Automatización del despliegue de una aplicación web en un cluster Swarm utilizando los servicios en la nube de Amazon Web Services y tecnología Docker

Trabajo Fin de Máster

Autora: María García Ramírez

Tutora: Francisca Quintana Domínguez

Tutor: Carmelo Cuenca Hernández

Índice

1. Introducción y objetivos ... 6
1.1 Resumen del trabajo fin de máster ... 6
1.2 Objetivos y planificación y seguimiento ... 7
1.2.1 Objetivos ... 7
1.2.2 Planificación ... 8
1.2.3 Seguimiento ... 9
1.3 Aportaciones ... 10
1.4 Justificación de las competencias específicas cubiertas ... 10
1.4.1 Competencia TI01: Capacidad para modelar, diseñar, definir la arquitectura, implantar, gestionar, operar, administrar y mantener aplicaciones, redes, sistemas, servicios y contenidos informáticos. ... 10
1.4.2 Competencia TI06: Capacidad para diseñar y evaluar sistemas operativos y servidores, y aplicaciones y sistemas basados en computación distribuida. ... 11
1.4.3 Competencia TI12: Capacidad para la creación y explotación de entornos virtuales, y para la creación y distribución de contenidos multimedia. ... 11
2. Estado del arte ... 11
2.1 Tecnologías ... 11
2.1.1 Orquestación ... 11
2.1.2 Los servicios de orquestación y DevOps ... 12
2.1.3 Orquestación y automatización ... 12
2.1.4 Contenerización ... 13
2.1.5 Docker ... 16
2.1.6 Docker Hub ... 20
2.1.7 Docker frente a otras tecnologías de contenerización ... 21
2.1.8 Plataformas para el soporte de contenedores ... 22
2.2 Proveedores de Infraestructura como Servicio (IaaS) ... 25
2.2.1 Amazon Web Services (AWS) ... 27
2.3 Otras herramientas tecnológicas ... 31
2.3.1 Virtualbox ... 31
3. Desarrollo ... 32
3.1 Análisis de la aplicación básica sample_app_rails_4 en Ruby on Rails ... 32

3.2 Iteración 1: Conversión a una arquitectura de microservicios con el uso de Docker 36
3.2.1 Preparación del repositorio local y remoto 37
3.2.2 Cambio y configuración de la base de datos 37
3.2.3 Creación de la imagen Docker de la aplicación de Ruby 38
3.2.4 Cambio y configuración del servidor web 39
3.2.5 Configuración automatizada para la creación de los contenedores 41
3.3 Iteración 2: Conversión a una arquitectura de microservicios con el uso de Docker Compose. 46
3.3.1 Configuración automatizada para la creación de los contenedores con docker-compose 46
3.3.2 Resultados 48
3.4 Iteración 3: Adaptación de la configuración para que cada una de las capas de la aplicación funcione en tres MV locales diferentes. 50
3.4.1. Separación de los servicios en tres ficheros docker-compose diferentes 50
3.4.2. Adaptación de la aplicación para que cada capa se ejecute en una MV diferente con docker-machine y docker-compose. 54
3.4.3. Resultados 56
3.5 Iteración 4: Despliegue remoto de la aplicación, en Amazon Web Services (AWS) 60
3.5.1 Configuración desde la consola de AWS 60
3.5.2 Modificación del script de despliegue de la aplicación 63
3.5.3 Despliegue de la aplicación en AWS 65
3.5.4 Resultados 66
3.6 Iteración 5: Despliegue de un clúster Swarm local con un maestro y 2 nodos para desplegar la infraestructura 70
3.6.1 Modificación de los ficheros de configuración para desplegar la aplicación en un cluster Swarm en local 71
3.6.2 Resultados 77
3.7 Iteración 6: Despliegue de un clúster con un maestro y 2 nodos agentes de manera remota en AWS 84
3.7.1 Creación de los grupos de seguridad 85
3.7.2 Modificación del script de despliegue para desplegar el cluster en AWS 86
3.7.3 Despliegue del cluster Swarm en AWS 89
3.7.4 Resultados 89
4. Resultados y conclusiones 97
4.1 Resultados 97
4.2 Conclusiones 98

4.2.1 Consecución de objetivos 98
5 Anexos 101
5.1 Repositorio de la aplicación 101
5.2 Instalación de los componentes básicos para realizar el despliegue de la aplicación 101
6. Referencias 103

1. Introducción y objetivos

1.1 Resumen del trabajo fin de máster

DevOps es el acrónimo inglés de *development* (desarrollo) y *operations* (operaciones) que da nombre al movimiento que propone la homogeneización de los entornos de desarrollo y producción, y la colaboración de los desarrolladores con las personas responsables de las infraestructuras tecnológicas. Los entornos de trabajos similares y la colaboración entre desarrolladores y operadores implican la utilización de tecnologías locales y remotas, tanto de virtualización como de contenerización.

A menudo se encuentran muchos obstáculos para mover una aplicación durante el ciclo de desarrollo y eventualmente al moverla a producción. La mayoría de los problemas son consecuencia de las dependencias de la aplicación hacia el entorno de desarrollo, del escalado de la aplicación y la actualización de componentes sin afectar la aplicación por completo. Con lo cual el trabajo de desarrollo conlleva la implementación de una aplicación que responda apropiadamente a cada entorno, con el objetivo final de que la aplicación se abstraiga del entorno y facilitar así las futuras tareas de mantenimiento de la misma.

Docker junto al diseño orientado a servicios permite resolver todos estos problemas. Las aplicaciones pueden descomponerse en componentes funcionales y manejables empaquetados individualmente con todas sus dependencias, e implementarlas en arquitecturas irregulares fácilmente. La escalabilidad y actualización de componentes también se simplifican.

Este TFM construye la infraestructura virtual necesaria para una aplicación web de tres capas, ya desarrollada, de manera que un miembro del equipo de desarrollo pueda llevar a cabo el despliegue en entornos de desarrollo locales y remotos similares a los entornos de producción utilizando en la medida de lo posible el principio de convención frente a configuración.

El objetivo final de este proyecto es tener como resultado los ficheros y procedimientos necesarios para llevar a cabo la automatización del despliegue en un entorno de desarrollo de una aplicación web en un cluster Swarm utilizando los servicios en la nube de Amazon Web Services y tecnología Docker.

1.2 Objetivos y planificación y seguimiento

1.2.1 Objetivos

Los objetivos iniciales de este TFM consistían en obtener como resultado un repositorio que incluirá un manual de procedimiento para desplegar la aplicación tanto en un cluster Swarm local como remotamente en la nube pública de Amazon Web Services, mediante la ejecución de operaciones básicas. Este repositorio incluirá además todos los *scripts* y ficheros de configuración para hacer uso de las tecnologías utilizadas.

Estos objetivos los hemos refinado en objetivos generales y objetivos específicos.

Los ***objetivos generales*** son los siguientes:

El objetivo principal de este TFM es hacer uso de los sistemas y proveedores de servicios en la nube más utilizados y valorar las utilidades y beneficios que aportan.

Para el desarrollo de este trabajo fin de máster se va a hacer un estudio de la tecnología disponible para el despliegue de aplicaciones en la nube en el que se analizará el conjunto de herramientas que proporciona cada tecnología y los beneficios que proporciona cada una.

Se realizará un análisis del estado actual y de los nuevos modelos de desarrollo de software basados en la DevOps y en la arquitectura de microservicios.

Los ***objetivos específicos*** son los siguientes:

Se va a realizar el despliegue de una aplicación básica que consta de tres capas: una base de datos Postgres, una aplicación codificada en Ruby on Rails y un servidor web Nginx, que será utilizado como proxy de la aplicación.

Se realizará un análisis de Docker y del conjunto de herramientas proporcionados por el mismo (docker-machine, docker-compose y docker Swarm).

El resultado del trabajo es un repositorio que incluye un manual de procedimiento para ejecutar el despliegue de la aplicación tanto en un cluster Swarm local como remotamente en la nube pública de Amazon Web Services (AWS), mediante la ejecución de operaciones básicas

haciendo uso del conjunto de herramientas proporcionadas por la tecnología Docker. Este repositorio incluirá además todos los scripts y ficheros de configuración para hacer uso de las tecnologías utilizadas.

1.2.2 Planificación

Estudio previo/Análisis

Tarea 1.1 - Familiarización con las tecnologías a utilizar y Entendimiento del proyecto.

Diseño/Desarrollo/Implementación

Tarea 2.1 - Configuración de un máquina Vagrant con VirtualBox y Docker instalados.

Tarea 2.2 - Adaptación de la aplicación web de tres capas. Despliegue local de la aplicación en una única máquina virtual (MV) mediante contenedores lanzados desde un shell.

Tarea 2.3 - Despliegue local de la aplicación mediante un fichero de descripción utilizando la herramienta docker-compose.

Tarea 2.4 - Adaptación de la configuración Vagrant para que cada una de las capas de la aplicación funcione al menos en dos MV diferentes (una para el servicio de base de datos y otra para el servidor web y de aplicaciones). Despliegue local y remoto de la aplicación, en Amazon Web Services (AWS), mediante un fichero de descripción, utilizando las herramientas docker-machine y docker-compose. Resolución del problema del descubrimiento de los servicios a través de Consul u overlay networking.

Tarea 2.5 - Despliegue de un clúster Swarm local con al menos un maestro y dos nodos para desplegar la infraestructura.

Tarea 2.6 - Despliegue de un clúster Swarm remoto (AWS) con al menos un maestro y dos nodos para desplegar la infraestructura.

Evaluación/Validación/Prueba

Tarea 3.1 - La aplicación a desplegar utiliza TDD, las pruebas consistirán en comprobar que los tests continúan en el ciclo verde cuando la infraestructura está desplegada.

Documentación/Presentación

Tarea 4.1 - Realización de la memoria a medida que los diferentes hitos del proyecto son conseguidos.

1.2.3 Seguimiento

Los objetivos iniciales en los que se basa este TFM no fueron modificados durante el desarrollo del proyecto aunque sí fueron refinaron para produndizar en los objetivos generales y específicos del mismo.

La planificación del proyecto, por su parte, ha sufrido alguna modificación. Es decir, en el apartado 2.1 y 2.4 de la implementación en el que se consideraba a una máquina con Vagrant como host de la aplicación, se ha descartado el uso de Vagrant puesto que Virtualbox no da soporte a la virtualización anidada, con lo que no se permite lanzar máquinas con Docker Machine desde el host virtual creado. Por otro lado, Docker Machine ya proporciona todas las herramientas necesarias para la creación de máquinas virtuales y su gestión desde el host, por medio de una configuración sencilla y rápida.

La fase de validación y pruebas, apartado 3.1, se ha centrado en la comprobación del funcionamiento completo de la aplicación tras cada modificación a través del Shell y del navegador. Tras cada iteración, se ha comprobado la conexión entre los distintos componentes que conforman la aplicación: la conexión con la base de datos y comprobación de su correcto funcionamiento y el acceso a la aplicación de Ruby a través del servidor Nginx.

1.3 Aportaciones

Este trabajo se centra en la nueva tendencia de desarrollos basados en DevOps que se fundamenta en la integración entre desarrolladores software y administradores de sistemas y cuya finalidad es agilizar el proceso de desarrollo y entrega, automatizando para ello los proceso de entrega del software y de las modificaciones en las infraestructuras.

En este TFM se exploran los componentes de Docker y se utilizan las distintas herramientas que presenta tanto para la virtualización de hosts (Docker Machine) como para la virtualización de aplicaciones o servicios en forma de imágenes o contenedores a partir de los ficheros Dockerfile y la posterior creación y lanzamiento de servicios, definidos a través de los ficheros de configuración de Docker Compose y que se crearán a partir de las imágenes creadas de los componentes de la aplicación.

Además se hará un repaso de las tecnologías y proveedores de servicios actuales que ofrecen las herramientas y plataformas para la automatización del desarrollo y del despliegue.

Como resultado de este trabajo se obtendrá un repositorio con los pasos a seguir y los ficheros necesarios para automatizar el despliegue de una aplicación de tres capas en un cluster Swarm que se ejecute en AWS, integrando la tecnología Docker, las utilidades de Docker Machine, Docker Compose y Swarm con los servicios que ofrece Amazon para ejecutar aplicaciones en la Nube.

1.4 Justificación de las competencias específicas cubiertas

1.4.1 Competencia TI01: Capacidad para modelar, diseñar, definir la arquitectura, implantar, gestionar, operar, administrar y mantener aplicaciones, redes, sistemas, servicios y contenidos informáticos.

Se va a realizar el despliegue de una aplicación de tres capas (base de datos, aplicación y servidor web) a partir de una arquitectura definida y una configuración de servicios iniciales en la que se ha decidido distribuir cada capa de la aplicación en una máquina distinta dentro de un cluster. Además de la implantación del sistema se realizará operación de administración con el mismo.

1.4.2 Competencia TI06: Capacidad para diseñar y evaluar sistemas operativos y servidores, y aplicaciones y sistemas basados en computación distribuida.

Esta competencia se cumple con este TFM puesto que se va a desplegar una aplicación distribuida entre varios nodos de un cluster, en el que cada nodo será proveedor de un servicio de la aplicación.

1.4.3 Competencia TI12: Capacidad para la creación y explotación de entornos virtuales, y para la creación y distribución de contenidos multimedia.

Se va a hacer uso de contenedores y máquinas virtuales para el despliegue de la aplicación. Los contenedores se corresponden con servicios virtuales y las máquinas virtuales harán de host de un servicio de la aplicación.

2. Estado del arte

2.1 Tecnologías

2.1.1 Orquestación

La tecnología ha evolucionado a lo largo de los años y ha pasado de ser una utilidad para una empresa a convertirse en el conductor de la misma. Hoy en día los tiempos, el escalado y los servicios que ofrecen los IT son factores críticos para la empresa por lo que no se puede permitir que nada ralentice el proceso. Debido a ello, los sistemas que funcionaban tradicionalmente en el pasado a día de hoy han quedado obsoletos. Todavía hoy los administradores necesitan involucrarse en el proceso, pero la forma en la que se involucran también ha cambiado.

Los departamentos han hecho uso de la automatización para ayudar a afrontar estos retos. Los administradores crean scripts y otras herramientas de automatización para completar pasos o tareas de forma rápida. Esto permite a los equipos IT a responder de forma más eficaz a las necesidades de la compañía y concede además una mayor eficiencia y escalabilidad del sistema. A medida que las compañías se mueven hacia el *DevOps*, la automatización basada en tareas ha

empezado a mostrar sus limitaciones. Los servicios de orquestación conectan conceptos y flujos de la automatización basada en tareas con la lógica de negocio, usando automatización para ofrecer los servicios.

2.1.2 Los servicios de orquestación y DevOps

El DevOps combina las prácticas de los desarrolladores, operaciones y testeo de la calidad en algo que puede producir resultados mayores que la suma de las partes por separado. Es en el entorno de DepOps donde los servicios de orquestación pueden producir los mayores beneficios a una compañía. La automatización tradicional a este nivel estaría todavía limitada por la naturaleza de las barreras entre procesos, mientras que la orquestación está diseñada para funcionar dentro y fuera de dichas barreras.

Debido que el DevOps se mueve entre diferentes entornos, la orquestación de servicios es fundamental para unificar diferentes piezas de la automatización de los diferentes conjuntos de herramientas de las DevOps.

El hecho de que los DevOps pueden adoptar el conjunto de herramientas que proporcionan los servicios de orquestación permite que los equipos de DevOps puedan crear rápidamente tareas de automatización y unirlas a las plataformas de orquestación sin tener que parar y recrear todo de nuevo.

2.1.3 Orquestación y automatización

La orquestación y la automatización son dos términos que suelen llevar a confusión aunque tienen diferentes significados y objetivos. La *automatización* ejecuta de forma automatizada tareas o pasos para completar una tarea y está típicamente destinada a alcanzar un único objetivo, como puede ser desplegar un servidor. Los servicios de orquestación toman esas tareas y las sitúan dentro de un proceso o workflow, que puede incluir múltiples tareas de automatización y además coordinar los mismos procesos y workflow.

La automatización de tareas es ideal cuando se tienen que ejecutar múltiples pasos en una única área. Cuando se va más allá del área de influencia la automatización deja de funcionar. Esto es lo que se denomina tradicionalmente como workflow porque se trata de un proceso que se mueve entre múltiples grupos o departamentos en una secuencia de etapas.

Las herramientas de automatización son independientes a la orquestación mientras que la orquestación depende por completo de la automatización para conducir tareas y eventos. La integración de estas distintas herramientas en el proceso de desarrollo de una aplicación es crítico para el éxito del DevOps. La plataforma de orquestación no reemplaza a la automatización sino que une lo anterior en una plataforma.

Los workflow se están convirtiendo en elementos críticos debido a que ayudan a los equipos de DevOps a establecer checkpoints, de manera que puedan suprimir pasos omitidos o problemas de configuración durante el proceso de despliegue. Cuando se aplica la orquestación se suprime el factor humano del workflow, con lo que se suprime el riesgo a concurrir en errores debidos al factor humano y permite centrarse en tareas de mayor criticidad. El objetivo final es que cuando los desarrolladores soliciten el recurso, éste sea provisionado por el entorno virtual a través de la automatización, siendo este proceso controlado mediante la orquestación. El recurso virtualizado es añadido a las herramientas de monitorización y operaciones a través también de la automatización y bajo el control de la orquestación. La orquestación se encarga así de controlar la automatización.

2.1.4 Contenerización

La contenerización, también denominada virtualización basada en contenedores, es un tipo de virtualización a nivel de sistema operativo para el despliegue y ejecución de aplicaciones distribuidas sin la necesidad de lanzar una máquina virtual al completo por cada aplicación. En su lugar, se lanzan múltiples sistemas aislados denominados contenedores, que corren sobre un único host y acceden a un mismo kernel.

Al compartir el mismo SO (sistema operativo) que el kernel del host, pueden ejecutarse de manera más eficiente que una MV (máquina virtual), la cual requiere instancias de SO separadas.

En la figura 2.1 se muestra las diferencias entre la arquitectura de la contenerización y la virtualización, mientras que los contenedores se ejecutan sobre el kernel del host, en la virtualización cada guest ejecuta su propio SO y requiere de un hipervisor que se encargue de la virtualización de los recursos del host.

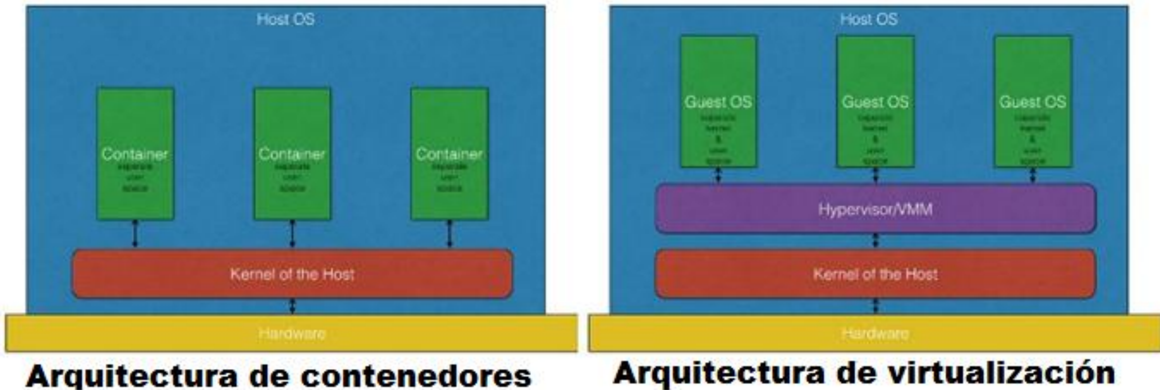

Figura 2.1 Comparación de la arquitectura del despliegue: Contenedores (izqda.) VS Virtualización (dcha.) [9]

Los contenedores tienen los componentes necesarios para ejecutar el software deseado, así como ficheros, variables de entorno y librerías. El sistema operativo del host controla el acceso del contenedor a los recursos físicos del host (como CPU y memoria) de forma que un único contenedor no consuma todos los recursos.

Beneficios del uso de contenedores

Los contenedores son un método de virtualización del sistema operativo que permiten ejecutar una aplicación y sus dependencias en procesos de recursos aislados. Los contenedores también permiten empaquetar con facilidad el código de una aplicación, sus configuraciones y sus dependencias en bloques de construcción de uso sencillo que aportan uniformidad de entorno, eficacia operativa, productividad para los desarrolladores y control de versiones. Los contenedores ayudan a garantizar la implementación rápida, consistente y de confianza de las aplicaciones independientemente del entorno. Los contenedores también aportan un control más minucioso de los recursos, lo que se traduce en una mayor eficacia de la infraestructura.

Los contenedores facilitan la portabilidad y ayudan a reducir las dificultades organizativas y técnicas que supone guiar una aplicación a lo largo del ciclo de vida de desarrollo, pruebas y producción. Los contenedores incorporan todos los archivos de la aplicación y dependencias de software necesarios que permite su implementación sobre cualquier sistema, independientemente de las configuraciones del software o sistema operativo. Todo lo que se empaqueta como contenedor de forma local se implementará y ejecutará del mismo modo tanto en un entorno de

pruebas como en uno de producción. De ese modo, se evita tener que configurar cada uno de los servidores manualmente y permite incorporar nuevas características con mayor rapidez.

Por otro lado, se puede especificar la cantidad exacta de memoria, espacio en disco y CPU que ha de usar un contenedor en una instancia. Debido a que constituyen un único proceso del sistema operativo ofrecen tiempos de inicio rápidos. Se pueden crear y eliminar aplicaciones o tareas de un contenedor de forma que se puede ampliar o reducir el tamaño de las aplicaciones con rapidez. Asimismo, la aplicación entera y todas sus dependencias se encuentran en una imagen. Los contenedores también aíslan el proceso, lo que permite colocar cada una de las aplicaciones y sus dependencias en un contenedor independiente y ejecutarlas en la misma instancia. Debido al aislamiento de las aplicaciones no existen dependencias compartidas ni incompatibilidades.

Además, se pueden crear imágenes de contenedores que sirvan como base para otras imágenes. Se puede crear una imagen base compuesta del sistema operativo, las configuraciones y las distintas utilidades que se deseen y construir luego una aplicación sobre dicha imagen base. Eso permite al equipo de desarrollo evitar las complejidades de la configuración un servidor.

Los contenedores incrementan la productividad de los desarrolladores al eliminar las dependencias y los conflictos entre servicios. Cada componente de una aplicación se puede dividir entre varios contenedores que ejecutan un microservicio distinto. Los contenedores están aislados entre sí, de manera que no existe preocupación de que las bibliotecas o dependencias estén sincronizadas para cada servicio, permitiendo a los desarrolladores actualizar cada uno de los servicios de manera independiente.

Otro de los beneficios más obvios es que el uso de los contenedores ofrece una gran *escalabilidad*. Por ejemplo, en el caso de múltiples instancias de servicios web cada una con su propia base de datos, en un despliegue tradicional éste tendría lugar sobre una única base de datos, lo cual dificulta la gestión y priorización de la carga de trabajo generada por cada sitio. Los contenedores en su lugar proporcionan una mayor flexibilidad en la gestión de la carga de trabajo sin la necesidad de desplegar múltiples máquinas virtuales.

Este tipo de implementación en un tipo de Plataforma como servicio (PaaS o Platform as a Service), donde el contenedor no necesita realizar mantenimiento o parcheado ya que estas tareas son desempeñadas por el mismo sistema operativo.

En la siguiente tabla (tabla 2.1) se muestra una comparativa de las características de la contenerización y la virtualización.

Contenerización	Virtualización
Virtualización basada en SO **Múltiples contenedores sobre el mismo OS**	**Virtualización basada en hardware** Múltiples SO de guests comparten los recursos del host
Ligero **Servicios de aplicaciones compartiendo recursos del SO**	**Pesado** SO adicionales gestionando recursos para cada guest
Provisionamiento en tiempo real **De forma sencilla, en el momento en el que se comienzan a ejecutar nuevos servicios de aplicaciones**	**Provisionamiento lento** Requiere inicialización del guest, más el tiempo de arranque del SO
Ejecución nativa **Acceso directo a los recursos del hardware**	**Ejecución limitada** Acceso a los recursos del hardware por medio de la capa de virtualización
Menos seguridad **Aislamiento únicamente a nivel de procesos**	**Alta seguridad** Aislamiento total

Tabla 2.1 Comparativa de la contenerización con la virtualización [9].

2.1.5 Docker

Docker es un proyecto de código abierto basado en contenedores de Linux. Se trata de un motor de contenedores que usan las características del Kernel de Linux, como espacios de nombres y controles de grupos, para crear contenedores encima del Sistema operativo y automatizar el despliegue de aplicaciones en estos contenedores.

La implementación de la arquitectura de Docker es una de las más exitosas de las arquitecturas de contenerización. Ofrece un conjunto completo de servicios como el almacenamiento, realizando las conexiones con las aplicaciones con complementos que ofrecen una gran escalabilidad.

Docker usa una arquitectura cliente-servidor como se muestra en la figura 2.2. El cliente de Docker se comunica con el demonio de Docker, el cual se encarga de crear, correr y distribuir los contenedores. Tanto el cliente como el demonio pueden ejecutarse en el mismo Sistema o, también, se puede conectar un cliente remoto a un demonio de Docker.

El cliente de Docker es la principal interfaz de usuario para Docker, éste acepta los comandos del usuario y se comunica con el demonio de Docker.

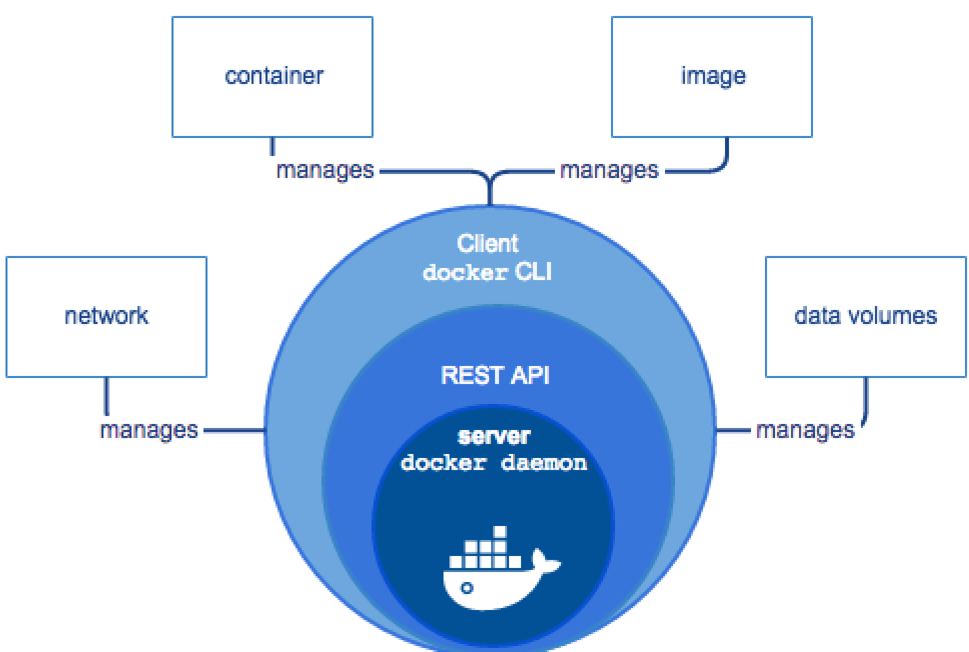

Figura 2.2 Docker Engine [1]

Por lo tanto, los principales componentes de la aplicación cliente-servidor del Docker Engine son los siguientes:

- Un servidor el cual es un proceso lanzado mediante el comando ‘dockerd’.
- Un REST API en el que se especifican las interfaces que pueden usar los programas para comunicarse con el demonio de Docker e indicarle qué hacer.

- Una interfaz por línea de comandos (Command line interface, CLI), a partir de las herramientas proporcionadas por el comando ‘docker, que actúa como cliente.

Para comprender cómo funciona Docker internamente se deben conocer tres componentes: las imágenes de Docker, los registros de Docker y los contenedores de Docker.

- **Imágenes de Docker (Docker Images):** Las imágenes de Docker son plantillas de solo lectura, es decir, los contenedores se crearán en función de la configuración definida en estas imágenes. Si se hacen cambios en el contenedor una vez lanzado, al detenerlo las modificaciones no se verán reflejadas en la imagen. Las imágenes del contenedor de Docker incorporan un archivo de manifiesto (Dockerfile), a partir del cual se describe su configuración, que permite mantener y supervisar las versiones de un contenedor, examinar diferencias entre versiones y volver a versiones anteriores con facilidad. De esta manera, se puede empaquetar una aplicación en una unidad estandarizada para el desarrollo de software, que contiene todo lo que necesita para funcionar: código, tiempo de ejecución, herramientas y bibliotecas del sistema, etc.
- **Registros de Docker (Docker Registries):** Los registros de Docker son el componente de distribución de Docker. Guardan las imágenes en repositorios públicos o privados donde podemos subir o descargar imágenes. El registro público lo provee el Docker Hub que pone a la disposición una colección de imágenes para su uso.
- **Contenedores de Docker (Docker Containers):** El contenedor de Docker se compone de todo lo necesario para ejecutar una aplicación. Cada contenedor es creado a partir de una imagen de Docker. Cada contenedor es una plataforma aislada.

Docker permite implementar las aplicaciones de forma rápida, fiable y sistemática, en cualquier entorno. La ejecución de Docker en AWS proporciona una manera muy fiable y económica de crear, ejecutar, probar e implementar aplicaciones distribuidas de forma rápida y a cualquier escala. AWS proporciona soporte técnico para las soluciones de código abierto o comercial de Docker, dentro de los servicios de AWS.

La tecnología de Docker proporciona herramientas para la creación y gestión de clúster (Docker Swarm), para la creación de hosts virtuales (Docker Machine) y para la creación y ejecución de servicios de aplicaciones a partir de ficheros de configuración *yaml* (Docker Compose). Estas herramientas se describen a continuación:

- **Docker Swarm**

  Docker Swarm es una herramienta para la creación y programación de clusters creado para contenedores Docker. Los administradores y desarrolladores pueden establecer y administrar un grupo de nodos Docker tratándolos como un único sistema virtual.

  Docker Engine CLI incluye los comandos para la gestión del cluster, así como para añadir o eliminar nodos y todos los comandos necesarios para el despliegue de los servicios. Cuando se ejecuta el Docker Engine fuera del modo Swarm se ejecutan los comandos propios del contenedor, mientras que cuando se encuentra integrado en el cluster se utilizan para gestionar los servicios. Las máquinas que integran el cluster son ejecutadas en modo Swarm. Las máquinas pueden entrar en modo Swarm (enjambre) tanto inicializando las máquinas en el modo como uniéndose a un cluster ya constituido.

  Para el despliegue de la aplicación en el cluster se envían la definición del servicio al nodo manager. Este nodo envía unidades de trabajo, denominanas tareas, a los nodos esclavos. Los managers son los encargados de dirigir y gestionar el cluster. El agente del cluster notifica al manager acerca del estado de las tareas del mismo para el logro del estado deseado del cluster.

  Los nodos esclavos reciben y ejecutan las tareas designadas por el manager. Por defecto los nodos manager son también nodos esclavos, pero pueden ser configurados para ser únicamente managers.

- **Docker Machine**

  Es una herramienta que permite instalar Docker en host virtuales y gestionar dichos hosts con el comando ‘docker-machine’. Se puede usar tanto para crear hosts en máquinas locales como remotas, a través de proveedores de servicios en la nube como AWS o Digital Ocean. Los comandos de ‘docker-machine’ permiten arrancar máquinas, inspeccionarlas, pararlas, reiniciarlas, actualizar el cliente Docker y configurarlo para que se comunique con el host. Además, permite que se puedan ejecutar comandos en las máquinas virtuales creadas directamente desde un shell del host, estableciendo para ello el entorno de la máquina con el comando ‘eval $(docker-machine env nombremaquina)’ y ejecutando a continuación los comandos que se quieren ejecutar dentro de la máquina virtual.

Docker Machine permite provisionar múltiples hosts de Docker remotos.

- **Docker Compose**

  Es una herramienta que permite definir y ejecutar aplicaciones de Docker multicontanedor. La configuración de los servicios de la aplicación se realiza a través del fichero 'docker-compose.yml'. A partir de este fichero y ejecutando los comandos correspondientes se crearán y ejecutarán los servicios configurados. Es una herramienta útil para los entornos de desarrollo y test.

  El uso de Docker Compose se resume en tres pasos: La definición del entorno de la app con un Dockerfile para poderla reproducir desde cualquier lugar, definición de los servicios en el fichero de 'docker-compose.yml' de manera que puedan corren en un entorno aislado y, por último, ejecución del comando 'docker-compose up' que arrancará la aplicación al completo.

2.1.6 Docker Hub

Docker Hub es un servicio en la nube que permite a los usuarios compartir las imágenes Docker construidas, subiéndolas para ello al repositorio creado por los usuarios, del mismo modo que se sube el código a los repositorios de Github.

Además de permitir a un usuario a subir imágenes, también permite hacer uso de las imágenes creadas por otros usuarios o las imágenes oficiales de Postgres, Redis, Mysql, Ubuntu, entre otros proyectos.

El archivo Dockerfile con el que construimos una imagen podemos hospedarlo en un repositorio de GitHub y hacer que Docker Hub lo obtenga para construir la imagen. Esta opción se configura enlazando el repositorio de Github con la cuenta de Docker Hub. La otra manera de crear una imagen Docker y subirla al Docker Hub es mediante la consola de comandos construyendo la imagen a partir de un Dockerfile, creando un tag, haciendo login en la cuenta de DockerHub y realizando por último el *push* imagen.

Docker Hub ofrece repositorios públicos en los que colocar las imágenes que cualquier otro usuario puede acceder y usar o repositorios privados con cierto coste según el número de repositorios privados, siendo el primer repositorio privado gratuito.

