Proyecto Fin de Carrera
Ingeniería de Telecomunicación

Despliegue de un servicio web en una plataforma cloud IaaS usando contenedores

Autor: Fernando Gómez Romero

Tutor: Isabel Román Martínez

Departamento de Ingeniería Telemática
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Sevilla, 2016

Proyecto Fin de Carrera
Ingeniería de Telecomunicación

Despliegue de un servicio web en una plataforma cloud IaaS usando contenedores

Autor:
Fernando Gómez Romero

Tutor:
Isabel Román Martínez

Departamento de Ingeniería Telemática
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla
Sevilla, 2016

Proyecto Fin de Carrera: Despliegue de un servicio web en una plataforma cloud IaaS usando contenedores

Autor: Fernando Gómez Romero

Tutor: Isabel Román Martínez

El tribunal nombrado para juzgar el Proyecto arriba indicado, compuesto por los siguientes miembros:

Presidente:

Vocales:

Secretario:

Acuerdan otorgarle la calificación de:

Sevilla, 2016

El Secretario del Tribunal

Agradecimientos

"¡Felicidades Fernando parece que esto va llegando a su fin!", "Venga ánimo que es el último esfuerzo", son algunos ejemplos de mensajes de ánimo con los que mi tutora, Isabel Román, siempre cerraba los correos que nos mandábamos acerca de este proyecto. Sin su apoyo y dedicación no habría sido posible terminarlo.

A mis padres y mis abuelos, sus continuos ánimos y la pregunta formulada frecuentemente "*¿Cómo llevas el proyecto?"* han sido el combustible para que la rueda no pare de girar hasta el final.

Marta ha estado ahí desde el primer minuto soportando estoicamente y con gran paciencia mis quejas y lloros durante el camino. Su respaldo diario ha sido imprescindible para este proyecto. Aun sin haber estudiado ingeniería, ella no te mirará raro si le hablas de un contenedor en la nube.

Gracias.

Fernando Gómez Romero

Sevilla, 2016

Resumen

La tecnología de los contenedores se encuentra en desarrollo desde las últimas décadas, sin embargo hasta la aparición de Docker en 2014 no ha alcanzado la madurez suficiente como para que se adapte en sistemas en producción. Por otra parte, los servicios en la nube que ofrecen grandes compañías como Amazon, Microsoft o Google están en pleno auge y han revolucionado el panorama aportando múltiples ventajas.

En el desarrollo de este proyecto se hará uso de contenedores y de instancias en la nube para desplegar una aplicación web, dejando atrás el modelo anterior de máquina virtual monolítica. Gracias a esto aparecerán nuevas opciones y funcionalidades que habrían sido imposibles en un despliegue tradicional.

Abstract

Container technology has been in development over the last decades, however up to Docker arrival in 2014 it has not reached enough maturity to be used in production systems. On the other hand, cloud-based services offered by large companies as Amazon, Microsoft or Google have changed the whole scene bringing multiple benefits.

During this project development, containers and cloud instances will be used to deploy a web application, leaving behind the previous model of a monolithic virtual machine. This approach will bring new features and options that would have been impossible using a traditional deployment.

Índice

ÍNDICE DE ILUSTRACIONES

ÍNDICE DE CÓDIGOS

1 INTRODUCCIÓN

1.1 Motivación y objetivo

La motivación del proyecto surge del interés en explorar una innovadora tecnología de reciente auge como son los contenedores y el despliegue de aplicaciones haciendo uso de servicios de computación en la nube. Cuando estas dos tecnologías se usan en conjunto utilizando herramientas de orquestación de contenedores ofrecen gran potencial.

En el desarrollo del proyecto se desplegará una aplicación de seguimiento de pacientes renales crónicos llamada E-nefro. Actualmente esta aplicación se utiliza en una máquina virtual tradicional de Linux que contiene un servidor de aplicaciones y una base de datos. El objetivo será desplegar E-nefro usando contenedores y máquinas virtuales en la nube, además de añadir características extras a la infraestructura de la aplicación tales como:

- Automatización del despliegue.
- Uso de un proxy que redirigirá las peticiones a la aplicación, con la posibilidad de que actúe de balanceador entre varias instancias del servidor de aplicación.
- Elasticidad en la nube, tanto horizontal como vertical.
- Transparencia de fallo.

Estas características se desarrollarán en tres escenarios, cada uno construido sobre el anterior.

El primero implicará un despliegue sin el uso de servicios en la nube sirviendo a modo de introducción del funcionamiento de los contenedores con la tecnología Docker. Se utilizará una única máquina donde conviven un cliente y un host de Docker. Desde el cliente se mandarán comandos al host para la construcción y ejecución de contenedores.

Inicialmente la aplicación se separará en dos contenedores, uno para la base de datos y otro para el servidor web. Cuando esté en funcionamiento, se añadirá un nuevo contenedor a modo de proxy que también balanceará la carga entre varias réplicas que se crearán del contenedor del servidor web. Al final se introducirá una herramienta de administración multi-contenedor que facilitará la administración de todos estos contenedores.

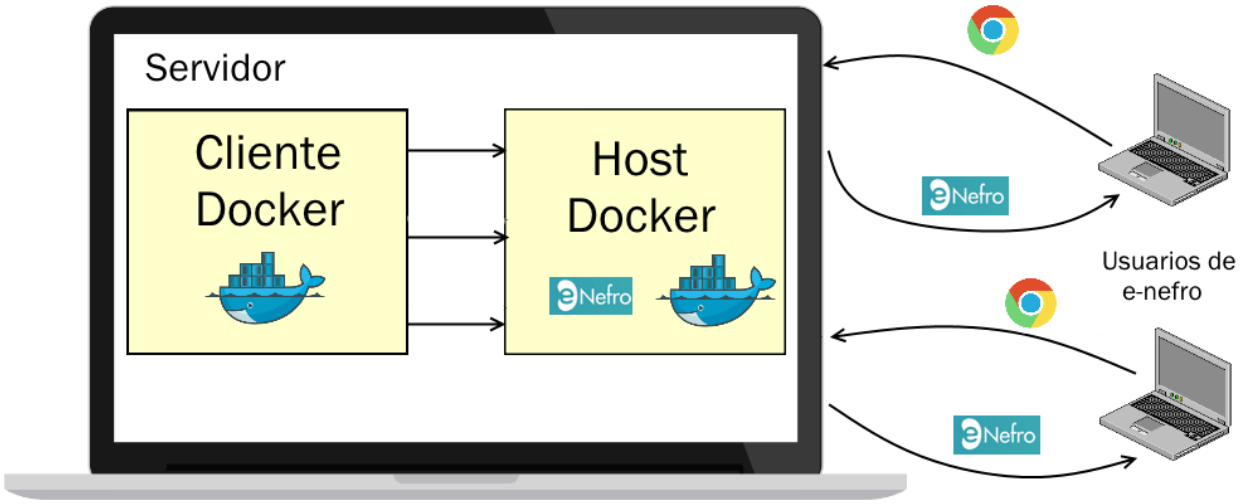


Ilustración 1 - Esquema del primer escenario

El segundo escenario traslada el entorno de contenedores anterior a la infraestructura como servicio en la nube (IaaS) que ofrece Microsoft Azure, funcionando sobre una única instancia. Esta vez el cliente de Docker estará sobre una máquina local externa desde la cual se mandarán los comandos de Docker necesarios al host en la nube.

Una vez que los contenedores estén en ejecución dentro de la máquina virtual se utilizarán las ventajas que ofrece la nube, tales como corta fuegos, elasticidad de los recursos y la fácil replicación del entorno por medio de clonación de imágenes.

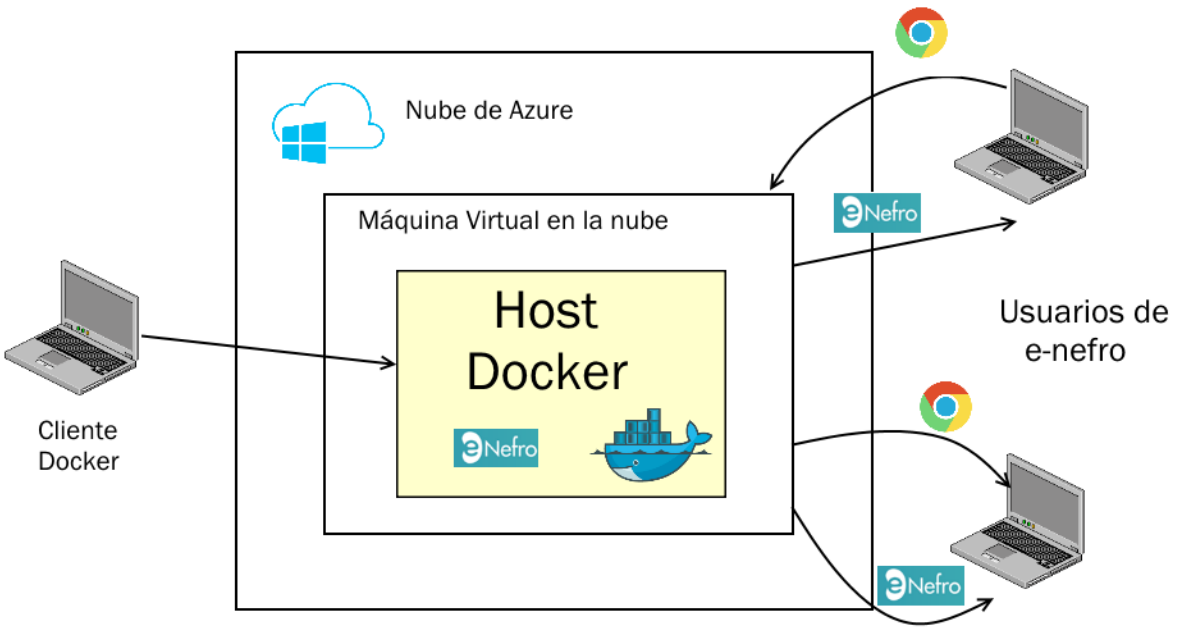


Ilustración 2 - Esquema del segundo escenario

El último escenario continuará en la nube pero utilizando un sistema distribuido de varias máquinas virtuales (llamadas nodos). Como el objetivo será que los contenedores puedan funcionar en los nodos sin importar donde se está ejecutando cada uno, en esta ocasión se utilizará una herramienta de orquestación de contenedores llamada Kubernetes. Esta herramienta aportará muchas ventajas como la transparencia de fallo, balanceo de carga entre los nodos o una mejor administración de los contenedores. Las peticiones

web y los comandos de ejecución de contenedores se gestionarán a través de un nodo especial llamado master.

