Treball de Fi de Grau

GRAU D'ENGINYERIA INFORMÀTICA

Facultat de Matemàtiques
Universitat de Barcelona

CoreOs per a aplicacions robustes i escalables

Miquel Puigdomenech Poch

Tutors: Jesús Cerquides Bueno,
Marc Pujol Gonzalez

Realitzat a: Facultat de Matemàtiques
de la Universitat de Barcelona

Barcelona,16 gener de 2014

Índex

Índex d'imatges

Abstract

This project starts by researching the current software and architecture design trends to develop applications for the cloud. The main goal we want to achieve is to define both a system and an application architecture that enables small to mid-sized businesses to develop cloud-based scalable and resilient applications. Furthermore, the proposed architecture should not rely on the specific services provided by any of the big providers (e.g.: Amazon AWS, Google Cloud), to avoid tying the company's fate to those.

We reach this goal first researching about the different cloud-oriented software paradigms, and identifying the Operating System as a Service (OSaaS) offerings as a promising foundation on which to build our system. OSaaS is a very recent paradigm that lies between the more well-stablished Platform as a Service (PaaS) and an Infrastructure as a Service (IaaS) offerings. We pick OSaaS because it offers more portability than PaaS systems (that tie your system to the specific vendor platform) while still providing useful coordination and orchestration tools (whereas you must develop those yourself in IaaS offerings).

We then compare two similar technologies in that space: MesOS and CoreOs. CoreOS is more strict in its approaches and offers a unified experience, whereas MesOS employs several semi-independent tools to provide a similar experience. After comparing these platforms we choose CoreOs to develop our proposed architecture. Essentially, CoreOS is a minimal linux distribution specifically designed to run as a containerization host, plus a number of orchestration services to coordinate multiples CoreOS nodes in a cluster. As a result, it allows for the design of resilient and consistent services and applications (the two main things we want).

Next we analyze the requirements that our software must fulfill to exploit the capabilities of such clusters. We explore the 12 factor app concept, that describes a number of aspects to consider when developing software for the cloud. In doing so, we adapt some of these aspects and relate the general guidelines to how these will affect the application performance within our CoreOS cluster.

Finally we deploy a small virtual cluster implementing our proposed architecture and experiment with it. We deploy several services and test their functionality and resilience. These experiments demonstrate that certain kinds of services (such as web applications) are better suited for our architecture, whereas others (such as databases) still present some unsolved cha
llenges.

Resum

L'objectiu d'aquest treball de final de grau és realitzar una recerca en l'entorn de les aplicacions o serveis cloud. En concret el que busca és definir una estructura tan de sistema com d'aplicació per a satisfer les necessitats determinades d'una empresa mitjana.

Assolim aquest objectiu investigant un seguit de paradigmes que han sorgit fa relativament poc , en especial ens centrem en investigar que és OSaaS. Sistema operatiu com a servei és un terme nou el qual ens descriu el fet de proporcionar un servei entremig de un PaaS (Plataforma com a servei) i un IaaS (infraestructura com a servei).

Per analitzar aquest nou paradigma compararem dues tecnologies similars de les quals una ofereix aquest nou tipus de servei directament i l'altre amb una combinació d'elements també. Després de la comparació ens decantarem per una tecnologia i l'analitzarem a fons.

Nosaltres en aquest cas ens hem decantat per CoreOs un sistema operatiu linux dissenyat per a funcionar en clúster , oferir-nos solidesa i fiabilitat. Amb la idea de l'estructura passarem a investigar quins requisits ha de tenir una aplicació que funcioni a sobre d'aquesta, analitzarem els 12 factor app per a veure si encaixen amb les necessitats de la nostra aplicació.

Una vegada tinguem això proposarem una estructura de clúster funcional i una estructura per a les aplicacions que han de funcionar a sobre d'aquest clúster.

Finalment implantarem aquestes aplicacions al clúster i avaluarem el seu resultat per a realitzar una conclusió sobre el funcionament del sistema.

1. Introducció i Antecedents

L'evolució de les noves tecnologies ens ha portat a un món constantment connectat i consumidor voraç d'aplicacions. En aquest entorn els usuaris cada vegada més demanden la disponibilitat ininterrompuda dels serveis que utilitzen i critiquen durament qualsevol caiguda dels mateixos. Degut a això el paradigma al que es tendeix darrerament és el d'oferir serveis cada vegada més escalables i fiables. Això implica tenir un sistema distribuït al darrere, que pugui suportar fallades de varis nodes mantenint la qualitat del servei i que no s'hagi de parar ni tan sols per actualitzar. No obstant, mantenir un sistema d'aquestes característiques suposa un cost molt elevat i necessita una estructura complexa que difícilment pot ser desenvolupada per petites i/o mitjanes empreses.

Davant d'aquest problema trobem que hi ha diverses iniciatives per millorar la manera de mantenir, desenvolupar i actualitzar aquests serveis. S'acostuma a buscar augmentar l'eficiència i reduir el cost. Malgrat això mantenir una infraestructura d'aquest tipus suposa un cost molt elevat i per tant impossible d'assolir amb empreses no gaire grans.

Una de les empreses referència que ens trobem en aquest sector és Amazon, que amb els seus serveis Amazon Web Services (AWS) s'emporta una important quota de mercat. Aquest domini s'explica pel fet de ser l'empresa pionera en apostar per llogar aquest tipus d'arquitectura massivament distribuïda acompanyada d'un conjunt de serveis bàsics per facilitar-ne l'explotació.

A part de la infraestructura mateixa, la novetat fonamental introduïda per Amazon en aquesta època és el fet de poder-la llogar per a períodes curts de temps, fins i tot per una sola hora. Gràcies a aquesta possibilitat, es poden començar a desenvolupar aplicacions que escalen segons les necessitats del moment, afegint o traient màquines segons convingui.

En el panorama actual també ens trobem un seguit de sistemes que busquen optimitzar al màxim el rendiment reduint els costos. Es formulen també com una alternativa a les grans companyies com Amazon o Google. Per això l'objectiu del projecte és analitzar aquest tipus de sistemes i indagar en el potencial que ens poden arribar a oferir actualment. Proposant també una estructura funcional a utilitzar a l'hora d'implementar una infraestructura per a poder fer funcionar un conjunt de serveis.

Concretament volem trobar una estructura capaç d'adaptar-se a les necessitats d'una empresa mitjana, que no vulgui dependre d'un proveïdor únic i vulgui una

alternativa més econòmica i modular capaç d'adaptar-se a la majoria de sistemes. Un exemple d'això seria una start-up que volgués oferir un servei de creació i hosting de botigues virtuals.

En la resta d'aquest apartat ens endinsarem primer en les motivacions que donen origen a aquest projecte. Tot seguit analitzarem l'evolució dels sistemes fins arribar a la idiosincràsia actual, quins reptes plantegen i quines són les aproximacions proposades per solucionar aquests reptes. A continuació farem una estimació del cost del projecte. Finalment, definirem clarament els objectius a aconseguir en aquest projecte.

1.1- Motivació

Els usuaris d'avui en dia demanden una alta disponibilitat dels serveis, i la màxima fiabilitat de les seves dades emmagatzemades. És més, el gran avantatge que esperen de les aplicacions al núvol és precisament aquest: disponibilitat en qualsevol moment i ubicació, amb màximes garanties de fiabilitat. És per això que cada vegada més resulta imprescindible desenvolupar aplicacions basades en serveis escalables i tolerants a fallades. Aquests requeriments han donat lloc a la recerca d'arquitectures que permetin la utilització simultània de múltiples nodes coordinats, facilitant l'escalabilitat del sistema i garantint-ne la disponibilitat enfront a fallades de l'equipament.

Malgrat els avantatges potencials d'aquest nou paradigma són clars, també comporten un increment significatiu de la complexitat a l'hora de dissenyar, implementar i mantenir el sistema. Essencialment, aquest tipus de sistemes requereixen mecanismes de coordinació entre els nodes participants, mecanismes que en nombroses ocasions són inclús més complexes que els propis serveis o aplicacions oferts pel sistema. Per tant, el desenvolupament d'entorns que proveeixin aquests mecanismes és un repte important, que actualment està rebent gran atenció i recursos tant en recerca com en el món empresarial.

El present projecte respon doncs a la inquietud de veure com aprofitar aquestes iniciatives per desenvolupar aplicacions al núvol que satisfacin els requeriments actuals dels usuaris. A tal efecte, és necessari disposar d'un anàlisis amb profunditat dels diferents tipus de sistemes disponibles i les plataformes lliures que s'estan desenvolupant actualment per implementar-los, tenint sempre en compte la divergència de requeriments entre diferents tipologies d'aplicacions. Idealment, també hauríem de disposar de recomanacions clares sobre l'estructura funcional a utilitzar a l'hora d'implementar una aplicació d'aquest estil.

No obstant, abans d'analitzar en profunditat la conjuntura actual és necessari

comprendre com s'ha arribat a aquest punt. Per tant, en el següent apartat repassarem breument l'evolució de les plataformes de computació des dels seus orígens fins al moment actual.

1.2- Evolució de les plataformes/arquitectures

A continuació presentem breument l'evolució del concepte de computació, des dels seus orígens fins arribar a la situació en que ens trobem actualment.

Els inicis de la informàtica i els grans ordinadors

Als inicis de la informàtica, els ordinadors eren grans màquines enormement costoses de construir i operar, fins i tot requerint nombrosos operadors humans. Aquestes màquines eren propietat de grans institucions i el seu ús diligentment controlat, de manera que només especialistes podien emprar-les i tant sols per propòsits molt específics.

Ordinador Personal (PC)

Gràcies a nombrosos avanços tecnològics, més tard s'inicia l'era de l'ordinador personal (PC), que democratitza la informàtica fent-la accessible per primer cop al públic en general. És durant aquesta era quan es comencen a informatitzar nombroses tasques quotidianes, com la redacció i gestió de documents o el manteniment de registres informatitzats. Tot i així, el concepte inicial de PC és el d'un ordinador aïllat, que només pot interactuar amb altres a través de dispositius d'emmagatzematge extern.

Xarxes i serveis remots

El següent que es va fer va ser connectar aquests PCs entre ells per tal de compartir informació. Des de les connexions en local (LAN) fins la implantació de internet, passant per connexions de grup (BBS), les xarxes donen lloc a una nova era de compartició d'informació, serveis i recursos. Gràcies a aquestes xarxes es comença a treballar en l'oferiment de serveis remots, és a dir, en permetre a usuaris remots utilitzar serveis que no s'executen en el seu ordinador. Durant aquesta època, el model client-servidor per a la prestació i desenvolupament de serveis guanya en notorietat fins a convertir-se en un dels pilars fonamentals dels sistemes actuals, amb la Web al capdavant com exemple d'èxit rotund d'aquesta aproximació.

Clústers

A mesura que els servidors creixen en complexitat i nombre d'usuaris a servir, es fa palesa la necessitat de desenvolupar mecanismes que protegeixin de fallades d'aquests sistemes i en millorin l'escalabilitat. Degut a aquestes

necessitats, s'impulsa amb força la idea de desenvolupar clústers d'ordinadors que treballin conjuntament per oferir un mateix servei. Inicialment, aquests clústers es desenvolupen principalment per a realitzar grans tasques de computació «fàcilment» paral·lelitzable, on és justificable l'elevat cost d'adquirir nombrosos ordinadors i interconnectar-los.

Compute on demand

Amb la presentació el 2006 de l'Elastic Computing (EC2) per part d'Amazon entrem de ple en el que es coneix com a Cloud Computing[1] (informàtica al núvol) que permet disposar d'un clúster d'ordinadors i d'altres serveis a un preu assequible, podent llogar ordinadors a conveniència i per curts espais de temps. Tal i com abans les companyies ens oferien uns serveis en remot però amb un cost fix i escalables amb dificultat ara les companyies ofereixen serveis sota demanda fins i tot per hores. Això ens proporciona una escalabilitat molt gran i ens permet assolir una tolerància a fallades molt bona. Tot i que requereix un augment de la complexitat i el cost del sistema ja que aquest ha de tenir en compte moltes variables , afrontar grans reptes com la consistència de les dades o obtenir un rendiment adequat davant un gran volum de dades.

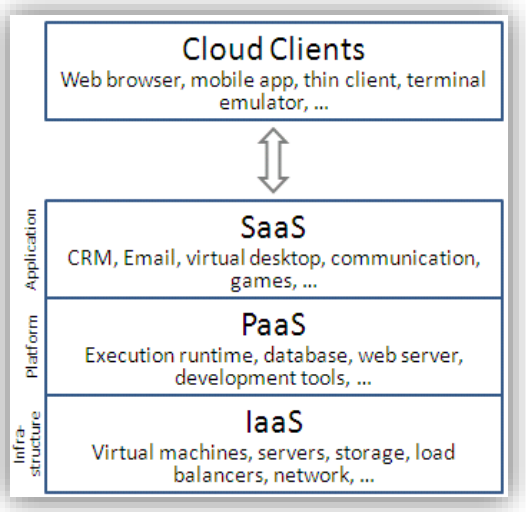
Cada vegada més doncs, es tendeix a veure més el núvol com una font de recursos i serveis, independentment de l'estructura que tinguin i per tant sense requeriments. La mancança de requeriments específics ha donat lloc a multitud de conceptes de núvol diferents. És per això que, fruït de la necessitat, s'ha establert una classificació «de facto» per diferenciar a grans trets les diferents aproximacions per aprofitar el concepte de computació al núvol, i que presentem a continuació:

SaaS: Software as a Service o software com a servei és el concepte de proporcionar un servei a l'usuari, des d'un gestor de correu fins a una aplicació de comptabilitat.

PaaS: Platform as a Service o plataforma com a servei és una de les categories de serveis al *cloud* que ens proporciona una plataforma per a treballar. Un conjunt d'eines a partir les quals podem desenvolupar els nostres serveis, des d'una base de dades fins a un servidor web.

IaaS: Infraestructure as a Service o infraestructura com a servei és la part de la infraestructura que ens ofereixen que controla la memòria, els servidors, la xarxa.

OSaaS[2]: Operating System as a Service o sistema operatiu com a servei és un terme més modern el qual es trobaria entremig d'un Paas i un IaaS ja que ens ofereix una plataforma sobre la qual desenvolupar i distribuir les nostres aplicacions gestionant per sota tot l'estat del clúster.



Imatge 1: Estructura de serveis cloud desglossada

Per aprofitar els beneficis d'aquesta nova estructura es necessari reconsiderar la manera de construir software i per això sorgeixen patrons de desenvolupament com la integració contínua (CI).