2.1.7 Docker frente a otras tecnologías de contenerización

El mecanismo que Docker utiliza forma parte del kernel de Linux, no es algo desarrollado por ellos, con lo que, éste y los demás sistemas de contenerización basados en Linux se basan en dos elementos fundamentalmente, 'cgroups' y 'namespaces' para crear y aislar los contenedores.
Por lo tanto, cuando se compara Docker con otra tecnología de contenerización, lo que la diferencia del resto es que presenta una API muy bien documentada, un formato de imágenes y una gran comunidad. Estos tres elementos se encuentran en una fase más madura que otras tecnologías como Rocket (rkt), que presenta una comunidad menos activa.

Rocket (rkt) surgió en 2014 como alternativa a Docker debido a algunas vulnerabilidades encontradas en el mismo e aquella época. Docker hasta la versión 1.8 no tenía forma de autenticar el servidor de una imagen, con lo que sería posible que un atacante pudiera cambiar una imagen por otra infectada por malware. A partir de la versión 1.8 se añadió un nuevo complemento denominado 'Docker Content Trust' para que automáticamente sea verificada la firma de la persona que lo publica. A diferencia de Docker, en Rocket la verificación de la autenticidad se realiza por defecto, de forma que, cuando se descarga la imagen se comprueba la firma y se comprueba si ha sido adulterada.

Por otro lado, Docker se ejecuta con privilegios de super usuario (root) creando los nuevos contenedores y sus subprocesos en ese modo. Por lo tanto, una vulnerabilidad en el contenedor puede permitir que un atacante acceda al servidor con permisos de super usuario. Por ello, Docker recomienda ejecutar los contenedores con SELinux o AppArmor. Rocket por su parte no crea los contenedores desde un proceso con privilegios de root, de manera que, aunque un contenedor presente vulnerabilidades el atacante no podrá entrar con privilegios de super usuario.

Open Container Initiative (OCI) fue creada en Junio del 2015 por Docker y otros líderes en la industria de los contenedores para promover los estándares en el formato y ejecución de los contenedores.

2.1.8 Plataformas para el soporte de contenedores

Otro componente necesario en la construcción de un grupo de contenedores es el planificador. Los planificadores son los responsables de iniciar los contenedores en los host disponibles, cargan los contenedores en los host pertinentes e inician, detienen y administran el ciclo de vida del proceso. Debido a que el planificador debe interactuar con cada host en el grupo, las funciones del administrador de grupo típicamente están incorporadas. Estas permiten al planificador obtener información acerca de los miembros y realizar tareas administrativas. La orquestación en este contexto generalmente se refiere a la combinación de un contenedor y la planificación de administración del host.

Algunos de los proyectos más populares que funcionan como planificadores y herramientas de gestión, a parte de los proporcionados por la tecnología Docker (Swarm y Compose) son Kurbenetes y Mesos.

Kubernetes

Kubernetes es una plataforma de código abierto que automatiza las operaciones con contenedores de Linux. Elimina la mayoría de los procesos manuales involucrados en el despliegue y escalado de aplicaciones construidas con contenedores. Kubernetes ayuda a gestionar de manera fácil y eficiente los clusters. Estos hosts pueden estar distribuidos en nubes privadas, públicas o híbridas.

Fue originalmente diseñado y desarrollado por ingenieros de Google, siendo Google uno de los primeros contribuidores a la tecnología de contenedores de Linux. Google genera más de dos billones de despliegues de contenedores en una semana, ejecutándose sobre la plataforma Borg, la cual fue la predecesora de Kubernetes.

Kubernetes proporciona las herramientas para orquestar y manejar las capacidades requeridas para el despliegue de contenedores y su escalado para las cargas de trabajo presentes en los cluster.

La orquestación de Kubernetes (figura 2.3) permite construir servicios de aplicaciones que contienen múltiples contenedores, organizando dichos contenedores a través del cluster, escalándolos y controlando la salud de los mismos a lo largo del tiempo.

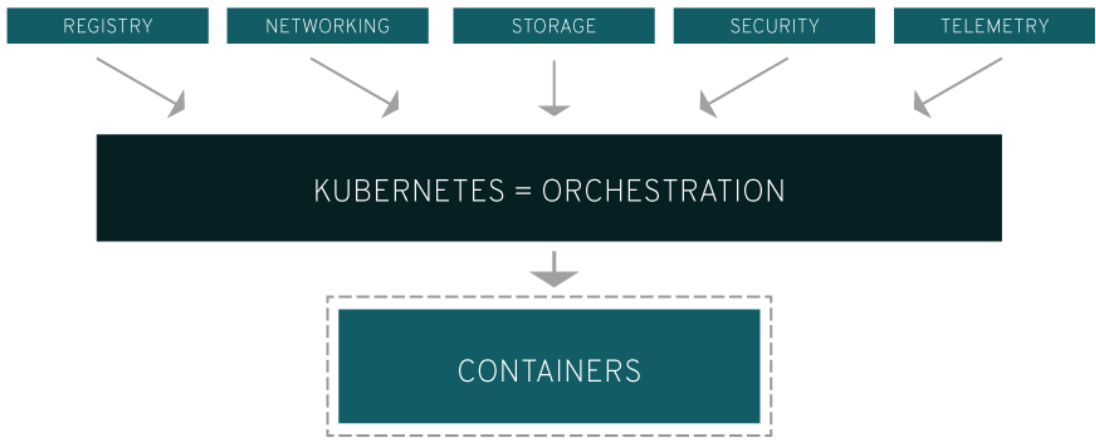

Figura 2.3 Diagrama que muestra la forma en la que trabaja la plataforma Kubernetes. [12]

La agrupación de contenedores añade una capa de abstracción que ayuda a organizar la carga de trabajo, proporcionando a los contenedores los servicios necesarios como red y almacenamiento. Otras componentes de Kubernetes ayudan a balancear la carga a lo largo de las agrupaciones de contenedores para asegurar que se tenga el número correcto de contenedores en ejecución para dar soporte a la carga de trabajo. Con la implementación correcta de Kubernetes se pueden orquestar toda la infraestructura de contenedores.

La principal ventaja de usar Kubernetes es que proporciona una plataforma para organizar y ejecutar contenedores sobre clusters en máquina físicas o virtuales, permitiendo que se pueda implementar en un entorno de producción.

Kubernetes permite:

- Orquestar contenedores a lo largo de múltiples hosts y optimizar el uso del hardware.
- Controlar y automatizar el despliegue de aplicaciones y actualizaciones.
- Añadir y montar almacenamiento para ejecutar aplicaciones con estados.

- Escalar aplicaciones contenerizadas y sus recursos en el momento.

- Realizar el 'Health Check' y la auto reparación de la aplicaciones con auto-restart, auto-replicación y autoscaling.

Mesos

Apache Mesos es un software de código abierto para la gestión de clusters diseñado para escalar clusters de amplias dimensiones, de 100 a 1000 hosts. Mesos da soporte a múltiples tipos de carga de trabajo como las tareas Hadoop, aplicaciones en la nube nativas, etc. Su arquitectura está diseñada para que el sistema sea resistente y dar soporte a la alta disponibilidad.

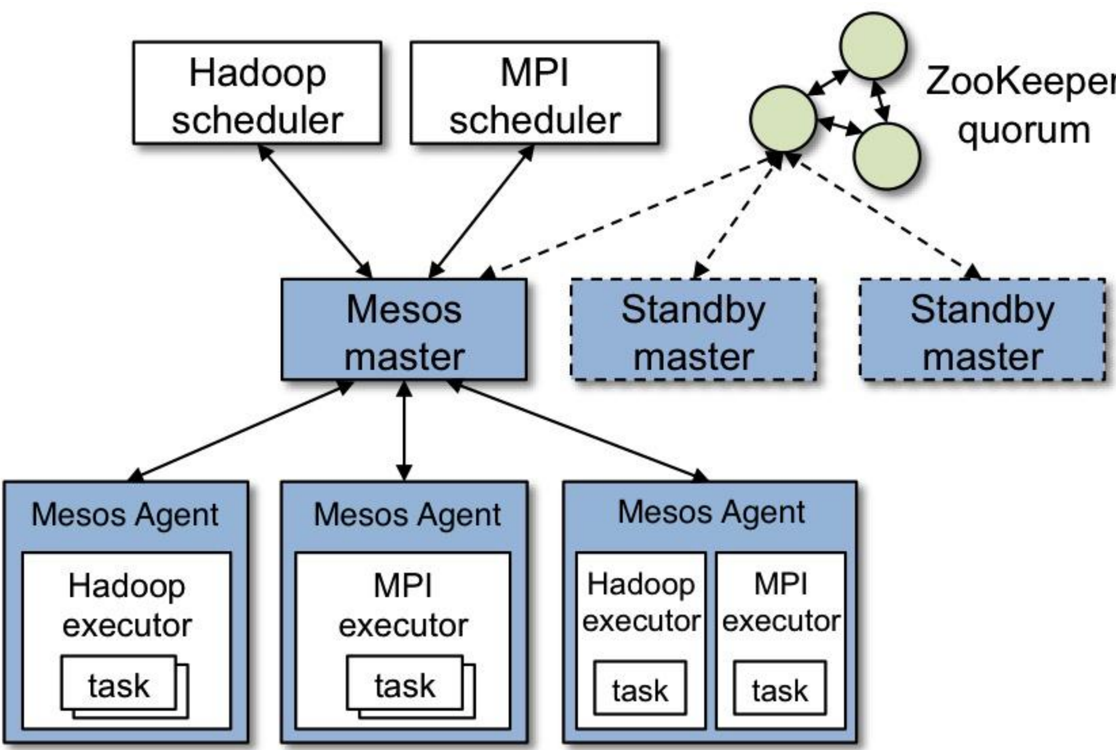

Figura 2.4. Ejecución del cluster Mesos sobre el framework Marathon. [13]

Los principales componentes (figura 2.4) del cluster de Mesos son:

- Nodos agentes: Responsables de ejecutar las tareas. Todos los agentes entregan una lista de sus recursos disponibles al máster.
- Máster: Responsable de enviar las tareas a los agentes. Mantiene la lista de recursos disponibles. Decide qué recursos ofrecer basándose en la estrategia de distribución. En

una configuración típica habrán más agentes en fase de espera para el caso en que el actual máster falle.

- ZooKeeper - Se utiliza en las elecciones y para averiguar la dirección del máster actual. Se ejecutan varias instancias de ZooKeeper para asegurar la disponibilidad y gestionar los cambios.
- Frameworks - Se coordinan con el máster para organizar las tareas en los nodos agentes. Se componen de dos partes: el proceso ejecutor que se ejecuta sobre los agentes y controla las tareas en ejecución, y el organizador que se registra con el máster y selecciona qué recursos usar basándose en las ofertas del máster.

Existen múltiples frameworks que se ejecutan sobre el cluster de Mesos ya que para registrar un trabajo no se interactúa directamente con Mesos sino que tienen que interactuar con un framework como intermediario.

En la figura 2.4, el cluster de Mesos se ejecuta junto con Marathon, un framework que actúa como organizador. Este organizador usa ZooKeeper para localizar el máster a quien enviar las tareas.

Marathon está diseñado para comenzar, monitorizar y escalar aplicaciones de larga ejecución, incluyendo aplicaciones nativas en la nube. Los clientes interactúan con Marathon a través de una REST API. Otros complementos incluyen soporte al control de la salud del cluster y un streaming de los eventos que puede ser usado para integrarlo con los balanceadores de carga o para el análisis de las métricas.

2.2 Proveedores de Infraestructura como Servicio (IaaS)

Los proveedores de Infraestructuras como Servicio (IaaS, Infrastructures as a service) forman parte de los modelos fundamentales en el campo de la computación en la nube junto con la Plataforma como Servicio (PaaS, *Platform as a Service)* y el de Software como Servicio (SaaS, *Software as a Service).*

El IaaS proporciona acceso a recursos informáticos situados en un entorno virtualizado a través de una conexión pública. En el caso de IaaS los recursos informáticos que se proveen consisten en hardware virtualizado o infraestructuras 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.

El conjunto de recursos de hardware disponibles procede de multitud de servidores y redes, generalmente distribuidos entre numerosos centros de datos, de cuyo mantenimiento se encarga el proveedor del servicio en la nube. El cliente, por su parte, obtiene acceso a los componentes virtualizados para construir con ellos su propia plataforma informática.

El modelo IaaS coincide con las otras dos modalidades de hosting en la nube en que puede ser utilizado por los clientes para crear soluciones informáticas económicas y fáciles de ampliar, cuya complejidad y costes asociados a la administración del hardware subyacente se externaliza al proveedor del servicio en la nube.

A continuación se describen aplicaciones concretas del modelo IaaS:

- Infraestructura corporativa: Las redes internas de la empresa como las clouds privadas y las redes locales virtuales que utilizan recursos de red y de servidores agrupados en un repertorio común donde se puede almacenar los datos y ejecutar las aplicaciones que se necesite. Las empresas en crecimiento pueden ampliar su infraestructura a medida que aumente su volumen de actividad, mientras que las clouds privadas permiten proteger el almacenamiento y transferencia de los datos delicados que algunas empresas necesitan manejar.
- Una web alojada en una plataforma cloud puede beneficiarse de la redundancia que aporta la gigantesca escala de la red de servidores físicos y su escalabilidad en función de la demanda para afrontar el tráfico en su web.
- Virtual Data Centers (VDC): Una red virtualizada de servidores virtuales interconectados que puede utilizarse para ofrecer funcionalidades para implementar la infraestructura informática.

Entre las ventajas características de una implementación basada en el modelo de IaaS se encuentran:

- La escalabilidad, los recursos se encuentran disponibles a medida que se vayan necesitando con lo que desaparecen los tiempos de espera a la hora de ampliar la capacidad y no se desaprovecha la capacidad que no se vaya a utilizar.
- El hardware físico subyacente sobre el que funciona el servicio IaaS es configurado y mantenido por el proveedor del servicio, por lo que evita tener que dedicar tiempo y dinero a realizar esa instalación en el lado del cliente.
- El servicio está accesible a demanda y el cliente sólo paga por los recursos que realmente utiliza.
- Se puede acceder al servicio desde cualquier lugar, siempre y cuando se disponga de una conexión a internet y el protocolo de seguridad del servicio lo permita.
- Seguridad física en los centros de datos, los servicios disponibles a través de una infraestructura cloud pública o en privadas están alojadas externamente en las instalaciones del proveedor del servicio, con lo que se benefician de la seguridad física de la cual disfrutan los servidores alojados dentro de un centro de datos
- Si falla un servidor o un conmutador, el servicio global no se verá afectado, gracias a la gran cantidad restante de recursos de hardware y configuraciones redundantes. En muchos servicios, incluso la caída de un centro de datos entero, y no digamos de un solo servidor, no afecta en absoluto al funcionamiento del servicio IaaS.

2.2.1 Amazon Web Services (AWS)

Amazon Web Services ofrece un amplio conjunto de productos basados en la nube que incluye computación, almacenamiento, bases de datos, analíticas, red, herramientas para desarrolladores, herramientas para la gestión, seguridad y aplicaciones de empresas. Estos servicios ayudan a las organizaciones a actuar más rápido, a menor coste y proporcionando servicios de escalado.

AWS está situado en 11 Regiones geográficas: EE.UU. Este (Norte de Virginia), EE.UU. Oeste (Norte de California), EE.UU. Oeste (Oregón), AWS GovCloud (EE.UU.), São Paulo (Brasil), Irlanda, Singapur, Tokio y Sydney. Cada región está totalmente contenida dentro de un solo país y todos sus datos y servicios permanecen dentro de la región designada.

Cada región tiene múltiples "zonas de disponibilidad", que son los diferentes centros de datos que proporcionan servicios de AWS. Las zonas de disponibilidad están aisladas unas de otras

para evitar la propagación de cortes entre las zonas. Varios servicios operan a través de zonas de disponibilidad, mientras que otros pueden estar configurados para reproducirse a través de zonas para extender la demanda y evitar el tiempo de inactividad ante fallos.

Se puede acceder a sus servicios a través de la consola de AWS (AWS Management Console), la interface por línea de comandos (CLI, Command Line Interface), o a través del Software Development Kits (SDKs).

- AWS Management Console se accede a través de la web (https://aws.amazon.com/console/) y ofrece una interfaz de usuario sencilla e intuitiva. También se puede acceder a través de la aplicación móvil (https://aws.amazon.com/console/mobile/).
- Command Line Interface permite gestionar los recursos en AWS a través de la línea de comandos descargando para ello la herramienta desde https://aws.amazon.com/cli/. Permite controlar múltiples servicios de AWS desde la línea de comandos y automatizarlos por medio de scripts.
- Software Development Kits simplifica el uso de los servicios de AWS a través de una API ajustada al lenguaje de programación que se prefiera. Se puede instalar desde la página https://aws.amazon.com/tools/.

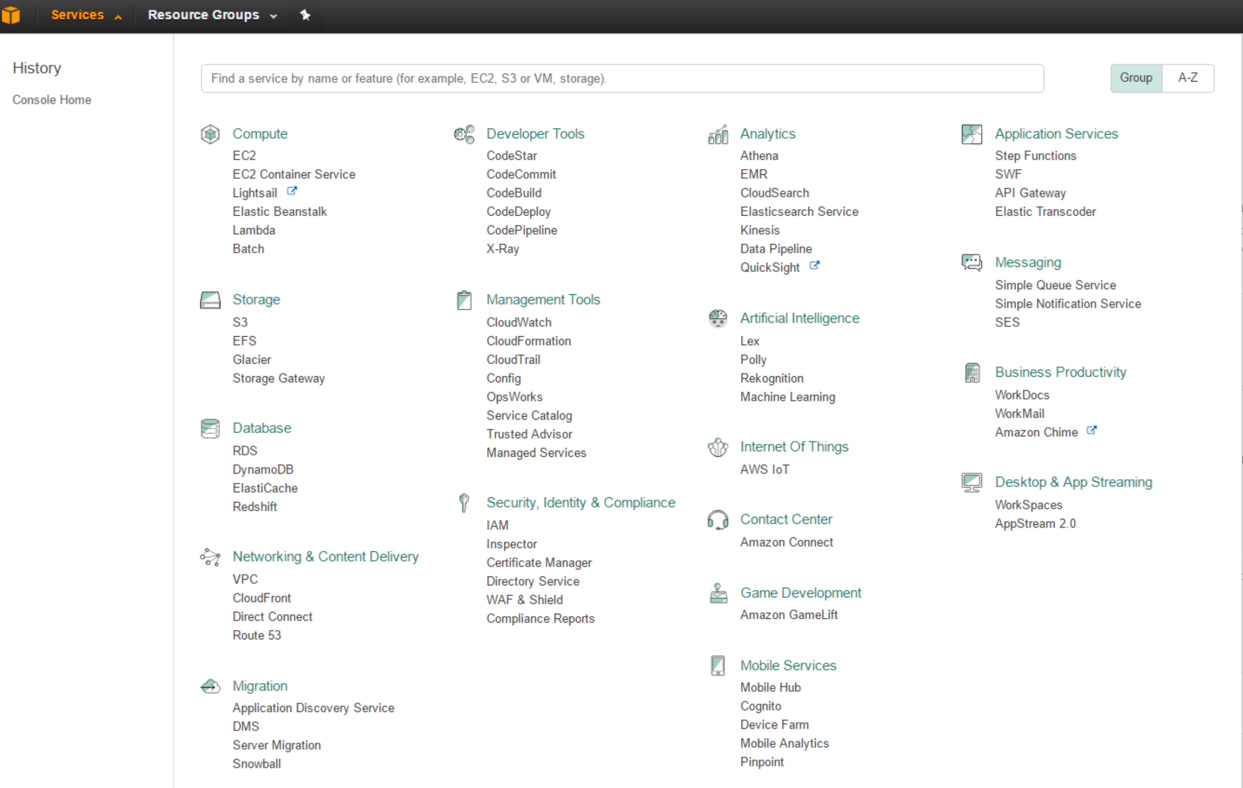

Figura 2.5 Servicios disponibles en la consola de AWS

Los servicios de AWS utilizados en este TFM son los siguientes:

- **AWS Identity and Access Management (IAM):** Permite controlar los accesos de los usuarios a los recursos y servicios de la consola de AWS. Se pueden crear y gestionar usuarios y grupos y usar permisos para permitir o denegar el acceso a los recursos.

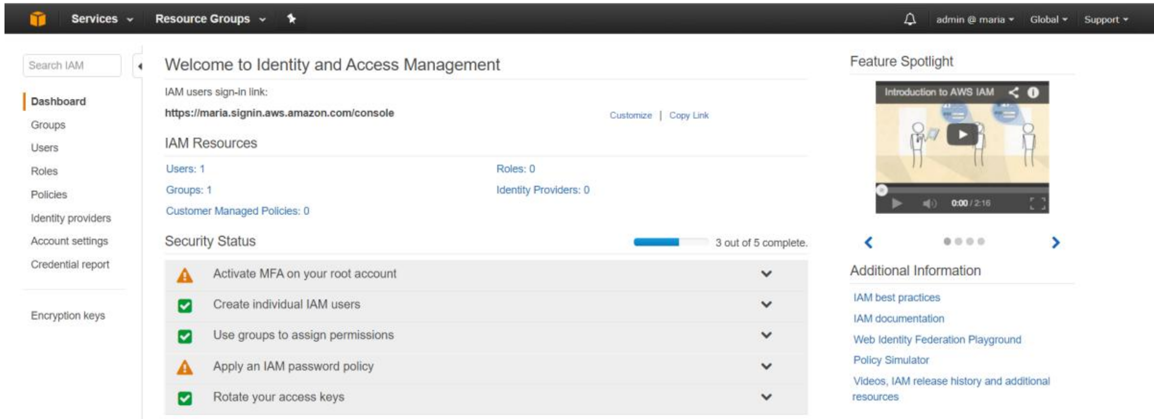

Figura 2.6 IAM Dashboard.

De esta manera se aporta una mayora seguridad ya que se conoce quienes son los usuarios responsables de desempeñar cada tarea y a qué secciones tienen acceso. Los usuarios son otorgados credenciales de seguridad permanentes o temporales para que tengan accesos a recursos en momentos determinados (claves de acceso, contraseñas). Los usuarios podrán llevar a cabo únicamente las operaciones a las que se les haya concedido permiso.

- **Amazon EC2 (Elastic Compute Cloud)**: Es un servicio web que proporciona capacidad de cómputo en la nube de manera segura. Debido a que está controlado por el servicio web de la API, la aplicación puede ser escalada de forma automática acorde a sus necesidades. Está diseñado para facilitar el escalado a los desarrolladores, permitiendo incrementar o reducir la capacidad en minutos.

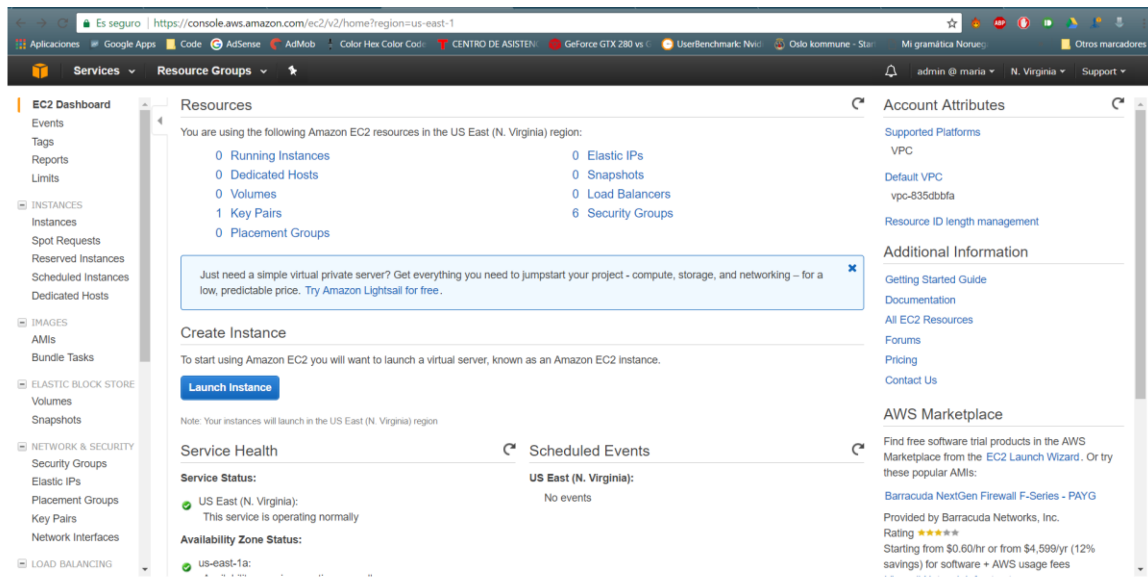

Figura 2.7 EC2 Dashboard

Se tiene completo control de las instancias pudiendo acceder a las mismas con permisos de administrador. Se puede elegir entre múltiples instancias, sistemas operativos y paquetes de software para crear las máquinas.

EC2 permite configurar las instancias con la configuración de memoria, CPU, almacenamiento y tamaño de partición. El servicio de EC2 permite integrar en el mismo la mayoría de servicios de AWS como Amazon Simple Storage Service (S3, para la gestión de los volúmenes de almacenamiento), Amazon Relational Database Service

(Amazon RDS, para la gestión de base de datos) y Amazon Virtual Private Cloud (Amazon VPC, para la gestión de redes).

Además se puede aumentar o disminuir la capacidad en minutos, permitiendo crear múltiples instancias de forma simultánea. Se puede configurar para que la aplicación escale el número de instancias y recursos de forma automáticas en función de las necesidades de demanda y de la aplicación.

Se puede asegurar las conexiones a las instancias indicando a través del VPC qué instancias son accesibles a través de direcciones públicas y cuáles se acceden exclusivamente a través de redes privadas. También permite asegurar las conexiones a puertos por medio de los 'Security Groups', configurando la apertura de puertos y a qué máquinas se les otorga el acceso desde el exterior.

La ejecución de contenedores en la nube de AWS permite crear aplicaciones y servicios sólidos y escalables al aprovechar los beneficios de la misma, como la elasticidad, disponibilidad, seguridad y economía de escala.

2.3 Otras herramientas tecnológicas

2.3.1 Virtualbox

Es un hipervisor que se utiliza para ejecutar sistemas operativos en un entorno especial, llamado máquina virtual, corriendo sobre un sistema operativo ya existente. VirtualBox está en constante desarrollo y se implementan nuevas características continuamente.

Con el fin de integrar las funciones del sistema anfitrión en los sistemas huéspedes, incluyendo carpetas compartidas y portapapeles, aceleración de vídeo y un modo de integración de ventanas fluido, se proporcionan complementos huéspedes ('guest additions') para algunos sistemas operativos invitados.

3. Desarrollo

3.1 Análisis de la aplicación básica sample_app_rails_4 en Ruby on Rails

La aplicación *sample_app_rails_4* es una aplicación ejemplo sacada de '*Ruby on Rails Tutorial: Learn Web Development with Rails*' de Michael Hartl. Se trata de una aplicación básica que permite el registro de usuarios y que estos usuarios puedan crear posts a los que se podrá acceder desde la página del usuario que los ha creado. Además esta aplicación permite que un usuario pueda seguir a otros usuarios, como en una red social típica. Para ello, la aplicación consta de tres modelos: *User, Micropost, y Relationship*. Este último es el modelo usado para representar el seguimiento de un usuario a otro.

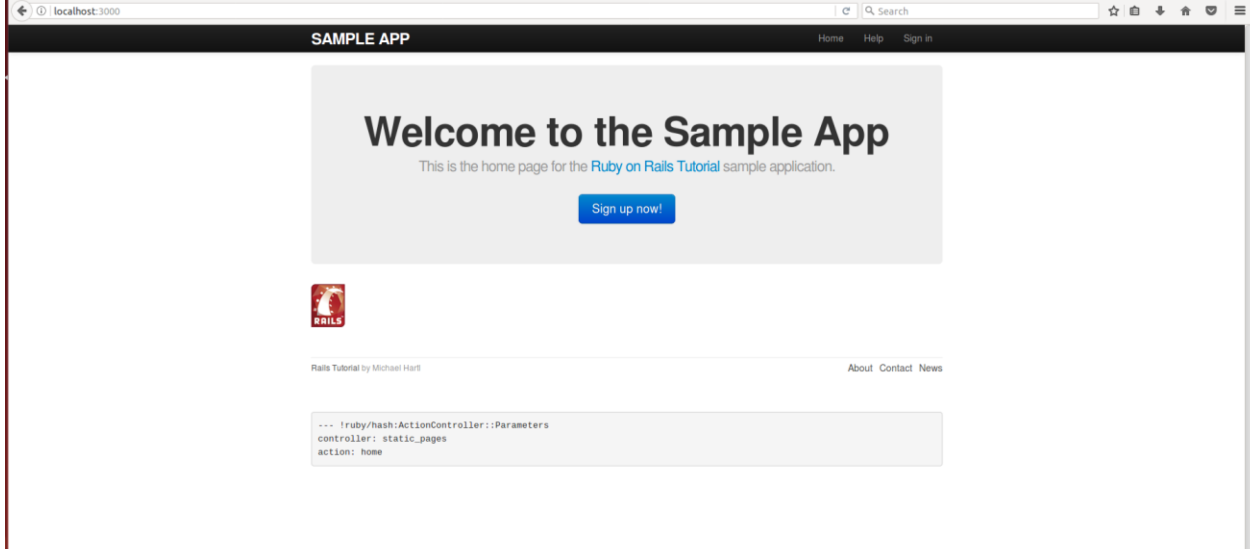

Figura 3.a Página principal aplicación *sample_app_rails_4*

Cuando se accede a la página principal de la aplicación (figura 3.a) se da la opción de iniciar sesión (margen superior derecha) o de registrarse si no se tiene una cuenta previa. Para crear una cuenta nueva se accede al botón central 'Sign up now'. En esta página se solicitan los datos básicos para la creación de la cuenta (nombre, email y password) como se muestra en la figura 3.b.

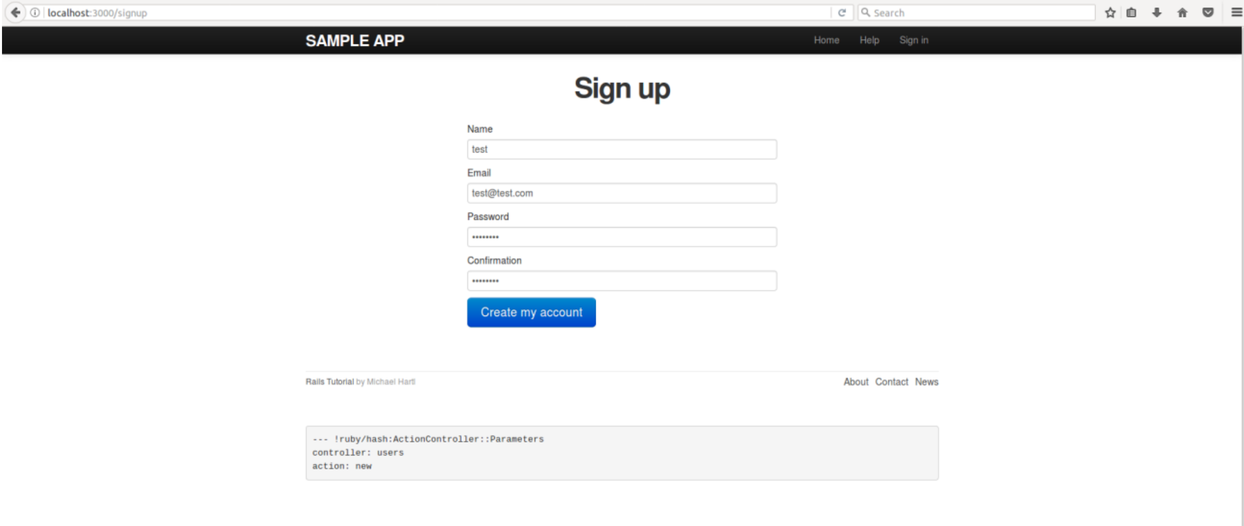

Figura 3.b Página para la creación de cuenta en la aplicación.

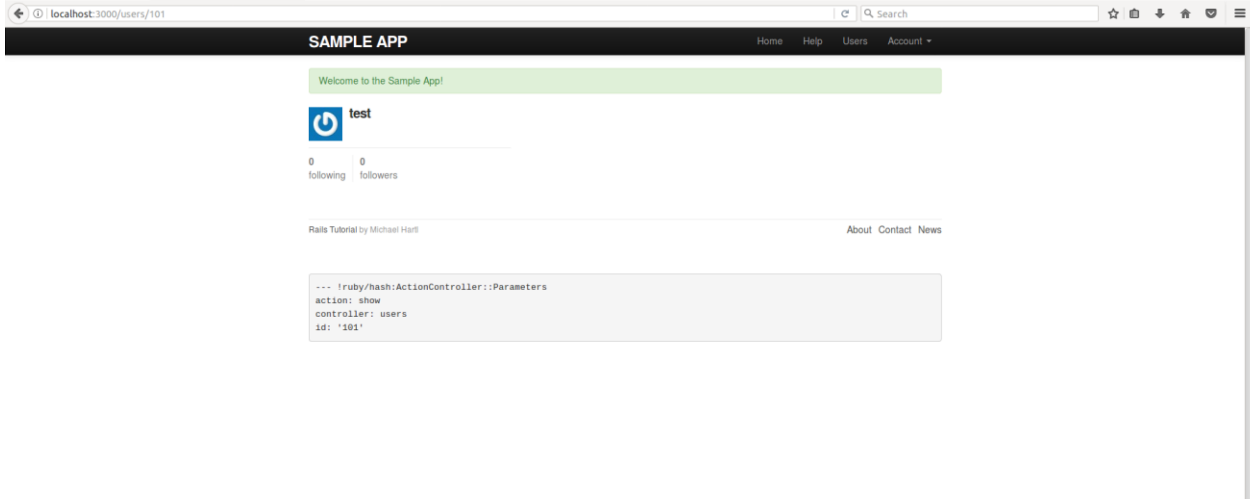

Figura 3.c Página principal del usuario test creado.