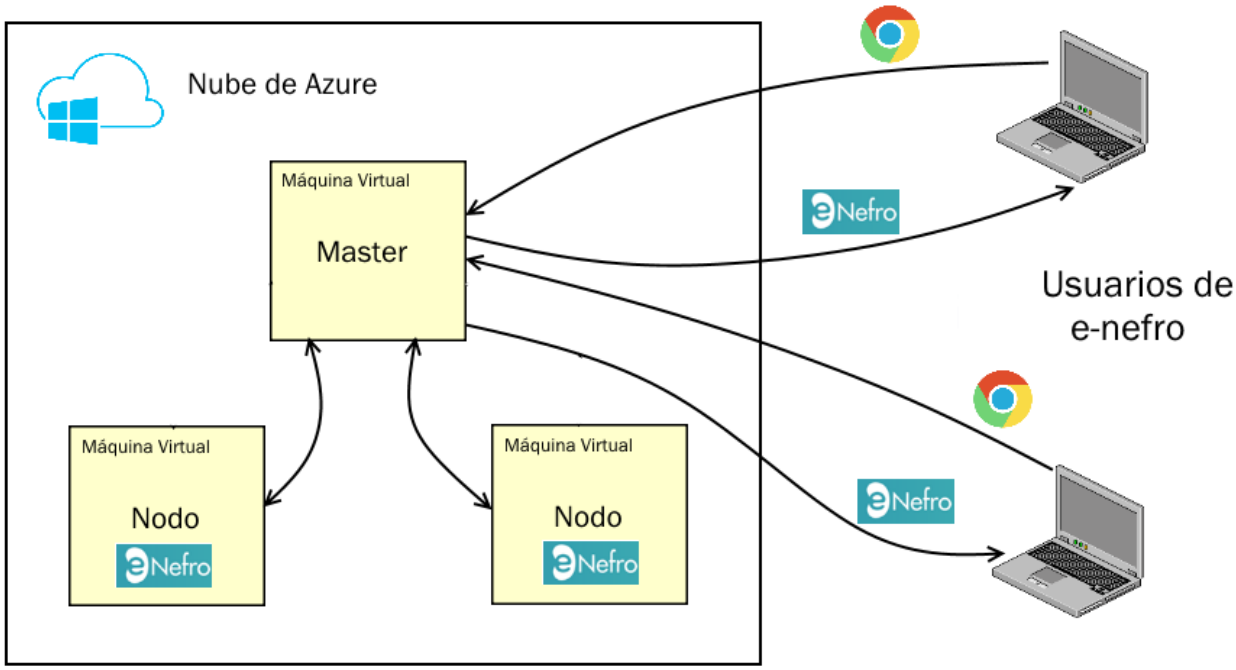


Ilustración 3 - Esquema de la tercera solución

Con estas soluciones se mejorará el control de la infraestructura de la aplicación de una manera que sería compleja siguiendo el entorno tradicional de una única máquina virtual alojada en un servidor local.

Este trabajo ayudará a analizar cómo se puede migrar del modelo tradicional a la aproximación en la nube basada en contenedores, además de las ventajas de esta migración.

1.2 Organización del documento

Después de esta breve introducción, en la siguiente sección se recopila información acerca de las tecnologías sobre las que trata el proyecto, comenzando con situarlas en el contexto temporal y el porqué de su aparición. Tras un análisis de las ventajas e inconvenientes que aportan al panorama tecnológico se enumeran las distintas implementaciones que han ido surgiendo, finalizando con una comparativa entre ellas.

Después de esta sección dedicada al estado de la técnica de forma general vendrá una sección de las herramientas escogidas en particular para el desarrollo de los objetivos planteados. En ella se presentará la aplicación principal E-nefro, su funcionamiento y estructura. Se mencionarán también qué implementaciones de contenedores y qué infraestructura en la nube se utilizarán en el transcurso del proyecto exponiendo sus características y conceptos principales.

La siguiente sección será el núcleo del proyecto donde se desarrollarán las soluciones al despliegue de E-nefro mediante contenedores y servicios en la nube, detallando cada paso necesario para hacer funcionar la aplicación.

Por último el proyecto finalizará con unas conclusiones donde se expondrán líneas de mejora y ampliación de las soluciones propuestas y se reflexionará del impacto que ha supuesto la llegada de estas nuevas tecnologías.

2 ESTADO DE LA TÉCNICA

2.1 Contenedores

Un contenedor es una forma de virtualización de un sistema operativo que almacena un sistema de ficheros con justo lo necesario para ejecutar un único servicio, código o herramienta. Inicialmente se planteó sobre sistemas operativos Unix, añadiéndose durante 2016 el desarrollo de soluciones en Windows (1). Aunque los contenedores comparten el *kernel* de la máquina en la que se está ejecutando, los procesos que corren dentro de ellos actúan de forma aislada al resto del sistema, reservando en exclusividad recursos como CPU o memoria (2).

Siguiendo la metodología de despliegue con microservicios, los contenedores ofrecen grandes posibilidades al poder tener diferentes funcionalidades aisladas entre sí sin afectar a las demás, siendo accesibles por medio de interfaces o APIs. Gracias a esto es más sencillo aumentar o reducir los recursos destinados a un único servicio de manera independiente a los demás. Un contenedor se puede guardar en una imagen, por lo que la replicación siempre va a ser idéntica, incluso van a ser portables entre diferentes distribuciones de Linux.

Este concepto puede parecer similar al de máquina virtual, no obstante existen muchas diferencias claves. La principal es que los contenedores van encima del sistema operativo compartiendo directamente el *kernel* mientras que una máquina virtual necesita un hipervisor como intermediario que monta por encima un sistema operativo completo.

Los contenedores están diseñados para ser más ligeros y necesitar menor tiempo de despliegue al contrario que las máquinas virtuales, que aportan el sistema operativo completo. Un contenedor solo necesita las librerías y binarios que vaya a utilizar el servicio que ejecuta.

A nivel de rendimiento, al no depender de un hipervisor los contenedores van a consumir muchos menos recursos en la máquina principal. Esta diferencia es importante si el escenario va a necesitar desplegar miles de servicios en una misma máquina, algo muy sencillo de conseguir por medio de contenedores pero prácticamente imposible con máquinas virtuales.

Al compartir el *kernel* las aplicaciones que van sobre contenedores se interconectarán más fácilmente que si corrieran en diferentes máquinas virtuales.

En el siguiente esquema se muestran las diferencias entre la arquitectura de máquinas virtuales y contenedores:

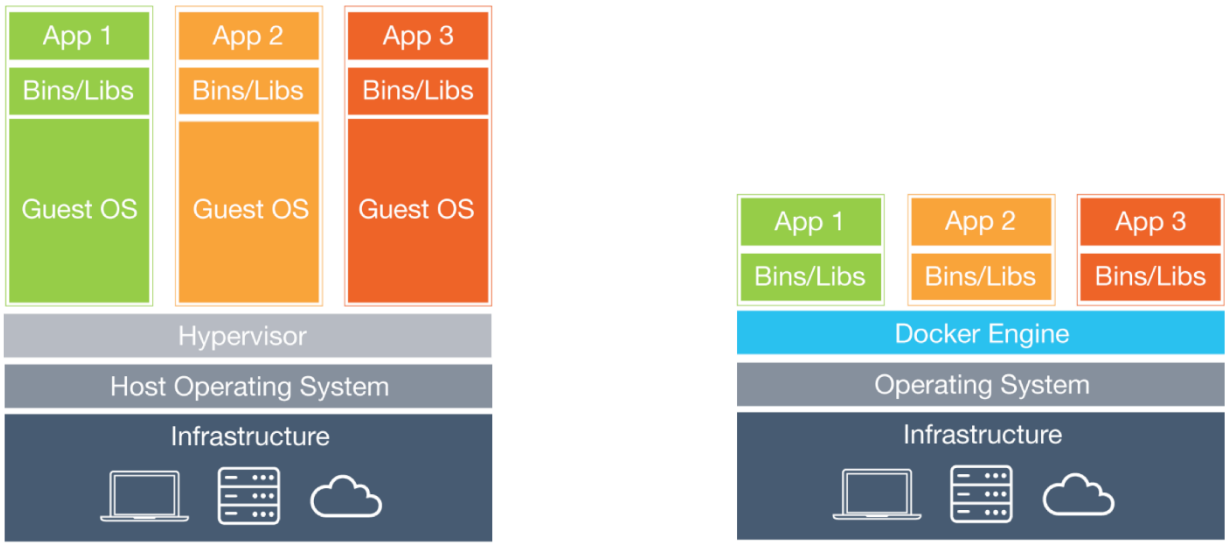


Ilustración 4 - Funcionamiento de máquinas virtuales (izquierda) y contenedores (derecha)

Como desventajas, los contenedores están limitados a utilizar el mismo sistema operativo ya que comparten el *kernel*. Sobre máquinas virtuales se pueden tener varias aplicaciones en diferentes sistemas operativos y el hipervisor se encargará de establecer la comunicación con el núcleo del sistema de la máquina principal.

El hecho de que las máquinas virtuales aporten todo el sistema operativo hace que el aislamiento y la seguridad sean mayores. Aunque los contenedores aíslan sus recursos por medio de técnicas que se presentarán posteriormente, acceden directamente al núcleo del sistema favoreciendo la existencia de vulnerabilidades.

Los contenedores están enfocados a la ejecución de una determinada tarea prevista con anterioridad, normalmente el funcionamiento de un servicio o un trabajo como puede ser un servidor web o testear una aplicación.

Sumando esta última característica a las comentadas anteriormente, como su ligereza y su menor tiempo de arranque, hacen que el ciclo de vida de los contenedores sea efímero, permitiendo que puedan ser sustituidos rápidamente por otro igual sin verse alterado el funcionamiento del sistema.

Al tener una vida efímera se plantea un problema con la persistencia de información, necesitando almacenar los datos en algún lugar externo al propio contenedor para que cuando un contenedor necesite ser sustituido la información no se vea afectada.

Tecnología	Ventajas	Inconvenientes
Contenedores	- Muy ligeros al compartir el sistema operativo - Consumen pocos recursos, por lo que podemos lanzar muchos fácilmente - Despliegue en segundos	- Menos aislamiento y seguridad debido a que comparten kernel. - No pueden funcionar en varios sistemas operativos - Persistencia de datos, los contenedores tienen un corto periodo de vida
Máquinas Virtuales	- La virtualización del sistema operativo es completa - Buen aislamiento del resto del sistema	- Consume muchos recursos del sistema y es menos escalable - Necesita un hipervisor como intermediario

Tabla 1 - Comparativa entre contenedores y máquinas virtuales

Aunque en los últimos años ha habido un resurgimiento de las tecnologías basadas en contenedores no se trata de un concepto nuevo en la historia de la computación.