Ara que ja hem vist l'estat actual i com hem arribat fins aquest passarem a veure quins són els objectius plantejats per a aquest projecte.

1.3- Objectiu general

L'objectiu principal d'aquest projecte és analitzar l'estat de l'art en plataformes de OSaaS per a poder recomanar un model d'aplicació. En concret es busca tant un model d'aplicació com la infraestructura necessària per oferir un servei distribuït, escalable i tolerant a fallades.

Per aconseguir aquest objectiu hem dividit el projecte en aquests objectius més específics:

- Anàlisis dels requeriments estructurals d'una aplicació per al núvol.
- Investigació de les plataformes OSaaS disponibles actualment.
- Identificació dels tipus d'aplicacions on aquesta aproximació és aconsellable i les seves limitacions.
- Recomanació d'una infraestructura OSaaS, de codi lliure, escalable i tolerant a fallades.
- Desplegament i prova del sistema proposat.

1.4- Objectius específics

A continuació desglossarem aquests objectius i els analitzarem.

- **Anàlisis dels requeriments estructurals d'una aplicació per al cloud**

Recerca sobre que ha de disposar una aplicació cloud en el context actual, tan el tipus de estructura que ha de tenir com el servei que ha de donar. Indagar també el tipus d'estructura necessària per a sostenir aquest tipus d'aplicacions.

- **Investigació de les plataformes OSaaS disponibles actualment**

Recerca i investigació de les diverses plataformes i sistemes que ens permeten crear un clúster distribuït i robust. Investigació dels elements necessaris per muntar un sistema distribuït, escalable i tolerant a fallades.

- **Identificació dels tipus d'aplicacions on aquesta aproximació és aconsellable i les seves limitacions**

Identificació de els avantatges que aquesta aproximació ens porta els seus beneficis , les seves limitacions i els principals sistemes que ens ho proporcionen.

- **Recomanació d'una infraestructura concreta, de codi lliure, escalable i tolerant a fallades**

Recomanació d'una infraestructura per a obtenir un clúster distribuït. Posant especial èmfasis en que sigui robust i consistent.

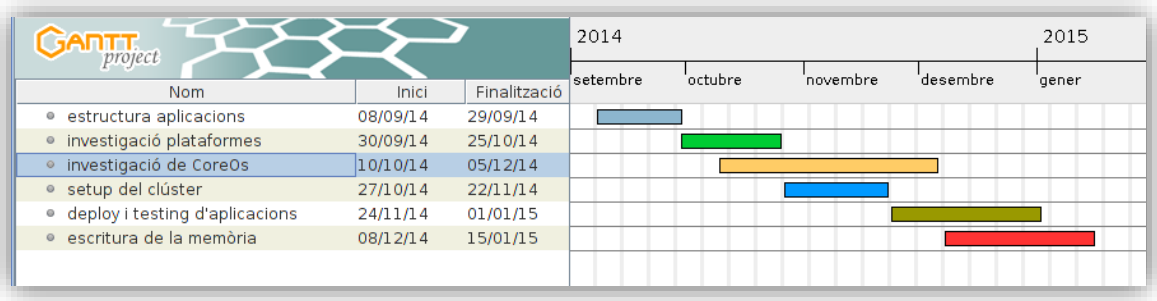
- **Desplegament i prova del sistema proposat**

Proves d'implantació d'un servei web i una base de dades. Avaluació del funcionament i la tolerància a fallades.

Una vegada ja hem vist tots els objectius i que volem aconseguir amb cada un d'ells en el següent apartat veurem la planificació que hem fet per tal de dur a terme aquest projecte.

1.5- Planificació

A continuació mostrarem la planificació del projecte, hem dividit els objectius en diverses tasques les quals expliquem després de mostrar-les utilitzant un diagrama de Gantt.



Imatge 2: Diagrama de Gantt del projecte

- **estructura aplicacions i sistema:** durant aquesta tasca hem realitzat una investigació de l'estructura de les aplicacions cloud i sistemes que han de suportar aquestes.

- **investigació plataformes :** durant aquesta tasca hem fet una recerca i investigació de les diverses plataformes i sistemes disponibles per a suportar una aplicació cloud.

- **investigació de CoreOs :** durant aquesta tasca hem realitzat una investigació en profunditat del funcionament i l'estructura del sistema CoreOs. També hem hagut d'investigar altres sistemes que utilitza internament.

- **configuració del clúster:** durant aquesta tasca hem configurat el clúster de CoreOs i hem comprovat que funcionés correctament.

- **deploy i testing d'aplicacions:** durant aquesta tasca hem buscat i desenvolupat aplicacions per a que funcionessin en el clúster. També hem realitzat una avaluació d'aquestes aplicacions i de la resposta del clúster en el cas de fallades.

- **escriptura de la memòria:** durant aquesta tasca hem agrupat tots els coneixements, hem extret un seguit de conclusions i resultats que hem plasmat en una memòria.

1.6- Organització de la memòria

La memòria d'aquest projecte està organitzada en els següents apartats:

- Introducció i Antecedents

En aquest apartat s'explica la motivació que ha portat a fer el projecte, es mencionen els objectius i també s'explica els antecedents a l'estat actual.

- Anàlisis de l’estat de l’Art

En aquest apartat s’analitza l’estat en el que es troba aquest paradigma (OSaaS), quins són els sistemes referència i quin d’aquests s’adequa més a les nostres necessitats.

- CoreOs i Docker

En aquesta part s’explica detalladament el funcionament de CoreOs i docker. Repassant les diverses eines que els formen

- Requeriments per a una aplicació Cloud

En aquest apartat realitzem una exposició dels requeriments que ha de tenir una aplicació cloud. En concret ens basem en els 12 factor app, els quals analitzem i contrastem amb els nostres requisits.

- Arquitectura proposada

En aquest apartat proposem una arquitectura d’infraestructura i de disseny de serveis per a funcionar sobre aquesta.

- Exemples proposats

En aquest apartat realitzem un desplegament sobre el clúster d’un seguit d’exemples per a poder provar-los i obtenir una visió del funcionament o no d’aquests.

- Conclusions i treball futur

En aquest apartat repassem la feina feta , també proposem vies per a seguir amb aquest projecte.

- Referències bibliogràfiques

- Annexe: Setup del clúster

En aquest apartat expliquem com muntar el clúster que hem utilitzat durant la implantació i la prova dels exemples proposats.

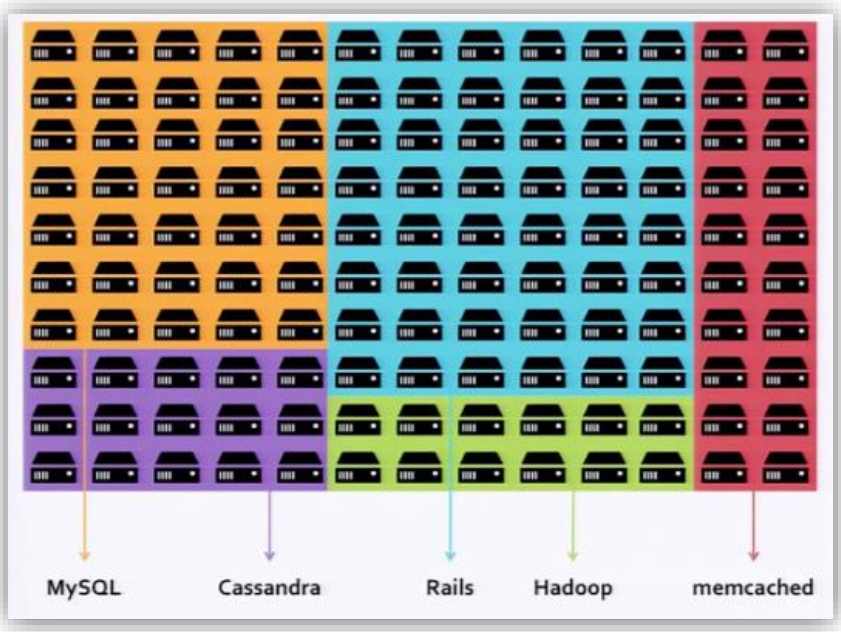
2.Anàlisis de l'estat de l'Art

Abans de posar-nos directament a analitzar aquest nou paradigma de programacions i arquitectures hem de tenir molt present que es troba en moment d'expansió i desenvolupament, encara manca temps per arribar a una solidificació, crear unes bases fixes i uns termes clars.

Per tant l'anàlisi que farem sobre aquests sistemes és important agafar-lo en el moment en el que ens trobem ja que algunes de les tecnologies que mencionarem estan en desenvolupament i encara no han arribat a producció ni a oferir tot el potencial possible.

Com que cada vegada més volem fer les coses modulars i escalables la majoria de tecnologies que veurem a continuació han optat per a canviar de metodologia i utilitzar la virtualització a baix nivell a l'hora de fer córrer els processos i serveis per als clústers.

Tal i com mencionava Benjamin Hindman, el cofundador de MesOS i membre tècnic de la plantilla de twitter, en segons quins casos no es pot tenir una estructura tradicional [3] (al núvol centralitzada i amb uns nodes dedicats a un seguit de tasques específiques).

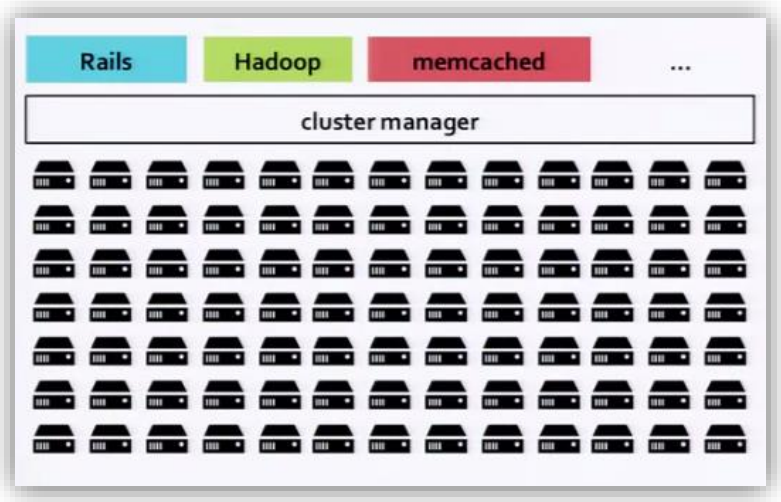


Imatge 3: Estructura de clúster tradicional

Aquesta estructura és inflexible i es pot donar el cas de que els serveis que tinguem funcionant al clúster tinguin necessitats variables. Amb l'estructura fixada no tenim capacitat de redirigir els recursos no utilitzats a un servei.

És per això que actualment les empreses tendeixen a adoptar un sistema que permeti disposar de tots els nodes per qualsevol tasca requerida, es a dir, tenir un gestor del clúster i que aquest decideixi quin node executa quina tasca i amb quin tipus de sistema. Cal mencionar que això pot semblar similar a

l'estructura que utilitza hadoop per a assignar tasques, però en aquest cas no estem parlant de tasques, sinó de processos servidors (dimonis).



Imatge 4: Estructura de clúster flexible

Al parlar de processos hem de tenir en compte que possiblement necessitarem tenir una còpia del sistema en el que s'executa aquest procés en cada node. Fer una aproximació a aquest nivell ens portaria a crear uns nodes grossos i ineficients que han de disposar de tot el programari que possiblement hagin d'executar incloent-hi bases de dades, servidors web, codi d'aplicacions...

L'idea d'aquest nou tipus de tecnologia es basa en tenir els menors components en cada node i en utilitzar tecnologia de virtualització a baix nivell per a obtenir un rendiment similar al comú sobre "hardware real" però amb més flexibilitat.

Quan parlem de virtualització a baix nivell hem de deixar de banda la imatge de màquina virtual que ens ve al cap directament, les màquines virtuals tradicionals en les que necessitem tenir un sistema sencer per a executar un servei determinat. Això, la majoria de vegades, ens generava problemes de memòria. Quan parlem de virtualització a baix nivell parlem d'una virtualització utilitzada majoritàriament en servidors on el kernel del sistema operatiu permet diverses instàncies d'usuari aïllades. Una de les tecnologies més conegudes que utilitza aquest sistema de virtualització és Docker [4].
Docker és un projecte que utilitza l'aïllament de recursos del kernel de linux per a permetre que diversos contenidors pugin funcionar en una mateixa instància de sistema.

Altres projectes similars a docker són : LXC, Solaris Containers, lmctfy, Rocket.
Nosaltres ens centrarem en explicar docker ja que és una alternativa open source i que s'està establint com un element de referència en aquest tipus de virtualització.

A part d'una tecnologia com aquesta també necessitem una manera de gestionar tot el clúster. Aquí és on entra el gestor de clúster (clúster manager), aquest element del sistema s'encarrega d'assignar processos, controlar els nodes i gestionar les fallades.

Tenim un ventall d'opcions molt gran també tant en la manera d'implementar la gestió del clúster com de les eines que utilitzen.

Aquí es on podem començar a veure la diferència entre dos tipus d'estructures. Tenim un tipus d'estructura que es basa en proporcionar uns recursos i assignar-los en funció de les necessitats i capacitats de cada node, i tenim altres estructures que es preocupen només de que les tasques es realitzin de manera constant prioritzant l'estabilitat del sistema.

La primera estructura ens dona eficiència i disponibilitat. El problema que ens afegeix però és que hem de comptabilitzar de manera periòdica els recursos de cada màquina i la seva disponibilitat en funció de la càrrega que estigui duent a terme.

La segona estructura en canvi ens dona estabilitat, seguretat i rapidesa, per altra banda es pot donar que no utilitzem els recursos de que disposem de manera eficient (ja que per defecte s'assignen a una màquina sense feina).

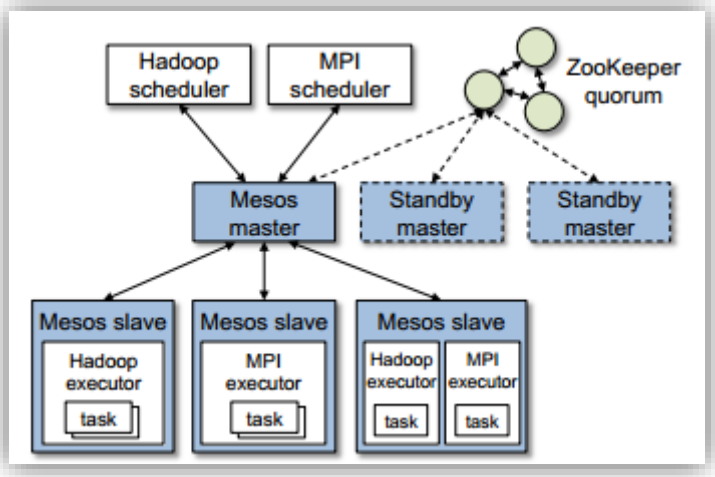
Per a comparar aquests dos tipus de funcionament agafarem dos sistemes que treballen cada un amb una de les estructures.

