada anteriorment (container runtime, kubelet i kube proxy).

Una arquitectura de clúster típica de kubernetes seria la figura 7

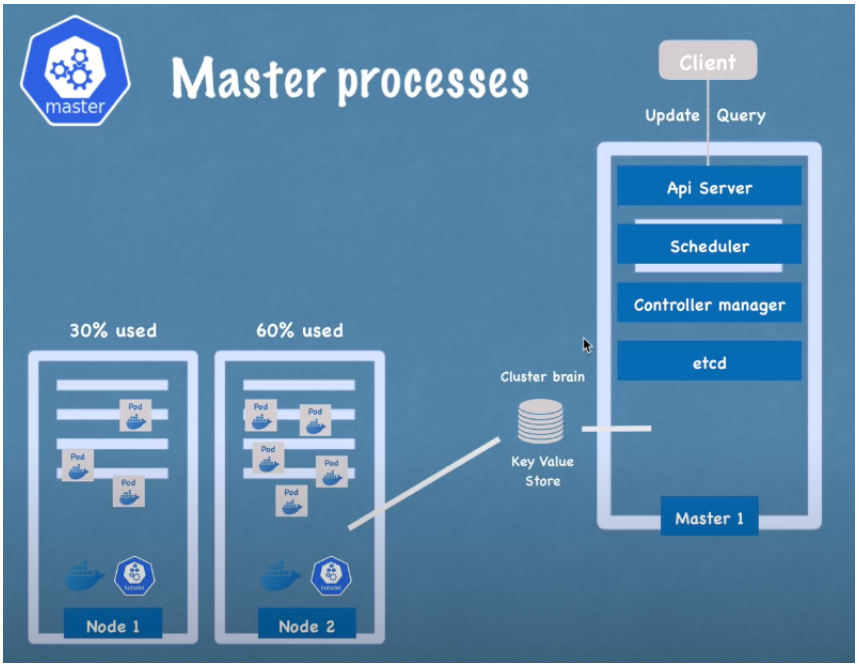

Figura 7: Arquitectura Master-Slave de Kubernetes amb 2 nodes slave i un master

3.4 Eines de Kubernetes

En les anteriors seccions, s'ha donat la teoria *bàsica* del funcionament de kubernetes i, a continuació, es nombraran algunes de les eines que permeten aplicar aquesta teoria a clústers de producció.

3.4.1 Kubectl: El kubernetes cluster manager

Aquest programa de cli ens permet comunicar-nos amb el/els nodes master per tal de configurar quins deployments o pods volem tenir, com ha de ser el balanceig de càrrega, quines xarxes donem accés als diferents nodes, etc. Algunes de les comandes més emprades són:

```
Mostra tots els nodes del clúster (exclou al master):
$ kubectl get nodes

Mostra tots els deployments creats:
$ kubectl get deployment

Crea un deployment anomenat nginx-depl
amb l'imatge de docker anomenada nginx:
$ kubectl create deployment nginx-depl --image=nginx

Edita la configuració del deployment anomenat nginx:
kubectl edit deployment nginx-depl
```

Una bona alternativa a kubectl si es prefereix tindre interficie gràfica pot ser lens, que permet visualitzar tots els diferents clústers i l'estat de cadascun de forma senzilla sense necessitat de memoritzar les comandes.

3.4.2 Minikube: Crear un clúster de kubernetes en local

Per tal de provar un clúster de kubernetes en local, podem emprar l'eina minikube.

Minikube inicia una màquina virtual amb un node master i un node slave, amb la que ens podem comunicar directament mitjançant kubectl o lens. Per tant, és una molt bona eina per practicar i provar entorns de kubernetes abans de portar-los a producció.

4 Relació amb el tema 1

En aquesta secció, s'explicarà la relació que té docker i kubernetes amb el tema 1 donat a classe. Òbviament, com s'ha anat veient al llarg d'aquest document, aquestes dues eines son emprades majoritariament per configurar sistemes distribuits, tot i que docker també té altres utilitats com, per exemple, l'integració continua (CI) i desenvolupament d'empreses amb diferents tipus de treballadors (uns utilitzen mac OS, uns altres linux, etc).

Kubernetes, en canvi, si que té un ús més enfocat únicament a sistemes distribuits i cloud computing. De fet, el que hem estat fent a la part pràctica (concretament la programació de lambdas) segurament estigui implementat amb Kubernetes i Docker.
Amazon Web Services (proveïdor cloud) segurament tingui un clúster amb kubernetes on, quan s'executa una petició, aixeca un pod amb una imatge de docker que conté la lambda que hem programat. Un cop aixecat el pod, si no s'ha rebut cap petició més en N segons, elimina el pod per tal de no malgastar cost computacional innecessàriament.

