



UNIVERSIDAD DE LAS PALMAS DE GRAN CANARIA
Escuela de Ingeniería Informática



TRABAJO DE FIN DE MÁSTER EN INGENIERÍA INFORMÁTICA

Despliegue de una aplicación Ruby on Rails utilizando las tecnologías de virtualización Docker y CoreOS en la nube pública de Amazon Web Services

Carolina Santana Martel

Tutor: Dr. Carmelo Cuenca Hernández

Co-Tutora: Dra. Francisca Quintana Domínguez

Julio de 2017

Agradecimientos

Quisiera agradecer, en primer lugar, a mis tutores, Carmelo Cuenca Hernández y Francisca Quintana Domínguez, por compartir conmigo su idea, por su disponibilidad constante para guiarme, orientarme y apoyarme durante todo el proceso, su confianza y el ser tan alentadores. Ha sido un placer haber podido trabajar con dos grandes profesionales, siempre investigando y transmitiendo nuevas tecnologías y prácticas.

Además, dar las gracias a todos aquellos docentes que me han formado durante el Máster, a los que, junto a los del Grado, siempre deberé parte de lo que hoy en día sé.

También acordarme de mi familia. A mis padres María Dolores Martel y Juan Manuel Santana, a mi hermana Yasmina Santana y a mi pareja Daniel Tejera. Gracias por el apoyo incondicional, la fe depositada en mí, las infinitas conversaciones y los ánimos para que nunca pierda las ganas de seguir avanzando con seguridad en las fases venideras.

A esta etapa por permitirme descubrir a lo que me gustaría dedicarme, esperando no dejar nunca de aprender los conocimientos que a la larga se deriven del trabajo de todos los que ya formamos y formarán, en un futuro, parte de esta profesión.

Carolina Santana Martel

Índice

Prefacio	vii
1 Acerca de este Trabajo de Fin de Máster	1
1.1 Motivación	1
1.2 Objetivos	1
1.2.1 Objetivos iniciales	1
1.2.2 Objetivos generales	1
1.3 Metodología	2
1.4 Planificación inicial y seguimiento	2
1.5 Aportaciones	3
1.6 Justificación de la competencia específica cubierta	3
2 Estado del arte	5
2.1 Tecnologías de contenedores	5
2.1.1 Docker	6
2.1.2 Contenedores y Máquinas Virtuales	6
2.2 Sistemas operativos orientados a contenedores	7
2.2.1 CoreOS	8
2.3 Otros	10
2.4 Herramientas para la orquestación de contenedores	11
2.4.1 Fleet	11
2.4.2 Otras	12
2.5 Proveedores de Infraestructura como Servicio	14
2.5.1 Amazon Web Services	15
2.6 Otras herramientas tecnológicas	16
2.6.1 Vagrant	16
2.6.2 VirtualBox	17
2.6.3 GitHub	17
2.6.4 Travis CI	18
2.6.5 Softcover	18
3 Desarrollo	19
3.1 Análisis de la aplicación Ruby on Rails	19
3.2 Iteración 1: Arquitectura de microservicios	21
3.2.1 Preparación del repositorio local y remoto	22
3.2.2 Cambio y configuración de la base de datos	22
3.2.3 Cambio del alimentador idempotente de base de datos	23
3.2.4 Cambio y configuración del servidor web	24
3.2.5 Configuración para la creación de los contenedores	24
3.2.6 Resultado	27
3.3 Iteración 2: Subida de la imagen a Docker Hub	28
3.4 Iteración 3: Integración y Despliegue continuos	29

3.4.1	Vinculación con el repositorio remoto	30
3.4.2	Configuración de condiciones	30
3.4.3	Resultado	31
3.5	Iteración 4: Despliegue en VirtualBox	33
3.5.1	Preparación del repositorio local y remoto	34
3.5.2	Preparación de la cabecera del fichero user-data	35
3.5.3	Interpretación del fichero config.rb	36
3.5.4	Preparación del fichero Vagrantfile	37
3.5.5	Despliegue manual de las unidades systemd	38
3.5.6	Despliegue automático de las unidades systemd	44
3.5.7	Resultado	46
3.6	Iteración 5: Despliegue en Amazon Web Services	47
3.6.1	Obtención de la cuenta AWS	49
3.6.2	Creación de la red y subred virtual	49
3.6.3	Creación del par de claves y del grupo de seguridad	50
3.6.4	Configuración del fichero user-data	51
3.6.5	Configuración del fichero Vagrantfile	51
3.6.6	Despliegue de una instancia	53
3.6.7	Despliegue de tres instancias	55
3.6.8	Cambio de las unidades systemd a fleet	59
3.6.9	Configuración DNS usando SkyDNS	63
3.6.10	Balanceo de carga con Nginx usando Confd	69
3.6.11	Resultado	80
4	Aspectos Económicos	83
5	Resultados y Conclusiones	87
5.1	Resultados	87
5.2	Conclusiones	88
5.2.1	Consecución de objetivos	88
5.2.2	Trabajos futuros	89
5.2.3	Valoración personal	89
Fuentes de Información		91

Prefacio

Resumen

Este Trabajo de Fin de Máster trata la utilización de distintas tecnologías de virtualización, contenedores y orquestación sobre una aplicación web para generar una infraestructura como código reproducible, tanto en local como en la nube. Así, se toma una aplicación Ruby on Rails, se le realizan una serie de cambios y se redefine a una arquitectura de microservicios encapsulados en contenedores Docker. Además, se prepara para la integración y despliegue continuos con el uso de Travis CI, Docker Hub y GitHub. Ésta será ejecutada sobre el sistema operativo CoreOS, que contiene las funcionalidades mínimas necesarias para la implementación de aplicaciones basadas en contenedores, y distribuida en un clúster. La infraestructura resultante se despliega haciendo uso de Vagrant en sistemas virtuales y en una nube pública, en los proveedores VirtualBox y Amazon Web Services, respectivamente.

Palabras clave: Amazon Web Services, aplicación, clúster, contenedor, CoreOS, despliegue, Docker, Docker Hub, GitHub, infraestructura, integración, microservicio, nube, orquestación, Travis CI, Vagrant, VirtualBox, virtualización.

Abstract

This Master's degree work addresses the use of different virtualization, containers and orchestration technologies on a web application to generate an infrastructure as a reproducible code, both locally and in the cloud. Thus, a Ruby on Rails application is taken, a series of changes are made to it and it is redefined to an architecture of micro-services encapsulated in Docker containers. In addition, it is prepared for continuous integration and deployment with the use of Travis CI, Docker Hub and GitHub. It will be executed on the CoreOS operating system, which contains the functionality needed to implement container-based applications, and it will be distributed in a cluster. The resulting infrastructure is deployed using Vagrant in virtual systems and in a public cloud, in VirtualBox and Amazon Web Services providers, respectively.

Keywords: Amazon Web Services, application, cloud, cluster, container, CoreOS, deployment, Docker, Docker Hub, GitHub, infrastructure, integration, microservice, orchestration, Travis CI, Vagrant, VirtualBox, virtualization.

Capítulo 1

Acerca de este Trabajo de Fin de Máster

1.1 Motivación

La motivación para la realización del presente trabajo reside en el desarrollo e implementación de nuevas infraestructuras orientadas a la realización de despliegues automatizados. Para ello se quiere hacer uso de tecnologías emergentes que permiten crear medios más robustos, flexibles y escalables en la tendencia de la computación en la nube, la computación de altas prestaciones y la virtualización.

1.2 Objetivos

En esta sección se exponen los objetivos iniciales y generales a alcanzar.

1.2.1 Objetivos iniciales

El objetivo principal es la aplicación de distintas tecnologías de virtualización, contenedores y orquestación a una aplicación web para generar un entorno reproducible de desarrollo tanto en local como en la nube.

Además otro objetivo del trabajo es que la estudiante se forme en tecnologías actuales y emergentes de virtualización y computación en la nube.

1.2.2 Objetivos generales

Se trabajarán distintas tecnologías destacando las cualidades y beneficios del uso de tecnologías de virtualización y computación en la nube como herramientas para desarrollar infraestructuras robustas, flexibles y escalables.

La arquitectura basada en contenedores permitirá que si uno de los servicios de la aplicación falla no repercute en que los otros servicios puedan funcionar por separado. Esto permite detectar, aislar y corregir fallos de una manera directa y eficiente, donde cada contenedor dispone, únicamente, de los recursos que necesita para llevar a cabo sus funciones.

Otro de los objetivos es la introducción y configuración de nuevas unidades de servicio que permitan gestionar y controlar el correcto funcionamiento de cada componente de la infraestructura de manera efectiva.

Los servidores web de la aplicación serán ubicados en todos los miembros del clúster, apreciando la característica de redundancia. Un servidor proxy inverso será encargo de gestionar las peticiones entre los anteriores, utilizando el balanceo de carga. En el caso de despliegue en la nube se dispondrá, además, de servicios en alta disponibilidad.

1.3 Metodología

La metodología a llevar a cabo implica el desarrollo iterativo e incremental, donde se planifican 5 bloques o iteraciones. De esta forma los resultados de una iteración se utilizarán como punto de partida para el desarrollo y ampliación en la siguiente.

Las iteraciones a realizar, con las herramientas o tecnologías a usar en cada una de ellas, son:

1. Conversión de la aplicación a una arquitectura de microservicios con el uso de Docker (automatiza el despliegue de aplicaciones dentro de contenedores de software).
2. Subida de la imagen resultante de la aplicación al repositorio de imágenes Docker Hub (repositorio de imágenes Docker basado en la nube).
3. Integración y Despliegue continuos de la aplicación con Travis CI (servicio distribuido de integración y despliegue continuos para construir y probar proyectos de software alojados en GitHub) y GitHub (plataforma colaborativa para alojar proyectos utilizando el sistema de control de versiones Git).
4. Despliegue monomáquina con VirtualBox (herramienta de virtualización de código abierto multiplataforma), Vagrant (herramienta para la creación y configuración de entornos de desarrollo virtualizados), y CoreOS (sistema operativo basado en el kernel de Linux con las funcionalidades mínimas necesarias para la implementación de aplicaciones dentro de contenedores de software).
5. Despliegue en la nube pública de Amazon Web Services (colección de servicios de computación en la nube que forman una infraestructura como servicio y una plataforma como servicio).

1.4 Planificación inicial y seguimiento

La planificación inicial contemplaba las siguientes 4 fases, con una duración total estimada de 300 horas:

1. Estudio previo del estado del arte que contempla las tecnologías, sistemas operativos y herramientas para la orquestación orientadas a contenedores, así como proveedores de infraestructura como servicio y otras herramientas tecnológicas necesarias para la implementación. También se realiza el análisis de la aplicación Ruby on Rails. Duración estimada de 80 horas.
2. Diseño, desarrollo e implementación de las 5 iteraciones propuestas. Duración estimada 145 horas.
3. Evaluación, validación y prueba de las funcionalidades añadidas en cada una de las iteraciones. Duración estimada de 25 horas.
4. Preparación de la documentación y memoria del presente trabajo. Duración estimada de 50 horas.

Las 4 fases previstas se han llevado a cabo como han sido propuestas, organizando dicho trabajo para poder realizar la fase 3 en la finalización de cada una de las 5 iteraciones. Además, los objetivos iniciales se han mantenido como eje central del trabajo que se quería producir.

1.5 Aportaciones

Este trabajo se centra en presentar un nuevo arquetipo de infraestructura de aplicación, que enlaza el uso de varias tecnologías emergentes, para generar un entorno reproducible de desarrollo, tanto en local como en la nube.

Este tipo de práctica no se lleva a cabo durante los estudios de Máster ni de Grado y representa el estudio de nuevas nociones presentes en el mercado.

Estas actividades son llevadas a cabo por DevOps, acrónimo inglés de *development* y *operations*, que se refiere a un movimiento que se centra en la comunicación, colaboración e integración entre desarrolladores de software y profesionales en las tecnologías de la información. Así, con estos procedimientos se automatiza el proceso de entrega del software y los cambios en la infraestructura para crear entornos donde la construcción, prueba y lanzamiento de un software pueda ser más rápido y fiable.

A su vez, también son llevadas a cabo por otra nueva disciplina como es SRE, proveniente de *Site Reliability Engineering*. Esta disciplina incorpora y aplica aspectos de la ingeniería de software a operaciones destinadas a crear sistemas de software ultra escalables y altamente confiables.

Estos perfiles no están tan presentes en Canarias como internacionalmente, pero representan un fuerte nicho de mercado que comienza a potenciarse y a ser muy demandados para la aplicación de soluciones como el propuesto despliegue automatizado de la infraestructura implementada.

1.6 Justificación de la competencia específica cubierta

Este Trabajo de Fin de Máster, además de incluir las competencias generales recogidas en la Guía Docente de la asignatura 50915 - Trabajo Fin de Máster, ha cubierto la competencia específica del Máster en Ingeniería Informática TI01 - "Capacidad para modelar, diseñar, definir la arquitectura, implantar, gestionar, operar, administrar y mantener aplicaciones, redes, sistemas, servicios y contenidos informáticos". El presente trabajo modela, diseña y define una nueva arquitectura, basada en contenedores, de una aplicación. A partir de ella se implementa una infraestructura distribuida que gestiona, opera, administra y mantiene la aplicación, la información que manipula, los servicios por los que está compuesta y permiten su funcionamiento, y las redes en las que opera en los sistemas local y remoto en los que se ha realizado su despliegue.

Capítulo 2

Estado del arte

En este capítulo se realiza el estudio del estado del arte del presente trabajo.

2.1 Tecnologías de contenedores

Las tecnologías de contenedores se inspiran en el diseño orientado a servicios de aplicaciones. Éstas se dividen en componentes funcionales o microservicios empaquetados individualmente, junto con todas sus dependencias.

Las aplicaciones orientadas a servicios dividen la funcionalidad del sistema en contenedores que se comunican unos con otros a través de interfaces bien definidas. No tienen que preocuparse de las especificaciones del sistema anfitrión. En su lugar, cada contenedor proporciona interfaces de programación de aplicaciones (*APIs*) consistentes que los clientes puedan usar para acceder al servicio. Estas estrategias permiten intercambiar o actualizar cada componente mientras la *API* lo mantenga. Además, cada contenedor puede escalar o crecer de forma independiente.

Algunos de los beneficios del uso de estas tecnologías son:

- Abstracción del sistema anfitrión donde se ejecutará la aplicación basada en contenedores mediante interfaces definidas, con independencia de los recursos o arquitecturas del sistema anfitrión.
- Escalabilidad en pruebas y puesta en producción.
- Administración simplificada de dependencias y versiones de aplicaciones.
- Ambientes de ejecución aislados a nivel de proceso.
- Compartir contenedores evitando la duplicación y ocupando menor espacio en disco.

Dos de las destacadas tecnologías de contenedores son Docker[4], que será la escogida y detallada en el punto 2.1.1, y rkt[6]. Este último es un software de código abierto del equipo que desarrolla CoreOS[1]. Se trata de un gestor de contenedores de última generación para clústeres de Linux que descubre, verifica, extrae y ejecuta contenedores aislados de aplicaciones que pueden ser conectables. La interfaz principal de rkt comprende un solo ejecutable, en lugar de un demonio. Este diseño es aprovechado para integrarse con los sistemas de inicio existentes y con entornos de orquestación de clúster avanzados.



Figura 2.1: Tecnología de contenedores rkt.¹

¹GitHub rkt. (2017). RKT logo vector [Figura]. Recuperado de <https://github.com/rkt/rkt/blob/master/logos/rkt-horizontal-color.png>

2.1.1 Docker

En este trabajo aplicará Docker como tecnología de contenedores. Docker es un software de código abierto que proporciona las herramientas necesarias para crear y administrar contenedores ligeros y portables, automatizando el despliegue de aplicaciones. Surgió como un proyecto interno de la empresa dotCloud, por Salomón Hykes, y pasó a código abierto en marzo de 2013.



Figura 2.2: Docker.²

La arquitectura de Docker se corresponde con la Figura 2.3.

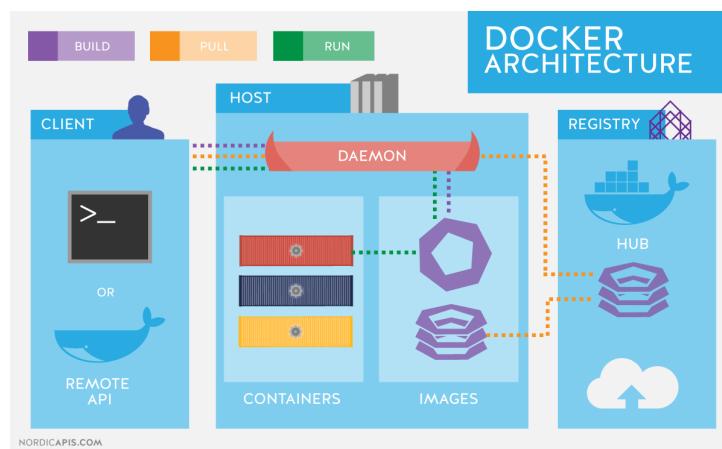


Figura 2.3: Arquitectura Docker.³

Docker Hub^[5] es un servicio de registro basado en la nube que contiene repositorios con imágenes Docker, que pueden ser públicas o privadas. Éstas son plantillas de solo lectura que consisten en una instantánea de una distribución Linux y un conjunto de aplicaciones.

Los pasos de configuración se especifican en un fichero de compilación, llamado **Dockerfile**, que describe cómo crear la imagen del contenedor.

Cada instalación de Docker incluye un cliente, una *API* remota y un demonio Docker. Tanto el cliente como el demonio pueden compartir un único sistema anfitrión o el demonio puede ejecutarse en un sistema anfitrión remoto. Así, los contenedores Docker contienen todo lo necesario para que la aplicación se ejecute de forma aislada, incluyendo el sistema operativo y un sistema de ficheros. Esto permite que los contenedores puedan moverse de sistema anfitrión sin riesgo de errores de configuración.

2.1.2 Contenedores y Máquinas Virtuales

Los contenedores se diferencian de las máquinas virtuales en la ubicación de la capa de virtualización y en la forma en que utilizan los recursos del sistema operativo. Esto se puede observar en la Figura 2.4.

²Docker. (2017). Brand Guidelines [Figura]. Recuperado de <https://www.docker.com/brand-guidelines>

³Mersch. (2017). API-Driven DevOps: Spotlight on Docker [Figura]. Recuperado de <http://nordicapis.com/api-driven-devops-spotlight-on-docker>

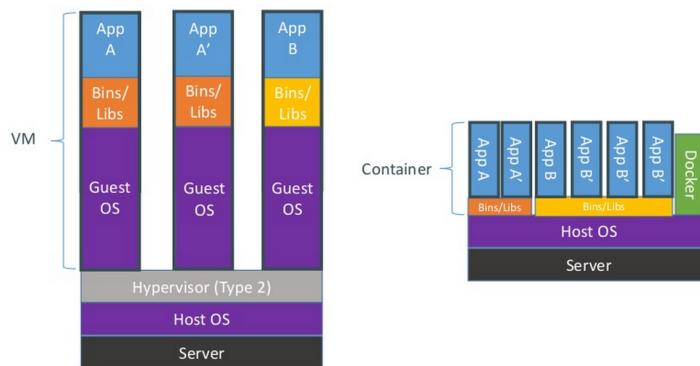


Figura 2.4: Diferencia entre máquina virtual y contenedor.⁴

Las máquinas virtuales se basan en un hipervisor que se instala sobre el hardware del sistema o del sistema operativo. Luego, las instancias de las máquinas virtuales se pueden aprovisionar a partir de los recursos disponibles en el sistema. Las máquinas virtuales están completamente aisladas unas de otras y se pueden migrar de un sistema virtualizado a otro sin tener en cuenta el hardware del sistema o los sistemas operativos.

El entorno de los contenedores está dispuesto de manera diferente. Los contenedores se instalan encima de un sistema operativo anfitrión. Éstos se pueden aprovisionar a partir de los recursos disponibles del sistema y se pueden implementar las aplicaciones necesarias. De esta manera cada aplicación contenedora comparte el mismo sistema operativo subyacente.

Los contenedores se consideran más eficientes que las máquinas virtuales desde el punto de vista de los recursos, puesto que no añaden recursos adicionales para cada sistema operativo. Las instancias resultantes son más pequeñas y más rápidas en su creación o migración y un único sistema puede albergar muchos más contenedores que máquinas virtuales.

Sin embargo, hay que tener en cuenta que el sistema operativo único presenta un único punto de error para todos los contenedores que lo utilizan. Por ejemplo, un ataque o bloqueo de *malware* del sistema operativo anfitrión puede inhabilitar o afectar a todos los contenedores. Además, aunque los contenedores son fáciles de migrar, éstos sólo se pueden migrar a otros servidores con núcleos de sistema operativo compatibles.

2.2 Sistemas operativos orientados a contenedores

Tras el éxito y popularidad de las tecnologías de contenerización han surgido sistemas operativos modernos diseñados para funcionar de manera óptima con un ecosistema de contenedores. Estos sistemas operativos proporcionan la mínima funcionalidad requerida para implementar aplicaciones, junto con propiedades de actualización y restablecimiento.

Los sistemas operativos orientados a contenedores han evolucionado el modelo operativo tradicional moviéndose hacia el despliegue de aplicaciones dentro de contenedores, frente a la implementación de aplicaciones en la capa de aplicación. Aquí las aplicaciones son binarios autónomos que se pueden mover en su entorno de contenedor. Esta tecnología también se conoce como la virtualización del sistema operativo en el que su núcleo permite la existencia de múltiples instancias o contenedores de espacio de usuario aisladas, en lugar de una sola.

En el desarrollo de este trabajo se utilizará el sistema operativo orientado a contenedores CoreOS, descrito en el punto 2.2.1.

⁴Zagur. (2016). Introducción a Docker [Parte 0] [Figura]. Recuperado de <http://portallinux.es/introduccion-docker-parte-0>

2.2.1 CoreOS

CoreOS es un sistema operativo ligero de código abierto, de la compañía con el mismo nombre, basado en el núcleo de Linux y lanzado en octubre de 2013. CoreOS proporciona las características necesarias para ejecutar contenedores con un enfoque práctico para las actualizaciones del sistema operativo.



Figura 2.5: CoreOS.⁵

CoreOS está diseñado para implementar una aplicación distribuida en un clúster de nodos. Así, se encarga de proporcionar la infraestructura necesaria para los despliegues en clúster, automatización, seguridad, fiabilidad y escalabilidad. CoreOS ofrece las funcionalidades mínimas necesarias para la implementación de aplicaciones dentro de contenedores de software, junto con mecanismos incorporados para el descubrimiento de servicios y el intercambio de configuración.

Los principales componentes que conforman la arquitectura de CoreOS, presente en la Figura 2.6, son su núcleo, systemd[25], etcd[7], fleet[8] y flannel[9].

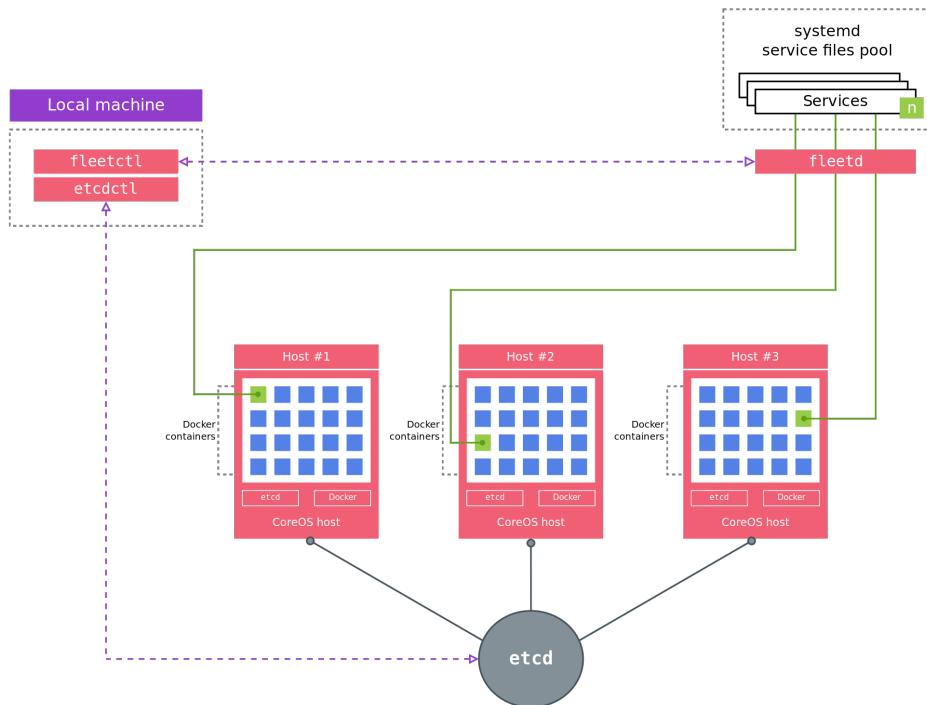


Figura 2.6: Arquitectura CoreOS.⁶

CoreOS no proporciona un gestor de paquetes, por lo que requiere que todas las aplicaciones se ejecuten dentro de sus contenedores. Para ello utiliza Docker y contenedores

⁵CoreOS. (2017). About CoreOS, Inc [Figura]. Recuperado de <https://coreos.com/about>

⁶Wikimedia Commons. (2015). A high-level illustration of the CoreOS cluster architecture [Figura]. Recuperado de https://commons.wikimedia.org/wiki/File:CoreOS_Architecture_Diagram.svg

Linux subyacentes de virtualización para la ejecución de múltiples sistemas Linux aislados en un único sistema anfitrión de control, que sería la instancia CoreOS. De esa manera, la partición de recursos se lleva a cabo a través de múltiples instancias de espacios de usuario aisladas, en lugar de utilizar un hipervisor y una máquina virtual.

A continuación se detallan los microservicios de CoreOS a utilizar en el desarrollo de este trabajo.

Systemd

Systemd es un sistema de inicio que CoreOS utiliza para iniciar, detener y administrar procesos. Como sistema de inicio tiene las funciones de ser el primer proceso en comenzar, controlar el orden y ejecución de todos los procesos de usuario, ocuparse de reiniciarlos si mueren o cuelgan y de sus propiedades y recursos. Si el servicio `systemd` se cancela, todos los procesos asociados con el servicio, incluidos los bifurcados, se eliminan.

Las unidades `systemd` se ejecutan y controlan una sola máquina. Éstas describen una tarea en particular junto con sus dependencias y orden de ejecución. Algunas unidades se inician en el sistema de forma predeterminada y otras por usuarios.

La interfaz de línea de comandos (*CLI*) `systemctl` puede utilizarse para controlar las unidades `systemd`. Éstas pueden ser creadas a partir de una plantilla y utilizarla para instanciar varias unidades.

Algunos tipos de unidad `systemd` son *service* y *socket*. El tipo más común es *service* y se utiliza para definir un servicio con sus dependencias. El tipo *socket* se utiliza para exponer los servicios al mundo externo. Por ejemplo, `docker.service` expone la conectividad externa al motor Docker a través de `docker.socket`. Los *sockets* también se pueden utilizar para exportar registros a máquinas externas.

Etcd

Etcd es un sistema de almacenamiento distribuido de pares clave-valor utilizado por todas las máquinas del clúster CoreOS para leer, escribir e intercambiar datos. Además, se utiliza para compartir la configuración y los datos de monitorización en los equipos CoreOS y para realizar el descubrimiento del servicio. Los demás servicios de CoreOS usan etcd como una base de datos distribuida.

La utilidad `etcdctl` es la interfaz *CLI* para etcd.

Los parámetros de configuración de etcd se pueden usar para modificar propiedades de un sólo miembro etcd o de todo el clúster. Estas opciones son:

- Miembro: Nombre, directorio de datos e intervalo de latido.
- Clúster: Señal de descubrimiento y nodos de clúster iniciales.
- Proxy: Activado/Desactivado e intervalos.
- Seguridad: Certificado y clave.
- Registro: Habilitar/Deshabilitar registro y niveles de registro.

Las operaciones principales que se pueden hacer usando etcd son:

- Establecer, obtener y eliminar operaciones de un par clave-valor.
- Establecer una clave que expire automáticamente tras un tiempo definido.
- Establecer una clave basada en la comprobación de una condición.
- Claves ocultas.
- Observación y espera ante cambios en claves.
- Creación de claves bajo petición.

Fleet

El servicio fleet es un gestor o planificador que controla la creación de servicios a nivel de clúster. Mientras que systemd actúa como sistema de inicio para un nodo, fleet es el sistema de inicio para el clúster y usa el servicio etcd para la comunicación entre nodos.

Al tratarse de una herramienta para la orquestación de contenedores será descrita con mayor detalle en el punto [2.4](#).

Flannel

Flannel utiliza una red de superposición para permitir que contenedores, a través de diferentes anfitriones, se comuniquen entre sí.

Entre sus características cabe destacar que flannel se ejecuta sin un servidor central y utiliza etcd para la comunicación entre los nodos.

Cada nodo del clúster solicita un rango de direcciones IP para los contenedores creados en ese anfitrión y lo registra con etcd. Como cada nodo del clúster conoce el rango de direcciones IP asignado para cada otro nodo, sabe cómo llegar a los contenedores creados en cualquier nodo del clúster. Cuando se crean contenedores, los contenedores obtienen una dirección IP dentro del rango asignado al nodo y si necesitan comunicarse a través de anfitriones, flannel hace la encapsulación basada en el protocolo elegido. Así, en el nodo destino desencapsula el paquete y lo entrega al contenedor.

Las razones por las que se necesita crear redes de contenedores son:

- Los contenedores necesitan comunicarse con el mundo exterior.
- Los contenedores deben ser accesibles desde el mundo exterior para que éste pueda utilizar los servicios que proporcionan.
- Los contenedores necesitan comunicarse con la máquina anfitriona.
- Debería haber conectividad entre contenedores en el mismo anfitrión y entre anfitriones.

Confd

La herramienta confd está diseñada para ver cambios en los almacenes distribuidos de clave-valor. Se ejecuta dentro de un contenedor Docker y se utiliza para activar modificaciones de configuración y recargas de servicio.

Esta herramienta de gestión de configuración está construida en la parte superior de etcd. Así, confd puede ver ciertas claves en etcd y actualizar los archivos de configuración relacionados a ellas tan pronto como cambie. Luego, confd puede volver a cargar o reiniciar las aplicaciones relacionadas con los archivos de configuración actualizados. Esto permite automatizar los cambios de configuración en todos los servidores del clúster y asegura que todos los servicios están siempre buscando la última configuración.

2.3 Otros

Otros sistemas operativos orientados a contenedores son Red Hat Enterprise Linux (RHEL) Project Atomic[[11](#)] y Snappy Ubuntu Core[[12](#)].

Project Atomic, lanzado en 2014, facilita la arquitectura centrada en la aplicación proporcionando una solución para desplegar aplicaciones basadas en contenedores de forma rápida y confiable. La actualización atómica y la reversión de aplicaciones y anfitriones permiten la implementación de pequeñas mejoras frecuentes.



Figura 2.7: Project Atomic.⁷

Por su parte Snappy Ubuntu Core, lanzado en 2014, es un sistema operativo que usa una imagen de servidor mínima con las mismas bibliotecas que el sistema operativo Ubuntu actual. Su enfoque ágil es más rápido, más fiable y permite ofrecer mayores garantías de seguridad para aplicaciones y usuarios. Las aplicaciones se pueden actualizar de forma atómica y revertirse si es necesario, pensando en contenedores.



Figura 2.8: Ubuntu Core.⁸

2.4 Herramientas para la orquestación de contenedores

Un sistema de orquestación de contenedores trata el hardware dispar de la infraestructura como una colección y lo representa para la aplicación como un único recurso. También, programa los contenedores basándose en las restricciones de los usuarios y utiliza la infraestructura de la manera más eficiente posible, escalando los contenedores dinámicamente y manteniendo los servicios en alta disponibilidad.

La herramienta para la orquestación de contenedores usada en el presente trabajo es Fleet, descrita en el punto 2.4.1.

2.4.1 Fleet

Como se comentó en el punto 2.2.1, fleet es un componente de CoreOS que controla la creación de servicios a nivel de clúster.

Fleet utiliza el modelo maestro-esclavo, donde el motor fleet desempeña el papel de maestro y el agente fleet representa el papel de esclavo. El motor es responsable de programar las unidades fleet y el agente de ejecutarlas y divulgar su estado al motor. Las unidades fleet también constan de metadatos para controlar dónde se ejecuta la unidad con respecto a la propiedad del nodo, así como la base de otros servicios que se ejecutan en ese nodo en particular. El agente procesa la unidad y la ofrece a systemd para su ejecución. Si un nodo muere se elige un nuevo motor y las unidades de programa en ese nodo se reprograman en un nuevo nodo. Systemd proporciona alta disponibilidad a nivel de nodo, mientras que fleet lo hace a nivel de clúster. Así, fleet se utiliza principalmente para la orquestación de servicios críticos del sistema, usando systemd.

Los metadatos se pueden utilizar en archivos de servicio fleet para controlar la programación y las opciones X-Fleet se usan para especificar restricciones al programar el servicio. Entre las disponibles se encuentran:

⁷GitHub Project Atomic. (2017). Project Atomic [Figura]. Recuperado de <https://github.com/projectatomic>

⁸(2016). Ubuntu Core 16 delivers foundation for secure IoT [Figura]. Recuperado de <https://insights.ubuntu.com/2016/11/03/ubuntu-core-16-delivers-foundation-for-secure-iot>

- *MachineMetaData*: El servicio se programa basado en metadatos coincidentes.
- *Global*: El mismo servicio se programa en todos los nodos del clúster.

2.4.2 Otras

Otras herramientas para la orquestación de contenedores en la actualidad son Kubernetes[10], Docker Swarm[18] y Apache Mesos[19].

Kubernetes

Kubernetes es una plataforma de código abierto para la orquestación de contenedores, iniciada por Google.