Per un costat tenim MesOS

Imatge 5: Logo d'Apache MesOs

Apache MesOs[5] és un sistema dissenyat a la Universitat de California Berkeley el 2010. MesOs és un gran exemple de PaaS i acostuma a funcionar sobre de CentOS o Ubuntu server; és un bon exemple del primer tipus d'estructura ja que el seu sistema es basa en comptabilitzar els recursos que té, nombre de nodes i tipus, per a llavors distribuir la tasca, o jobs, que li arribin a través del clúster de màquines en funció del tipus de màquines que hi hagin i el tipus de tasques a realitzar. La seva arquitectura es basa en un node mestre i un seguit de nodes esclaus els quals realitzen la feina assignada pel node mestre.
La fiabilitat i tolerància a fallades és controlada pel node mestre, el qual disposa d'un parell de nodes "backUp" o com ells anomenen Standby Master. Aquests nodes poden reconstruir l'estat del node master, en qualsevol moment, gràcies

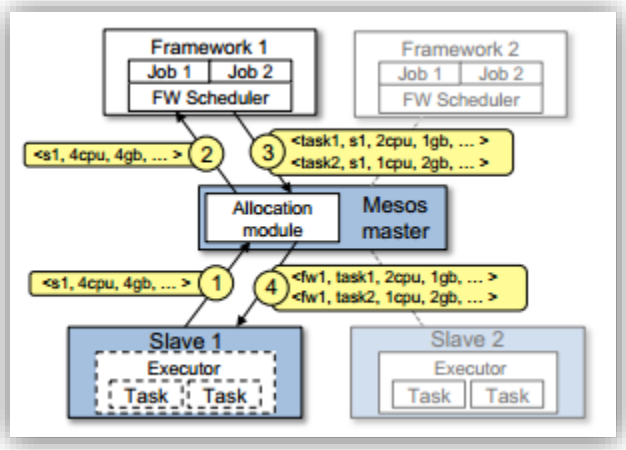
a un sistema de missatges periòdics que mantenen amb ell. En cas molt greu de fallada també pot passar-se el control a un altre dels mestres.



Imatge 6: Estructura de masters de MesOs

El sistema de distribució de les tasques es basa en un sistema d'oferta/demanda. El node mestre pregunta el tipus de recursos que te cada node i llavors en funció de la capacitat de processat i de la memòria disponible assigna un tipus de tasques o unes altres.

A la següent figura podem comprovar el funcionament d'aquesta arquitectura.
Mesos aprofita el fet de que la majoria de tasques proposades als nodes són curtes per a que aquestes no hagin d'esperar i així poder reubicar recursos ràpidament quan aquestes acaben.



Imatge 7: Exemple del repartiment de Jobs amb MesOs

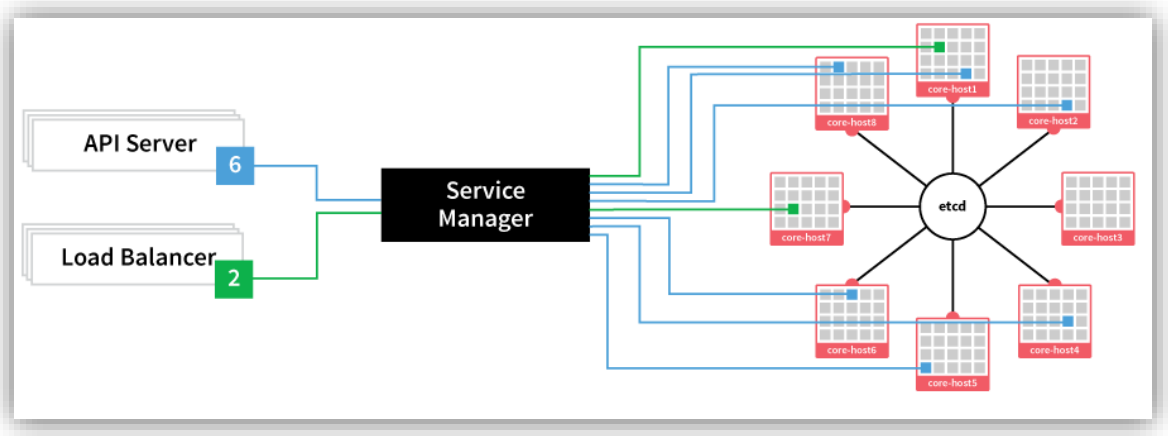
Per gestionar la fiabilitat i la tolerància a fallades sobre els nodes utilitza una eina anomenada Zookeeper.
Aquesta tecnologia també es beneficia de la tecnologia de virtualització per a obtenir un aïllament sobre el framework amb el que vulguem treballar (Scala, hadoop,…) i des de l'última versió del sistema suporten els contenidors de Docker.

Per l'altre costat tenim CoreOs

Imatge 8: Logo de CoreOs

CoreOs [6] és un sistema amb alguns trets similars a l'arquitectura anterior però completament diferent en altres aspectes.

Tal i com abans parlàvem d'una plataforma la qual necessitava un sistema base a l'hora de funcionar ara en canviparlem d'un sistema que és sistema i plataforma alhora. A diferència de MesOs però CoreOs no té un màster que gestioni directament la càrrega de cada node en funció de la tasca a realitzar, sinó que el conjunt de nodes (en sistemes molt grans és un petit conjunt de nodes) utilitzen un sistema de puges per a cada job i es posen d'acord entre ells per assignar-ho a un o altre node. A diferència de MesOs està més dissenyat a tenir serveis actius i no tant a realitzar tasques curtes.
El fet de que treballi sobre la màquina i estigui dissenyat específicament per a funcionar a través de contenidors fa que sigui extremadament lleuger i consumeixi molt pocs recursos.



Imatge 9: Esquema de la distribució de tasques a CoreOs

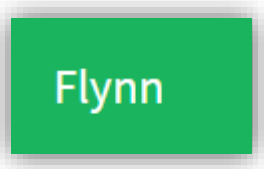
El funcionament dels nodes és bastant diferent al de MesOs ja que aquí tots ells utilitzen una instància a etcd (tecnologia que explicarem mes endavant) per a coordinar-se entre si. A la mateixa vegada fleet (una altra tecnologia muntada sobre etcd) s'encarrega de gestionar el clúster i de gestionar qualsevol fallada o reestructuració del servei.

Aquest sistema també s'aprofita de la tecnologia dels contenidors per a desplegar els serveis corresponents.

Tal i com hem mencionat és un sistema molt lleuger, tan lleuger i escalable que muntar una instància de coreos sobre el mateix hardware per a afegir-ho en un clúster i tenir-lo funcionant pot realitzar-se en qüestió de pocs minuts[7].

Un altre dels grans avantatges que té és el seu sistema d'actualitzacions amb doble partició mantenint una partició activa mentre actualitza l'altra assegurant estabilitat i possibilitat de fer un rollback en qualsevol moment.

A part d'aquestes dues tecnologies també en trobem d'altres, algunes inspirades en aquestes dues i d'altres que funcionen en conjunció amb aquestes. Les més destacades són :Deis, Flynn, Tsuru. Particularment hem escollit aquestes tecnologies ja que són les que més s'aproximen al funcionament que hem estat veient fins ara.



Imatge 10: Logo de Flynn

Flynn [8]: és una plataforma en fase beta que funciona sobre UbuntuServer utilitzant contenidors de Docker per a gestionar un clúster. Té una estructura de capes les quals donen modularitat al sistema i permeten posar les eines que preferim a l'hora de gestionar el clúster. Per exemple permet utilitzar tan etcd com Zookeeper.

Imatge 11: Logo de Tsuru

Tsuru [9]: és una plataforma que funciona sobre UbuntuServer i que ofereix un PaaS amb la intenció de que utilitzant un controlador de versions git puguis desplegar aplicacions de manera automàtica. Com les tecnologies que hem vist anteriorment utilitza Docker a l'hora de fer funcionar les aplicacions. El problema és que encara està en fase de desenvolupament i encara no disposa d'una versió estable.

Imatge 12: Logo de DEIS

Deis [10]: és una tecnologia que busca agilitzar el desenvolupament i execució de les aplicacions. Funciona sobre CoreOs. És una eina per a crear automàticament un desplegament de les aplicacions des d'un servei de control de versions git. El sistema s'encarrega d'empaquetar l'aplicació en un contenidor i llençar-la a través del clúster. Es troba en desenvolupament i encara no hi ha versió estable.

Després de veure aquest seguit de tecnologies podem veure que clarament

unes són més convenients en un entorn i unes altres en un altre entorn.

Si per exemple volem un clúster amb molta capacitat de processat i eficiència clarament hauríem de muntar un clúster de maquines amb MesOs a sobre per a obtenir un millor rendiment.

En canvi si volem desplegar una aplicació sobre la cloud que no requereixi massa rendiment i computació podem muntar una estructura amb CoreOs.

L'avantatge d'aquestes tecnologies és que són compatibles per tant també podríem muntar un clúster de CoreOs amb MesOs a sobre, així com llençar contenidors docker sobre MesOs.

Posant la vista al món empresarial cada vegada més es tendeix a treballar amb grans volums de dades i amb més serveis distribuïts. Com ja hem vist anteriorment això requereix una gran elasticitat i modularitat a l'hora de fer funcionar aquests serveis.

Tal i com comentava Chris Aniszczyk [11] a twitter tenen un seguit de clústers corrent mesOS [12] ja que van veure que era molt potent i en aquell moment no hi havia rés igual.

Amb el temps però han sortit altres sistemes, com CoreOs, que tot i no estar en producció i garantir un servei encara del tot estable i complert si que ofereix avantatges com:

- Una estructura molt lleugera i descentralitzada.
- Fàcilment escalable.
- Tolerància alta a fallades de nodes.
- Sistema innovador a l'hora d'actualitzar versions.
- Open source.
- Utilitza docker per defecte.
- Nou paradigma OSaaS que ens dona més eficiència.

Aquests avantatges ens han fet decantar-nos per a CoreOs a l'hora de realitzar la investigació i la implantació d'aplicacions.

Ara que ja hem vist les diverses opcions , al següent apartat aprofundirem més en el funcionament intern d'aquesta tecnologia i de les diverses eines que utilitza.

3. CoreOS i Docker

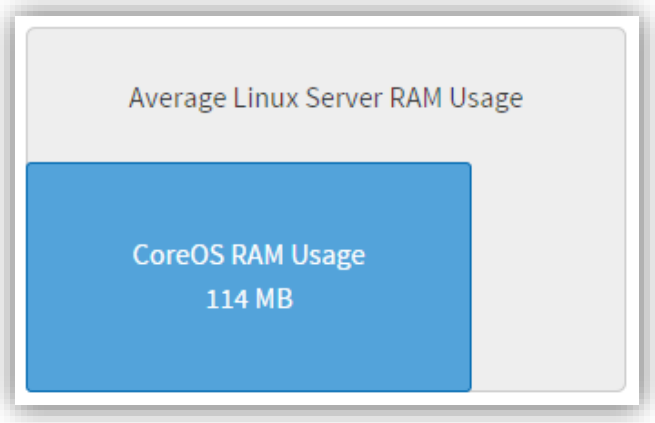
Una vegada hem vist l'estat actual i perquè hem escollit aquest sistema passarem a analitzar el sistema en si. Si parlem de CoreOs hem de parlar també de Docker (de moment[13]) ja que no pot funcionar sense aquesta tecnologia.
Primer doncs parlarem del Sistema CoreOs i després veurem Docker.

3.1. CoreOs

Tal i com he mencionat anteriorment CoreOs és un sistema operatiu dissenyat per a ser escalable i distribuït. Pensat específicament per a treballar en clúster utilitza tecnologia molt diversa (fleet, etcd, btrfs, docker) per a mantenir un clúster operatiu tolerant a fallades i consistent.
Basat en el kernel de linux, concretament en gentoo, és un sistema amb molt poques funcionalitats inicials ja que parteix de la base de ser molt lleuger. És un sistema amb llicencia apache 2.0

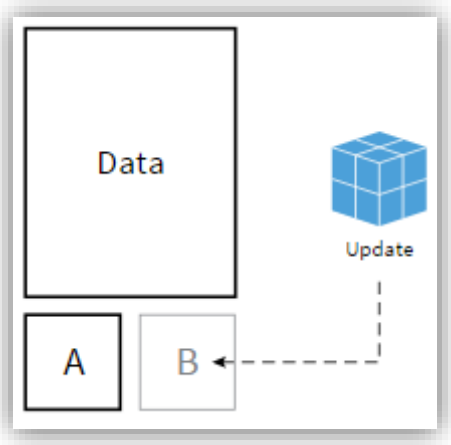
Originalment és un fork de ChromeOs i la primera release que es va fer és del 3 d'octubre del 2013. Com podem comprovar per tant és un sistema bastant nou amb poc temps al mercat i que encara es troba en una intensa fase de desenvolupament.

Al tractar-se d'un sistema tan senzill no proporciona pas entorn gràfic ni gestor de paquets. Per a fer servir les aplicacions el que utilitza és una virtualització a nivell de kernel per a fer funcionar els serveis. El fet de que sigui un servei tan senzill fa que no consumeixi gran quantitat de recursos.



Imatge 13: Comparativa d'utilització de memòria

També disposa d'un sistema de doble partició per a actualitzacions del sistema operatiu. Aquest sistema permet fer les actualitzacions de cop i de manera segura ja que al fer-ho en dos passos sempre es pot fer un rollback a l'estat anterior si sorgeix algun inconvenient.



Imatge 14: Estructura d'actualitzacions

Tal i com he mencionat anteriorment utilitza contenidors docker per a fer funcionar els serveis. El sistema d'assignació d'aquests contenidors és fleet.

A continuació aprofundirem en les eines i tecnologies que el sistema utilitza per a funcionar.