La primera funcionalidad que se acercó a lo que conocemos como contenedores fue el comando *chroot* introducido durante 1979 en la versión 7 de Unix. *chroot* es un comando de Unix que toma un directorio indicado como el raíz, por lo que un programa que se ejecute dentro no podrá modificar ningún fichero en algún directorio externo (3).

Sin embargo no fue hasta los 2000 cuando se dieron las primeras implementaciones de contenedores, como los *jails* de FreeBDS o *VServer* incluido en el kernel de Linux en 2001 (4) (5).

En Google han utilizado contenedores para desarrollos internos desde el año 2004, lo que implicó que aportaran grandes avances al panorama. En 2007 liberaron para el proyecto de Linux los grupos de control o *cgroups* (6), lo que sentó las bases de los entornos con contenedores. Un *cgroup* permite agrupar una serie de procesos juntos y asegura que cada uno de estos grupos va a tener su propia memoria, CPU y disco, evitando que un grupo monopolice estos recursos. Por otra parte los *namespaces* (7) fueron otra pieza clave desarrollada por ellos. Estos permiten que un proceso tenga sus propios usuarios del sistema y que un proceso se ejecute con permisos de superusuario exclusivamente dentro del contenedor.

Estas innovaciones se fueron agregando al núcleo de Linux y llevaron a la creación de los *LXC*, contenedores de Linux, por parte de IBM en 2008 (8). Estos proporcionaban herramientas que hacían uso de los *cgroups* y *namespaces* para la administración de contenedores, su red de conexiones y almacenamiento. En sus inicios su uso implicaba muchos problemas de seguridad que poco a poco se fueron solucionando hasta que se publicó en 2014 la versión 1.0, incluyendo soporte para *SELinux* y *Seccomp* (9) (10).

Tras haber experimentado durante muchos años con los contenedores, Google publicó en 2013 como código abierto *Let me contain that for you (LMCTFY)* (11), una herramienta de administración de contenedores que funciona al mismo nivel que LXC. El hecho de que Google llevara desde 2004 utilizando esta tecnología de manera interna hizo que fuera una herramienta sólida.

Sin embargo la importancia que han adquirido actualmente los contenedores se debe en gran parte al surgimiento de Docker. Solomon Hykes fundó Docker en 2013 como un proyecto de código libre que agrupa todos los desarrollos anteriores y añade mejoras de uso con el fin de que cualquier desarrollador pudiera empaquetar sus aplicaciones sobre contenedores de una forma sencilla. Además del uso de *cgroups* y *namespaces*, su gran aportación fue la facilidad de capturar y construir imágenes de contenedores, lo que posteriormente llevó a la creación del *Docker Hub*, un gran repositorio público de imágenes donde los usuarios pueden descargar cualquier imagen que un desarrollador o compañía haya subido.

En la versión 0.9 de Docker dejaron de utilizar principalmente LXC para usar *libcontainer*, una librería que permite acceso directo a la API del *kernel* de Linux (12). De esta forma Docker cuenta con varias interfaces de comunicación con el *kernel* para lanzar y administrar los recursos de los contenedores.

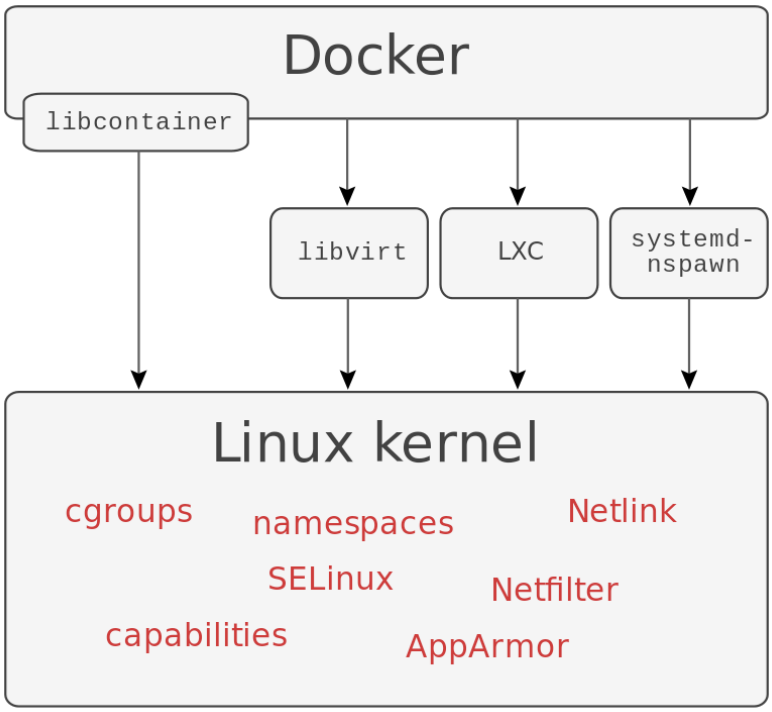


Ilustración 5 - Arquitectura de Docker

Gracias a una fuerte campaña de marketing Docker trasladó el espíritu de los contenedores que llevaba gestándose desde inicios de los 2000 al gran público y ha conseguido ganar la batalla como referente de uso de contenedores. Los desarrolladores de la tecnología *LMCTFY* de Google anunciaron durante 2015 que iban a parar de mantenerlo y se centrarían en complementar Docker y *libcontainer* (13).

Como respuesta a los contenedores de Docker, CoreOS lanzó en 2014 Rocket (14) con el objetivo de mejorar algunos de los defectos que tiene Docker. CoreOS es una compañía que surgió para la creación de una distribución de Linux muy ligera, segura y escalable, por lo que es ideal para utilizarse como base de contenedores.

Uno de los grandes defectos que se le ha achacado a Docker es la carencia de una buena seguridad lo que ha llevado que pocas compañías se atrevan a utilizarlo en producción. Rocket pretende subsanar esto añadiendo un sistema de seguridad más riguroso y contando con un estándar más abierto que el de Docker (15).

App Container (appc) (16) es una declaración abierta que define diferentes aspectos de cómo ejecutar una aplicación sobre contenedores. Rocket está creado como una implementación de *appc* lo que favorece que otras herramientas diferentes puedan utilizar las imágenes de contenedores de Rocket sin usar su tecnología.

En la siguiente tabla se muestran las ventajas e inconvenientes que aportan las tecnologías de contenedores mencionadas anteriormente:

Tecnología	Año	Ventajas	Inconvenientes
Docker	2013	- Se ha convertido en el estándar, todas las plataformas lo implementan y lo apoyan - Es muy sencillo de utilizar y cuenta con otros servicios como el Hub, un repositorio de imágenes.	- No es fiable en sistemas en producción
Rocket	2014	- Arregla los fallos de seguridad de Docker - Tiene como objetivo ser más abierto y aceptar contribuciones de la comunidad	- Ha llegado más tarde, lo que implica que no tenga mucho apoyo - La curva de aprendizaje es más pronunciada
LXC	2008	- Es la base de las demás tecnologías	- Se ha quedado anticuado en favor de otras tecnologías construidas por encima.
lmctfy	2013	- Es uno de los pioneros, cuenta con el apoyo de Google, compañía que ha contribuido en el desarrollo de la arquitectura de los contenedores.	- Se dejó de mantener durante 2015 en favor de Docker

Tabla 2 - Comparativa entre distintas tecnologías de contenedores

Orquestación de contenedores

Los contenedores no son suficientes cuando se quiere desplegar una aplicación en un entorno donde se van a necesitar muchas instancias o porque la disponibilidad de la aplicación deba ser alta. Para ello serán necesarias herramientas que, desde un nivel de abstracción superior, permitan administrar aplicaciones multi-contenedor, distribuidas en diferentes máquinas.

Docker Compose

La herramienta más sencilla de orquestación de contenedores es Docker Compose, que surgió tras la adquisición en julio de 2014 por parte de Docker de una compañía llamada Fig (17).

Compose sirve para administrar aplicaciones que hagan uso de varios contenedores, pero siempre en una única máquina. Haciendo uso de un fichero de configuración donde se definen los servicios de la aplicación, Compose crea e instancia todos los contenedores de la aplicación ejecutando tan solo un comando en vez de tener que construir y lanzar cada contenedor de forma individual.

Una vez que la aplicación esté en ejecución, se puede monitorizar el estado de los servicios, ver la salida de los registros de los contenedores en ejecución, ejecutar comandos dentro de ellos o reconstruir y relanzar cualquiera de ellos.

Compose no permite tener un cluster de contenedores funcionando sobre muchas máquinas repartidas en distintas localizaciones ni balancear la carga entre varias instancias del mismo contenedor, por lo que se necesitarían otras herramientas más avanzadas.

Docker Swarm

Swarm es una herramienta mantenida por Docker con la cual se puede crear un cluster de máquinas que actúan como hosts de Docker y que el usuario administre como un único host de manera virtual. Swarm va

a actuar como intermediario para hacer más sencillo el despliegue de una aplicación con contenedores sobre varias máquinas, como podemos ver en el siguiente esquema.

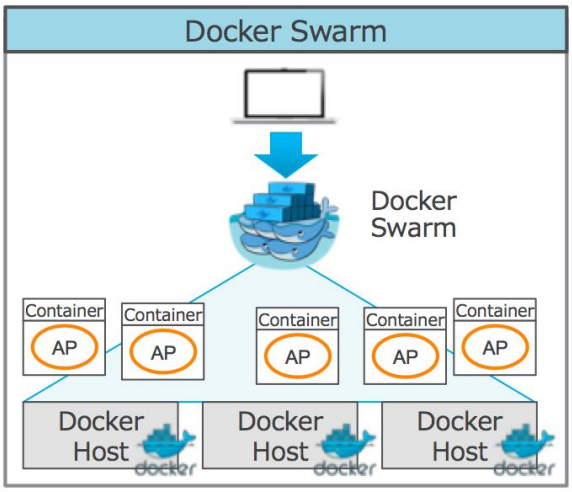


Ilustración 6 - Docker Swarm

La gran ventaja que aporta es que utiliza la API de Docker, por lo que cualquier herramienta que se comunique con el demonio de Docker puede utilizar Swarm de forma transparente para desplegar la aplicación sobre múltiples máquinas.