Figura 2.9: Kubernetes.⁹

La unidad más pequeña en Kubernetes es un *pod*. Los *pods* son un conjunto de contenedores que se encuentran juntos en un único nodo. Cada *pod* tiene una dirección IP y todos los contenedores de ese *pod* la comparten. Kubernetes sigue la arquitectura maestro-esclavo.

Como se muestra en la Figura 2.10, Kubernetes se compone de múltiples servicios. Kubelet es responsable del estado de ejecución de cada nodo. Se ocupa de iniciar, detener y mantener los contenedores de aplicaciones. Kube-proxy se ocupa de la redirección de servicios y el balanceo de carga del tráfico en los *pods*. El servidor *API* sirve a la *API* estándar utilizando *JSON* sobre *HTTP*, proporcionando tanto la interfaz interna como externa a Kubernetes. Así, el servidor *API* procesa y valida las peticiones *REST* y actualiza el estado de los objetos *API* en etcd, permitiendo a los clientes configurar cargas de trabajo y contenedores. El planificador es el componente conectable que selecciona en qué nodo debería ejecutarse un *pod* no programado basado en la disponibilidad de recursos y las cargas de trabajo existentes. Por su parte, el controlador de replicación es necesario para mantener la alta disponibilidad de *pods* y crear varias instancias.

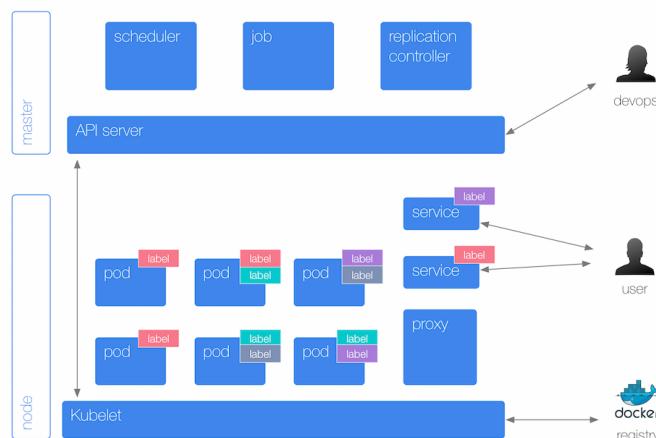


Figura 2.10: Arquitectura Kubernetes.¹⁰

⁹(2017). Kubernetes logo [Figura]. Recuperado de <https://opencredo.com/kubernetes>

Docker Swarm

Docker Swarm es la solución de orquestación nativa de Docker.



Figura 2.11: Docker Swarm.¹¹

Como se muestra en la Figura 2.12, con su uso se administra el clúster como entidad en lugar de administrar nodos individuales. Esta herramienta tiene un planificador integrado que decide la ubicación de los contenedores en el clúster y utiliza restricciones y afinidades específicas del usuario para decidir esa ubicación. En su arquitectura existe un maestro que se encarga de la planificación de contenedores Docker basado en el algoritmo planificador, restricciones y afinidades. Para proporcionar alta disponibilidad se pueden ejecutar múltiples maestros en paralelo, distribuyendo las cargas de trabajo uniformemente. Los agentes se ejecutan en cada nodo y se comunican con el maestro a partir del descubrimiento, necesario porque se ejecutan en diferentes nodos no iniciados por él.

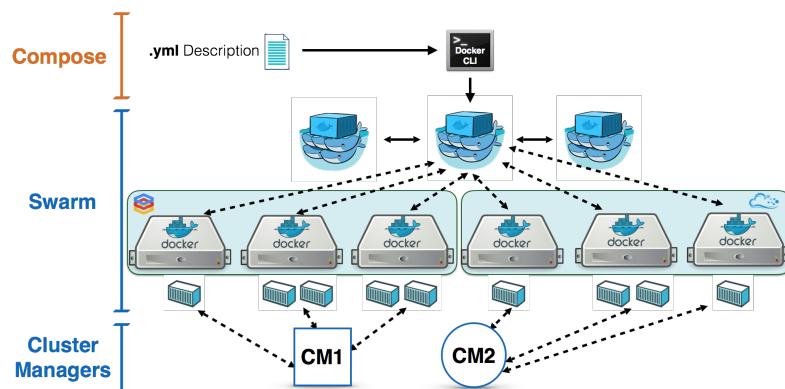


Figura 2.12: Composición de maestros y agentes en Docker Swarm.¹²

Apache Mesos

Apache Mesos, desarrollado en la Universidad de California Berkeley, combina el sistema operativo con el administrador de clústeres.



Figura 2.13: Apache Mesos.¹³

¹⁰Mhausenblas. (2016). Kubernetes Community Resources [Figura]. Recuperado de <http://k8s.info/cs.html>

¹¹(2017). Swarm: a Docker-native clustering system [Figura]. Recuperado de <https://github.com/docker/swarm>

¹²Alba. (2015). Deploy and Manage Any Cluster Manager with Docker Swarm [Figura]. Recuperado de <https://blog.docker.com/2015/11/deploy-manage-cluster-docker-swarm>

Como se muestra en la Figura 2.14, Mesos consiste en un demonio maestro que gestiona los demonios agentes que se ejecutan en cada nodo del clúster y los marcos de Mesos que ejecutan tareas en estos agentes. El maestro permite un uso compartido de recursos a través de marcos. Cada oferta de recursos contiene una lista de agentes. El maestro decide cuántos recursos ofrecer a cada marco de acuerdo con una determinada política de organización. Para soportar un conjunto diverso de políticas, el maestro emplea una arquitectura modular que facilita la adición de nuevos módulos de asignación a través de un mecanismo de complemento. Un *framework* que se ejecuta en la parte superior de Mesos consta de dos componentes: un programador que se registra con el maestro y un proceso ejecutor que se ejecuta en los nodos del agente para ejecutar las tareas del marco. Mientras que el maestro determina cuántos recursos se ofrecen a cada marco, los planificadores de los marcos seleccionan cuál de los recursos ofrecidos utilizar.

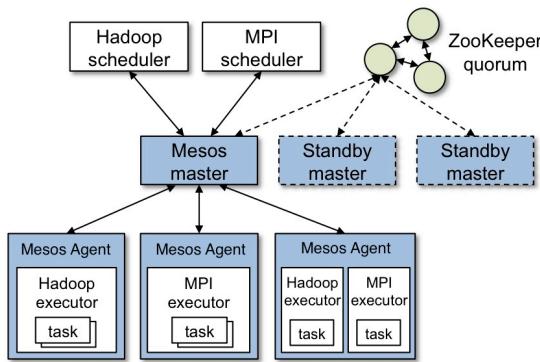


Figura 2.14: Abstracción de nodos en Apache Mesos.¹⁴

2.5 Proveedores de Infraestructura como Servicio

La Infraestructura como Servicio (*IaaS*) es uno de los tres modelos fundamentales en el campo de la computación en la nube, junto con la Plataforma como Servicio (*PaaS*) y el Software como Servicio (*SaaS*). Se trata de una infraestructura informática inmediata que se aprovisiona y administra a través de una conexión pública, normalmente Internet.

Este modelo de servicio proporciona acceso a recursos informáticos situados en un entorno virtualizado, la nube. De esta manera permite reducir o escalar recursos concretos con rapidez para ajustarlos a la demanda, pagando por uso y evitando el gasto y complejidad que suponen la compra y administración de una infraestructura física propia.

Existen proveedores de estos servicios informáticos en la nube que administran la infraestructura, mientras que el cliente solo tiene que configurar y administrar su propio software.

Los recursos informáticos ofrecidos consisten en hardware virtualizado o infraestructura de procesamiento. La definición de *IaaS* abarca aspectos como el espacio en servidores virtuales, conexiones de red, ancho de banda, direcciones IP y balanceadores de carga. Físicamente, el repertorio de recursos hardware disponibles procede de multitud de servidores y redes, generalmente distribuidos entre numerosos centros de datos.

¹³(2017). OpenCredo are Experts in Apache Mesos [Figura]. Recuperado de <https://opencredo.com/expertise/experts-in-apache-mesos>

¹⁴Apache Mesos. (2017). Mesos Architecture [Figura]. Recuperado de <http://mesos.apache.org/documentation/latest/architecture>

Entre las ventajas de *IaaS* se encuentran:

- Eliminación del gasto en capital y corriente eléctrica.
- Adquisición de nuevos recursos con rapidez.
- Respuesta rápida ante cambios de demanda, aumentando y reduciendo recursos.
- Mayor eficiencia, puesto que se garantiza que se utiliza la capacidad máxima de la infraestructura física.
- Mayor seguridad y protección de los recursos de información.

Los ejemplos más conocidos son Amazon Web Services[14] y DigitalOcean[13].

DigitalOcean, creado en 2011 por Ben y Moisey Uretsky, es un proveedor Estadounidense de servidores virtuales privados.



Figura 2.15: DigitalOcean.¹⁵

Entre sus características más destacadas está que sus servidores en la nube, llamados *droplets*, pueden ser provisionados típicamente en 55 segundos. Además, provee discos duros SSD de alto rendimiento y virtualización *KVM*. En general aporta servicios mensuales por 5 dólares al mes, continuando con facturación por hora.

En la ejecución de este proyecto se aplicará el uso de Amazon Web Services como proveedor de Infraestructura como Servicio, detallado en el punto 2.5.1.

2.5.1 Amazon Web Services

Amazon Web Services (AWS) es una plataforma de servicios de computación en la nube ofrecida a través de Internet por Amazon.com y lanzada en 2006.



Figura 2.16: Amazon Web Services.¹⁶

AWS está situado en distintas regiones geográficas. Cada región está totalmente contenida dentro de un solo país y todos sus datos y servicios permanecen dentro de ella. Cada una tiene múltiples zonas de disponibilidad con diferentes centros de datos que proporcionan servicios de AWS.

Este proveedor dispone de una capa gratuita diseñada para permitir obtener experiencia práctica con los servicios en la nube de AWS. Ésta incluye 12 meses a partir de la fecha de inscripción, así como ofertas de servicios adicionales que no vencen al final de este período. No obstante existen límites de uso. A partir de este primer año se empieza a pagar por horas de uso.

En la elaboración de este trabajo se utilizarán los siguientes servicios:

¹⁵DigitalOcean (2017). Logos and badges [Figura]. Recuperado de <https://www.digitalocean.com/company/logos-and-badges>

¹⁶Amazon Web Services (2017). AWS Co-Marketing Tools [Figura]. Recuperado de <https://aws.amazon.com/co-marketing>

AWS Identity and Access Management (IAM)

AWS Identity and Access Management (IAM)^[15] es un servicio web gratuito que permite controlar de forma segura el acceso a los recursos de AWS por parte de los usuarios. Facilita el acceso compartido a la cuenta AWS, concediendo permisos a otros usuarios para administrar y utilizar los recursos sin tener que compartir la clave de acceso, y la concesión de permisos diferentes a distintos usuarios para diferentes recursos.

Amazon Elastic Compute Cloud (Amazon EC2)

Amazon Elastic Compute Cloud (Amazon EC2)^[16] proporciona la capacidad de computación escalable en la nube de AWS. Permite realizar el despliegue de servidores virtuales, configurar seguridad, redes y administrar el almacenamiento. Este servicio permite escalar o reducir los requisitos bajo demanda o picos de popularidad. La principal característica es el entorno virtual de computación o instancia. Además, ofrece otras funciones como plantillas preconfiguradas, varios tipos de instancias o configuraciones, inicio de sesión seguro utilizando pares de claves, volúmenes de almacenamiento temporales, regiones y zonas de disponibilidad y grupos de seguridad.

Amazon Virtual Private Cloud (Amazon VPC)

Amazon Virtual Private Cloud (Amazon VPC)^[17] permite lanzar los servicios AWS en una red virtual definida. Una nube virtual privada (VPC) es una red virtual dedicada a la cuenta de AWS. Está aislada de otras redes virtuales en la nube AWS y permite iniciar los recursos en ella. Permite seleccionar un rango de direcciones IP, crear subredes, configurar tablas de rutas, pasarelas de red y realizar otras configuraciones de seguridad. Otro elemento a usar es la subred que es un rango de direcciones IP en la VPC. Los grupos de seguridad se utilizan para proteger los recursos de AWS en cada subred.

2.6 Otras herramientas tecnológicas

A continuación se describirán otra serie de herramientas necesarias para la elaboración de este trabajo.

2.6.1 Vagrant

Vagrant^[21] es una herramienta de código abierto para crear y configurar entornos de desarrollo portátiles y virtualizados, centrada en la automatización.



Figura 2.17: Vagrant.¹⁷

¹⁷Wikipedia (2013). Vagrant (software) [Figura]. Recuperado de [https://es.wikipedia.org/wiki/Vagrant_\(software\)](https://es.wikipedia.org/wiki/Vagrant_(software))

Vagrant fue creada en enero de 2010 por Mitchell Hashimoto, que en 2012 creó la organización HashiCorp para respaldar su desarrollo a tiempo completo, contando con la contribución de una fuerte comunidad de desarrolladores. Está escrito en lenguaje Ruby pero puede ser utilizado en proyectos escritos en otros lenguajes de programación.

La idea central detrás de su creación reside en el hecho de que el mantenimiento del entorno de desarrollo se hace cada vez más difícil a medida que el proyecto crece. De esta manera, Vagrant proporciona ambientes de trabajo menos complejos de configurar, reproducibles y portátiles.

Esta herramienta utiliza aprovisionadores y proveedores como bloques de construcción para administrar los entornos de desarrollo. Los primeros son herramientas que permiten a los usuarios personalizar la configuración de entornos virtuales, mientras que los proveedores son los servicios que Vagrant utiliza para lanzar y crear los propios entornos virtuales.

Para su puesta en marcha se utiliza un fichero de configuración denominado **Vagrantfile**. Su función principal es describir el tipo de máquina requerida y cómo configurarla y proveerla. Por otro lado, existen cajas que son el formato de paquetes para los ambientes Vagrant. Una caja puede ser utilizada por cualquier persona en cualquier plataforma que soporte Vagrant para crear un entorno de trabajo idéntico.

2.6.2 VirtualBox

VirtualBox[22] es un software de virtualización de código abierto para arquitecturas *x86/amd64*.



Figura 2.18: VirtualBox.¹⁸

Este hipervisor de tipo II o *hosted*, se caracteriza porque permite instalar sistemas operativos adicionales, conocidos como sistemas invitados o máquinas virtuales, dentro de otro sistema operativo, sistema anfitrión, cada uno con su propio ambiente virtual.

VirtualBox fue creado originalmente por la empresa alemana Innotek GmbH en enero de 2007. Actualmente es desarrollado por Oracle Corporation y existen dos versiones gratuitas. Por un lado la llamada Oracle VM VirtualBox, sujeta a la licencia de "Uso Personal y de Evaluación VirtualBox". Por otro lado, VirtualBox OSE, sujeta a la licencia GPL.

Entre sus características más importantes se destaca que es multiplataforma, en referencia a las arquitecturas soportadas; multi huéspedes, por la variedad y cantidad de sistemas operativos que puede virtualizar; y ofrece portabilidad, al poder importar y exportar las máquinas virtuales a otros sistemas.

2.6.3 GitHub

GitHub[20] es una plataforma de desarrollo colaborativo de software para alojar proyectos utilizando el sistema de control de versiones Git. El código se almacena de forma pública, aunque también se puede hacer de forma privada, creando una cuenta de pago.

¹⁸Wikipedia (2010). VirtualBox logo [Figura]. Recuperado de https://commons.wikimedia.org/wiki/File:Virtualbox_logo.png



Figura 2.19: GitHub.¹⁹

Esta herramienta permite alojar repositorios de código y ofrece utilidades para el trabajo en equipo, dentro de un proyecto, algunas de las cuales son:

- Wiki: Mantenimiento de distintas versiones de páginas.
- Sistema de seguimiento de problemas: Permite a los miembros de un equipo detallar un problema con el software o una sugerencia.
- Visor de ramas: Permite comparar los progresos realizados en las distintas ramas del repositorio.

Además, se puede contribuir a mejorar el software de otros usuarios, a partir de la creación de programas de código abierto que fomentan el software libre.

2.6.4 Travis CI

Travis CI^[24] es una compañía y sistema distribuido que ofrece servicios de Integración y Despliegue continuos (CI/CD) gratuitos o de pago.



Figura 2.20: Travis CI.²⁰

El servicio gratuito permite conectar el repositorio de GitHub público para poder regenerar el proyecto tras cada cambio. Cuando lo detecta utiliza el fichero de configuración `.travis.yml` para realizar las acciones descritas en él. Este fichero contiene las precondiciones, condiciones y postcondiciones necesarias para construir y desplegar la aplicación de manera automática. Ejemplos de dichas acciones son la comprobación de los tests, construir una nueva imagen Docker y subirla al repositorio Docker Hub.

Entre los beneficios que aporta están la reducción de riesgos y tiempo, de procesos manuales repetitivos, la obtención de una versión de software mediante un proceso conocido, confiable, probado, versionado y repetible y la mejora la visibilidad del estado del proyecto. Además, mantiene la calidad de todo el sistema mediante la ejecución periódica de los tests, identificando problemas y ayudando a corregirlos antes de pasar a producción.

2.6.5 Softcover

Para la elaboración de la memoria se utiliza la gema Ruby Softcover^[23] que es un sistema de composición de libros electrónicos para autores técnicos. Depende de otra gema, polytexnic, para convertir la entrada de *Markdown* o *PolyTeX* a *HTML* y *LaTeX*, y de ahí a *EPUB*, *MOBI* y *PDF*.

¹⁹GitHub (2017). GitHub Logos and Usage [Figura]. Recuperado de <https://github.com/logos>

²⁰Travis CI (2017). Logo [Figura]. Recuperado de <https://travis-ci.com/logo>

Capítulo 3

Desarrollo

En este capítulo se expone el análisis de la aplicación web Ruby on Rails, de 3 capas, utilizada para el desarrollo del presente trabajo. A partir de la misma se irán elaborando una serie de iteraciones en las que se detallarán las decisiones de diseño y desarrollo tomadas para cada una de ellas. El resultado final de esta fase será la obtención de la infraestructura como código que permita realizar el despliegue de la aplicación con las tecnologías Docker y CoreOS en la nube pública de Amazon Web Services, así como en el proveedor VirtualBox.

3.1 Análisis de la aplicación Ruby on Rails

La aplicación web `sample_app_rails_4` es parte de un tutorial[2] sobre el uso del *framework* de desarrollo de aplicaciones web Ruby on Rails. Está desarrollada mediante una combinación de simulaciones, pruebas de desarrollo TDD y pruebas de integración.

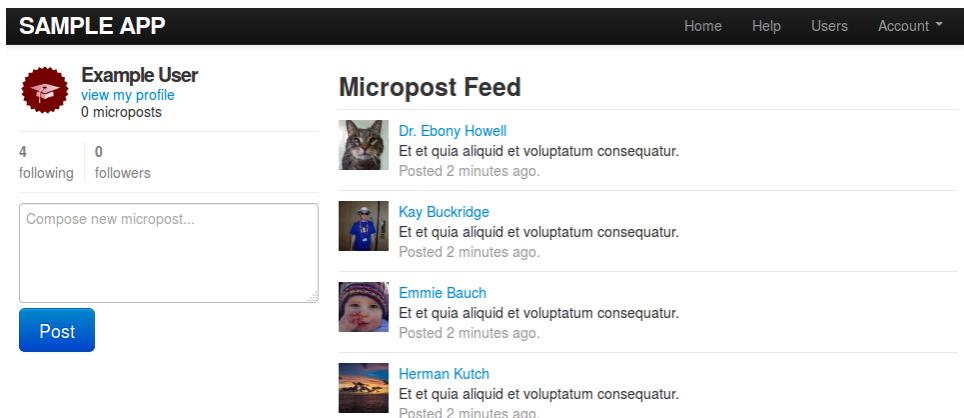


Figura 3.1: Aplicación web `sample_app_rails_4`.

Tiene una arquitectura de 3 capas: cliente, aplicación y base de datos. Está creada a partir de páginas estáticas con contenido dinámico. Tiene un diseño de sitio web, un modelo de datos de usuario y un sistema completo de registro y autenticación, incluida la activación de cuentas y restablecimiento de contraseñas. Además, cuenta con funciones de *microblogging* y redes sociales. Así, la aplicación tendrá usuarios que crearán *microposts* dentro de un marco de autenticación e inicio de sesión, donde la interfaz gráfica de la página principal de un usuario concreto se visualizará como muestra la Figura 3.1.

La arquitectura Ruby on Rails de esta aplicación se pueden apreciar en la Figura 3.2. Sus características son:

- Arquitectura Modelo-Vista-Controlador (MVC): Mejora la capacidad de mantenimiento, desacoplamiento y pruebas de la aplicación.
- Arquitectura *Representational State Transfer* (REST) para servicios web.
- Soporta las principales bases de datos como MySQL y PostgreSQL.

- Generadores de *scripts* para automatizar tareas.
- Uso del formato de serialización de datos *YAML*.

Las características se distribuyen en los siguientes componentes de Rails:

- *Action Mailer*: Proporciona servicios de correo electrónico.
- *Action Pack*: Capta las solicitudes de usuario realizadas por el navegador y las asigna a acciones definidas en la capa de controladores.
 - *Action Controller*: Enruta solicitudes al controlador.
 - *Action Dispatcher*: Controla, analiza y procesa el enrutamiento de la solicitud del navegador web.
 - *Action View*: Realiza la presentación de la página web solicitada.
- *Active Model*: Define la interfaz entre el *Action Pack* y los módulos *Active Record*.
- *Active Record*: Proporciona la capacidad de crear relaciones o asociaciones entre modelos y construye la capa Modelo que conecta las tablas de la base de datos con su representación en las clases Ruby.
- *Active Resource*: Administra la conexión entre servicios web *RESTful* y objetos de negocio.
- *Active Support*: Colección de clases de utilidad y extensiones de bibliotecas estándar de Ruby útiles para el desarrollo en Ruby on Rails.
- *Railties*: Código básico de Rails que construye nuevas aplicaciones.

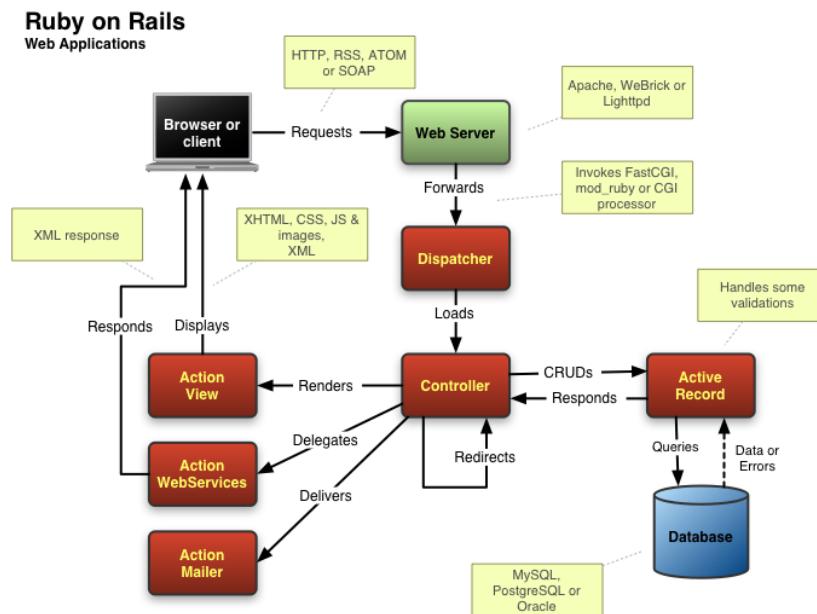


Figura 3.2: Integración entre características y componentes de Ruby on Rails.¹

Una vez vistas las características y componentes de la aplicación Ruby on Rails en estudio, se elabora un trabajo adicional a partir de ella, aplicando una serie de cambios y adiciones para la consecución de los objetivos inicialmente propuestos.

¹Mejia (2011). Ruby on Rails Architectural Design [Figura]. Recuperado de <http://adrianmejia.com/blog/2011/08/11/ruby-on-rails-architectural-design>

3.2 Iteración 1: Conversión a una arquitectura de microservicios con el uso de Docker

Como se muestra en la Figura 3.3, la primera iteración consiste en adecuar la aplicación para que utilice contenedores en su funcionamiento. Esta tarea supone el uso de la tecnología de contenedores Docker.

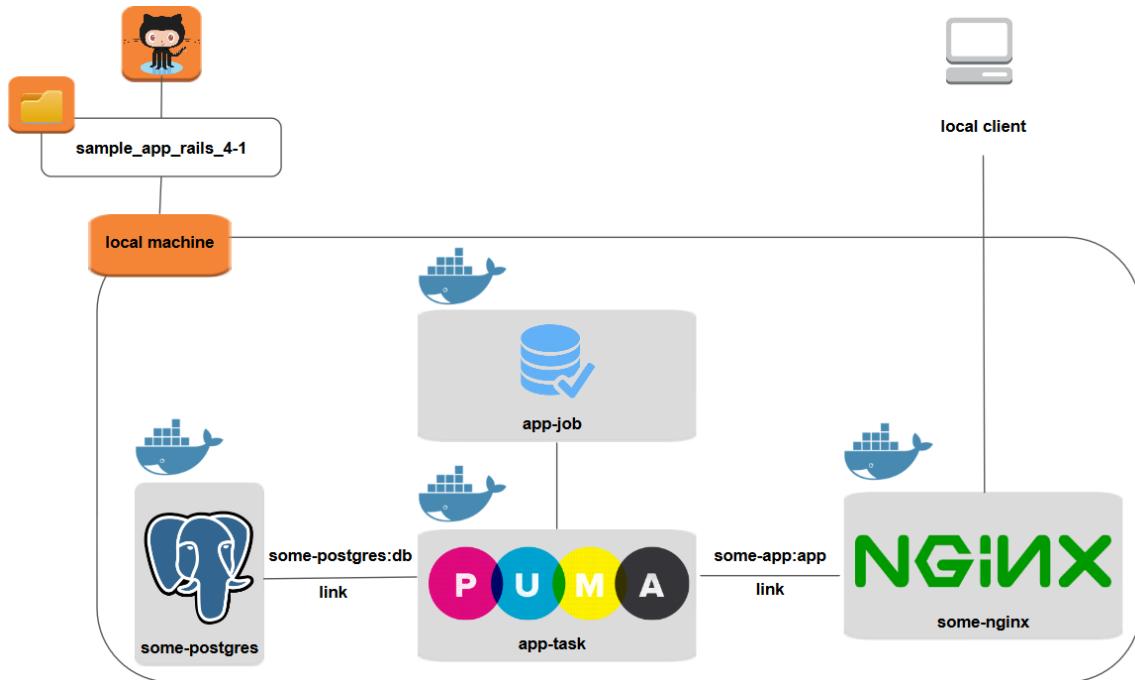


Figura 3.3: Infraestructura de la aplicación con contenedores Docker.

La aplicación **sample_app_rails_4** está disponible en un repositorio GitHub. Para trabajar con ella se hará una copia del repositorio a través de la opción *fork*. Esto crea una bifurcación que permite la libre experimentación de cambios, sin afectar el proyecto original, utilizando el proyecto de otra persona como punto de partida de una idea propia.

Mediante Docker se obtienen las imágenes pertenecientes a los servicios **PostgreSQL** y **Nginx**. El primero implementa la funcionalidad de la base de datos y el segundo el servidor web y proxy inverso. El proxy inverso es un proxy que aparenta ser un servidor web ante los clientes, pero que en realidad reenvía las solicitudes que recibe a uno o más servidores de origen. Se escoge **Nginx** por ser multiplataforma, ligero, de alto rendimiento y software libre.

La aplicación en su origen utiliza una base de datos **SQLite** que se cambia por **PostgreSQL**, presente en el contenedor **some-postgres**.

Otro de los cambios a implementar es la sustitución del servidor web **rails s** por **puma**, construido para ofrecer mayor velocidad y paralelismo.

Para crear un contenedor que contenga la aplicación habrá que crear su imagen Docker, que se llamará **sample_app_rails_4_image**. A efectos de mantenerla remotamente se subirá al repositorio de imágenes Docker Hub con la etiqueta **initial**.

El principal propósito de este despliegue es que pueda realizarse automáticamente tras la ejecución de un *script*. Este fichero comprobará que las variables de entorno que especifican el nombre y la contraseña del usuario de las distintas bases de datos PostgreSQL existentes (test,

desarrollo y producción) están correctamente establecidas.

Luego se creará el contenedor `some-postgres` con el servidor `PostgreSQL` y, seguidamente, se creará el contenedor de la aplicación web, llamado `some-app`, a partir de la imagen de ésta, enlazándola con el contenedor `some-postgres`, que será su base de datos.

Con la intención de utilizar como servidor web de la aplicación un contenedor diferente al propio, se crea el contenedor `some-nginx` que proporciona el servidor `Nginx`. Se le indicará que el tráfico del puerto 8080 en el sistema anfitrión se redirija al 80 del contenedor. Así, el sitio web será accesible por la dirección local en el puerto 8080. Además, se enlazará a la aplicación web mediante el contenedor en el que se ejecuta.

También se crea un volumen Docker de datos, llamado `volume-public`, para compartirlo entre la aplicación web, la base de datos y el servidor proxy.

Con todo ello, quedará construir, migrar y poblar la base de datos para, finalmente, ejecutar el servidor `puma` dentro del contenedor `some-app` y comprobar desde el sistema anfitrión que la página principal de la aplicación `sample_app_rails_4` está disponible en el puerto 8080. Esto producirá el resultado de la Figura 3.7.

3.2.1 Preparación del repositorio local y remoto

En primer lugar se realiza un *fork* del repositorio GitHub de la aplicación `sample_app_rails_4` y se clona esta copia para trabajar localmente:

```
$ git clone https://github.com/CarolinaSantana/sample_app_rails_4-1.git
```

Luego se entra en la carpeta local que lo contiene, seleccionando como *gemset*, conjunto aislado de gemas incorporadas en la aplicación para la versión Ruby en uso, `2.0.0@railstutorial_rails_4_0`. Además se deja lista la configuración de base de datos de ejemplo, con la intención de probar que los test que comprueban su funcionalidad pasen positivamente:

```
$ cd sample_app_rails_4-1
$ cp config/database.yml.example config/database.yml
$ bundle install --without production
$ bundle exec rake db:migrate
$ bundle exec rake db:test:prepare
$ bundle exec rspec spec/
```

3.2.2 Cambio y configuración de la base de datos

Con la intención de realizar el cambio de la base de datos se elimina del fichero `Gemfile` la gema `sqlite3 1.3.8`, como base de datos en desarrollo, y se cambia por una base de datos PostgreSQL, concretamente la versión `pg 0.15.1`. Para instalar la nueva gema se ejecuta:

```
$ bundle install --without production
```

La instalación de `PostgreSQL` requiere credenciales de usuario con contraseña para acceder y se especifica mediante las variables de entorno `$POSTGRES_USER` y `$POSTGRES_PASSWORD` en el fichero local `~/.postgres/credentials`, cuyo contenido es:

Listado 3.1: Fichero `~/.postgres/credentials`

```
export POSTGRES_USER=postgres
export POSTGRES_PASSWORD=postgres
```

Se da permiso de ejecución tanto al fichero como a la carpeta que lo contiene y se exportan las variables de entorno en el directorio de la aplicación:

```
$ chmod 0700 ~/.postgres && chmod 0700 ~/.postgres/credentials
$ . ~/.postgres/credentials
```

El fichero de configuración de la base de datos ha de especificar lo siguiente:

Listado 3.2: Cambios en el fichero `config/database.yml`

```
default: &default
  adapter: postgresql
  encoding: unicode
  pool: 5
  username: <%= ENV['POSTGRES_USER'] %>
  password: <%= ENV['POSTGRES_PASSWORD'] %>
  port: 5432
development:
  <<: *default
  database: sample_app_development
  host: db
test:
  <<: *default
  database: sample_app_test
  host: db
production:
  <<: *default
  database: sample_app_production
```

El nombre de usuario y contraseña de la base de datos de desarrollo, test y producción se indica con variables de entorno. Además, se especifica que ha de conectarse a *db*, que será el nombre por el que descubrir el contenedor Docker que provisiona el servidor PostgreSQL.

Por motivos de seguridad el fichero de configuración de la base de datos no debe subirse al repositorio remoto, por lo que se hace una copia para tener esta configuración de ejemplo:

```
$ cp config/database.yml config/database.yml.postgres
```

3.2.3 Cambio del alimentador idempotente de base de datos

Cuando se alimenta la base de datos solo han de crearse datos en ella si éstos aún no existen. Para ello se cambia a *seed* idempotente el fichero que viene por defecto, `lib/tasks/sample_data.rake`, quedando como:

Listado 3.3: Fichero `lib/tasks/sample_data.rake`

```
namespace :db do
  desc "Fill database with sample data"
  task populate: :environment do
    make_users_microposts_relationships
```

```

    end
end
def make_users_microposts_relationships
  User.find_or_create_by(email:"example@railstutorial.org") do |admin|
    admin.name = "Example User"
    admin.password = "foobar"
    admin.password_confirmation = "foobar"
    admin.admin = true
  99.times do |n|
    name = Faker::Name.name
    email = "example-#{n+1}@railstutorial.org"
    password = "password"
    User.create!(name: name,
                email: email,
                password: password,
                password_confirmation: password)
  end
  users = User.all(limit: 6)
  50.times do
    content = Faker::Lorem.sentence(5)
    users.each { |user| user.microposts.create!(content: content) }
  end
  users = User.all
  user = users.first
  followed_users = users[2..50]
  followers = users[3..40]
  followed_users.each { |followed| user.follow!(followed) }
  followers.each { |follower| follower.follow!(user) }
end
end

```

3.2.4 Cambio y configuración del servidor web

Para acceder a la aplicación se escoge cambiar el servidor web `rails s` por `puma`. Para ello se añade al fichero `Gemfile` la gema `puma` y se instala:

```
$ bundle install --without production
```

3.2.5 Configuración para la creación de los contenedores

En primer lugar se configuran los registros o *logs* que crearán los contenedores añadiendo en el fichero `config/application.rb` lo siguiente:

Listado 3.4: Fichero `config/application.rb`

```
.
config.logger = Logger.new(STDOUT)
.
```

Para que la imagen Docker que se construya del repositorio sea más ligera se añade el fichero `.dockerignore`, como copia de `.gitignore`:

Listado 3.5: Fichero `.dockerignore`

```
.
cp .gitignore .dockerignore
.
```

La creación de los contenedores se lleva a cabo ejecutando el script `docker-microservice.sh`.