En quant a la transparència, kubernetes aconsegueix:

- Transparència de la localització on es troba un recurs. No sabem a quin node del clúster s'esta executant aquell recurs. El node podría ser un servidor d'un proveïdor cloud, una màquina situada a la pròpia empresa, etc.
- Transparència de migració: S'oculta la migració de serveis (pods), ja que únicament l'usuari contacta amb el node màster i com aquest hagi distribuït els pods en els diferents nodes slave no és una informació que conegui.
- Transparència de reubicació: La reubicació també es transparent, ja que el clúster pot moure un recurs a un altre node mentre s'utilitza i l'usuari no ho pot saber.
- La replicació és transparent gràcies als deployments i services de kubernetes.
- Les fallades i les recuperacions d'aquestes es fan automàticament gràcies al node màster. Són totalment transparents per l'usuari final.

Com a resum, podem dir que Kubernetes ens abstreu on es troben situats els serveis, de les seves caigudes, etc. Mentre que docker es una eina que ens permet encapsular els serveis i traslladar-los amb facilitat, independentment de les seves dependències i entorn.

5 Conclusions

Després de l'anàlisi exhaustiu de les diferents eines (docker i kubernetes) i l'arquitectura basada en microserveis, s'ha pogut concluir que:

1. Docker és una eina de creació de contenidors, els quals permeten aixecar serveis molt més ràpidament que les maquines virtuals amb "l'inconvenient" d'emprar el kernel del sistema operatiu.

2. Docker té eines com docker-compose que ens ajuden a abstreure comandes més complexes, però que no ens solucionen la creació i escalabilitat de molts containers.

3. Les *container orchestration tools* com kubernetes o docker swarm són necessàries per tal de gestionar un conjunt gran de contenidors.

4. Docker i Kubernetes no són competència entre sí, sinó que son eines diferents que s'empren per a propòsits diferents.
   Connectades, ens ajuden a formar clústers potents i escalables (tal i com es fa en els servidors cloud).

6 Bibliografia

Referències

[2] IBM. *Cloud Native Applications*. Maig de 2019. URL: https://www.ibm.com/cloud/learn/cloud-native (cons. 21-03-2022).

[6] Microsoft. *What is Docker?* Set. de 2021. URL: https://docs.microsoft.com/en-us/dotnet/architecture/microservices/container-docker-introduction/docker-defined (cons. 29-03-2022).

[7] Microsoft. *What is a container?* URL: https://azure.microsoft.com/en-us/overview/what-is-a-container/#overview (cons. 29-03-2022).

[10] OpusSoftwareTeam. *Microservices: Concepts and Characteristics*. URL: https://www.opus.software/microservices-concepts-and-characteristics/ (cons. 21-03-2022).

Més informació no citada

[1] Docker. *Dockerfile reference*. URL: https://docs.docker.com/engine/reference/builder/ (cons. 22-03-2022).

[3] IBM. *learn microservices*. Març de 2021. URL: https://www.ibm.com/cloud/learn/microservices (cons. 14-03-2022).

[4] Nana Janashia. *Docker Tutorial for Beginners*. Oct. de 2020. URL: https://www.youtube.com/watch?v=3c-iBn73dDE (cons. 14-03-2022).

[5] Nana Janashia. *Kubernetes Tutorial for Beginners*. Nov. de 2020. URL: https://www.youtube.com/watch?v=X48VuDVv0do (cons. 14-03-2022).

[8] NetworkChuck. *You need to learn Docker RIGHT NOW!* Abr. de 2020. URL: https://www.youtube.com/watch?v=eGz9DS-aIeY (cons. 14-03-2022).

[9] NetworkChuck. *You need to learn Kubernetes RIGHT NOW!* Set. de 2020. URL: https://www.youtube.com/watch?v=7bA0gTroJjw (cons. 14-03-2022).

[11] Christian Posta. *The Hardest Part About Microservices: Your Data*. Ag. de 2016. URL: https://developers.redhat.com/blog/2016/08/02/the-hardest-part-about-microservices-your-data (cons. 14-03-2022).

[12] Txema Rodríguez. *De Docker a Kubernetes: entendiendo qué son los contenedores y por qué es una de las mayores revoluciones de la industria del desarrollo*. Set. de 2019. URL: https://www.xataka.com/otros/docker-a-kubernetes-entendiendo-que-contenedores-que-mayores-revoluciones-industria-desarrollo (cons. 14-03-2022).

[13] Sarabjeet Singh. *Docker containers vs Images*. URL: https://www.educba.com/docker-containers-vs-images/ (cons. 22-03-2022).

[14] Microsoft Azure Team. *Kubernetes vs Docker*. URL: https://azure.microsoft.com/es-es/topic/kubernetes-vs-docker/ (cons. 14-03-2022).