En vez de tener que decidir en qué máquina se quiere cada contenedor, es posible indicarle a Swarm que se quiere iniciar un determinado contenedor y este decidirá de forma autónoma en segundo plano en que máquina comenzarlo. Para decidir en qué host lo va a lanzar se sirve de varios mecanismos, como el comprobar el estado de CPU y memoria de cada máquina y lanzarlo en el que tenga más recursos disponibles.

Tanto Swarm como otras alternativas de orquestación de contenedores necesitan alguna herramienta de descubrimiento de servicios. Estas sirven para administrar como los procesos y servicios de un cluster se encuentran y comunican entre ellos. Necesita por tanto de un registro de servicios que se pueda explorar de forma rápida. En este registro se almacena el nombre del servicio y la información necesaria para conectarse a él a través de un servicio de nombrado (18).

Swarm está diseñado para ser compatible con varias de estas herramientas de descubrimiento de servicios, siendo las tres más comunes z*ookeeper*, *etcd* y *cónsul* (19) (20).

Kubernetes

Google anunció en junio de 2014 su propia herramienta de código abierto de orquestación de contenedores llamada Kubernetes. Está fuertemente influenciado por *Borg* (21), el software interno que Google utilizó durante más de 10 años sin liberarlo y que servía como orquestador de recursos en sus *datacenters* y para el despliegue y administración de sus aplicaciones.

Su objetivo es proveer una plataforma de despliegue, escalado y operaciones automáticas de aplicaciones sobre contenedores a lo largo de un cluster de hosts (22). Kubernetes está centrado en el despliegue de aplicaciones, por lo que sus funcionalidades principales pasan por el escalado y actualización de los componentes sin necesidad de parar los servicios o la optimización del hardware en función de los recursos que necesite la aplicación.

Es muy portable ya que está diseñado para que funcione sobre todo tipo de soluciones, desde locales hasta en la nube o sobre máquinas virtuales (23).

Kubernetes no es un mero sistema de orquestación ya que aporta independencia en la ejecución de sus procesos. Busca continuamente que el estado actual de los contenedores sea el estado deseado que el usuario ha especificado previamente. Por ejemplo, si se le indica que un determinado servicio se mantenga en ejecución y debido a un error tiene que detenerse, Kubernetes se encargará de relanzar una nueva instancia con las mismas condiciones iniciales de manera automática. Para la transición entre instancias se puede hacer uso de algún mecanismo de persistencia de datos como los volúmenes. Gracias a estas características el sistema final es más robusto e infalible.

Mesosphere+Marathon

Apache Mesos es un administrador de clusters de código abierto que simplifica el despliegue de aplicaciones sobre múltiples máquinas. Mesosphere ofrece más funcionalidades por encima de Mesos como modificar el número de instancias, sustitución automática de contenedores ante fallos o descubrimiento de servicios con *zookeeper*.

Gracias a Marathon, un *framework* de Mesos, podemos utilizar la infraestructura de Mesosphere con contenedores con posibilidad de escalado a más de 1000 nodos y orientado a ofrecer alta disponibilidad en sus nodos.

Requiere un aprendizaje más profundo para desplegar una aplicación sobre un cluster debido a su cantidad de componentes y posibilidades.

Tecnología	**Ventajas**	**Inconvenientes**
Swarm	- Al estar desarrollado por Docker tiene una gran integración con contenedores y Compose - Muy simple de utilizar	- Código muy inmaduro, le faltan muchas funcionalidades por implementar - Difícil de utilizar en entornos más complejos
Kubernetes	- Integración con diferentes plataformas y proveedores de cloud - Está diseñado para trabajar de forma autónoma ofreciendo mucha robustez	- Funcionalidad limitada con más de 100 nodos - Está optimizado para la cloud de Google
Mesosphere	- Permite escalar a más de 1000 nodos	- Es complejo de configurar y mantener

Tabla 3 - Comparativa entre tecnologías de orquestación de contenedores

2.2 Computación en la nube

La computación en la nube pública traslada la tradicional infraestructura de máquinas físicas a un *datacenter* ajeno donde un agente externo se encargará del mantenimiento de recursos tales como servidores, almacenamiento o aplicaciones. Estos proveedores tienen redes con muchísimos servidores que consumen muy poco y que maximizan su capacidad y rendimiento mediante técnicas de virtualización, lo que conlleva que ofrezcan servicios muy competitivos económicamente.

La nube también podrá ser privada dependiendo de quién aloje la infraestructura. La privada está destinada a compañías que ya tienen su propio *datacenter* donde pueden establecer su sistema de nube con sus propias

medidas de seguridad, sin embargo la administración y mantenimiento es responsabilidad de la compañía. Durante el resto de este proyecto nos referiremos exclusivamente a la nube pública.

La llegada de la computación en la nube y la bajada de precios de los servicios basados en infraestructura como los de Amazon Web Services o Azure ha cambiado por completo el panorama. Como Nicholas Carr apunta en su publicación *The Big Switch* (24), el presente se enfrenta a un cambio similar al ocurrido con la aparición de compañías proveedoras de electricidad. Hasta entonces los fabricantes se encargaban de su propia electricidad por medio de molinos o ruedas hidráulicas. Estos fueron viendo cómo se abarataban considerablemente los gastos confiando en un proveedor de electricidad. De manera similar en la actualidad se aprecia una transición de infraestructuras internas y privadas a la subcontratación de proveedores que se encarguen de todo.

La aparición de proveedores públicos ha revolucionado el mercado ya que con la computación en la nube cualquier compañía puede utilizar recursos y almacenamiento compartido antes que construir, operar y mantener su propia infraestructura. Pequeñas compañías como *startups* ahora pueden desarrollar su modelo de negocio sin necesidad de una gran inversión económica en infraestructura de comunicaciones, algo imposible con el modelo tradicional.

Una de las características clave que aporta confiar una infraestructura en la nube es la elasticidad. En función de la demanda que requiera un determinado servicio se ofrece la posibilidad de crecer o reducir la capacidad de los recursos destinados de manera instantánea. Algunos proveedores ofertan el autoescalado, donde el sistema se adapta en tiempo real según el tráfico. Finalmente solo se pagará la capacidad utilizada y evitará la necesidad de crear sistemas que se encarguen de la sobrecarga.

El concepto de computación en la nube parece moderno pero es una visión que se lleva desarrollando desde los inicios de internet con ARPANET. Sin embargo hasta finales de los años 90 no llegaron los avances tecnológicos y ancho de banda suficiente para desarrollar la infraestructura necesaria.

El grupo de estudio 13 del ITU-T (25) trabaja en redes de nueva generación y es el principal grupo que se encarga de la investigación de la computación en la nube. En su recomendación Y. 3500 (26) se define de forma general qué es la nube y los conceptos que la rodean. Este grupo también ha publicado otras recomendaciones en la serie de Y.3500-3999 donde se investiga sobre temas como la arquitectura, seguridad o modelos de servicio (27).

Los proveedores de nube han popularizado la forma de ofrecer los servicios siguiendo varios modelos :

- Software como Servicio (SaaS): el proveedor ofrece una aplicación ya desplegada y configurada en sus servidores. El alojamiento y las tecnologías que dan soporte a la aplicación son transparentes para el cliente. Un ejemplo es un cliente de correo electrónico, donde es transparente todo lo que no tenga que ver con la aplicación en sí.
- Plataforma como Servicio (PaaS): consiste en un entorno preparado para que los desarrolladores puedan elaborar sus aplicaciones y ofrecerlas al cliente final. Por tanto consiste en un conjunto de servicios accesibles tales como sistema operativo, entorno de desarrollo del lenguaje de programación que elija el cliente, base de datos o servidor web. La plataforma proporciona características como replicación, balanceo de carga, fiabilidad o persistencia a las soluciones desarrolladas.
- Infraestructura como Servicio (IaaS): ofrece máquinas virtuales y almacenamiento, liberando al cliente de la infraestructura física. Contratar una infraestructura en la nube permite que se adapten fácilmente las necesidades del usuario en cada instante, además de contar con medidas de seguridad como cortafuegos o copias de seguridad.

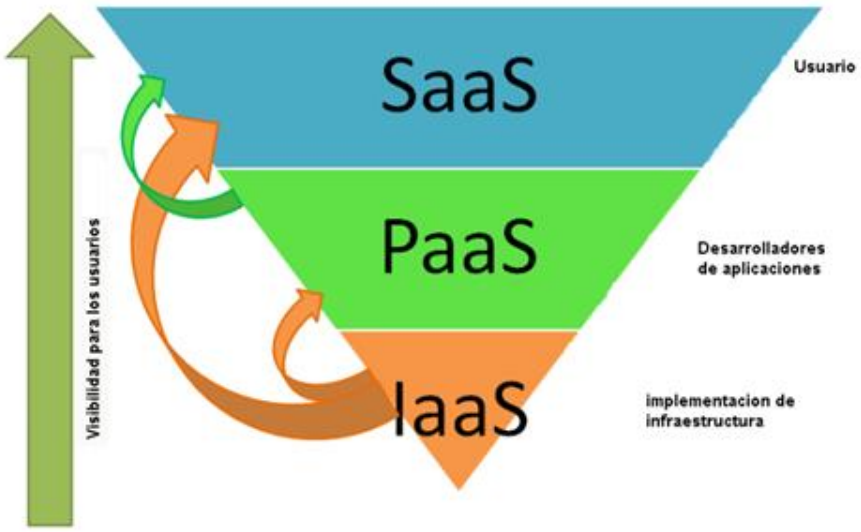


Ilustración 7 - Modelos de servicio en la nube

En 2003 Chris Pinkham y Benjamin Black, dos ingenieros que trabajaban en Amazon, publicaron un artículo (28) que buscaba mejorar la infraestructura de la compañía describiéndola como completamente estandarizada, automatizada y construida sobre servicios web. Además también comentaban la posibilidad de vender servidores virtuales como servicio a compañías externas. Siguiendo la estela de este documento, en 2006 Jeff Bezos, presidente de Amazon, anunció los servicios EC2 y S3.

Amazon Elastic Compute Cloud (EC2) es el engranaje clave de todos los servicios que Amazon ofrece. Provee servidores virtuales fácilmente configurables, a través de una API o mediante la interfaz web es posible tener un servidor funcional en cuestión de minutos, siendo posible posteriormente cambiar de nuevo los recursos tanto aumentándolos como disminuyéndolos. Al final de todo el usuario tan solo paga los recursos utilizados.

Amazon Simple Storage Service (S3) es el servicio de almacenamiento. Continuando la filosofía de escalabilidad de EC2, en los denominados *buckets* de datos el usuario puede guardar y recuperar datos de forma escalable y económica.