Primero comprueba que las variables de entorno con el nombre y la contraseña del usuario de la base de datos PostgreSQL están establecidas.

Para crear el contenedor de la aplicación hay que construir su imagen, `sample_app_rails_4_image`, previamente. El fichero `Dockerfile` especifica cómo crearla, indicando que lo hará a partir de una imagen Ruby y que debe actualizar todos los paquetes e instalar `nodejs`, requerido por la aplicación, y `netcat`, para comprobar que el servicio PostgreSQL está listo, terminando con una limpieza de las capas intermedias.

La lógica de construcción de la aplicación, basada en su imagen, pasa por la creación de dos contenedores. El primer contenedor, `app-job`, se configurará a partir de un *entrypoint*, llamado `setup.sh`, que va a permitir ejecutar el contenedor como un ejecutable. Este contenedor esperará a que el servicio *PostgreSQL* esté activo en el contenedor `some-postgres` para crear, migrar, alimentar y poblar las bases de datos. Además, la aplicación implementa el modo de autenticación por medio de *tokens*. Cuando el cliente se autentica mediante su usuario y contraseña cada petición HTTP que hace irá acompañada de un *token* en la cabecera, permitiendo que sea identificado. En el actual entorno de desarrollo va a ser utilizado un determinado *token* de prueba. Para ello se realiza la copia del actual `.secret`, localizado en el directorio principal, como `.secret.example`. Luego se incluye como *token* en el fichero. Una vez acaba, el contenedor `app-task` vuelve al estado inactivo.

Listado 3.6: Fichero `setup.sh`

```
#!/bin/sh

cp config/database.yml.postgresql config/database.yml
echo "Waiting PostgreSQL to start on 5432..."
while ! nc -z some-postgres 5432; do
    sleep 0.1
done
echo "PostgreSQL started"
echo "Creating databases..."
rake db:create
cp ./secret.example ./secret
echo "Migrating to databases..."
rake db:migrate
echo "Seeding databases..."
rake db:seed
echo "Populating databases..."
rake db:populate
echo "Ready databases"
```

Así como se tiene el contenedor *job* se necesita un segundo contenedor *task*, llamado `app-task`. Este contenedor se construye tras el anterior.

Listado 3.7: Contenido de `Dockerfile`

```
FROM ruby:2.0-onbuild
LABEL sample_app_rails_4_image.version="0.1"
      sample_app_rails_4_image.release-date="2016-12-10"
MAINTAINER Carolina Santana "c.santanamartel@gmail.com"
RUN apt-get update && apt-get -y install nodejs && \
    apt-get -y install netcat && \
    apt-get autoclean && apt-get clean && \
    rm -rf /var/lib/apt/lists/* && \
    rm -f /tmp/* /var/tmp/*
```

Con la intención de exportar un volumen compartido entre los contenedores de la aplicación y `some-nginx` para que comparten el directorio `/usr/src/app/public` y el directorio `/var/lib/postgresql` con el contenedor `some-postgres` se crea un volumen Docker llamado `volume-public`.

La siguiente acción es la creación del contenedor `some-postgres` con el servidor **PostgreSQL**, a partir de la imagen `postgres` indicada con `-d`. Para acceder a la base de datos es necesario pasarle las credenciales, con `-e`. El volumen creado se exporta mediante `-v`.

Ahora se crea el contenedor *job* de la aplicación web, llamado `app-job`, a partir de la imagen de ésta, `-d`, indicando como `-entrypoint` el *script* comentado y enlazando la base de datos de la misma con el contenedor `some-postgres` a través del indicador `-link`. Además, se le pasan las variables de entorno para acceder a la base de datos, con `-e`. Por su parte, el indicador `-it` ofrece un terminal interactivo dentro del contenedor y `-w` indica que se va a compartir el directorio inicial del proyecto local en el contenedor. El volumen creado se exporta con `-v`.

El siguiente paso crea el contenedor *task* de la aplicación web, llamado `app-task`. Se construye como el anterior sin el *entrypoint*, configurando y preparando las bases de datos, el *token* de prueba y ejecutando el servidor `puma` en el puerto **9292**. Todo ello a través de `/bin/bash -c`

Para crear el contenedor `some-nginx` que proporciona el servidor Nginx se indica la imagen con `-d`, que el tráfico del puerto 8080 en el sistema anfitrión se redirija al 80 del contenedor, `-p`, y se enlaza la aplicación web con el contenedor en el que se ejecuta, a través del indicador `-link`. Además se monta el volumen `volume-public` con el indicador `-v`. Por último, se crea localmente el fichero de configuración `/etc/nginx/conf.d/nginx.conf` en el que se especifica que este servidor escuchará en el puerto 80, hará uso del directorio `/usr/src/app/public`, se agregan los campos de redirección y nombre del servidor al encabezado de solicitud, se deshabilita la redirección a otra ruta, se indican los ficheros para el procesamiento de solicitudes, así como que ha de resolver la aplicación en el puerto 9292, usado por `puma`.

Listado 3.8: Fichero `nginx.conf`

```
server {
    listen 80;
    root /usr/src/app/public;
    location / {
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        proxy_set_header Host $http_host;
        proxy_redirect off;
        try_files $uri /page_cache/$uri /page_cache/$uri.html @app;
    }
    location @app{
        proxy_pass http://app:9292;
        break;
    }
}
```

El *script* `./docker-microservices.sh`, con permiso `chmod +x` es:

Listado 3.9: Contenido de `docker-microservices.sh`

```
#!/bin/bash

if [ "$POSTGRES_USER" = "" ] || [ "$POSTGRES_PASSWORD" = "" ]; then
    echo "Environment variables for POSTGRES not found"
    exit
```

```

fi
docker build -t sample_app_rails_4_image .
docker volume create --name volume-public
docker run --name some-postgres -e POSTGRES_USER=$POSTGRES_USER \
-e POSTGRES_PASSWORD=$POSTGRES_PASSWORD \
-v volume-public:/var/lib/postgresql -d postgres
docker run -i --name app-job --entrypoint ./setup.sh \
-e POSTGRES_USER=$POSTGRES_USER -e POSTGRES_PASSWORD=$POSTGRES_PASSWORD \
-w /usr/src/app -v volume-public:/var/lib/postgresql \
--link some-postgres:db sample_app_rails_4_image
docker run -d -it --name app-task -e POSTGRES_USER=$POSTGRES_USER \
-e POSTGRES_PASSWORD=$POSTGRES_PASSWORD -w /usr/src/app \
-v volume-public:/usr/src/app/public --link some-postgres:db \
sample_app_rails_4_image \
/bin/bash -c "cp config/database.yml.postgresql config/database.yml && \
cp ./secret.example ./secret && puma -p 9292"
docker run --name some-nginx \
-v "${PWD}/nginx.conf":/etc/nginx/conf.d/default.conf \
-p 8080:80 --link app-task:app -v volume-public:/usr/src/app/public -d nginx

```

3.2.6 Resultado

Como resultado el *script* anterior crea los elementos propuestos en el planteamiento.

La imagen Docker creada y las descargadas son:

REPOSITORY	TAG	IMAGE ID	CREATED
sample_app_rails_4_image	latest	ca61d72a1605	9 minutes ago
postgres	latest	4023a747a01a	18 hours ago
nginx	latest	a39777a1a4a6	3 days ago
ruby	2.0-onbuild	7f6b98152faf	10 months ago

Figura 3.4: Imágenes docker mostradas con `docker images`.

Los contenedores Docker creados son:

CONTAINER ID	IMAGE	COMMAND	CREATED
82473e3988c1	nginx	"nginx -g 'daemon ...'"	3 minute ago
s ago	Up 3 minutes	443/tcp, 0.0.0.0:8080->80/tcp	some-nginx
6cd600b4c3c6	sample_app_rails_4_image	"puma -p 9292"	5 minute ago
s ago	Up 5 minutes		app-task
e5e495863dc7	sample_app_rails_4_image	"/setup.sh"	5 minute ago
s ago	Exited (0) 5 minutes ago		app-job
51b06d1c5ffb	postgres	"/docker-entrypoint..."	5 minute ago
s ago	Up 5 minutes	5432/tcp	some-postgres

Figura 3.5: Contenedores docker mostrados con `docker ps -a`.

Así, cuando se hace una petición, mediante el comando `curl`, el contenedor **some-nginx** la recibe y redirige al contenedor **app-task** donde se encuentra el servidor web:

```
$ curl http://localhost:8080
```

```
root@carolina-SM:/home/carolina/proyecto/sample_app_rails_4-1
root@carolina-SM:/home/carolina/proyecto/sample_app_rails_4-1# curl http://localhost:8080
<!DOCTYPE html>
<html>
  <head>
    <title>Ruby on Rails Tutorial Sample App</title>
    <link data-turbolinks-track="true" href="/assets/application.css?body=1" media="all" rel="stylesheet" />
    <link data-turbolinks-track="true" href="/assets/custom.css?body=1" media="all" rel="stylesheet" />
    <link data-turbolinks-track="true" href="/assets/sessions.css?body=1" media="all" rel="stylesheet" />
    <link data-turbolinks-track="true" href="/assets/static_pages.css?body=1" media="all" rel="stylesheet" />
    <link data-turbolinks-track="true" href="/assets/users.css?body=1" media="all" rel="stylesheet" />
    <script data-turbolinks-track="true" src="/assets/jquery.js?body=1"></script>
```

Figura 3.6: Resultado de la petición `curl http://localhost:8080`.

También puede visualizarse desde el navegador web:

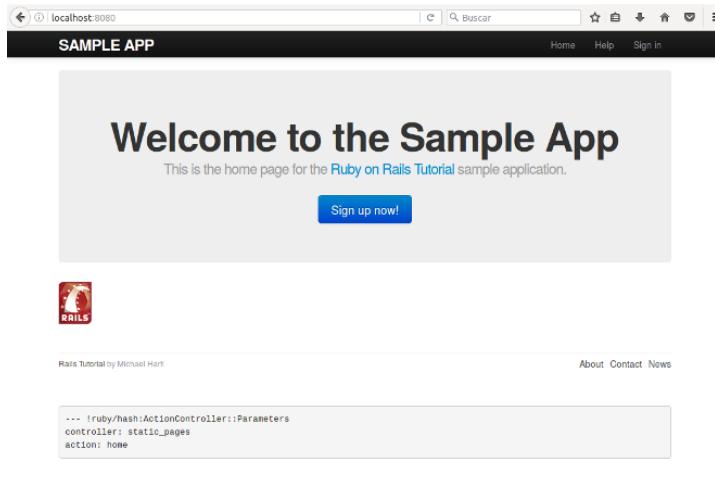


Figura 3.7: Resultado de la Iteración 1 con el uso de Docker.

3.3 Iteración 2: Subida de la imagen resultante de la aplicación al repositorio de imágenes Docker Hub

A efectos de mantener la imagen remotamente se sube al repositorio Docker Hub, previa creación de una cuenta en él. Para ello se inicia sesión por y se añade una etiqueta a la imagen `sample_app_rails_4_image`, especificando su identificador y el repositorio escogido:

```
$ docker login
$ docker tag <image-id> carolina/sample_app_rails_4_image:initial
$ sudo docker push carolina/sample_app_rails_4_image
```

Puede ser comprobado localmente mediante:

```
$ docker images carolina/sample_app_rails_4_image:initial
```

```
root@carolina-SM: /home/carolina/proyecto/sample_app_rails_4-1
root@carolina-SM:/home/carolina/proyecto/sample_app_rails_4-1# docker images carolina/sample_app_rails_
4_image:initial
REPOSITORY          TAG      IMAGE ID      CREATED     SIZE
carolina/sample_app_rails_4_image  initial   ca61d72a1605  About an hour ago  867 MB
root@carolina-SM:/home/carolina/proyecto/sample_app_rails_4-1#
```

Figura 3.8: Versión *initial* de la imagen `sample_a_rails_4` en local.

La imagen etiquetada como `initial` se aprecia ahora en Docker Hub:

PUBLIC REPOSITORY
carolina/sample_app_rails_4_image ☆
Last pushed: a few seconds ago

Tag Name	Compressed Size	Last Updated
initial	338 MB	a few seconds ago

Figura 3.9: Versión *initial* de la imagen `sample_a_rails_4` en Docker Hub.

3.4 Iteración 3: Integración y Despliegue continuos de la aplicación con Travis CI, Docker Hub y GitHub

En esta iteración se busca adaptar la aplicación Ruby on Rails para que funcione con Travis CI, definiendo la estructura presentada en la Figura 3.10.

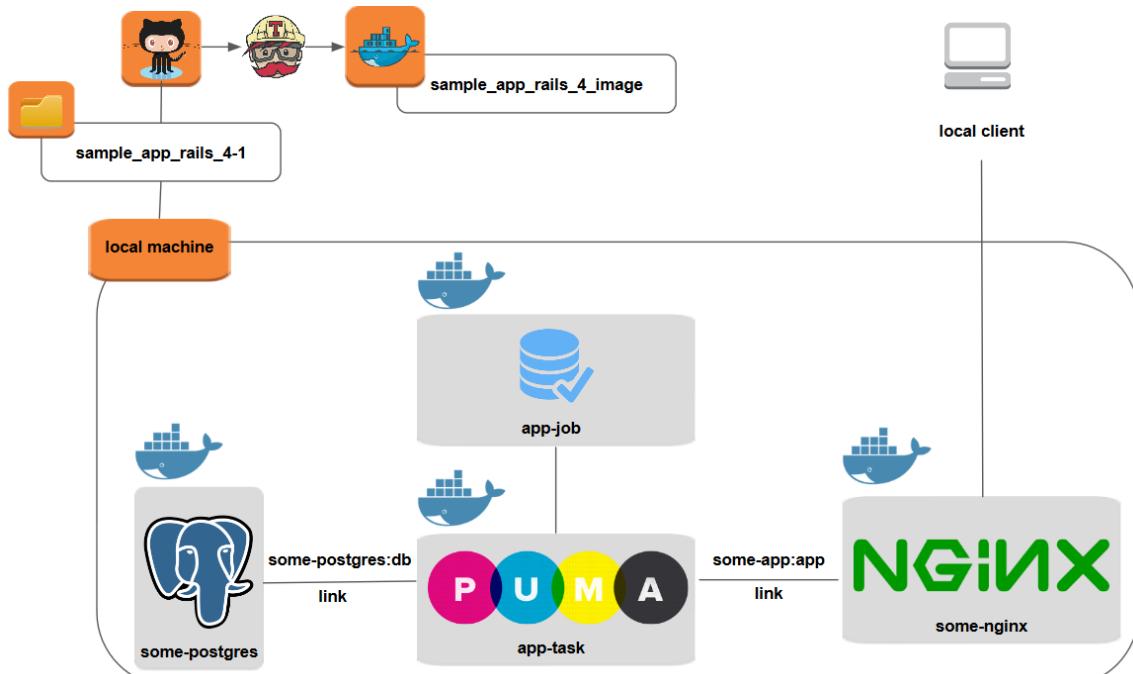


Figura 3.10: Integración y Despliegue continuos con Travis CI.

Travis CI se conectará al repositorio de GitHub `sample_app_rails_4-1` con la intención de poder generar una imagen Docker de la aplicación tras cada cambio. Esto permite trabajar en un entorno de desarrollo en el que la integración de los cambios en el proyecto es continua, así como también su despliegue es automatizado.

Las precondiciones, condiciones y postcondiciones necesarias para construir y desplegar la aplicación en uso de manera automática se especifican en el fichero `.travis.yml`. A este fichero se le añadirán, además, las credenciales de la cuenta de Docker Hub y GitHub mediante variables de entorno encriptadas. Las acciones a realizar serán la comprobación de que los tests pasan satisfactoriamente, la construcción de una nueva imagen Docker y su subida al repositorio Docker Hub.

Para llevarlo a cabo es necesario añadir una configuración de base de datos específica. En este caso se llamará `travis_ci_test`.

Finalmente, el resultado permitirá que cuando se haga un nuevo cambio o *commit* acompañado de una subida del mismo al repositorio GitHub comenzará, también, la construcción en Travis CI. Tanto en caso de fallo como de éxito se configura el envío de un correo electrónico que lo anuncie. Como se puede ver en la figura 3.17, al repositorio en Docker Hub se subirán 3 copias comprimidas de la imagen. La primera etiquetada con el hash del *commit*, la segunda con el número de construcción en Travis CI y la tercera como la última versión, *latest*.

3.4.1 Vinculación con el repositorio remoto

En primer lugar hay que ingresar en el sitio Travis CI con la cuenta de GitHub y añadir el repositorio con el que se trabaja actualmente:

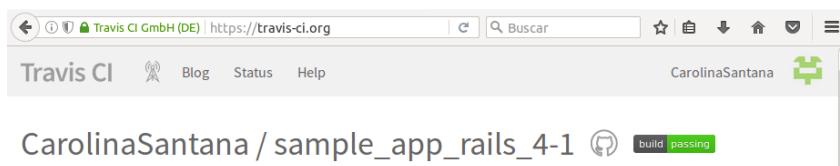


Figura 3.11: Vinculación de GitHub con Travis CI.

3.4.2 Configuración de condiciones

En segundo lugar se procede a instalar la gema `travis`. Para ello se edita el fichero `Gemfile`, incluyéndola en el grupo de desarrollo y test.

Luego se instala mediante:

```
$ bundle install --without production
```

Para probar el correcto funcionamiento es necesario crear un nuevo fichero de configuración para la base de datos en Travis CI:

Listado 3.10: Fichero `config/database.yml.travis`

```
test:
  adapter: postgresql
  database: travis_ci_test
  username: postgres
```

Para configurar Travis CI se crea el fichero `.travis.yml` y se establecen las variables de entorno de las credenciales de Docker Hub:

```
$ touch .travis.yml
$ travis env set DOCKER_USERNAME carolina
$ travis env set DOCKER_PASSWORD *****
```

También se agrega una variable de entorno que contenga los 8 primeros caracteres del *hash* del *git commit*, justo debajo de las anteriores.

En la sección `after_success` del fichero se inicia sesión en Docker Hub y luego se construye la imagen. Al repositorio en Docker Hub se subirán 3 copias comprimidas de la imagen. La primera etiquetada con el hash del *commit* correspondiente, la segunda con el número de construcción en Travis CI y la tercera como la última versión, *latest*. Además se le indica la base de datos de prueba para la ejecución de los tests y la versión de Ruby en uso.

Listado 3.11: Fichero `.travis.yml`

```
env:
  global:
    - COMMIT=${TRAVIS_COMMIT::8}
language: ruby
rvm:
  - 2.0.0-p648
bundler_args: --without production
addons:
  postgresql: '9.3'
services:
  - docker
before_script:
  - cp config/database.yml.travis config/database.yml
  - psql -c 'create database travis_ci_test;' -U postgres
  - RAILS_ENV=test bundle exec rake db:migrate --trace
script:
  - bundle exec rspec
notifications:
  email:
    recipients:
      - c.santanamartel@gmail.com
    on_success: always
    on_failure: always
sudo: required
after_success:
  - docker login -u $DOCKER_USERNAME -p $DOCKER_PASSWORD
  - export REPO=$DOCKER_USERNAME/sample_app_rails_4_image
  - docker build -f Dockerfile -t $REPO:$COMMIT .
  - docker tag $REPO:$COMMIT $REPO:latest
  - docker tag $REPO:$COMMIT $REPO:travis-$TRAVIS_BUILD_NUMBER
  - docker push $REPO
```

3.4.3 Resultado

Finalmente, cuando se haga un nuevo *commit* y subida al repositorio GitHub comenzará también la construcción en Travis CI. Tanto si se produce un fallo como si termina con éxito se ha configurado el envío de un correo electrónico para informarlo.

A continuación, se prueba el caso de fallo comentando la línea que crea la base de datos de prueba `travis_ci_test` y el de éxito con ella. Los correos electrónicos recibidos son:

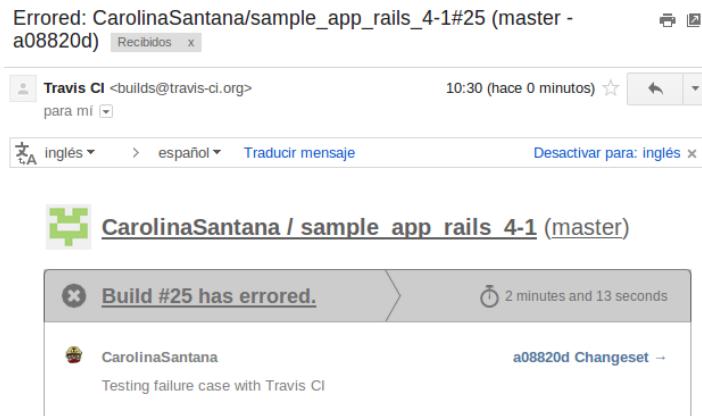


Figura 3.12: Correo electrónico de Travis CI en caso de fallo.

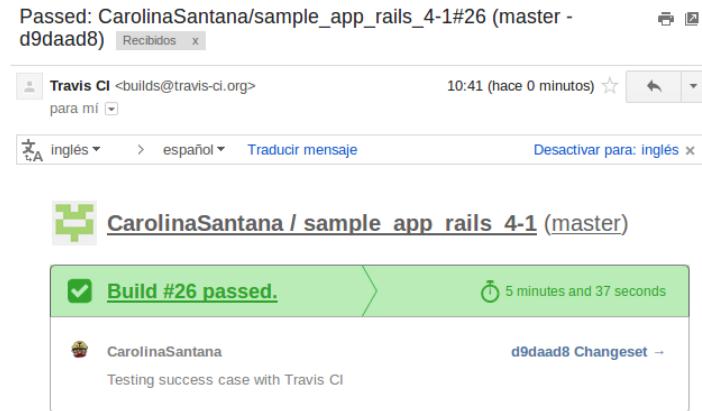


Figura 3.13: Correo electrónico de Travis CI en caso de éxito.

Dirigiendo a Travis CI donde se comenta el fallo o el éxito:

Figura 3.14: Caso de fallo en Travis CI.

```
526 The command "RAILS_ENV=test bundle exec rake db:migrate --trace" failed and exited with 1 during .
527
528 Your build has been stopped.
```

Figura 3.15: Motivo de fallo en Travis CI.

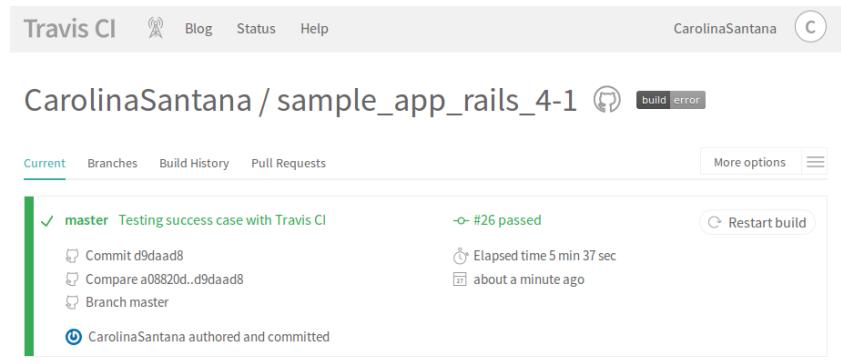
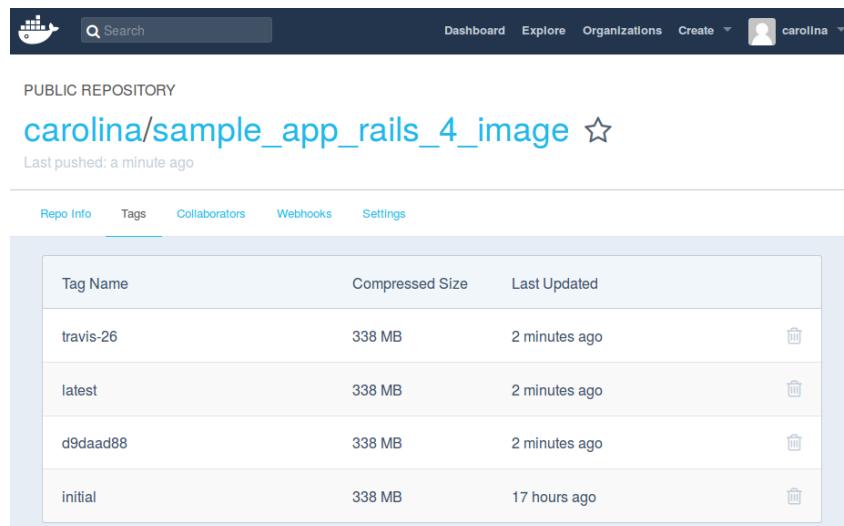


Figura 3.16: Caso de éxito en Travis CI.

Esta construcción terminará con la creación y subida de las tres imágenes, comentadas en el planteamiento de esta iteración, en el repositorio Docker Hub:

Figura 3.17: Repositorio de `sample_app_rails_4_image`.

3.5 Iteración 4: Despliegue monomáquina con unidades de servicio usando Vagrant, VirtualBox y CoreOS

En esta nueva iteración se aplica el uso del sistema operativo orientado a contenedores CoreOS. De esta manera se crearán unidades de servicio `systemd` que permitirán la correcta aplicación y funcionamiento de los contenedores Docker en los que se divide la aplicación. La infraestructura del nuevo planteamiento se corresponde con la Figura 3.18.

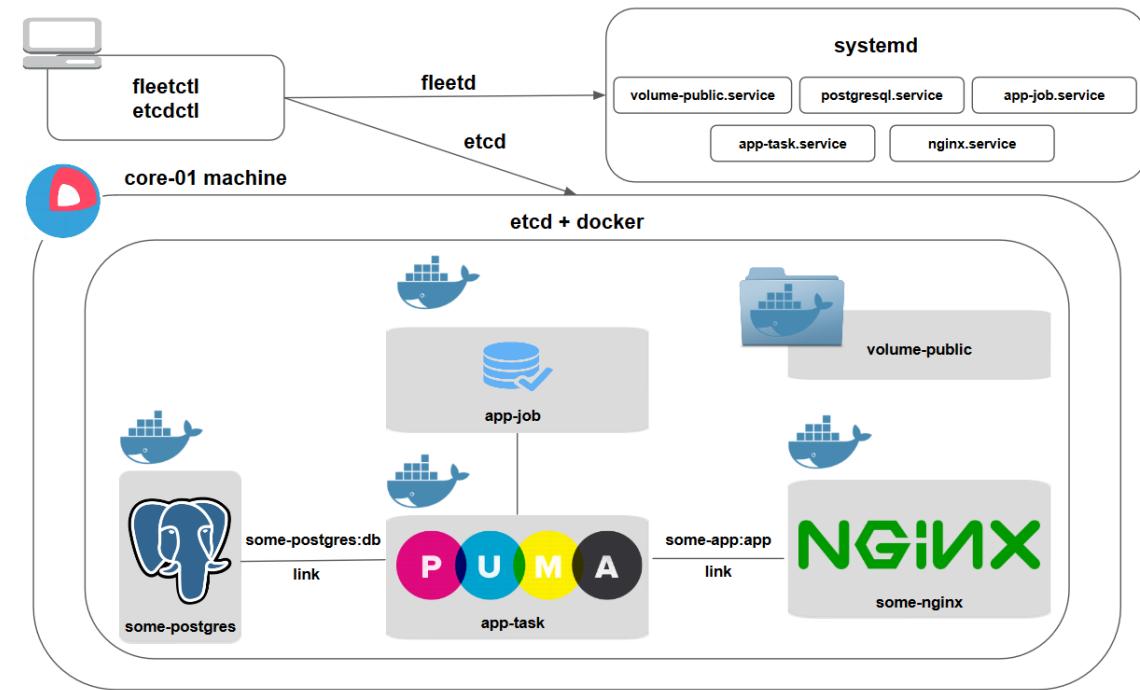


Figura 3.18: Infraestructura de la aplicación con unidades de servicio CoreOS.

En local existen dos herramientas de línea de comandos llamadas `fleetctl` y `etcdctl` utilizadas internamente para comunicarse con los elementos del sistema CoreOS `fleetd` y `etcd`. Fleet se encarga de dotar a la máquina CoreOS con los servicios que se desean implementar. Etcd2 permite almacenar los datos de las máquinas CoreOS.

Para llevar a cabo esta iteración se utiliza el repositorio GitHub [coreos/coreos-vagrant](#) que proporciona una plantilla `Vagrantfile` para configurar el entorno de una máquina virtual CoreOS, usando el hipervisor software VirtualBox.

En este procedimiento se utilizan tres archivos relevantes que se describirán a lo largo del proceso. De esta forma se utilizará la misma configuración para el fichero `config.rb`, se creará un nuevo `cloud-config` que será llamado `user-data.sampleapp.vbox` y se modificará el fichero `Vagrantfile`.

3.5.1 Preparación del repositorio local y remoto

En primer lugar se realiza un *fork* del repositorio [coreos/coreos-vagrant](#). Automáticamente se crea la copia en la cuenta personal y se clona en local:

```
$ git clone https://github.com/CarolinaSantana/coreos-vagrant.git
```

Luego, se accede a la carpeta local que lo contiene y se prepara para comprobar su funcionamiento con los datos de usuario y la configuración de ejemplo. Esto se prueba ejecutando el `Vagrantfile` y conectándose a la máquina virtual creada. En este caso se utiliza VirtualBox, lo que se conseguiría con `vagrant up --provider=virtualbox` o `vagrant up`, puesto que es el proveedor por defecto.

```
$ cd coreos-vagrant
$ cp user-data.sample user-data
$ cp config.rb.sample config.rb
$ vagrant up && vagrant ssh core-01
```

Comprobando que funciona correctamente:

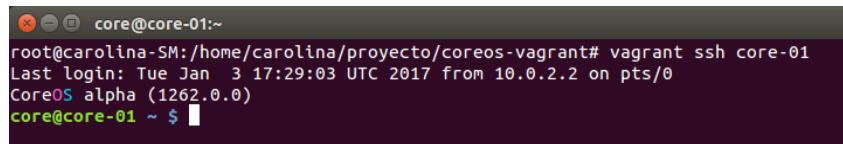


Figura 3.19: Conexión ssh a la máquina virtual **core-01**.

Además puede visualizarse la máquina mediante VirtualBox:



Figura 3.20: Máquina virtual **core-01** desde VirtualBox.

3.5.2 Preparación de la cabecera del fichero user-data

El fichero **user-data** es el archivo de configuración **cloud-config**. Especifica el *discovery token* que forma parte de la *URL* de descubrimiento que conecta las instancias. También indica las variables de entorno y la lista de unidades de servicio que se deben iniciar de forma predeterminada. Los parámetros **coreos.etcd**, **coreos.fleet** y **coreos.flannel** son traducidos a una unidad **systemd** parcial actuando como un archivo de configuración **etcd2**, **fleet** y **flannel**, respectivamente.

Como el entorno de plataforma admite la función de plantilla de **coreos-cloudinit**, es posible automatizar la configuración de **etcd2** con los campos **\$private_ip4** y **\$public_ip4**. Al generar un *discovery token* se establece el tamaño del clúster, ya que **etcd2** lo utiliza para determinar si todos los miembros se han unido a él. Después de inicializarse, el clúster puede crecer o reducirse. Por su parte **fleet** y **flannel** usan variables de entorno en su configuración.

La siguiente sección de este fichero pertenece a las unidades de servicio, de las que ya se encuentran definidas **etcd2.service**, **fleet.service**, **flanneld.service**, donde esta última especifica la red en CoreOS, siendo utilizada la 10.1.0.0/16 como pública para los contenedores, y **docker-tcp.socket**, el cual permite que se pueda usar el servicio **docker-service** dentro de cada máquina.

CoreOS posee la opción de actualizar automáticamente el sistema operativo cuando una actualización es detectada o la opción de que no sean reiniciadas con la intención de que lo haga el usuario. Como se está en un entorno de desarrollo se escoge la segunda opción.

Listado 3.12: Fichero **user-data**

```
#cloud-config
---
coreos:
  update:
    reboot-strategy: "off"
  etcd2:
    advertise-client-urls: http://$public_ipv4:2379
    initial-advertise-peer-urls: http://$private_ipv4:2380
    listen-client-urls: http://0.0.0.0:2379,http://0.0.0.0:4001
    listen-peer-urls: http://$private_ipv4:2380,http://$private_ipv4:7001
    discovery: https://discovery.etcd.io/acd611cb07df434a400bbd7e36d793a0
  fleet:
    public-ip: "$public_ipv4"
  flannel:
    interface: "$public_ipv4"
  units:
    - name: etcd2.service
      command: start
    - name: fleet.service
      command: start
    - name: flanneld.service
      drop-ins:
        - name: 50-network-config.conf
          content: |
            [Service]
            ExecStartPre=/usr/bin/etcctl set /coreos.com/network/config \
            '{ "Network": "10.1.0.0/16" }'
            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
```

3.5.3 Interpretación del fichero config.rb

El segundo fichero importante para este despliegue es **config.rb**. Especifica el número de nodos CoreOS y proporciona una opción para generar automáticamente el *discover token*, valor que se reemplaza automáticamente cuando se inicia el despliegue o levantamiento de una máquina. Además permite la necesaria corrección del valor del parámetro perteneciente a la estrategia de reinicio de CoreOS, al tener que ser la entrada en el fichero **user-data** de una manera distinta a como es traducido por YAML.