En la página del perfil del usuario (figura 3.c) se muestra el listado de microposts creados por el usuario y el número de usuarios a los que sigue y que le siguen.

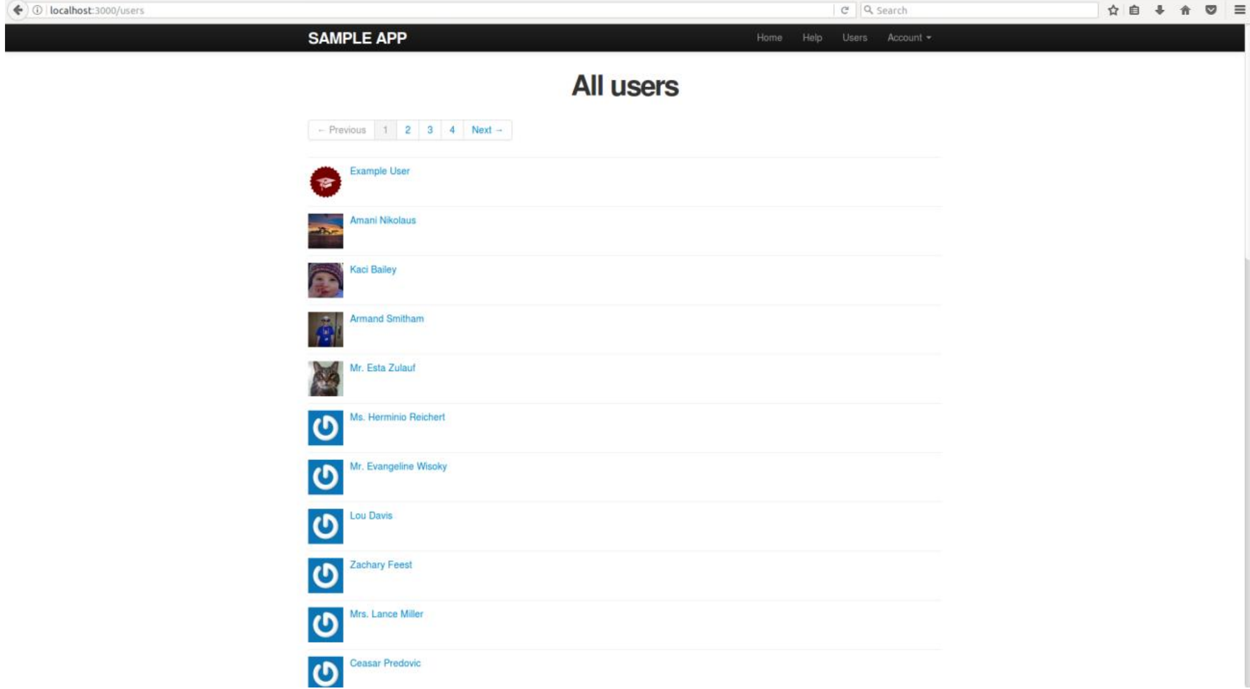

Figura 3.d Listado de usuarios

Para ver el listado de usuarios registrados en la aplicación se puede acceder a la sección 'users' como se muestra en la figura 3.d.

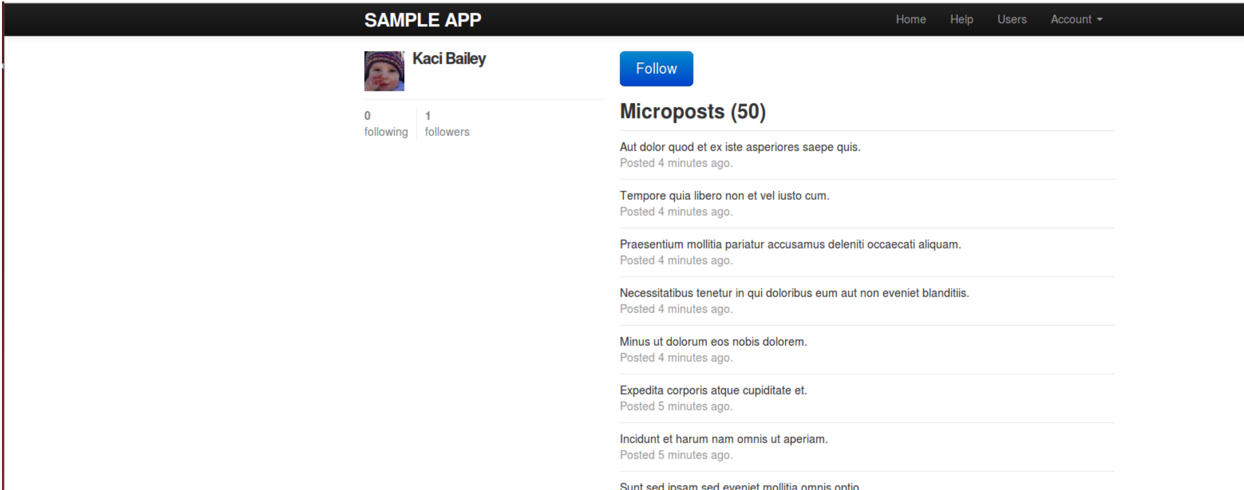

Figura 3.e Perfil de usuario con micropost añadido

Si accedemos a un usuario podremos ver el listado de post que ha creado y el número de personas que le siguen y a las que sigue (figura 3.e). Si se accede a 'following' o a 'followers' (figura 3.f) se podrá ver el listado de usuarios a los que sigue y que le siguen respectivamente.

Además se da la opción de realizar un ‘follow’ al usuario, quedando registrado en el perfil de usuario como persona a la que sigue.

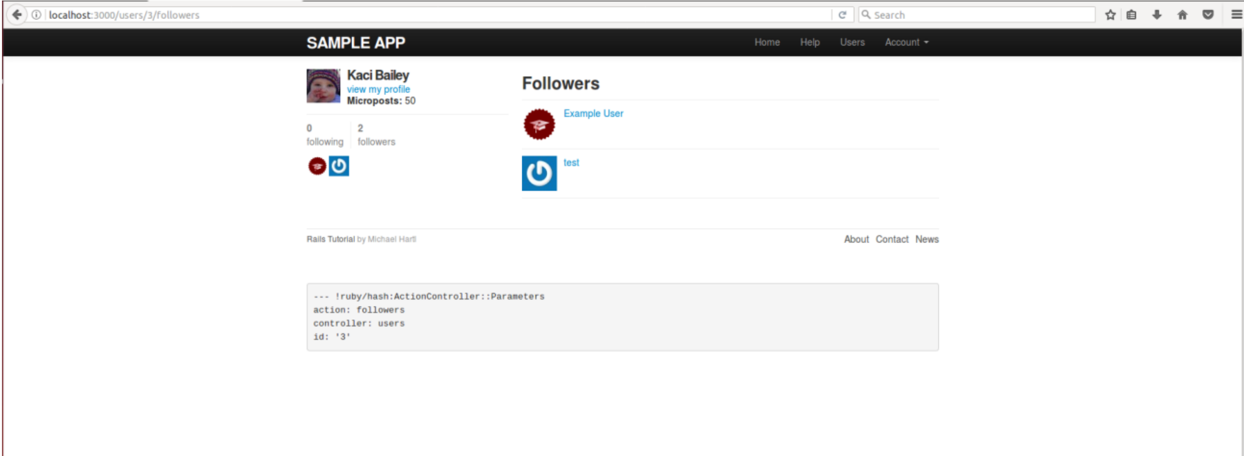

Figura 3.f Listado de seguidores del usuario 3

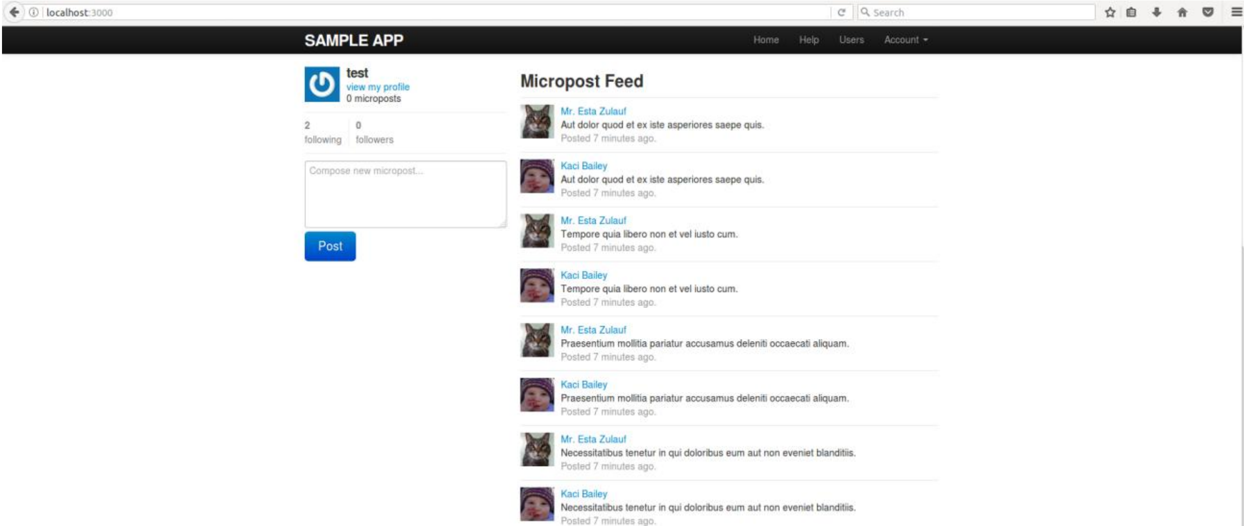

Figura 3.g Página principal de la aplicación con el usuario logueado.

Cuando se accede a la página principal de la aplicación con la sesión de usuario iniciada (figura 3.g), se muestra a la derecha el listado de post de todos los usuarios con los post más recientes en primer lugar y a la izquierda el usuario con un cuadro de textos para añadir posts.

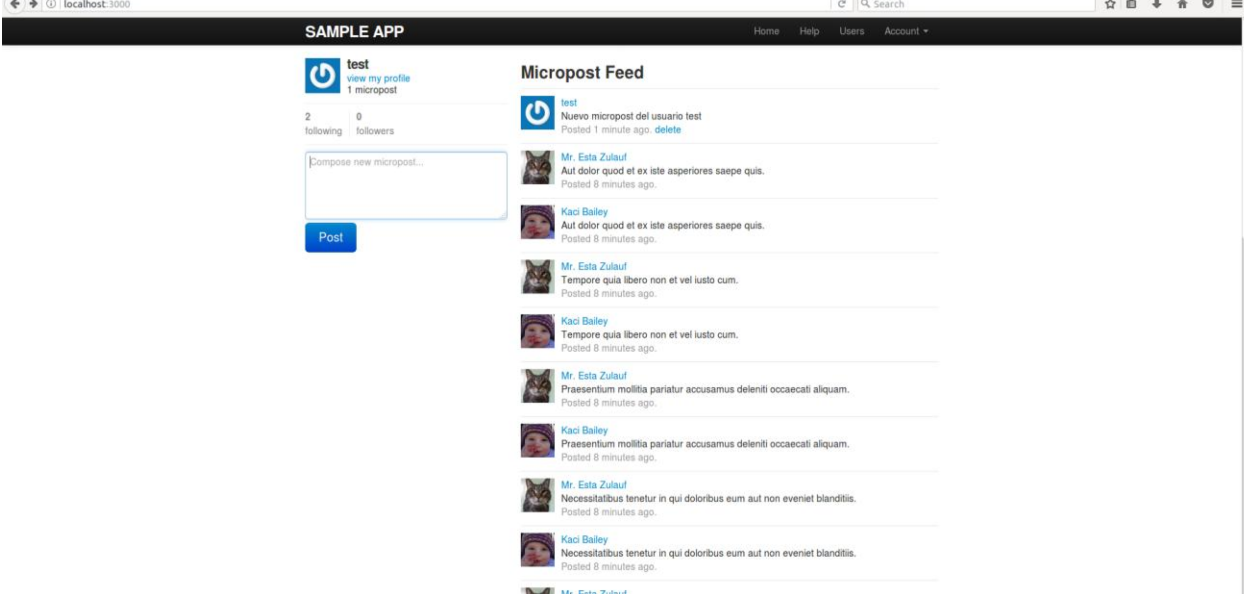

Figura 3.h Post del usuario test añadido.

Si añadimos un post y se actualiza la pantalla, se verá el post recientemente añadido en primer lugar del listado (figura 3.h). Si accedemos al perfil de nuestro usuario podremos eliminar los posts creados por el mismo.

Sample_app_rails_4 está preparada para funcionar con una base de datos Sqlite, por ello, se le realizarán modificaciones para que funciones con una base de datos Postgres, editando el fichero de configuración de base de datos y llevando a cabo un procedimiento para su despliegue en las distintas iteraciones del proyecto.

Se puede acceder a la aplicación original en https://github.com/railstutorial/sample_app_rails_4.

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

En esta primera etapa del proyecto se va a realizar el despliegue de la aplicación desde un único host, descomponiendo cada servicio en un contenedor Docker y obteniendo como resultado dos scripts, uno para lanzar el contenedor de base de datos junto con el contendor de la aplicación *sample_app_rails_4* y otro script para el lanzamiento del contenedor del servidor web Nginx.

3.2.1 Preparación del repositorio local y remoto

En primer lugar, se clonará el repositorio de *sample_app_rails_4* y se creará un repositorio nuevo para obtener la misma aplicación pero con las modificaciones necesarias para que se ajuste con las especificaciones indicadas.

```
# git clone https://github.com/railstutorial/sample_app_rails_4.git
```

Figura 3.1 Clonación del repositorio de sample_app_rails_4

A continuación, se copiará el contenido del proyecto a una nueva carpeta excluyendo el directorio de configuración del repositorio de *git*. Finalmente, se inicializa el repositorio de *git* *'mayrez/sample_app_rails_4'* y se realiza el primer *commit*.

```
# git init
# git add .
# git commit -m "first commit"
# git remote add origin https://github.com/mayrez/sample_app_rails_4.git
# git push -u origin master
```

Figura 3.2 Commit del repositorio de sample_app_rails_4 a modificar para ajustarse a los objetivos del proyecto

Las siguientes operaciones a realizar sobre el nuevo repositorio serán para la adaptación de la base de datos a una configuración con Postgres y la conversión de la aplicación en una imagen de Docker.

3.2.2 Cambio y configuración de la base de datos

Para configurar la aplicación con una base de datos Postgres crearemos un nuevo fichero *'database.yml.postgres.yml'* dentro de la carpeta *'config/'* del proyecto *'sample_app_rails_4'* con la configuración correspondiente. Modificaremos los entornos de test y desarrollo para que tomen los valores de usuario, contraseña y host a partir de variables de entorno (*POSTGRES_USER, POSTGRES_PASSWORD, POSTGRESIP*), quedando de la manera en que se muestra en la figura 3.3.

```
development:
  <<: *default
  database: sample_app_development
  username: <%= ENV['POSTGRES_USER'] %>
  password: <%= ENV['POSTGRES_PASSWORD'] %>
  host: <%= ENV['POSTGRESIP'] %>
test:
  <<: *default
  database: sample_app_test
  username: <%= ENV['POSTGRES_USER'] %>
  password: <%= ENV['POSTGRES_PASSWORD'] %>
  host: <%= ENV['POSTGRESIP'] %>
```

Figura 3.3 Modificaciones en el fichero de configuración database.yml

3.2.3 Creación de la imagen Docker de la aplicación de Ruby

Se creará una imagen Docker a partir del directorio del proyecto *'sample_app_rails_4'* y luego esta imagen será añadida al repositorio de Docker (Docker Hub). Para ello, se creará un fichero '*Dockerfile'* a partir del cual será creada la imagen Docker de la aplicación. En este fichero se indicará a partir de qué imagen la crearemos.

El contenido del '*Dockerfile'* para la creación de la imagen de la aplicación de Ruby se muestra en la figura 3.4.

```
FROM ruby:2.2.6-onbuild
RUN mkdir -p /usr/src/app
WORKDIR /usr/src/app
EXPOSE 3000
CMD ["rails", "server", "-b", "0.0.0.0"]
```

```
RUN apt-get update \
  && apt-get -y install nodejs  && rm -rf /var/lib/apt/lists/* \
RUN cp config/database.yml.postgres config/database.yml
COPY Gemfile /usr/src/app
RUN bundle install
COPY . /usr/src/app
```

Figura 3.4 Contenido del Dockerfile para construir la imagen Docker de la aplicación de Ruby

En este caso se creará a partir de una imagen Ruby ya que para que la aplicación pueda ejecutarse necesita tener instalado Ruby. Además, se indican los comandos adicionales que deben ejecutarse para configurar la aplicación e instalar sus dependencias.

Con el '*Dockerfile*' definido, se creará el *tag* del contenedor a partir del id de la imagen creada que se puede ver al ejecutar 'docker images'. Finalmente, se realizará el inicio de sesión en Docker y se hará un *push* de la imagen al repositorio.

En la figura 3.5 se describen los comandos a ejecutar para crear la imagen Docker de la aplicación y subirla al repositorio.

```
# git clone https://github.com/mayrez/sample_app_rails_4.git
# cd sample_app_rails_4
# docker build -t sample_app_image .
# docker images
# docker tag 7d9495d03763 mayrez/sample_app_image:0.5
# docker login
# docker push mayrez/sample_app_image:0.5
```

Figura 3.5 Inicio de sesión en Docker desde la consola y *push* de la imagen al repositorio de imágenes de Docker

3.2.4 Cambio y configuración del servidor web

Para configurar el servidor web de Nginx crearemos una imagen Docker del mismo. Para ello, se creará un fichero '*Dockerfile*' a partir del cual será creada la imagen Docker. En este fichero se indicará a partir de qué imagen la crearemos y la acción a realizar en su arranque. El contenido del '*Dockerfile*' se muestra en la figura 3.6.

```
FROM nginx
COPY ./etc/nginx/conf.d/default.conf  /etc/nginx/conf.d/default.conf
```

Figura 3.6 Contenido fichero '*Dockerfile*' de servidor web Nginx

En el '*Dockerfile*' se especifica que se copie el fichero de configuración que hemos creado en la ruta especificada del directorio del proyecto en la ruta indicada del contenedor.

El contenido del fichero *'default.conf'* se muestra en la figura 3.7. En esta se configura el servidor Nginx para que se redirecciones la puerto 80 del mismo las peticiones recibidas por la aplicación de Ruby a través del puerto 3000.

```
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:3000;
       break;
    }
}
```

Figura 3.7 Fichero de configuración Nginx '*/etc/conf.d/default.conf*'

Con el *'Dockerfile'* y el fichero de configuración definidos, se procede con la creación de la imagen de la aplicación (figura 3.8).

```
# git clone https://bitbucket.org/tftdp/docker_project
# cd nginx
# docker build -t some-nginx .
```

Figura 3.8 Creación de imagen Docker en local del servidor Nginx

Con la imagen creada localmente se crea el *tag* del contenedor (figura 3.9) a partir del id de la imagen que se puede ver al ejecutar 'docker images'.

```
# docker tag 7d9495d03763 mayrez/some-nginx:0.1
```

Figura 3.9 Creación del *tag* de la imagen Nginx creada

Finalmente, se inicia sesión en Docker y se realiza un *push* de la imagen al repositorio (figura 3.10).

```
# docker login
# docker push mayrez/some-nginx:0.1
```

Figura 3.10 Inicio de sesión en Docker desde la consola y *push* de la imagen al repositorio de imágenes de Docker

3.2.5 Configuración automatizada para la creación de los contenedores

El despliegue de la aplicación *sample_app_rails_4* se llevará a cabo ejecutando 3 comandos 'docker run...' en el shell: uno para el servidor web, otro para el servidor de aplicaciones y otro para la base de datos.

En primer lugar, se creará el host desde el que se realizará el despliegue de la aplicación:

```
# docker-machine create --virtualbox-disk-size "32768"  -d virtualbox machine1
#  eval $(docker-machine env machine1)
```

Figura 3.11 Creación de la máquina virtual que hará de host de la aplicación y definición en el shell del entorno de la MV a utilizar

Para la configuración de la aplicación de la base de datos y Ruby se ha definido el fichero *'tarea1-db-Ruby.sh'* (figura 3.12), en el que, en primer lugar, se declaran las variables de entorno para lanzar la base de datos Postgres y, a continuación, se lanza la base de datos y la aplicación de Ruby mediante dos comandos 'docker run'.

```
#!/bin/bash
export POSTGRES_USER=postgres
export POSTGRES_PASSWORD=mysecretpassword
export POSTGRESIP=$(docker-machine ip machine1)
docker run --name some-postgres -p 5432:5432 -e POSTGRES_USER=$POSTGRES_USER \
 -e POSTGRES_PASSWORD=$POSTGRES_PASSWORD -d postgres
```

```
docker run -it -e POSTGRESIP=$POSTGRESIP -e POSTGRES_USER=$POSTGRES_USER \
-e  POSTGRES_PASSWORD=$POSTGRES_PASSWORD  -p 3000:3000 \
--name some-Ruby --link some-postgres:db  mayrez/sample_app_image:0.5 /bin/bash -c \
        'rake db:setup  &&
        rake db:migrate &&
        rake db:populate &&
        rails server'
```

Figura 3.12 Contenido fichero '*tarea1-db-Ruby.sh*' con el que se lanza el contenedor de la base de datos y el contenedor de la aplicación

Para el despliegue de la base de datos y la aplicación se ejecuta el script *'tarea1-db-Ruby.sh'* (figura 3.13).

```
# sh tarea1-db-Ruby.sh
```

Figura 3.13 Ejecución del fichero '*tarea1-db-Ruby.sh*'

Como resultado del comando anterior tendremos la aplicación ejecutándose, lanzada a través del contenedor *some-ruby* y recibiendo las peticiones a través del puerto 3000.

Con la aplicación de Ruby en ejecución lanzaremos el contenedor *some-nginx* con el fichero *'tarea1-nginx.sh'*, para que actúe como proxy de la aplicación y redireccionar el puerto por el que se reciben las peticiones de la aplicación al puerto 8080 de la máquina virtual en la que se ejecutan los contenedores.

```
#!/bin/bash
docker run --name some-nginx --link some-ruby:app -p 8080:80 -d mayrez/some-nginx:0.1
```

Figura 3.14 Contenido fichero *tarea1-nginx.sh*

Como se aprecia en la figura 3.14, el contenedor conectará con la aplicación en ejecución *some-ruby* ('*--link some-Ruby:app*') y se indicará los puertos a través de los cuales se encontrará el servidor en escucha, realizando el binding del puerto 80 del contenedor, al que se habían redireccionado las peticiones realizadas a la aplicación por el puerto 3000 en el fichero *'default.conf'*, con el puerto del host 8080).

Para lanzar el servidor Nginx se ejecuta el comando de la figura 3.15.

```
# sh tarea1-nginx.sh
```

Figura 3.15 Ejecución del fichero '*tarea1-nginx.sh*'

Se puede comprobar el funcionamiento de la aplicación realizando un 'curl' a la máquina que actúa como host en el puerto 8080

```
# curl $(docker-machine ip machine1):8080/public
```

Figura 3.16 Comprobación del funcionamiento de la aplicación con 'curl'

También se puede comprobar en el navegador del host accediendo a través de la ip de la máquina *machine1*.

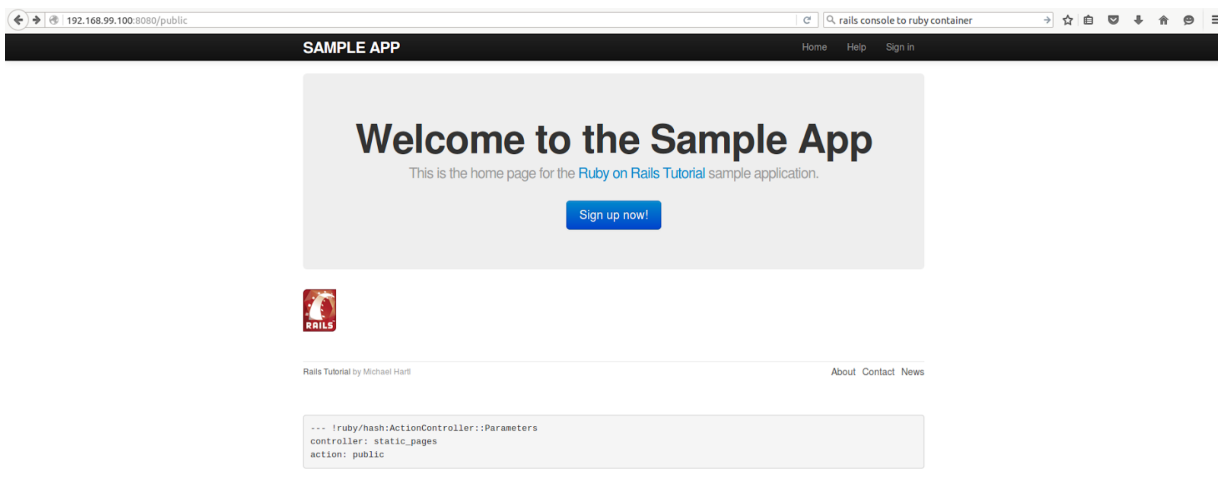

Figura 3.17 Acceso a la aplicación *sample_app_rails_4* desde el host a través de la ip de *machine1*

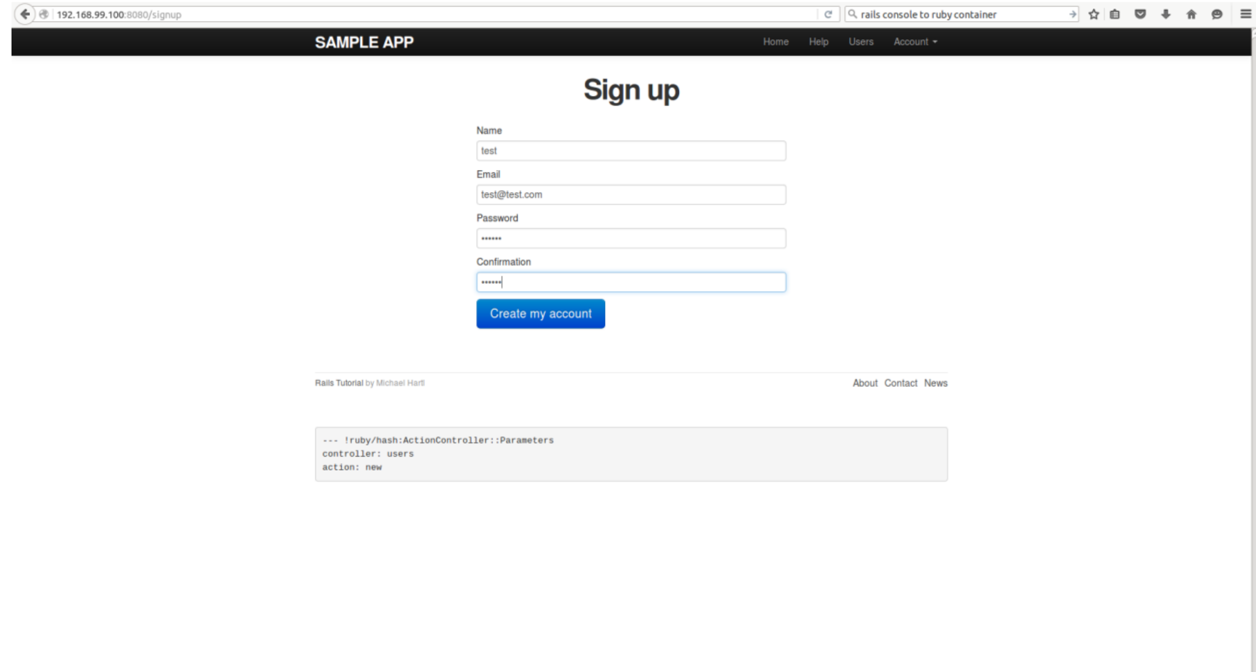

Figura 3.18 Acceso a la pantalla de *sign up*

Para comprobar el correcto funcionamiento de la base de datos ejecutaremos el ‘rails console’ y se realizarán operaciones con la misma. Para obtener el id del contenedor de Ruby y poder acceder a su consola, ejecutamos 'docker ps′ para obtener el id del contenedor. Una vez obtenido el id del contenedor ejecutaremos 'docker exec -it idContenedor bash' y accederemos a la consola del mismo.

```
# docker exec -it ed16a03dee8d bash
root@ed16a03dee8d:/usr/src/app# rails console
Loading development environment (Rails 4.0.8)
irb(main):001:0> User.first
 User Load (1.0ms)  SELECT "users".* FROM "users" ORDER BY "users"."id"
ASC
```

Figura 3.19 Acceso al bash del contenedor y ejecución del comando ‘rails console’ para comprobar el funcionamiento de la base de datos, con la ejecución de ‘User.first’

```
naox@naox-SATELLITE-P50-A-12Z:~/Desktop/docker_swarm_project/docker_project$ docker ps
CONTAINER ID        IMAGE                        COMMAND                  CREATED             STATUS              PORTS                          NAMES
792a18db9c49        mayrez/some-nginx:0.1        "nginx -g 'daemon off"   10 minutes ago      Up 10 minutes       443/tcp, 0.0.0.0:8080->80/tcp  some-nginx
ed16a03dee8d        mayrez/sample_app_image:0.5  "/bin/bash -c 'rake d"   15 minutes ago      Up 15 minutes       0.0.0.0:3000->3000/tcp         some-ruby
4481a7294ec1        postgres                     "docker-entrypoint.sh"   16 minutes ago      Up 15 minutes       0.0.0.0:5432->5432/tcp         some-postgres
naox@naox-SATELLITE-P50-A-12Z:~/Desktop/docker_swarm_project/docker_project$ docker exec -it ed16a03dee8d bash
root@ed16a03dee8d:/usr/src/app# rails console
Loading development environment (Rails 4.0.8)
irb(main):001:0> User.first
  User Load (1.0ms)  SELECT "users".* FROM "users" ORDER BY "users"."id" ASC LIMIT 1
=> #<User id: 1, name: "Example User", email: "example@railstutorial.org", created_at: "2017-04-24 16:12:45", updated_at: "2017-04-24 16:12:45", password_digest: "$2a$10$jshXYIy5vQNagWEJcsSMfe23y.oSrvXv.w/
RwEyfc89V...", remember_token: "ac60197bf21ccf2b2b60a3d4136a8db94977c24e", admin: true>
irb(main):002:0> User.all
  User Load (1.1ms)  SELECT "users".* FROM "users"
=> #<ActiveRecord::Relation [#<User id: 1, name: "Example User", email: "example@railstutorial.org", created_at: "2017-04-24 16:12:45", updated_at: "2017-04-24 16:12:45", password_digest: "$2a$10$jshXYIy5v
QNagWEJcsSMfe23y.oSrvXv.w/RwEyfc89V...", remember_token: "ac60197bf21ccf2b2b60a3d4136a8db94977c24e", admin: true>, #<User id: 2, name: "Mrs. Isabell Morissette", email: "example-1@railstutorial.org", creat
ed_at: "2017-04-24 16:12:48", updated_at: "2017-04-24 16:12:48", password_digest: "$2a$10$iTPVa9yeWeClhaOYb43rMOPZ6fy39tvXsPbSl0oQw2nx...", remember_token: "5f91694178c21b08918abc1dbe3d4968ce3c4bc5", admin
: nil>, #<User id: 3, name: "Danial Green", email: "example-2@railstutorial.org", created_at: "2017-04-24 16:12:48", updated_at: "2017-04-24 16:12:48", password_digest: "$2a$10$.J.kwil8TzihaMM2c.sSGeSjWV/S
J.eVuIGinW6s9yIL...", remember_token: "def56ecaf965b1d06491892d629f1526b1f079ad", admin: nil>, #<User id: 4, name: "Roel Auer", email: "example-3@railstutorial.org", created_at: "2017-04-24 16:12:48", upda
ted_at: "2017-04-24 16:12:48", password_digest: "$2a$10$kPZp0l75W9cwiYqJe64nuuMmwNZFQjNI/0D6m4O/jL2Y...", remember_token: "5a96e1308cf8f50ffa73f2c138e4ff4a66305892", admin: nil>, #<User id: 5, name: "Taylo
r Heller IV", email: "example-4@railstutorial.org", created_at: "2017-04-24 16:12:48", updated_at: "2017-04-24 16:12:48", password_digest: "$2a$10$vjxMDJ2ua1mjjCyRTuKpGOKlznCUaysEEepHxKYJjn1Y...", remember
_token: "96bef0cd19d8afa7583133efa76d0651c9c91c30", admin: nil>, #<User id: 6, name: "Florence Schumm", email: "example-5@railstutorial.org", created_at: "2017-04-24 16:12:48", updated_at: "2017-04-24 16:1
2:48", password_digest: "$2a$10$2pj1o9kEQDfxRxxMvki8su6MdnkxLLpr9RdcXNKyaUO5...", remember_token: "fec91bf6f1b09f929b7f5e355016f37a0f832e2c", admin: nil>, #<User id: 7, name: "Neva Abernathy", email: "exam
ple-6@railstutorial.org", created_at: "2017-04-24 16:12:48", updated_at: "2017-04-24 16:12:48", password_digest: "$2a$10$HfLfWapKi7ZpbUdL3/acH.cKhIU7ywVvsndRHbLpG4qj...", remember_token: "70ca21ac9c9962b84
781a424c6c3415d3bb5dbbb", admin: nil>, #<User id: 8, name: "Roger Fisher II", email: "example-7@railstutorial.org", created_at: "2017-04-24 16:12:48", updated_at: "2017-04-24 16:12:48", password_digest: "$
2a$10$qwlv850s285xcdzgvDiIsun0ZfNFw8X1K19hEvGZENUj...", remember_token: "ebcada92f45d721450a509171f8e96d76efb3829", admin: nil>, #<User id: 9, name: "Winfield Huels", email: "example-8@railstutorial.org",
created_at: "2017-04-24 16:12:48", updated_at: "2017-04-24 16:12:48", password_digest: "$2a$10$M3qCQ46CoDraW49xBpbWV.hBt.YzMpYHiXq3h83J0p1X...", remember_token: "87cc56bd127194c7af389aaad3f5879a6fd1810d",
admin: nil>, #<User id: 10, name: "Zelda Leannon", email: "example-9@railstutorial.org", created_at: "2017-04-24 16:12:48", updated_at: "2017-04-24 16:12:48", password_digest: "$2a$10$XRMEkQ5yK3JrfWfMTqdkR
O8Jt/DtDx/JThYcmIK6c3Op...", remember_token: "7283b8043aff717e345f5065f00bc364f37cd450", admin: nil>, ...]>
irb(main):003:0> Micropost.create
   (0.1ms)  BEGIN
   (0.2ms)  ROLLBACK
=> #<Micropost id: nil, content: nil, user_id: nil, created_at: nil, updated_at: nil>
irb(main):004:0> Micropost.create :content => 'micropost de prueba', :user_id => 8
   (0.2ms)  BEGIN
  SQL (4.7ms)  INSERT INTO "microposts" ("content", "created_at", "updated_at", "user_id") VALUES ($1, $2, $3, $4) RETURNING "id"  [["content", "micropost de prueba"], ["created_at", Mon, 24 Apr 2017 16:46
:47 UTC +00:00], ["updated_at", Mon, 24 Apr 2017 16:46:47 UTC +00:00], ["user_id", 8]]
   (0.9ms)  COMMIT
=> #<Micropost id: 301, content: "micropost de prueba", user_id: 8, created_at: "2017-04-24 16:46:47", updated_at: "2017-04-24 16:46:47">
irb(main):005:0> @micropost = Micropost.find(301)
  Micropost Load (0.8ms)  SELECT "microposts".* FROM "microposts" WHERE "microposts"."id" = $1 ORDER BY created_at DESC LIMIT 1  [["id", 301]]
=> #<Micropost id: 301, content: "micropost de prueba", user_id: 8, created_at: "2017-04-24 16:46:47", updated_at: "2017-04-24 16:46:47">
irb(main):006:0> @micropost.content
=> "micropost de prueba"
irb(main):007:0>
```

Figura 3.20 Ejecución de ‘rails console’ desde la consola del contenedor de Ruby.