3.1.1 Etcd

Una tecnologia que utilitza CoreOS és etcd un servei distribuït d'emmagatzematge de parelles (claus, valors). Específicament el sistema l'utilitza per a compartir configuració i serveis de descobriment. CoreOs utilitza etcd ja que disposa d'una altra eina (fleet) que funciona sobre aquest servei. Etcd està escrit en Go i sota llicència apache 2.0.

Imatge 15: Logo de etcd

Etcd es basa en ser:

- Simple : ja que s'utilitza a través de url amb http+json.
- Segur : permet utilitzar certificats ssl .
- Ràpid : té una escriptura i lectura molt ràpides.
- Fiable :ja que utilitza l'algorisme de raft concensus per a tractar amb els sistemes distribuïts.

CoreOs en concret utilitza etcdctl, el qual és una versió per línia de comandes d'aquest sistema. Utilitzem aquesta tecnologia bàsicament per a mantenir paràmetres de configuració entre els diversos nodes del nostre clúster.
L'eina etcdctl ens permet escriure claus amb valors, obtenir-les, modificar-les i esborrar-les de manera totalment distribuïda i mantenint fiabilitat. Per aconseguir això utilitza l'algorisme de Raft Concensus per a gestionar el

sistema de claus i valor. Aquest algorisme ens descriu que la informació de cada node ha d'estar replicada a tots els altres nodes i es canvia mitjançant un sistema de consens que explicarem al següent apartat.

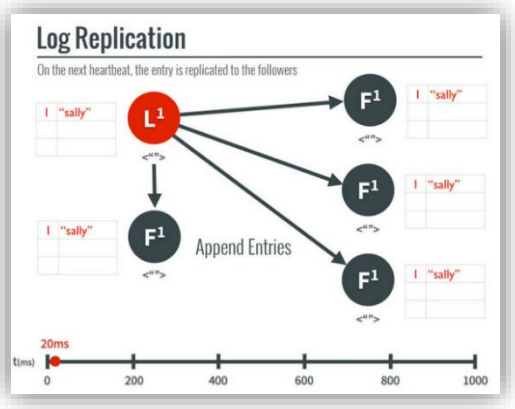
3.1.1.1 Raft consensus

Tal i com he mencionat anteriorment la tecnologia de etcd es basa en aquest algorisme, el qual està dissenyat per ser una alternativa a Paxos [14]. Aquest algorisme intenta resoldre el problema de fiabilitat i tolerància a fallades dels sistemes distribuïts.
Es basa en que cada node té una màquina d'estats i un seguit de logs. La màquina d'estats és el que volem fer tolerant a fallades i els logs els canvis que hem de fer. Quan volem fer un canvi del log hem de posar-nos d'acord mitjançant consens per a realitzar aquest canvi. Si s'aprova tots els nodes realitzen aquest canvi i l'algorisme s'ha d'assegurar que totes les màquines l'apliquen i per tant acaben amb el mateix estat.

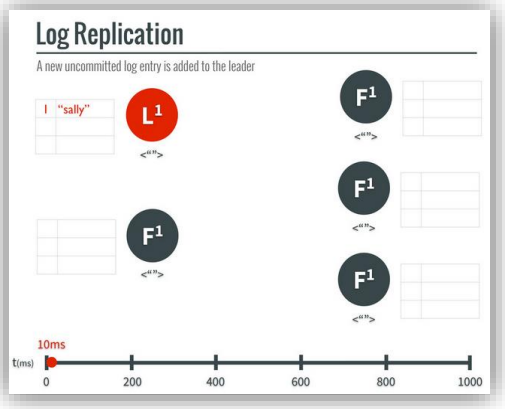
Així obtenim un conjunt de màquines que mantenen el mateix resultat i els mateixos estats. Al tractar-se d'un algorisme de consens necessitem una majoria de nodes per tal de poder mantenir consistència de les dades. Per exemple amb 5 nodes si en perdem 3 el sistema ja no pot continuar funcionant correctament.

Un exemple més pràctic seria en la replicació de registres. Suposem que tenim un conjunt de 5 nodes sense cap líder, tots aquests son candidats i proposen als altres nodes que votin per a ells. A mesura que passa el temps es realitzen votacions, si s'arriba a un consens s'elegeix aquest node com a líder i la resta com a seguidors. Si no s'arriba a un consens es segueixen fent votacions, i per a que no entrin en un deadlock cada node té un rellotge intern que amb un interval aleatori torna a proposar la seva candidatura a la resta de nodes.

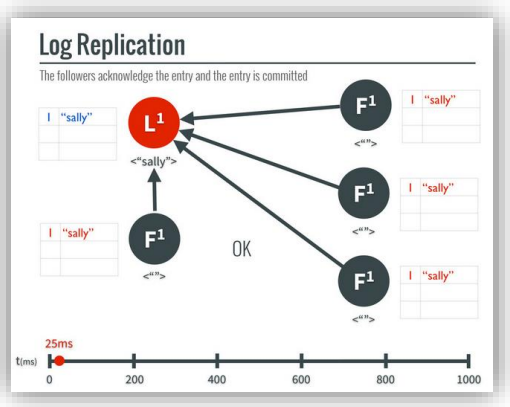
Una vegada tenim el node principal ,que vol canviar un valor, (imatge 16.1) el proposa a la resta de nodes (imatge 16.2), aquests responen al node conforme estan d'acord amb l'assignació (imatge 16.3) i finalment es fa efectiu el canvi (imatge 16.4). Si algun dels nodes no realitza el canvi tots els altres tornen a l'estat anterior i no realitzen el canvi.



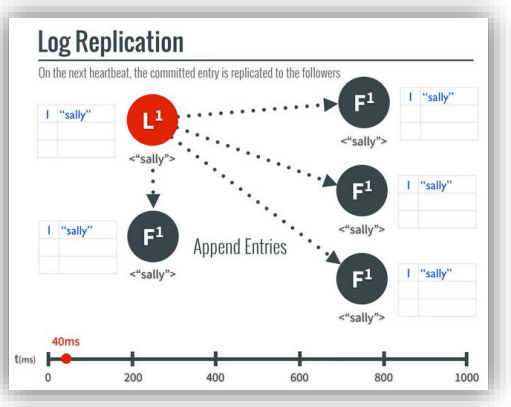
Imatge 16.1: Pas 1 de l'algorisme



Imatge 16.2: Pas 2 de l'algorisme



Imatge 16.3: Pas 3 de l'algorisme



Imatge 16.4: Pas 4 de l'algorisme

En cas de que el es produeixi una segmentació de la xarxa els nodes segueixen els seus líders fins a que es torni a unir. En aquell moment el node que és elegit líder força els altres nodes, els que han tornat a la xarxa, a canviar el seu registre.

L'explicació més visual d'això es pot trobar en aquesta presentació[15] de la qual he extret les imatges anteriors.

3.1.2 Systemd

És un conjunt de programes, llibreries i utilitats dissenyades per a gestionar un sistema operatiu linux. Systemd reemplaça l'antic init [16] de Linux i ens permet gestionar i configurar el sistema. A dia d'avui la majoria de distribucions de linux ja l'han adoptat, CoreOs no és una excepció.

Per a gestionar un node localment o veure el seu estat des d'una connexió ssh podem utilitzar systemd. Es tan simple com fer un systemctl –help per a obtenir la llista de comandes. També podem crear serveis amb una nomenclatura i estructura que més endavant a l'apartat de fleet explicarem.
Cal mencionar però que aquest funcionament no és l'adequat a l'hora de gestionar els processos del clúster ja que disposem d'una eina que ens permet visualitzar millor l'estat d'aquests sense necessitat d'estar al mateix node. Aquesta eina s'anomena fleet i l'explicarem a continuació

3.1.3 Fleet

Tal i com he dit en l'apartat anterior es pot obtenir informació d'una màquina o d'un servei mitjançant systemd, però no és la manera optima ja que ens hem de connectar a la màquina per un canal addicional. En canvi si utilitzem fleet podem realitzar les mateixes comandes des d'un node.
Fleet ens permet,a grans trets, utilitzar systemd sobre tot el clúster. Si volem saber l'estat d'una màquina, engegar un servei o veure l'estat del clúster

podem utilitzar aquesta eina.
Construït sobre systemd i etcd, fleet corre en background (com a dimoni) **a cada node** del clúster.

Fleet està format per dues parts un Motor (engine) i un agent, tota la comunicació que fan els agents amb els motors es fa a través de etcd.

- Engine : el motor és el responsable d'organitzar i prendre decisions al clúster, per a fer això utilitza un bucle que periòdicament s'activa a través d'alguns events de etcd. El sistema de fleet té en compte que no hi hagi funcionant més d'un engine a la vegada dins el mateix clúster.
  L'estructura que segueix és la següent:
  1- l'engine realitza un snapshot de l'estat general del clúster tan unitats com agents.
  2- veient l'estat actual intenta reconciliar aquest amb l'estat desitjat per fer això, intenta obtenir el lideratge del clúster.
  3- es realitza una elecció en la que es proposa aquest engine
  4- si aquesta es realitza amb èxit la reconciliació es duu a terme, sinó l'engine entra en un estat de standby fins que la següent elecció comenci.
  Per a assignar unitats l'engine es basa en un model senzill de buscar els agents menys carregats.

- Agent: l'agent és el responsable d'executar les unitats al sistema. Utilitza D-Bus [17] per a comunicar-se amb la instància local de systemd.
  Com l'engine l'agent també utilitza un bucle periòdicament per a determinar què ha de fer, fent un snapshot de etcd i realitzant els canvis necessaris per arribar a l'estat que allí es reflecteix. També és el responsable de transmetre l'estat de les seves unitats al etcd.

Fleet també utilitza etcd per a guardar totes les dades tan persistents com momentànies que utilitza. També fa servir uns objectes : els Units (o unitats de processament) i estats:

- Units : representen un conjunt de propietats de configuració del servei que es vol executar. Una vegada posats al clúster no es poden modificar (nomes en el flag de estat desitjat). Una unitat també pot especificar un seguit de requeriments que ha de complir el node per a poder-lo executar. Les unitats són tractades com a un servei (no com un dimoni), si alguna cau, fleet s'encarrega de reubicar-la en una altra màquina.
- Estats: Tan les màquines com els fitxers Unit tenen un estat dinàmic el qual és publicat al clúster.

L'estructura a l'hora de declarar una unitat en fleet no difereix molt de la manera de declarar una unitat amb systemd, com podem veure amb l'exemple següent.

```
[Unit]
Description=subgun

[Service]
ExecStartPre=-/usr/bin/docker kill subgun-1
ExecStartPre=-/usr/bin/docker rm subgun-1
ExecStart=/usr/bin/docker run -rm -name subgun-1 -e SUBGUN_LISTEN=127.0.0.1:8080 -e SUBGUN_ ...
ExecStop=/usr/bin/docker kill subgun-1

[X-Fleet]
X-Conflicts=subgun-http.*.service
```

Imatge 17: Exemple de servei en fleet

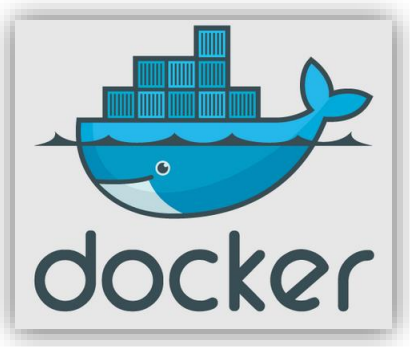
Si hem treballat amb systemd veurem que la única cosa que té diferent és el tag del final ([X-Fleet]). L'estructura de configuració d'aquests serveis la veurem més endavant quan parlarem de la estructura proposada de software.

3.1.4 Btrfs

El sistema de fitxers que utilitza CoreOs, és un sistema de fitxers COW [18] (copy-on-write). Copia en escriure ens defineix un sistema el qual es tracten les dades com unes dades molt estàtiques i que són modificades ocasionalment. Així doncs es donen punters al mateix recurs als diversos processos que vulguin treballar, tenint en compte que si volen modificar les dades primer s'han de fer una còpia en local, modificar-la i després modificar-la a l'origen.
El sistema que utilitza CoreOs concretament s'anomena Btrfs (B-tree file system) i és especialment útil ja que realitza un seguit de snapshots i checksums de les dades en diversos moments per a mantenir consistència i detectar possibles errors. Aquest tipus de sistema de fitxers ens ofereix avantatges tan grans com la reparació automàtica en alguns casos [19].

3.2 Docker

Com ja hem mencionat anteriorment docker és un projecte open-source que utilitza l'aïllament de recursos del kernel de linux per a permetre que diversos contenidors pugin funcionar en una mateixa instància de sistema.
A manera pràctica el que fa és virtualitzar a nivell molt baix, per això està construït a sobre de libvirt i LXC. Està escrit en Go. El que aconseguim amb aquest sistema és crear un contenidor amb la nostra aplicació i poder-lo executar en qualsevol sistema que suporti docker sense preocupar-nos de les dependències del sistema destí.



Imatge 18: Logo de docker

Aquest sistema ens dóna gran llibertat i portabilitat a l'hora d'executar les nostres aplicacions en servidors amb entorn linux. Docker disposa d'una plataforma anomenada Docker Hub en la que hi han un seguit d'imatges de docker per a diversos serveis. Per a utilitzar aquestes imatges és tan senzill com fer un pull de la imatge i un run després.

Si no trobem la imatge en qüestió al repositori també podem crear-nos la nostra pròpia imatge de manera molt simple. A continuació mostrem un exemple.

```
# Set the base image to Ubuntu
FROM ubuntu
MAINTAINER nomAutor
RUN apt-get update

# Install MongoDB Following the Instructions at MongoDB Docs
RUN apt-key adv --keyserver hkp://keyserver.ubuntu.com:80 --recv 7F0CEB10
RUN echo 'deb http://downloads-distro.mongodb.org/repo/ubuntu-upstart dist 10gen'
| tee /etc/apt/sources.list.d/mongodb.list
RUN apt-get update
RUN apt-get install -y mongodb-10gen
RUN mkdir -p /data/db

# Expose the default port
EXPOSE 27017

# Default port to execute the entrypoint (MongoDB)
CMD ["--port 27017"]

# Set default container command
ENTRYPOINT usr/bin/mongod
```

Imatge 19: Exemple de creacio d'una imatge de docker

El que fem en aquest fitxer és crear una imatge de mongoDB. En concret agafem un sistema de base, en aquest cas ubuntu, i realitzem un seguit de comandes de configuració i instal·lem mongoDB. Una vegada tenim fet aquest fitxer utilitzem la comanda build per a construir la imatge i posteriorment podem utilitzar la comanda run per a crear un contenidor corrent aquesta imatge.