Listado 3.13: Fichero **config.rb**

```
$num_instances=1
$new_discovery_url="https://discovery.etcd.io/new?size=#{$num_instances}"
if File.exists?('user-data') && ARGV[0].eql?('up')
  require 'open-uri'
  require 'yaml'
  token = open($new_discovery_url).read
  data = YAML.load(IO.readlines('user-data')[1..-1].join)
  if data.key? 'coreos' and data['coreos'].key? 'etcd'
    data['coreos']['etcd']['discovery'] = token
  end
```

```

if data.key? 'coreos' and data['coreos'].key? 'etcd2'
  data['coreos']['etcd2']['discovery'] = token
end
# Fix for YAML.load() converting reboot-strategy from 'off' to `false`
if data.key? 'coreos' and data['coreos'].key? 'update'
  and data['coreos']['update'].key? 'reboot-strategy'
    if data['coreos']['update']['reboot-strategy'] == false
      data['coreos']['update']['reboot-strategy'] = 'off'
    end
end
yaml = YAML.dump(data)
File.open('user-data', 'w') { |file| file.write("#cloud-config\n\n#{yaml}") }
end

```

3.5.4 Preparación del fichero Vagrantfile

El fichero viene preparado para el uso de los proveedores VirtualBox y VMware. Al no utilizar el segundo se prescinde de sus configuraciones y se reagrupan las instrucciones pertinentes a VirtualBox.

El fichero **Vagrantfile** establece el número de máquinas CoreOS que ha de ser creado por Vagrant. En este caso una sola instancia. Otras especificaciones son las opciones de configuración por defecto a utilizar para crear las máquinas y la imagen a partir de la que hacerlo. El fichero viene preparado para aplicar la redirección de puertos de la máquina virtual al sistema anfitrión. Las direcciones IP que tomarán las máquinas en la red privada empezarán en 172.17.8.10X. Además, se facilita la opción para habilitar el almacenamiento compartido entre anfitrión y máquina. En último lugar se realiza la copia de la configuración del **user-data** en **/var/lib/coreos-vagrant/vagrantfile-user-data**, dentro de la máquina virtual. Así, **coreos-cloudinit** lee **vagrantfile-user-data** en cada inicio y lo utiliza para crear y reproducir lo que indica.

Listado 3.14: Fichero **Vagrantfile**

```

# -*- mode: ruby -*-
# # vi: set ft=ruby :
require 'fileutils'
Vagrant.require_version ">= 1.6.0"
if (!ARGV.nil? && ARGV.join('').include?('provider=virtualbox')) ||
  (!ARGV.nil? && !ARGV.join('').include?('provider'))
  FileUtils.cp_r(File.join(File.dirname(__FILE__), "user-data.sampleapp.vbox"),
    File.join(File.dirname(__FILE__), "user-data"), :remove_destination => true)
end
CLOUD_CONFIG_PATH = File.join(File.dirname(__FILE__), "user-data")
CONFIG = File.join(File.dirname(__FILE__), "config.rb")
$num_instances = 1
$instance_name_prefix = "core"
$update_channel = "alpha"
$image_version = "current"
$vm_gui = false
$vm_memory = 1024
$vm_cpus = 1
$vb_cpuexecutioncap = 100
$shared_folders = {}
$forwarded_ports = {}
if ENV["NUM_INSTANCES"].to_i > 0 && ENV["NUM_INSTANCES"]
  $num_instances = ENV["NUM_INSTANCES"].to_i
end
if File.exist?(CONFIG)
  require CONFIG
end
def vm_gui
  $vb_gui.nil? ? $vm_gui : $vb_gui
end

```

```

end
def vm_memory
  $vb_memory.nil? ? $vm_memory : $vb_memory
end
def vm_cpus
  $vb_cpus.nil? ? $vm_cpus : $vb_cpus
end
Vagrant.configure("2") do |config|
  config.ssh.insert_key = false
  config.ssh.forward_agent = true
  config.vm.box = "coreos-%s" % $update_channel
  if $image_version != "current"
    config.vm.box_version = $image_version
  end
  config.vm.provider :virtualbox do |vb, override|
    override.vm.box_url = "https://storage.googleapis.com/%s.release.core-os.net/
      amd64-usr/%s/coreos_production_vagrant.json" %
      [$update_channel,$image_version]
    vb.check_guest_additions = false
    vb.functional_vboxsf = false
  end
  if Vagrant.has_plugin?("vagrant-vbguest") then
    config.vbguest.auto_update = false
  end
  (1..$num_instances).each do |i|
    config.vm.define vm_name = "%s-%02d" % [$instance_name_prefix, i] do |config|
      config.vm.hostname = vm_name
      config.vm.provider :virtualbox do |vb, override|
        $forwarded_ports.each do |guest, host|
          override.vm.network "forwarded_port", guest: guest, host: host,
            auto_correct: true
        end
        vb.gui = vm_gui
        vb.memory = vm_memory
        vb.cpus = vm_cpus
        vb.customize ["modifyvm", :id, "--cpus", "--cpusexecutioncap",
          "#{$vb_cpus}"]
        ip = "172.17.8.{i+100}"
        override.vm.network :private_network, ip: ip
      end
      config.vm.provider :virtualbox do |vb, override|
        # Uncomment below to enable NFS for sharing the host machine into the VM.
        #override.vm.synced_folder ".", "/home/core/share", id: "core",
        #:nfs => true, :mount_options => ['nolock,vers=3,udp']
        $shared_folders.each_with_index do |(host_folder, guest_folder), index|
          override.vm.synced_folder host_folder.to_s, guest_folder.to_s,
            id: "core-share%02d" % index, nfs: true,
            mount_options: ['nolock,vers=3,udp']
        end
      end
      if File.exist?(CLOUD_CONFIG_PATH) && ARGV[0].eql?('up')
        config.vm.provider :virtualbox do |vb, override|
          override.vm.provision :file, :source => "#{CLOUD_CONFIG_PATH}",
            :destination => "/tmp/vagrantfile-user-data"
          override.vm.provision :shell, :inline => "mv /tmp/vagrantfile-user-data
            /var/lib/coreos-vagrant/", :privileged => true
        end
      end
    end
  end
end

```

3.5.5 Despliegue manual de las unidades systemd

El primer paso consiste en permitir que exista almacenamiento NFS compartido entre la máquina local y la máquina virtual CoreOS, para pasar a esta última las unidades de servicio a implementar. Para ello es necesario descomentar la siguiente línea, dentro del fichero **Vagrantfile**:

Listado 3.15: Fichero **Vagrantfile**

```
.override.vm.synced_folder ".", "/home/core/share", id: "core", :nfs => true, \
:mount_options => ['nolock,vers=3,udp']
```

También será necesario instalar el adaptador NFS en el sistema anfitrión y cargar los cambios. Esto creará el directorio **share**, compartido con el sistema anfitrión:

```
$ sudo apt-get install nfs-kernel-server -y
$ vagrant reload --provision
```

Unidad volume-public.service

La primera unidad de servicio será el volumen Docker compartido entre los contenedores. Se crea la ruta **etc/sysctl/volume-public.service**. En este fichero se configura el servicio **volume-public.service**. Su contenido especifica que comenzará después de que lo haga el servicio Docker, requerido para su funcionamiento. El servicio empezará directamente creando el volumen, añadiendo un enlace simbólico al mismo para todos los usuarios de la máquina.

Listado 3.16: Fichero **etc/sysctl/volume-public.service**.

```
[Unit]
Description= volume-public share between some-postgres, app-job, app-task and
            some-nginx
After=docker.service
Requires=docker.service
[Service]
TimeoutStartSec=0
ExecStart=/usr/bin/docker volume create --name volume-public
[Install]
WantedBy=multi-user.target
```

Unidad postgresql.service

La siguiente es la unidad de servicio correspondiente a la base de datos de la aplicación. Para ello se crea la ruta **etc/sysctl/postgresql.service**. En este fichero se configura el servicio **postgresql.service**. Se especifica que comenzará después del servicio Docker y **volume-public.service**, requeridos para el correcto funcionamiento. Empezará directamente sin tiempo de espera, se reiniciará siempre si ocurre algún problema y se le indica el fichero del que leer las variables de entorno. Si este servicio ya se ha ejecutado se procederá terminándolo y eliminando el contenedor. Seguidamente, se comprobará, si ya existe la imagen del contenedor en la máquina, que sea la última versión disponible, de no ser así se obtendrá de nuevo. Llegado este punto la unidad de servicio comenzará creando el contenedor, especificando la opción a realizar si quiere pararse. Además se añade un enlace simbólico al mismo para todos los usuarios de la máquina.

Listado 3.17: Fichero **etc/sysctl/postgresql.service**.

```
[Unit]
Description=PostgreSQL database
```

```

After=docker.service volume-public.service
Requires=docker.service volume-public.service
[Service]
TimeoutStartSec=0
Restart=always
EnvironmentFile=/home/core/share/etc/sysctl/postgres-credentials.env
ExecStartPre=/usr/bin/docker kill some-postgres
ExecStartPre=/usr/bin/docker rm some-postgres
ExecStartPre=/usr/bin/docker pull postgres
ExecStart=/usr/bin/docker run --rm --name some-postgres \
-e "POSTGRES_USER=${POSTGRES_USER}" \
-e "POSTGRES_PASSWORD=${POSTGRES_PASSWORD}" \
-v "volume-public:/var/lib/postgresql" -p "5432:5432" postgres
ExecStop=/usr/bin/docker stop some-postgres
[Install]
WantedBy=multi-user.target

```

El contenido del fichero que especifica las variables de entorno hace referencia a las credenciales de la base de datos.

Listado 3.18: Contenido del fichero /etc/postgres-credentials.env

```

POSTGRES_USER=postgres
POSTGRES_PASSWORD=postgres

```

Unidad app-job.service

La siguiente unidad de servicio será el contenedor ejecutable de la aplicación que crea, migra y alimenta la base de datos. Para ello se crea la ruta `etc/sysctl/app-job.service`. En este fichero se configura el servicio `app-job.service`. Se especifica que comenzará después de que lo haga el servicio Docker, `volume-public.service` y `postgresql.service`, requeridos para el correcto funcionamiento. Empezará directamente sin tiempo de espera, indicándole el fichero del que leer las variables de entorno necesarias. Si este servicio ya se ha ejecutado se procederá terminándolo y eliminando el contenedor. Seguidamente, se comprobará, si ya existe la imagen del contenedor en la máquina, que sea la última versión disponible, de no ser así se obtendrá de nuevo. La unidad de servicio comenzará creando el contenedor, especificando la opción a realizar si quiere pararse. Además se añade un enlace simbólico al mismo para todos los usuarios de la máquina.

Listado 3.19: Fichero etc/sysctl/app-job.service.

```

[Unit]
Description=executable app-job container that creates, migrates, seeds and
populates the database
After=docker.service volume-public.service postgresql.service
Requires=docker.service volume-public.service postgresql.service
[Service]
TimeoutStartSec=0
EnvironmentFile=/home/core/share/etc/sysctl/postgres-credentials.env
ExecStartPre=/usr/bin/docker kill app-job
ExecStartPre=/usr/bin/docker rm app-job
ExecStartPre=/usr/bin/docker pull carolina/sample_app_rails_4_image:latest
ExecStart=/usr/bin/docker run --rm --name app-job \
-v "volume-public:/usr/src/app/public" --entrypoint "./setup.sh" \
-e "POSTGRES_USER=${POSTGRES_USER}" \
-e "POSTGRES_PASSWORD=${POSTGRES_PASSWORD}" \
-w "/usr/src/app" --link "some-postgres:db" \
carolina/sample_app_rails_4_image:latest
[Install]
WantedBy=multi-user.target

```

Unidad app-task.service

La siguiente unidad de servicio será el contenedor de la aplicación que ejecuta el servidor puma. Se crea la ruta `etc/sysctl/app-task.service`. En este fichero se configura el servicio `app-task.service`. El contenido del mismo especifica que comenzará después de que lo haga el servicio Docker, `volume-public.service`, `postgresql.service` y `app-job.service`, requeridos para el correcto funcionamiento. El servicio empezará sin tiempo de espera, se reiniciará si falla y se le indica el fichero del que leer las variables de entorno necesarias. Si este servicio ya se ha ejecutado se procederá terminándolo y eliminando el contenedor. Seguidamente, se comprobará, si ya existe la imagen del contenedor en la máquina, que sea la última versión disponible, de no ser así se obtendrá de nuevo. Llegado este punto la unidad de servicio comenzará creando el contenedor, especificando la opción a realizar si quiere pararse. Además se añade un enlace simbólico al mismo para todos los usuarios de la máquina.

Listado 3.20: Fichero `etc/sysctl/app-task.service`.

```
[Unit]
Description=app-task container that runs the server puma
After=docker.service volume-public.service postgresql.service app-job.service
Requires=docker.service volume-public.service postgresql.service app-job.service
[Service]
TimeoutStartSec=0
Restart=always
EnvironmentFile=/home/core/share/etc/sysctl/postgres-credentials.env
ExecStartPre=-/usr/bin/docker kill app-task
ExecStartPre=-/usr/bin/docker rm app-task
ExecStartPre=/usr/bin/docker pull carolina/sample_app_rails_4_image:latest
ExecStart=/usr/bin/docker run --rm --name app-task \
-e "POSTGRES_USER=${POSTGRES_USER}" \
-e "POSTGRES_PASSWORD=${POSTGRES_PASSWORD}" \
-w "/usr/src/app" -v "volume-public:/usr/src/app/public" \
--link "some-postgres:db" carolina/sample_app_rails_4_image:latest \
/bin/bash -c "cp config/database.yml.postgresql config/database.yml && \
cp ./secret.example ./secret && puma -p 9292"
ExecStop=/usr/bin/docker stop app-task
[Install]
WantedBy=multi-user.target
```

Unidad nginx.service

La siguiente unidad del servicio será el contenedor que ejecuta el servidor Nginx. Para ello se crea la ruta `etc/sysctl/nginx.service`. En este fichero se configura el servicio `nginx.service`. El contenido del mismo especifica que comenzará después de que lo haga el servicio Docker, `volume-public.service`, `postgresql.service`, `app-job.service` y `app-task.service`, requeridos para el correcto funcionamiento. El servicio empezará directamente sin tiempo de espera y se reiniciará si falla. Si ya se ha ejecutado se procederá terminándolo y eliminando el contenedor. Seguidamente, se comprobará, si ya existe la imagen del contenedor en la máquina, que sea la última versión disponible, de no ser así se obtendrá de nuevo. La unidad de servicio comenzará creando el contenedor, teniendo en cuenta que ahora la redirección de puertos se ha establecido para trabajar el puerto 80 de la máquina CoreOS con el puerto 80 de la aplicación, especificando la opción a realizar si dicho contenedor quiere pararse. Además se añade un enlace simbólico al mismo para todos los usuarios de la máquina.

Listado 3.21: Fichero `etc/sysctl/nginx.service`.

```
[Unit]
Description=some-nginx container that runs a reverse proxy server and a
web server
After=docker.service volume-public.service postgresql.service app-job.service
       app-task.service
Requires=docker.service volume-public.service postgresql.service
         app-job.service app-task.service
[Service]
TimeoutStartSec=0
Restart=always
ExecStartPre=-/usr/bin/docker kill some-nginx
ExecStartPre=-/usr/bin/docker rm some-nginx
ExecStartPre=/usr/bin/docker pull nginx
ExecStart=/usr/bin/docker run --rm --name some-nginx \
-v "/home/core/share/etc/sysctl/nginx.conf:/etc/nginx/conf.d/default.conf" \
-p "80:80" --link "app-task:app" -v "volume-public:/usr/src/app/public" nginx
ExecStop=/usr/bin/docker stop some-nginx
[Install]
WantedBy=multi-user.target
```

Despliegue manual

Una vez que ya se han configurado y escrito las 5 unidades de servicio necesarias el siguiente paso es preparar su despliegue. Para el reconocimiento de las unidades éstas tienen que ubicarse bajo `systemd`, por lo que se copian al directorio `/etc/systemd/system/`. El funcionamiento pasa por la habilitación del servicio, creando el enlace simbólico de la unidad para todos los usuarios, y el comienzo del mismo.

Se crea un *script* denominado `coreos-service-units-deploy.sh`, con permisos `chmod +x`, que se encargará de todo lo mencionado.

Listado 3.22: Fichero `coreos-service-units-deploy.sh`.

```
#!/bin/bash

sudo cp share/etc/sysctl/* /etc/systemd/system/
sudo systemctl enable volume-public.service
sudo systemctl enable postgresql.service
sudo systemctl enable app-job.service
sudo systemctl enable app-task.service
sudo systemctl enable nginx.service
sudo systemctl start volume-public.service
sudo systemctl start postgresql.service
sudo systemctl start app-job.service
sudo systemctl start app-task.service
sudo systemctl start nginx.service
```

Con la intención de que se ejecute al iniciar la máquina CoreOS se añade al fichero `Vagrantfile` una línea que indicará la ruta del mismo para provisionar la máquina con las directivas incluidas en él.

Listado 3.23: Fichero `Vagrantfile`

```
.
config.vm.provision "shell", path: "coreos-service-units-deploy.sh"
.
```

Por último se inicia la máquina y se accede a ella:

```
$ vagrant up && vagrant ssh core-01
```

Si se inicia el despliegue manual se comprueba el correcto funcionamiento de la aplicación, donde todos los servicios estarán en estado activo o cargado:

```
core@core-01 ~ $ sudo systemctl status volume-public.service
● volume-public.service - volume-public share between some-postgres, app-job, app-task and some-nginx
  Loaded: loaded (/etc/systemd/system/volume-public.service; disabled; vendor preset: disabled)
  Active: inactive (dead) since Sun 2017-02-19 11:57:10 UTC; 23min ago
    Main PID: 3694 (code=exited, status=0/SUCCESS)
```

Figura 3.21: Estado de la unidad **volume-public.service**.

```
core@core-01 ~ $ sudo systemctl status postgresql.service
● postgresql.service - PostgreSQL database
  Loaded: loaded (/etc/systemd/system/postgresql.service; enabled; vendor preset: disabled)
  Active: active (running) since Sun 2017-02-19 11:26:23 UTC; 54min ago
    Main PID: 1758 (docker)
      Tasks: 7
     Memory: 5.2M
        CPU: 36ms
       CGroup: /system.slice/postgresql.service
                 └─1758 /usr/bin/docker run --rm --name some-postgres -e POSTGRES_USER=postgres -e
```

Figura 3.22: Estado de la unidad **postgresql.service**.

```
core@core-01 ~ $ sudo systemctl status app-job.service
● app-job.service - executable app-job container that creates, migrates, seeds and populates
  Loaded: loaded (/etc/systemd/system/app-job.service; enabled; vendor preset: disabled)
  Active: inactive (dead) since Sun 2017-02-19 11:27:04 UTC; 1h 5min ago
    Main PID: 1939 (code=exited, status=0/SUCCESS)
```

Figura 3.23: Estado de la unidad **app-job.service**.

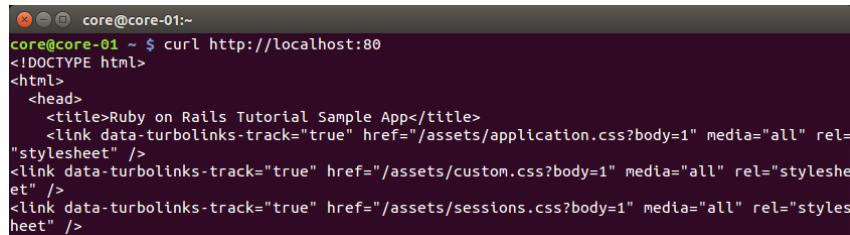
```
core@core-01 ~ $ sudo systemctl status app-task.service
● app-task.service - app-task container that runs the server puma
  Loaded: loaded (/etc/systemd/system/app-task.service; enabled; vendor preset: disabled)
  Active: active (running) since Sun 2017-02-19 11:26:32 UTC; 55min ago
    Main PID: 2134 (docker)
      Tasks: 11
     Memory: 7.2M
        CPU: 36ms
       CGroup: /system.slice/app-task.service
                 └─2134 /usr/bin/docker run --rm --name app-task -e POSTGRES_USER=postgres -e POSTG
```

Figura 3.24: Estado de la unidad **app-task.service**.

```
core@core-01 ~ $ sudo systemctl status nginx.service
● nginx.service - some-nginx container that runs a reverse proxy server and a web server
  Loaded: loaded (/etc/systemd/system/nginx.service; enabled; vendor preset: disabled)
  Active: active (running) since Sun 2017-02-19 11:26:35 UTC; 55min ago
    Main PID: 2277 (docker)
      Tasks: 11
     Memory: 7.2M
        CPU: 35ms
       CGroup: /system.slice/nginx.service
                 └─2277 /usr/bin/docker run --rm --name some-nginx -v /etc/systemd/system/nginx.con
```

Figura 3.25: Estado de la unidad **nginx.service**.

Finalmente, se accede a la aplicación desde la máquina CoreOS y desde el sistema anfitrión:



```
core@core-01:~$ curl http://localhost:80
<!DOCTYPE html>
<html>
  <head>
    <title>Ruby on Rails Tutorial Sample App</title>
    <link data-turbolinks-track="true" href="/assets/application.css?body=1" media="all" rel="stylesheet" />
    <link data-turbolinks-track="true" href="/assets/custom.css?body=1" media="all" rel="stylesheet" />
    <link data-turbolinks-track="true" href="/assets/sessions.css?body=1" media="all" rel="stylesheet" />
```

Figura 3.26: Acceso a la aplicación desde la máquina CoreOS.

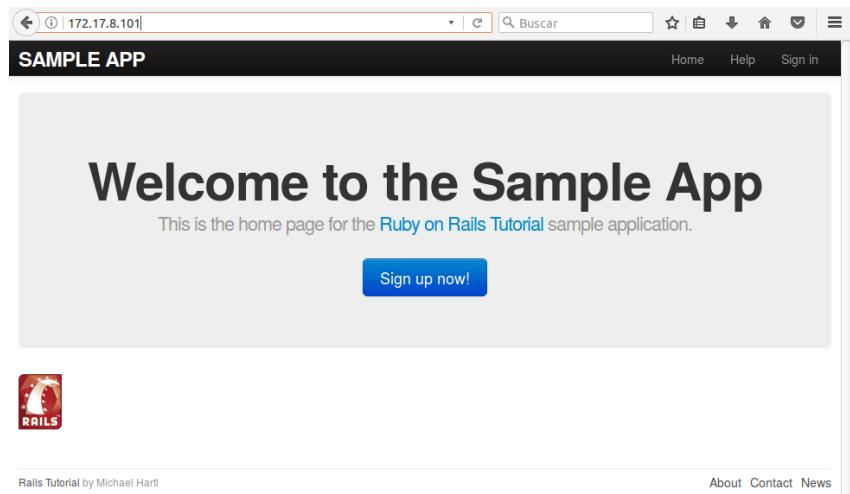


Figura 3.27: Acceso a la aplicación desde el sistema anfitrión.

3.5.6 Despliegue automático de las unidades systemd

Para realizar el despliegue de manera automática se hace uso del fichero `cloud-config` denominado `user-data.sampleapp.vbox`. Este fichero especifica el orden de las unidades a desplegar y las acciones de habilitación y comienzo de los servicios.

Esta vez no se usará el almacenamiento NFS compartido por lo que ya no es necesaria la línea de provisión del *script* en el fichero `Vagrantfile`. Además, puede volver a comentarse la línea que habilitaba este tipo de almacenamiento. De esta manera, también será necesario definir las variables de entorno, indicando la ruta, sus permisos y contenido. Lo mismo se realiza para el fichero que contiene la configuración Nginx.

Así, los añadidos al fichero `user-data.sampleapp.vbox` son:

Listado 3.24: Fichero `user-data.sampleapp.vbox`

```
#cloud-config
-----
write_files:
  - path: "/etc/postgres-credentials.env"
    permissions: "0644"
    content: |
      POSTGRES_USER=postgres
      POSTGRES_PASSWORD=postgres
  - path: "/etc/nginx.conf"
    permissions: "0644"
```

```

content: |
  server {
    listen 80;
    root /usr/src/app/public;
    location / {
      proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
      proxy_set_header Host $http_host;
      proxy_redirect off;
      try_files $uri /page_cache/$uri /page_cache/$uri.html @app;
    }
    location @app{
      proxy_pass http://app:9292;
      break;
    }
  }
coreos:
.
units:
.
- name: volume-public.service
  command: start
  enable: true
  content: |
    [Unit]
    Description= volume-public share between some-postgres, app-job, app-task
    and some-nginx
    After=docker.service
    Requires=docker.service
    [Service]
    TimeoutStartSec=0
    ExecStart=/usr/bin/docker volume create --name volume-public
    [Install]
    WantedBy=multi-user.target
- name: postgresql.service
  command: start
  enable: true
  content: |
    [Unit]
    Description=PostgreSQL database
    After=docker.service volume-public.service
    Requires=docker.service volume-public.service
    [Service]
    TimeoutStartSec=0
    Restart=always
    EnvironmentFile=/etc/postgres-credentials.env
    ExecStartPre=-/usr/bin/docker kill some-postgres
    ExecStartPre=-/usr/bin/docker rm some-postgres
    ExecStartPre=/usr/bin/docker pull postgres
    ExecStart=/usr/bin/docker run --rm --name some-postgres \
    -e "POSTGRES_USER=${POSTGRES_USER}" \
    -e "POSTGRES_PASSWORD=${POSTGRES_PASSWORD}" \
    -v "volume-public:/var/lib/postgresql" -p "5432:5432" postgres
    ExecStop=/usr/bin/docker stop some-postgres
    [Install]
    WantedBy=multi-user.target
- name: app-job.service
  command: start
  enable: true
  content: |
    [Unit]
    Description=executable app-job container that creates, migrates, seeds
    and populates the database
    After=docker.service volume-public.service postgresql.service
    Requires=docker.service volume-public.service postgresql.service
    [Service]
    TimeoutStartSec=0
    EnvironmentFile=/etc/postgres-credentials.env
    ExecStartPre=-/usr/bin/docker kill app-job
    ExecStartPre=-/usr/bin/docker rm app-job
    ExecStartPre=/usr/bin/docker pull carolina/sample_app_rails_4_image:latest
    ExecStart=/usr/bin/docker run --rm --name app-job \
      -v "volume-public:/usr/src/app/public" --entrypoint "./setup.sh" \

```

```

-e "POSTGRES_USER=${POSTGRES_USER}" \
-e "POSTGRES_PASSWORD=${POSTGRES_PASSWORD}" \
-w "/usr/src/app" --link "some-postgres:db" \
carolina/sample_app_rails_4_image:latest
[Install]
WantedBy=multi-user.target
- name: app-task.service
command: start
enable: true
content: |
[Unit]
Description=app-task container that runs the server puma
After=docker.service volume-public.service postgresql.service
    app-job.service
Requires=docker.service volume-public.service postgresql.service
    app-job.service
[Service]
TimeoutStartSec=0
Restart=always
EnvironmentFile=/etc/postgres-credentials.env
ExecStartPre=/usr/bin/docker kill app-task
ExecStartPre=/usr/bin/docker rm app-task
ExecStartPre=/usr/bin/docker pull carolina/sample_app_rails_4_image:latest
ExecStart=/usr/bin/docker run --rm --name app-task \
-e "POSTGRES_USER=${POSTGRES_USER}" \
-e "POSTGRES_PASSWORD=${POSTGRES_PASSWORD}" \
-w "/usr/src/app" -v "volume-public:/usr/src/app/public" \
--link "some-postgres:db" carolina/sample_app_rails_4_image:latest \
/bin/bash -c "cp config/database.yml.postgresql \
config/database.yml && cp ./secret.example ./secret && puma -p 9292"
ExecStop=/usr/bin/docker stop app-task
[Install]
WantedBy=multi-user.target
- name: nginx.service
command: start
enable: true
content: |
[Unit]
Description=some-nginx container that runs a reverse proxy server and
    a web server
After=docker.service volume-public.service postgresql.service
    app-job.service app-task.service
Requires=docker.service volume-public.service postgresql.service
    app-job.service app-task.service
[Service]
TimeoutStartSec=0
Restart=always
ExecStartPre=/usr/bin/docker kill some-nginx
ExecStartPre=/usr/bin/docker rm some-nginx
ExecStartPre=/usr/bin/docker pull nginx
ExecStart=/usr/bin/docker run --rm --name some-nginx \
-v "/etc/nginx.conf:/etc/nginx/conf.d/default.conf" \
-p "80:80" --link "app-task:app" \
-v "volume-public:/usr/src/app/public" nginx
ExecStop=/usr/bin/docker stop some-nginx
[Install]
WantedBy=multi-user.target

```

El fichero es validado con la utilidad *Config Validator*.

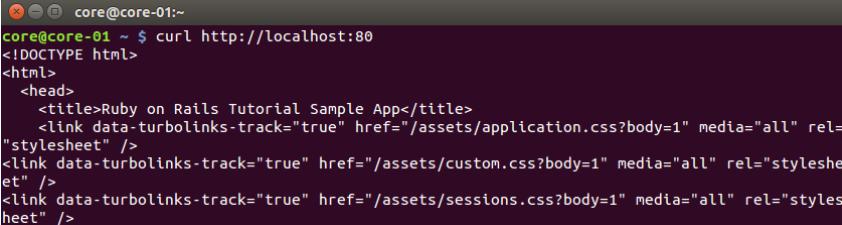
Con el despliegue se comprueba el correcto funcionamiento de la aplicación:

```
$ vagrant up && vagrant ssh core-01
```

3.5.7 Resultado

Tanto si se decide realizar el despliegue manual o automático con unidades usando CoreOS se obtiene el correcto funcionamiento de la aplicación a través de cada servicio independiente. El

resultado en ambos casos ha sido positivo:



```
core@core-01:~$ curl http://localhost:80
<!DOCTYPE html>
<html>
  <head>
    <title>Ruby on Rails Tutorial Sample App</title>
    <link data-turbolinks-track="true" href="/assets/application.css?body=1" media="all" rel="stylesheet" />
    <link data-turbolinks-track="true" href="/assets/custom.css?body=1" media="all" rel="stylesheet" />
    <link data-turbolinks-track="true" href="/assets/sessions.css?body=1" media="all" rel="stylesheet" />
```

Figura 3.28: Despliegue manual y automático. Acceso: máquina CoreOS.

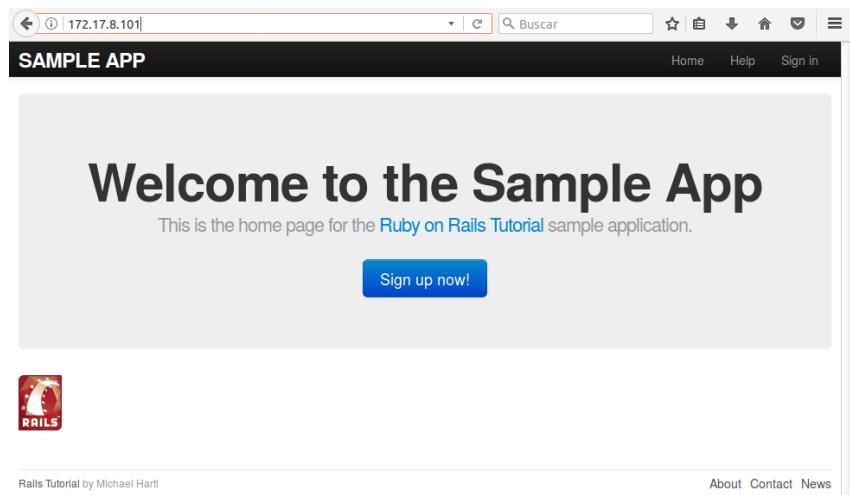


Figura 3.29: Despliegue manual y automático. Acceso: sistema anfitrión.

3.6 Iteración 5: Despliegue en la nube pública de Amazon Web Services

En esta iteración se pretende desplegar la infraestructura en la nube pública de Amazon Web Services. Para ello se trabajará el mismo fichero `config.rb`, el nuevo fichero `user-data.sampleapp.aws` y el fichero `Vagrantfile` en el que se introducirá el tratamiento e integración de este proveedor.

En la Figura 3.30 puede distinguirse la distribución de las unidades de servicio, y por tanto, de los contenedores. El proxy, presente en el contenedor `some-nginx`, que redirige al servidor web de la aplicación, en la máquina `core-01`. Nginx será el servicio al cual los clientes y usuarios finales realizan las peticiones para acceder a la aplicación. No solo se va a tener un servidor de la aplicación, éste puede estar replicado desde en una a todas las instancias de la infraestructura y el servicio Nginx deberá balancear la carga entre ellos. Para controlar los servidores web que se registran y/o dejan de estar en el sistema se implementará un nuevo contenedor, `confd`, y un nuevo volumen Docker, `conf-data`. A su vez, el contenedor `some-postgres` para la base de datos se ubicará en la máquina `core-02`. A él tiene que acceder tanto el contenedor `app-job`, que la crea, migra y alimenta, como el contenedor `app-task` que se dirige al mismo para obtener y escribir la información solicitada y añadida en la aplicación web. Ello se consigue con la implementación del contenedor `skydns` que les servirá como servidor DNS, trabajando con las claves y valores de etc.

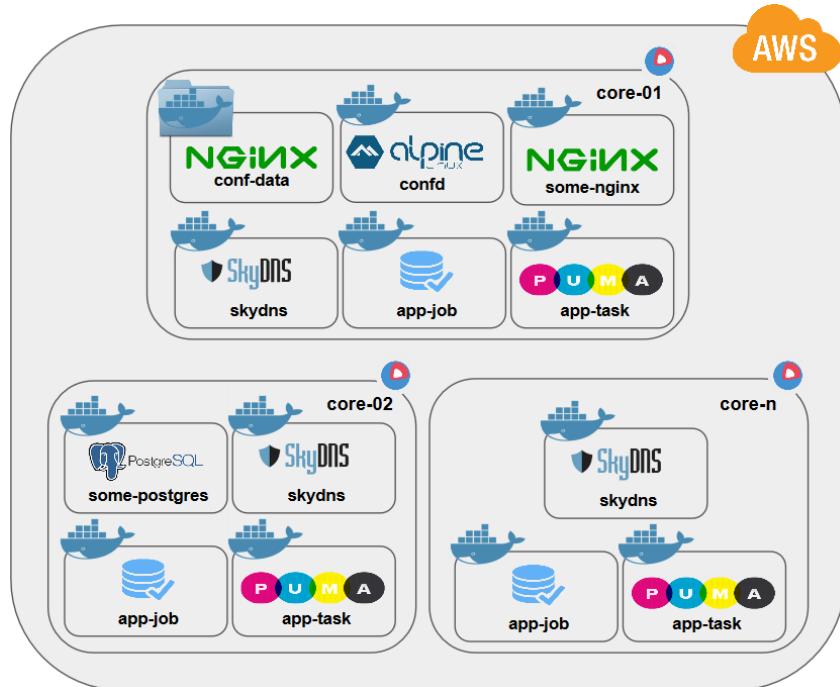


Figura 3.30: Distribución de la infraestructura a desplegar en AWS.