A través de ‘rails console’ se obtienen los registros de usuarios por medio de la ejecución de ‘User.all’. También se ha podido comprobar que se crean microposts correctamente ejecutando la acción de creación del modelo y realizando una query a continuación que ha devuelto el micropost creado.

También se puede acceder al micropost creado a través del navegador del host accediendo a la página del usuario que lo ha creado por medio de su id.

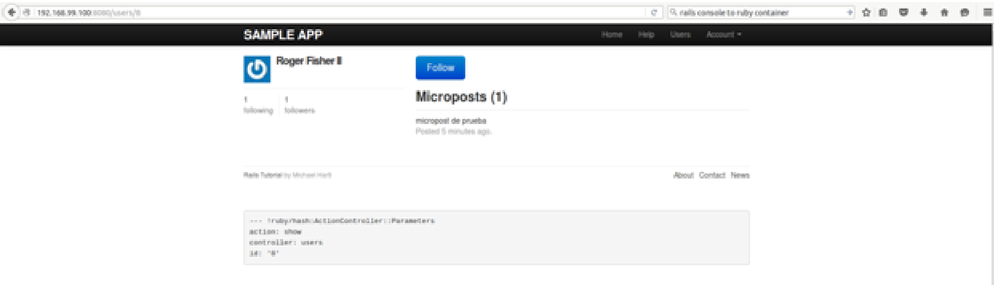

Figura 3.21 Acceso al micropost creado a través del ‘rails console’ accediendo a la página del usuario que lo ha creado.

3.3 Iteración 2: Conversión a una arquitectura de microservicios con el uso de Docker Compose.

En esta segunda iteración se realizará el despliegue de la aplicación en único host con un único fichero '*docker-compose.yml*', en este caso el fichero '*docker-compose-tarea2.yml*'. En este fichero se describirán los tres servicios que compondrán la aplicación (Postgres, aplicación de Ruby y Nginx), a partir de los comandos descritos en la 1ª iteración transcritos al formato del fichero de configuración de docker-compose.

3.3.1 Configuración automatizada para la creación de los contenedores con docker-compose

En la figura 3.22 se puede observar la configuración del fichero *'docker-compose-tarea2.yml'*.

```
version: '2'
services:
  some-postgres:
     build: postgres
    container_name: postgres
    ports:
      - 5432:5432
    environment:
      - POSTGRES_USER=${POSTGRES_USER}
      - POSTGRES_PASSWORD=${POSTGRES_PASSWORD}
    restart: always
  some-Ruby:
    image: mayrez/sample_app_image:0.1
    ports:
      - 3000:3000
    links:
      - some-postgres:db
    working_dir: /usr/src/app
    environment:
      - POSTGRES_USER=$POSTGRES_USER
      - POSTGRES_PASSWORD=$POSTGRES_PASSWORD
      - POSTGRESIP=$POSTGRES
      - WEB_CONCURRENCY=1 # How many worker processes to run
      - RAILS_MAX_THREADS=16 # Configure to be the maximum number of threads
    restart: always
    command:
     /bin/bash -c \
       'cp config/database.yml.postgres config/database.yml &&
       rake db:setup &&
       rake db:migrate &&
```

```
      rake db:populate &&
      rails server'
    depends_on:
      - some-postgres
  some-nginx:
    image: mayrez/some-nginx:0.1
    ports:
      - 8080:80
    links:
      - some-ruby:app
    restart: always
```

Figura 3.22 Fichero docker-compose-tarea2.yml para el despliegue de la aplicación en un único host

El servicio Postgres se construye a partir del directorio Postgres del proyecto. Este directorio contiene un fichero '*Dockerfile'* que indica de dónde obtener la imagen.

El servicio de Ruby depende del contenedor de Postgres. Se le pasan las variables de configuración de la base de datos y los comandos a ejecutar para configurar la base de datos de la aplicación.

Por último, se crea el servicio de Nginx que enlaza con la aplicación de Ruby.

Para desplegar la aplicación se ejecuta el siguiente comando:

```
# docker-compose -f docker-compose-tarea2.yml up
```

Figura 3.23 Comando para ejecutar el despliegue de la aplicación con docker-compose

```
naox@naox-SATELLITE-P50-A-12Z:~/Desktop/docker_swarm_project/docker_project$ docker-compose -f docker-compose-tarea2.yml up
Starting postgres
Recreating dockerproject_some-ruby_1
Recreating dockerproject_some-nginx_1
Attaching to postgres, dockerproject_some-ruby_1, dockerproject_some-nginx_1
postgres        | LOG:  database system was interrupted; last known up at 2017-04-24 18:59:56 UTC
postgres        | LOG:  database system was not properly shut down; automatic recovery in progress
postgres        | LOG:  invalid record length at 0/154EB68: wanted 24, got 0
postgres        | LOG:  redo is not required
postgres        | LOG:  MultiXact member wraparound protections are now enabled
postgres        | LOG:  database system is ready to accept connections
postgres        | LOG:  autovacuum launcher started
some-ruby_1     | [DEPRECATION] `last_comment` is deprecated.  Please use `last_description` instead.
some-ruby_1     | [DEPRECATION] `last_comment` is deprecated.  Please use `last_description` instead.
some-ruby_1     | [DEPRECATION] `last_comment` is deprecated.  Please use `last_description` instead.
some-ruby_1     | [DEPRECATION] `last_comment` is deprecated.  Please use `last_description` instead.
some-ruby_1     | [DEPRECATION] `last_comment` is deprecated.  Please use `last_description` instead.
postgres        | ERROR:  database "sample_app_development" already exists
postgres        | STATEMENT:  CREATE DATABASE "sample_app_development" ENCODING = 'unicode'
some-ruby_1     | sample_app_development already exists
some-ruby_1     | sample_app_test already exists
some-ruby_1     | -- enable_extension("plpgsql")
some-ruby_1     |    -> 0.0129s
some-ruby_1     | -- create_table("microposts", {:force=>true})
some-ruby_1     |    -> 0.0071s
some-ruby_1     | -- add_index("microposts", ["user_id", "created_at"], {:name=>"index_microposts_on_user_id_and_created_at", :using=>:btree})
some-ruby_1     |    -> 0.0025s
some-ruby_1     | -- create_table("relationships", {:force=>true})
some-ruby_1     |    -> 0.0044s
some-ruby_1     | -- add_index("relationships", ["followed_id"], {:name=>"index_relationships_on_followed_id", :using=>:btree})
some-ruby_1     |    -> 0.0022s
some-ruby_1     | -- add_index("relationships", ["follower_id", "followed_id"], {:name=>"index_relationships_on_follower_id_and_followed_id", :unique=>true, :using=>:btree})
some-ruby_1     |    -> 0.0030s
some-ruby_1     | -- add_index("relationships", ["follower_id"], {:name=>"index_relationships_on_follower_id", :using=>:btree})
some-ruby_1     |    -> 0.0024s
some-ruby_1     | -- create_table("users", {:force=>true})
some-ruby_1     |    -> 0.0054s
some-ruby_1     | -- add_index("users", ["email"], {:name=>"index_users_on_email", :unique=>true, :using=>:btree})
some-ruby_1     |    -> 0.0025s
some-ruby_1     | -- add_index("users", ["remember_token"], {:name=>"index_users_on_remember_token", :using=>:btree})
some-ruby_1     |    -> 0.0022s
some-ruby_1     | -- initialize_schema_migrations_table()
some-ruby_1     |    -> 0.0027s
some-ruby_1     | [DEPRECATION] `last_comment` is deprecated.  Please use `last_description` instead.
some-ruby_1     | [DEPRECATION] `last_comment` is deprecated.  Please use `last_description` instead.
some-ruby_1     | [DEPRECATION] `last_comment` is deprecated.  Please use `last_description` instead.
some-ruby_1     | [DEPRECATION] `last_comment` is deprecated.  Please use `last_description` instead.
some-ruby_1     | [DEPRECATION] `last_comment` is deprecated.  Please use `last_description` instead.
some-ruby_1     | [DEPRECATION] `last_comment` is deprecated.  Please use `last_description` instead.
some-ruby_1     | [DEPRECATION] `last_comment` is deprecated.  Please use `last_description` instead.
some-ruby_1     | [DEPRECATION] `last_comment` is deprecated.  Please use `last_description` instead.
some-ruby_1     | [DEPRECATION] `last_comment` is deprecated.  Please use `last_description` instead.
some-ruby_1     | [DEPRECATION] `last_comment` is deprecated.  Please use `last_description` instead.
some-ruby_1     | => Booting Puma
some-ruby_1     | => Rails 4.0.8 application starting in development on http://0.0.0.0:3000
some-ruby_1     | => Run `rails server -h` for more startup options
some-ruby_1     | => Ctrl-C to shutdown server
```

Figura 3.24 Resultado ejecución ‘docker-compose up’

3.3.2 Resultados

Finalmente, se comprueba el funcionamiento de la aplicación a través de la línea de comandos mediante la utilidad ‘curl’ (figura 3.25).

```
# curl $(docker-machine ip machine1):8080/public
```

Figura 3.25 Comprobación del funcionamiento de la aplicación con el comando *curl* accediendo a la dirección de la aplicación

También se puede comprobar en el navegador del host accediendo a través de la ip de la máquina *machine1*.

Para comprobar el correcto funcionamiento de la base de datos ejecutaremos el ‘rails console’ y se realizarán operaciones con la misma.

Para obtener el id del contenedor de Ruby y poder acceder a su consola, ejecutamos 'docker ps' para obtener el id del contenedor. Una vez obtenido el id del contenedor ejecutaremos 'docker exec -it idContenedor bash' y accederemos a la consola del mismo.

```
# docker exec -it ed16a03dee8d bash
root@ed16a03dee8d:/usr/src/app# rails console
Loading development environment (Rails 4.0.8)
irb(main):001:0> User.first
 User Load (1.0ms)  SELECT "users".* FROM "users" ORDER BY "users"."id"
ASC LIMIT 1
=> #<User id: 1, name: "Example User", email: "example@railstutorial.org",
created_at: "2017-04-24 16:12:45", updated_at: "2017-04-24 16:12:45",
password_digest:
"$2a$10$jshXYIy5vQNagWEJcsSMfe23y.oSrvXv.w/RwEyfc89V...",
remember_token: "ac60197bf21ccf2b2b60a3d4136a8db94977c24e", admin:
true>
```

Figura 3.26 Acceso a la consola del contenedor de Ruby y comprobación del funcionamiento de la base de datos mediante accesos a la misma por medio del 'rails console'

Tras las pruebas realizadas a la base de datos de la aplicación se comprueba que está funcionando correctamente. A continuación, se muestran imágenes con los resultados de las comprobaciones en las figuras 3.27 y 3.28.

```
naox@naox-SATELLITE-P50-A-12Z:~/Desktop/docker_swarm_project/docker_project$ docker ps
CONTAINER ID        IMAGE                            COMMAND                  CREATED             STATUS               PORTS                              NAMES
ba4374a1c6e5        mayrez/some-nginx:0.1            "nginx -g 'daemon off"   2 minutes ago       Up About a minute    443/tcp, 0.0.0.0:8080->80/tcp      dockerproject_some-nginx_1
14c0c2300d76        mayrez/sample_app_image:0.1      "/bin/bash -c ' cp co"   2 minutes ago       Up About a minute    0.0.0.0:3000->3000/tcp             dockerproject_some-ruby_1
f9e1d505ad5b        dockerproject_some-postgres      "docker-entrypoint.sh"   5 minutes ago       Up About a minute    0.0.0.0:5432->5432/tcp             postgres
naox@naox-SATELLITE-P50-A-12Z:~/Desktop/docker_swarm_project/docker_project$ docker exec -it 14c0c2300d76 bash
root@14c0c2300d76:/usr/src/app# rails console
Loading development environment (Rails 4.0.8)
irb(main):001:0> User.all
  User Load (1.6ms)  SELECT "users".* FROM "users"
=> #<ActiveRecord::Relation [#<User id: 1, name: "Example User", email: "example@railstutorial.org", created_at: "2017-04-24 19:00:45", updated_at: "2017-04-24 19:00:45", password_digest: "$2a$10$F0ELNkQ7g
7ahQEmf23zb3uf61GlfnwY3krEMVmUiJFHh...", remember_token: "1ec940ad5c03ca1bd4946aa2bf1f5852000049ef", admin: true>, #<User id: 2, name: "Dora Green", email: "example-1@railstutorial.org", created_at: "2017-
04-24 19:00:45", updated_at: "2017-04-24 19:00:45", password_digest: "$2a$10$uxNSbBYhqCMjTxOyDvj4VuUlcNneQZIALaSYpDycyvl0...", remember_token: "fd1ce307d3c2f4076f9be90f5f4e6458cd9a71ef", admin: nil>, #<Use
r id: 3, name: "Mr. Berniece Brakus", email: "example-2@railstutorial.org", created_at: "2017-04-24 19:00:45", updated_at: "2017-04-24 19:00:45", password_digest: "$2a$10$ZANteC0zGjUCKQnf5dBTNehRgL7ECljKcE
i3my78sHxk...", remember_token: "26dc42cfc6ffcf537dec5422df4c312bc5076e87", admin: nil>, #<User id: 4, name: "Leone Deckow", email: "example-3@railstutorial.org", created_at: "2017-04-24 19:00:46", updated
_at: "2017-04-24 19:00:46", password_digest: "$2a$10$f6NuHHFhj/hDvcnvCmBiuealGhoIHnlDJ95h4Con.mbW...", remember_token: "cf27b04cbbd83e60acab90e5e8d7c32c9ec717c9", admin: nil>, #<User id: 5, name: "Nathanae
l Reinger", email: "example-4@railstutorial.org", created_at: "2017-04-24 19:00:46", updated_at: "2017-04-24 19:00:46", password_digest: "$2a$10$ZWWLTe8xUF3vyz7P4QldLuDVvmS2vyZO0vcloB9nDmla...", remember_t
oken: "1405e29c9efa8611bba52047c77da95cc1943b82", admin: nil>, #<User id: 6, name: "Efren Konopelski", email: "example-5@railstutorial.org", created_at: "2017-04-24 19:00:46", updated_at: "2017-04-24 19:00
:46", password_digest: "$2a$10$iUMeDO9IwzJf9KO8UG3lMuWPLc9IB1kBUlhyBXhkWDIk...", remember_token: "90ff0936258b238a6dae1c05623e529498a9aea7", admin: nil>, #<User id: 7, name: "Lesly Pagac", email: "example-
6@railstutorial.org", created_at: "2017-04-24 19:00:46", updated_at: "2017-04-24 19:00:46", password_digest: "$2a$10$bT2oNuE4SWPN2csbZIRYoup4brQVbR027wXoAbeMYpsu...", remember_token: "03a1ee82e228622e60ff3
0838d72f3ab31af4f83", admin: nil>, #<User id: 8, name: "Chadrick Witting", email: "example-7@railstutorial.org", created_at: "2017-04-24 19:00:46", updated_at: "2017-04-24 19:00:46", password_digest: "$2a$
10$mGX8.cziR2tFuhLPvkAZ5e0kxMsf1PzlIKBpUtRaaVAZ...", remember_token: "a8511864b6557cc797971ca17e5f8944e2a76d66", admin: nil>, #<User id: 9, name: "Kenya Raynor", email: "example-8@railstutorial.org", creat
ed_at: "2017-04-24 19:00:46", updated_at: "2017-04-24 19:00:46", password_digest: "$2a$10$E4SqG.4ixA/GVZZ1CJ3tYOdYJ79m6kfodc03VKLIKl1p...", remember_token: "7daee50facc3589a8043de7ccc011f550ec87039", admin
: nil>, #<User id: 10, name: "Jovanny Grant V", email: "example-9@railstutorial.org", created_at: "2017-04-24 19:00:46", updated_at: "2017-04-24 19:00:46", password_digest: "$2a$10$27931w/0p9eXza6HTqAwmeGX
qDVpJYGXTLR2H2JGGAAX...", remember_token: "6a487226d731847af7930a7caf2409cd549c6dc0", admin: nil>, ...]>
irb(main):002:0> Micropost.create :user_id => 10, :content => 'micropost creado en la tarea 2'
   (0.1ms)  BEGIN
  SQL (4.5ms)  INSERT INTO "microposts" ("content", "created_at", "updated_at", "user_id") VALUES ($1, $2, $3, $4) RETURNING "id"  [["content", "micropost creado en la tarea 2"], ["created_at", Mon, 24 Apr
 2017 19:03:33 UTC +00:00], ["updated_at", Mon, 24 Apr 2017 19:03:33 UTC +00:00], ["user_id", 10]]
   (0.9ms)  COMMIT
=> #<Micropost id: 301, content: "micropost creado en la tarea 2", user_id: 10, created_at: "2017-04-24 19:03:33", updated_at: "2017-04-24 19:03:33">
irb(main):003:0> @post = Micropost.find(301)
  Micropost Load (2.2ms)  SELECT "microposts".* FROM "microposts" WHERE "microposts"."id" = $1 ORDER BY created_at DESC LIMIT 1  [["id", 301]]
=> #<Micropost id: 301, content: "micropost creado en la tarea 2", user_id: 10, created_at: "2017-04-24 19:03:33", updated_at: "2017-04-24 19:03:33">
irb(main):004:0> @post.content
=> "micropost creado en la tarea 2"
irb(main):005:0>
```

Figura 3.27 Prueba en 'rails console', accediendo a la consola del contenedor de Ruby

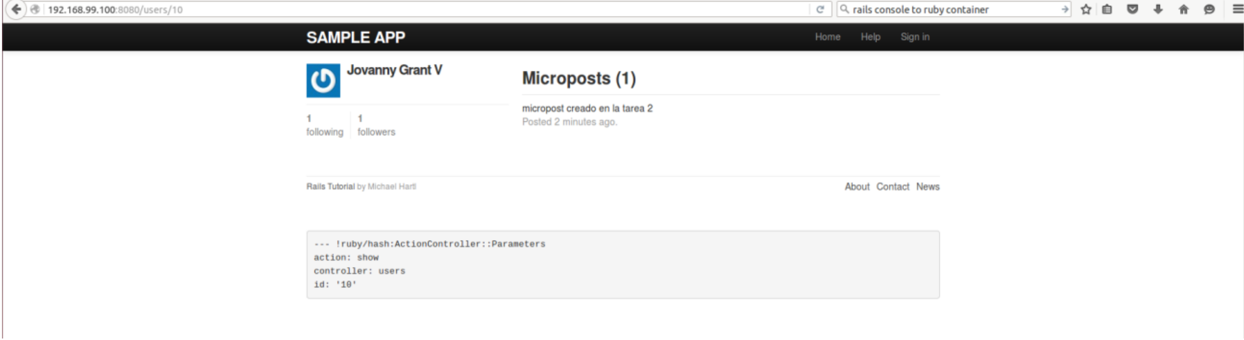

Figura 3.28 Acceso al usuario desde el que se ha creado el post de prueba

3.4 Iteración 3: Adaptación de la configuración para que cada una de las capas de la aplicación funcione en tres MV locales diferentes.

En esta iteración, se va a separar cada capa de la aplicación que era desplegaba previamente con un único fichero docker-compose y desde una única máquina virtual, en varios ficheros docker-compose y en tres máquinas virtuales diferentes creadas con docker-machine.

Ahora cada servicio funcionará independientemente en una máquina distinta sin conexión con las otras, con lo que habrá que solucionar el problema de descubrimiento de los servicios entre las distintas máquinas. Para arreglar este problema del descubrimiento de los servicios, existen varias opciones, en este caso se utilizará el overlay networking y Consul para resolver el problema.

El overlay networking permite que un contenedor se conecte a otro contenedor usando una ip local. Este tipo de conexión requiere de un servicio de almacenamiento clave-valor para la resolución de nombres de servicios. En este caso se usará Consul como servicio almacenamiento clave-valor. Consul permite que un contenedor se pueda conectar a otro usando el nombre.

3.4.1. Separación de los servicios en tres ficheros docker-compose diferentes

El fichero de docker-compose que teníamos previamente para el despliegue de la aplicación de tres capas en local, se va a descomponer ahora en tres ficheros cada uno para el despliegue de una capa diferente.

El primer fichero, '*docker-compose-tarea3-postgres.yml'*, describirá el servicio de Consul y la base de datos Postgres. El contenido del fichero se muestra en la figura 3.29.

```
version: '2'
services:
  some-postgres:
    build: postgres
    container_name: postgres
    ports:
      - 5432:5432
    environment:
      - POSTGRES_USER=${POSTGRES_USER}
      - POSTGRES_PASSWORD=${POSTGRES_PASSWORD}
    restart: always
  consul:
    image: progrium/consul
    restart: always
    ports:
      - "8500:8500" # REST API & UI
      - "8300:8300"
      - "8301:8301"
      - "8301:8301/udp"
      - "8302:8302"
      - "8302:8302/udp"
      - "8400:8400"
      - "53:53/udp" # DNS
    command:
      "-server -bootstrap -ui-dir /ui -advertise $CONSULIP"
      container_name: consul
```

Figura 3.29 Contenido del fichero docker-compose para el despliegue del servidor Consul y la base de datos

La aplicación de Ruby viene descrita por el fichero '*docker-compose-tarea3-ruby.yml'* cuyo contenido se muestra en la figura 3.30. A partir de este fichero también se creará un agente Consul que se conecte con el servidor principal para la resolución del descubrimiento de servicios.

```
version: '2'
services:
  some-ruby:
    image: mayrez/sample_app_image:0.1
    ports:
      - 3000:3000
    external_links:
      - some-postgres:db
    working_dir: /usr/src/app
    environment:
      - POSTGRES_USER=$POSTGRES_USER
      - POSTGRES_PASSWORD=$POSTGRES_PASSWORD
      - POSTGRESIP=$POSTGRES
      - WEB_CONCURRENCY=1 # How many worker processes to run
      - RAILS_MAX_THREADS=16 # Configure to be the maximum number of threads
    restart: always
    command:
      /bin/bash -c \
        'cp config/database.yml.postgres config/database.yml &&
        rake db:setup &&
        rake db:migrate &&
        rake db:populate &&
        rails server'
  agent-1:
    image: progrium/consul
    container_name: consul_agent_1
    ports:
      - "8500:8500" # REST API & UI
      - "8300:8300"
      - "8301:8301"
      - "8301:8301/udp"
      - "8302:8302"
      - "8302:8302/udp"
      - "8400:8400"
      - "53:53/udp" # DNS
    environment:
      - "constraint:node==node2"
    command: -ui-dir /ui -join $CONSULIP -advertise $NODE2
```

```
networks:
  default:
    external:
      name: multi-host-net
```

Figura 3.30 Contenido del fichero docker-compose para el despliegue de la aplicación de Ruby y el agente Consul.

Por último, el servidor web viene descrito por el fichero '*docker-compose-tarea3-nginx*' cuyo contenido se muestra en la figura 3.31. También se creará a partir de este fichero el segundo agente Consul que se conectará con el servidor principal para la resolución de nombre de servicios.

```
version: '2'
services:
  some-nginx:
    image: mayrez/some-nginx:0.1
    ports:
      - 8080:80
    external_links:
      - some-ruby:app
    restart: always
  agent-2:
    image: progrium/consul
    container_name: consul_agent_2
    ports:
      - "8500:8500" # REST API & UI
      - "8300:8300"
      - "8301:8301"
      - "8301:8301/udp"
      - "8302:8302"
      - "8302:8302/udp"
      - "8400:8400"
      - "53:53/udp" # DNS
    environment:
      - "constraint:node==node3"
    command: -ui-dir /ui -join $CONSULIP -advertise $NODE3
networks:
  default:
```

```
  external:
    name: multi-host-net
```

Figura 3.31 Contenido del fichero docker-compose ('*docker-compose-tarea3-nginx.yml*') para el despliegue del servidor de Nginx y el agente Consul.

3.4.2. Adaptación de la aplicación para que cada capa se ejecute en una MV diferente con docker-machine y docker-compose.

Se crearán tres máquinas diferentes cada una de las cuales desplegará un servicio de la aplicación diferente:

- En la máquina *postgres-machine* se desplegará el servidor de Consul y la base de datos Postgres. Para ello, se ejecutará desde esta máquina el fichero '*docker-compose-tarea3-postgres.yml'*.

- En la máquina *ruby-machine* se desplegará la aplicación de Ruby y un agente Consul. Para ello, se ejecutará desde esta máquina el fichero '*docker-compose-tarea3-ruby.yml'*.

- En la máquina *ruby-machine* se desplegará el servidor Nginx y un agente Consul. Para ello, se ejecutará desde esta máquina el fichero '*docker-compose-tarea3-nginx.yml'*.

Para desplegar la aplicación se ha definido un script con cada uno de los pasos a ejecutar desde la creación de las MV con docker-machine hasta la configuración de cada máquina ejecutando los ficheros docker-compose.

El contenido de este fichero se muestra en la figura 3.32 que aparece a continuación:

```
#!/bin/bash
export POSTGRES_USER=postgres
export POSTGRES_PASSWORD=mysecretpassword

docker-machine create -d virtualbox postgres-machine
docker-machine regenerate-certs postgres-machine -f
eval $(docker-machine env postgres-machine)
export CONSULIP=$(docker-machine ip postgres-machine)
export POSTGRES=$(docker-machine ip postgres-machine)
```

```
docker-compose -f docker-compose-tarea3-postgres.yml up &
until PGPASSWORD="$POSTGRES_PASSWORD" psql -h "$POSTGRES" -U
"$POSTGRES_USER" -c '\l'; do
  >&2 echo "Postgres is unavailable - sleeping"
  sleep 1
done &&

docker-machine create -d virtualbox  \
--engine-opt="cluster-store=consul://$(docker-machine ip postgres-machine):8500" \
--engine-opt="cluster-advertise=eth1:2376" \
ruby-machine

docker-machine regenerate-certs ruby-machine -f

docker-machine create -d virtualbox  \
--engine-opt="cluster-store=consul://$(docker-machine ip postgres-machine):8500" \
--engine-opt="cluster-advertise=eth1:2376" \
nginx-machine

docker-machine regenerate-certs nginx-machine -f

export NODE2=$(docker-machine ip ruby-machine)
eval $(docker-machine env ruby-machine)
docker network create -d overlay multi-host-net
docker-compose -f docker-compose-tarea3-ruby.yml up &

until  wget $(docker-machine ip ruby-machine):3000/public -O /dev/null -S --quiet 2>&1
| grep  '200 OK' ; do
  >&2 echo "Ruby server is not running - sleeping"
  sleep 1
done &&

eval $(docker-machine env nginx-machine)
export NODE3=$(docker-machine ip nginx-machine)
docker-compose -f docker-compose-tarea3-nginx.yml up &
```

Figura 3.32 Contenido del script tarea3-up.sh con el que se realiza el despliegue de la aplicación en tres MV diferentes.

Como se aprecia en la figura 3.32 en primer lugar ser crea la máquina ‘postgres-machine’ y se ejecuta en esta el fichero ‘*docker-compose-tarea3-postgres.yml’*. Para que no se cree el servicio de Ruby antes de que se haya creado la base de datos, se ha añadido un bucle para que se espere hasta que se tenga conexión con la misma.

Seguidamente se creará las otras dos máquinas virtuales, ‘ruby-machine’ y ‘nginx-machine’. Primero se lanzará la aplicación de Ruby desde la máquina ‘ruby-machine’. Para que no se lance el servidor web antes de que se haya creado la aplicación de la que depende para desplegarse la misma, se ha añadido un bucle que evita que se pase al despliegue del servidor Nginx antes de que la página principal de la aplicación de Ruby sea accesible.

Para desplegar la aplicación se seguirán los pasos que se muestran en la figura 3.33.

```
# git clone https://bitbucket.org/tftdp/docker_project
# cd Paso3
# sh tarea3-up.sh
# curl $(docker-machine ip nginx-machine):8080/public
```

Figura 3.33 Pasos para el despliegue de la aplicación y comprobación de la conexión con la página principal de la aplicación de Ruby.

```
naox@naox-SATELLITE-P50-A-12Z:~$ curl $(docker-machine ip nginx-machine):8080/public
<!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>
<script data-turbolinks-track="true" src="/assets/jquery_ujs.js?body=1"></script>
<script data-turbolinks-track="true" src="/assets/turbolinks.js?body=1"></script>
<script data-turbolinks-track="true" src="/assets/bootstrap-transition.js?body=1"></script>
```

Figura 3.34 Resultado ‘curl’ a la página principal a través de la ip de la máquina del servidor web Nginx.

3.4.3. Resultados

Se podrá acceder a la aplicación a través del navegador del host, con la ip de la máquina con el servidor Nginx y accediendo al puerto 8080 (figura 3.35).

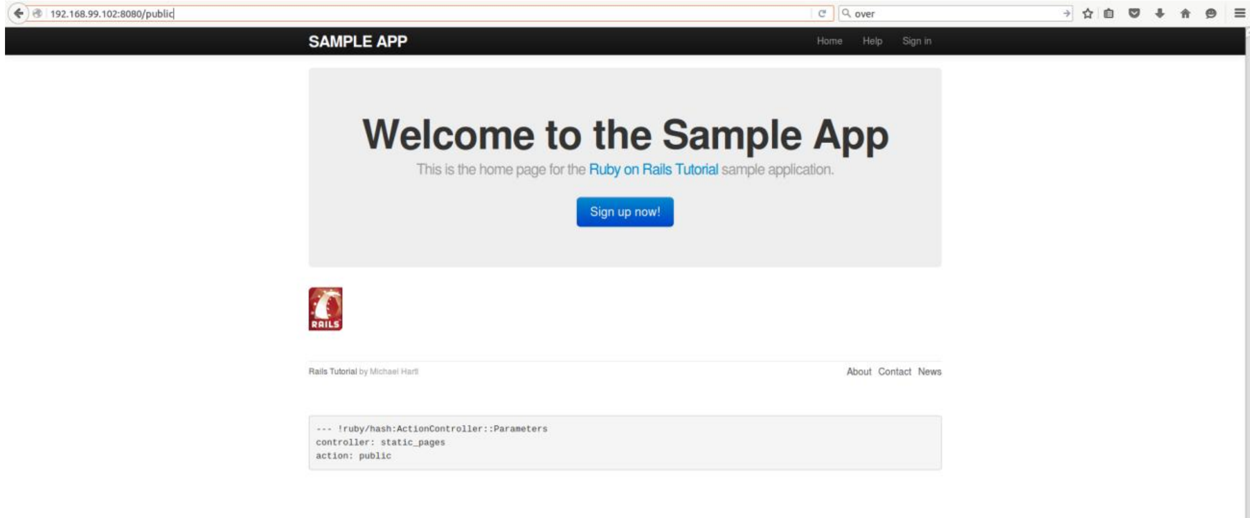

Figura 3.35 Acceso a la aplicación a través del navegador del host accediendo con la ip de la máquina con Nginx.