Aunque Amazon fue la primera gran compañía en adelantarse como proveedor de recursos en la nube, Microsoft anunció en 2008 Azure, una oferta bastante similar a la que Amazon ofrecía con EC2, aunque no vio la luz hasta 2010. Por otra parte, Google lanzó en 2008 su *App Engine*, que permitía construir directamente en la nube aplicaciones web. El servicio completo de infraestructura de Google no llegó hasta 2012 con *Google Compute Engine*.

Poco a poco todos los proveedores fueron aumentando su variedad de servicios de forma paralela con ligeras diferencias entre ellos. Estos servicios se resumen en los siguientes:

- Computación y redes: es la base de toda la infraestructura, compuesta por los servidores, las redes que los conectan, balanceadores de cargas, servicios de DNS o de elasticidad automática de los recursos de las instancias a tiempo real en función del tráfico.
- Bases de datos: se ofrecen desde bases de datos relacionales convencionales hasta del tipo *NoSQL*.
- Almacenamiento: espacio en la nube para almacenar datos estáticos, CDNs o discos que se pueden montar en servidores virtuales.
- Otros sevicios: email, colas de mensajes.

La aparición de la nube ha creado un nuevo paradigma que ofrece muchísimas ventajas, pero también acarrea ciertas consecuencias negativas. A modo de resumen final se exponen en la siguiente tabla las ventajas e inconvenientes de la computación en la nube.

Ventajas	Inconvenientes
• Elasticidad, los recursos físicos se adaptan a las necesidades del usuario, incluso existe la posibilidad de autoescalado en función del uso. • Economía, solo se pagan los recursos que se utilizan. • Simplicidad, gracias a los distintos modelos de servicio puedes eliminar capas que solo sumarían complejidad en el sistema. • Fiabilidad, el proveedor se encarga de los problemas que puedan surgir debido a un desastre	• Seguridad, confiar tus recursos informáticos a un tercero nunca será igual que mantenerlo tú mismo. • Privacidad, alojar en internet datos sensibles sin la certeza de que puedan acabar expuestos. • Control de hardware, al estar operando a altos niveles no son posibles modificaciones en los equipos físicos por parte del cliente. • Migración, adaptar infraestructuras antiguas a este nuevo paradigma puede ser muy costoso

Tabla 4 - Ventajas e inconvenientes del uso de la nube pública

La nube y contenedores

La aparición de Docker en 2013 puso todas las miradas sobre los contenedores, incluidas las de los proveedores de computación en la nube que intentan adelantarse a lo que sus usuarios van a demandar en los próximos años. Esto hizo que comenzaran a estudiar qué servicios pueden ofrecer para atraer clientes por encima de la competencia, una guerra que en pleno 2016 continúa muy activa y llena de innovaciones.

Actualmente la propia Docker ofrece un servicio que ellos denominan Contenedor como Servicio (CaaS) (29) que permite lanzar contenedores en la nube con funcionalidades como cortafuegos y con posibilidad de fácil orquestación usando Swarm.

A finales de 2014 Amazon anunció su propio servicio de contenedores denominado EC2 Container Service (ECS) (30). Como su nombre indica, funciona por encima de instancias virtuales en EC2, dando la posibilidad de crear un cluster formado por varios contenedores en varias máquinas. De manera similar a las herramientas de orquestación de contenedores, se ofrece la capacidad de sustitución en caso de errores. Otra funcionalidad es la integración con balanceadores de carga, lo que posibilita el reparto de tráfico entre varios contenedores. Un año después, Amazon complementó ECS con un registro de imágenes (ECR) compitiendo con el registro de Docker (Docker Hub). Los usuarios tienen la capacidad de subir las imágenes de sus propios contenedores para que estén públicas y cualquiera pueda usarlas o mantenerla en privado para que solo sean accesibles mediante autentificación, de manera similar a *GitHub* con los repositorios de Git.

La adaptación de los servicios en la nube por parte de Google fue mucho más sencilla gracias a su experiencia en la construcción de sistemas basados en contenedores. Su servicio de contenedores se denomina Google Container Engine (GKE) y está fuertemente inspirado en su herramienta de orquestación Kubernetes. La manera más sencilla de desplegar un cluster de contenedores con Kubernetes es en GKE debido a que Google ha dedicado muchos recursos en mejorar su experiencia de uso (31).

El último en adelantar posiciones sobre los contenedores ha sido Microsoft, que no ha publicado hasta 2016 su servicio denominado Azure Container Service (ACS). Hasta entonces había pervivido con una extensión desarrollada para sus máquinas virtuales y que permitía el fácil despliegue de contenedores sobre servidores previamente lanzados. Microsoft ofrece contenedores de Ubuntu o Centos en este servicio (32) ya que los contenedores de Windows no han llegado aún. Con el anuncio de ACS Microsoft da la posibilidad de la administración de contenedores sobre Docker Swarm o incluso orquestación por medio de Mesos.

El utilizar servidores en la nube ha cambiado el paradigma para los denominados *DevOps* (hace referencia a los profesionales que integran labores de desarrollo, *Development,* y operaciones, *Operations*) y administradores de sistemas ya que han tenido que actualizar la forma en la que tratan a los servidores. La comparación entre mascotas y ganado ejemplifica este cambio (33). La manera tradicional de administrar servidores es dándoles un cuidado especial, si alguno tiene alguna avería se usan los recursos que hagan falta para arreglarlo y mejorarlo de manera similar a una mascota que acompaña a su dueño muchos años. Sin embargo a los servidores en la nube se les trata de manera similar a ganado, donde se puede reemplazar fácilmente una máquina por otra y no compensa gastar tiempo y dinero en una reparación.

La aparición de los contenedores y en concreto de Docker en 2013 no hace más que favorecer esta nueva visión donde separar los diferentes servicios de una aplicación en cajones independientes da gran consistencia al sistema. Al igual que una máquina virtual en la nube es fácilmente reemplazable por otra, la vida del contenedor está planeada para que sea efímera y que rápidamente se pueda cambiar por otro perfectamente funcional en caso de que muestre síntomas de error. Con herramientas de orquestación como Kubernetes todo esto ocurre de forma transparente al usuario debido a que es autónomo, e incluso se busca el reparto de los servicios entre diferentes máquinas virtuales en la nube para tener la máxima eficiencia.

3 HERRAMIENTAS EMPLEADAS

3.1 E-nefro

E-nefro es una aplicación desarrollada para el Grupo de Ingeniería Biomédica de la Universidad de Sevilla que ayuda a mantener un seguimiento médico de pacientes con problemas renales por medio de un registro de enfermos y sus diagnósticos. La aplicación permite distintos casos de uso en función de varios roles como administrador, médico, enfermero o paciente.

Está desarrollada con Java *servlets* por lo que necesitará un programa como Tomcat que funcione de contenedor web. Para persistir la información se usa un gestor de base de datos, PostgreSQL.

En el manual de instalación de E-nefro se indica la posibilidad de desplegar la aplicación usando una máquina virtual con toda la configuración preparada o por medio de unas instrucciones de instalación desde cero. Para el despliegue con contenedores y en la nube no será necesaria la máquina virtual y se usarán las instrucciones.

La aplicación está formada por los siguientes ficheros:

- *enefro.war*: contiene la aplicación web lista para ser desplegada en Tomcat.

- Tres ficheros .sql: servirán para inicializar la base de datos y popular las tablas con la información necesaria para la aplicación.

Las instrucciones de instalación del manual son las siguientes:

1. Instalar Tomcat y PostgreSQL

2. Crear los directorios con los *tablespaces* de la base de datos. Serán tres y por defecto estarán en */opt/POSTGRESDB*:
   - *TS_ENEFRO_SEVILLAROCIO_IDX*
   - *TS_ENEFRO_SEVILLAROCIO_DATOS*
   - *TS_ENEFRO_SEVILLAROCIO_TMP*

3. Importar los tres ficheros .sql en el siguiente orden y con los usuarios adecuados:
   - *createBBDD_Sevillarocio.sql* como usuario postgres
   - *createBBDD_Sevillarocio_User.sql* como el usuario de la aplicación y en la base de datos de la aplicación.
   - *database_sevillarocio.sql* como el usuario de la aplicación y en la base de datos de la aplicación.
4. Configurar correctamente el fichero *aplicación.properties* dentro del fichero *.war*. Descomprimir y volver a comprimir el fichero *.war* si fuera necesario modificarlo.

5. Desplegar la aplicación en Tomcat.

Durante el desarrollo del proyecto se adaptarán estas instrucciones al modelo deseado sobre contenedores y la nube.

3.2 Docker

Tras el estudio de las diferentes tecnologías de contenedores en la sección anterior, finalmente en el desarrollo del proyecto se hará uso de Docker. Esta decisión se debe a la creciente aceptación que ha ido adquiriendo en los últimos dos años por parte de la comunidad y grandes compañías. Por ello existe mucha bibliografía sobre su uso y es posible utilizarlo en múltiples plataformas. Otra de las ventajas es el número de imágenes de contenedores que hay disponibles en el registro de Docker, cada día más desarrolladores suben imágenes de sus aplicaciones allí. Aunque Rocket es el competidor más cercano a Docker actualmente no tiene el mismo apoyo ni cantidad de imágenes disponibles.

La estructura de Docker está formada por una aplicación cliente y una máquina host que pueden funcionar en el mismo o en diferentes entornos. El host ejecuta un proceso demonio encargado de escuchar y ejecutar órdenes tales como lanzar un contenedor, matarlo o saber su estado. También será el que se encargue de albergar los contenedores. El cliente a su vez será el que mande las órdenes al demonio por medio de una API REST o usando *sockets*. El cliente puede funcionar en la misma máquina que el demonio o conectarse de forma remota (34).

Actualmente Docker solo funciona de forma nativa en Linux. Para instalarlo en alguna de sus distribuciones se puede descargar usando los repositorios, las instrucciones se encuentran en el siguiente enlace https://docs.docker.com/engine/installation/linux/.

Para Mac o Windows hay una solución no nativa en la que se instala un entorno llamado Docker Toolbox consistente en una máquina virtual de Linux desde donde correrá Docker, aunque para el usuario el funcionamiento será transparente. Esta alternativa se encuentra en los siguientes enlaces: https://docs.docker.com/engine/installation/windows https://docs.docker.com/engine/installation/mac/.

El comando principal de Docker es *docker run*, que lanza un contenedor a partir de una imagen y ejecuta el comando que se le indique. En el siguiente ejemplo se lanzará un contenedor de la imagen llamada *ubuntu:14.04* y se ejecutará el comando especificado, en este caso imprimir *Hola mundo.*

```
$ docker run ubuntu:14.04 /bin/echo 'Hola mundo!'
Hola mundo!
```

Un contenedor de Docker solo permanece en ejecución mientras el proceso lanzado esté en ejecución. Por tanto en este ejemplo el contenedor muere una vez que se ha impreso *Hola mundo*.