Ara que ja hem vist les eines que farem servir i com funcionen ja podem passar a veure quins són els requeriments per a una aplicació cloud. En el següent apartat indagarem en aquests requeriments i veurem com es comparen a l'estructura que utilitzem ara.

4.Requeriments per una aplicació Cloud

A l'apartat anterior hem vist com funciona la tecnologia que farem servir, ara doncs ens falta veure com hem d'estructurar les aplicacions que hem de fer funcionar a sobre.

Com hem vist al primer apartat tot el tema del cloud està evolucionant i han sorgit diversos paradigmes i diverses terminologies, una d'elles és SaaS (Software-as-a-service). Aquest terme relativament nou ha generat diversos paradigmes de programació i varies metodologies associades.

Després de veure'n varies, la que més s'aproxima amb els requeriments per una aplicació d'aquest estil és el Twelve-factor app [20] escrit per Adam Wiggins, el creador de heroku[28] .

Tal i com existeixen els 12 principis d'Agile [21] d'una manera similar aquest dissenyador de software alemany ha escrit els 12 requeriments essencials i indispensables que hauria de tenir una aplicació cloud.

Els principis són :

1-Code base (base per al codi) : Aquest primer principi ens diu que hem d'utilitzar un sistema per aportar un control sobre el codi, les modificacions i actualitzacions que aquest tingui. També fa èmfasis en que un repositori ha de servir per una aplicació i només una. Comenta el fet de que si es vol compartir repositoris s'ha de fer a través de llibreries.

2- Dependencies (dependències): En aquest segon punt ens força a utilitzar algun sistema per a aïllar les dependències del projecte i unificar-les sobre un mateix paquet. Això ens dona el benefici de que l'entorn és fàcilment configurable i actualitzable amb les diverses dependències. També ens facilita la feina si volem realitzar testos unitaris del codi. En el nostre cas aquesta separació és molt fàcil d'aconseguir i només ens cal fer funcionar les diverses dependències de la nostra aplicació amb varis contenidors.

3- Config (configuració): En aquest punt ens comenta els beneficis de tenir una manera unificada de guardar la configuració, credencials i valors o adreces de manera separada del codi. En el nostre cas però trobem que guardar variables d'entorn no és el que més s'adequa al nostre sistema.

4- Backing Services (serveis de guardar dades com a recursos): En aquest punt comenta la necessitat de fer suficient modular la connexió amb altres serveis externs per a poder-los modificar/ canviar sense tocar el codi. També es desentén de serveis que requereixen persistència de dades dient que ja les hauríem de tenir com a un servei i no pas gestionar-les des de l'aplicació. En el nostre cas es podria adaptar perfectament afegint un servei de base de dades al servei web.

5- Build, release,run (separar les tres fases) : En aquest punt ens parla de la importància de separar estrictament les tres fases del desenvolupament del software :

build : es produeix quan convertim el codi en un conjunt executable en el nostre cas seria quan construïm un contenidor.

release: es produeix quan agafem la build i la combinem amb la configuració que tenim, així ho deixem llest per a executar-se. Aquí nosaltres el que faríem es escriure un servei de fleet per a poder configurar l'aplicació i llençar-la.

run: és la part en que fem córrer la app en un entorn determinat. En aquest moment és quan executem el servei que ens crea una instància de l'aplicació al clúster.

També menciona la importància d'identificar les release amb un identificador únic. Per això ens és molt útil utilitzar un repositori de contenidors com dockehub.

6-Processes (processos): En aquest apartat menciona el fet de que l'aplicació s'hauria d'executar com a un o més processos sense cap estat i entre ells no intercanvien informació. Si aquests processos volen guardar informació han d'utilitzar un backing service (vist anteriorment).

7-Port binding (enllaç amb un port): En aquest punt ens diu que les aplicacions han d'estar connectades a un port. No les hem de injectar en moment de execució, com faríem amb una aplicació de java sobre un Tomcat. Ens menciona que la via per assolir això és la declaració de dependències com hem vist anteriorment. Apunta també que la majoria de les aplicacions que ofereixen un servei han de funcionar així . Això ens permet mantenir el que parlàvem al punt 4. En el nostre cas la majoria de serveis que utilitzem fan servir aquesta metodologia.

8-Concurrency (Concurrència): en aquest apartat ens menciona la importància de que el programador pugui tenir opinió en com s'ha d'executar el procés en diversos entorns. Per exemple si tenim un servei que rep peticions web, en comptes de deixar a una JVM que li proporcioni tot el programador hauria de ser capaç de permetre que el procés pugi executar altres treballadors en background per a no sobrecarregar-se. Així també aconseguirem més independència de les aplicacions, cosa que ens permetrà que siguin més escalables. Docker en aquest punt ens facilita molt la feina ja que permet utilitzar des d'un contenidor un altre contenidor diferent.

9- Disposability (Capacitat d'eliminació): en aquest punt ens parla de com els processos han de ser eliminables, de manera que puguin iniciar-se o parar-se quan convingui per a maximitzar l'escalabilitat. També diu que s'ha de minimitzar el temps d'inici d'un procés ja que això contribueix a afegir escalabilitat i solidesa al sistema. Menciona també que és important que els processos no s'aturin immediatament al rebre una senyal d'aturada, primer han

d'acabar el que estan fent i posteriorment aturar-se. Recalca també que els processos han d'estar preparats per a caigudes instantànies per causes externes. En el nostre cas això s'adapta a la perfecció amb la estructura que tenim muntada de clúster i la capacitat per a tombar serveis i tornar-los a aixecar.

10- Dev/prod parity: en aquest punt es posiciona a favor del deploy continuat un patró de disseny que es basa en pujar codi contínuament per a obtenir ràpidament feedback i poder fer modificacions. També ens anima a tenir el mateix entorn de producció i de desenvolupament per a que no es puguin produir problemes a l'hora de realitzar canvis. Un dels avantatges que les empreses que utilitzen docker mencionen és la senzillesa amb la que es pot aplicar el patró de deploy continuat.

11-logs: en aquesta part ens especifica que els logs del sistema no els hem de tractar, els hem de llençar per un canal i deixar que una aplicació externa els tracti. En el nostre cas també seria senzillament assolible construint un contenidor que s'encarregués d'això.

12-admin processes: finalment en aquest punt ens parla de la importància de posar processos administratius dins de l'aplicació que publiquem per a que si s'ha de fer alguna gestió un desenvolupador pugi accedir de forma remota i arreglar problemes. Nosaltres no hem contemplat aquesta opció però és fàcilment adaptable construint un contenidor que tingui aquest objectiu.

Tots aquests principis són una bona base per a fer funcionar aplicacions distribuïdes com podem veure en la majoria s'intenta limitar molt la funcionalitat de cada procés precisament per a que sigui molt escalable i fàcilment reproduïble. També veiem que es tendeix a treballar més sobre una arquitectura agile en que es prioritza tenir una part funcional que anirem actualitzant a mesura que anem desenvolupant intentant mantenir el servei inactiu el menor temps possible.

En la nostra implementació ens hem pres el conjunt de principis com una estructura a seguir en el cas de una aplicació funcional relativament gran i per tant no l'hem aplicat sencer als petits exemples que hem realitzat (precisament un dels exemples és una base de dades el que incompliria el punt 4). Malgrat això creiem necessari mencionar-los com a una estructura a analitzar i extreure les necessitats per a cada aplicació en particular.

Una vegada vistos aquests requisits passarem, al següent apartat, a veure la estructura que recomanem i que hem utilitzat a l'hora de fer funcionar l'aplicació.

5.Arquitectura proposada

Després d'analitzar la tecnologia i veure els requeriments de les aplicacions que han de funcionar sobre aquest sistema, a continuació veurem l'arquitectura amb la que hem treballat i l'arquitectura proposada per a un sistema distribuït amb CoreOs.

5.1 Arquitectura de sistema

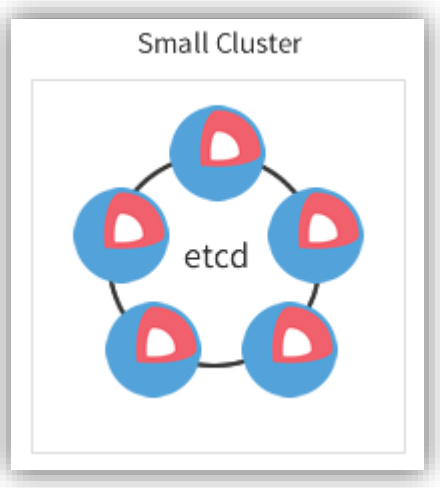
Per tal de poder realitzar una avaluació del sistema de CoreOs hem utilitzat una estructura de clúster relativament petita formada per 5 nodes en el cas més gran.

Aquesta estructura és ideal per entorns de desenvolupament i està formada per varis nodes iguals. Es tracta d'una estructura molt fàcil i ràpida de muntar.

Al tractar-se de un clúster petit cada node té una instància de etcd corrent en local que es comunica amb els altres nodes del clúster.

A diferència d'altres sistemes en aquesta estructura reduïda no necessitem realitzar una configuració específica per a cada node del clúster, ja que amb un parell de fitxers de configuració comuns podem configurar tot el clúster.

En el nostre cas per a avaluar el sistema hem treballat sobre vagrant per a gestionar un seguit d'instàncies de virtual box de manera més senzilla, a l'apartat Annex veurem com fer servir vagrant i com aixecar el sistema.



Imatge 20: Esquema del nostre clúster

A continuació veurem els dos fitxers necessaris per tal de configurar el clúster de CoreOs en una màquina virtual. Així també veurem els serveis principals que farem servir sobre el clúster ja només d'entrada.

Els dos fitxers de configuració són el config.rb i el user-data. Cal mencionar que aquests fitxers són únicament per a treballar en un entorn de màquina virtual ja sobre altres plataformes **només necessitarem** el cloud-config.

Cloud-config és un fitxer escrit en YAML[22] que és interpretat pel programa coreos-cloudinit el qual utilitza aquests fitxers per a configurar el sistema a l'inici. El fitxer ens permet modificar vàries coses del sistema en diversos nivells des de la configuració de la xarxa fins als usuaris i a les unitats systemd. A cada arrancada del clúster aquest programa processa aquest fitxer, si tenim un fitxer invàlid el programa no el processarà però si que guardarà un log amb l'error.

Tal i com podem veure pel nom que té (coreos-cloudinit) el programa està inspirat en el projecte cloud-config [23] de cloud-init.

El fitxer que utilitza vagrant per a configurar les diverses màquines i per a veure l'estructura del clúster és el config.rb. En aquest fitxer tal i com podem veure a continuació es defineixen el nombre de nodes del clúster, el tipus de versió que volem carregar a cada sistema (estable, beta, alfa) i els atributs de les màquines virtuals.

```
# Size of the CoreOS cluster created by Vagrant
$num_instances=5

# Official CoreOS channel from which updates should be downloaded
$update_channel='stable'

# Setting for VirtualBox VMs
$vb_gui = false
$vb_memory = 2024
#$vb_cpus = 1
```

Imatge 21: Fitxer config.rb

L'altre fitxer és user-config, aquest és el fitxer cloud-config que utilitza el sistema i en el que es declaren un seguit de paràmetres i serveis. A continuació podem veure un exemple d'aquest fitxer i analitzar-lo més profundament.

```
#cloud-config

coreos:
  etcd:
    # generate a new token for each unique cluster from https://discovery.etcd.io/new
    # WARNING: replace each time you 'vagrant destroy'
    discovery: https://discovery.etcd.io/201bc55b98ef20b07964b14ae76b8b9f
    addr: $public_ipv4:4001
    peer-addr: $public_ipv4:7001
  fleet:
    public-ip: $public_ipv4
  units:
    - name: etcd.service
      command: start
    - name: fleet.service
      command: start
    - name: docker-tcp.socket
      command: start
      enable: true
      content: |
        [Unit]
        Description=Docker Socket for the API

        [Socket]
        ListenStream=2375
        Service=docker.service
        BindIPv6Only=both

        [Install]
        WantedBy=sockets.target
```

Imatge 22: Fitxer user-data

En YAML definim un seguit de paràmetres com faríem amb python, utilitzant espais i salts de línia. Aquí podem veure que en el tag central de coreos pengen 3 tags secundaris. Tenim els paràmetres de configuració del servei de etcd, els de fleet i els de les unitats o processos. Podem veure que les ip's del sistema són afegides per paràmetre en comptes de directament, això és degut a que alguns sistemes(EC2, OpenStack, Rackspace) suporten aquesta manera.

etcd: tenim diversos paràmetres, el primer ens mostra la url utilitzada per a guardar un seguit de valors de configuració; en el segon ens mostra l'adreça pública amb el port per a la comunicació amb el client; finalment l'últim ens mostra el port i adreça on s'adreçarà amb el servidor.

fleet: el fleet necessita l'adreça en la que es publicarà l'estat de la màquina i la informació del seus sockets.

units: aquest paràmetre com podem observar ens defineix un seguit de serveis que s'iniciaran només engegar el node. En el nostre cas tenim 3 serveis el de fleet, el de etcd i el de docker. Degut a que els dos primers ja tenen una configuració específica no necessiten gaires paràmetres a diferència del servei de docker.

Malgrat hem parlat només d'un clúster virtual petit la construcció és pot aplicar a un clúster gran canviant un parell de coses. Hem de tenir en compte que en un clúster gran necessitaríem un sub-clúster de etcd ja que l'algorisme de Raft amb una quantitat molt gran de nodes es molt lent. Així doncs els nodes no tindrien un etcd en local sinó que referenciarien un clúster. L'altra cosa que hem de canviar és el fitxer de configuració ja que només necessitarem un cloud-config.

5.2 Arquitectura d'aplicació

A l'apartat anterior hem estat muntant el sistema i ara ja disposem d'un clúster de CoreOs de 5 nodes. Una vegada tenim aquest clúster l'hi hem de treure tot el profit possible i per a realitzar això a continuació veurem com crear els serveis o processos que funcionaran sobre el nostre clúster.

L'estructura genèrica que hem de seguir a l'hora de crear un servei distribuït a través del clúster es basa en tenir dos fitxers un fitxer de servei i un fitxer de presentació o descobriment.