Para comprobar la correcta conexión de la aplicación a la base de datos se ejecutará 'rails console' desde el contenedor de Ruby. Para ello, se busca el id del contenedor accediendo al entorno de la máquina desde la que se ha lanzado el contenedor, ejecutando 'docker ps' y accediendo a su bash ejecutando 'docker exec -it idContenedor bash' (figura 3.36).

```
# eval $(docker-machine env ruby-machine)

# docker ps

# docker exec -it ed16a03dee8d bash

root@ed16a03dee8d:/usr/src/app# rails console

Loading development environment (Rails 4.0.8)

irb(main):001:0> User.first

  User Load (1.0ms) SELECT "users".* FROM "users" ORDER BY "users"."id" ASC LIMIT 1

=> #<User id: 1, name: "Example User", email: "example@railstutorial.org", created_at: "2017-04-24 16:12:45", updated_at: "2017-04-24 16:12:45", password_digest: "$2a$10$jshXYIy5vQNagWEJcsSMfe23y.oSrvXv.w/RwEyfc89V...", remember_token: "ac60197bf21ccf2b2b60a3d4136a8db94977c24e", admin: true>
```

Figura 3.36 Acceso a la consola del contenedor de Ruby y comprobación de la conexión con la base de datos a través del 'rails console'

Los resultados de la comprobación de la conexión con la base de datos se muestran en las figuras de la 3.37 a la 3.39. En estas se muestra como se ha creado un post, con el contenido 'Post de prueba tarea 3', asignándoselo al usuario 10 y luego se ha hecho una query a dicho post para comprobar su creación. También se ha accedido a través de la interfaz web al perfil del usuario, donde se ha podido visualizar el post creado.

```
naox@naox-SATELLITE-P50-A-12Z:~$ eval $(docker-machine env ruby-machine)
naox@naox-SATELLITE-P50-A-12Z:~$ docker ps
CONTAINER ID        IMAGE                          COMMAND                  CREATED             STATUS              PORTS
                                                          NAMES
09dcb012f5ff        mayrez/sample_app_image:0.1    "/bin/bash -c ' cp co"   8 minutes ago       Up 7 minutes        0.0.0.0:3000->3000/tcp
                                                          midockerproject_some-ruby_1
7b8a682414e7        progrium/consul                "/bin/start -ui-dir /"   8 minutes ago       Up 7 minutes        0.0.0.0:53->53/udp, 0.0.0.0:8300-8302->8300-8302/tcp, 0.0.0.0:8400->8400/tcp, 53/tcp, 0.0.
0.0:8301-8302->8301-8302/udp, 0.0.0.0:8500->8500/tcp   consul_agent_1
naox@naox-SATELLITE-P50-A-12Z:~$ docker exec -it 09dcb012f5ff bash
root@09dcb012f5ff:/usr/src/app# rails console
Loading development environment (Rails 4.0.8)
irb(main):001:0> User.all
  User Load (3.0ms)  SELECT "users".* FROM "users"
=> #<ActiveRecord::Relation [#<User id: 1, name: "Example User", email: "example@railstutorial.org", created_at: "2017-04-30 18:59:24", updated_at: "2017-04-30 18:59:24", password_digest: "$2a$10$AciwcSH8y
c6KQSHdmBEnUewR3XupIr1Oy9ft71WMIXkV...", remember_token: "8c560c9716e2cc5d3dfb747a1f8f562f4d50a61c", admin: true>, #<User id: 2, name: "Ms. Flossie Mueller", email: "example-1@railstutorial.org", created_a
t: "2017-04-30 18:59:24", updated_at: "2017-04-30 18:59:24", password_digest: "$2a$10$2v5lA1JbV/G9.PPV0DLGjO8LthaJ.mCqzXAcwxTVfNy7...", remember_token: "14faccc436779dc6c68ee30be14472cea07ea7b7", admin: ni
l>, #<User id: 3, name: "Ramona Goyette", email: "example-2@railstutorial.org", created_at: "2017-04-30 18:59:24", updated_at: "2017-04-30 18:59:24", password_digest: "$2a$10$hKbnYWIrS30Ycry0eJcyY.lV7Gk09M
LS25tha1CUQhcX...", remember_token: "83448984a8efcd2001088966954e3c1dfdaf31d1", admin: nil>, #<User id: 4, name: "Anais Hauck", email: "example-3@railstutorial.org", created_at: "2017-04-30 18:59:24", upda
ted_at: "2017-04-30 18:59:24", password_digest: "$2a$10$YaXCPyBJfKlVe0XHOYF0rOzhJNhzBCoLUglONu5wUKht...", remember_token: "0ed7daf03f91f079b6414c5dd50328de60241fb4", admin: nil>, #<User id: 5, name: "Rudy
Hintz DVM", email: "example-4@railstutorial.org", created_at: "2017-04-30 18:59:24", updated_at: "2017-04-30 18:59:24", password_digest: "$2a$10$ZcY3EIzVD.Ii62fYdfjJ4ewljM0DtqYLvQe6f/qNLD/S...", remember_t
oken: "668f360313ac5561676c2e34a7fbab2b180610e2", admin: nil>, #<User id: 6, name: "Mr. Fabian Ratke", email: "example-5@railstutorial.org", created_at: "2017-04-30 18:59:24", updated_at: "2017-04-30 18:59
:24", password_digest: "$2a$10$JBZJ4bhQvxxCa5c3DmSyD.xaOFphnYGXd4XHd.0puOCQ...", remember_token: "fa9e4b82d6a67a3c7a085918f441bcb4887a1841", admin: nil>, #<User id: 7, name: "Jean Kirlin II", email: "examp
le-6@railstutorial.org", created_at: "2017-04-30 18:59:24", updated_at: "2017-04-30 18:59:24", password_digest: "$2a$10$zLNjBIqdZ16rfdaIt7l56uwW3tEpmN3W6f7KXQvJO0wy...", remember_token: "f36ec247044f22da4e
06675f90795e6c68dfdffc", admin: nil>, #<User id: 8, name: "Taurean Jewess", email: "example-7@railstutorial.org", created_at: "2017-04-30 18:59:24", updated_at: "2017-04-30 18:59:24", password_digest: "$2a
$10$aRKi9H.pmG5cztCx2E5HaOZKj1GxX.L8QFZ6QA0WUL6M...", remember_token: "79769316ed3c349d7d5898c6cc087514e184bcd4", admin: nil>, #<User id: 9, name: "Ali Kreiger", email: "example-8@railstutorial.org", creat
ed_at: "2017-04-30 18:59:24", updated_at: "2017-04-30 18:59:24", password_digest: "$2a$10$Cv3XHvcqCD.9SOuJ97ki.O56oYdUGIgUtvJDDeOTg52U...", remember_token: "fa2bdd3a4f6c1805ca3e9df00d38161e1ce82a4b", admin
: nil>, #<User id: 10, name: "Ahmad Erdman", email: "example-9@railstutorial.org", created_at: "2017-04-30 18:59:24", updated_at: "2017-04-30 18:59:24", password_digest: "$2a$10$0LFgAhdinc35XWN3Q/7efeDj3sH
ttp8PnhhMp4aXzX3h...", remember_token: "788e4cf35228ff043f1fe3ec506c99cf42a7e22f", admin: nil>, ...]>
```

Figura 3.37 Acceso a la consola del contenedor de Ruby y comprobación de la conexión con la base de datos a través del 'rails console' ejecutando queries al modelo User.

```
irb(main):005:0> Micropost
=> Micropost(id: integer, content: string, user_id: integer, created_at: datetime, updated_at: datetime)
irb(main):006:0> Micropost.create :user_id => 10, :content => "Post de prueba tarea 3"
   (0.9ms)  BEGIN
  SQL (4.2ms)  INSERT INTO "microposts" ("content", "created_at", "updated_at", "user_id") VALUES ($1, $2, $3, $4) RETURNING "id"  [["content", "Post de prueba tarea 3"], ["created_at", Sun, 30 Apr 2017 19
:09:18 UTC +00:00], ["updated_at", Sun, 30 Apr 2017 19:09:18 UTC +00:00], ["user_id", 10]]
   (0.6ms)  COMMIT
=> #<Micropost id: 301, content: "Post de prueba tarea 3", user_id: 10, created_at: "2017-04-30 19:09:18", updated_at: "2017-04-30 19:09:18">
irb(main):007:0> Micropost.find(301)
  Micropost Load (2.0ms)  SELECT "microposts".* FROM "microposts" WHERE "microposts"."id" = $1 ORDER BY created_at DESC LIMIT 1  [["id", 301]]
=> #<Micropost id: 301, content: "Post de prueba tarea 3", user_id: 10, created_at: "2017-04-30 19:09:18", updated_at: "2017-04-30 19:09:18">
irb(main):008:0>
```

Figura 3.38 Creación de un micropost con el contenido 'Post de prueba tarea 3' y el usuario con id 10

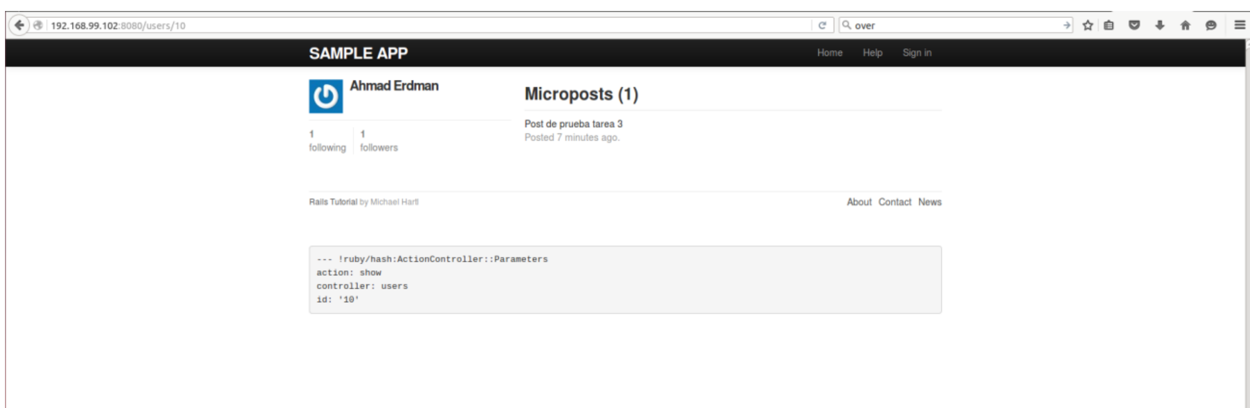

Figura 3.39 Acceso a la página del usuario 10 y comprobación del contenido creado

Para comprobar el funcionamiento de Consul se puede acceder a la interfaz del servicio por medio de la máquina con el servidor de Consul a través de su dirección ip y el puerto 8500. Esta página presenta una pestaña con los servicios, otra pestaña que muestra los nodos y otra que muestra las claves-valores almacenados. Desde la interfaz de esta pestaña se pueden crean y almacenar nuevas claves y valores.

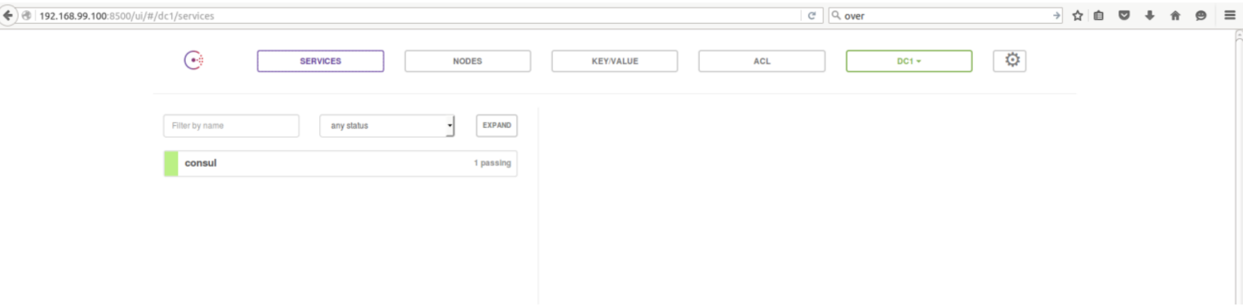

Figura 3.40 Acceso a la página de Consul a través de la ip de la MV con el servidor Consul y el puerto 8500

Si se accede a la pestaña ‘nodes’ (figura 3.41) se verán los tres nodos que componen la aplicación.

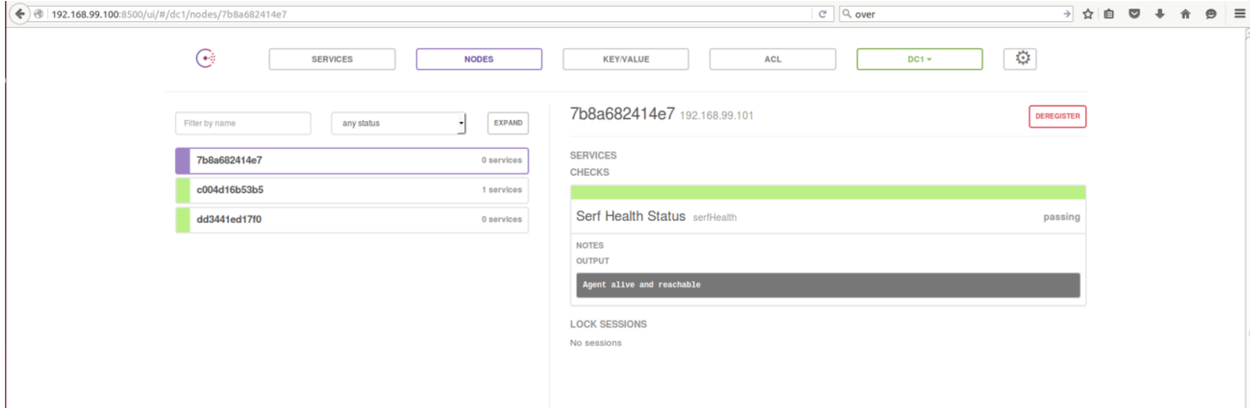

Figura 3.41 Pestaña ‘nodes’ en la página de Consul, accedido a través de la ip de la MV con el servidor Consul y el puerto 8500

Si se accede a la pestaña ‘key/Value’ se podrán ver las direcciones ip y puerto de conexión de Docker de los agentes de Consul.

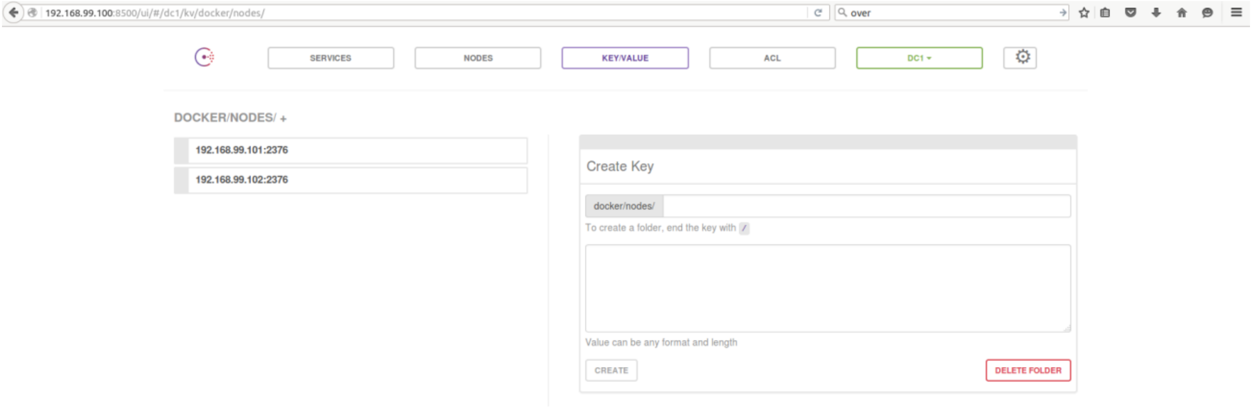

Figura 3.42 Pestaña 'key/value' en la página de Consul, accedida a través de la ip de la MV con el servidor Consul y el puerto 8500

3.5 Iteración 4: Despliegue remoto de la aplicación, en Amazon Web Services (*AWS*)

En esta iteración se procederá a realizar el despliegue de la aplicación en AWS. Para ello, se realizará modificaciones en el script utilizado para el despliegue en la iteración anterior, para que las máquinas virtuales sean creadas ahora de forma remota en la plataforma de AWS. Se modificará el comando de 'docker-machine create' para que en lugar de utilizar Virtualbox como driver utilice el driver 'amazonec2' que lanzarán las instancias de máquinas virtuales en AWS.

3.5.1 Configuración desde la consola de AWS

Para lanzar instancias en AWS con 'docker-machine' será necesario pasarle al comando los siguientes parámetros:

- *--amazonec2-access-key:* El ID de la clave de acceso que obtenemos al crear un usuario en el IAM Dashboard.

- *--amazonec2-secret-key:* La clave secreta de acceso que se obtiene al crear un usuario en el IAM Dashboard. Se descarga junto con la ID de la clave en un fichero con extensión *'.cvs'*.

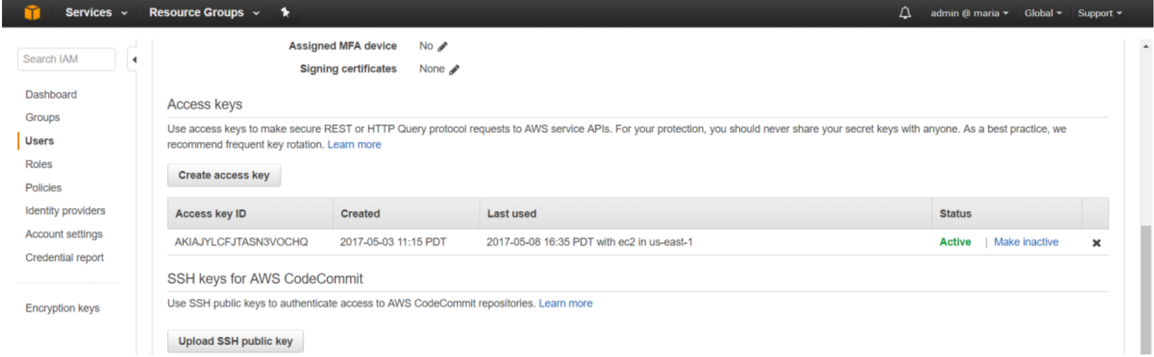

Figura 3.43 Sección 'Access Key' del usuario de la consola de AWS.

- *--amazonec2-vpc-id:* La id del VPC (Virtual Private Cloud).

Figura 3.44 VPC disponible accesible desde el VPC Dashboard.

- *--amazonec2-security-group:* Nombre del grupo de seguridad que hemos configurado para la apertura de los puertos necesarios para el funcionamiento de los servicios de las máquinas virtuales.

Figura 3.45.a Sección del EC2 Dashboard para la gestión de los 'Security Groups'.

- *--amazonec2-region:* Región desde la que se lanza la instancia.
- *--amazonec2-zone:* Zona de la región.

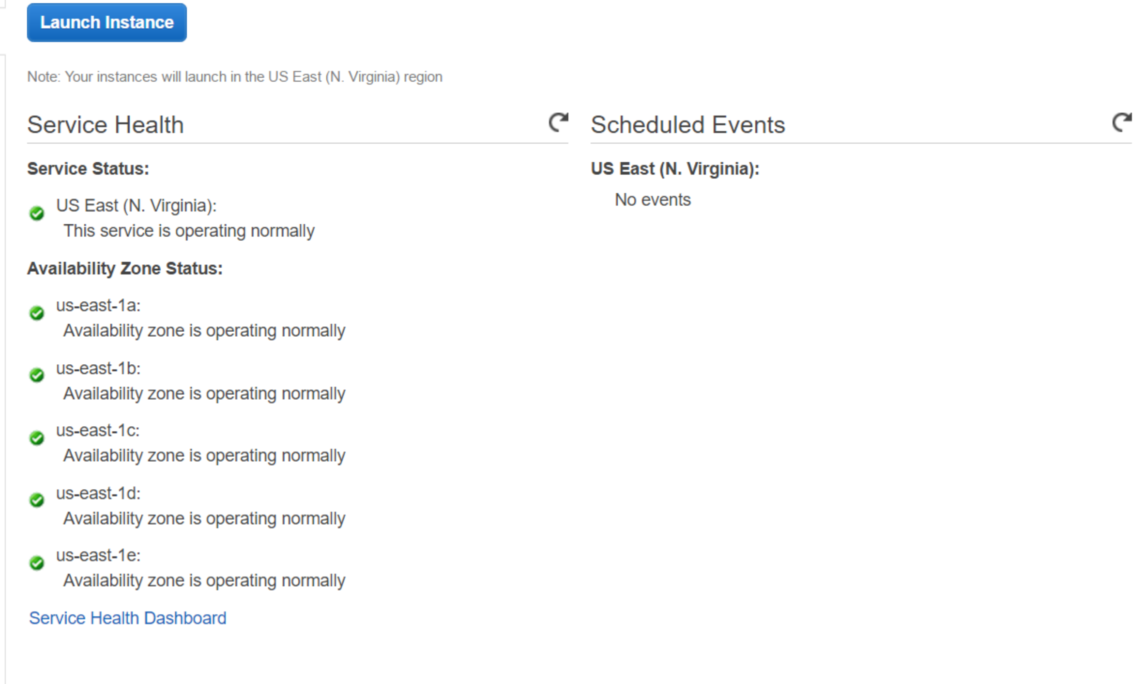

Figura 3.45.b Región desde donde se lanza la instancia y zonas disponibles de dicha región.

Para que la aplicación funcione desde el servidor en la nube será necesario abrir los puertos desde la consola en el EC2 para lo cual se crearán grupos de seguridad, uno por cada máquina, ya que ofrecen servicios diferentes.

Se crearán tres grupos de seguridad en el apartado del EC2 'Security Groups' de la consola de AWS: uno para la máquina de Consul-Postgres, otro para la máquina con Ruby y la última para el servidor web Nginx. En todos ellos se añadirá como 'Inbound Rules' la apertura de puertos (indicando como source 'Anywhere'):

- 22 TCP (SSH)
- 2376 TCP (Docker)
- 8500 TCP, 8300 - 8302 TCP, 8301-8302 UDP, 53 UDP (Consul)

Otras reglas 'inbound' a añadir son las siguientes:

- En el grupo de seguridad de Consul-Postgres se abrirá además el puerto de Postgres, el 5432 TCP.

- En el de Ruby se abrirá el puerto 3000 TCP.
- En el Nginx se abrirá el puerto 8080 TCP.
- Para el funcionamiento de la network overlay se abrirán en los grupos de seguridad de Ruby y Nginx los puertos 7946 TCP, 4789 UDP y 7946 UDP.

3.5.2 Modificación del script de despliegue de la aplicación

Para desplegar la aplicación en AWS se utilizará el mismo script que en la tarea 3, pero en este caso se crearán las máquinas como instancias de Amazon EC2 utilizando para ello en la creación como driver 'amazonec2', añadiendo además los parámetros necesarios para lanzar las instancias en AWS.

También se modificará la interfaz de red a través de la que se produce el 'cluster-advertise' de eth1 a eth0, puesto que en estas máquinas no se encuentra la eth1.

En la figura 3.46 se puede observar la nueva estructura del script para el despliegue de la aplicación en la nube.

```
#!/bin/bash
export POSTGRES_USER=postgres
export POSTGRES_PASSWORD=mysecretpassword
export AWS_ACCESS_KEY_ID=$aws_access_key_id
export AWS_SECRET_ACCESS_KEY=$aws_secret_access_key
export AWS_VPC_ID=vpc-835dbbfa
export AWS_DEFAULT_REGION=us-east-1
export AWS_ZONE=c
export AWS_SECURITY_GROUP=Consul-Postgres

docker-machine -D create --driver amazonec2 \
  --amazonec2-access-key $AWS_ACCESS_KEY_ID \
  --amazonec2-secret-key $AWS_SECRET_ACCESS_KEY \
  --amazonec2-vpc-id $AWS_VPC_ID \
  --amazonec2-security-group $AWS_SECURITY_GROUP \
  --amazonec2-region $AWS_DEFAULT_REGION \
  --amazonec2-zone $AWS_ZONE \
postgres-machine
```

```
docker-machine regenerate-certs postgres-machine -f
eval $(docker-machine env postgres-machine)
export CONSULIP=$(docker-machine ip postgres-machine)
export POSTGRES=$(docker-machine ip postgres-machine)
docker-compose -f docker-compose-tarea4-postgres.yml up &

until PGPASSWORD="$POSTGRES_PASSWORD" psql -h "$POSTGRES"  \
-U "$POSTGRES_USER" -c '\q'; do
  >&2 echo "Postgres is unavailable - sleeping"
  sleep 1
done &&
echo "Postgres available"

export AWS_SECURITY_GROUP=Ruby
docker-machine -D create --driver amazonec2 \
  --amazonec2-access-key $AWS_ACCESS_KEY_ID \
  --amazonec2-secret-key $AWS_SECRET_ACCESS_KEY \
  --amazonec2-vpc-id $AWS_VPC_ID \
  --amazonec2-security-group $AWS_SECURITY_GROUP \
  --amazonec2-region $AWS_DEFAULT_REGION \
  --amazonec2-zone $AWS_ZONE  \
  --engine-opt="cluster-store=consul://$(docker-machine ip postgres-machine):8500" \
  --engine-opt="cluster-advertise=eth0:2376" \
ruby-machine

docker-machine regenerate-certs ruby-machine -f

export AWS_SECURITY_GROUP=Nginx
docker-machine -D create --driver amazonec2 \
  --amazonec2-access-key $AWS_ACCESS_KEY_ID \
  --amazonec2-secret-key $AWS_SECRET_ACCESS_KEY \
  --amazonec2-vpc-id $AWS_VPC_ID \
  --amazonec2-security-group $AWS_SECURITY_GROUP \
  --amazonec2-region $AWS_DEFAULT_REGION \
  --amazonec2-zone $AWS_ZONE  \
  --engine-opt="cluster-store=consul://$(docker-machine ip postgres-machine):8500" \
  --engine-opt="cluster-advertise=eth0:2376" \
nginx-machine
```

```
docker-machine regenerate-certs nginx-machine -f

eval $(docker-machine env ruby-machine)
export NODE2=$(docker-machine ip ruby-machine)
export CONSULIP=$(docker-machine ip postgres-machine)
docker network create -d overlay --subnet=10.0.9.0/24 multi-host-net
docker-compose -f docker-compose-tarea4-ruby.yml up &

until  wget $(docker-machine ip ruby-machine):3000/public -O /dev/null -S --quiet 2>&1 | grep
'200 OK' ; do
  >&2 echo "Ruby server is not running - sleeping"
  sleep 1
done &&

eval $(docker-machine env nginx-machine)
export NODE3=$(docker-machine ip nginx-machine)
docker-compose -f docker-compose-tarea4-nginx.yml up &
```

Figura 3.47 Contenido del script tarea4-up.sh con el que se realiza el despliegue de la aplicación en AWS.

3.5.3 Despliegue de la aplicación en AWS

Los comandos a ejecutar para llevar a cabo el despliegue se muestran en la figura 3.48.

```
# git clone https://bitbucket.org/tftdp/docker_project

# cd Tarea4

# sh tarea4-up.sh

# curl $(docker-machine ip nginx-machine):8080/public
```

Figura 3.48 Pasos para el despliegue de la aplicación y comprobación de la conexión con la página principal de la aplicación de Ruby.

Como resultado del comando ‘curl’ a la dirección de la aplicación se obtiene la página principal (figura 3.48).

```
naox@naox-SATELLITE-P50-A-12Z:~$ curl $(docker-machine ip nginx-machine):8080/public
<!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>
<script data-turbolinks-track="true" src="/assets/jquery_ujs.js?body=1"></script>
<script data-turbolinks-track="true" src="/assets/turbolinks.js?body=1"></script>
<script data-turbolinks-track="true" src="/assets/bootstrap-transition.js?body=1"></script>
```

Figura 3.48 Resultado ‘curl’ a la página principal a través de la ip de la máquina del servidor web Nginx.

3.5.4 Resultados

Como resultado se tendrán tres instancias corriendo en AWS como se muestra en la figura 3.49.

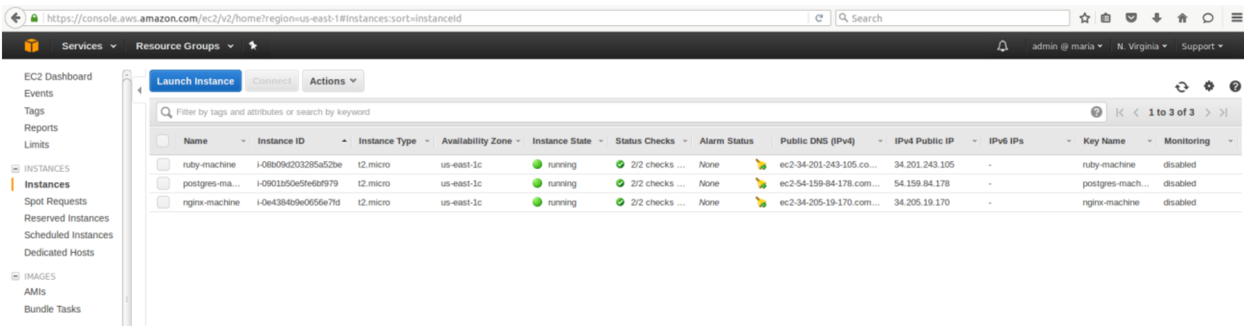

Figura 3.49 Instancias creadas ejecutándose en AWS.

Para comprobar la correcta conexión de la aplicación a la base de datos se ejecutará ‘rails console’ desde el contenedor de Ruby. Para ello, se busca el id del contenedor accediendo al entorno de la máquina desde la que se ha lanzado el contenedor, ejecutando ‘docker ps’ y accediendo a su bash ejecutando ‘docker exec -it idContenedor bash’ (figura 3.50).

```
$ eval $(docker-machine env ruby-machine)
$ docker ps
$ docker exec -it ed16a03dee8d bash
root@ed16a03dee8d:/usr/src/app# rails console
Loading development environment (Rails 4.0.8)
irb(main):001:0> User.first
```

User Load (1.0ms) SELECT "users". FROM "users" ORDER BY "users"."id" ASC LIMIT 1*

=> #<User id: 1, name: "Example User", email: "example@railstutorial.org", created_at: "2017-04-24 16:12:45", updated_at: "2017-04-24 16:12:45", password_digest: "$2a$10$jshXYIy5vQNagWEJcsSMfe23y.oSrvXv.w/RwEyfc89V...", remember_token: "ac60197bf21ccf2b2b60a3d4136a8db94977c24e", admin: true>

Figura 3.50 Pasos para comprobar el funcionamiento de la aplicación y la base de datos.

El resultado de la comprobación del funcionamiento de la aplicación en AWS se muestra en las figuras de la 3.51 a la 3.53. Se ha creado en la base de datos un post con el contenido ‘Post prueba tarea 4’, asignándoselo al usuario 10. Luego se ha realizado una query al post para comprobar su creación. Finalmente, se ha accedido al perfil del usuario 10 a través de la interfaz web y se ha podido visualizar el mismo.

Figura 3.51 Comprobación del funcionamiento de la base de datos desde el bash del contenedor con Ruby. Creación de un micropost con el contenido ‘Post prueba tarea 4’ y el usuario 10.

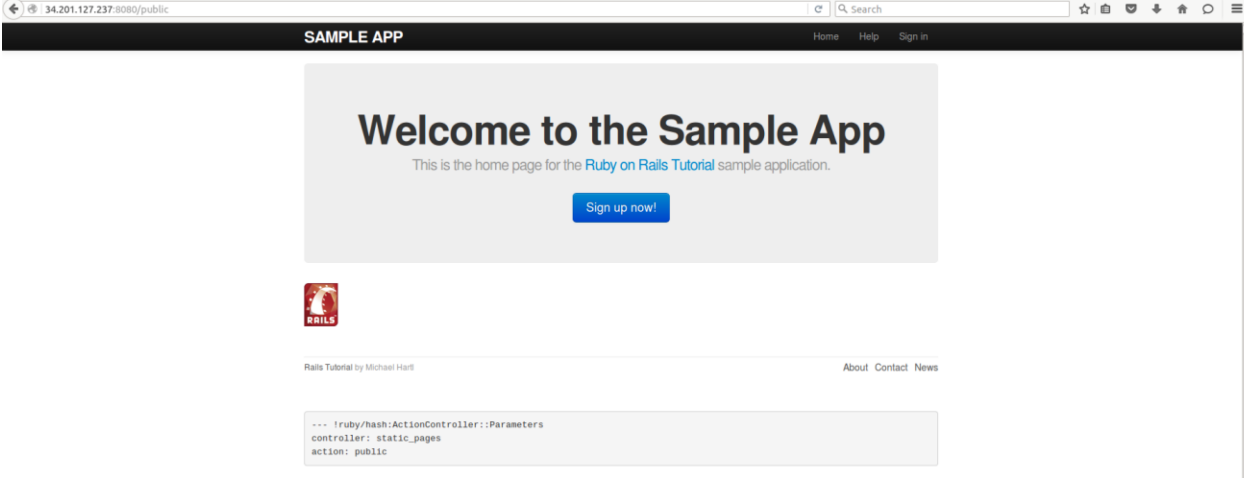

Figura 3.52 Acceso a la página principal de la aplicación desde la máquina con el servidor web Nginx.

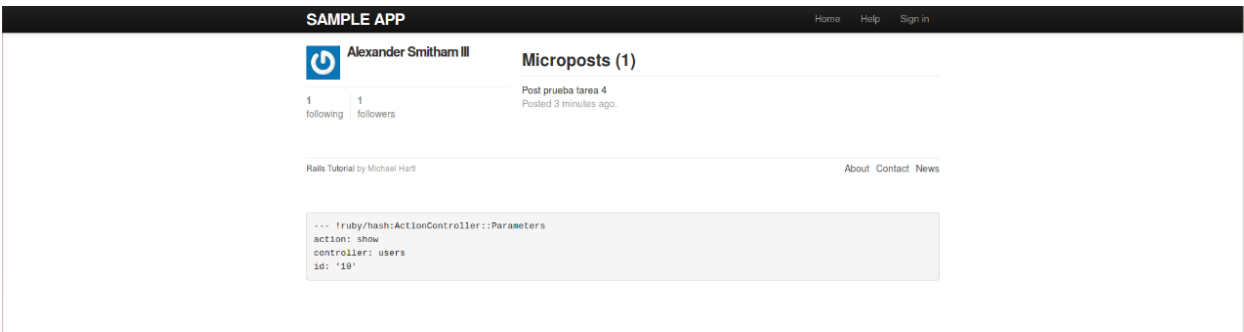

Figura 3.53 Acceso a la página del usuario con id 10 que muestra el micropost creado desde el 'rails console'.