El siguiente ejemplo imprimirá el mensaje cada un segundo con la diferencia de que el contenedor no morirá al ser un bucle *while* infinito.

```
$ docker run ubuntu /bin/bash -c "while true; do echo 'Hola mundo!'; sleep 1;
done"
Hola mundo!
Hola mundo!
Hola mundo!
…
```

Si se añade la opción *–d* el contenedor funcionará en segundo plano. Con el comando *docker ps* se pueden ver los contenedores que están vivos en el host junto con qué procesos están ejecutando (35).

Como se ha visto en los ejemplos anteriores, para ejecutar un contenedor se necesita una imagen. Una imagen de Docker es una plantilla donde se especifica qué software irá dentro del contenedor y qué comandos se ejecutarán. Se puede crear una imagen nueva, utilizar directamente imágenes construidas por otros o modificarlas añadiendo cambios. Las imágenes ya construidas se encuentran en un registro

denominado Docker Hub (36). En él muchas compañías han subido imágenes oficiales de su propio software, como el caso de la fundación Apache que ofrece imágenes de Tomcat.

En el host de Docker hay un registro local donde se van almacenando las imágenes que se van utilizando, tanto del registro online como las personalizadas, similar a una memoria caché. Cuando se lanza un contenedor usando *docker run*, Docker busca el nombre de la imagen que se ha especificado en el comando localmente y en caso de que no existiera la descarga del Hub.

En la siguiente figura se representa el proceso de un cliente remoto mandando comandos al proceso demonio que está dentro del host:

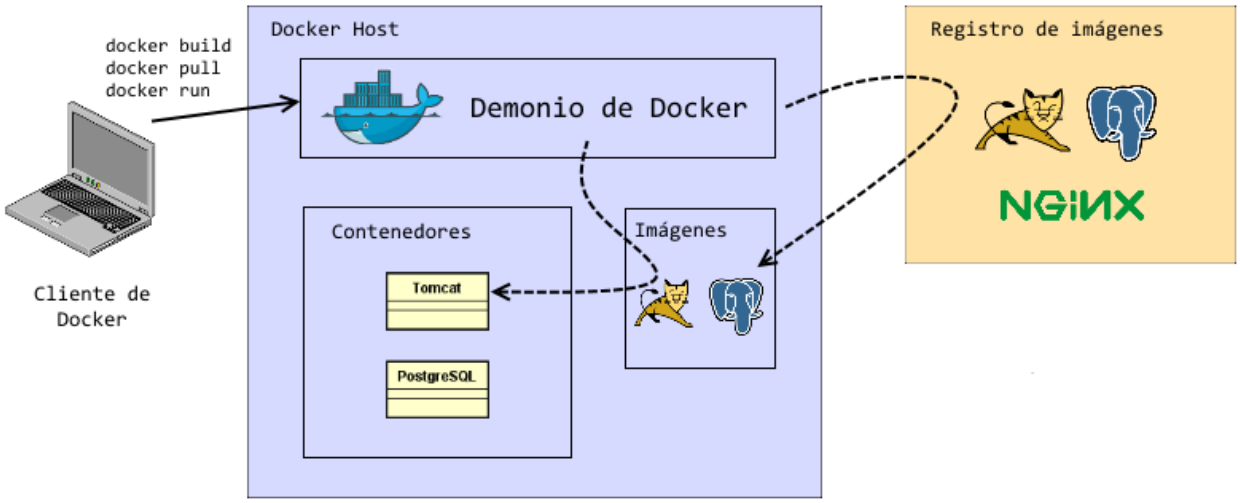


Ilustración 8 - Funcionamiento de Docker

El cliente tiene dos comandos relacionados con las imágenes. El primero es *docker pull*, encargado de descargar la imagen del registro de Docker al registro local del host. El segundo es *docker build*, que construirá una imagen personalizada por medio de un fichero denominado *Dockerfile*. En él se especifica una imagen base del registro y sobre ella se hacen las modificaciones escritas en las posteriores líneas. Cada una de estas líneas va a ir creando una capa diferente en la imagen, como podría ser descargar un paquete o mover un fichero de un directorio a otro.

En el siguiente ejemplo tendremos un servidor Apache en un contenedor. Para ello primero es necesario construir una imagen con un *Dockerfile* y el comando *docker build* para terminar usando *docker run* con esa imagen. El *Dockerfile* será el siguiente:

```
FROM ubuntu:14.04
RUN apt-get update && apt-get install -y apache2
```

Dockerfile

Código 1 – Dockerfile de servidor apache

En la primera línea se indica con la instrucción *FROM* que la imagen base será Ubuntu 14.04. La segunda instrucción *RUN* sirve para ejecutar el comando, en este caso actualizar el gestor de paquetes del sistema e instalar Apache.

Este fichero necesita llamarse *Dockerfile* para que cuando se ejecute el comando *docker build* construya la imagen.

```
$ docker build -t ejemplo-apache /directorio/con/dockerfile
```

-t es para nombrar a la imagen como *ejemplo-apache* y el directorio es donde estará el fichero *Dockerfile*.

Podemos listar las imágenes que están en el registro local usando el comando *images.*

```
$ docker images
REPOSITORY      TAG          IMAGE ID       CREATED             SIZE
ubuntu         14.04        4s0402sfpo98    1 day ago          187.9 MB
ejemplo-apache              ahd0f21sx3c4    1 day ago          201.4 MB
```

Con la imagen construida, se puede ejecutar el servicio de apache en segundo plano con *–d* y utilizando el comando de iniciación del servicio *httpd -DFOREGROUND*:

```
$ docker run –d ejemplo-apache httpd –DFOREGROUND
```

Aunque apache esté en ejecución no será accesible hasta que no se haga una traducción de puertos. De manera similar a NAT se establece un mapeo de puertos del contenedor y del host por medio de la opción *–p puerto-host:puerto-contenedor*. Por defecto apache usa el puerto 80 así que para tener el servidor web en el puerto 8080 habría que ejecutar el siguiente comando:

```
$ docker run –p 8080:80 –d ejemplo-apache httpd –DFOREGROUND
```

Si no se define un comando al final de docker run se ejecutará el que se haya escrito en el Dockerfile por medio de la instrucción CMD.

```
FROM ubuntu:14.04
RUN apt-get update && apt-get install -y apache2
CMD [‘httpd’, ‘-DFOREGROUND’]
```

Dockerfile

Código 2 - Dockerfile de servidor apache con CMD

El funcionamiento será igual si se construye esta imagen con nombre *ejemplo-apache2* y se ejecuta el siguiente *docker run:*

```
$ docker run –p 8080:80 –d ejemplo-apache2
```

La vida de un contenedor está planeada para ser corta y que sea fácilmente sustituible por otro en segundos. Este hecho puede llevar a la pregunta de qué ocurre con la información persistente como nuevas entradas en una base de datos o ficheros subidos al servidor web. Para guardar la información y compartirla entre contenedores Docker proporciona mecanismos llamados volúmenes y contenedores de datos (37).

Un volumen es un directorio dentro del contenedor que se señala para que sea persistente y su contenido se guardará pase lo que pase con la vida del contenedor. Un volumen también puede ser compartido entre varios contenedores. Cuando el contenedor se crea, el volumen se inicializa y la información que existía previamente en ese volumen se copia al nuevo contenedor.

El volumen se define durante la ejecución del contenedor en el comando *docker run* con la opción *–v /directorio/volumen*. En el siguiente ejemplo se guardaría la carpeta */home/user/images*:

```
$ docker run –v /home/user/images ubuntu
```

Es posible montar un directorio del host dentro del contenedor como volumen. Si en el host existe el directorio */src/info* con información requerida para el contenedor, se puede montar dentro de él en el directorio */home/user/info* con la opción *–v /src/info:/home/user/info*.

```
$ docker run –v /src/info:/home/user/info ubuntu
```

Si ya existiera información en el directorio */home/user/info* dentro del contenedor se copia lo que haya en el host sin eliminar nada del contenido anterior.

Haciendo uso de volúmenes, para persistir información de un directorio en un determinado contenedor es recomendable lanzar un contenedor de datos. Este contenedor no ejecutará ninguna aplicación, tan solo persistirá un directorio.

```
$ docker run –v /directorio/persistir --name contenedordatos ubuntu echo "Volumen de datos"
```

Entonces en el contenedor que se quiera persistir una carpeta se usa la opción *--volumes-from*

```
$ docker run --volumes-from contenedordatos imagen-contenedor
```

De esta manera cada contenedor nuevo que se lance con la opción *--volumes-from contenedordatos* respetará el contenido de la carpeta */directorio/persistir*.

Aunque se pare el contenedor de datos o se borre, los volúmenes definidos no se eliminarán del host. La única forma de que se borren es con el comando *docker rm –v nombre-contenedor*. La opción *–v* indica que se borren los volúmenes asociados a ese contenedor.

3.3 Azure

El servicio de nube que se utilizará en el proyecto será Azure. Amazon o Google ofrecen una mejor interacción con contenedores al haber invertido antes en ellos, no obstante Microsoft aporta ventajas a estudiantes, como crédito gratuito en la nube, hecho fundamental para tomar la decisión de usar Azure.

Aunque Azure ofrece varios modelos, el que se va a usar será el de infraestructura como servicio (IaaS). Concretamente se lanzarán máquinas virtuales y se hará uso de las facilidades que da la nube.

Gracias a la elasticidad de la nube, será posible modificar el tamaño de las instancias, aumentando o disminuyendo la memoria y CPU de las máquinas virtuales en tiempo real sin que las aplicaciones dejen de funcionar.

La infraestructura viene acompañada de un sistema de cortafuegos denominado *endpoints*, que permiten abrir los puertos necesarios para direcciones IP concretas, pudiendo incluso utilizar mecanismos de traducción de puertos de una manera similar a NAT.

Azure da la posibilidad de crear una imagen de una instancia y guardarla para posteriores usos, como puede ser una copia de seguridad o para replicar varias instancias iguales.

Para la comunicación con Azure se hará uso de dos herramientas. La primera será la interfaz por línea de comandos y la segunda la interfaz web accesible desde http://portal.azure.com.