Abans de veure això però hem de conèixer quin llenguatge o estructura utilitzarem per a aquests serveis. Com ja hem vist anteriorment CoreOs utilitza systemd per a gestionar el conjunt de processos, si volem per tant crear un servei d'aquest tipus com que l'hem d'engegar amb fleet necessitarem conèixer l'estructura d'aquests fitxers.

5.2.1 Systemd services estructura

Tot i que no és necessari que disposin de tots els apartats l'esquema que emprarem nosaltres és el que mostrem a l'imatge.

```
[Unit]
Description=MyApp
After=docker.service
Requires=docker.service

[Service]
TimeoutStartSec=0
ExecStartPre=-/usr/bin/docker kill busybox1
ExecStartPre=-/usr/bin/docker rm busybox1
ExecStartPre=/usr/bin/docker pull busybox
ExecStart=/usr/bin/docker run --name busybox1 busybox /bin/sh -c "while true;
do echo Hello World; sleep 1; done"

[Install]
WantedBy=multi-user.target
```

Imatge 23: Servei de systemd

Podem veure diversos apartats a l'estructura anterior, no aprofundirem molt en la funcionalitat de cada un d'ells ja que no és l'objectiu d'aquest projecte. Tenim però la part de Unit que s'encarrega de descriure quin tipus de servei és i pot especificar si està lligat a un altre servei o no, també pot especificar si va abans o després. L'apartat de Service ens permet especificar un seguit de comandes que s'executaràn abans (ExecStartPre),en el moment d'execució del servei (ExecStart) i una vegada aquest s'aturi (ExcStop). A l'últim apartat ens permet especificar el sistema l'objectiu d'aquest servei.

Aquesta és la estructura bàsica, si volem fer funcionar un servei en local en un dels nodes ho podem fer però per a les aplicacions distribuïdes que volem executar necessitarem utilitzar l'estructura de fleet, la qual és exactament igual en tots els apartats menys en l'últim, com ja hem vist anteriorment. A continuació passarem a veure aquesta estructura.

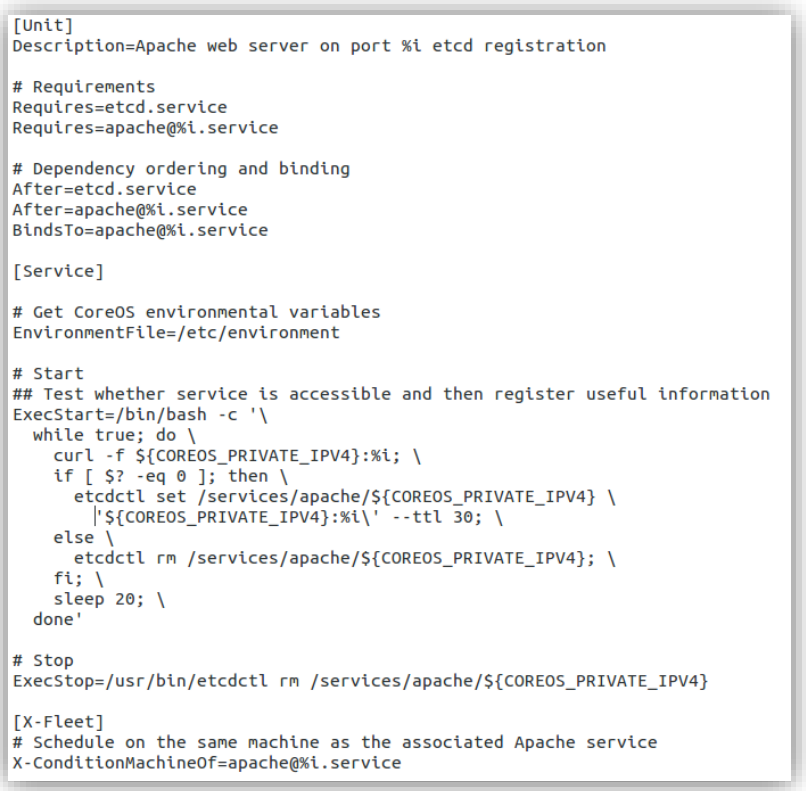
5.2.1 Service

L'estructura del nostre servei es divideix en dos processos o fitxers, tenim un procés que s'encarrega de crear un registre a la instància de etcd i un altre procés que s'encarrega d'engegar el sistema.

5.2.1.1 Service discovery

Aquest segon procés de descobriment o discovery bàsicament el que fa és registrar una clau etcdctl amb un temps de vida determinat. També realitza un bucle en que va comprovant si el servei està actiu i en bon estat per mantenir la clau o treure-la.

A la següent imatge podem veure un exemple amb un servei d'apache.



```
[Unit]
Description=Apache web server on port %i etcd registration

# Requirements
Requires=etcd.service
Requires=apache@%i.service

# Dependency ordering and binding
After=etcd.service
After=apache@%i.service
BindsTo=apache@%i.service

[Service]

# Get CoreOS environmental variables
EnvironmentFile=/etc/environment

# Start
## Test whether service is accessible and then register useful information
ExecStart=/bin/bash -c '\
  while true; do \
    curl -f ${COREOS_PRIVATE_IPV4}:%i; \
    if [ $? -eq 0 ]; then \
      etcdctl set /services/apache/${COREOS_PRIVATE_IPV4} \
        '${COREOS_PRIVATE_IPV4}:%i\' --ttl 30; \
    else \
      etcdctl rm /services/apache/${COREOS_PRIVATE_IPV4}; \
    fi; \
    sleep 20; \
  done'

# Stop
ExecStop=/usr/bin/etcdctl rm /services/apache/${COREOS_PRIVATE_IPV4}

[X-Fleet]
# Schedule on the same machine as the associated Apache service
X-ConditionMachineOf=apache@%i.service
```

Imatge 24: Servei de discovery d'apache

És necessari que en l'apartat de descriure el servei (Unit) definim que necessita del servei en si per a funcionar. També es preferible que definim a l'apartat de fleet (X-feet) que el servei ha d'executar-se a la mateixa màquina que l'altre servei.

5.2.1.2 Service

Abans hem vist el procés que s'encarregava de publicar el servei a l'etcd ara veurem l'altre procés que s'encarrega de fer funcionar el servei. En si el que fa el servei és crear un contenidor de docker a partir d'una imatge i executar el que hi hagi a dins.

Tal i com podem veure a la imatge 25, el que fa el servei quan s'inicia és mirar si hi ha algun altre servei com ell per eliminar-lo i destruir-lo. Una vegada ha fet això realitza un pull de la imatge de docker que posteriorment executarà. Després inicia el servei de docker amb un nom determinat, en una ubicació determinada i amb una imatge específica; en aquest cas també li demanem que s'executi en background.

Per a poder executar mes d'una instància d'un servei en una mateixa màquina (o en una diferent) podem assignar-li diversos ports sense necessitat de modificar el fitxer original, només cal que el nom del fitxer sigui tipusDeServei@.service. Així quan creem la instància dins del fitxer podem recuperar el valor amb %i. En diversos casos el que es fa és canviar el port per algun tipus de servei (un apache per exemple) i per tant es crida al servei amb

4 números (del port) els quals dins del servei fem servir per assignar la localització d'aquest.

```
[Unit]
Description=Apache web server service on port %i

# Requirements
Requires=etcd.service
Requires=docker.service
Requires=apache-discovery@%i.service

# Dependency ordering
After=etcd.service
After=docker.service
Before=apache-discovery@%i.service

[Service]
# Let processes take awhile to start up (for first run Docker containers)
TimeoutStartSec=0

# Change killmode from "control-group" to "none" to let Docker remove
# work correctly.
KillMode=none

# Get CoreOS environmental variables
EnvironmentFile=/etc/environment

# Pre-start and Start
## Directives with "=-" are allowed to fail without consequence
ExecStartPre=-/usr/bin/docker kill apache.%i
ExecStartPre=-/usr/bin/docker rm apache.%i
ExecStartPre=/usr/bin/docker pull hirocat/apache
ExecStart=/usr/bin/docker run --name apache.%i -p ${COREOS_PRIVATE_IPV4}:%i:80 \
hirocat/apache /usr/sbin/apache2ctl -D FOREGROUND

# Stop
ExecStop=/usr/bin/docker stop apache.%i

[X-Fleet]
# Don't schedule on the same machine as other Apache instances
X-Conflicts=apache@*.service
```

Imatge 25: Servei d'apache

A l'apartat x-Feet del final del fitxer podem especificar si volem que funcioni en una màquina específica o si volem permetre o no que hi hagin múltiples serveis iguals corrent en una mateixa màquina.

6. Exemples proposats

Ara que ja hem vist l'estructura que tenen aquest serveis passarem a veure els exemples que hem implantat sobre el nostre clúster i avaluarem el seu funcionament. El primer d'aquests exemples i potser el més senzill és un servei web, el segon que veurem és un servei de base de dades.

6.1 Web service

La primera opció que hem provat és d'implantar un servei d'apache, a continuació veurem com s'implementa i posteriorment farem un anàlisis dels resultats obtinguts durant les proves realitzades

6.1.1 Apache

6.1.1.1 Anàlisis del servei

Com ja hem vist anteriorment,en l'apartat anterior quan parlàvem de les estructures genèriques dels serveis, aquest servei també consisteix de dos serveis un servei de descobriment i un servei per executar el servei en si.

A continuació veurem els dos serveis. Primer de tot tenim el servei de descobriment, el nom del fitxer del qual és apache-discovery@.service (Imatge 24). Com a l'estructura que hem vist abans aquest fitxer consta d'una part inicial en la que es descriu el nom del fitxer i es posen un seguit de requisits. En el nostre cas demanem com a requisits que hi hagi un servei de etcd funcionant (recordem que l'objectiu del nostre servei és modificar els valors d'aquest servei) que hi hagi un servei d'apache funcionant (el servei principal al que volem descobrir) i que s'uneixi al servei d'apache funcionant. Una vegada fet això passem a la part de service en la que primer obtenim les variables d'entorn de CoreOs i posteriorment executem un seguit de paràmetres. L'estructura de paràmetres és una estructura bastant genèrica utilitzada per la majoria d'aplicacions. Com ja hem comentat abans el que fa l'estructura és controlar que el servei estigui actiu i guardar la direcció etcd, quan el servei cau o canvia de node s'actualitza aquesta clau. Quan el servei és destruït s'elimina la clau.

Finalment també s'utilitza un paràmetre de configuració de fleet per a demanar que el procés s'executi a la mateixa màquina que el servei al que descobreix.

Una vegada vist el servei de descobriment passarem a veure el servei en si (Imatge 25).

L'estructura que segueix també es molt similar al servei de descobriment ja que al principi demanem un servei de etcd, un servei de descobriment d'apache i també demanem un servei de docker.

Una vegada fet això passem a fer un pull del contenidor (que en aquest cas és un contenidor creat per nosaltres des del repositori de docker) i seguidament fem un run del servei.

A la part de fleet especifiquem que no hi pot haver un altre instància del mateix servei funcionant al mateix node.

6.1.1.2 Testeig

De moment ja tenim els dos fitxers, el que ens falta ara és fer un deploy de la nostra aplicació al clúster. Per a fer un deploy del servei hem de dir-li a fleet a través de fleetctl start que creï una instància (o vàries).

```
core@core-01 ~ $ ls
apache-discovery@.service  apache@.service  instances  share  static  templates
core@core-01 ~ $ fleetctl start apache-discovery@7777 apache@7777
Unit apache@7777.service launched on 151bf971.../172.17.8.102
Unit apache-discovery@7777.service launched on 151bf971.../172.17.8.102
core@core-01 ~ $
```

Imatge 26: Llençament del servei d'apache

En aquest cas hem demanat a fleet que ens creï una instància de cada servei. Una vegada ha creat les instàncies ens retorna que el servei ha estat llençat i la màquina en la que es troba.

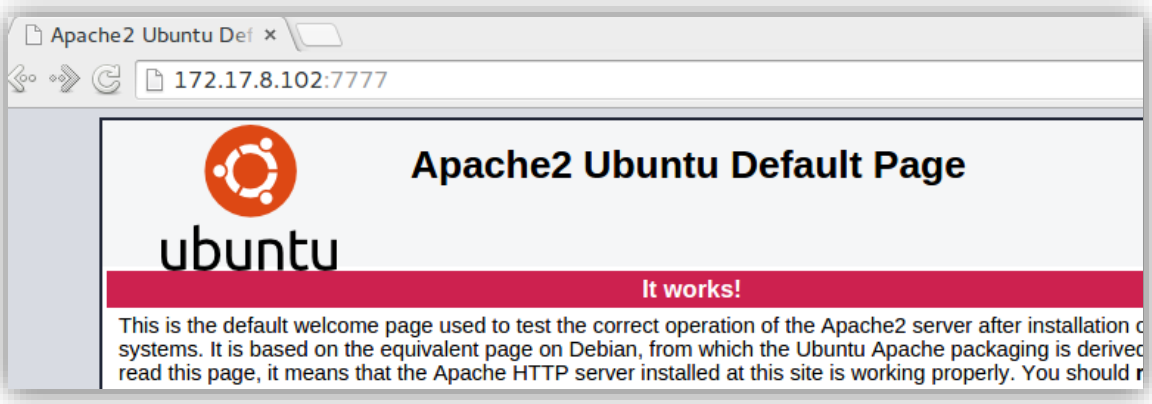
Si utilitzem fleetctl list-units podem veure tots els processos que hem llençat al clúster.

```
core@core-01 ~ $ fleetctl list-units
UNIT                            MACHINE                         ACTIVE  SUB
apache-discovery@7777.service   151bf971.../172.17.8.102        active  running
apache@7777.service             151bf971.../172.17.8.102        active  running
core@core-01 ~ $
```

Imatge 27: Mostra de l'estat del clúster

Efectivament tenim els dos processos que corren al mateix node (mateixa ip).

Podem comprovar també que el servei està actiu posant al navegador el port i la ip corresponent.