La relación entre los contenedores, independientemente de su ubicación y replicación en otras instancias, en la manera en la que se ha comentado, puede apreciarse con mayor detalle en la Figura 3.31.

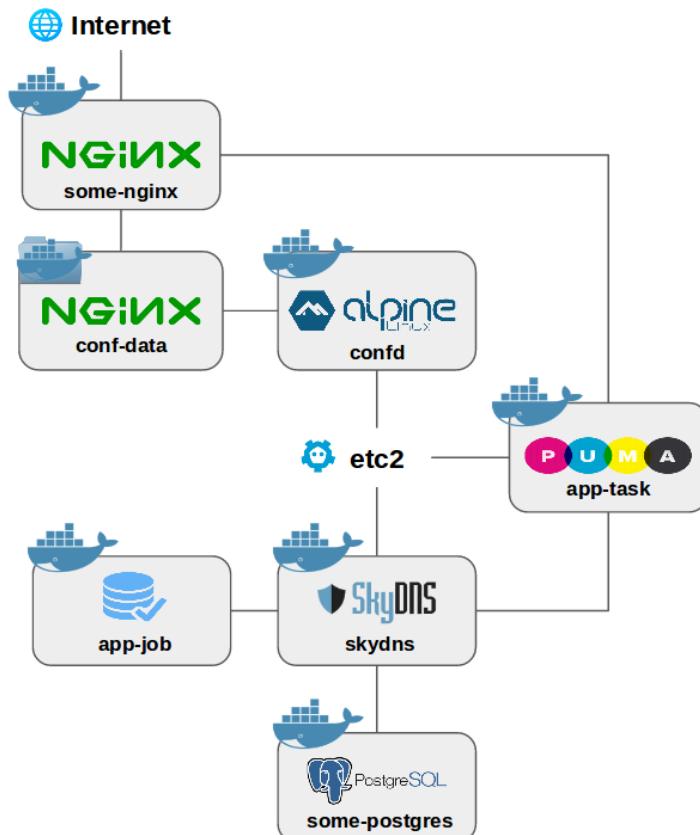


Figura 3.31: Relación entre los contenedores de la Infraestructura.

Cuando la infraestructura se despliega, el contenedor `some-postgres` se registra en `skydns`, que utiliza etc para almacenar las claves y valores. De esta manera, `some-postgres` puede ser descubierto por los contenedores `app-job`, donde el primero en iniciarse creará, migrará y alimentará la base de datos. Cuando arrancan los contenedores `app-task`, servidores web de la aplicación, se registran en etc. El contenedor `confd` vigila estos cambios cada 5 segundos para actualizar la configuración de `some-nginx` y éste pueda resolver hacia ellos. Puesto que se comparte un fichero entre dos contenedores se utiliza el volumen `conf-data`. Así, cuando un cliente realiza una petición a través de Internet a `some-nginx`, éste puede redirigirle a uno de los servidores `app-task`, el cuál consultará `skydns` para poder obtener y escribir información en la base de datos. La política para redireccionar a los servidores web será *Round Robin*, del primero al último.

Al ya no estar los contenedores en la misma instancia, compartir el volumen Docker `volume-public` de la manera en la que se ha ido haciendo no es posible. Como no es el propósito directo ya no será implementado y se incorporará a la sección de Trabajos Futuros 5.2.2.

El servicio quedará disponible mediante el despliegue de la infraestructura desde Vagrant, estableciendo antes las variables de entorno correspondientes al proveedor. El acceso al servicio, que puede ser visto en la Figura 3.71, se hará desde la máquina `core-01`, que contiene el proxy.

```
$ . ~/.aws-credentials/aws-credentials
$ vagrant up --provider=aws && vagrant ssh core-01
# curl http://localhost:80
```

3.6.1 Obtención de la cuenta AWS

Para la realización de esta iteración es necesario crear una cuenta en Amazon Web Services. En el caso presente se utilizará una cuenta que posee una capa gratuita, con una duración de un año, que se usa en conjunto con un crédito de \$50 dólares. Este crédito ha sido facilitado al pertenecer al programa *AWS Educate*[27]. Se trata de una iniciativa de Amazon que proporciona a los estudiantes y personal docente los recursos necesarios para acelerar el aprendizaje relacionado con la nube.

Los servicios de AWS se localizan en múltiples ubicaciones por todo el mundo. En este trabajo se escogerá la región *US East* y zona *N. Virginia*.

Se hace uso del servicio AWS *Identity and Access Management* (IAM) con la intención de controlar de forma segura el acceso a servicios y recursos. Con el objetivo de tener un usuario para realizar las actividades de desarrollo, el primer paso será la creación del grupo *deployers* con permiso para manejar las instancias EC2. Luego, se crea el usuario *deployer* con tipo de acceso *Management Console access*. Al mismo se le facilita una contraseña y se le indica que pertenece al grupo recién creado.

3.6.2 Creación de la red y subred virtual

El servicio *Virtual Private Cloud* (VPC) permite controlar todos los aspectos del entorno de red virtual. Será usado para hacer la selección del rango de direcciones IP que se les dará a las máquinas virtuales y para crear una subred acorde.

En primer lugar se crea una VPC llamada `vpc-sampleapp` con el rango de direcciones IPv4 10.0.0.0/16:

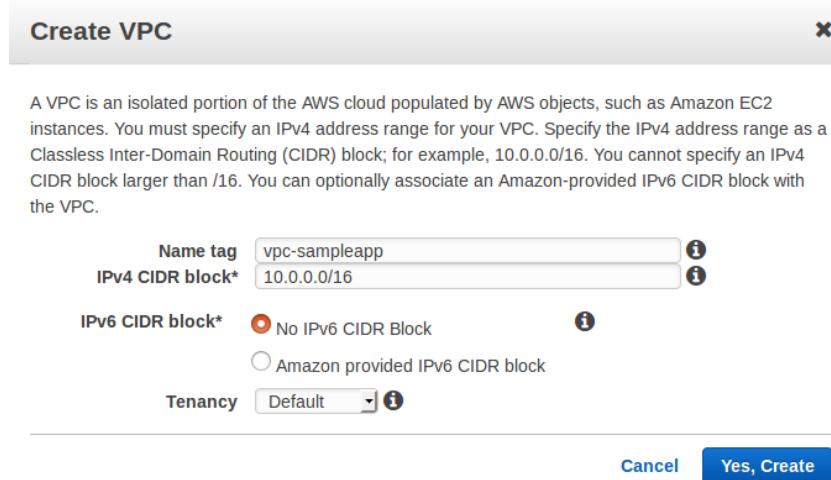


Figura 3.32: Creación de la VPC *vpc-sampleapp*.

En segundo lugar se crea una subred llamada *subnet-sampleapp*, perteneciente a la anterior VPC, con el rango de direcciones IPv4 10.0.0.0/24:

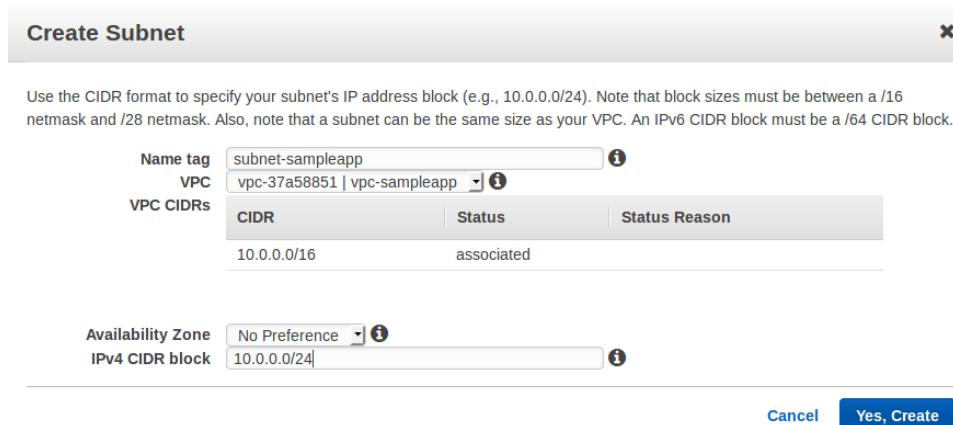


Figura 3.33: Creación de la subred *subnet-sampleapp*.

3.6.3 Creación del par de claves y del grupo de seguridad

Lo siguiente es preparar una pareja de claves *key pair* para poder entrar por SSH en la instancia. Para ello es necesario crear un *Security Group* que defina los puertos de entrada por los que podrá recibir peticiones.

Lo primero se realiza accediendo al servicio Amazon Elastic Cloud Computing (Amazon EC2) y navegando hasta *Network & Security, Key Pairs*. Aquí, se crea una nueva pareja de claves con el nombre **keypair.pem**, la cual se guarda en un directorio oculto de nombre **~/.aws-credentials**, facilitando permisos de lectura y escritura al propietario.

Ahora se navega hasta *Network & Security, Security Groups* y se crea el grupo de seguridad *deploying-security-group* con las reglas TCP entrantes que permitan dejar pasar a la instancia las peticiones que llegan para conexiones a los puertos asignados a los protocolos SSH (22) y HTTP (80). También se añaden las reglas TCP a los puertos que utiliza CoreOS en sus comunicaciones con los clientes, siendo 2379, 2380, 4001 y 7001. Para que las distintas máquinas virtuales que se creen, formando un clúster CoreOS, tengan conectividad con los

contenedores de otras máquinas virtuales miembro se habilita el puerto 8285 mediante una regla UDP. Finalmente, se añaden las reglas TCP sobre los puertos que afectan a la aplicación, 9292 para el servidor puma y 5432 para el servidor PostgreSQL.

3.6.4 Configuración del fichero user-data

Para llevar a cabo esta parte se necesita una nueva configuración **cloud-config**. Este nuevo **user-data** recibirá el nombre de **user-data.sampleapp.aws**. Tendrá, únicamente, dos diferencias respecto al anterior, por ello se copia:

```
$ cp user-data.sampleapp.vbox user-data.sampleapp.aws
```

La primera de las diferencias será especificar al servicio flannel que ha de usar el rango de direcciones IPv4 privadas, pues los contenedores internos a la máquina usarán una red privada para no ser accesibles desde el exterior. El segundo cambio detalla que la red privada a usar será la 172.17.0.0/16 y el protocolo UDP.

Listado 3.25: Fichero **user-data.sampleapp.aws**

```
coreos:
  .
  flannel:
    interface: "$private_ipv4"
  units:
    .
    - name: flanneld.service
      drop-ins:
        - name: 50-network-config.conf
          content: |
            [Service]
            ExecStartPre=/usr/bin/etcctl set /coreos.com/network/config
              '{ "Network": "172.17.0.0/16", "Backend": { "Type": "udp" } }'
```

3.6.5 Configuración del fichero Vagrantfile

Primero se instala el plugin necesario para utilizar AWS como proveedor:

```
$ vagrant plugin install vagrant-aws
```

Luego se añade al fichero **Vagrantfile** la comprobación de que se encuentre instalado en el sistema anfitrión.

Para iniciar sesión en la cuenta AWS como usuario *deployer* se agregará un fichero local en el que especificar estos datos, con la intención de ser tratados como variables de entorno. Para ello se crea el fichero **~/.aws-credentials/aws-credentials** con permiso de ejecución *chmod +x* y contenido:

Listado 3.26: Fichero **~/.aws-credentials/aws-credentials**

```
export AWS_KEY='XXXXXXXXXXXXXXXXXXXX'
export AWS_SECRET='XXXXXXXXXXXXXXXXXXXXXXXXXX'
export AWS_KEYNAME='keypair'
export AWS_KEYPATH='~/.aws-credentials/keypair.pem'
```

Así, en el propio **Vagrantfile** se especificará que las variables de entorno **ENV['AWS_KEY']**, **ENV['AWS_SECRET']** y **ENV['AWS_KEYNAME']** deben establecerse para continuar. Para ello se ejecuta en el directorio actual:

```
$ . ~/aws-credentials/aws-credentials
```

Como en el caso anterior, se prepara la copia del fichero **cloud-config** de **user-data.sampleapp.aws** a **user-data**.

Las opciones necesarias para el despliegue desde Vagrant con este proveedor determinan la configuración de la instancia o instancias a lanzar. Las características a definir son:

- **access_key_id**: Primera clave, perteneciente al identificador del usuario.
- **secret_access_key**: Segunda clave, perteneciente a la contraseña secreta.
- **keypair_name**: Nombre del archivo que contiene el par de claves.
- **aws_region**: Región.
- **aws_availability_zone**: Zona de disponibilidad.
- **aws_subnet_id**: Identificador de la subred.
- **aws_security_groups**: Identificador del grupo de seguridad.
- **aws_ami**: Identificador de la *Amazon Machine Image* (AMI). Se escoge *CoreOS-alpha-1339.0.0-hvm - ami-00598116*, de 64 bits con virtualización HVM.
- **aws_instance_type**: Tipo de instancia a lanzar. El tipo será *t2.micro Free tier eligible*.
- **aws_elastic_ip**: Uso de direcciones IP elásticas para el acceso remoto al servicio.
- **private_ip_addresses**: Dirección IP privada a asignar a la instancia.
- **user_data**: Contenido del fichero **user-data**.

Los valores que tomarán estos parámetros son los relativos a las diferentes creaciones que se han ido realizando con la cuenta en uso.

Otro aspecto a indicar es la deshabilitación de la carpeta compartida por defecto, **/vagrant**, con el directorio actual del sistema anfitrión, pues es prescindible. También será necesario indicar la **url** que utilizará Vagrant como **box**, imagen para la máquina virtual. Dicha indicación ha de ir acompañada del nombre de usuario, ruta del fichero del par de claves y la opción de que no se desea clave para iniciar sesión. Todo ello especificado al protocolo SSH (22), usado para conectarse con la instancia.

Finalmente queda copiar el fichero **cloud-config** a la instancia. Aquí se ha optado por crear una copia del fichero **user-data** por máquina, identificándola con un número. El contenido del fichero original será copiado al segundo mediante métodos propios a **YAML**. Una vez hecha la copia, se indica la dirección IP que la máquina va adquirir, siendo 10.0.0.10X, y se pasa el contenido mediante su lectura, haciendo uso del campo **user_data**.

Así, al fichero **Vagrantfile** se le añade lo siguiente:

Listado 3.27: Fichero **Vagrantfile**

```
if (!ARGV.nil? && ARGV.join(' ').include?('provider=aws'))
  unless Vagrant.has_plugin?("vagrant-aws")
    abort("Did not detect vagrant-aws plugin...
          vagrant plugin install vagrant-aws")
  end
  unless ENV['AWS_KEY'] && ENV['AWS_SECRET'] && ENV['AWS_KEYNAME']
    abort("$AWS_KEY && $AWS_SECRET && $AWS_KEYNAME should set before...")
  end
  FileUtils.cp_r(File.join(File.dirname(__FILE__), "user-data.sampleapp.aws"),
                 File.join(File.dirname(__FILE__), "user-data"), :remove_destination => true)
end
```

```

$aws_region = 'us-east-1'
$aws_availability_zone = 'us-east-1a'
$aws_subnet_id = 'subnet-b17d79ea'
$aws_security_groups = 'sg-3319964c'
$aws_ami = 'ami-00598116'
$aws_instance_type = 't2.micro'
$aws_elastic_ip = true

Vagrant.configure("2") do |config|
  config.vm.provider :aws do |aws, override|
    aws.access_key_id = ENV['AWS_KEY']
    aws.secret_access_key = ENV['AWS_SECRET']
    aws.keypair_name = ENV['AWS_KEYNAME']
    aws.security_groups = $aws_security_groups
    aws.ami = $aws_ami
    aws.instance_type = $aws_instance_type
    aws.region = $aws_region
    aws.subnet_id = $aws_subnet_id
    aws.elastic_ip = $aws_elastic_ip
    override.vm.synced_folder ".", "/vagrant", disabled: true
    override.ssh.username = "core"
    override.ssh.private_key_path = ENV['AWS_KEYPATH']
    override.ssh.insert_key = false
    override.vm.box_url = "https://github.com/mitchellh/vagrant-aws/raw/master/
      dummy.box"
  end

  (1..$num_instances).each do |i|
    config.vm.define vm_name = "%s-%02d" % [$instance_name_prefix, i] do |config|
      if File.exist?(CLOUD_CONFIG_PATH) && ARGV[0].eql?('up')

        config.vm.provider :aws do |aws, override|
          user_data_specific = "#{CLOUD_CONFIG_PATH}-#{i}"
          require 'yaml'
          data = YAML.load(IO.readlines(CLOUD_CONFIG_PATH)[1..-1].join)
          yaml = YAML.dump(data)
          File.open(user_data_specific, 'w') do |file|
            file.write("#cloud-config\n\n#{yaml}")
          end
          aws.private_ip_address = "10.0.0.#{100+i}"
          aws.user_data = File.read(user_data_specific)
        end
      end
    end
  end
end

```

3.6.6 Despliegue de una instancia

A continuación se despliega esta infraestructura y se accede a la instancia:

```
$ vagrant up --provider=aws && vagrant ssh core-01
```

Ahora se prueba que la salud del clúster es positiva:

```
# etcdctl cluster-health
```

```
core@ip-10-0-0-101:~$ etcdctl cluster-health
member 13ff859acff9b3e1 is healthy: got healthy result from http://34.203.92.1:2379
cluster is healthy
core@ip-10-0-0-101:~$
```

Figura 3.34: Comprobación de la salud del clúster con 1 instancia.

Además se comprueba que la red virtual funciona correctamente:

```
# sudo systemctl status flanneld
```

```
core@ip-10-0-0-101:~$ sudo systemctl status flanneld
● flanneld.service - Flannel - Network fabric for containers (System Application Container)
  Loaded: loaded (/usr/lib/systemd/system/flanneld.service; enabled; vendor preset: disabled)
  Drop-In: /etc/systemd/system/flanneld.service.d
            └─50-network-config.conf
    Active: active (running) since Wed 2017-04-05 14:02:33 UTC; 3min 48s ago
      Docs: https://github.com/coreos/flannel
   Process: 1043 ExecStartPre=/usr/bin/etcctl set /coreos.com/network/config { "Network": "1
   Process: 1031 ExecStartPre=/usr/bin/rkt rm --uuid-file=/var/lib/coreos/flannel-wrapper.uvi
   Process: 1023 ExecStartPre=/usr/bin/mkdir --parents /var/lib/coreos /run/flannel (code=exit
   Process: 1022 ExecStartPre=/sbin/modprobe ip_tables (code=exited, status=0/SUCCESS)
 Main PID: 1067 (flanneld)
    Tasks: 8
   Memory: 9.7M
      CPU: 2.921s
     CGroup: /system.slice/flanneld.service
             └─1067 /opt/bin/flanneld --ip-masq=true
```

Figura 3.35: Comprobación de la red virtual - **core01**.

Por último se comprueba desde la consola de comandos que se puede acceder al servicio correctamente con el comando **curl**:

```
# curl http://localhost:80 | tail -n 15
```

```
core@ip-10-0-0-101:~$ curl http://localhost:80 | tail -n 15
% Total    % Received % Xferd  Average Speed   Time   Time     Current
          Dload  Upload Total   Spent    Left Speed
100  3983     0  3983     0      0  139k      0  ---:---:---:---:---:--- 144k
<nav>
  <ul>
    <li><a href="/about">About</a></li>
    <li><a href="/contact">Contact</a></li>
    <li><a href="http://news.railstutorial.org/">News</a></li>
  </ul>
</nav>
</footer>
<pre class="debug_dump">--- !ruby/hash:ActionController::Parameters
controller: static_pages
action: home
</pre>
</div>
</body>
</html>core@ip-10-0-0-101:~$
```

Figura 3.36: Acceso al servicio desde la máquina **core01**.

Además, se utiliza la IP elástica para acceder a través del navegador web:

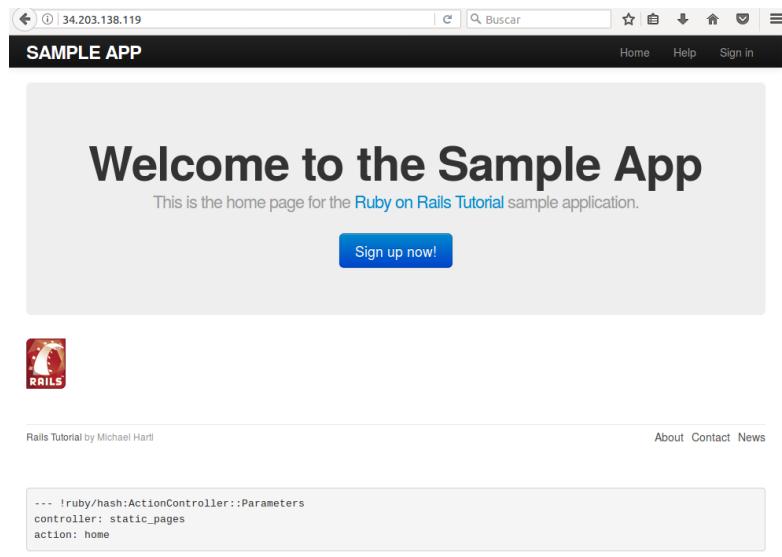


Figura 3.37: Acceso al servicio desde el sistema anfitrión.

Desde la consola de AWS se puede apreciar dicha instancia:

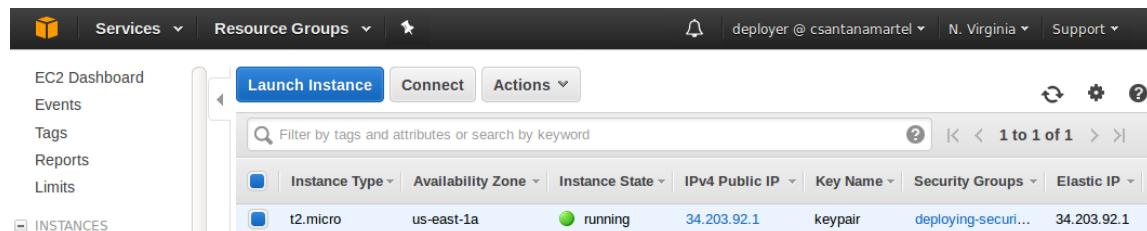


Figura 3.38: Máquina **core-01** desde la consola de AWS.

Para destruir la infraestructura y los recursos asociados a ella, se ejecuta:

```
$ vagrant destroy -f
```

3.6.7 Despliegue de tres instancias

Para realizar el despliegue de un clúster con varias máquinas virtuales se cambia el valor de la variable `num_instances` a 3 en los ficheros `Vagrantfile` y `config.rb`. De esta manera se tiene en cada una de las instancias una copia del mismo servicio.

A continuación se despliega esta infraestructura:

```
$ vagrant up --provider=aws
```

Se prueba, desde el sistema anfitrión, que la salud del clúster es positiva en las máquinas:

```
$ for i in 1 2 3; do vagrant ssh core-0$i -c 'etcdctl cluster-health'; done
```

```
root@carolina-SM:/home/carolina/proyecto/coreos-vagrant
root@carolina-SM:/home/carolina/proyecto/coreos-vagrant# for i in 1 2 3; do vagrant ssh core-0$i -c 'etcdctl cluster-health'; done
member 2672d9a008a8ae59 is healthy: got healthy result from http://34.192.157.142:2379
member e3064790da22bef4 is healthy: got healthy result from http://50.17.170.158:2379
member fff973a6a741abf5 is healthy: got healthy result from http://54.152.231.62:2379
cluster is healthy
Connection to 34.192.157.142 closed.
member 2672d9a008a8ae59 is healthy: got healthy result from http://34.192.157.142:2379
member e3064790da22bef4 is healthy: got healthy result from http://50.17.170.158:2379
member fff973a6a741abf5 is healthy: got healthy result from http://54.152.231.62:2379
cluster is healthy
Connection to 50.17.170.158 closed.
member 2672d9a008a8ae59 is healthy: got healthy result from http://34.192.157.142:2379
member e3064790da22bef4 is healthy: got healthy result from http://50.17.170.158:2379
member fff973a6a741abf5 is healthy: got healthy result from http://54.152.231.62:2379
cluster is healthy
Connection to 54.152.231.62 closed.
root@carolina-SM:/home/carolina/proyecto/coreos-vagrant#
```

Figura 3.39: Comprobación de la salud del clúster.

Además, se comprueba que la red virtual funciona correctamente:

```
$ for i in 1 2 3; do vagrant ssh core-0$i -c 'sudo systemctl flanneld \
| head -n16'; done
```

```
root@carolina-SM:/home/carolina/proyecto/coreos-vagrant
root@carolina-SM:/home/carolina/proyecto/coreos-vagrant# for i in 1 2 3; do vagrant ssh core-0$i -c 'sudo systemctl flanneld | head -n16'; done
● flanneld.service - flannel - Network fabric for containers (System Application Container)
  Loaded: loaded (/usr/lib/systemd/system/flanneld.service; enabled; vendor preset: Drop-In: /etc/systemd/system/flanneld.service.d
          └─50-network-config.conf
    Active: active (running) since Fri 2017-04-07 08:59:51 UTC; 2min 50s ago
      Docs: https://github.com/coreos/flannel
   Process: 1087 ExecStartPre=/usr/bin/etcdctl set /coreos.com/network/config { "Netwo
   Process: 1078 ExecStartPre=/usr/bin/rkt rm --uuid-file=/var/lib/coreos/flannel-wrap
   Process: 1074 ExecStartPre=/usr/bin/mkdir --parents /var/lib/coreos /run/flannel (co
   Process: 1068 ExecStartPre=/sbin/modprobe ip_tables (code=exited, status=0/SUCCESS)
 Main PID: 1095 (flanneld)
    Tasks: 8 (limit: 32768)
   Memory: 98.6M
     CPU: 853ms
    CGroup: /system.slice/flanneld.service
            └─1095 /opt/bin/flanneld --ip-masq=true
Connection to 34.192.157.142 closed.
● flanneld.service - flannel - Network fabric for containers (System Application Container)
  Loaded: loaded (/usr/lib/systemd/system/flanneld.service; enabled; vendor preset: Drop-In: /etc/systemd/system/flanneld.service.d
          └─50-network-config.conf
    Active: active (running) since Fri 2017-04-07 08:59:43 UTC; 3min 5s ago
      Docs: https://github.com/coreos/flannel
   Process: 943 ExecStartPre=/usr/bin/etcdctl set /coreos.com/network/config { "Netwo
   Process: 939 ExecStartPre=/usr/bin/rkt rm --uuid-file=/var/lib/coreos/flannel-wrap
   Process: 935 ExecStartPre=/usr/bin/mkdir --parents /var/lib/coreos /run/flannel (co
   Process: 778 ExecStartPre=/sbin/modprobe ip_tables (code=exited, status=0/SUCCESS)
 Main PID: 1016 (flanneld)
    Tasks: 8 (limit: 32768)
   Memory: 116.3M
     CPU: 868ms
    CGroup: /system.slice/flanneld.service
            └─1016 /opt/bin/flanneld --ip-masq=true
Connection to 50.17.170.158 closed.
● flanneld.service - flannel - Network fabric for containers (System Application Container)
  Loaded: loaded (/usr/lib/systemd/system/flanneld.service; enabled; vendor preset: Drop-In: /etc/systemd/system/flanneld.service.d
          └─50-network-config.conf
    Active: active (running) since Fri 2017-04-07 09:00:01 UTC; 2min 55s ago
      Docs: https://github.com/coreos/flannel
   Process: 969 ExecStartPre=/usr/bin/etcdctl set /coreos.com/network/config { "Netwo
   Process: 958 ExecStartPre=/usr/bin/rkt rm --uuid-file=/var/lib/coreos/flannel-wrap
   Process: 955 ExecStartPre=/usr/bin/mkdir --parents /var/lib/coreos /run/flannel (co
   Process: 948 ExecStartPre=/sbin/modprobe ip_tables (code=exited, status=0/SUCCESS)
 Main PID: 976 (flanneld)
    Tasks: 8 (limit: 32768)
   Memory: 96.6M
     CPU: 828ms
    CGroup: /system.slice/flanneld.service
            └─976 /opt/bin/flanneld --ip-masq=true
Connection to 54.152.231.62 closed.
root@carolina-SM:/home/carolina/proyecto/coreos-vagrant#
```

Figura 3.40: Comprobación de la red virtual.

Para probar que las máquinas virtuales tienen conexión con los contenedores de otras máquinas virtuales se obtiene, en primer lugar, la dirección IP del contenedor **some-postgres** de la máquina virtual **core-03**:

```
# docker inspect some-postgres
```

```
root@carolina-SM: /home/carolina/proyecto/coreos-vagrant
        "Gateway": "172.17.82.1",
        "IPAddress": "172.17.82.2",
        "IPPrefixLen": 24,
        "IPv6Gateway": "",
        "GlobalIPv6Address": "",
        "GlobalIPv6PrefixLen": 0,
        "MacAddress": "02:42:ac:11:52:02"
    }
}
]
core@ip-10-0-0-103 ~ $
```

Figura 3.41: Obtención de la dirección IP de un contenedor en **core-03**.

Luego se prueba que haya conexión desde la máquina virtual **core-01**:

```
# ping 172.17.82.2
```

```
core@ip-10-0-0-101:~
core@ip-10-0-0-101 ~ $ ping 172.17.82.2
PING 172.17.82.2 (172.17.82.2) 56(84) bytes of data.
64 bytes from 172.17.82.2: icmp_seq=1 ttl=61 time=0.913 ms
64 bytes from 172.17.82.2: icmp_seq=2 ttl=61 time=0.899 ms
^C
--- 172.17.82.2 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1064ms
rtt min/avg/max/mdev = 0.899/0.906/0.913/0.007 ms
core@ip-10-0-0-101 ~ $
```

Figura 3.42: Prueba de conexión a un contenedor de **core-03** desde **core-01**.

Por último se comprueba que se puede acceder al servicio desde la consola de comandos, con el comando **curl**, y desde el navegador web, con la dirección IP elástica. Desde la consola de AWS se podrán observar las instancias:

```
$ for i in 1 2 3; do vagrant ssh core-0$! -c 'curl http://localhost:80 \
| tail -n 15'; done
```

```

root@carolina-SM: /home/carolina/proyecto/coreos-vagrant
root@carolina-SM:/home/carolina/proyecto/coreos-vagrant# for i in 1 2 3; do vagrant
ssh core-0$i -c 'curl http://localhost:80' | tail -n 15; done
Connection to 34.192.157.142 closed.
<nav>
  <ul>
    <li><a href="/about">About</a></li>
    <li><a href="/contact">Contact</a></li>
    <li><a href="http://news.railstutorial.org/">News</a></li>
  </ul>
</nav>
</footer>
<pre class="debug_dump">--- !ruby/hash:ActionController::Parameters
controller: static_pages
action: home
</pre>
</div>
</body>
</html>Connection to 50.17.170.158 closed.
<nav>
  <ul>
    <li><a href="/about">About</a></li>
    <li><a href="/contact">Contact</a></li>
    <li><a href="http://news.railstutorial.org/">News</a></li>
  </ul>
</nav>
</footer>
<pre class="debug_dump">--- !ruby/hash:ActionController::Parameters
controller: static_pages
action: home
</pre>
</div>
</body>
</html>Connection to 54.152.231.62 closed.
<nav>
  <ul>
    <li><a href="/about">About</a></li>
    <li><a href="/contact">Contact</a></li>
    <li><a href="http://news.railstutorial.org/">News</a></li>
  </ul>
</nav>
</footer>
<pre class="debug_dump">--- !ruby/hash:ActionController::Parameters
controller: static_pages
action: home
</pre>
</div>
</body>
</html>root@carolina-SM: /home/carolina/proyecto/coreos-vagrant#

```

Figura 3.43: Acceso al servicio por consola de comandos.

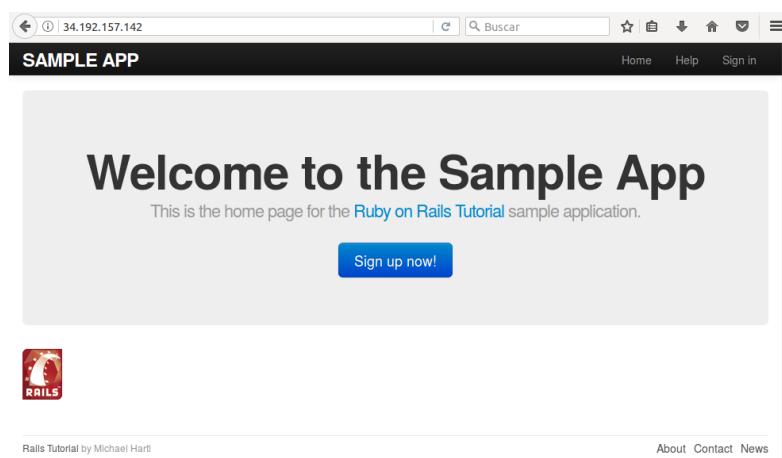


Figura 3.44: Acceso al servicio por el navegador web desde **core-01**.