Para comprobar el funcionamiento de Consul se puede acceder a la interfaz del servicio a través de la dirección de la máquina que corre el servidor de Consul por medio de su dirección ip y el puerto 8500. Esta página presenta una pestaña con los servicios, otra pestaña que muestra los nodos y otra que muestra las claves-valores.

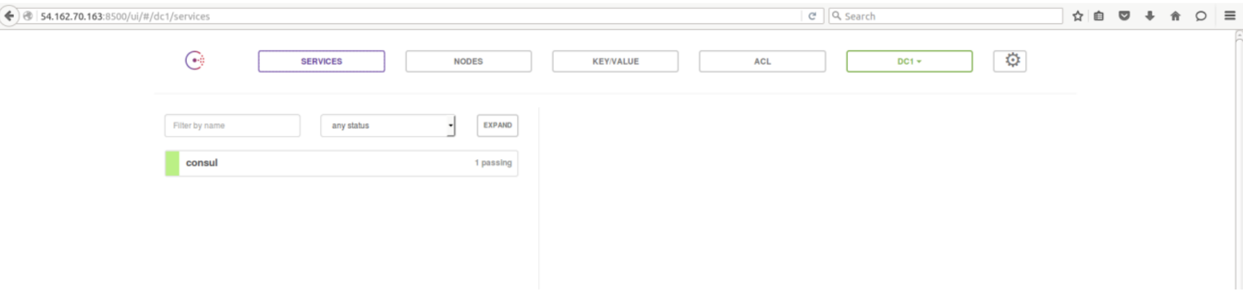

Figura 3.54 Acceso a la página principal de la interfaz de usuario de Consul.

Si se accede a la pestaña de 'nodes' se verán los nodos registrados en el servidor de Consul (figura 3.55).

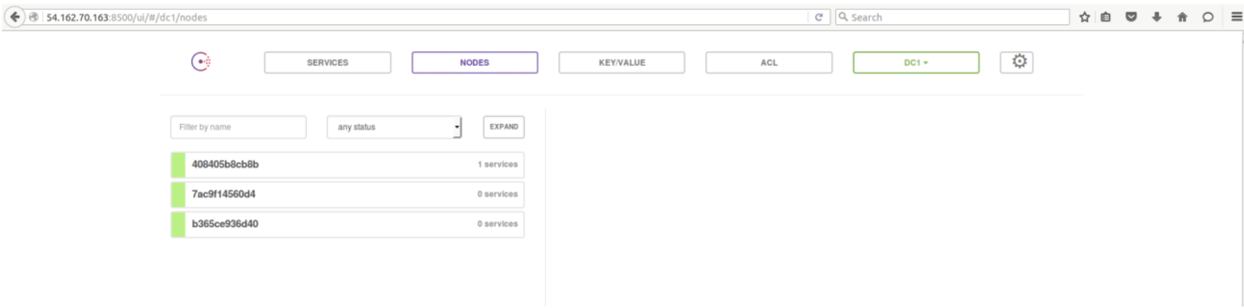

Figura 3.55 Nodos registrados en el servidor Consul.

En la pestaña de 'Key/Value' (figura 3.56) se muestran las direcciones ip privadas de los nodos agentes de Consul, éstos son la máquina que ejecuta la aplicación de Ruby y la máquina que ejecuta el servidor Nginx. Desde la pestaña 'Key/Value' se puede acceder también a la network overlay almacenada en el servidor de Consul (figura 3.57).

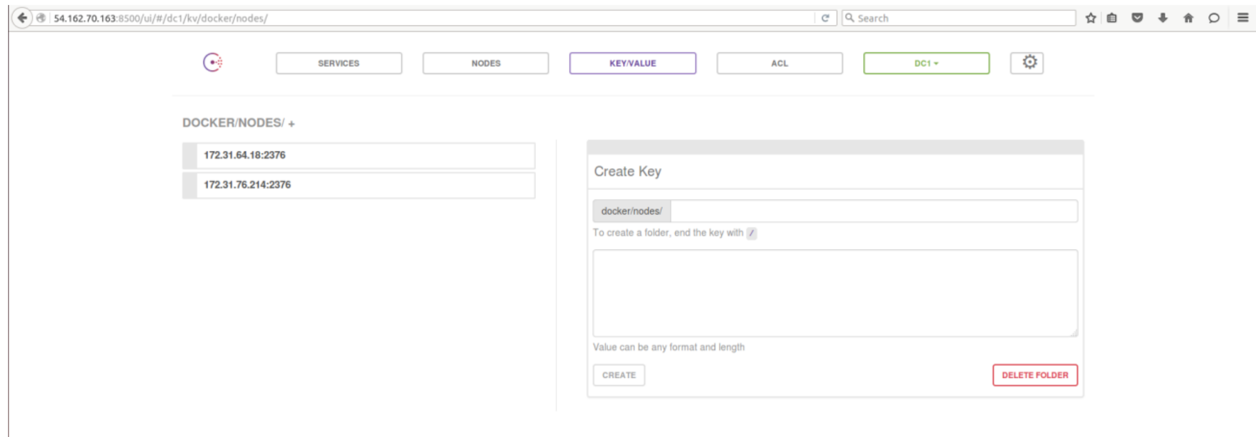

Figura 3.56 Penstaña 'Key/Value'. Ips privadas de las máquinas de Ruby y Nginx almacenadas en la key-value store.

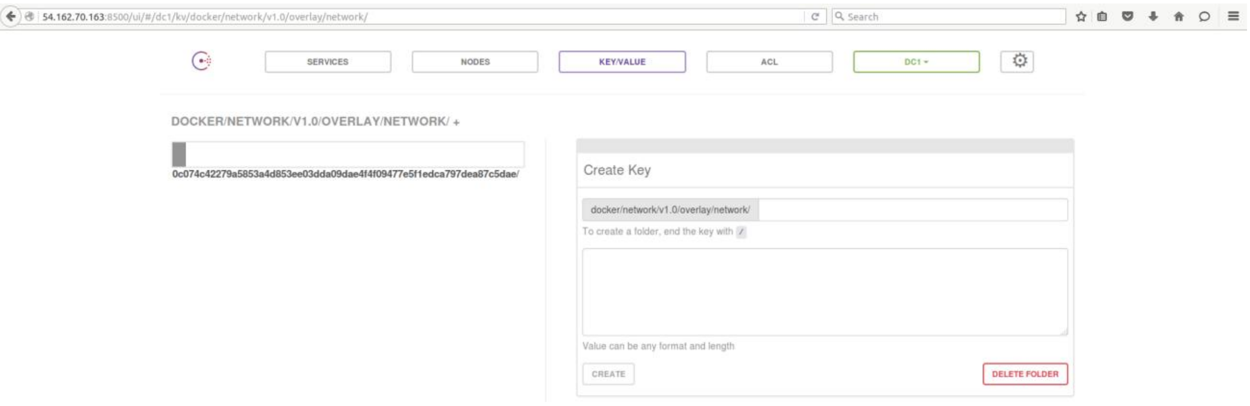

Figura 3.57 Network overlay almacenada en la key-value store.

3.6 Iteración 5: Despliegue de un clúster Swarm local con un maestro y 2 nodos para desplegar la infraestructura

En esta iteración se va a realizar el despliegue de la aplicación en un cluster local con un nodo manager y dos agentes, utilizando para crear el cluster Swarm Docker Machine con los flags '--swarm'. Ahora se tendrán 4 máquinas virtuales, una con Consul (consul-machine) y las otras tres formarán parte del clúster: una será el nodo máster (nginx-machine) y las otras dos serán esclavos (ruby-machine y postgres-machine).

Se ha separado el fichero docker-compose de la máquina Consul-Postgres en dos ficheros, uno para cada servicio que se será desplegado en máquinas diferentes. Además, se ha realizado una modificación en el script de la tarea 4 para que se creen ahora 4 máquinas y para que las máquinas pertenecientes al clúster se creen con la configuración de Swarm, indicando cuál es el nodo máster y el tipo de 'Swarm discovery', que en este caso utilizará Consul.

En esta iteración se ha incluido además el registro de servicios de los nodos del cluster en Consul por medio de la imagen 'gliderlabs/registrator', que registra y desregistra servicios de todos los contenedores de cada host. En Consul quedará registrada la ip y puerto desde el cual se ofrece el servicio. Para ello, se ha añadido las instrucciones necesarias al script 'tarea5-up.sh' para que se

descargue la imagen 'gliderlabs/registrator' en cada host perteneciente al cluster antes de realizar el despliegue de servicios con docker-compose.

3.6.1 Modificación de los ficheros de configuración para desplegar la aplicación en un cluster Swarm en local

En la figura 3.58 se muestra el fichero docker-compose de configuración de la máquina con el servicio de Consul, ahora creará y lanzará únicamente dicho servicio.

```
version: '2'
services:
  consul:
    image: progrium/consul
    restart: always
    ports:
      - "8500:8500" # REST API & UI
      - "8300:8300"
      - "8301:8301"
      - "8301:8301/udp"
      - "8302:8302"
      - "8302:8302/udp"
      - "8400:8400"
      - "53:53/udp" # DNS
    command:
      "-server -bootstrap -ui-dir /ui -advertise $CONSULIP"
    container_name: consul
```

Figura 3.58 Fichero *'docker-compose-tarea5-consul.yml'* para la configuración de Consul.

El nuevo fichero de configuración del servicio de Postgres se muestra en la figura 3.59. Ahora la máquina con Postgres será configurada como un agente más de Consul. También le será añadida a la configuración la red network overlay que será añadida como red externa, como en los otros nodos agentes.

```
version: '2'
services:
  some-postgres:
    build: postgres
    container_name: postgres
```

```
    ports:
      - 5432:5432
    environment:
      - POSTGRES_USER=${POSTGRES_USER}
      - POSTGRES_PASSWORD=${POSTGRES_PASSWORD}
    restart: always
  agent-2:
    image: progrium/consul
    container_name: consul_agent_2
    ports:
      - "8500:8500" # REST API & UI
      - "8300:8300"
      - "8301:8301"
      - "8301:8301/udp"
      - "8302:8302"
      - "8302:8302/udp"
      - "8400:8400"
      - "53:53/udp" # DNS
    environment:
      - "constraint:node==node2"
    command: -ui-dir /ui -join $CONSULIP -advertise $NODE2
networks:
  default:
    external:
      name: multi-host-net
```

Figura 3.59 Fichero *'docker-compose-tarea5-postgres.yml'* para la configuración de Postgres.

Los ficheros de configuración del servicio de Ruby (figura 3.60) y de Nginx (figura 3.61) se mantienen igual en esta iteración, con la única diferencia que se modifica el nombre del nodo en el fichero. Debido a que la máquina con Nginx pasará a ser el nodo manager, se ha renombrado la variable de entorno a definir a '$MANAGER'.

```
version: '2'
services:
  some-ruby:
    image: mayrez/sample_app_image:0.1
    ports:
      - 3000:3000
    external_links:
```

```
      - some-postgres:db
    working_dir: /usr/src/app
    environment:
      - POSTGRES_USER=$POSTGRES_USER
      - POSTGRES_PASSWORD=$POSTGRES_PASSWORD
      - POSTGRESIP=$POSTGRES
      - WEB_CONCURRENCY=1 # How many worker processes to run
      - RAILS_MAX_THREADS=16 # Configure to be the maximum number of threads
    restart: always
    command:
      /bin/bash -c \
        'cp config/database.yml.postgres config/database.yml &&
        rake db:setup &&
        rake db:migrate &&
        rake db:populate &&
        rails server'
  agent-3:
    image: progrium/consul
    container_name: consul_agent_3
    ports:
      - "8500:8500" # REST API & UI
      - "8300:8300"
      - "8301:8301"
      - "8301:8301/udp"
      - "8302:8302"
      - "8302:8302/udp"
      - "8400:8400"
      - "53:53/udp" # DNS
    environment:
      - "constraint:node==node3"
    command: -ui-dir /ui -join $CONSULIP -advertise $NODE3
networks:
  default:
    external:
      name: multi-host-net
```

Figura 3.60 Fichero '*docker-compose-tarea5-ruby.yml*' para la configuración de Ruby.

```
version: '2'

services:
```

```
  some-nginx:
    image: mayrez/some-nginx:0.1
    ports:
      - 8080:80
    external_links:
      - some-ruby:app
    restart: always
  agent-1:
    image: progrium/consul
    container_name: consul_agent_1
    ports:
      - "8500:8500" # REST API & UI
      - "8300:8300"
      - "8301:8301"
      - "8301:8301/udp"
      - "8302:8302"
      - "8302:8302/udp"
      - "8400:8400"
      - "53:53/udp" # DNS
    environment:
      - "constraint:node==node1"
    command: -ui-dir /ui -join $CONSULIP -advertise $MANAGER

networks:
  default:
    external:
      name: multi-host-net
```

Figura 3.61 Fichero *'docker-compose-tarea5-nginx.yml'* para la configuración de Nginx.

El script para el despliegue de la aplicación en cluster en local se muestra en la figura 3.62. La máquina con Consul se crea en el primer lugar. Cuando la máquina 'consul-machine' haya sido lanzada se creará el servicio de Consul en la misma. Para que no se creen las otras tres máquinas antes de que se haya creado el servicio de Consul, ya que se requiere que se encuentre disponible para realizar el 'Swarm doscovery', se ha añadido un script que evita que se prosiga hasta que haya sido arrancado. Luego se crean las máquinas manager (nginx-machine) y las dos agentes (postgres-machine y ruby-machine). Una vez creadas las máquinas se creará la red overlay desde el nodo manager. Con la red overlay creada se prosigue con la creación de los servicios en cada máquina. Para cada servicio se ha creado un script que compruebe que el servicio se encuentre en funcionamiento y que evita que se prosiga con el siguiente comando si no ha terminado de crearse.

```
#!/bin/bash

export POSTGRES_USER=postgres
export POSTGRES_PASSWORD=mysecretpassword
docker-machine create -d virtualbox consul-machine
docker-machine regenerate-certs consul-machine -f
eval $(docker-machine env consul-machine)
export CONSULIP=$(docker-machine ip consul-machine)
docker-compose -f docker-compose-tarea5-consul.yml up &

until  wget $(docker-machine ip consul-machine):8500 -O /dev/null -S \
--quiet 2>&1 | grep '200 OK' ; do
  >&2 echo "Consul server is not running - sleeping"
  sleep 1
done &&
echo "Consul server available"

docker-machine create -d virtualbox  \
--swarm --swarm-master \
--swarm-discovery="consul://$(docker-machine ip consul-machine):8500" \
--engine-opt="cluster-store=consul://$(docker-machine ip consul-machine):8500" \
--engine-opt="cluster-advertise=eth1:2376" \
nginx-machine

docker-machine regenerate-certs nginx-machine -f

docker-machine create -d virtualbox  \
--swarm \
--swarm-discovery="consul://$(docker-machine ip consul-machine):8500" \
--engine-opt="cluster-store=consul://$(docker-machine ip consul-machine):8500" \
--engine-opt="cluster-advertise=eth1:2376" \
ruby-machine

docker-machine regenerate-certs ruby-machine -f

docker-machine create -d virtualbox  \
--swarm  \
--swarm-discovery="consul://$(docker-machine ip consul-machine):8500" \
--engine-opt="cluster-store=consul://$(docker-machine ip consul-machine):8500" \
--engine-opt="cluster-advertise=eth1:2376" \
postgres-machine

docker-machine regenerate-certs postgres-machine -f

export MANAGER=$(docker-machine ip nginx-machine)
export NODE2=$(docker-machine ip postgres-machine)
```

```
export NODE3=$(docker-machine ip ruby-machine)

docker-machine ssh nginx-machine sudo docker network create -d overlay \
--subnet=10.0.9.0/24 multi-host-net

eval $(docker-machine env nginx-machine)
docker run -d --name=registrator -h $(docker-machine ip nginx-machine) \
-v=/var/run/docker.sock:/tmp/docker.sock gliderlabs/registrator:v6 \
consul://$(docker-machine ip consul-machine):8500

eval $(docker-machine env postgres-machine)
docker run -d --name=registrator -h $(docker-machine ip postgres-machine) \
-v=/var/run/docker.sock:/tmp/docker.sock gliderlabs/registrator:v6 \
consul://$(docker-machine ip consul-machine):8500

eval $(docker-machine env ruby-machine)
docker run -d --name=registrator -h $(docker-machine ip ruby-machine) \
-v=/var/run/docker.sock:/tmp/docker.sock gliderlabs/registrator:v6 \
consul://$(docker-machine ip consul-machine):8500

eval $(docker-machine env postgres-machine)
export POSTGRES=$(docker-machine ip postgres-machine)
docker-compose -f docker-compose-tarea5-postgres.yml up &

until PGPASSWORD="$POSTGRES_PASSWORD" psql -h "$POSTGRES" \
-U "$POSTGRES_USER" -c '\q'; do
  >&2 echo "Postgres is unavailable - sleeping"
  sleep 1
done &&

eval $(docker-machine env ruby-machine)
docker-compose -f docker-compose-tarea5-ruby.yml up &

until  wget $(docker-machine ip ruby-machine):3000/public -O /dev/null -S \
--quiet 2>&1 | grep '200 OK' ; do
  >&2 echo "Ruby server is not running - sleeping"
  sleep 1
done &&

eval $(docker-machine env nginx-machine)
docker-compose -f docker-compose-tarea5-nginx.yml up &

until  wget $(docker-machine ip nginx-machine):8080/public -O /dev/null -S \
--quiet 2>&1 | grep '200 OK' ; do
  >&2 echo "Cannot access application from Nginx server"
  sleep 1
```

```
done &&

echo "App available from Nginx server"
```

Figura 3.62 Fichero *'tarea5-up.sh'*, que contiene el script para el despliegue de la aplicación.

Los pasos para el despliegue y comprobación de la aplicación se muestran en la figura 3.63. Se clona la aplicación, se ejecuta el script del setup y se comprueba la conexión con la página por medio del comando 'curl'.

```
# git clone https://bitbucket.org/tftdp/docker_project

# cd Tarea5

# sh tarea5-up.sh

# curl $(docker-machine ip nginx-machine):8080/public
```

Figura 3.63 Pasos para el despliegue de la aplicación y comprobación de la conexión con la página principal de la aplicación de Ruby.

3.6.2 Resultados

Una vez las máquinas están lanzadas y se haya creado el cluster, al ejecutar 'docker-machine ls', como se ve en la figura 3.64, se pueden identificar los nodos pertenecientes al cluster y cuál es su nodo manager.

```
naox@naox-SATELLITE-P50-A-12Z:~/Desktop/docker_swarm_project/docker_project$ docker-machine ls
NAME              ACTIVE   DRIVER       STATE     URL                         SWARM                    DOCKER        ERRORS
consul-machine    -        virtualbox   Running   tcp://192.168.99.100:2376                            v17.05.0-ce
nginx-machine     -        virtualbox   Running   tcp://192.168.99.101:2376   nginx-machine (master)   v17.05.0-ce
postgres-machine  -        virtualbox   Running   tcp://192.168.99.103:2376   nginx-machine            v17.05.0-ce
ruby-machine      -        virtualbox   Running   tcp://192.168.99.102:2376   nginx-machine            v17.05.0-ce
naox@naox-SATELLITE-P50-A-12Z:~/Desktop/docker_swarm_project/docker_project$
```

Figura 3.64 Resultado ejecución 'docker-machine ls'.

Seleccionando el entorno del nodo máster ('nginx-machine') y ejecutando 'docker info' como se observa en las figuras 3.65 y 3.66, se puede ver la información de los nodos pertenecientes al cluster y el número de contenedores corriendo bajo el mismo.

```
# eval $(docker-machine env --swarm nginx-machine)
```

 # docker info

Figura 3.65 Comandos ver la información de los nodos del cluster

```
naox@naox-SATELLITE-P50-A-12Z:~/Desktop/docker_swarm_project/docker_project/Tarea5$ docker info
Containers: 4
 Running: 4
 Paused: 0
 Stopped: 0
Images: 3
Server Version: swarm/1.2.6
Role: primary
Strategy: spread
Filters: health, port, containerslots, dependency, affinity, constraint, whitelist
Nodes: 3
 nginx-machine: 192.168.99.103:2376
  └ ID: CSX6:TWGX:EIM3:WFV5:KNJI:ZFIO:E7WY:PHCS:OZ2U:C5LH:K4JX:DQ3D
  └ Status: Healthy
  └ Containers: 1 (1 Running, 0 Paused, 0 Stopped)
  └ Reserved CPUs: 0 / 1
  └ Reserved Memory: 0 B / 1.021 GiB
  └ Labels: kernelversion=4.4.66-boot2docker, operatingsystem=Boot2Docker 17.05.0-ce (TCL 7.2); HEAD : 5ed2840 - Fri May  5 21:04:09 UTC 2017, provider=virtualbox, storagedriver=aufs
  └ UpdatedAt: 2017-05-09T19:43:02Z
  └ ServerVersion: 17.05.0-ce
 postgres-machine: 192.168.99.101:2376
  └ ID: X4QZ:R5NF:4E5D:2HWU:G75L:RU6Q:XB7S:3MVQ:TYTJ:SJGZ:3XNA:DC7E
  └ Status: Healthy
  └ Containers: 2 (2 Running, 0 Paused, 0 Stopped)
  └ Reserved CPUs: 0 / 1
  └ Reserved Memory: 0 B / 1.021 GiB
  └ Labels: kernelversion=4.4.66-boot2docker, operatingsystem=Boot2Docker 17.05.0-ce (TCL 7.2); HEAD : 5ed2840 - Fri May  5 21:04:09 UTC 2017, provider=virtualbox, storagedriver=aufs
  └ UpdatedAt: 2017-05-09T19:43:00Z
  └ ServerVersion: 17.05.0-ce
 ruby-machine: 192.168.99.102:2376
  └ ID: LDES:VWN2:AVSA:VE2F:BJJI:ULUS:QTAN:GWDV:Z4N5:CIPK:UNAD:X5BU
  └ Status: Healthy
  └ Containers: 1 (1 Running, 0 Paused, 0 Stopped)
  └ Reserved CPUs: 0 / 1
  └ Reserved Memory: 0 B / 1.021 GiB
  └ Labels: kernelversion=4.4.66-boot2docker, operatingsystem=Boot2Docker 17.05.0-ce (TCL 7.2); HEAD : 5ed2840 - Fri May  5 21:04:09 UTC 2017, provider=virtualbox, storagedriver=aufs
  └ UpdatedAt: 2017-05-09T19:43:08Z
  └ ServerVersion: 17.05.0-ce
Plugins:
 Volume:
 Network:
Swarm:
 NodeID:
 Is Manager: false
 Node Address:
Security Options:
Kernel Version: 4.4.66-boot2docker
Operating System: linux
Architecture: amd64
CPUs: 3
Total Memory: 3.063 GiB
Name: 0473ada1403e
Docker Root Dir:
Debug Mode (client): false
Debug Mode (server): false
WARNING: No kernel memory limit support
```

Figura 3.66 Ejecución del comando 'docker info' desde el entorno del nodo máster para ver la información del clúster

Para comprobar la correcta conexión de la aplicación a la base de datos se ejecutará 'rails console' desde el contenedor de Ruby. Para ello, se busca el id del contenedor accediendo al entorno de la máquina desde la que se ha lanzado el contenedor, ejecutando 'docker ps' y accediendo a su bash ejecutando 'docker exec -it idContenedor bash' (figura 3.67).

$ eval $(docker-machine env ruby-machine)

$ docker ps

$ docker exec -it ed16a03dee8d bash

root@ed16a03dee8d:/usr/src/app# rails console

Loading development environment (Rails 4.0.8)

irb(main):001:0> User.first

User Load (1.0ms) SELECT "users". FROM "users" ORDER BY "users"."id" ASC LIMIT 1*

```
=> #<User id: 1, name: "Example User", email: "example@railstutorial.org",
created_at: "2017-04-24 16:12:45", updated_at: "2017-04-24 16:12:45",
password_digest:
"$2a$10$jshXYIy5vQNagWEJcsSMfe23y.oSrvXv.w/RwEyfc89V...",
remember_token: "ac60197bf21ccf2b2b60a3d4136a8db94977c24e", admin:
true>
```

Figura 3.67 Comandos para la comprobación de la conexión con la base de datos desde la máquina que contiene la aplicación de Ruby.

El resultado de la comprobación del funcionamiento de la aplicación en AWS se muestra en las figuras de la 3.67 a la 3.76.

Conectándose con la consola del contenedor de Ruby se ha accedido al ‘rails console’ y se ha creado un post con el contenido ‘Post prueba tarea 4’, asignado al usuario 10 (figura 3.68). Accediendo a la aplicación a través de la dirección ip del nodo manager, se comprueba que se muestra el nuevo post creado en el perfil del usuario con id 10, accediendo a ésta con la url ‘ipNodoNginx:8080/users/10’ (figura 3.70).

```
naox@naox-SATELLITE-P50-A-12Z:~$ eval $(docker-machine env ruby-machine)
naox@naox-SATELLITE-P50-A-12Z:~$ docker ps
CONTAINER ID        IMAGE                        COMMAND                  CREATED             STATUS              PORTS
                                                      NAMES
182c1f75211f        mayrez/sample_app_image:0.1   "/bin/bash -c ' cp co"   5 minutes ago       Up 4 minutes        0.0.0.0:3000->3000/tcp
                                                      tarea5_some-ruby_1
42d876684e0f        progrium/consul              "/bin/start -ui-dir /"   5 minutes ago       Up 4 minutes        0.0.0.0:53->53/udp, 0.0.0.0:8300-8302->8300-8302/tcp, 0.0.0.0:8400->8400/tcp, 53/tcp, 0.0.
0.0:8301-8302->8301-8302/udp, 0.0.0.0:8500->8500/tcp   consul_agent_2
a2332cd6c02e        swarm:latest                 "/swarm join --advert"   11 minutes ago      Up 8 minutes        2375/tcp
                                                      swarm-agent
naox@naox-SATELLITE-P50-A-12Z:~$ docker exec -it 182c1f75211f  bash
root@182c1f75211f:/usr/src/app# rails console
Loading development environment (Rails 4.0.8)
irb(main):001:0> User.all
  User Load (3.3ms)  SELECT "users".* FROM "users"
=> #<ActiveRecord::Relation [#<User id: 1, name: "Example User", email: "example@railstutorial.org", created_at: "2017-05-09 17:53:17", updated_at: "2017-05-09 17:53:17", password_digest: "$2a$10$3nb8JJQxi
zpHeFsfWi9G.ug1aRMJU/K6pUfWcDaokOfN...", remember_token: "9e3597f068b8c2d449cae15211c5957e278f2bfb", admin: true>, #<User id: 2, name: "Mr. George Murphy", email: "example-1@railstutorial.org", created_at:
 "2017-05-09 17:53:18", updated_at: "2017-05-09 17:53:18", password_digest: "$2a$10$7Glopgph7eY1pvG9PuDpWeyhkzn9MDELrEimC7p5GV6H...", remember_token: "1b24c1cf49aab85f44106a6995000de558c63b02", admin: nil>
, #<User id: 3, name: "Dr. Jaida Ebert", email: "example-2@railstutorial.org", created_at: "2017-05-09 17:53:18", updated_at: "2017-05-09 17:53:18", password_digest: "$2a$10$kTK7XS3TQ/HhShtNfxhwFevAyLhLmYH
VEdeQUVPM78SR...", remember_token: "469aebb120e273070c7f57236fbbad314adb32c4", admin: nil>, #<User id: 4, name: "Lavina Hodkiewicz", email: "example-3@railstutorial.org", created_at: "2017-05-09 17:53:18",
 updated_at: "2017-05-09 17:53:18", password_digest: "$2a$10$lbw8u9cnN8Ft.qCMXL7mN.nbsdeUB6yYa7DHbdAnsp1s...", remember_token: "5411ba314650487bc6638c0d974081b268ce0b8a", admin: nil>, #<User id: 5, name: "
Hershel Reichert", email: "example-4@railstutorial.org", created_at: "2017-05-09 17:53:18", updated_at: "2017-05-09 17:53:18", password_digest: "$2a$10$4WOWI6hjV53dy16XU.CbUe5iZhef7wo1xzpRzzhEI89Q...", rem
ember_token: "68fe2170c4d1809c5830a4f781ddaee659c5cf6d", admin: nil>, #<User id: 6, name: "Martina Barton", email: "example-5@railstutorial.org", created_at: "2017-05-09 17:53:18", updated_at: "2017-05-09
17:53:18", password_digest: "$2a$10$cZYP.W/pD0w8oOjn.lFb7uhubX7JausSrJR4sobw.f3w...", remember_token: "db64dac1ef56803f470dfb26dab1355bc257852b", admin: nil>, #<User id: 7, name: "Mr. Rozella Kuphal", emai
l: "example-6@railstutorial.org", created_at: "2017-05-09 17:53:18", updated_at: "2017-05-09 17:53:18", password_digest: "$2a$10$LxnyX.MDyzCV8iv/T1.7V.up/JDwcjFjmmRysdFcLVOA...", remember_token: "057a424a8
3d8e57bf003fcbf612940d5d87f233e", admin: nil>, #<User id: 8, name: "Muriel Parker", email: "example-7@railstutorial.org", created_at: "2017-05-09 17:53:18", updated_at: "2017-05-09 17:53:18", password_dige
st: "$2a$10$KnYVleo3nb/xGNn75pb2pOVEcOPeaEKvtzwAPGz48QFk...", remember_token: "725083174c7be58f4ebe6138942b7f5f2e1c59c1", admin: nil>, #<User id: 9, name: "Mrs. Sigmund Walker", email: "example-8@railstuto
rial.org", created_at: "2017-05-09 17:53:18", updated_at: "2017-05-09 17:53:18", password_digest: "$2a$10$C8W73spE7QhFIydeReQoSuVN3dlx/N66J75uOJz8KXOf...", remember_token: "ca0e0c6f8b7cee4714a39016b7caeb6d
0fff8810", admin: nil>, #<User id: 10, name: "Isaias Bashirian I", email: "example-9@railstutorial.org", created_at: "2017-05-09 17:53:18", updated_at: "2017-05-09 17:53:18", password_digest: "$2a$10$7juAD
NL0fTD.Y76eScdcteMk6F3GXKhH7XkTyxQjP4uI...", remember_token: "4b6ee69196963d36d4e2c79db5136dc2474ffbdc", admin: nil>, ...]>
irb(main):002:0> Micropost
=> Micropost(id: integer, content: string, user_id: integer, created_at: datetime, updated_at: datetime)
irb(main):003:0> Micropost.create :content=>"Prueba post tarea 5"
   (0.5ms)  BEGIN
   (0.6ms)  ROLLBACK
=> #<Micropost id: nil, content: "Prueba post tarea 5", user_id: nil, created_at: nil, updated_at: nil>
irb(main):004:0> Micropost.create :content=>"Prueba post tarea 5" :user_id=>10
SyntaxError: (irb):4: syntax error, unexpected ':', expecting end-of-input
Micropost.create :content=>"Prueba post tarea 5" :user_id=>10
                                                  ^
        from /usr/local/bundle/gems/railties-4.0.8/lib/rails/commands/console.rb:90:in `start'
        from /usr/local/bundle/gems/railties-4.0.8/lib/rails/commands/console.rb:9:in `start'
        from /usr/local/bundle/gems/railties-4.0.8/lib/rails/commands.rb:62:in `<top (required)>'
        from bin/rails:4:in `require'
        from bin/rails:4:in `<main>'
irb(main):005:0> Micropost.create :content=>"Prueba post tarea 5", :user_id=>10
   (0.7ms)  BEGIN
  SQL (4.8ms)  INSERT INTO "microposts" ("content", "created_at", "updated_at", "user_id") VALUES ($1, $2, $3, $4) RETURNING "id"  [["content", "Prueba post tarea 5"], ["created_at", Tue, 09 May 2017 17:59
:35 UTC +00:00], ["updated_at", Tue, 09 May 2017 17:59:35 UTC +00:00], ["user_id", 10]]
   (33.7ms)  COMMIT
=> #<Micropost id: 301, content: "Prueba post tarea 5", user_id: 10, created_at: "2017-05-09 17:59:35", updated_at: "2017-05-09 17:59:35">
irb(main):006:0> Micropost.find(301)
  Micropost Load (1.5ms)  SELECT "microposts".* FROM "microposts" WHERE "microposts"."id" = $1 ORDER BY created_at DESC LIMIT 1  [["id", 301]]
=> #<Micropost id: 301, content: "Prueba post tarea 5", user_id: 10, created_at: "2017-05-09 17:59:35", updated_at: "2017-05-09 17:59:35">
irb(main):007:0>
```

Figura 3.68 Comprobación del funcionamiento de la base de datos desde el bash del contenedor con Ruby. Creación de un micropost con el contenido ‘Post prueba tarea 4’ y el usuario 10.