Imatge 28: Plana web mostrant el servei

La part interessant bé quan agafem el node en el que s'està executant el servei i l'aturem. Podem fer això utilitzant la comanda de vagrant halt.

```
$>vagrant status
Current machine states:

core-01                   running (virtualbox)
core-02                   running (virtualbox)
core-03                   poweroff (virtualbox)
core-04                   running (virtualbox)
core-05                   running (virtualbox)

This environment represents multiple VMs. The VMs are all listed
above with their current state. For more information about a specific
VM, run `vagrant status NAME`.
$>vagrant halt core-02
==> core-02: Attempting graceful shutdown of VM...
$>
```

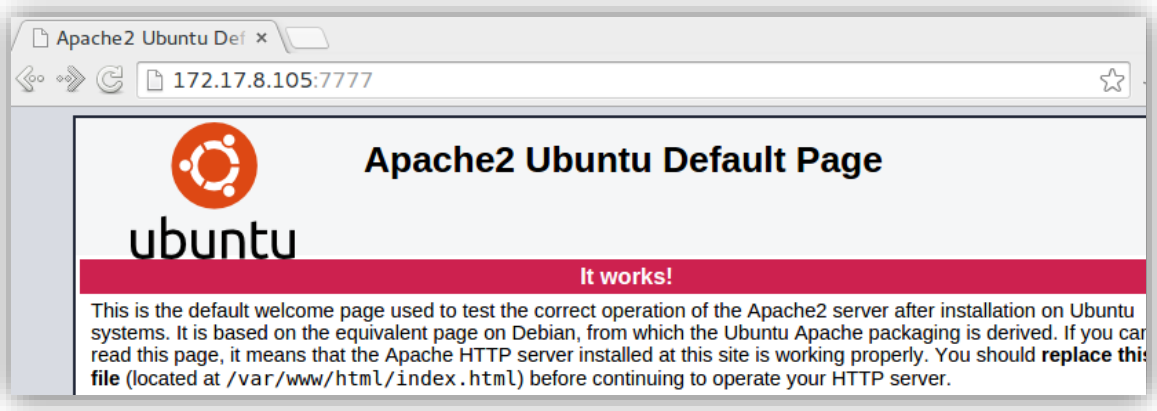
Imatge 29: Eliminació d'un dels nodes

Si llavors tornem a entrar al clúster tornem a mostrar la llista de processos veiem que els serveis tornen a estar actius, en una màquina diferent.

```
core@core-01 ~ $ fleetctl list-units
UNIT                            MACHINE                         ACTIVE  SUB
apache-discovery@7777.service   44aa01cf.../172.17.8.105        active  running
apache@7777.service             44aa01cf.../172.17.8.105        active  running
core@core-01 ~ $
```

Imatge 30: Mostra de l'estat del clúster

Per tant si anem a la nova ip del servei veiem que la pàgina continua activa.



Imatge 31: Mostra del servei funcionant un altre cop

Una vegada hem vist aquest servei passarem a veure'n un d'una topologia i una estructura molt diferent.

6.2 Servei de base de dades

A diferència del servei anterior un servei de base de dades porta una complexitat addicional pel simple fet que ha de mantenir unes dades de manera consistent i ràpida. La primera aproximació que hem implantat és un servei de mysql.

6.2.1 Mysql

6.2.1.1 Anàlisis del servei

Per a implementar aquest servei hem seguit l'estructura anterior utilitzada per al webservice. També disposem de dos fitxers un servei de descobriment i el servei. Degut a que part de la configuració de la base de dades s'ha de realitzar abans de crear la imatge del mateix contenidor mantindrem el mateix port de connexió (3636). Per a no repetir-nos amb els fitxers de configuració al ja haver vist la estructura mostrarem només els canvis significatius, com a l'hora de llançar un altre contenidor. El servei de descobriment funciona com al servei d'apache anterior, tret de la part en la que li hem especificat el port directament.

```
ExecStart=/bin/bash -c '\
  while true; do \
    curl -f ${COREOS_PRIVATE_IPV4}:3306; \
    if [ $? -eq 0 ]; then \
      etcdctl set /services/mysql/${COREOS_PRIVATE_IPV4} \'${COREOS_PRIVATE_IPV4}:3306\' --ttl 30; \
    else \
      etcdctl rm /services/mysql/${COREOS_PRIVATE_IPV4}; \
    fi; \
    sleep 20; \
  done'
```

Imatge 32: Mostra de part del servei de descobriment de mysql

El servei en si també funciona de la mateixa manera que el servei anterior d'apache, tret del contenidor i els paràmetres que utilitzem a l'hora de crear-lo.

```
ExecStart=/usr/bin/docker run --name mysql.%i -v /mysqlApp -e "MYSQL_DATABASE=DataBase" -e
"MYSQL_ROOT_PASSWORD=ThatEnormousPass" -p 3306:3306 ctlc/mysql
```

Imatge 33: Mostra del a part del servei en si

En aquest cas quan aixequem el contenidor amb docker li especifiquem el lloc on ha de guardar les dades del node, també li especifiquem el nom de la base de dades, la contrasenya de l'usuari mestre i el port, respectivament. També li hem d'especificar la imatge; si no realitzem un pull anteriorment (com en l'exemple d'apache) el mateix sistema ja farà un pull si no troba la imatge al repositori local de docker.

6.2.1.2 Testeig

Ara que ja disposem dels dos fitxers provarem de fer un deploy al clúster del nostre servei. Seguirem els mateixos passos i comandes que hem fet servir en el servei d'apache anterior.

```
core@core-01 ~/share/BackUPCopy/templates $ fleetctl start mysql@7777 mysql-discovery@7777
Unit mysql@7777.service launched on 031c6e65.../172.17.8.102
Unit mysql-discovery@7777.service launched on 031c6e65.../172.17.8.102
core@core-01 ~/share/BackUPCopy/templates $ fleetctl list-units
UNIT                            MACHINE                         ACTIVE  SUB
mysql-discovery@7777.service    031c6e65.../172.17.8.102        active  running
mysql@7777.service              031c6e65.../172.17.8.102        active  running
core@core-01 ~/share/BackUPCopy/templates $
```

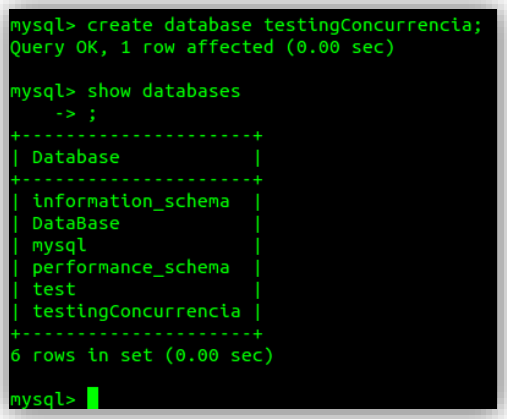
Imatge 34: Mostra de l'estat del clúster

Amb el servei aixecat provarem de connectar-nos a la base de dades d'una terminal fora del clúster.

```
$>mysql -h 172.17.8.102  -u root  -p DataBase
Enter password:
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 2
Server version: 5.5.35-0ubuntu0.12.04.2-log (Ubuntu)

Copyright (c) 2000, 2014, Oracle and/or its affiliates. All rights reserved.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql>
```

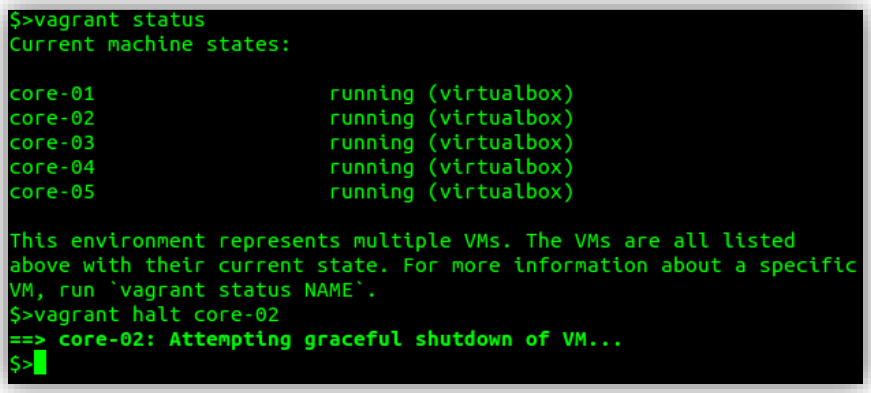
Imatge 35: Proba de connexió a la base de dades

I crearem una base de dades de proves per a veure si el sistema és capaç de mantenir-la.



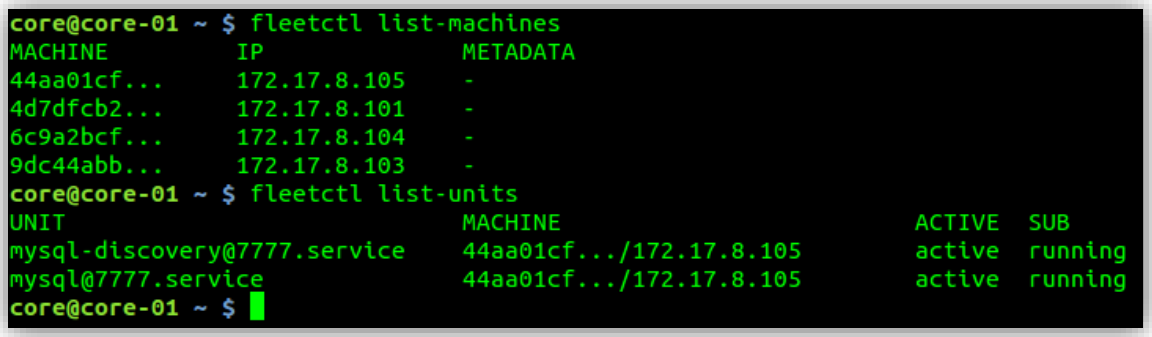
Imatge 36: Creació d'una base de dades

Una vegada ja tenim la taula creada procedirem a parar el node en que es troba el nostre servei.



Imatge 37: Destrucció d'un dels nodes

I comprovem que fleet ens ha recol·locat el servei en un altre node.



Imatge 38: Comprovació de que s'ha recol·locat el servei

El problema però ens el trobem quan ens connectem a la base de dades i mostrem una llista de les taules

```
$>mysql -h 172.17.8.105  -u root  -p DataBase
Enter password:
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 1
Server version: 5.5.35-0ubuntu0.12.04.2-log (Ubuntu)
```

Imatge 40: Connexió amb la base de dades

```
mysql> show databases;
+--------------------+
| Database           |
+--------------------+
| information_schema |
| DataBase           |
| mysql              |
| performance_schema |
| test               |
+--------------------+
5 rows in set (0.00 sec)

mysql>
```

Imatge 39: Llista de taules

Com veiem a la imatge la base de dades que hem creat abans de proves s'ha eliminat. Això és degut a que al guardar la informació al node quan aquest desapareix fleet torna a crear el servei en un altre node però no s'encarrega de mantenir les dades que haguem modificat. En veure el problema hem buscat una altre tipus de base de dades que ens permeti replicar elements per tal d'intentar mantenir la informació.

6.2.2 MongoDB

L'alternativa que sembla tenir més sentit és agafar una base de dades No-Sql, una de les alternatives que ja incorpora replicació és MongoDB.

6.2.2.1 Anàlisis del servei

A diferència de l'exemple anterior aquesta implementació de mongoDB [24] és bastant més complexa i no està separada utilitzant la estructura que hem vist abans.
Degut a l'extensió del fitxer no hem posat imatges i farem referència pel nom, els fitxers que es poden trobar al codi adjunt.

Aquesta solució proposa utilitzar dos fitxers, un per a fer córrer un servei de mongo genèric als nodes que sigui necessari (mongo.service), i un altre per a gestionar la rèplica (mongo-replica-config.service) quan s'afegeixi o es tregui un node. Els passos que segueixen els posarem a continuació:

- Primer aixequem el primer node i iniciem també el servei de rèplica que s'encarrega d'establir un seguit de claus i valors per a que els altres nodes es connectin.
- Una vegada s'han generat els valors els altres nodes es connecten i funcionen com a rèplica.
- Finalment mantenim un control de etcd per si apareix un nou node poder-lo connectar al node principal per a que pugi replicar-se

Observant els fitxers i l'estructura de la solució podem veure que no és la més adient i que una forma més correcta seria creant un programa que gestionés la replica de fitxers i fer-lo funcionar dins un contenidor.

6.2.2.2 Testeig

Després de fer un testeig del sistema ens hem adonat que no funciona correctament i que quan un node cau i el sistema de fleet el reubica el servei de configuració no ho detecta i per tant no realitza cap canvi. També hem comprovat que si el node principal és el que cau el sistema cau sencer.

7.Conclusions i treball futur

Al principi hem parlat del problema de la quantitat de recursos que eren necessaris per a mantenir un servei distribuït i robust. Hem posat l'exemple d'una empresa proveïdora de serveis de compra online que volia mantenir la seva estructura personalitzada i sense utilitzar grans plataformes com AWS o GoogleCloud. Per això volíem veure diverses infraestructures per tal d'escollir la més adient a l'hora de implantar un servei cloud. Per a trobar una solució al problema hem explorat les diverses infraestructures i hem escollit una d'elles, CoreOs. Malgrat ser una infraestructura molt nova encara en desenvolupament hem cregut que era la més adequada per a desplegar la nostra aplicació.

Amb el primer objectiu que ens vàrem proposar volíem veure què necessitàvem i què havíem de tenir en compte a l'hora de desenvolupar la nostra aplicació. Amb l'anàlisi de 12 factor app hem vist quins principis són necessaris i com hauria de ser l'estructura. Una vegada vist això ens hem centrat més en la infraestructura que hauria de sostenir una aplicació d'aquest tipus, com definíem en el segon objectiu. Hem realitzat una investigació de les diverses plataformes i hem vist que per les nostres necessitats , el cas de l'empresa , el més adient era muntar un sistema de CoreOS. El fet de que sigui un sistema totalment portable fa que no depenguem d'un proveïdor en concret i ens dona la llibertat de muntar-lo sobre el mateix hardware si és necessari, això per una empresa que no té molt clar si necessitarà expandir-se o no dona una llibertat absoluta i totalment necessària.
En el següent objectiu plasmàvem la necessitat d'explorar aquest nou paradigma (OSaaS) i veure fins a quin punt és aconsellable i les seves limitacions. Hem investigat sobretot en el sistema referencia en aquest paradigma, CoreOs i hem descobert que per aplicacions que no requereixin persistència de dades és un sistema molt robust i consistent. Malgrat això el fet de que encara estigui en desenvolupament ens obre la possibilitat que de cara un futur millori en solucionar aquest problema.