La herramienta de línea de comandos se puede instalar usando un instalador en este enlace https://azure.microsoft.com/en-us/documentation/articles/xplat-cli-install/ o instalarlo como paquete *npm* con el siguiente comando:

```
$ npm install azure-cli –g
```

Tras haberlo instalado se puede acceder a la ayuda ejecutando:

```
$ azure help
```

3.4 Kubernetes

Las herramientas de orquestación de contenedores son necesarias cuando se quiere desplegar una aplicación entre varias máquinas. Durante el desarrollo del proyecto se escogerá Kubernetes (38).

Esta decisión se toma debido a que actualmente Kubernetes está en un nivel de madurez superior al de otras alternativas, además de ofrecer un script para el despliegue sencillo sobre Azure. En su web se encuentra una extensa documentación (39) con enlaces a código de ejemplo en GitHub. La razón de no usar Docker Swarm es darle diversidad al proyecto evitando continuar con tecnologías desarrolladas por Docker.

Aunque Kubernetes no está desarrollado por Docker está basado en sus contenedores. Su funcionalidad principal es monitorizar y mantener el sistema en un estado indicado previamente. Por ejemplo si se requiere que tres contenedores de un servidor web y uno de la base de datos estén corriendo, Kubernetes va a estar analizando continuamente todos los nodos para hacerse cargo de que esos cuatro contenedores estén funcionando correctamente, y si detecta que uno de ellos no funciona se encargará de reemplazarlo inmediatamente.

Además, permite realizar las siguientes acciones que se desarrollarán en el capítulo 4.3:

- Automatizar el despliegue y la replicación de los contenedores
- Escalar el número de contenedores sin necesidad de pararlos
- Organizar los contenedores por grupos y balancear el tráfico
- En caso de que un contenedor se pare, relanzar otro igual

Conceptos de Kubernetes

En el siguiente esquema se muestran los conceptos que se van a ir definiendo:

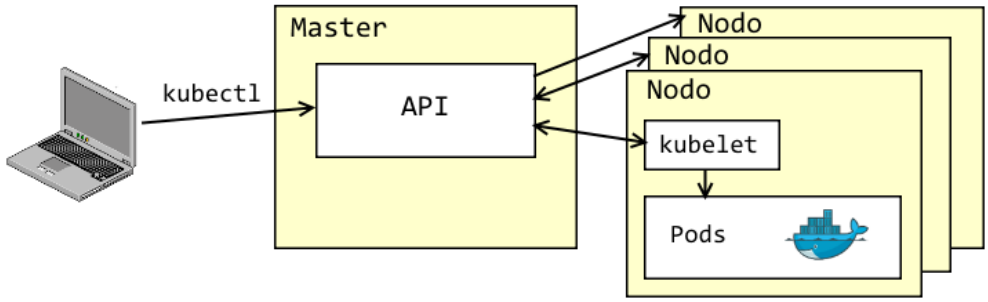


Ilustración 9 - Arquitectura simplificada de Kubernetes

Cluster

Es el conjunto de máquinas físicas o virtuales en las cuales se ejecutan los servicios que Kubernetes necesita. Está compuesto por las máquinas que funcionan como nodos y el master.

Un cluster se puede desplegar en múltiples plataformas, desde infraestructuras físicas, en la nube o en máquinas virtuales locales (40).

Master

El control principal de un cluster de Kubernetes se desarrolla en un servidor denominado master. Este contiene diversos servicios que sirven para manejar las comunicaciones y la administración de otros nodos. La comunicación entre los servicios funciona a través de una API REST.

El servicio más importante es *etcd* (41), un sistema de almacenamiento distribuido de tipo clave-valor (DHT (42)) donde se guardará toda la información referente a los nodos. Se utilizará para descubrir y guardar el estado de cada componente.

Un usuario puede comunicarse con el master utilizando un cliente de línea de comandos llamado *kubectl* que permite administrar el cluster, similar al cliente de Docker. Este cliente puede ser remoto o estar dentro de la propia máquina master.

Nodo o Minion (43)

Son las máquinas virtuales o físicas administradas por el master. Será donde los contenedores estén corriendo, por ello es necesario que tengan Docker instalado.

La comunicación de los nodos con el master es a través de un servicio denominado *kubelet* y de manera transparente para el usuario. Es diferente a la herramienta *kubectl* que acepta los comandos que manda un usuario al master. El servicio *kubelet* recibe las órdenes del master y ejecuta los comandos necesarios dentro del nodo, como iniciar un pod.

Pod (44)

Se trata de un conjunto de uno o más contenedores que funcionarán en un mismo nodo, por lo que compartirán la red y recursos. Al igual que los contenedores, se consideran efímeros y fácilmente reemplazables.

La agrupación de contenedores en pods debe hacerse teniendo en cuenta que comparten recursos. Si por ejemplo dos contenedores necesitan compartir un mismo volumen, estos dos deberían estar juntos en el mismo pod. En una aplicación con la arquitectura típica de servidor web más base de datos se usará un pod para cada uno ya que estos no necesitan compartir recursos, tan solo necesitan comunicarse.

Los pods estarán distribuidos entre los nodos disponibles, pudiendo funcionar el pod servidor web en una máquina y el pod base de datos en otra. Si un pod del mismo tipo tuviese varias réplicas, estas réplicas también estarán distribuidas entre los nodos.

Replication Controller (45)

En la declaración de un pod se pueden especificar cuantas réplicas va a tener este mediante un *replication controller (RC)*. El *RC* asegura que un número determinado de réplicas de un pod va a estar activo en todo momento. Se encargará de monitorizar cuantos pods están corriendo y matar o crear los que hagan falta para cumplir el estado deseado.

En el siguiente ejemplo se muestra el funcionamiento de un *RC*:

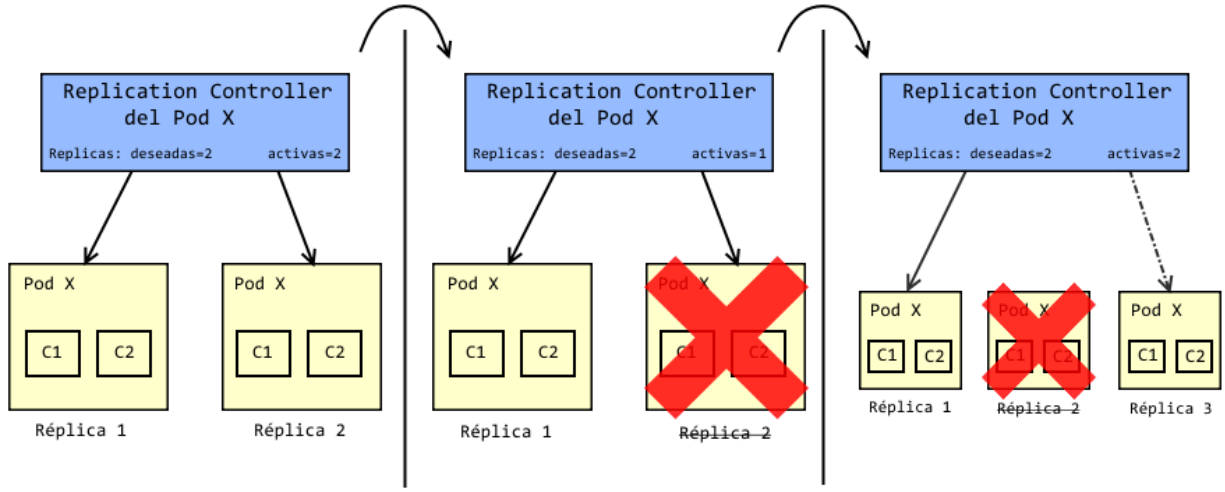


Ilustración 10 - Ejemplo Replication Controller

Un pod X compuesto de dos contenedores C1 y C2 declara un *RC* con dos réplicas activas. Tras unos segundos de funcionamiento, la segunda réplica deja de funcionar. Cuando el *RC* se da cuenta, lanza una tercera réplica, cumpliendo el estado deseado de dos réplicas activas.

Kubernetes permite escalar el número de réplicas de pods de forma dinámica, por lo que si se indica aumentar o disminuir las réplicas el *RC* es el que los crea o destruye.

Cuando el *RC* crea una nueva réplica de un pod lo hará en cualquiera de los nodos que tenga disponible en función de los recursos disponibles en cada uno de ellos. Es decir, no todas las mismas réplicas de un mismo pod tienen por qué funcionar en el mismo nodo, estarán distribuidas.

Servicio [46]

Cada pod puede tener asociado un servicio. Un servicio permite que aunque las réplicas de un mismo pod se destruyan o sustituyan estos sigan siendo localizables y la comunicación no se rompa. Si por ejemplo un pod servidor web y otro pod base de datos se comunican entre sí por la dirección IP pero el pod de la base de datos se reinicia y su dirección cambia, el servicio va a permitir que la conexión continúe activa.

Otra función de los servicios es de balanceador de carga entre las distintas réplicas de un pod. El servicio asociado al pod recibe las peticiones entrantes y las manda a una réplica determinada.

La labor de los servicios es fundamental ya que Kubernetes es un sistema distribuido entre varios nodos y estos se encargan de que la comunicación entre pods y sus réplicas, aunque estén en diferentes máquinas, nunca se pierda.

Volúmenes persistentes (47)

Al igual que ocurre con los contenedores de Docker, Kubernetes ofrece volúmenes para persistir la información. Sin embargo mientras que en Docker todos los contenedores conviven en un mismo servidor en Kubernetes la dificultad es mayor al ser un sistema distribuido en varias máquinas.

Si un pod con varias réplicas distribuidas por los nodos quisiera persistir la información, necesita un volumen persistente que se monte en un lugar accesible para todos los nodos.

La manera más sencilla de tener un volumen persistente es usando la nube de Google o Amazon, ya que Kubernetes implementa soluciones donde se puede lanzar un disco independiente a los nodos que almacene

la información de los volúmenes. Otras alternativas están basadas en lanzar servidores con tecnologías como NFS (48).

En Azure esta solución aún no está desarrollada, por lo que para persistir la información se utilizará un *HostPath* (49), un volumen montado en el nodo donde esté el pod. El problema que implica esta alternativa es que el sistema deberá tener un único nodo. Durante el desarrollo del proyecto se tendrá en cuenta esta limitación al crear un volumen.

La definición de un volumen está compuesta de dos elementos. La primera es la declaración del volumen en sí, indicando la solución adoptada (como un disco en la nube de Google o un *HostPath)*, la capacidad que tendrá o los modos de acceso (escritura/lectura). La segunda parte es la petición por parte de un pod de acceder a un volumen llamada *PersistentVolumeClaim*.

Plantillas

Para la definición de pods, servicios, *RC* y volúmenes se utilizan plantillas en formato *yaml* en el cual se definen sus propiedades, etiquetas y metadatos.