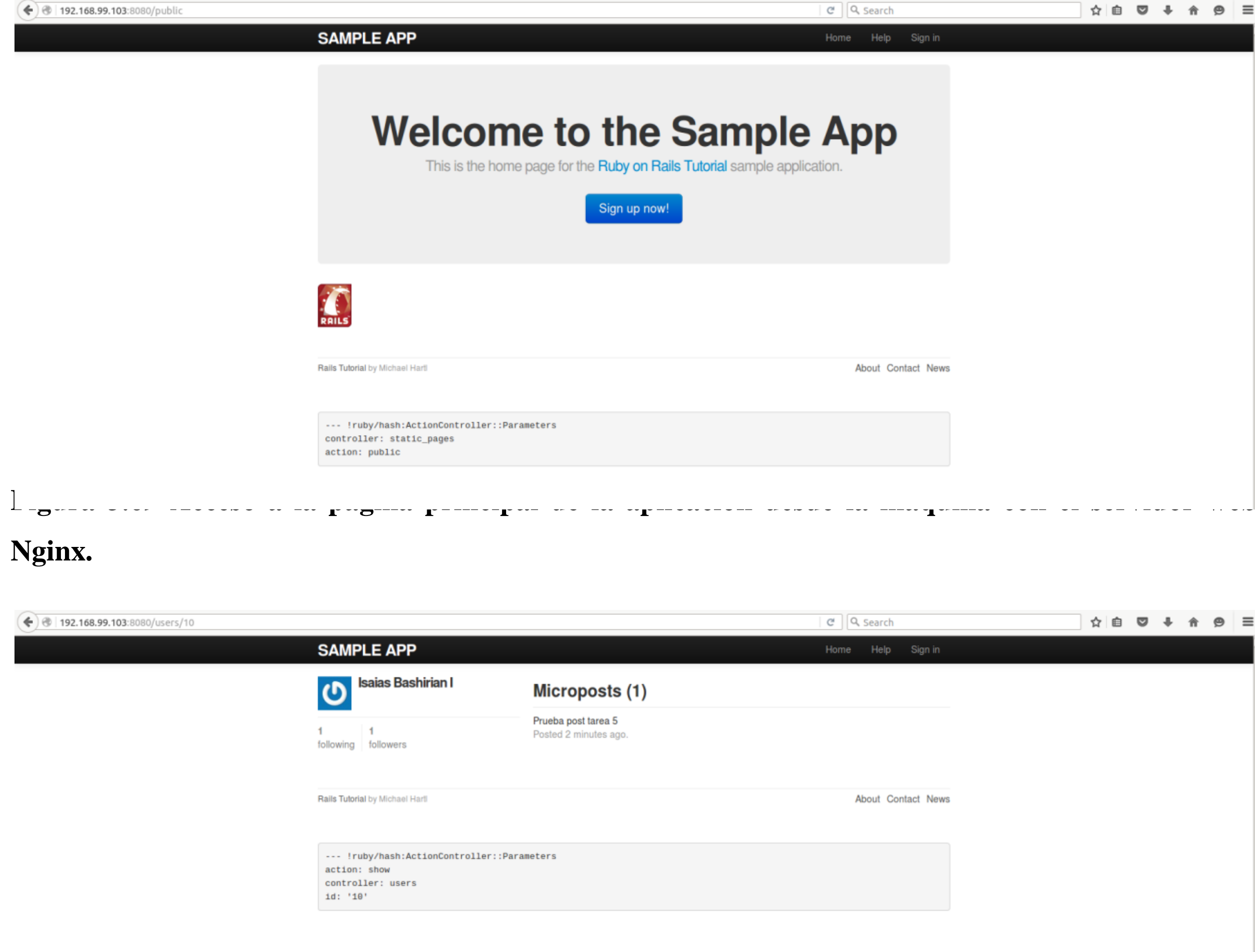

Figura 3.69 Acceso a la página principal de la aplicación desde la máquina con el servidor web Nginx.

Figura 3.70 Acceso a la página del usuario con id 10 que muestra el micropost creado desde el 'rails console'.

Para comprobar el funcionamiento de Consul se puede acceder a la interfaz del servicio por medio de la máquina con el servidor de Consul a través de su dirección ip y el puerto 8500. Esta página presenta una pestaña con los servicios, otra pestaña que muestra los nodos y otra que muestra las claves-valores almacenados.

En la pestaña 'services' (figura 3.71) se muestran los servicios pertenecientes a los nodos del cluster registrados a través del contenedor 'registrator' ejecutado en cada nodo. Si se accede a los servicios se mostrará la ip y el puerto desde la que se ofrecen los servicios (figura 3.72).

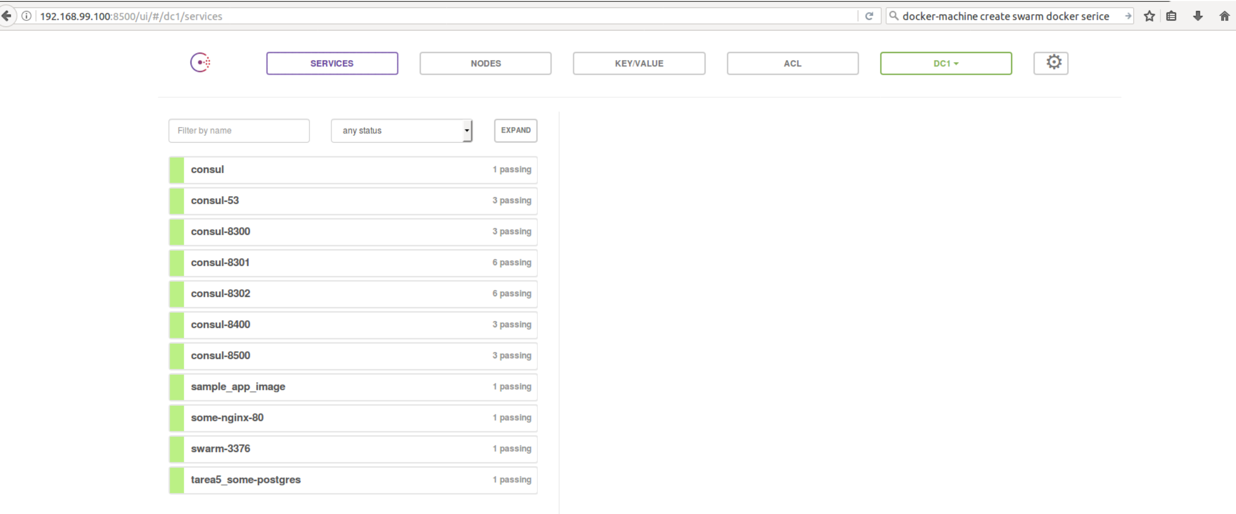

Figura 3.71 Acceso a la página de los servicios registrados en Consul.

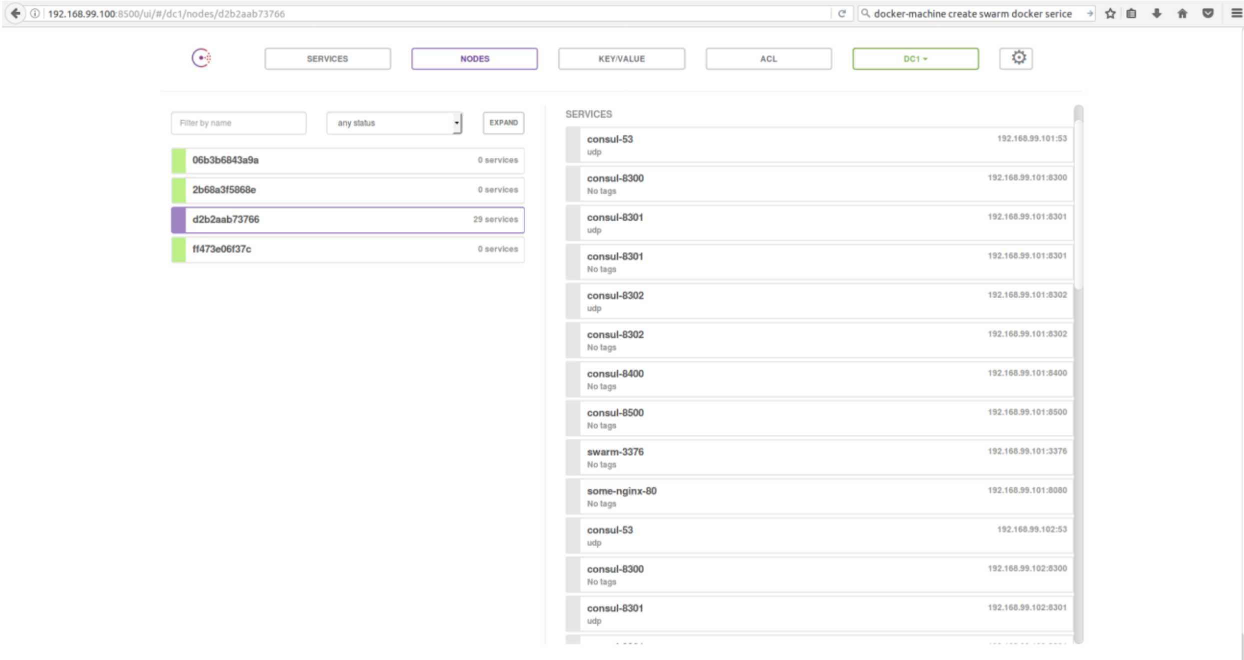

Figura 3.72 Nodo Consul con los servicios del cluster registrados y descripción de ip y puerto desde el que se ofrece el servicio.

El la pestaña de los key-value (figura 3.73) se muestran las direcciones ip de los nodos agentes de Consul y ahora además tendrá un registro con el valor almacenado para el cluster swarm (Figura 3.75).

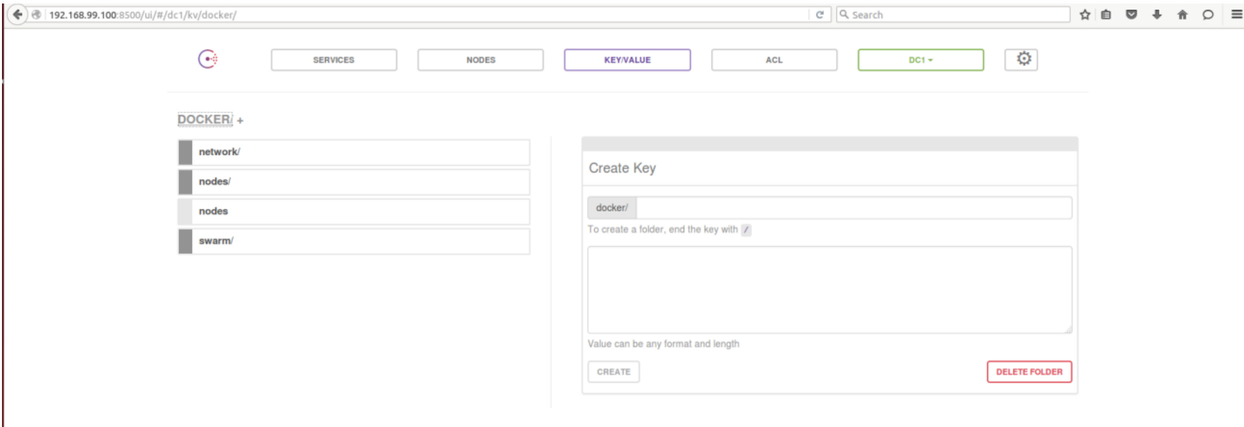

Figura 3.73 Valores almacenados en el key-value store. Ahora además de la red incluye el Swarm

Accediendo a ‘nodes’ desde la pestaña de ‘Key/Value’ (figura 3.74) se ven almacenados los valores de ip y puerto Docker de los tres nodos agentes (Nginx, Postgres y Ruby).

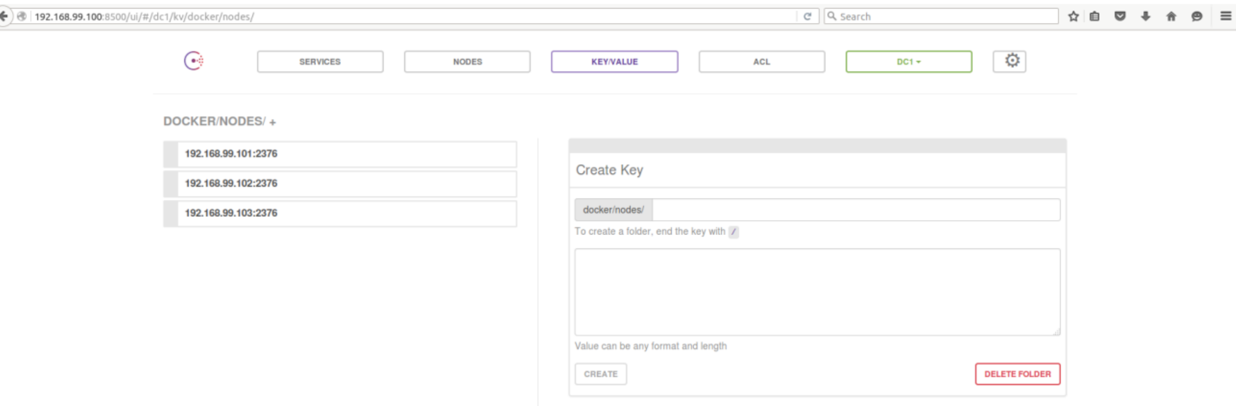

Figura 3.74 Pestaña ‘Key/Value’ sección ‘nodes’.

El registro de la network overlay se puede ver accediendo también a la pestaña ‘Key/Value’ como se muestra en la figura 3.75.

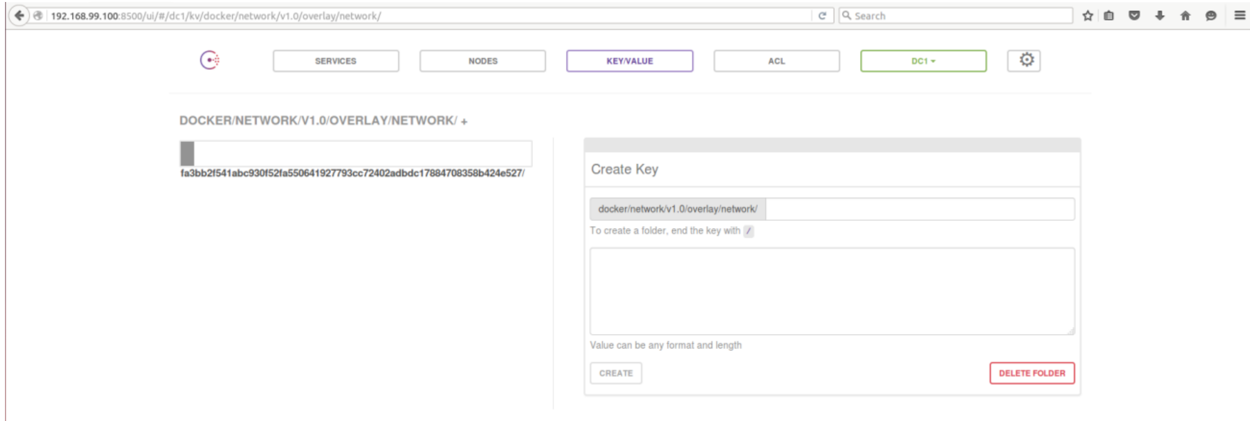

Figura 3.75 Network overlay almacenada en la key-value store.

Como se muestra en la figura 3.76 también se muestran los valores almacenados para el cluster Swarm, que son los valores de ip con el puerto Docker de cada nodo perteneciente al cluster, el manager y los dos esclavos.

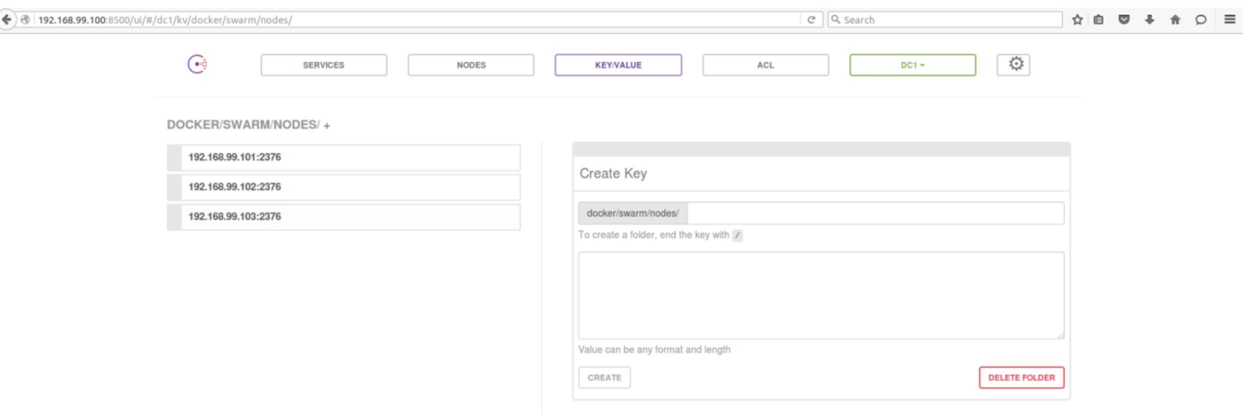

Figura 3.76 Nodos pertenecientes al Swarm almacenados en la key-value store.

Se pueden listar los nodos del clúster con el siguiente comando de la figura 3.77.

```
# docker run swarm list consul://$(docker-machine ip consul-machine):8500
```

Figura 3.77 Comando para listar los nodos pertenecientes a un cluster utilizando Consul

```
naox@naox-SATELLITE-P50-A-12Z:~/Desktop/docker_swarm_project/docker_project/Tare
a5$ docker run swarm list consul://$(docker-machine ip consul-machine):8500
time="2017-05-09T18:07:02Z" level=info msg="Initializing discovery without TLS"
192.168.99.101:2376
192.168.99.102:2376
192.168.99.103:2376
naox@naox-SATELLITE-P50-A-12Z:~/Desktop/docker_swarm_project/docker_project/Tare
a5$
```

Figura 3.78 Listado de nodos pertenecientes al cluster

Para ver los contenedores ejecutándose en el cluster swarm, se accede al entorno del nodo máster (‘eval $(docker-machine env --swarm nginx-master)’) y se ejecuta ‘docker ps’ (figura 3.79 y 3.80)

```
# eval $(docker-machine env --swarm nginx-master)
# docker ps
```

Figura 3.79 Comando para listar los contenedores ejecutándose en el cluster Swarm.

```
naox@naox-SATELLITE-P50-A-12Z:~/Desktop/docker_swarm_project/docker_project$ eval $(docker-machine env --swarm nginx-machine)
naox@naox-SATELLITE-P50-A-12Z:~/Desktop/docker_swarm_project/docker_project$ docker ps
CONTAINER ID        IMAGE                          COMMAND                  CREATED              STATUS               PORTS
                                                                                                NAMES
6d3eb4d28e7b        progrium/consul                "/bin/start -ui-di..."   About a minute ago   Up About a minute    192.168.99.101:53->53/udp, 192.168.99.101:8300-8302->8300-8302/tcp, 192.168.99.101:8400->
8400/tcp, 53/tcp, 192.168.99.101:8301-8302->8301-8302/udp, 192.168.99.101:8500->8500/tcp   nginx-machine/consul_agent_1
23087fdc1fa9        mayrez/some-nginx:0.1          "nginx -g 'daemon ..."   About a minute ago   Up About a minute    443/tcp, 192.168.99.101:8080->80/tcp
                                                                                                nginx-machine/tarea5_some-nginx_1
4ed21841c3e7        mayrez/sample_app_image:0.1    "/bin/bash -c ' cp..."   4 minutes ago        Up 4 minutes         192.168.99.102:3000->3000/tcp
                                                                                                ruby-machine/tarea5_some-ruby_1
dc1ebd137be2        progrium/consul                "/bin/start -ui-di..."   4 minutes ago        Up 4 minutes         192.168.99.102:53->53/udp, 192.168.99.102:8300-8302->8300-8302/tcp, 192.168.99.102:8400->
8400/tcp, 53/tcp, 192.168.99.102:8301-8302->8301-8302/udp, 192.168.99.102:8500->8500/tcp   ruby-machine/consul_agent_3
318f317ccbf0        progrium/consul                "/bin/start -ui-di..."   7 minutes ago        Up 7 minutes         192.168.99.103:53->53/udp, 192.168.99.103:8300-8302->8300-8302/tcp, 192.168.99.103:8400->
8400/tcp, 53/tcp, 192.168.99.103:8301-8302->8301-8302/udp, 192.168.99.103:8500->8500/tcp   postgres-machine/consul_agent_2
f59d5a90fd10        tarea5_some-postgres           "docker-entrypoint..."   7 minutes ago        Up 7 minutes         192.168.99.103:5432->5432/tcp
                                                                                                postgres-machine/postgres
naox@naox-SATELLITE-P50-A-12Z:~/Desktop/docker_swarm_project/docker_project$
```

Figura 3.80 Contenedores ejecutándose dentro del cluster Swarm.

3.7 Iteración 6: Despliegue de un clúster con un maestro y 2 nodos agentes de manera remota en AWS

Para desplegar la aplicación en un cluster Swarm en AWS se utilizará el mismo script que en la tarea 5, pero en este caso se crearán las máquinas como instancias de Amazon EC2 utilizando para ello en la creación como driver 'amazonec2' en lugar de 'virtualbox'. También se modificará la interfaz por la que se hace el 'cluster-advertise' a eth0 puesto que en estas máquinas no se encuentra la eth1.

Al igual que en la iteración anterior, se utilizará el servicio ‘registrator’ de la imagen ‘gliderlabs/registrator’ para registrar en Consul los servicios ejecutándose en los nodos del cluster.

3.7.1 Creación de los grupos de seguridad

Se crearán tres grupos de seguridad desde el dashboard del EC2 dentro del apartado de 'Security Groups' de la consola de AWS: uno para la máquina de Consul (Security Group con nombre Consul), otro para la máquina con Postgres (Security Group con nombre Postgres) otro para la máquína con Ruby (Security Group con nombre Ruby) y la última para el servidor web Nginx (Security Group con nombre Nginx). En todos ellos se añadirá como 'inbound rules' la apertura de puertos (selección de 'anywhere'):

- 22 TCP (SSH)
- 2376 TCP (Docker)
- 8500 TCP, 8300 - 8302 TCP, 8301-8302 UDP, 53 UDP (Consul)

Otras reglas 'inbound' a añadir son las siguientes:

- En el grupo de seguridad de Postgres se abrirá además el puerto de Postgres, el 5432 TCP.
- En el de Ruby se abrirá el puerto 3000 TCP.
- En el Nginx (máster) se abrirá el puerto 8080 TCP y además el puerto 3376 TCP, que se abre en el nodo máster para las comunicaciones en el Swarm.
- Para el funcionamiento de la network overlay se abrirán en los grupos de seguridad de Ruby y Nginx los puertos 7946 TCP, 4789 UDP y 7946 UDP.

En la figura 3.81 se muestran los 'Security Groups' registrados en AWS para el funcionamiento de los servicios del cluster.

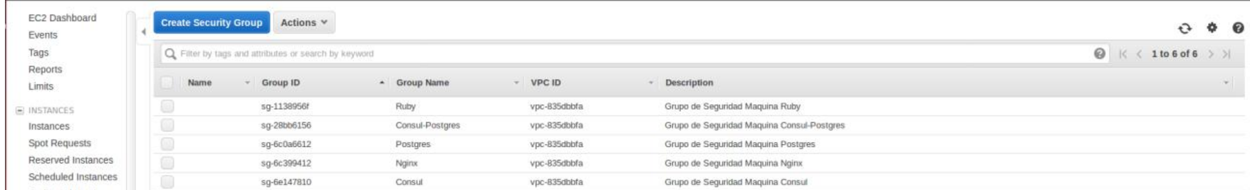

Figura 3.81 'Security Groups' registrados en AWS para el funcionamiento de los servicios del cluster.

3.7.2 Modificación del script de despliegue para desplegar el cluster en AWS

En la figura 3.82 se muestra el script modificado para desplegar el cluster Swarm en AWS, añadiendo para ello los flag necesarios de AWS en la ejecución del comando 'docker-machine create'.

El 'AWS_VPC_ID' es el id del Virtual Private Cloud que se puede ver en 'Security Groups' en la columna 'VPC ID'. El 'AWS_SECURITY_GROUP' es el nombre dado al grupo de seguridad creado en la consola de AWS.

En la figura 3.82 se muestra el script del setup para la iteración 6, el fichero '*tarea6-up.sh'*, con las modificaciones necesarias al script previo de la iteración 5 para que se ejecute en AWS.

```
#!/bin/bash

export POSTGRES_USER=postgres
export POSTGRES_PASSWORD=mysecrectpassword
export AWS_ACCESS_KEY_ID=$aws_access_key_id
export AWS_SECRET_ACCESS_KEY=$aws_secret_access_key
export AWS_VPC_ID=vpc-835dbbfa
export AWS_DEFAULT_REGION=us-east-1
export AWS_ZONE=c
export AWS_SECURITY_GROUP=Consul
docker-machine -D create --driver amazonec2 \
  --amazonec2-access-key $AWS_ACCESS_KEY_ID \
  --amazonec2-secret-key $AWS_SECRET_ACCESS_KEY \
  --amazonec2-vpc-id $AWS_VPC_ID \
  --amazonec2-security-group $AWS_SECURITY_GROUP \
  --amazonec2-region $AWS_DEFAULT_REGION \
  --amazonec2-zone $AWS_ZONE \
consul-machine

export CONSULIP=$(docker-machine ip consul-machine)
eval $(docker-machine env consul-machine)
docker-compose -f docker-compose-tarea6-consul.yml up &

until  wget $(docker-machine ip consul-machine):8500 -O /dev/null -S \
--quiet 2>&1 | grep '200 OK' ; do
  >&2 echo "Consul server is not running - sleeping"
```

```
  sleep 1
done &&
echo "Consul server available"

export AWS_SECURITY_GROUP=Nginx
docker-machine -D create --driver amazonec2 \
  --swarm --swarm-master \
  --swarm-discovery="consul://$(docker-machine ip consul-machine):8500" \
  --amazonec2-access-key $AWS_ACCESS_KEY_ID \
  --amazonec2-secret-key $AWS_SECRET_ACCESS_KEY \
  --amazonec2-vpc-id $AWS_VPC_ID \
  --amazonec2-security-group $AWS_SECURITY_GROUP \
  --amazonec2-region $AWS_DEFAULT_REGION \
  --amazonec2-zone $AWS_ZONE \
  --engine-opt="cluster-store=consul://$(docker-machine ip consul-machine):8500" \
  --engine-opt="cluster-advertise=eth0:2376" \
nginx-machine
export AWS_SECURITY_GROUP=Postgres
docker-machine -D create --driver amazonec2 \
  --swarm \
  --swarm-discovery="consul://$(docker-machine ip consul-machine):8500" \
  --amazonec2-access-key $AWS_ACCESS_KEY_ID \
  --amazonec2-secret-key $AWS_SECRET_ACCESS_KEY \
  --amazonec2-vpc-id $AWS_VPC_ID \
  --amazonec2-security-group $AWS_SECURITY_GROUP \
  --amazonec2-region $AWS_DEFAULT_REGION \
  --amazonec2-zone $AWS_ZONE \
  --engine-opt="cluster-store=consul://$(docker-machine ip consul-machine):8500" \
  --engine-opt="cluster-advertise=eth0:2376" \
postgres-machine

export AWS_SECURITY_GROUP=Ruby
docker-machine -D create --driver amazonec2 \
  --swarm \
  --swarm-discovery="consul://$(docker-machine ip consul-machine):8500" \
  --amazonec2-access-key $AWS_ACCESS_KEY_ID \
  --amazonec2-secret-key $AWS_SECRET_ACCESS_KEY \
  --amazonec2-vpc-id $AWS_VPC_ID \
  --amazonec2-security-group $AWS_SECURITY_GROUP \
  --amazonec2-region $AWS_DEFAULT_REGION \
  --amazonec2-zone $AWS_ZONE  \
  --engine-opt="cluster-store=consul://$(docker-machine ip consul-machine):8500" \
  --engine-opt="cluster-advertise=eth0:2376" \
ruby-machine

export MANAGER=$(docker-machine ip nginx-machine)
```

```
export NODE2=$(docker-machine ip postgres-machine)
export NODE3=$(docker-machine ip ruby-machine)

docker-machine ssh nginx-machine sudo docker network create -d overlay \
--subnet=10.0.9.0/24 multi-host-net

eval $(docker-machine env nginx-machine)
docker run -d --name=registrator -h $(docker-machine ip nginx-machine) \
-v=/var/run/docker.sock:/tmp/docker.sock gliderlabs/registrator:v6 \
consul://$(docker-machine ip consul-machine):8500
eval $(docker-machine env postgres-machine)
docker run -d --name=registrator -h $(docker-machine ip postgres-machine) \
-v=/var/run/docker.sock:/tmp/docker.sock gliderlabs/registrator:v6 \
consul://$(docker-machine ip consul-machine):8500
eval $(docker-machine env ruby-machine)
docker run -d --name=registrator -h $(docker-machine ip ruby-machine) \
-v=/var/run/docker.sock:/tmp/docker.sock gliderlabs/registrator:v6 \
consul://$(docker-machine ip consul-machine):8500

eval $(docker-machine env postgres-machine)
export POSTGRES=$(docker-machine ip postgres-machine)
docker-compose -f docker-compose-tarea6-postgres.yml up &

until PGPASSWORD="$POSTGRES_PASSWORD" psql -h "$POSTGRES" \
-U "$POSTGRES_USER" -c '\q'; do
  >&2 echo "Postgres is unavailable - sleeping"
  sleep 1
done &&
echo "Postgres available"

eval $(docker-machine env ruby-machine)
export CONSULIP=$(docker-machine ip consul-machine)
docker-compose -f docker-compose-tarea6-ruby.yml up &

until  wget $(docker-machine ip ruby-machine):3000/public -O /dev/null -S \
--quiet 2>&1 | grep '200 OK' ; do
  >&2 echo "Ruby server is not running - sleeping"
  sleep 1
done &&

eval $(docker-machine env nginx-machine)
docker-compose -f docker-compose-tarea6-nginx.yml up &

until  wget $(docker-machine ip nginx-machine):8080/public -O /dev/null -S \
--quiet 2>&1 | grep '200 OK' ; do
  >&2 echo "Cannot access application from Nginx server"
```

```
  sleep 1
done &&

echo "App available from Nginx server"
```

Figura 3.82 Fichero *'tarea6-up.sh'*, que contiene el script para el despliegue de la aplicación.

3.7.3 Despliegue del cluster Swarm en AWS

Los comandos a ejecutar para llevar a cabo el despliegue son los siguientes se muestran en la figura 3.83. Se clona la aplicación se hace 'cd' en la carpeta con los ficheros para la iteración 6, se ejecuta el script de setup y, finalmente, se comprueba la aplicación por medio del comando 'curl'.

```
# git clone https://bitbucket.org/tftdp/docker_project

# cd Tarea6

# sh tarea6-up.sh

# curl $(docker-machine ip nginx-machine):8080/public
```

Figura 3.83 Pasos para el despliegue del cluster Swarm y comprobación de la conexión con la página principal de la aplicación de Ruby.

3.7.4 Resultados

Si se ejecuta 'docker-machine ls' se pueden ver los nodos pertenecientes al clúster y cuál es el máster (figura 3.84 y 3.85).

```
# docker-machine ls
```

Figura 3.84 Comando que muestra las máquinas ejecutándose

```
naox@naox-SATELLITE-P50-A-12Z:~$ docker-machine ls
NAME               ACTIVE   DRIVER      STATE     URL                          SWARM                    DOCKER        ERRORS
consul-machine     -        amazonec2   Running   tcp://54.234.156.197:2376                             v17.05.0-ce
nginx-machine      -        amazonec2   Running   tcp://52.87.233.238:2376     nginx-machine (master)   v17.05.0-ce
postgres-machine   -        amazonec2   Running   tcp://54.196.176.224:2376    nginx-machine            v17.05.0-ce
ruby-machine       *        amazonec2   Running   tcp://54.87.240.185:2376     nginx-machine            v17.05.0-ce
naox@naox-SATELLITE-P50-A-12Z:~$
```

pertenecientes al cluster Swarm y se indica quién es el máster.

Si se accede desde AWS a la consola del servicio EC2 en el apartado 'Running instances' aparecerán las 4 máquinas ejecutándose.

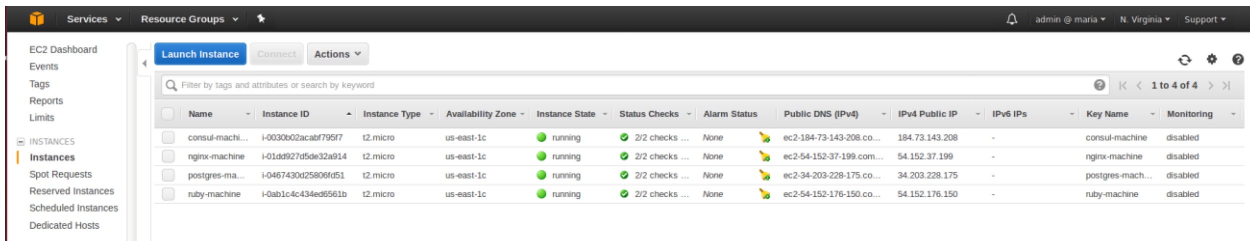

Figura 3.86 EC2 dashboard, máquinas ejecutándose en AWS

Para comprobar la conexión con la base de datos, nos conectamos a la máquina con el contenedor de Ruby (ruby-machine) y ejecutamos 'rails console' (figura 3.87).

```
# eval $(docker-machine env ruby-machine)

# docker ps

# docker exec -it ed16a03dee8d bash

root@ed16a03dee8d:/usr/src/app# rails console

Loading development environment (Rails 4.0.8)

irb(main):001:0> User.first

  User Load (1.0ms)  SELECT "users".* FROM "users" ORDER BY "users"."id" ASC LIMIT 1

=> #<User id: 1, name: "Example User", email: "example@railstutorial.org", created_at: "2017-04-24 16:12:45", updated_at: "2017-04-24 16:12:45", password_digest: "$2a$10$jshXYIy5vQNagWEJcsSMfe23y.oSrvXv.w/RwEyfc89V...", remember_token: "ac60197bf21ccf2b2b60a3d4136a8db94977c24e", admin: true>
```

Figura 3.87 Comprobación de la conexión con la base de datos desde el contenedor con la aplicación.