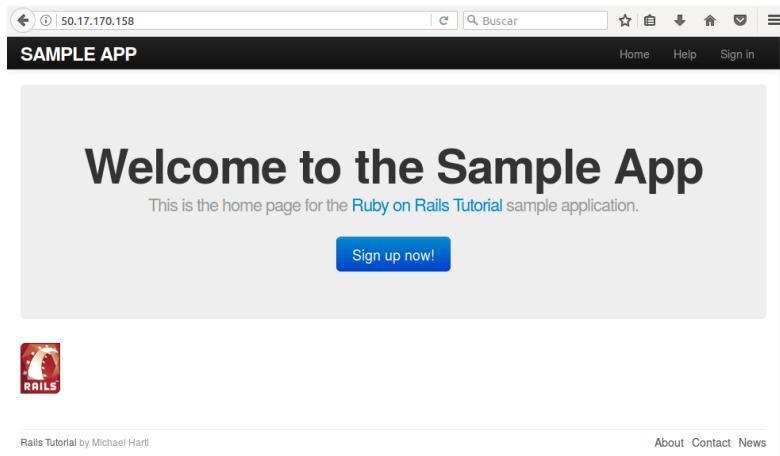


Figura 3.45: Acceso al servicio por el navegador web desde **core-02**.

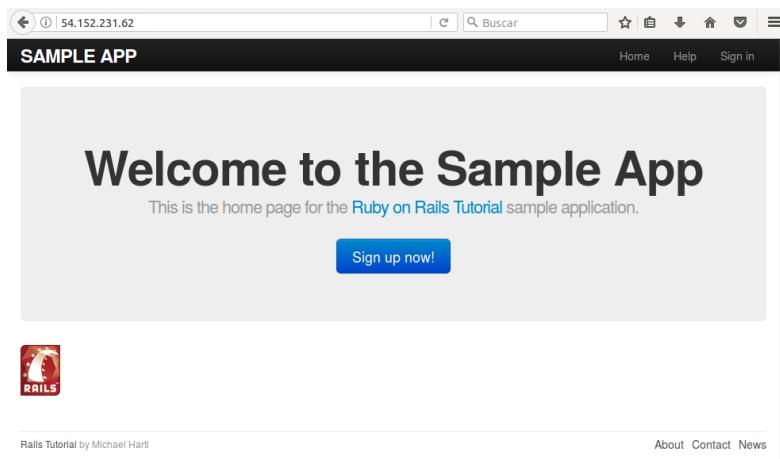


Figura 3.46: Acceso al servicio por el navegador web desde **core-03**.

Instance Type	Availability Zone	Instance State	IPv4 Public IP	Key Name	Security Groups	Elastic IP
t2.micro	us-east-1a	running	34.192.157.142	keypair	deploying-secur...	34.192.157.142
t2.micro	us-east-1a	running	54.152.231.62	keypair	deploying-secur...	54.152.231.62
t2.micro	us-east-1a	running	50.17.170.158	keypair	deploying-secur...	50.17.170.158

Figura 3.47: Consola AWS EC2 Instances - **core-01**, **core-02** y **core-03**.

3.6.8 Cambio de las unidades systemd a fleet

Para mejorar la distribución de la infraestructura se van a cambiar las unidades de servicio `systemd` por `fleet`. Para ello los servicios tienen que ser especificados como ficheros a escribir en el sistema. Concretamente se van a situar en la ruta `/home/core/`. La sección `[Install]` se cambia por la propia `fleet`, `[X-Fleet]`. Aquí se especificará el parámetro `Global` que programa la unidad en todos los agentes del clúster, de forma que desde cada una

de las instancias que lo componen se puedan ver todos los servicios que hay en ellas. Luego se crean como unidades de servicio systemd los servicios empezados por **fleet**-, para cada uno de ellos, que indicará a fleet que ha de empezar los servicios pasados como ficheros.

Continuando con el clúster de 3 instancias, se realizan las siguientes modificaciones en el fichero **user-data.sampleapp-aws**:

Listado 3.28: Fichero **user-data.sampleapp-aws**

```
#cloud-config
-----
write_files:
  -
    path: "/home/core/volume-public.service"
    permissions: "0644"
    content: |
      [Unit]
      Description= volume-public share between some-postgres, app-job, app-task
                  and some-nginx
      After=docker.service
      Requires=docker.service
      [Service]
      TimeoutStartSec=0
      ExecStart=/usr/bin/docker volume create --name volume-public
      [X-Fleet]
      Global=true
  -
    path: "/home/core/postgresql.service"
    permissions: "0644"
    content: |
      [Unit]
      Description=PostgreSQL database
      After=docker.service volume-public.service
      Requires=docker.service volume-public.service
      [Service]
      TimeoutStartSec=0
      Restart=always
      EnvironmentFile=/etc/postgres-credentials.env
      ExecStartPre=/usr/bin/docker kill some-postgres
      ExecStartPre=/usr/bin/docker rm some-postgres
      ExecStartPre=/usr/bin/docker pull postgres
      ExecStart=/usr/bin/docker run --rm --name some-postgres \
      -e "POSTGRES_USER=${POSTGRES_USER}" \
      -e "POSTGRES_PASSWORD=${POSTGRES_PASSWORD}" \
      -v "volume-public:/var/lib/postgresql" -p "5432:5432" postgres
      ExecStop=/usr/bin/docker stop some-postgres
      [X-Fleet]
      Global=true
  -
    path: "/home/core/app-job.service"
    permissions: "0644"
    content: |
      [Unit]
      Description=executable app-job container that creates, migrates, seeds and
                  populates the database
      After=docker.service volume-public.service postgresql.service
      Requires=docker.service volume-public.service postgresql.service
      [Service]
      TimeoutStartSec=0
      EnvironmentFile=/etc/postgres-credentials.env
      ExecStartPre=/usr/bin/docker kill app-job
      ExecStartPre=/usr/bin/docker rm app-job
      ExecStartPre=/usr/bin/docker pull carolina/sample_app_rails_4_image:latest
      ExecStart=/usr/bin/docker run --rm --name app-job \
      -v "volume-public:/usr/src/app/public" --entrypoint "./setup.sh" \
      -e "POSTGRES_USER=${POSTGRES_USER}" \
      -e "POSTGRES_PASSWORD=${POSTGRES_PASSWORD}" \
      -w "/usr/src/app" --link "some-postgres:db" \
      carolina/sample_app_rails_4_image:latest
      [X-Fleet]
      Global=true
  -
    path: "/home/core/app-task.service"
```

```

permissions: "0644"
content: |
  [Unit]
  Description=app-task container that runs the server puma
  After=docker.service volume-public.service postgresql.service
    app-job.service
  Requires=docker.service volume-public.service postgresql.service
    app-job.service
  [Service]
  TimeoutStartSec=0
  Restart=always
  EnvironmentFile=/etc/postgres-credentials.env
  ExecStartPre=/usr/bin/docker kill app-task
  ExecStartPre=/usr/bin/docker rm app-task
  ExecStartPre=/usr/bin/docker pull carolina/sample_app_rails_4_image:latest
  ExecStart=/usr/bin/docker run --rm --name app-task \
-e "POSTGRES_USER=${POSTGRES_USER}" \
-e "POSTGRES_PASSWORD=${POSTGRES_PASSWORD}" \
-w "/usr/src/app" -v "volume-public:/usr/src/app/public" \
--link "some-postgres:db" \
carolina/sample_app_rails_4_image:latest \
/bin/bash -c "cp config/database.yml.postgresql config/database.yml && \
cp ./secret.example ./secret && puma -p 9292"
  ExecStop=/usr/bin/docker stop app-task
  [X-Fleet]
  Global=true
- path: "/home/core/nginx.service"
permissions: "0644"
content: |
  [Unit]
  Description=some-nginx container that runs a reverse proxy server and a
    web server
  After=docker.service volume-public.service postgresql.service
    app-job.service app-task.service
  Requires=docker.service volume-public.service postgresql.service
    app-job.service app-task.service
  [Service]
  TimeoutStartSec=0
  Restart=always
  ExecStartPre=/usr/bin/docker kill some-nginx
  ExecStartPre=/usr/bin/docker rm some-nginx
  ExecStartPre=/usr/bin/docker pull nginx
  ExecStart=/usr/bin/docker run --rm --name some-nginx \
-v "/etc/nginx.conf:/etc/nginx/conf.d/default.conf" \
-p "80:80" --link "app-task:app" \
-v "volume-public:/usr/src/app/public" nginx
  ExecStop=/usr/bin/docker stop some-nginx
  [X-Fleet]
  Global=true
coreos:
.
units:
.
- name: fleet-volume-public.service
  command: start
  content: |
    [Unit]
    Description=Start volume-public.service using fleet
    [Service]
    ExecStart=/usr/bin/fleetctl start /home/core/volume-public.service
- name: fleet-postgresql.service
  command: start
  content: |
    [Unit]
    Description=Start postgresql.service using fleet
    [Service]
    ExecStart=/usr/bin/fleetctl start /home/core/postgresql.service
- name: fleet-app-job.service
  command: start
  content: |
    [Unit]
    Description=Start app-job.service using fleet

```

```
[Service]
ExecStart=/usr/bin/fleetctl start /home/core/app-job.service
- name: fleet-app-task.service
  command: start
  content: |
    [Unit]
    Description=Start app-task.service using fleet
    [Service]
    ExecStart=/usr/bin/fleetctl start /home/core/app-task.service
- name: fleet-nginx.service
  command: start
  content: |
    [Unit]
    Description=Start nginx.service using fleet
    [Service]
    ExecStart=/usr/bin/fleetctl start /home/core/nginx.service
```

Para comprobar que funciona se levanta la infraestructura:

```
$ vagrant up --provider=aws
```

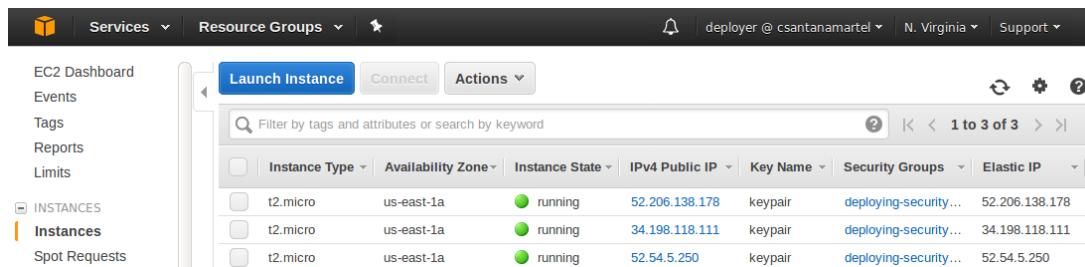


Figura 3.48: Consola AWS EC2 Instances - **core-01**, **core-02** y **core-03**.

Una vez terminado basta con entrar en la máquina **core-01** para comprobar que las unidades funcionan correctamente:

```
$ vagrant ssh core-01
# fleetctl list-units
```

```
core@ip-10-0-0-101 ~ $ fleetctl list-units
UNIT           MACHINE      ACTIVE SUB
app-job.service 4dc888ce.../34.198.118.111  inactive dead
app-job.service 7740845e.../52.54.5.250  inactive dead
app-job.service 9e009160.../52.206.138.178  inactive dead
app-task.service 4dc888ce.../34.198.118.111  active  running
app-task.service 7740845e.../52.54.5.250  active  running
app-task.service 9e009160.../52.206.138.178  active  running
nginx.service   4dc888ce.../34.198.118.111  active  running
nginx.service   7740845e.../52.54.5.250  active  running
nginx.service   9e009160.../52.206.138.178  active  running
postgresql.service 4dc888ce.../34.198.118.111  active  running
postgresql.service 7740845e.../52.54.5.250  active  running
postgresql.service 9e009160.../52.206.138.178  active  running
volume-public.service 4dc888ce.../34.198.118.111  inactive dead
volume-public.service 7740845e.../52.54.5.250  inactive dead
volume-public.service 9e009160.../52.206.138.178  inactive dead
```

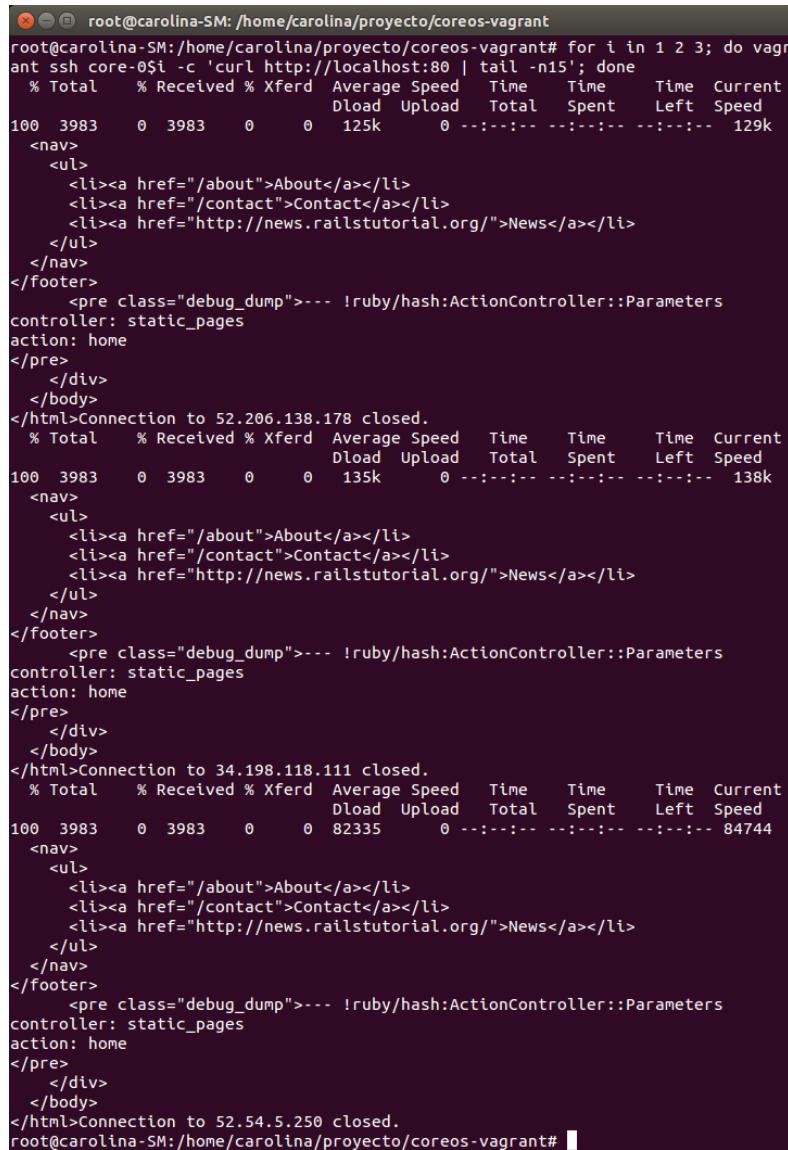
Figura 3.49: Listado de unidades fleet en cada una de las máquinas.

Como se puede apreciar, cada una de las máquinas, tiene una copia de cada servicio. Todas se han activado y se encuentran en ejecución, salvo las unidades **app-job.service** y

volume-public.service que aparecen inactivas porque son contenedores ejecutables, es decir, cuando acaban su cometido terminan.

También se comprueba que la actividad de la aplicación sigue siendo correcta, solicitándola con el comando **curl** en cada una de las máquinas:

```
$ for i in 1 2 3; do vagrant ssh core-0${i} -c 'curl http://localhost:80 \
| tail -n 15'; done
```



```
root@carolina-SM: /home/carolina/proyecto/coreos-vagrant
root@carolina-SM:/home/carolina/proyecto/coreos-vagrant# for i in 1 2 3; do vagrant ssh core-0${i} -c 'curl http://localhost:80 | tail -n15'; done
      % Total    % Received % Xferd  Average Speed   Time   Time     Time  Current
                                         Dload  Upload Total Spent   Left Speed
100  3983     0  3983     0      0   125k      0  --::-- --::-- --::-- 129k
<nav>
  <ul>
    <li><a href="/about">About</a></li>
    <li><a href="/contact">Contact</a></li>
    <li><a href="http://news.railstutorial.org/">News</a></li>
  </ul>
</nav>
</footer>
<pre class="debug_dump">--- !ruby/hash:ActionController::Parameters
controller: static_pages
action: home
</pre>
</div>
</body>
</html>Connection to 52.206.138.178 closed.
      % Total    % Received % Xferd  Average Speed   Time   Time     Time  Current
                                         Dload  Upload Total Spent   Left Speed
100  3983     0  3983     0      0   135k      0  --::-- --::-- --::-- 138k
<nav>
  <ul>
    <li><a href="/about">About</a></li>
    <li><a href="/contact">Contact</a></li>
    <li><a href="http://news.railstutorial.org/">News</a></li>
  </ul>
</nav>
</footer>
<pre class="debug_dump">--- !ruby/hash:ActionController::Parameters
controller: static_pages
action: home
</pre>
</div>
</body>
</html>Connection to 34.198.118.111 closed.
      % Total    % Received % Xferd  Average Speed   Time   Time     Time  Current
                                         Dload  Upload Total Spent   Left Speed
100  3983     0  3983     0      0  82335      0  --::-- --::-- --::-- 84744
<nav>
  <ul>
    <li><a href="/about">About</a></li>
    <li><a href="/contact">Contact</a></li>
    <li><a href="http://news.railstutorial.org/">News</a></li>
  </ul>
</nav>
</footer>
<pre class="debug_dump">--- !ruby/hash:ActionController::Parameters
controller: static_pages
action: home
</pre>
</div>
</body>
</html>Connection to 52.54.5.250 closed.
root@carolina-SM:/home/carolina/proyecto/coreos-vagrant#
```

Figura 3.50: Acceso al servicio por consola de comandos.

3.6.9 Configuración DNS usando SkyDNS

Con el objetivo de aislar la base de datos en una sola máquina se hace uso de un servidor de sistema de nombres de dominio (DNS). Es necesario puesto que los contenedores **app-job** y **app-task** necesitan acceder al contenedor **some-postgres**. Este contenedor, al no estar alojado en la misma máquina, tendrá que ser registrado en el servidor DNS para que pueda ser descubierto desde las otras máquinas. Así, los contenedores **app-job** y **app-task** resolverán

la base de datos mediante dicho servidor DNS, que tendrá presencia en todas las máquinas como un nuevo contenedor, llamado **skydns**. SkyDNS es un servicio distribuido para el anuncio y descubrimiento de servicios construidos sobre etcd, creando y leyendo registros en él. Se trata de un proyecto de código abierto que puede ser encontrado en GitHub, bajo el repositorio **skynetservices/skydns**.

Así, el primer paso consistirá en la división de la infraestructura de forma que los contenedores **volume-public**, **app-job**, **app-task** y **some-ningx** sean creados en la máquina **core-01**. Por otro lado, el contenedor **some-postgres** será creado en la máquina **core-02**. Esto se consigue usando la opción *metadata* presente en las unidades fleet. Para especificarlo es necesario añadir al fichero **Vagrantfile** un metadato diferente para cada una de las máquinas, de manera que **core-01** reciba *compute=web* y **core-02** *compute=db*. Esto ha de ser añadido bajo la carga de cada una de las líneas del fichero **cloud-config**.

Listado 3.29: Fichero **Vagrantfile**

```
:
data = YAML.load(IO.readlines(CLOUD_CONFIG_PATH) [1..-1].join)
if data['coreos'].key? 'fleet' and i==1
  data['coreos']['fleet']['metadata'] = 'compute=web'
end
if data['coreos'].key? 'fleet' and i==2
  data['coreos']['fleet']['metadata'] = 'compute=db'
end
:
```

Ahora será necesario añadir el metadato a los ficheros de las unidades de servicio, de los contenedores nombrados, en la sección *[X-Fleet]*. Para la máquina **core-01** el parámetro *MachineMetadata=compute=web* y en el fichero correspondiente a la base de datos el parámetro *MachineMetadata=compute=db*, en la máquina **core-02**.

Como el contenedor **some-postgres** ya no se estará en la misma máquina que el resto de contenedores no es posible compartir **volume-public** con ellos. Por lo tanto se eliminará en su definición el comando *-v*, así como su inclusión en los parámetros *After* y *Require*.

Lo siguiente será crear el contenedor que se corresponde con el servidor DNS, presente en todas las máquinas. Este último hecho es posible no añadiendo ningún metadato en la sección *[X-Fleet]*. Se escribe un nuevo fichero llamado **/home/core/skydns.service** antes del fichero **/home/core/volumen-public.service**. Además se le indica que será iniciado después de Docker y etcd2. Luego se procede de la misma manera que con las otras unidades de servicio. Antes de comenzar el servicio borrará el contenedor si existe previamente y luego descargará la imagen desde Docker Hub. Además, se establece la configuración de SkyDNS de forma que se especifica la dirección IP y puerto en la que debe escuchar, que será la local a la máquina por el puerto 53. También se establece el dominio para el que es autoritario, escogiéndose **sampleapp.local.**, y el tiempo de vida (TTL) de las respuestas a 30 segundos. A la hora de comenzar el servicio restará indicarle que las máquinas etcd se comunican en la dirección IP privada en uso y puerto 2379. Como se trata de una unidad fleet, será necesario añadir el servicio global **fleet-skydns.service** antes de **fleet-volume-public.service** para comenzar el servicio.

Así, habrá que añadir a **user-data.sampleapp.aws**:

Listado 3.30: Fichero **user-data.sampleapp.aws**

```
:
write_files:
```

```

- path: "/home/core/skydns.service"
  permissions: "0644"
  content: |
    [Unit]
    Description=SkyDNS
    After=docker.service etcd2.service
    Requires=docker.service etcd2.service
    [Service]
    Restart=always
    TimeoutStartSec=0
    ExecStartPre=/usr/bin/docker rm -f skydns
    ExecStartPre=/usr/bin/docker pull skynetservices/skydns
    ExecStartPre=/usr/bin/etcdctl set /skydns/config \
      '{"dns_addr":"0.0.0.0:53", "domain": "sampleapp.local.", "ttl":30}'
    ExecStart=/usr/bin/docker run --rm --name skydns \
      -e ETCD_MACHINES="http://$private_ipv4:2379" skynetservices/skydns
    ExecStop=/usr/bin/docker stop skydns
    [X-Fleet]
    Global=true

coreos:
  units:
    - name: fleet-skydns.service
      command: start
      content: |
        [Unit]
        Description=Start skydns.service using fleet
        [Service]
        ExecStart=/usr/bin/fleetctl start /home/core/skydns.service
  .

```

Además, al fichero `/home/core/postgresql.service` hay que añadirle el registro de la dirección IP del contenedor `some-postgres` para que pueda ser encontrado por los otros contenedores, desde las otras máquinas. Este registro ha de ser añadido después de que el contenedor esté en ejecución, por lo que se especificará como `ExecStartPost`. Los contenedores `app-job` y `app-task` necesitan resolverlo como `db`. Sin embargo, `app-job` también resuelve por `some-postgres`, puesto que antes de comenzar sus tareas espera a que dicho contenedor esté activo en el puerto 5432. Teniendo en cuenta estas condiciones se realizan dos registros, el primero tipo nombre canónico (CNAME) de `some-postgres.sampleapp.local.` a `db.sampleapp.local.` y el segundo tipo dirección IP (A) de `db.sampleapp.local.` a dirección IP de `some-postgres`, accesible por el puerto 5432.

Listado 3.31: Fichero `user-data.sampleapp.aws`

```

write_files:
  -
    - path: "/home/core/postgresql.service"
      content: |
        [Service]
        ExecStartPost=/bin/bash -c 'while ! \
          [ $(/usr/bin/docker inspect -f ="{{.State.Running}}" some-postgres) \
          == "true" ]; do sleep 1; done; \
          /usr/bin/etcdctl set /skydns/local/sampleapp/some-postgres \
          "{ \"host\": \"db.sampleapp.local.\" }" ; \
          /usr/bin/etcdctl set /skydns/local/sampleapp/db \
          "{ \"host\": \"$(/usr/bin/docker inspect -f \
          \"{{ .NetworkSettings.IPAddress }}\" some-postgres)\" , \"port\":5432}"'
  .

```

En los contenedores `app-job` y `app-task` habrá que eliminar la opción `--link` y utilizar

las opciones `--dns` que apuntará a la dirección IP del contenedor `skydns` y `--dns-search` donde se especifica el dominio trabajado `sampleapp.local` para poder resolver por `db` y `some-postgres` sin especificar el dominio. La sección `ExecStart` de ambas queda:

Listado 3.32: Fichero `user-data.sampleapp.aws`

```
write_files:
  - path: "/home/core/app-job.service"
    [Service]
    ExecStart=/bin/bash -c 'usr/bin/docker run --rm --name app-job \
    -v "volume-public:/usr/src/app/public" --entrypoint "./setup.sh" \
    -e "POSTGRES_USER=${POSTGRES_USER}" \
    -e "POSTGRES_PASSWORD=${POSTGRES_PASSWORD}" \
    -w "/usr/src/app" --dns $(/usr/bin/docker inspect \
    -f "{{ .NetworkSettings.IPAddress }}" skydns) \
    --dns-search "sampleapp.local" carolina/sample_app_rails_4_image:latest'
  - path: "/home/core/app-task.service"
    [Service]
    ExecStart=/bin/bash -c 'usr/bin/docker run --rm --name app-task \
    -e "POSTGRES_USER=${POSTGRES_USER}" \
    -e "POSTGRES_PASSWORD=${POSTGRES_PASSWORD}" \
    -w "/usr/src/app" -v "volume-public:/usr/src/app/public" \
    --dns $(/usr/bin/docker inspect \
    -f "{{ .NetworkSettings.IPAddress }}" skydns) \
    --dns-search "sampleapp.local" carolina/sample_app_rails_4_image:latest \
    /bin/bash -c "cp config/database.yml.postgresql config/database.yml && \
    cp ./secret.example ./secret && puma -p 9292"'
.
```

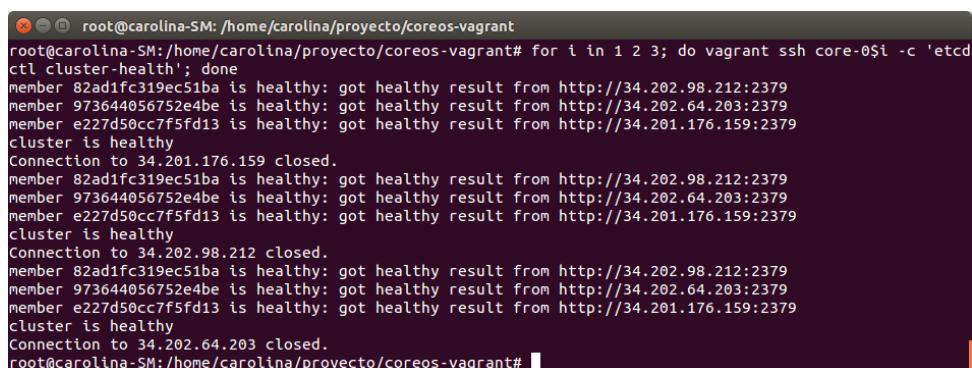
Además habrá que quitar de las secciones `After` y `Requires` el servicio `postgresql.service` en todas las unidades y añadir `skydns.service` antes de `volume-public.service`.

A continuación se despliega la infraestructura:

```
$ vagrant up --provider=aws
```

Primero se comprueba la salud del clúster, contemplada por cada máquina:

```
$ for i in 1 2 3; do vagrant ssh core-0$i -c 'etcdctl cluster-health'; done
```



```
root@carolina-SM: /home/carolina/proyecto/coreos-vagrant
root@carolina-SM: /home/carolina/proyecto/coreos-vagrant# for i in 1 2 3; do vagrant ssh core-0$i -c 'etcdctl cluster-health'; done
member 82ad1fc319ec51ba is healthy: got healthy result from http://34.202.98.212:2379
member 973644056752e4be is healthy: got healthy result from http://34.202.64.203:2379
member e227d50cc7f5fd13 is healthy: got healthy result from http://34.201.176.159:2379
cluster is healthy
Connection to 34.201.176.159 closed.
member 82ad1fc319ec51ba is healthy: got healthy result from http://34.202.98.212:2379
member 973644056752e4be is healthy: got healthy result from http://34.202.64.203:2379
member e227d50cc7f5fd13 is healthy: got healthy result from http://34.201.176.159:2379
cluster is healthy
Connection to 34.202.98.212 closed.
member 82ad1fc319ec51ba is healthy: got healthy result from http://34.202.98.212:2379
member 973644056752e4be is healthy: got healthy result from http://34.202.64.203:2379
member e227d50cc7f5fd13 is healthy: got healthy result from http://34.201.176.159:2379
cluster is healthy
Connection to 34.202.64.203 closed.
root@carolina-SM: /home/carolina/proyecto/coreos-vagrant#
```

Figura 3.51: Salud del clúster contemplada por las 3 máquinas.

Luego se listan las máquinas y se comprueba que han recibido el metadato:

```
for i in 1 2 3; do vagrant ssh core-0$i -c 'fleetctl list-machines'; done
```

```
root@carolina-SM:/home/carolina/proyecto/coreos-vagrant
root@carolina-SM:/home/carolina/proyecto/coreos-vagrant# for i in 1 2 3; do vagrant ssh core-0$i -c 'fleetctl list-machines'; done
MACHINE          IP                  METADATA
5bb48749...     34.202.64.203    -
b36b7400...     34.202.98.212   compute=db
f3cf13ee...     34.201.176.159  compute=web
Connection to 34.201.176.159 closed.
MACHINE          IP                  METADATA
5bb48749...     34.202.64.203    -
b36b7400...     34.202.98.212   compute=db
f3cf13ee...     34.201.176.159  compute=web
Connection to 34.202.98.212 closed.
MACHINE          IP                  METADATA
5bb48749...     34.202.64.203    -
b36b7400...     34.202.98.212   compute=db
f3cf13ee...     34.201.176.159  compute=web
Connection to 34.202.64.203 closed.
root@carolina-SM:/home/carolina/proyecto/coreos-vagrant#
```

Figura 3.52: Información de las máquinas contemplada por las 3 máquinas.

También se comprueba que las unidades de servicio se crearon donde se indicó:

```
$ for i in 1 2 3; do vagrant ssh core-0$i -c 'fleetctl list-units'; done
```

```
root@carolina-SM:/home/carolina/proyecto/coreos-vagrant
root@carolina-SM:/home/carolina/proyecto/coreos-vagrant# for i in 1 2 3; do vagrant ssh core-0$i -c 'fleetctl list-units'; done
UNIT            MACHINE          ACTIVE   SUB
app-job.service f3cf13ee.../34.201.176.159  inactive  dead
app-task.service f3cf13ee.../34.201.176.159  active   running
nginx.service   f3cf13ee.../34.201.176.159  active   running
postgresql.service b36b7400.../34.202.98.212  active   running
skydns.service  5bb48749.../34.202.64.203  active   running
skydns.service  b36b7400.../34.202.98.212  active   running
skydns.service  f3cf13ee.../34.201.176.159  active   running
volume-public.service f3cf13ee.../34.201.176.159  inactive  dead
Connection to 34.201.176.159 closed.
UNIT            MACHINE          ACTIVE   SUB
app-job.service f3cf13ee.../34.201.176.159  inactive  dead
app-task.service f3cf13ee.../34.201.176.159  active   running
nginx.service   f3cf13ee.../34.201.176.159  active   running
postgresql.service b36b7400.../34.202.98.212  active   running
skydns.service  5bb48749.../34.202.64.203  active   running
skydns.service  b36b7400.../34.202.98.212  active   running
skydns.service  f3cf13ee.../34.201.176.159  active   running
volume-public.service f3cf13ee.../34.201.176.159  inactive  dead
Connection to 34.202.98.212 closed.
UNIT            MACHINE          ACTIVE   SUB
app-job.service f3cf13ee.../34.201.176.159  inactive  dead
app-task.service f3cf13ee.../34.201.176.159  active   running
nginx.service   f3cf13ee.../34.201.176.159  active   running
postgresql.service b36b7400.../34.202.98.212  active   running
skydns.service  5bb48749.../34.202.64.203  active   running
skydns.service  b36b7400.../34.202.98.212  active   running
skydns.service  f3cf13ee.../34.201.176.159  active   running
volume-public.service f3cf13ee.../34.201.176.159  inactive  dead
Connection to 34.202.64.203 closed.
root@carolina-SM:/home/carolina/proyecto/coreos-vagrant#
```

Figura 3.53: Información de las unidades contemplada por las 3 máquinas.

Se comprueba que los registros se hayan establecido correctamente:

```
$ for i in 1 2 3; do vagrant ssh core-0$i -c '/usr/bin/etcddctl get \
/skydns/local/sampleapp/db; \
/usr/bin/etcddctl get /skydns/local/sampleapp/some-postgres'; done
```

```
root@carolina-SM:/home/carolina/proyecto/coreos-vagrant
root@carolina-SM:/home/carolina/proyecto/coreos-vagrant# for i in 1 2 3; do vagrant ssh core-0$i -c '/usr/bin/etcctl get /skydns/local/sampleapp/db; /usr/bin/etcctl get /skydns/local/sampleapp/some-postgres'; done
{ "host": "172.17.96.3" , "port":5432}
{ "host": "db.sampleapp.local."}
Connection to 34.201.176.159 closed.
{ "host": "172.17.96.3" , "port":5432}
{ "host": "db.sampleapp.local."}
Connection to 34.202.98.212 closed.
{ "host": "172.17.96.3" , "port":5432}
{ "host": "db.sampleapp.local."}
Connection to 34.202.64.203 closed.
root@carolina-SM:/home/carolina/proyecto/coreos-vagrant#
```

Figura 3.54: Lectura de los registros en SkyDNS contemplada por las 3 máquinas.