Els altres dos objectius que teníem ens portaven a recomanar una estructura i provar de desplegar un seguit de serveis sobre aquesta. Tot hi que hem proposat una estructura molt petita per a entorns de desenvolupament i prova és la més adequada. Tampoc limita la seva utilització ja que modificant poques coses es pot dur el mateix sistema en un clúster de producció.
També hem vist com havíem de desplegar un seguit de serveis , hem vist com desplegar un servei sense persistència de dades és molt senzill i ens permet obtenir una tolerància molt alta a fallades i una resposta gairebé immediata del sistema. Per altra banda també hem vist el problema que suposa la persistència de dades i com no es pot arribar a una solució de manera senzilla, el nostre consell en aquest punt és el de seguir el 4t principi de 12 factor app i buscar-nos un servei extern (Amazon RDS, Redis, ...) a l'hora de guardar aquestes dades.

Creiem doncs que amb aquest projecte hem assolit tots els objectius proposats i hem obtingut els coneixements necessaris com per a implementar un sistema d'aquestes característiques i aconsellar correctament a la empresa del principi que utilitzi un altre tipus de sistema més desenvolupat o que esperi un temps a

que millori aquest tipus d'arquitectura.

7.1 Treball futur

Com ja hem comentat al llarg de tota la memòria la tecnologia amb la que estem treballant evoluciona constantment, això ha fet que en el moment en el que ens trobem hi hagin més eines que degut a la seva novetat no hem pogut analitzar i incloure en aquest projecte. Per això deixem mencionat aquí cap on s'ha de treballar i quines d'aquestes eines és interessant investigar.

- **Investigació sobre fannel**

Una eina afegida al sistema de manera estable des d'aquest gener, que ens permet realitzar una comunicació entre contenidors via UDP.

- **Investigar sobre Kubernetes**

Una eina creada per Google per a organitzar un clúster amb contenidors docker, que hauria de funcionar sobre CoreOs.

- **Investigar més sobre rocket**

El desembre passat CoreOs va anunciar que trauria el seu propi servei de contenidors, el seu propi Docker, al·legant que el sistema de Docker s'havia anat desviant massa del seu objectiu. Per tant des de llavors tenim un nou sistema que és diu Rocket el qual des del desembre passat es pot descarregar de github[25] però que encara està en fase de prototip.

- **Provar d'utilitzar una base de dades com Cassandra en comptes de mongoDB**

En preguntar a un membre de l'staff de CoreOs [26] sobre el funcionament de CoreOs amb les bases de dades aquest recomanava utilitzar Cassandra com a base de dades. És una opció a valorar ja que Apache Cassandra és utilitzat per a varies companyies amb clústers molt grans de nodes i amb grans volums de dades. En aquest projecte primer es va intentar realitzar un aproximament simple al problema intentant utilitzar un servei de postgresql, el qual va fallar dramàticament. I posteriorment es va optar per a mongoDB el qual no va acabar de funcionar.

- **Buscar algun sistema que permeti una visualització més interactiva de l'estat del clúster, Consul o provar CoreOS Managed Linux**

Un dels problemes de CoreOs és que s'ha de realitzar tot el control del clúster a través d'una consola amb comandes. Per evitar això estaria bé provar una gestió més interactiva del clúster per exemple utilitzant un servei com Consul o contractant el servei que el mateix CoreOs ofereix.

- **Investigar més sobre Vulcand**

Després d'experimentar amb diversos serveis balancejadors de càrrega i "servidors" de tràfic, al no obtenir un resultat clar he decidit no incloure'ls en el projecte, però són un element a expandir clarament.

- **Experimentació mes profunda en altres sistemes**

Alguns sistemes nous com Flyn o Tsuru prometen molt

8.Referències bibliogràfiques

[1] **Cloud computing** , 10/09/2014, http://en.wikipedia.org/wiki/Cloud_computing
[2] **OSaaS**, 15/10/2014 ,http://www.cio-today.com/article/index.php?story_id=13200G72P1BC
[3] **Benjamin hindman sobre estructures de clúster** , 11/09/2014, https://www.youtube.com/watch?v=F1-UEIG7u5g#t=733
[4] **Docker**, 29/08/2014, https://docs.docker.com/
[5] **Apache mesos**, 4/09/2014, http://mesos.apache.org/documentation/latest/
[6] **CoreOs** ,13/10/2014 , http://www.sebastien-han.fr/blog/2013/09/03/first-glimpse-at-coreos/
[7] **Carl Sverre sobre CoreOs**, 11/10/2014, https://www.youtube.com/watch?v=uJirOCUg67o#t=1493
[8] **Flynn,** 15/10/2014, https://flynn.io/docs
[9] **Tsuru,** 15/10/2014, http://docs.tsuru.io/en/latest/
[10] **Deis**, 15/10/2014, http://docs.deis.io/en/latest/toctree/#toctree
[11] **Chris Aniszczyk sobre Twitter i MesOs**, 15/10/2014, http://opensource.com/business/14/8/interview-chris-aniszczyk-twitter-apache-mesos
[12] **Benjamin Hindman sobre Twitter i MesOs**, 15/10/2014, http://static.usenix.org/event/nsdi11/tech/full_papers/Hindman_new.pdf
[13] **CoreOs i Rocket** , 10/12/2014, https://gigaom.com/2014/12/02/why-coreos-just-fired-a-rocket-at-docker/
[14] **Paxos,** 15/12/2014, http://en.wikipedia.org/wiki/Paxos_%28computer_science%29
[15] **Estructura de Raft**, 19/12/2014, https://speakerdeck.com/benbjohnson/raft-the-understandable-distributed-consensus-protocol.
[16] **Init** , 10/01/2015, http://en.wikipedia.org/wiki/Init
[17] **DBus** , 10/01/2015, http://en.wikipedia.org/wiki/D-Bus
[18] **Cow**, 10/01/2015, http://en.wikipedia.org/wiki/Copy-on-write
[19] **Arxius Cow,** 13/10/2014, http://arstechnica.com/information-technology/2014/01/bitrot-and-atomic-cows-inside-next-gen-filesystems/1/
[20] **Twelve factor app,** 11/01/2015, http://12factor.net/
[21] **Agile manifesto**, 11/01/2015, http://www.agilemanifesto.org/principles.html
[22] **YAML,** 10/01/2015, https://en.wikipedia.org/wiki/YAML
[23] **Cloud-init,** 12/01/2015, http://cloudinit.readthedocs.org/en/latest/
[24] **Implementació de mongoDB per a CoreOs,** 15/11/2014, https://github.com/auth0/coreos-mongodb
[25] **Rocket**, 14/01/2015, https://github.com/coreos/rocket
[26] **CoreOs i les bases de dades,** 14/01/2015, https://www.youtube.com/watch?v=VF5ecG6vdN8#t=1425
[27] **Coreos**, 11/10/2014, https://coreos.com/docs/
[28] **Heroku,** 11/10/2014, https://www.heroku.com/

9.Annexe: Setup del clúster

Annex a la memòria del tfg trobem un zip comprimit que conté una imatge de vagrant i el codi de configuració dels serveis (els unit files). Per a poder muntar el nostre clúster de CoreOs necessitem un seguit de coses:

- Connexió a internet.
- Virtual box.
- Vagrant.
- Arxiu comprimit adjunt amb la memòria.

Necessitarem la connexió a internet ja que docker ha de baixar un seguit d'imatges, vagrant a l'inici també haurà de descarregar la imatge de CoreOs. És important que configurem vagrant de manera que és descarregui una versió anterior a la 557.0.0, sinó l'exemple de MongoDB no ens funcionarà.

Una vegada ens haguem assegurat que disposem de tots els requisits només ens faltarà descomprimir la memòria i situar-nos dins de la carpeta coreos-vagrant.

El primer que farem serà un vagrant up , ens demanarà que ens autentifiquem com a usuari root per a muntar la carpeta actual com a carpeta compartida dins de cada node. Ni no volem això podem comentar la línia 95 del fitxer vagrant que hi ha dins de la carpeta.

```
$>ls
coreos-vagrant
$>cd coreos-vagrant/
$>vagrant up
Bringing machine 'core-01' up with 'virtualbox' provider...
Bringing machine 'core-02' up with 'virtualbox' provider...
Bringing machine 'core-03' up with 'virtualbox' provider...
Bringing machine 'core-04' up with 'virtualbox' provider...
Bringing machine 'core-05' up with 'virtualbox' provider...
==> core-01: Checking if box 'coreos-alpha' is up to date...
==> core-01: Clearing any previously set forwarded ports...
==> core-01: Clearing any previously set network interfaces...
```

Una vegada hagi acabat afegirem la clau ssh de vagrant per a poder-nos conectar, abans d'entrar a un node mirarem que els tinguem creats i funcionant.

```
$>ssh-add ~/.vagrant.d/insecure_private_key
Identity added: /home/hiro/.vagrant.d/insecure_private_key (/home/hiro/.vagrant
d/insecure_private_key)
$>vagrant status
Current machine states:

core-01                   running (virtualbox)
core-02                   running (virtualbox)
core-03                   running (virtualbox)
core-04                   running (virtualbox)
core-05                   running (virtualbox)

This environment represents multiple VMs. The VMs are all listed
above with their current state. For more information about a specific
VM, run `vagrant status NAME`.
$>
```

Després de comprovar que funcionen passarem a entrar a un node del clúster, normalment sempre solem entrar al core-01 però perfectament podríem realitzar tot aquest procés des de qualsevol altre node. Per a connectar-nos utilitzarem la comanda de vagrant ssh i a continuació el nom de la màquina a la que ens volem connectar , també li afegirem el flag “-- -A” per a temes d’autentificació.

```
$>vagrant ssh core-01 -- -A
Last login: Fri Jan 23 01:24:49 2015 from 10.0.2.2
CoreOS alpha (561.0.0)
core@core-01 ~ $
```

Una vegada dins del node podem veure l’estat del clúster mitjançant fleet (fleetctl). Podem veure tan l’estat de les màquines com l’estat dels processos.

```
core@core-01 ~/share/serveis/apache $ fleetctl list-machines
MACHINE         IP              METADATA
1a332719...     172.17.8.105    -
546343f3...     172.17.8.103    -
8841ba4c...     172.17.8.102    -
911d35d5...     172.17.8.104    -
b5c0f7c1...     172.17.8.101    -
core@core-01 ~/share/serveis/apache $ fleetctl list-units
UNIT                            MACHINE                         ACTIVE          SUB
apache-discovery@7777.service   1a332719.../172.17.8.105        inactive        dead
apache@7777.service             1a332719.../172.17.8.105        activating      start-pre
core@core-01 ~/share/serveis/apache $
```

Si volem iniciar un servei tenim diverses maneres de fer-ho la manera més ràpida és fer un fleet start amb el nom del fitxer del servei. Si ens situem sobre la carpeta de share hi trobarem les carpetes i fitxers que posem dins de la carpeta coreos-vagrant.

```
core@core-01 ~/share/serveis/apache $ fleetctl start apache@7777.service
apache-discovery@7777.service
Unit apache@7777.service launched on 1a332719.../172.17.8.105
Unit apache-discovery@7777.service launched on 1a332719.../172.17.8.105
```

Si volem comprovar l'estat del servei podem utilitzar unes altres dues comandes que ens permeten veure un log de registres i el seu estat actual.

```
core@core-01 ~/share/serveis/apache $ fleetctl journal apache@7777
-- Logs begin at Fri 2015-01-23 01:23:27 UTC, end at Fri 2015-01-23 05:02:11 UTC. --
Jan 23 04:58:29 core-05 docker[923]: 5506de2b643b: Pulling metadata
Jan 23 04:58:30 core-05 docker[923]: 5506de2b643b: Pulling fs layer
Jan 23 04:58:32 core-05 docker[923]: 5506de2b643b: Download complete
Jan 23 04:58:32 core-05 docker[923]: cbbfc8dd8563: Pulling metadata
Jan 23 04:58:33 core-05 docker[923]: cbbfc8dd8563: Pulling fs layer
Jan 23 04:59:03 core-05 docker[923]: cbbfc8dd8563: Download complete
Jan 23 04:59:03 core-05 docker[923]: cbbfc8dd8563: Download complete
Jan 23 04:59:03 core-05 docker[923]: Status: Downloaded newer image for hirocat/apache:latest
Jan 23 04:59:03 core-05 systemd[1]: Started Apache web server service on port 7777.
```

```
core@core-01 ~/share/serveis/apache $ fleetctl list-units
UNIT                            MACHINE                         ACTIVE  SUB
apache-discovery@7777.service   1a332719.../172.17.8.105        active  running
apache@7777.service             1a332719.../172.17.8.105        active  running
core@core-01 ~/share/serveis/apache $ fleetctl status apache@7777
● apache@7777.service - Apache web server service on port 7777
   Loaded: loaded (/run/fleet/units/apache@7777.service; linked-runtime; vendor preset: disabled)
   Active: active (running) since Fri 2015-01-23 04:59:03 UTC; 2min 57s ago
  Process: 923 ExecStartPre=/usr/bin/docker pull hirocat/apache (code=exited, status=0/SUCCESS)
  Process: 913 ExecStartPre=/usr/bin/docker rm apache.%i (code=exited, status=1/FAILURE)
  Process: 852 ExecStartPre=/usr/bin/docker kill apache.%i (code=exited, status=1/FAILURE)
 Main PID: 1018 (docker)
   CGroup: /system.slice/system-apache.slice/apache@7777.service
           └─1018 /usr/bin/docker run --name apache.7777 -p 172.17.8.105:7777:80 hirocat/apache /usr/sbin/apa
che2ctl -D FOREGROUND
```

En el cas de que ens volguéssim connectar a una node per configurar alguna cosa també podem fer a través de l'id de la màquina o de les unitats que té executant.

```
core@core-01 ~/share/serveis/apache $ fleetctl ssh apache@7777
Last login: Fri Jan 23 05:02:11 2015 from 172.17.8.101
CoreOS alpha (561.0.0)
core@core-05 ~ $
```

Si volem finalitzar un servei només hem de fer un destroy a través de fleet i aquest s'encarrega d'eliminar-lo.

```
core@core-01 ~/share/serveis/apache $ fleetctl destroy apache@7777 apache-discovery@7777
Destroyed apache@7777.service
Destroyed apache-discovery@7777.service
```

A part de fleet també podem utilitzar etcd (etcdctl) per a establir parelles de claus i valors.

```
core@core-01 ~/share/serveis/apache $ etcdctl set /prova/testing provaDeTFG
provaDeTFG
core@core-01 ~/share/serveis/apache $ etcdctl ls --recursive
/coreos.com
/coreos.com/updateengine
/coreos.com/updateengine/rebootlock
/coreos.com/updateengine/rebootlock/semaphore
/mongo
/mongo/replica
/mongo/replica/name
/services
/services/apache
/prova
/prova/testing
```