Desde el ‘rails console’ se crea un nuevo post con el contenido ‘Post de prueba tarea 6 user 8’ asignándoselo al usuario 8 (figura 3.88).

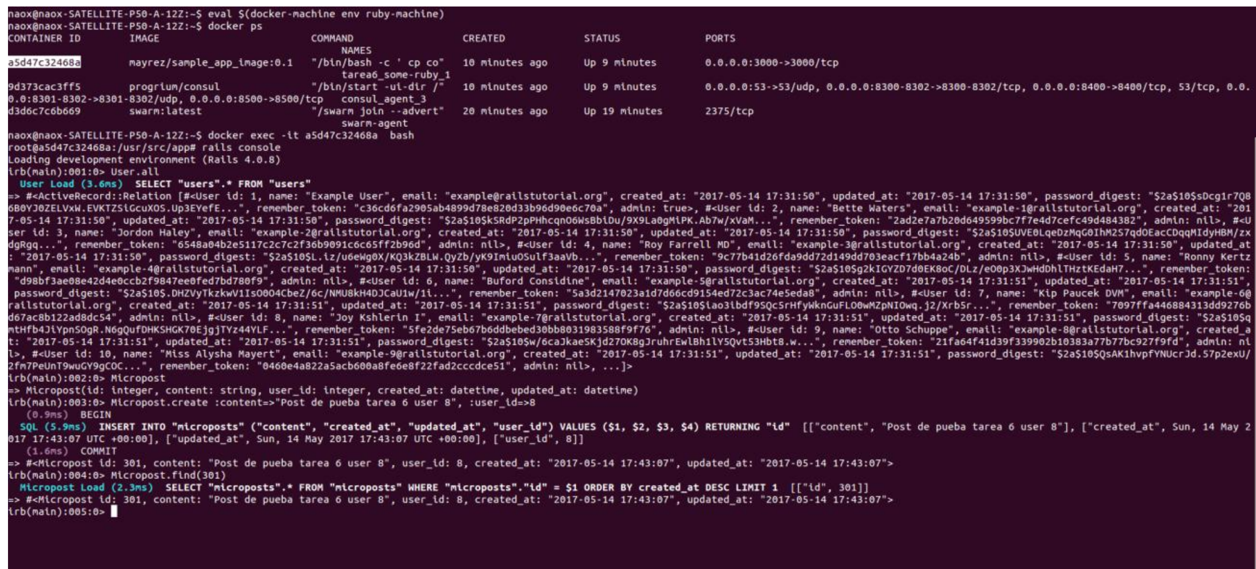

Figura 3.88 Prueba de la conexión de la aplicación con la base de datos. Se listan todos los usuarios y se crea un ‘micropost’ con el contenido ‘Post de prueba tarea 6 user 8’ y el id de usuario 8.

En la figura 3.90 se accede al usuario 8 y se comprueba que se ha añadido el nuevo post al perfil del usuario.

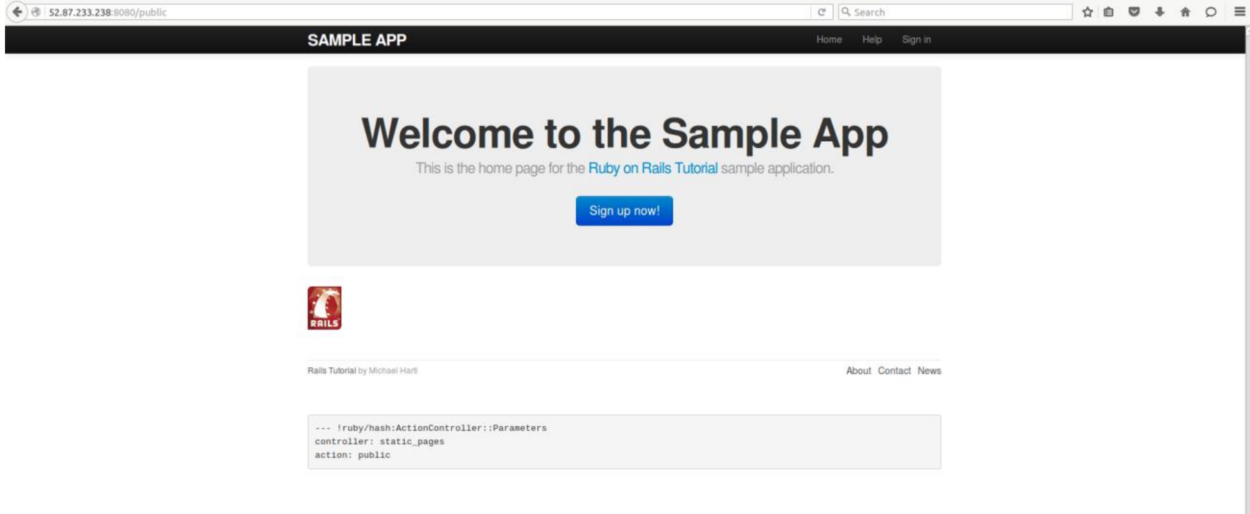

Figura 3.89 Acceso a la página principal de la aplicación a través de la dirección nodoMasterIP:8080

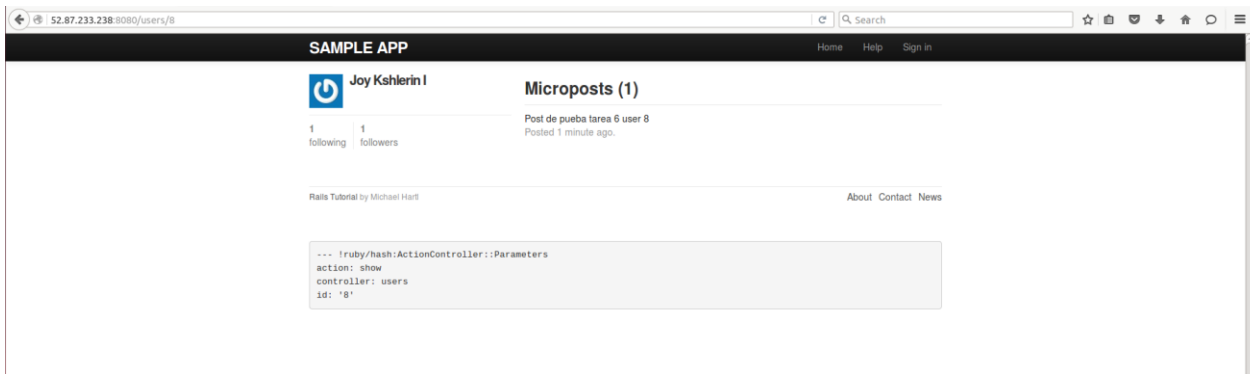

Figura 3.90 Acceso al post creado desde la consola del contenedor con Ruby.

Para comprobar el estado del cluster se establecerá el entorno del nodo máster (‘nginx-machine’) y se ejecutará ‘docker info’ como se muestra en las figuras 3.91 y 3.92. En la figura 3.92 se muestra la información de los tres nodos pertenecientes al cluster y el número de contenedores que se están ejecutando en el cluster.

```
# eval $(docker-machine env --swarm nginx-machine)

# docker info
```

Figura 3.91 Ejecución de ‘docker info’ en el nodo manager del cluster swarm.

```
naox@naox-SATELLITE-P50-A-12Z:~$ eval $(docker-machine env --swarm nginx-machine)
naox@naox-SATELLITE-P50-A-12Z:~$ docker info
Containers: 10
 Running: 10
 Paused: 0
 Stopped: 0
Images: 10
Server Version: swarm/1.2.6
Role: primary
Strategy: spread
Filters: health, port, containerslots, dependency, affinity, constraint, whitelist
Nodes: 3
 nginx-machine: 52.87.233.238:2376
  └ ID: ZQ6Q:UR5B:OM6B:MG3R:5XCC:HRVE:DENG:DDZ6:5OK4:MLZU:DITR:7M45
  └ Status: Healthy
  └ Containers: 4 (4 Running, 0 Paused, 0 Stopped)
  └ Reserved CPUs: 0 / 1
  └ Reserved Memory: 0 B / 1.016 GiB
  └ Labels: kernelversion=4.4.0-43-generic, operatingsystem=Ubuntu 16.04.1 LTS, provider=amazonec2, storagedriver=aufs
  └ UpdatedAt: 2017-05-14T17:45:52Z
  └ ServerVersion: 17.05.0-ce
 postgres-machine: 54.196.176.224:2376
  └ ID: ZBRZ:VAUW:N4QG:6RS2:EVC7:RLGA:O5PT:RTA4:H6JH:2SVP:K3AZ:TX75
  └ Status: Healthy
  └ Containers: 3 (3 Running, 0 Paused, 0 Stopped)
  └ Reserved CPUs: 0 / 1
  └ Reserved Memory: 0 B / 1.016 GiB
  └ Labels: kernelversion=4.4.0-43-generic, operatingsystem=Ubuntu 16.04.1 LTS, provider=amazonec2, storagedriver=aufs
  └ UpdatedAt: 2017-05-14T17:46:28Z
  └ ServerVersion: 17.05.0-ce
 ruby-machine: 54.87.240.185:2376
  └ ID: MFYN:X7VP:PLYL:434H:4246:FJGK:OQJ3:MSGD:PZIY:RA2C:L65Z:IVLV
  └ Status: Healthy
  └ Containers: 3 (3 Running, 0 Paused, 0 Stopped)
  └ Reserved CPUs: 0 / 1
  └ Reserved Memory: 0 B / 1.016 GiB
  └ Labels: kernelversion=4.4.0-43-generic, operatingsystem=Ubuntu 16.04.1 LTS, provider=amazonec2, storagedriver=aufs
  └ UpdatedAt: 2017-05-14T17:46:15Z
  └ ServerVersion: 17.05.0-ce
Plugins:
 Volume:
 Network:
Swarm:
 NodeID:
 Is Manager: false
 Node Address:
Security Options:
Kernel Version: 4.4.0-43-generic
Operating System: linux
Architecture: amd64
CPUs: 3
Total Memory: 3.048 GiB
Name: f2cde0c89195
```

Figura 3.92 Resultado ejecución de 'docker info'

Para ver todos los contenedores que se ejecutan en el cluster se ejecutará 'docker ps' desde el nodo manager.

```
# eval $(docker-machine env --swarm nginx-machine)
# docker ps
```

Figura 3.93 Ejecución de 'docker ps' en el nodo manager del cluster swarm.

Se pueden listar los nodos del clúster con el comando 'docker run swarm list' apuntando al servicio de Consul, como se muestra en la figura 3.94 y 3.95

```
# docker run swarm list consul://$(docker-machine ip consul-machine):8500
```

Figura 3.94 Ejecución de 'docker run swarm list' con Consul para listar los nodos del cluster swarm.

```
naox@naox-SATELLITE-P50-A-12Z:~$ docker run swarm list consul://$(docker-machine ip consul-machine):8500
time="2017-05-14T17:47:44Z" level=info msg="Initializing discovery without TLS"
52.87.233.238:2376
54.196.176.224:2376
54.87.240.185:2376
naox@naox-SATELLITE-P50-A-12Z:~$
```

Figura 3.95 Resultado de listar los nodos pertenecientes al Swarm con el comando 'docker run swarm list'

Para comprobar el funcionamiento de Consul se puede acceder a la interfaz del servicio por medio de la máquina con el servidor de Consul a través de su dirección ip y el puerto 8500. Esta página presenta una pestaña con los servicios, otra pestaña que muestra los nodos y otra que muestra las claves-valores almacenados.

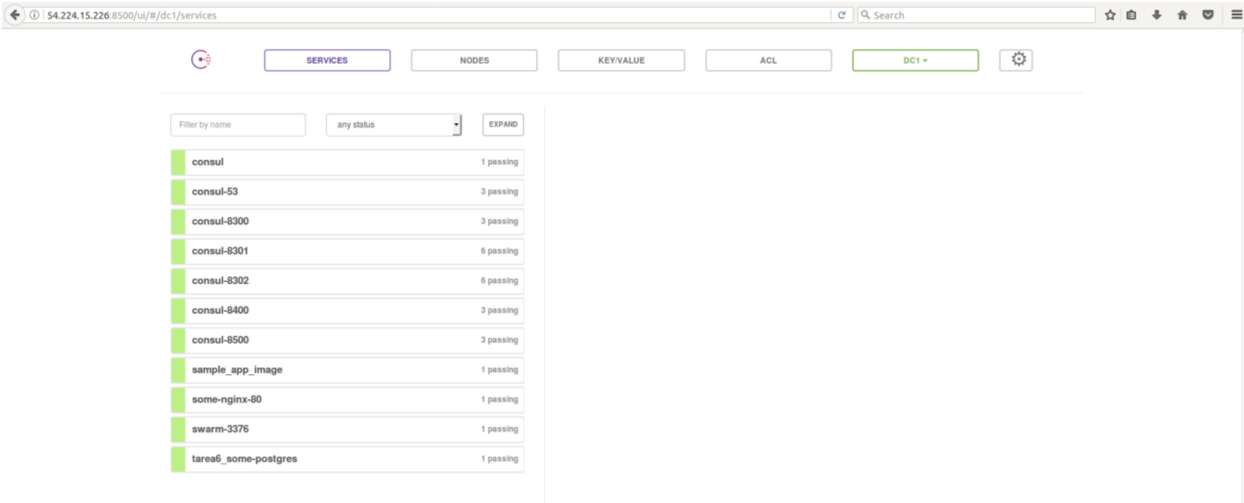

Figura 3.96 Acceso a la página de los servicios registrados en Consul.

En la pestaña 'services' (figura 3.96) se muestran los servicios pertenecientes a los nodos del cluster registrados a través del contenedor 'registrator' ejecutado en cada nodo. Si se accede a los servicios se mostrará la ip y el puerto desde la que se ofrecen los servicios (figura 3.97).

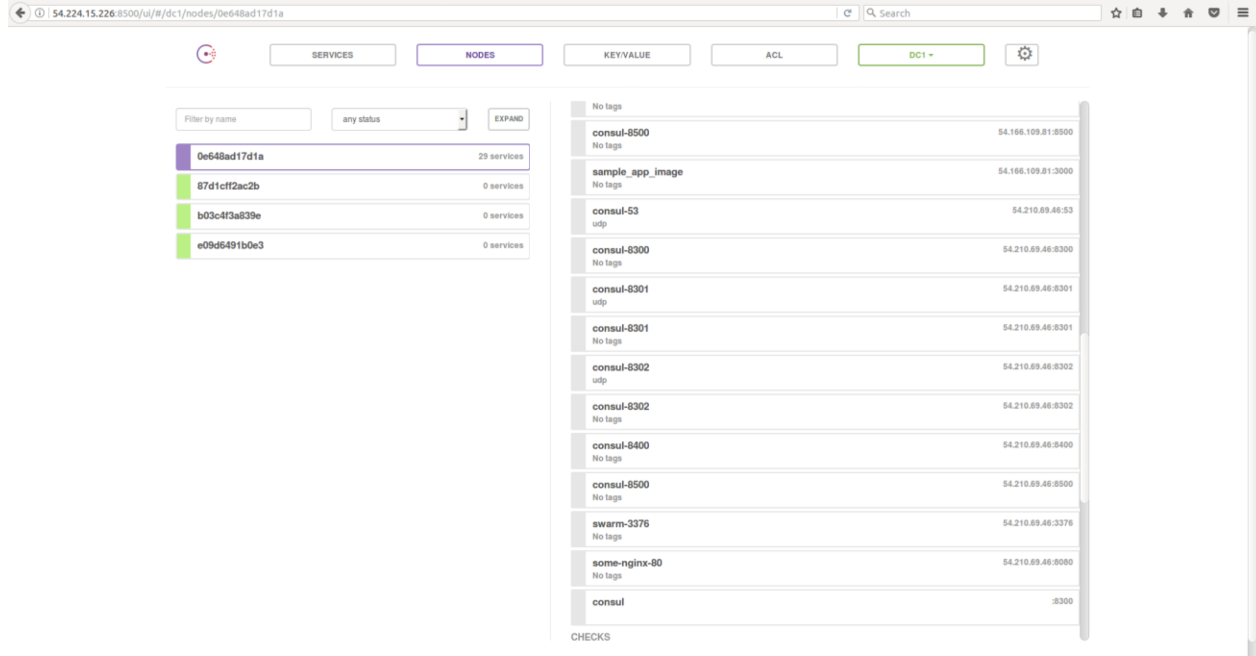

Figura 3.97 Nodo Consul con los servicios del cluster registrados y descripción de ip y puerto desde el que se ofrece el servicio.

En la pestaña de los key-value (figura 3.98) se muestran las secciones para 'network', 'nodes' y 'swarm', con los respectivos valores almacenados. Si se accede desde la pestaña 'Key/Value' a 'nodes se muestran las direcciones ip privadas de los nodos agentes de Consul (figura 3.99). Accediendo a 'network' y luego 'overlay' se verá la red overlay almacenada (figura 3.100). También en 'Key/Value' se incluye un registro con el valor almacenado para el cluster Swarm (Figura 3.101).

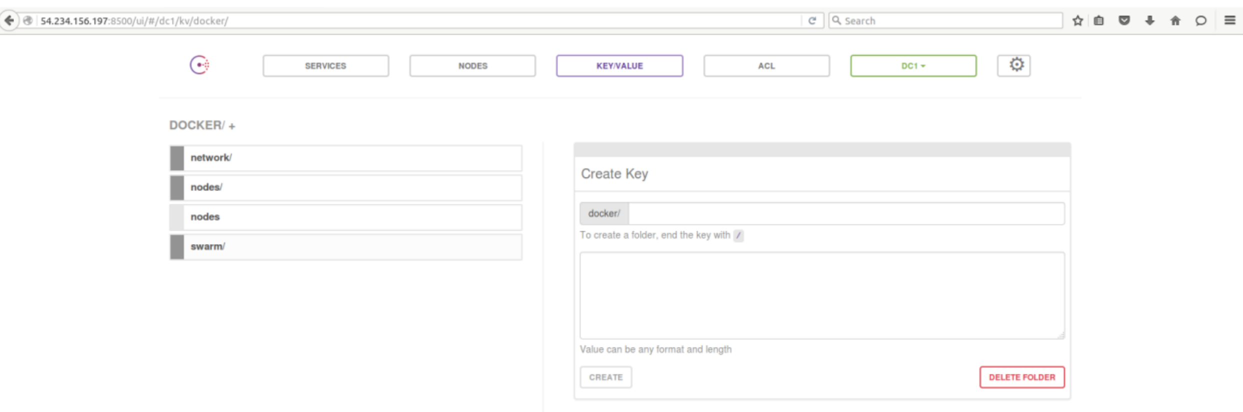

Figura 3.98 Valores almacenados en el key-value store. Ahora además de la red incluye el Swarm

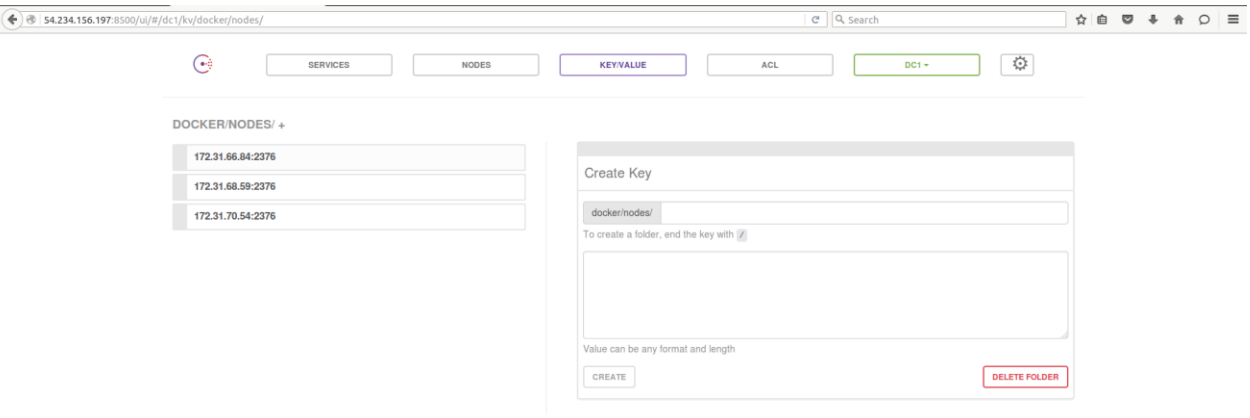

Figura 3.99 Ips privadas (network overlay) de las máquinas de Ruby, Nginx y Postgres almacenadas en la key-value store.

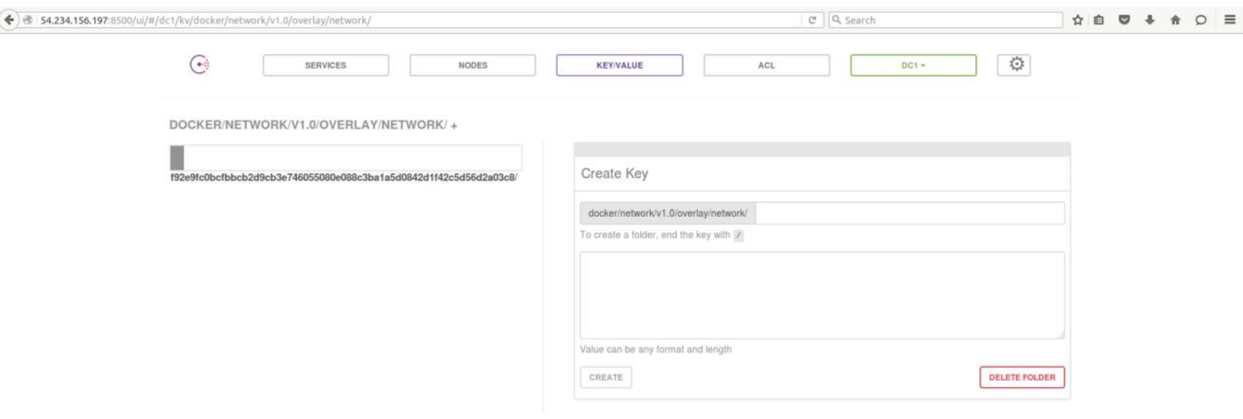

Figura 3.100 Network overlay almacenada en la key-value store.

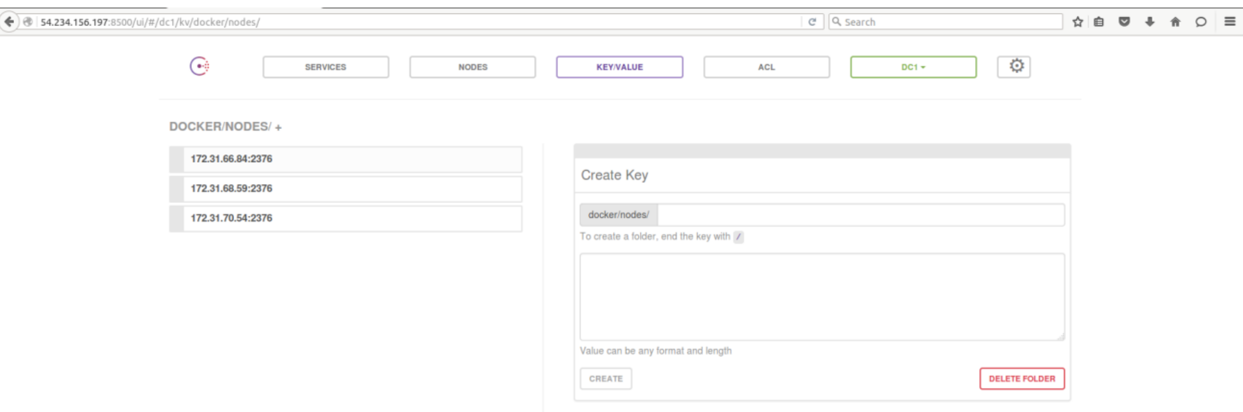

Figura 3.101 Nodos pertenecientes al Swarm almacenados en la key-value store.

4. Resultados y conclusiones

4.1 Resultados

A lo largo de este TFM se ha elaborado una guía en el https://bitbucket.org/tftdp/docker_project/ con los pasos a seguir desde la creación de las imágenes Docker, para la creación de servicios y la aplicación, hasta su despliegue, primero en local y en una única máquina y, en la última iteración, de forma remota en AWS y separando los servicios en tres máquinas virtuales creando un cluster Swarm.

Siguiendo los pasos establecidos de cada iteración definida en este TFM, se ha conseguido lanzar la aplicación en instancias de AWS por medio de un único script con la configuración necesaria para que la aplicación se ejecute de forma remota.

Como resultado, se ha conseguido que se ejecuten los tres servicios que definen la aplicación de tres capas (Base de datos, servidor web y aplicación) en tres máquinas distintas, consiguiendo que se comuniquen entre ellas de forma que se ofrezca el servicio como si se hubiese desplegado desde una única máquina.

El tipo de despliegue llevado a cabo en este TFM es sólo apto para un entorno de desarrollo y pruebas y no para entornos de producción, de ahí que se hayan usado las últimas versiones del ‘Swarm mode’ y Docker Compose y no se haya configurado la aplicación con la seguridad y monitorización requerida para entornos de producción.

4.2 Conclusiones

4.2.1 Consecución de objetivos

Iteración 1

En la primera iteración se ha dockerizado la aplicación, creando la imagen de la aplicación de Ruby y de todos los servicios necesarios para el despliegue de la aplicación de tres capas, siendo lanzadas por último desde una única máquina en local mediante comandos 'docker run'.

Iteración 2

En esta iteración se desplegó la aplicación, que en la iteración anterior había sido lanzada mediante comandos 'docker run', con todos sus servicios a partir de un único fichero de configuración Docker Compose, desde una única máquina virtual en local.

Iteración 3

En esta iteración se separó la configuración de los servicios en tres ficheros de Docker Compose para que cada servicio se arrancara en una máquina virtual diferente en local. Se crearon tres máquinas, una para la base de datos y el servicio de Consul, otra para el servidor Nginx y la última para la aplicación de Ruby. Debido a que los servicios se iban a lanzar desde diferentes máquinas, se necesitaba añadir un servicio para el descubrimiento de nombres de servicios con lo cual se añadió el servicio de Consul y la red overlay.

Iteración 4

En esta iteración se consiguió lanzar la aplicación de forma remota en tres máquinas diferentes desde AWS. Para ello, en lugar de usar como driver Virtualbox se usó el driver de 'amazonec2' para la creación de máquinas con el comando de Docker Machine.

Iteración 5

En esta iteración se integró Docker Swarm con la arquitectura de servicios diseñada anteriormente, desplegando la aplicación localmente. Para configurar el cluster Swarm se

añadieron los flag de Swarm necesarios para la creación de máquinas virtuales con Docker Machine.

Iteración 6

Partiendo de la configuración anterior, pero ahora desplegando la aplicación de forma remota en AWS, se lanzó la aplicación a partir de un script con los comandos necesarios para que la aplicación se ejecute de forma remota en la nube.

4.2.2 Trabajos futuros

- Se tendrían que asegurar las conexiones a los puertos, abriéndolos únicamente para las ip de las máquinas que acceden a los servicios. En los Security Groups se debería cambiar la regla de accesible desde cualquier lugar ("anywhere") a que fuera únicamente accesible desde las máquinas que acceden a dichos puertos para comunicarse entre nodos u ofrecer el servicio. Para ello, habría que asignar a las máquinas ips fijas. La máquina donde se ejecuta la base de datos debería tener ip privada, al igual que la máquina de Ruby ya que se conecta con el resto de nodos del cluster mediante la red privada.

- Para monitorizar el cluster y los servicios se ha de usar software de terceros, como Riemann (https://blog.docker.com/2016/03/realtime-cluster-monitoring-docker-swarm-riemann/), puesto que Docker Swarm no ofrece dicho servicio.

- Se puede crear el cluster sin la necesidad de utilizar Consul, utilizando el 'Swarm mode' de Docker en lugar de utilizar el Docker Swarm. Para ello, se tendrían que crear las máquinas virtuales de la forma normal con Docker Machine y luego inicializar el cluster con un *token* y realizar un *join* de los nodos al cluster. Con esta versión del modo Swarm se pueden utilizar los comandos de 'docker service' para monitorizar la salud del clúster y los procesos en ejecución.

4.2.3 Valoración personal

El uso de tecnologías y plataformas en la nube agiliza el proceso de desarrollo debido a que proporcionan los recursos necesarios para el despliegue de los servicios e infraestructuras, permitiendo que los desarrolladores sólo se tengan que preocupar de la implementación y el despliegue de la aplicación y no del mantenimiento de los servidores. Por todo ello, el despliegue en la nube permite tener las aplicaciones en ejecución de forma más rápida.

Las plataformas en la nube ofrecen servicios que garantizan la disponibilidad y seguridad de sus servidores, de manera que se puede tener la seguridad de que la aplicación se mantendrá accesible y en ejecución aunque algún servidor falle.

Por otra parte, se puede hacer un uso más eficiente de los recursos debido a que se pueden asignar manualmente a medida que se van necesitando, permitiendo programar de forma automática el escalado de la aplicación en función de la demanda o de las necesidades de la aplicación.

Se pueden automatizar otros procesos para que se facilite la monitorización de la aplicación y los servicios, como puede ser obtener métricas, la observación del rendimiento de la aplicación o alertar cuando se sobrepase el uso predefinido de recursos cuando la aplicación haya sido definida para que sea escalada de manera automática, permitiendo tomar medidas a tiempo.

Docker, por su parte, permite configurar y lanzar la aplicación rápidamente por medio únicamente de la ejecución comandos y definiendo los servicios por medio de ficheros. Otro de los beneficios de usar Docker es la portabilidad que ofrece entre diferentes máquinas, ya que la aplicación y los servicios se ejecutan como contenedores virtualizados que son independientes de la versión del kernel de Linux del host. Además, pueden ser transferidos a cualquier máquina que tenga Docker instalado.

5 Anexos

5.1 Repositorio de la aplicación

El repositorio para el despliegue de la aplicación se encuentra en el siguiente enlace https://bitbucket.org/tftdp/docker_project/.

5.2 Instalación de los componentes básicos para realizar el despliegue de la aplicación

5.2.1 Instalación de Virtualbox

```
#  sudo sh -c 'echo "deb http://download.virtualbox.org/virtualbox/debian trusty  \
contrib" >> /etc/apt/sources.list'

# wget http://download.virtualbox.org/virtualbox/debian/oracle_vbox.asc -O- |  \
sudo apt-key add –

# sudo apt-get update

# sudo apt-get install virtualbox-4.3

# sudo apt-get install virtualbox-dkms
```

Figura 5.1 Instalación de Virtualbox 4.3 en Ubuntu 14.04

5.2.2 Instalación de Docker

```
#  sudo apt-get  -y update

#  sudo apt-get install \

    linux-image-extra-$(uname -r) \

    linux-image-extra-virtual

#  sudo apt-get install \

    apt-transport-https \

    ca-certificates \

    curl \

    software-properties-common
```

```
# curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key add -
# sudo add-apt-repository \
   "deb [arch=amd64] https://download.docker.com/linux/ubuntu \
   $(lsb_release -cs) \
   stable"
#  sudo apt-get update
# sudo apt-get install docker-ce
```

Figura 5.2 Instalación de Docker 1.12.5 en Ubuntu 14.04

5.2.3 Instalación de Docker Machine

```
# curl -L https://github.com/docker/machine/releases/download/v0.9.0-rc2/docker-machine-`uname -s`-`uname -m` >/tmp/docker-machine
 # chmod +x /tmp/docker-machine
 # sudo cp /tmp/docker-machine /usr/local/bin/docker-machine
```

Figura 5.3 Instalación de Docker Machine versión 0.9.0-rc2 en Ubuntu 14.04

5.2.4 Instalación de Docker Compose

```
#  curl -L https://github.com/docker/compose/releases/download/1.9.0/docker-compose-`uname -s`-`uname -m` > /usr/local/bin/docker-compose
# sudo chmod +x /usr/local/bin/docker-compose
```

Figura 5.4 Instalación de Docker Compose versión 1.9.0 en Ubuntu 14.04

6. Referencias

[1] Documentación de Docker. https://docs.docker.com/

[2] Instalación de Virtualbox 4.3. http://www.infoworld.com/article/2696611/install-virtualbox-4-3-14-in-ubuntu-14-04.html

[3] Instalación de Docker. https://docs.docker.com/engine/installation/linux/ubuntu/#install-using-the-repository

[4] Instalación de Docker Machine. https://docs.docker.com/machine/install-machine/

[5] Instalación de Docker Compose. https://docs.docker.com/compose/install/

[6] Docker Swarm y Network Overlay, con Consul como servicio de almacenamiento clave-valor. https://docs.docker.com/engine/userguide/networking/get-started-overlay/

[7] Amazon Web Services. https://aws.amazon.com/

[8] Contenerización. http://searchservervirtualization.techtarget.com/definition/container-based-virtualization-operating-system-level-virtualization

[9] Contenerización VS Virtualización. https://www.linkedin.com/pulse/containerization-vs-virtualization-approach-better-shady

[10] Orquestación. http://searchitoperations.techtarget.com/feature/Why-orchestration-services-are-important-in-the-DevOps-age

[11] Alternativas a Docker. http://searchcloudapplications.techtarget.com/tip/Five-development-containers-to-consider-that-arent-Docker

[12] Kurbenetes. https://www.redhat.com/en/containers/what-is-kubernetes

[13] Mesos. http://mesos.apache.org/documentation/latest/architecture/

[14] Servicios AWS. https://d0.awsstatic.com/whitepapers/aws-overview.pdf

[15] Docker Hub. https://hub.docker.com/