La máquina **core-01** resuelve tanto *db* como *some-postgres*:

```
# docker exec -it app-task sh -c 'ping -c 1 db'
# docker exec -it app-task sh -c 'ping -c 1 some-postgres'
```

```
core@ip-10-0-0-101:~
core@ip-10-0-0-101 ~ $ docker exec -it app-task sh -c 'ping -c 1 db'
PING db.sampleapp.local (172.17.96.3): 56 data bytes
64 bytes from 172.17.96.3: icmp_seq=0 ttl=60 time=0.693 ms
--- db.sampleapp.local ping statistics ---
1 packets transmitted, 1 packets received, 0% packet loss
round-trip min/avg/max/stddev = 0.693/0.693/0.693/0.000 ms
core@ip-10-0-0-101 ~ $ docker exec -it app-task sh -c 'ping -c 1 some-postgres'
PING db.sampleapp.local (172.17.96.3): 56 data bytes
64 bytes from 172.17.96.3: icmp_seq=0 ttl=60 time=0.660 ms
--- db.sampleapp.local ping statistics ---
1 packets transmitted, 1 packets received, 0% packet loss
round-trip min/avg/max/stddev = 0.660/0.660/0.660/0.000 ms
core@ip-10-0-0-101 ~ $
```

Figura 3.55: Conexión a *db* como *some-postgres* desde **app-task**.

De esta manera, los registros ingresados han quedado como se pretendía:

```
# docker exec -it skydns sh -c 'dig @localhost db.sampleapp.local. A'
```

```
core@ip-10-0-0-101:~
core@ip-10-0-0-101 ~ $ docker exec -it skydns sh -c 'dig @localhost some-postgres.sampleapp.local. A'

; <>> DiG 9.10.2-P3 <>> @localhost some-postgres.sampleapp.local. A
; (2 servers found)
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 17095
;; flags: qr aa rd ra; QUERY: 1, ANSWER: 2, AUTHORITY: 0, ADDITIONAL: 0

;; QUESTION SECTION:
;some-postgres.sampleapp.local. IN      A

;; ANSWER SECTION:
db.sampleapp.local.    30      IN      A      172.17.96.3
some-postgres.sampleapp.local. 30 IN      CNAME   db.sampleapp.local.

;; Query time: 3 msec
;; SERVER: ::1#53(::1)
;; WHEN: Mon Apr 24 17:37:16 UTC 2017
;; MSG SIZE  rcvd: 80
core@ip-10-0-0-101 ~ $
```

Figura 3.56: Registros DNS desde el contenedor **skydns**.

Así, el servicio funciona correctamente en la máquina **core-01**:

```
$ for i in 1 2 3; do vagrant ssh core-0$i -c 'curl http://localhost:80 | \ tail -n 15'; done
```

```
core@ip-10-0-0-101:~$ curl http://localhost:80 | tail -n 15
% Total    % Received % Xferd  Average Speed   Time     Time      Current
          Dload  Upload Total Spent   Left Speed
100 3983     0  3983     0      0  9507      0 --:--:-- --:--:-- 9528
<nav>
  <ul>
    <li><a href="/about">About</a></li>
    <li><a href="/contact">Contact</a></li>
    <li><a href="http://news.railstutorial.org/">News</a></li>
  </ul>
</nav>
</footer>
<pre class="debug_dump">--- !ruby/hash:ActionController::Parameters
controller: static_pages
action: home
id: '100'
</pre>
</div>
</body>
</html>core@ip-10-0-0-101 ~ $
```

Figura 3.57: Conexión al servicio mediante consola desde **core-01**.

Y mediante la web y la dirección IP elástica se ingresa con uno de los usuarios poblados en la base de datos, comprobando que sigue funcionando correctamente hasta este punto:

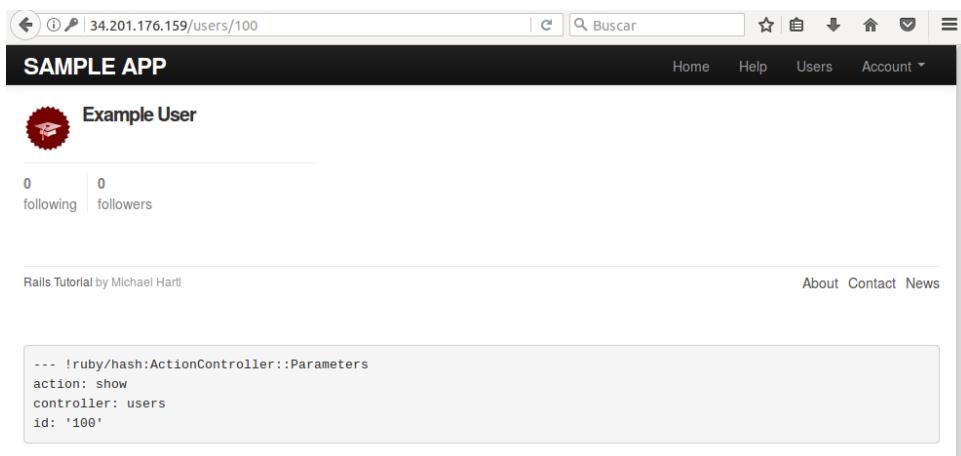


Figura 3.58: Inicio de sesión con el usuario *Example User* desde **core-01**.

3.6.10 Balanceo de carga con Nginx usando Confd

Con el propósito de implementar la infraestructura final se van a realizar una serie de cambios. En la primera máquina se sitúa el proxy, que balanceará la carga entre ellos. En la segunda máquina se encontrará la base de datos a la que acudirán los servidores de aplicación.

Al no encontrarse los distintos contenedores en la misma máquina la opción de tener el volumen Docker **volume-public** compartido entre ellos no es posible en la manera en la que se ha ido haciendo. Así, se elimina su definición en el fichero **user-data.sampleapp.aws**, en **/home/core/volume-public.service**, su inclusión en los parámetros *After* y *Requires* en los servicios que lo implementan, el parámetro *-v* que lo exporta en los contenedores **app-job**, **app-task** y **some-nginx**, así como la unidad global **fleet-volume-public.service**.

El despliegue de la infraestructura será controlado con metadatos. Para ello se mantiene a la máquina **core-02** con el metadato *compute=db* y se modifica en el fichero **Vagrantfile**, bajo la carga de la línea que contiene la sección de configuración fleet en el fichero **user-data**, el metadato para la máquina **core-01**, cambiándolo por *compute=proxy*.

Listado 3.33: Fichero **Vagrantfile**

```
data = YAML.load(IO.readlines(CLOUD_CONFIG_PATH) [1..-1].join)
if data['coreos'].key? 'fleet' and i==1
  data['coreos']['fleet']['metadata'] = 'compute=proxy'
end
.
```

Ahora será necesario modificar el metadato de la unidad de servicio **postgresql.service** a *MachineMetadata=compute=proxy*, en la sección *[X-Fleet]*. Además, se quita en las unidades de servicio **app-job.service** y **app-task.service**, para que sean ubicadas en cada máquina.

Para tener siempre el servidor web **app-task** activo en cada máquina, se prepara para que en caso de fallo vuelva a activarse. Esto se consigue con el parámetro *Restart=always*.

Como el servicio **nginx.service** tendrá que balancear la carga entre los distintos servidores web disponibles, será necesario que el servidor **app-task** se registre tras crearse. Este registro se va a realizar en etc, por lo tanto, éste se añade en los parámetros *After* y *Requires* como **etcd2.service** tras **docker.service**. Así, el registro se añade en la sección *ExecStartPost* con clave **/services/app/<dirección IP de la máquina en la que se encuentra>** y valor correspondiente con la dirección IP del contenedor y puerto 9292, pues se hace referencia al servidor puma. Esto permitirá tener tantas entradas **/services/app/*** como máquinas en las que se ejecute hayan. Es importante añadir que cuando el contenedor deje de ejecutarse dicho registro sea borrado. Para interpretar la dirección IP de la máquina es necesario indicar el fichero de variables de entorno **/etc/environment**.

Listado 3.34: Fichero **user-data.sampleapp.aws**

```
write_files:
  - path: "/home/core/app-task.service"
    [Service]
    .
    Restart=always
    EnvironmentFile=/etc/environment
    .
    ExecStartPost=/bin/bash -c 'while ! [ $(/usr/bin/docker inspect \
      -f="{{.State.Running}}"}" app-task) == "=true" ]; do sleep 1; done; \
      etcdctl set /services/app/${COREOS_PRIVATE_IPV4} ${/usr/bin/docker \
      inspect -f "{{.NetworkSettings.IPAddress }}"} app-task:9292'
    ExecStop=/usr/bin/etcctl rm /services/app/${COREOS_PRIVATE_IPV4}
    .

```

Para que el servicio **nginx.service** pueda registrar los servidores web puma se va a utilizar el servicio confd. Esta utilidad se encarga de controlar los cambios que se produzcan en etc, vigilando las claves que se le indique. Cuando hay un nuevo registro o baja transfiere estos cambios a la configuración de nginx, por medio de una plantilla, modificando el fichero

`/etc/nginx.conf` del contenedor `some-nginx`. Luego, hace efectivo el cambio reiniciando el servicio nginx. Este servicio puede ser encontrado en el repositorio `kelseyhightower/confd`, en GitHub, del que se hará una configuración específica.

Como se pretende compartir el fichero de configuración `nginx.conf` entre el contenedor `some-nginx` y el futuro contenedor `confd` se crea un volumen Docker, llamado `conf-data`, que compartirá el directorio `/etc/nginx` entre ambos. En su definición se indica requiere y ha de ser lanzado después del servicio `docker.service`. Se indica que el servicio será de tipo `oneshot`, lo que significa que el parámetro `ExecStart` ha de ejecutarse solo una vez, sin reintentarlo. También se especifica la opción `RemainAfterExit=yes` para que cuando el contenedor termine permanezca activo. Si el volumen existe en su lanzamiento se detiene y elimina. La imagen que se usa para el contenedor es la de nginx, para poder tener la misma disposición de directorios. Este volumen deberá ubicarse en la máquina `core-01`, por lo que se le añade el metadato `compute=proxy`. Además, será necesario crear la unidad global `fleet-conf-data.service` que de comienzo a este servicio fleet.

Listado 3.35: Fichero `user-data.sampleapp.aws`

```

write_files:
  - path: "/home/core/conf-data.service"
    permissions: "0644"
    content: |
      [Unit]
      Description=conf data container to share files between confd and some-nginx
      After=docker.service
      Requires=docker.service
      [Service]
      Type=oneshot
      RemainAfterExit=yes
      ExecStartPre=/usr/bin/docker kill conf-data
      ExecStartPre=/usr/bin/docker rm conf-data
      ExecStartPre=/usr/bin/docker pull nginx
      ExecStart=/usr/bin/docker run -v /etc/nginx --name conf-data nginx \
      echo "conf-data container created"
      [X-Fleet]
      Global=true
      MachineMetadata=compute=proxy

coreos:
  units:
    - name: fleet-conf-data.service
      command: start
      content: |
        [Unit]
        Description=Start conf-data.service using fleet
        [Service]
        ExecStart=/usr/bin/fleetctl start /home/core/conf-data.service

```

El servicio confd necesitará disponer de tres ficheros, que serán escritos en la máquina y exportados al contenedor a través del volumen `conf-data`.

El primero de ellos es la plantilla `nginx.conf.tpl`, correspondiente a la configuración de nginx. Al incorporarlo a `user-data.sampleapp.app` puede borrarse la definición del fichero `/etc/nginx.conf`, presente en la sección `write-files`. La información a añadir es el bloque `upstream` utilizado para definir los servidores web. Nginx seleccionará uno de ellos dependiendo del método de distribución escogido, en este caso *Round Robin*, del primero al último, que por defecto es el que se implementa. Mediante una plantilla en formato *Go*, que

permite gestionar contenido dinámico, se define este nuevo bloque, que ha de estar, a su vez, dentro del bloque *http* y requiere la existencia del bloque *events* para gestionar eventos, aunque no sea utilizado. Así, cuando confd analice etc en busca de cambios para las claves `/services/app/*` sustituirá dinámicamente su contenido por las líneas *server <dirección IP del contenedor app-task>:9292*, en lugar de tener que definirlos estáticamente. El servidor escogido pasará a *proxy_pass http://app;*.

Listado 3.36: Fichero `user-data.sampleapp.aws`

```

.
write_files:
.
- path: "/etc/nginx.conf.tpl"
  permissions: "0644"
  content: |
    events {
    }
    http {
      upstream app {
        {{ range getvs "/services/app/*" }}
        server {{ . }};
        {{ end }}
      }
      server {
        listen 80;
        root /usr/src/app/public;
        location / {
          proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
          proxy_set_header Host $http_host;
          proxy_redirect off;
          try_files $uri /page_cache/$uri /page_cache/$uri.html @app;
        }
        location @app{
          proxy_pass http://app;
          break;
        }
      }
    }
.

```

El siguiente, llamado `nginx.toml`, representa el fichero de configuración confd. Está escrito en formato TOML, comprueba el valor de las claves registradas bajo `/services/app` y realiza la acción del cambio del archivo `/etc/nginx/nginx.conf`, tomando como fuente la plantilla anterior, `nginx.conf.tpl`. El servicio confd ha de encargarse de reiniciar el servicio Nginx. Para ello ejecuta el fichero `/reload.sh` que permitirá ejecutar una orden mediante el comando curl, haciendo uso del socket de Docker, puesto que éste no estará instalado en el contenedor `confd`. Esta orden hará que capture el identificador del contenedor `some-nginx`, a través del lenguaje JSON `jq`, y detenga el proceso.

Listado 3.37: Fichero `user-data.sampleapp.aws`

```

.
write_files:
.
- path: "/home/core/reload.sh"
  permissions: "0644"
  content: |
    #!/bin/sh
    curl --unix-socket /var/run/docker.sock -XPOST \
    "http://v1.26/containers/$(`curl --unix-socket /var/run/docker.sock \
    'http://v1.26/containers/json' | jq -r '.[] | \
    select(.Names[0] == "/some-nginx") .Id')/kill"

```

```

- path: "/etc/nginx.toml"
  permissions: "0644"
  content: |
    [template]
    src = "nginx.conf.tpl"
    dest = "/etc/nginx/nginx.conf"
    keys = [ "/services/app" ]
    reload_cmd = "sh /reload.sh"
.
.
```

Para disponer de una imagen Docker propia de confd se crea una nueva, llamada **confd_image**, a partir del fichero Dockerfile. Se utiliza una imagen *Alpine Linux* con un índice de paquetes completo y solo 5 MB de tamaño, en su versión 3.3. Sobre esta imagen se instalarán los paquetes *curl*, para hacer la recarga del servicio, y el *jq*, para obtener el identificador del contenedor **some-nginx**. Luego se obtiene la herramienta confd para sistemas Linux con CPUs AMD de 64 bits, se le dan permisos de lectura, escritura y ejecución al propietario en **/usr/bin/confd** para poder realizar operaciones y se limpia la memoria caché utilizada. Además, se añade el directorio vacío **/etc/confd** que será utilizado para alojar los ficheros. Por último, se indica que una vez arrancado un contenedor, el servicio confd vigilará los cambios producidos en etc, en todo el clúster, cada 5 segundos.

Listado 3.38: Contenido de **Dockerfile**

```

FROM alpine:3.3
LABEL confd_image.version="0.1" confd_image.release-date="2017-05-02"
MAINTAINER Carolina Santana "c.santanamartel@gmail.com"
RUN apk add --update curl && apk add --update jq && \
    curl -o /usr/bin/confd -L https://github.com/kelseyhightower/confd/releases/
    download/v0.7.1/confd-0.7.1-linux-amd64 && \
    chmod 755 /usr/bin/confd && rm -rf /var/cache/apk/*
ADD etc/confd/ /etc/confd
CMD /usr/bin/confd -interval=5 -node=http://$COREOS_PRIVATE_IPV4:4001
```

De esta manera, será necesario crear en local el directorio vacío **etc/confd**, construir la imagen y subirla a Docker Hub para poder usarla:

```

$ mkdir etc/confd
$ docker login
$ docker build -t confd_image .
$ docker tag <image-id> carolina/confd_image:latest
```

Cuando la subida termina, se define el fichero **/home/core/confd.service** que creará el contenedor **confd**. Así, requiere y ha de ser creado tras los servicios **docker.service**, **etcd2.service**, pues leerá de él, y **conf-data.service**. Se indica que comience la ejecución de *ExecStart* y se añade el fichero de variables de entorno **/etc/environment** para que pueda conocer las direcciones IP de las máquinas que conforman el clúster. Si en el momento de ser lanzado ya existiera se manda a detener el proceso y se elimina el contenedor. Antes de crearlo se descarga la última versión de la imagen **confd_image**, si no se tiene. Además de pasarle la variable de entorno comentada, se exporta el *socket* de Docker, para poder usarlo a la hora de ejecutar la recarga del servicio nginx y los tres archivos escritos, donde los destinos finales serán **/etc/confd/templates/nginx.conf.tpl**, **/reload.sh** y **/etc/confd/conf.d/nginx.toml**. Por último se añade el volumen **conf-data**. Este contenedor deberá ubicarse en la máquina **core-01**, por lo que se le añade el metadato **compute=proxy**. También se crea la unidad global **fleet-confd.service** que comienza este servicio.

Listado 3.39: Fichero `user-data.sampleapp.aws`

```

write_files:
  - path: "/home/core/confd.service"
    permissions: "0644"
    content: |
      [Unit]
      Description=confd container that updates nginx.conf file and kill
                  some-nginx container to restart nginx service when detects
                  a new rails server has been registered
      After=docker.service etcd2.service conf-data.service
      Requires=docker.service etcd2.service conf-data.service
      [Service]
      TimeoutStartSec=0
      EnvironmentFile=/etc/environment
      ExecStartPre=/usr/bin/docker kill confd
      ExecStartPre=/usr/bin/docker rm confd
      ExecStartPre=/usr/bin/docker pull carolina/confd_image:latest
      ExecStart=/bin/bash -c '/usr/bin/docker run --rm --name confd \
      -e COREOS_PRIVATE_IPV4=${COREOS_PRIVATE_IPV4} \
      -v "/var/run/docker.sock:/var/run/docker.sock" \
      -v "/etc/nginx.conf.tpl:/etc/confd/templates/nginx.conf.tpl" \
      -v "/home/core/reload.sh:/reload.sh" \
      -v "/etc/nginx.toml:/etc/confd/conf.d/nginx.toml" \
      --volumes-from=conf-data carolina/confd_image'
      ExecStop=/usr/bin/docker stop confd
      [X-Fleet]
      Global=true
      MachineMetadata=compute=proxy

coreos:
  units:
    - name: fleet-confd.service
      command: start
      content: |
        [Unit]
        Description=Start confd.service using fleet
        [Service]
        ExecStart=/usr/bin/fleetctl start /home/core/confd.service

```

En último lugar se modifica el fichero `/home/core/nginx.service` de manera que se especifique en las opciones *After* y *Requires* que requiere y ha de ir tras los servicios `docker.service`, `conf-data.service` y `confd.service`. Como el servicio confd se encargará de detener el proceso de `some-nginx` se añade la opción *Restart=always* que hará que la unidad de servicio vuelva a ejecutarse con la nueva configuración Nginx. Además, en la creación del contenedor, *ExecStart*, se elimina la opción `-link app-task:app` y el fichero de configuración que se exportaba anteriormente `-v /etc/nginx.conf:/etc/nginx/nginx.conf` y se añade la opción `-volumes-from=conf-data`.

A continuación se despliega la infraestructura:

```
$ . ~/.aws-credentials/aws-credentials && vagrant up --provider=aws
```

En primer lugar se comprueba la salud del clúster:

```
$ for i in 1 2 3; do vagrant ssh core-0$i -c 'etcdctl cluster-health'; done
```

```
root@carolina-SM: /home/carolina/proyecto/coreos-vagrant
root@carolina-SM:/home/carolina/proyecto/coreos-vagrant# for i in 1 2 3; do vagrant ssh core-0${i} -c 'etcdctl cluster-health'; done
member 9c5da0ea15430c72 is healthy: got healthy result from http://52.20.71.84:2379
member c9b273b8262f4917 is healthy: got healthy result from http://34.205.83.173:2379
member d52ee481bc450646 is healthy: got healthy result from http://52.44.190.201:2379
cluster is healthy
Connection to 52.20.71.84 closed.
member 9c5da0ea15430c72 is healthy: got healthy result from http://52.20.71.84:2379
member c9b273b8262f4917 is healthy: got healthy result from http://34.205.83.173:2379
member d52ee481bc450646 is healthy: got healthy result from http://52.44.190.201:2379
cluster is healthy
Connection to 34.205.83.173 closed.
member 9c5da0ea15430c72 is healthy: got healthy result from http://52.20.71.84:2379
member c9b273b8262f4917 is healthy: got healthy result from http://34.205.83.173:2379
member d52ee481bc450646 is healthy: got healthy result from http://52.44.190.201:2379
cluster is healthy
Connection to 52.44.190.201 closed.
root@carolina-SM:/home/carolina/proyecto/coreos-vagrant#
```

Figura 3.59: Salud del clúster contemplada por las 3 máquinas.

También, se visualizan las máquinas que conforman el clúster, con los metadatos asignados y las unidades fleet presentes en cada máquina:

```
$ for i in 1 2 3; do vagrant ssh core-0${i} -c 'fleetctl list-machines'; done
```

```
root@carolina-SM: /home/carolina/proyecto/coreos-vagrant
root@carolina-SM:/home/carolina/proyecto/coreos-vagrant# for i in 1 2 3; do vagrant ssh core-0${i} -c 'fleetctl list-machines'; done
MACHINE      IP          METADATA
3551810b...   52.20.71.84    compute=proxy
54d18cde...   34.205.83.173  compute=db
b07cef85...   52.44.190.201 -
Connection to 52.20.71.84 closed.
MACHINE      IP          METADATA
3551810b...   52.20.71.84    compute=proxy
54d18cde...   34.205.83.173  compute=db
b07cef85...   52.44.190.201 -
Connection to 34.205.83.173 closed.
MACHINE      IP          METADATA
3551810b...   52.20.71.84    compute=proxy
54d18cde...   34.205.83.173  compute=db
b07cef85...   52.44.190.201 -
Connection to 52.44.190.201 closed.
root@carolina-SM:/home/carolina/proyecto/coreos-vagrant#
```

Figura 3.60: Información de las máquinas contemplada por las 3 máquinas.

```
$ for i in 1 2 3; do vagrant ssh core-0${i} -c 'fleetctl list-units'; done
```

```
root@carolina-SM:/home/carolina/proyecto/coreos-vagrant
root@carolina-SM:/home/carolina/proyecto/coreos-vagrant# for i in 1 2 3; do vagrant ssh core-0${i} -c 'fleetctl list-units'; done
UNIT           MACHINE          ACTIVE SUB
app-job.service 3551810b.../52.20.71.84    inactive dead
app-job.service 54d18cde.../34.205.83.173   inactive dead
app-job.service b07cef85.../52.44.190.201  inactive dead
app-task.service 3551810b.../52.20.71.84    active  running
app-task.service 54d18cde.../34.205.83.173   active  running
app-task.service b07cef85.../52.44.190.201  active  running
confd.service    3551810b.../52.20.71.84    active  exited
confd.service    3551810b.../52.20.71.84    active  running
nginx.service    3551810b.../52.20.71.84    active  running
postgresql.service 54d18cde.../34.205.83.173  active  running
skydns.service   3551810b.../52.20.71.84    active  running
skydns.service   54d18cde.../34.205.83.173   active  running
skydns.service   b07cef85.../52.44.190.201  active  running
Connection to 52.20.71.84 closed.
UNIT           MACHINE          ACTIVE SUB
app-job.service 3551810b.../52.20.71.84    inactive dead
app-job.service 54d18cde.../34.205.83.173   inactive dead
app-job.service b07cef85.../52.44.190.201  inactive dead
app-task.service 3551810b.../52.20.71.84    active  running
app-task.service 54d18cde.../34.205.83.173   active  running
app-task.service b07cef85.../52.44.190.201  active  running
confd.service    3551810b.../52.20.71.84    active  exited
confd.service    3551810b.../52.20.71.84    active  running
nginx.service    3551810b.../52.20.71.84    active  running
postgresql.service 54d18cde.../34.205.83.173  active  running
skydns.service   3551810b.../52.20.71.84    active  running
skydns.service   54d18cde.../34.205.83.173   active  running
skydns.service   b07cef85.../52.44.190.201  active  running
Connection to 34.205.83.173 closed.
UNIT           MACHINE          ACTIVE SUB
app-job.service 3551810b.../52.20.71.84    inactive dead
app-job.service 54d18cde.../34.205.83.173   inactive dead
app-job.service b07cef85.../52.44.190.201  inactive dead
app-task.service 3551810b.../52.20.71.84    active  running
app-task.service 54d18cde.../34.205.83.173   active  running
app-task.service b07cef85.../52.44.190.201  active  running
confd.service    3551810b.../52.20.71.84    active  exited
confd.service    3551810b.../52.20.71.84    active  running
nginx.service    3551810b.../52.20.71.84    active  running
postgresql.service 54d18cde.../34.205.83.173  active  running
skydns.service   3551810b.../52.20.71.84    active  running
skydns.service   54d18cde.../34.205.83.173   active  running
skydns.service   b07cef85.../52.44.190.201  active  running
Connection to 52.44.190.201 closed.
root@carolina-SM:/home/carolina/proyecto/coreos-vagrant#
```

Figura 3.61: Información de las unidades fleet contemplada por las 3 máquinas.

Para continuar con las pruebas se escoge la máquina **core-01**.

Se comprueban los registros existentes bajo la clave **services/app/<dirección IP de app-task>**. Las claves se averiguan con el comando **etcdctl ls --recursive /services/app** y luego los valores con el comando **etcdctl get /services/app/<dirección IP de app-task>**:

```
core@ip-10-0-0-101:~$ etcdctl ls --recursive /services/app
/services/app/10.0.0.103
/services/app/10.0.0.102
/services/app/10.0.0.101
core@ip-10-0-0-101:~$ etcdctl get /services/app/10.0.0.103
172.17.16.4:9292
core@ip-10-0-0-101:~$ etcdctl get /services/app/10.0.0.102
172.17.8.5:9292
core@ip-10-0-0-101:~$ etcdctl get /services/app/10.0.0.101
172.17.2.6:9292
core@ip-10-0-0-101:~$
```

Figura 3.62: Obtención de claves y valores para **services/app/***.

Ahora que los contenedores **app-task** se han registrado se comprueba que el servicio confd ha actualizado el fichero de configuración Nginx, a medida que los ha ido detectando. Esto es observado desde los registros de operación del servicio **confd.service** y dentro del contenedor **some-nginx**, viendo el contenido del fichero **/etc/nginx/nginx.conf**:

```
# journalctl -u confd.service
```

```
INFO /etc/nginx/nginx.conf has md5sum f7984934bd6cab883e1f33d5129834bb should be 5c19ce03dc804d
INFO Target config /etc/nginx/nginx.conf out of sync
INFO Target config /etc/nginx/nginx.conf has been updated
INFO /etc/nginx/nginx.conf has md5sum 5c19ce03dc804d21a45a71ed5dbfa5ee should be e71664b5f218b4
INFO Target config /etc/nginx/nginx.conf out of sync
INFO Target config /etc/nginx/nginx.conf has been updated
INFO /etc/nginx/nginx.conf has md5sum e71664b5f218b4d74b40ba56f9d7e532 should be 2c5fb0b7ca5d12
INFO Target config /etc/nginx/nginx.conf out of sync
INFO Target config /etc/nginx/nginx.conf has been updated
```

Figura 3.63: Actualización de la configuración Nginx desde **confd**.

```
# docker exec -it some-nginx sh
# cat /etc/nginx/nginx.conf
```

```
core@ip-10-0-0-101:~ core@ip-10-0-0-101:~ $ docker exec -it some-nginx sh
# cat /etc/nginx/nginx.conf
events {
}
http {
    upstream app {
        server 172.17.16.4:9292;
        server 172.17.2.6:9292;
        server 172.17.8.5:9292;
    }
    server {
        listen 80;
        root /usr/src/app/public;
        location / {
            proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
            proxy_set_header Host $http_host;
            proxy_redirect off;
            try_files $uri /page_cache/$uri /page_cache/$uri.html @app;
        }
        location @app{
            proxy_pass http://app;
            break;
        }
    }
}
#
```

Figura 3.64: Fichero **nginx.conf** actualizado.

Tras ver el orden en el que se han colocado las distintas direcciones IP de los contenedores **app-task** puede conocerse de qué máquina proviene cada uno inspeccionando el contenedor en cada máquina:

```
$ docker inspect app-task -f "{{ .NetworkSettings.IPAddress }}"
```

```
core@ip-10-0-0-101:~ core@ip-10-0-0-101:~ $ docker inspect app-task -f "{{ .NetworkSettings.IPAddress }}"
172.17.2.6
core@ip-10-0-0-101 ~ $
```

Figura 3.65: Dirección IP del contenedor **app-task** en la máquina **core-01**.

```
core@ip-10-0-0-102:~$ docker inspect app-task -f "{{ .NetworkSettings.IPAddress }}"
17.2.17.8.5
core@ip-10-0-0-102 ~ $
```

Figura 3.66: Dirección IP del contenedor **app-task** en la máquina **core-02**.

```
core@ip-10-0-0-103:~$ docker inspect app-task -f "{{ .NetworkSettings.IPAddress }}"
17.2.17.16.4
core@ip-10-0-0-103 ~ $
```

Figura 3.67: Dirección IP del contenedor **app-task** en la máquina **core-03**.

Por lo tanto el orden establecido ha sido: **core-03**, **core-01** y **core-02**.

Así mismo, si se realizan 3 peticiones de servicio prácticamente juntas, desde la máquina **core-01** en el que se encuentra el proxy, se puede comprobar por la hora marcada como las peticiones se envían en dicho orden.

```
$ for i in 1 2 3; do curl http://localhost:80 | tail -n 15; done
```

Para comprobar cuándo fueron resueltas las peticiones se ejecuta:

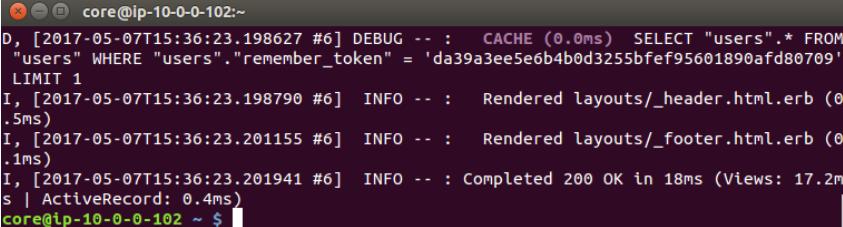
```
$ docker logs app-task
```

```
core@ip-10-0-0-103:~$ docker logs app-task
D, [2017-05-07T15:36:03.031063 #6] DEBUG -- :   CACHE (0.0ms)  SELECT "users".* FROM "users" WHERE "users"."remember_token" = 'da39a3ee5e6b4b0d3255bfef95601890afdb80709' LIMIT 1
I, [2017-05-07T15:36:03.031245 #6]  INFO -- :   Rendered layouts/_header.html.erb (0.0ms)
I, [2017-05-07T15:36:03.033562 #6]  INFO -- :   Rendered layouts/_footer.html.erb (0.1ms)
I, [2017-05-07T15:36:03.034365 #6]  INFO -- : Completed 200 OK in 18ms (Views: 16.9ms | ActiveRecord: 0.8ms)
core@ip-10-0-0-103 ~ $
```

Figura 3.68: Información de **app-task** sobre una petición en **core-03**.

```
core@ip-10-0-0-101:~$ docker logs app-task
D, [2017-05-07T15:36:11.693117 #6] DEBUG -- :   CACHE (0.0ms)  SELECT "users".* FROM "users" WHERE "users"."remember_token" = 'da39a3ee5e6b4b0d3255bfef95601890afdb80709' LIMIT 1
I, [2017-05-07T15:36:11.693300 #6]  INFO -- :   Rendered layouts/_header.html.erb (0.6ms)
I, [2017-05-07T15:36:11.695825 #6]  INFO -- :   Rendered layouts/_footer.html.erb (0.1ms)
I, [2017-05-07T15:36:11.696627 #6]  INFO -- : Completed 200 OK in 19ms (Views: 17.2ms | ActiveRecord: 0.7ms)
core@ip-10-0-0-101 ~ $
```

Figura 3.69: Información de **app-task** sobre una petición en **core-01**.



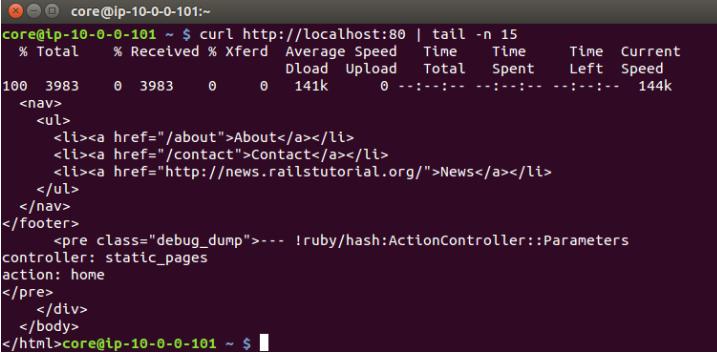
```

core@ip-10-0-0-102:~$ curl http://localhost:3001
D, [2017-05-07T15:36:23.198627 #6] DEBUG -- :   CACHE (0.0ms)  SELECT "users".* FROM "users" WHERE "users"."remember_token" = 'da39a3ee5e6b4b0d3255bfef95601890af80709' LIMIT 1
I, [2017-05-07T15:36:23.198790 #6]  INFO -- :   Rendered layouts/_header.html.erb (0.5ms)
I, [2017-05-07T15:36:23.201155 #6]  INFO -- :   Rendered layouts/_footer.html.erb (0.1ms)
I, [2017-05-07T15:36:23.201941 #6]  INFO -- : Completed 200 OK in 18ms (Views: 17.2ms | ActiveRecord: 0.4ms)
core@ip-10-0-0-102 ~ $ 

```

Figura 3.70: Información de **app-task** sobre una petición en **core-02**.

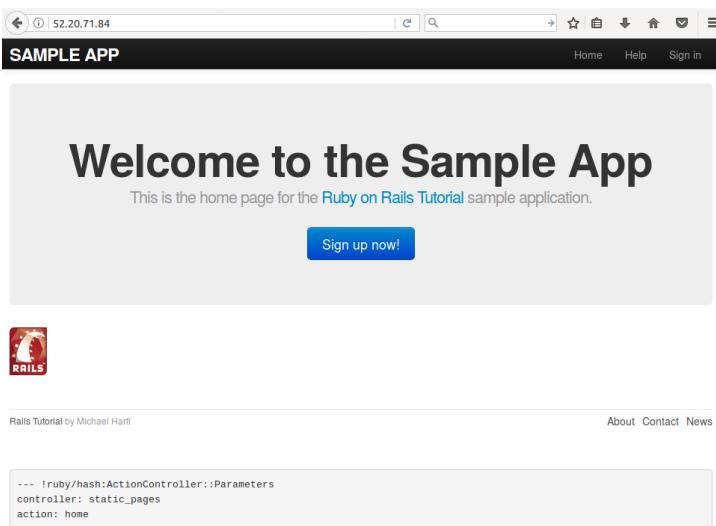
Finalmente se visualiza el servicio, siendo balanceada la carga entre los distintos servidores web de forma transparente al usuario.



```

core@ip-10-0-0-101:~$ curl http://localhost:80 | tail -n 15
% Total    % Received % Xferd  Average Speed   Time      Time     Current
          Dload  Upload Total   Spent    Left Speed
100  3983     0  3983     0      0  141k      0  --:--:--:--:--:-- 144k
<nav>
  <ul>
    <li><a href="/about">About</a></li>
    <li><a href="/contact">Contact</a></li>
    <li><a href="http://news.railstutorial.org/">News</a></li>
  </ul>
</nav>
</body>
</html>core@ip-10-0-0-101 ~ $ 

```

Figura 3.71: Respuesta al servicio por comandos en **core-01**.Figura 3.72: Respuesta al servicio por el navegador web en **core-01**.

Además se inicia sesión con uno de los usuarios con los que se ha alimentado la base de datos para comprobar el correcto funcionamiento de las solicitudes a la base de datos desde esta infraestructura. El usuario a usar es *example@railstutorial.org* y la contraseña *foobar*:

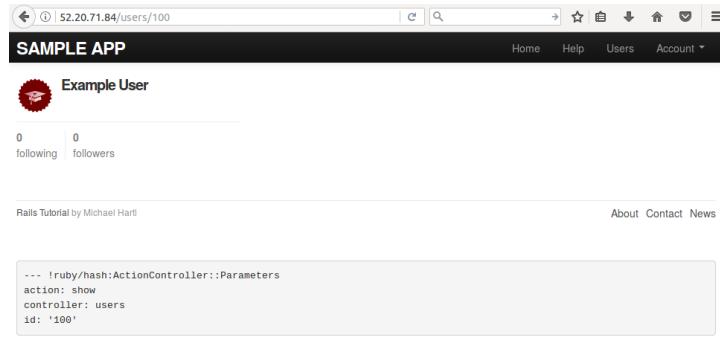


Figura 3.73: Inicio de sesión en el servicio con el usuario *Example User*.

Haciendo una escritura desde el mismo servidor que ha iniciado la sesión:

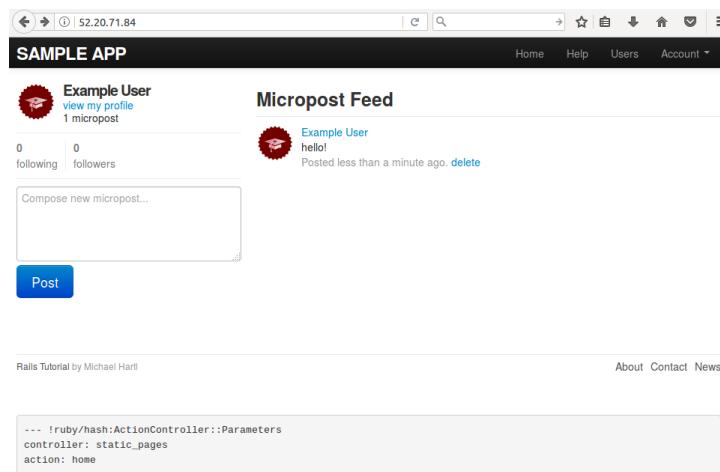


Figura 3.74: Escritura de un comentario con el usuario *Example User*.

La información sobre las instancias puede ser consultada desde la consola de AWS:

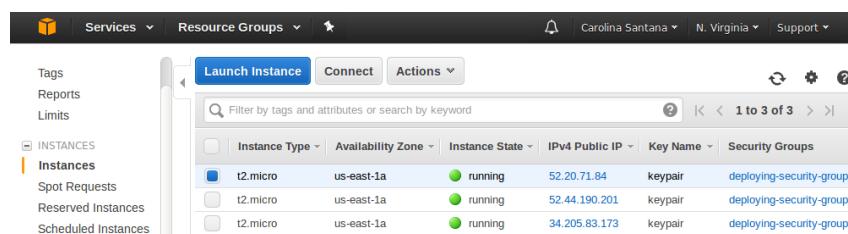


Figura 3.75: Visualización de las instancias desde la consola de AWS.

3.6.11 Resultado

En esta iteración se ha conseguido el despliegue de la aplicación distribuida en un clúster, compuesto de tres instancias, en la nube pública de Amazon Web Services.

De esta manera se posee una infraestructura con servicios web de la aplicación replicados en una o tantas instancias como existan en el clúster, apreciando la característica de redundancia. Además, el proxy dispone de la opción de balanceo de carga con el uso del servicio confd, que controla los nuevos registros o bajas de los servidores web y actualiza el fichero de configuración Nginx para que pueda resolver hacia ellos. Con el clúster constituido

sobre la nube pública de AWS, permitiendo mantener los servicios basados en contenedores, así como las máquinas monitorizadas entre sí, también está presente la alta disponibilidad de servicios. De esta forma, la configuración establecida responde ante el fallo de una unidad o máquina, recargándola para volver a ofrecer correctamente el servicio.

El correcto funcionamiento del servicio web de la aplicación quedará disponible mediante el despliegue de la infraestructura desde Vagrant, estableciendo las variables de entorno del proveedor. La manera de proceder es accediendo directamente por el proxy, como se puede ver en la [Figura 3.72](#), en la máquina **core-01**:

```
$ . ~/aws-credentials/aws-credentials
$ vagrant up --provider=aws && vagrant ssh core-01
# curl http://localhost:80
```

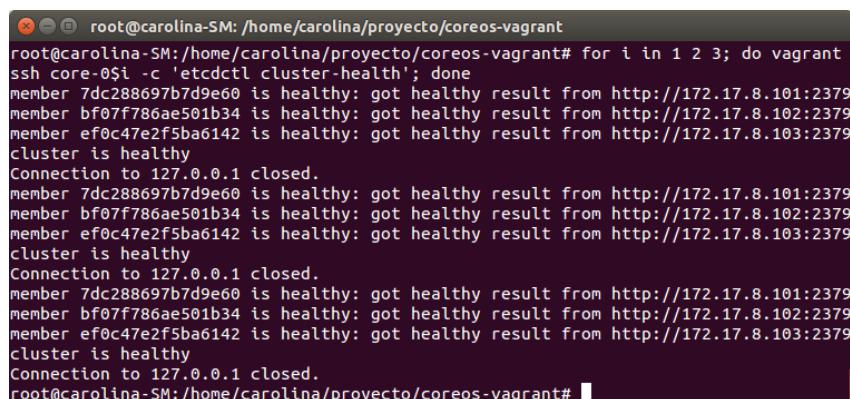
A su vez, se replica la infraestructura final para el proveedor VirtualBox, reemplazando el contenido del fichero **user-data.sampleapp.vbox** por el de **user-data.sampleapp.aws**, manteniendo la interfaz de flannel como pública y la red virtual 10.1.0.0/16. En el fichero **Vagrantfile** se pasarían a parte común la adición de los metadatos a las máquinas.

En este caso el despliegue será de la siguiente manera:

```
$ vagrant up
```

Cuando el despliegue termina se comprueba la salud positiva del clúster:

```
$ for i in 1 2 3; do vagrant ssh core-0$i -c 'etcdctl cluster-health'; done
```



```
root@carolina-SM: /home/carolina/proyecto/coreos-vagrant
root@carolina-SM:/home/carolina/proyecto/coreos-vagrant# for i in 1 2 3; do vagrant ssh core-0$i -c 'etcdctl cluster-health'; done
member 7dc288697b7d9e60 is healthy: got healthy result from http://172.17.8.101:2379
member bf07f786ae501b34 is healthy: got healthy result from http://172.17.8.102:2379
member ef0c47e2f5ba6142 is healthy: got healthy result from http://172.17.8.103:2379
cluster is healthy
Connection to 127.0.0.1 closed.
member 7dc288697b7d9e60 is healthy: got healthy result from http://172.17.8.101:2379
member bf07f786ae501b34 is healthy: got healthy result from http://172.17.8.102:2379
member ef0c47e2f5ba6142 is healthy: got healthy result from http://172.17.8.103:2379
cluster is healthy
Connection to 127.0.0.1 closed.
member 7dc288697b7d9e60 is healthy: got healthy result from http://172.17.8.101:2379
member bf07f786ae501b34 is healthy: got healthy result from http://172.17.8.102:2379
member ef0c47e2f5ba6142 is healthy: got healthy result from http://172.17.8.103:2379
cluster is healthy
Connection to 127.0.0.1 closed.
root@carolina-SM:/home/carolina/proyecto/coreos-vagrant#
```

Figura 3.76: Salud del clúster contemplada por las 3 máquinas.

También se obtienen las máquinas que componen el clúster desde la máquina **core-03** y las unidades de servicio desde **core-02**, como prueba de que se está implementando la infraestructura final:

```
$ vagrant ssh core-03
# fleetctl list-machines
```

```
core@core-03:~ $ fleetctl list-machines
MACHINE          IP           METADATA
1a4e67f3...     172.17.8.102   compute=db
3a6c826d...     172.17.8.103   -
60f7197d...     172.17.8.101   compute=proxy
core@core-03 ~ $
```

Figura 3.77: Información de las máquinas del clúster.

```
$ vagrant ssh core-02
# fleetctl list-units
```

UNIT	MACHINE	ACTIVE	SUB
app-job.service	1a4e67f3.../172.17.8.102	inactive	dead
app-job.service	3a6c826d.../172.17.8.103	inactive	dead
app-job.service	60f7197d.../172.17.8.101	inactive	dead
app-task.service	1a4e67f3.../172.17.8.102	active	running
app-task.service	3a6c826d.../172.17.8.103	active	running
app-task.service	60f7197d.../172.17.8.101	active	running
conf-data.service	60f7197d.../172.17.8.101	active	exited
confd.service	60f7197d.../172.17.8.101	active	running
nginx.service	60f7197d.../172.17.8.101	active	running
postgresql.service	1a4e67f3.../172.17.8.102	active	running
skydns.service	1a4e67f3.../172.17.8.102	active	running
skydns.service	3a6c826d.../172.17.8.103	active	running
skydns.service	60f7197d.../172.17.8.101	active	running

Figura 3.78: Información de las unidades de servicio del clúster.

Por último, se replican en VirtualBox cada una de las pruebas realizadas para el proveedor AWS, confirmando el correcto funcionamiento del servicio de la aplicación y balanceo de carga entre los diferentes servidores web. Así, se muestra el correcto resultado de una petición desde **core-01**:

```
$ vagrant ssh core-01
# curl http://localhost:80 | tail -n 15
```

```
core@core-01:~ $ curl http://localhost:80 | tail -n 15
% Total    % Received % Xferd  Average Speed   Time      Time     Current
          Dload  Upload Total   Spent    Left Speed
100  3983    0  3983    0     0  1642      0 --:--:--  0:00:02 --:--:-- 1643
<nav>
  <ul>
    <li><a href="/about">About</a></li>
    <li><a href="/contact">Contact</a></li>
    <li><a href="http://news.railstutorial.org/">News</a></li>
  </ul>
</nav>
</footer>
<pre class="debug_dump">--- !ruby/hash:ActionController::Parameters
controller: static_pages
action: home
</pre>
</div>
</body>
</html>core@core-01 ~ $
```

Figura 3.79: Acceso al servicio desde **core-01**.

El resultado final está presentado en los siguientes repositorios GitHub:

- [CarolinaSantana/sample_app_rails_4-1](#): a partir del que se construye la imagen de la aplicación.
- [CarolinaSantana/coreos-vagrant](#): Infraestructura como Código a desplegar en los proveedores VirtualBox o Amazon Web Services.

Capítulo 4

Aspectos Económicos

En este capítulo se muestran los aspectos económicos de la implementación en el proveedor Amazon Web Services.

En primer lugar, se presenta el coste anual que supondría mantener una instancia las 24 horas del día, según los tipos de instancia escogidos y sus tamaños correspondientes. En segundo lugar, se indica el coste anual por instancia que supondría realizar un determinado número de peticiones a la aplicación, teniendo en cuenta la transferencia de datos con direcciones IP elásticas o públicas. En tercer y último lugar, se exponen otras formas de reducir costes con una serie de estrategias que permiten establecer los recursos a utilizar con anterioridad a hacerlo.

Para el cálculo de ambas estimaciones se utiliza la herramienta AWS Calculator[31]. Esta utilidad permite conocer el coste mensual del uso de los servicios de AWS con diversas configuraciones. La opción de facturación escogida es la de pago por uso. También se tienen en cuenta como parámetros la región EE.UU. Este (Virginia) y clase de instancia Linux.

Amazon Elastic Compute Cloud (Amazon EC2) proporciona una amplia selección de tipos de instancias optimizadas para diferentes casos de uso. Los tipos abarcan varias combinaciones de capacidad de CPU, memoria, almacenamiento y redes. En el caso de almacenamiento, este proveedor posee el servicio Amazon Elastic Block Store (Amazon EBS)[32] que proporciona volúmenes de bloques persistentes.

Entre los tipos de instancias existentes se escogen las de uso general. El tipo de instancia **T2** es el utilizado en la fase de implementación. Cada tipo de instancia incluye uno o varios tamaños, lo que permite escalar los recursos según la carga de trabajo. Para la presente estimación se tienen en cuenta los principales tamaños: *micro, small, medium, large y extra large*.

En la [Tabla 4.1](#) se presentan los costes anuales para los tamaños escogidos de los tres diferentes tipos de instancia, teniendo en cuenta que estén encendidas las 24 horas del día.

Tipo	Modelo	CPU	Memoria GiB	Almacenamiento GB	Coste anual €
T2	micro	1	1	EBS	96,96
	small	1	2	EBS	181,80
	medium	2	4	EBS	412,92
	large	2	8	EBS	755,38
	xlarge	4	16	EBS	1.518,15
M3	medium	1	3,75	SSD 1 x 4	538,49
	large	2	7,5	SSD 1 x 32	1.068,85
	xlarge	4	15	SSD 2 x 40	2.137,69
M4	large	2	8	EBS	867,94
	xlarge	4	16	EBS	1.727,76

Tabla 4.1: Coste anual por tipo de instancia

A continuación se exponen brevemente las características principales de cada tipo de instancia y los casos comunes de uso:

- **T2:** Procesadores Intel Xeon de alta frecuencia y CPU en ráfagas que se rigen por créditos, acumulándolos cuando están inactivas y utilizándolos cuando están activas. Los casos de uso contemplan aplicaciones web, entornos de desarrollo, servidores de versiones, repositorios de código, microservicios, entornos de pruebas y reproducción y aplicaciones empresariales.
- **M3:** Procesadores Intel Xeon *E5-2670 v2* (Ivy Bridge) de frecuencia alta y almacenamiento basado en SSD. Los casos de uso contemplan bases de datos, tareas de procesamiento de datos que requieren memoria adicional, flotas de almacenamiento en caché, ejecución de servidores *back-end*, clústeres y aplicaciones empresariales.
- **M4:** Procesadores Intel Xeon *E5-2686 v4* (Broadwell) de 2,3 GHz o Intel Xeon *E5-2676 v3* (Haswell) de 2,4 GHz. Las instancias están optimizadas para el uso de volúmenes de almacenamiento Amazon EBS de manera predeterminada sin coste adicional. Los casos de uso contemplan los mismos que el tipo **M3**, con soporte para redes mejoradas.

Con la intención de conocer el tamaño de una petición a la aplicación para calcular el coste de su transferencia, se utiliza, como ejemplo, la página de inicio. Para medirlo se utiliza el comando *wc -c*, que expresa el tamaño en *bytes*, con el comando *curl*, en la máquina **core-01**:

```
core@ip-10-0-0-101:~$ curl http://34.202.188.229:80 | wc -c
% Total    % Received % Xferd  Average Speed   Time   Time     Current
          Dload  Upload Total   Spent    Left Speed
100  3983     0  3983     0      0  1309      0 --:--:--  0:00:03 --:--:-- 1309
3983
core@ip-10-0-0-101:~$
```

Figura 4.1: Tamaño de la página principal solicitada en B.

El resultado es de 3983 B, que serán tomados como 4 kB a efectos de la estimación del coste.

Cada instancia que se ejecute dispone del uso de una dirección IP elástica gratuita. La presente infraestructura no necesita hacer uso de direcciones IP públicas adicionales por lo que la utilización de este recurso no supone un coste adicional.

En la [Tabla 4.2](#) se presenta el coste del tráfico externo anual proveniente de la transferencia de datos con direcciones IP elásticas para un determinado número de peticiones por hora, con un valor individual de 4 kB.

Peticiones/Hora	Transferencia de Datos/Hora GB	Coste anual €
$1 * 10^6$	4	321,42
$2 * 10^6$	8	642,87
$4 * 10^6$	16	1.285,69
$8 * 10^6$	32	2.571,39
$16 * 10^6$	64	5.142,78
$32 * 10^6$	128	10.285,55
$64 * 10^6$	256	22.628,31
$128 * 10^6$	512	45.035,53

Tabla 4.2: Coste anual por instancia según el número de peticiones

En último lugar, existe la posibilidad de abaratar costes estableciendo una estrategia diferente a la del pago por uso. Entre ellas cabe destacar las siguientes:

- **Instancias de subasta:** Se puede pujar por capacidad informática libre en Amazon EC2 con descuentos de hasta el 90% en comparación con el precio bajo demanda. Preferible para aplicaciones solo viables con precios de computación muy bajos o para usuarios con necesidades de computación muy urgentes y voluminosas.
- **Instancias reservadas:** Ofrece un descuento hasta del 75% comparado con los precios de las instancias bajo demanda. Además, cuando se asignan instancias reservadas a una zona de disponibilidad específica, se proporciona una reserva de capacidad que se traduce en confianza adicional sobre la capacidad de lanzar instancias cuando se necesite. Son recomendadas cuando las aplicaciones tengan necesidades de estado constante o uso previsible y para usuarios que pueden comprometerse a utilizar Amazon EC2 durante un plazo de 1 o 3 años.
- **Hosts dedicados:** Servidor físico de Amazon EC2 dedicado para su uso. Reduce costos permitiendo usar las licencias existentes de software enlazado al servidor. Se puede adquirir bajo demanda, por hora, o como reserva con un descuento de hasta el 70% en comparación con el precio bajo demanda.

Capítulo 5

Resultados y Conclusiones

En este capítulo se presentan los resultados obtenidos, tras la fase de desarrollo, y las conclusiones alcanzadas del presente Trabajo de Fin de Máster.

5.1 Resultados

A continuación se presentan los resultados obtenidos.

La arquitectura de la aplicación está basada, ahora, en la filosofía de contenedores, utilizando Docker para su construcción y tratamiento. Cada servicio se encapsula con las mínimas funcionalidades necesarias.

La aplicación ha sido configurada para implementar los servicios con posterioridad a la construcción de su imagen Docker. Ésta vuelve a generarse con cada cambio o añadido en el repositorio GitHub en el que se desarrolla por medio de Travis CI, encargado de realizar la subida de la imagen resultante en el repositorio Docker Hub.

Los contenedores que implementan los servicios que permiten el funcionamiento de la aplicación son:

- **skydns**: Servidor DNS, SkyDNS, que permite a los contenedores de la aplicación encontrar a la base de datos.
- **some-postgres**: Base de datos PostgreSQL que contiene la información de la aplicación.
- **app-job**: Creador de las bases de datos, encargado de su migración y alimentación con información de ejemplo.
- **app-task**: Servidor web, puma, que constituye la aplicación y accede a la base de datos para obtener y escribir información.
- **some-nginx**: Servidor proxy inverso, Nginx, que recibe y dirige peticiones balanceando la carga entre los servidores web.
- **confd**: Controla si se registra o elimina un servidor web propiciando la recarga de la configuración de Nginx con la nueva información.
- **conf-data**: Almacenamiento compartido entre **some-nginx** y **confd** para poder hacer el cambio de configuración de Nginx.

Estos contenedores se distribuyen en un clúster dentro de máquinas CoreOS, como se observa en la [Figura 3.30](#), para n máquinas. Éste es posible realizarlo localmente en VirtualBox o remotamente en Amazon Web Services.

En todo caso deben haber 2 máquinas como mínimo. En la instancia **core-01** se ubica el contenedor **some-nginx**, **confd** y **conf-data**. Por su parte, el contenedor **some-postgres** se encuentra en la máquina **core-02**. Los contenedores **skydns**, **app-job** y **app-task** están replicados en todas las máquinas del clúster.

El resultado final del presente trabajo es la Infraestructura como Código que permite realizar el despliegue de la aplicación usando Vagrant. Ésta queda representada con los ficheros **Vagrantfile**, **config.rb** y **user-data.sampleapp.aws** para el proveedor

Amazon Web Services o `user-data.sampleapp.vbox` para el proveedor VirtualBox. El acceso a la aplicación es a través de Nginx, en la máquina `core-01`.

En el caso de VirtualBox:

```
$ vagrant up && vagrant ssh core-01
# curl http://localhost:80
```

En el caso de Amazon Web Services:

```
$ . ~/.aws-credentials/aws-credentials
$ vagrant up --provider=aws && vagrant ssh core-01
# curl http://localhost:80
```

El resultado final está presentado en los siguientes repositorios GitHub:

- [`CarolinaSantana/sample_app_rails_4-1`](https://github.com/CarolinaSantana/sample_app_rails_4-1): a partir del que se construye la imagen de la aplicación. Disponible en: https://github.com/CarolinaSantana/sample_app_rails_4-1.
- [`CarolinaSantana/coreos-vagrant`](https://github.com/CarolinaSantana/coreos-vagrant): Infraestructura como Código a desplegar en los proveedores VirtualBox o Amazon Web Services. Disponible en: <https://github.com/CarolinaSantana/coreos-vagrant>.

5.2 Conclusiones

Las conclusiones del presente trabajo se expresan en relación a la consecución de los objetivos, los trabajos futuros por realizar y la valoración personal.

5.2.1 Consecución de objetivos

Los objetivos específicos y generales definidos al inicio han sido logrados.

El estudio del estado del arte permitió conocer las tecnologías de virtualización, contenedores y orquestación actuales más relevantes, así como obtener mayor información sobre las propuestas para su uso. Por su parte, el análisis de la aplicación objeto hizo conocer las diferentes capas, características y componentes que permitieron comenzar la fase de implementación, respetando el orden y contemplando la inclusión de todas las iteraciones definidas inicialmente.

El desarrollo iterativo e incremental permitió ir constatando que cada uno de los cambios aplicados funcionaba correctamente y su integración seguía posibilitando el funcionamiento de la aplicación, pudiendo añadir nuevos servicios que no existían antes y que suponen una mejora en cada bloque.

Así, se ha conseguido implementar una infraestructura en desarrollo robusta, flexible y escalable. Todo ello bajo uno de los objetivos primordiales de la tecnología de contenedores, el aislamiento de manera que unos servicios no puedan repercutir en el funcionamiento de otros. También se ha logrado implementar un diseño que facilita la distribución en un clúster para aplicar el balanceo de carga y la alta disponibilidad de los servicios.

Todas las fases permitieron llegar al propósito final. Éste es la generación de un entorno reproducible de desarrollo, tanto en local como en la nube, a través de la infraestructura como código para el despliegue de una aplicación Ruby on Rails, utilizando las tecnologías de virtualización Docker y CoreOS en la nube pública de Amazon Web Services.

5.2.2 Trabajos futuros

En la presente sección se realizan propuestas con el fin de retroalimentar el presente trabajo y considerar su paso del entorno de desarrollo al de producción.

El contenedor **volume-public** no pudo continuar siendo compartido, de la manera en la que se había hecho, por los contenedores **some-postgres**, **app-job**, **app-task** y **some-nginx**. Para disponer de almacenamiento compartido entre ellos se propone la aplicación del protocolo iSCSI. Este protocolo utiliza TCP/IP para sus transferencias de datos, requiriendo una interfaz Ethernet para funcionar y permitiendo disponer de almacenamiento centralizado de bajo coste. iSCSI tiene una arquitectura cliente-servidor donde el primero se denomina *initiator* y el segundo *target*. Un *target* puede ofrecer uno o más recursos iSCSI por la red. El *initiator* consta de los módulos o *drivers* que proveen soporte para que el sistema operativo pueda reconocer discos iSCSI y un programa que gestiona las conexiones a dichos discos. Para ello Amazon Web Services dispone del servicio *AWS Storage Gateway*.

La aplicación implementa el modo de autenticación por medio de *tokens*. En el actual entorno de desarrollo se utiliza un determinado *token* de prueba que se pasa a los contenedores **app-job** y **app-task**. Esto podría ser cambiado de manera que el contenedor **confd** transmita el *token*, generado por el primer servidor web en realizar la petición, a los contenedores nombrados.

Otro de los cambios sería la manera en la que se crea, migra y alimenta la base de datos para que lo haga en el entorno de producción.

El inicio de sesión en la aplicación debe realizarse por medio del protocolo HTTPS, destinado a la transferencia segura de datos de hipertexto. Todo ello utilizando un cifrado basado en SSL/TLS para crear un canal cifrado y que la información sensible de las credenciales no pueda ser usada por un atacante que haya conseguido interceptar la transferencia de datos de la conexión. Además, el proxy inverso Nginx tendría que instalar el certificado SSL correspondiente.

Actualmente, en el caso del despliegue en la nube, cada instancia recibe una dirección IP pública. No obstante, tendría que recibirla solamente la instancia en la que reside el servidor proxy Nginx. Además, se trabaja con una red privada para situar las máquinas y una subred para los contenedores. A dicha subred se aplican las reglas definidas en el grupo de seguridad creado. La nueva propuesta consiste en crear distintas subredes para poder construir grupos de seguridad diferentes de forma que se apliquen determinadas reglas solo a las instancias que las necesitan. Por ejemplo, el puerto 80 solo se abriría a la máquina que haga uso de la dirección IP pública.

5.2.3 Valoración personal

En esta sección expongo mi valoración personal sobre el presente trabajo.

El desarrollo de esta propuesta de Trabajo de Fin de Máster me ha permitido profundizar en tecnologías actuales y emergentes de virtualización, contenedores y orquestación de aplicaciones.

Con su realización he podido consolidar algunas de las tecnologías vistas a lo largo del Máster, como son Vagrant, Travis CI, Docker y Amazon Web Services. Además, he trabajado con el sistema operativo orientado a contenedores CoreOS, una tecnología no tan conocida como prometedora. Su utilización ha aportado singularidad y riqueza al resultado final del trabajo, así como grandes aportaciones personales.

Reconozco la importancia y relevancia de los trabajos futuros a realizar para la mejora hacia

un entorno de producción. Estos trabajos no han podido ser llevados a cabo por una cuestión de tiempo, pues su aplicación requeriría una mayor duración a la contemplada para el Trabajo Fin de Máster.

A través de la definición inicial, reuniones virtuales y presenciales, he experimentado el hipotético caso del desarrollo de un arquetipo en base a las peticiones de clientes conocedores de qué tecnologías y herramientas quieren que use el mismo. Estos clientes están representados por los tutores del trabajo.

Este proyecto contribuye a la filosofía de uso y desarrollo de software libre, encontrándose disponible en la plataforma GitHub y Docker Hub, valorándolo como una de las herramientas de aprendizaje más potente y de transmisión efectiva de conocimiento y resultados hacia la presente y futura comunidad de informáticos. Todo ello esperando que, en la medida de lo posible, este trabajo pueda servir de ejemplo a otras personas para la construcción de arquetipos, basados en las tecnologías seleccionadas, con propósitos similares.

Fuentes de información

- [1] Sreenivas Makam. *Mastering CoreOS*. Packt Publishing, WEB 978-1-78528-830-2, Febrero de 2016.
- [2] Michael Hartl. *Ruby on Rails Tutorial (Rails 5)*. [Consulta: 17/11/16] Disponible en: <https://www.railstutorial.org>
- [3] Railstutorial. *railstutorial/sample_app_rails_4*. GitHub. [Consulta: 19/11/16] Disponible en: https://github.com/railstutorial/sample_app_rails_4
- [4] *What is Docker?*. Docker. [Consulta: 27/10/16] Disponible en: <https://www.docker.com/what-docker>
- [5] *Overview of Docker Hub*. Docker. [Consulta: 27/10/16] Disponible en: <https://docs.docker.com/docker-hub>
- [6] *A security-minded, standards-based container engine*. CoreOS. [Consulta: 27/10/16] Disponible en: <https://coreos.com/rkt>
- [7] *Getting started with etcd*. CoreOS. [Consulta: 28/10/16] Disponible en: <https://coreos.com/etcd/docs/latest/getting-started-with-etcd.html>
- [8] *Launching containers with fleet*. CoreOS. [Consulta: 28/10/16] Disponible en: <https://coreos.com/fleet/docs/latest/launching-containers-fleet.html>
- [9] *Configuring flannel for container networking*. CoreOS. [Consulta: 28/10/16] Disponible en: <https://coreos.com/flannel/docs/latest/flannel-config.html>
- [10] *Overview*. CoreOS. [Consulta: 28/10/16] Disponible en: <https://kubernetes.io/docs/tutorials/kubernetes-basics>
- [11] *Red Hat Enterprise Linux Atomic Host 7*. Red Hat Documentation. [Consulta: 28/10/16] Disponible en: <https://access.redhat.com/documentation/en/red-hat-enterprise-linux-atomic-host/7/single/installation-and-configuration-guide>
- [12] *UbuntuCore*. Ubuntu. [Consulta: 28/10/16] Disponible en: <https://www.ubuntu.com/core>
- [13] *DigitalOcean*. [Consulta: 28/10/16] Wikipedia, La enciclopedia libre, 25 de Septiembre de 2016. Disponible en: <https://es.wikipedia.org/wiki/DigitalOcean>
- [14] *Cloud Computing con Amazon Web Services*. AWS. [Consulta: 28/10/16] Disponible en: <https://aws.amazon.com/es/what-is-aws/>
- [15] *What is IAM?*. AWS. [Consulta: 28/10/16] Disponible en: http://docs.aws.amazon.com/es_es/IAM/latest/UserGuide/introduction.html
- [16] *What is Amazon EC2?*. AWS. [Consulta: 28/10/16] Disponible en: http://docs.aws.amazon.com/es_es/AWSEC2/latest/UserGuide/concepts.html

- [17] *What is Amazon VPC?*. AWS. [Consulta: 28/10/16] Disponible en: http://docs.aws.amazon.com/es_es/AmazonVPC/latest/UserGuide/VPC_Introduction.html
- [18] *Docker Swarm overview*. Docker Docs. [Consulta: 28/10/16] Disponible en: <https://docs.docker.com/swarm/overview>
- [19] *Apache Mesos*. Wikipedia, La enciclopedia libre, 10 de Noviembre de 2016. [Consulta: 28/10/16] Disponible en: https://en.wikipedia.org/wiki/Apache_Mesos
- [20] *GitHub Introducción*. Conociendo GitHub Docs. [Consulta: 28/10/16] Disponible en: <http://conociendogithub.readthedocs.io/en/latest/data/introduccion>
- [21] *About Vagrant*. Vagrant. [Consulta: 29/10/16] Disponible en: <https://www.vagrantup.com/about.html>
- [22] *VirtualBox*. [Consulta: 29/10/16] Wikipedia, La enciclopedia libre, 19 de Octubre de 2016. Disponible en: <https://en.wikipedia.org/wiki/VirtualBox>
- [23] Softcover. [Consulta: 30/10/16] *softcover/softcover*. GitHub. Disponible en: <https://github.com/softcover/softcover>
- [24] *Getting started with Travis CI*. Travis CI Docs. [Consulta: 17/12/16] Disponible en: <https://docs.travis-ci.com>
- [25] *Getting Started with systemd*. CoreOS. [Consulta: 14/01/17] Disponible en: <https://coreos.com/docs/launching-containers/launching/getting-started-with-systemd/>
- [26] CoreOS. *coreos/coreos-vagrant*. GitHub. [Consulta: 22/01/17]. Disponible en: <https://github.com/coreos/coreos-vagrant>
- [27] *AWS Educate*. AWS. [Consulta: 22/01/17]. Disponible en: <https://aws.amazon.com/es/education/awseducate>
- [28] Miek Gieben. *skynetservices/skydns*. GitHub. [Consulta: 24/04/17] Disponible en: <https://github.com/skynetservices/skydns>
- [29] Paul Dixon. *Load balancing with coreos, confd and nginx*. LordElph's Ramblings, 1 de Febrero de 2015. [Consulta: 02/05/17] Disponible en: <http://blog.dixo.net/2015/02/load-balancing-with-coreos>
- [30] Kelsey Hightower. *kelseyhightower/confd*. GitHub. [Consulta: 04/05/17] Disponible en: <https://github.com/kelseyhightower/confd>
- [31] *Monthly Calculator*. AWS Calculator. [Consulta: 05/05/17] Disponible en: <https://calculator.s3.amazonaws.com/index.html>
- [32] *Amazon Elastic Block Store*. AWS. [Consulta: 05/05/17] Disponible en: <https://aws.amazon.com/es/ebs>
- [33] *Precios de Amazon EC2*. AWS. [Consulta: 09/05/17] Disponible en: <https://aws.amazon.com/es/ec2/pricing>
- [34] *Tipos de instancias de Amazon EC2*. AWS. [Consulta: 09/05/17] Disponible en: <https://aws.amazon.com/es/ec2/instance-types>