El siguiente ejemplo es la plantilla de definición de un pod compuesto por un solo contenedor de MySQL:

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: mysql-pod
spec:
  containers:
  - image: mysql
    name: mysql-contenedor
    ports:
    - containerPort: 3306
      name: mysql-puerto
    volumeMounts:
    - mountPath: /home/user/datos
      name: mysql-volumenDatos
  volumes:
  - name: mysql-volumen
    persistentVolumeClaim:
          claimName: mysql-peticion-volumen
```

mysql-pod.yaml

Código 3 - Plantilla de un pod de MySQL

En *kind* se define el objeto que es, en este caso pod, y en *metadata* se le da nombre. En *spec* se indica qué contenedores forman el pod, indicando la imagen, el nombre y qué puertos requiere. Por último se especifican los volúmenes que necesite el contenedor en *volumeMounts*. En el apartado final de *volumes* es donde se hará la petición de volúmenes con la etiqueta *persistentVolumeClaim*.

4 DESARROLLO DEL PROYECTO

4.1 Despliegue de E-nefro sobre contenedores

Durante este apartado se automatizará el proceso de despliegue de E-nefro en un servidor local con la ayuda de los contenedores de Docker. El entorno será único, por lo que la misma máquina albergará el cliente y el host de Docker.

Cada servicio que necesite la aplicación funcionará en un contenedor diferente. Esto favorecerá la independencia entre ellos y la elasticidad de recursos. El sistema inicial va a quedar dividido en tres contenedores diferentes:

- El primero será el de Tomcat, que contendrá el fichero *enefro.war* para el despliegue de la aplicación.
- La base de datos irá en un contenedor de PostgreSQL que se inicializará con los ficheros sql de E-nefro.
- Un contenedor de datos que haga que la información introducida nueva persista aunque el contenedor de la base de datos se reinicie.

En la siguiente figura se muestra el sistema:

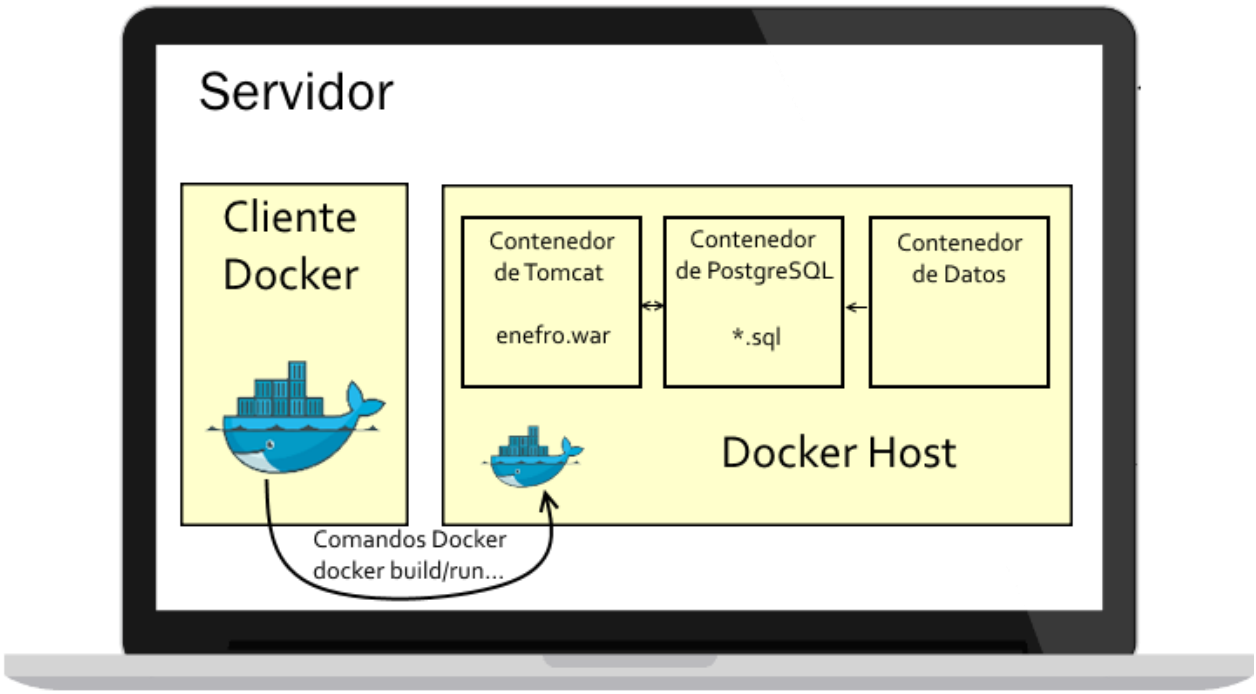


Ilustración 11 - Esquema del servidor

Esta sección se va a organizar en un apartado para cada uno de los contenedores necesarios: base de datos, Tomcat y persistencia de la información. Cuando la aplicación se haya desplegado correctamente se explicará el funcionamiento de un contenedor proxy que redirija a la aplicación y pueda balancear entre varias instancias de Tomcat.

Al final de la sección se automatizarán los comandos de ejecución de contenedores por medio de la herramienta Docker Compose.

4.1.1 Contenedor de PostgreSQL

PostgreSQL tiene una imagen oficial en el registro de Docker llamada *postgres*. Por tanto se lanzará un contenedor a partir de esta imagen con los ficheros sql de E-nefro.

En las instrucciones de instalación de la aplicación se pedía importar los tres ficheros sql en el siguiente orden y con los usuarios adecuados:

1. *createBBDD_Sevillarocio.sql* como usuario postgres
2. *createBBDD_Sevillarocio_User.sql* como el usuario de la aplicación y en la base de datos de la aplicación.
3. *database_sevillarocio.sql* como el usuario de la aplicación y en la base de datos de la aplicación.

En la documentación del contenedor de PostgreSQL (50) se indica que los ficheros *sql* y *sh* que se coloquen en el directorio */docker-entrypoint-initdb.d* se importarán como usuario *postgres* en la base de datos durante el proceso de inicialización del contenedor. Esto servirá para poder ejecutar los tres ficheros sql de la aplicación.

El primer fichero se inicializará correctamente ya que se importar como usuario *postgres.* Sin embargo los dos siguientes se deben importar como el usuario de la aplicación. Este problema se soluciona añadiendo la siguiente línea al inicio de los dos ficheros *createBBDD_Sevillarocio_User.sql* y *database_sevillarocio.sql*.

```
\c enefro_sevillarocio enefro_owner
[…]
                                                    database_sevillarocio.sql
                                                createBBDD_Sevillarocio_User.sql
```

Código 4 - Línea necesaria al inicio de los dos ficheros sql

Esta línea indica la conexión a la base de datos *enefro_sevillarocio* con el usuario *enefro_owner*. Estos nombres son los escogidos por defecto por los desarrolladores de E-nefro.

Para adjuntar los ficheros al contenedor de PostgreSQL se modificará la imagen oficial utilizando un fichero *Dockerfile*. Se va a utilizar el mismo entorno cliente y host de Docker, por lo que se construirá la imagen usando la consola del servidor. Con los siguientes comandos se creará una carpeta donde se copien los ficheros y se edite el *Dockerfile*:

```
$ mkdir contenedor-postgres/
$ cp createBBDD_Sevillarocio.sql contenedor-postgres/scripts/1-
createBBDD_Sevillarocio.sql
$ cp createBBDD_Sevillarocio_User.sql contenedor-postgres/scripts/2-
createBBDD_Sevillarocio_User.sql
$ cp database_sevillarocio.sql contenedor-postgres/scripts/3-
database_sevillarocio.sql
$ touch contenedor-postgres/Dockerfile
```

La carpeta quedará con los siguientes ficheros:

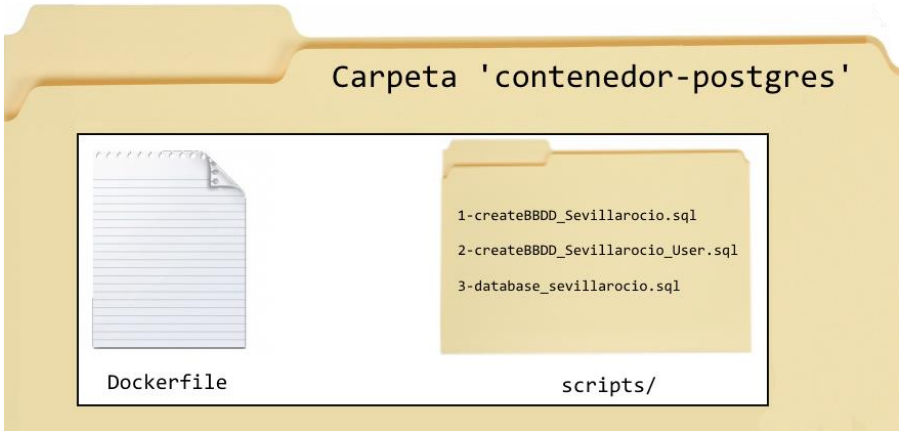


Ilustración 12 - Contenido de la carpeta contenedor-postgres

Se han numerado los tres ficheros sql al ser relevante para E-nefro el orden de ejecución y en la documentación de PostgreSQL se especifica que los ficheros se importan por orden alfabético.

El contenido completo del *Dockerfile* es el siguiente:

```
FROM postgres
ADD scripts/ /docker-entrypoint-initdb.d
RUN echo "listen_addresses='*'" >>
/var/lib/postgresql/data/postgresql.conf
RUN mkdir -p /opt/POSTGRESDB/TS_ENEFRO_SEVILLAROCIO_TMP
/opt/POSTGRESDB/TS_ENEFRO_SEVILLAROCIO_DATOS
/opt/POSTGRESDB/TS_ENEFRO_SEVILLAROCIO_IDX
RUN chown postgres:postgres -R /opt/*
                                                    Dockerfile
```

Código 5 - Dockerfile del contenedor de PostgreSQL

La primera línea es la declaración de su imagen base, la oficial de PostgreSQL:

```
FROM postgres
```

Se copian los ficheros sql dentro de la carpeta */docker-entrypoint-initdb.d* del contenedor, que como detallaba la documentación permitirá ejecutarlos en el arranque. Para ello se usa la instrucción *ADD origen destino*, que añade el fichero o directorio *origen* en el cliente a *destino* dentro del contenedor.

```
ADD scripts/ /docker-entrypoint-initdb.d
```

En las instrucciones de E-nefro se pedía crear las carpetas para los *tablespace* en */opt/POSTGRESDB*. Además se le da permisos al usuario *postgres* para que pueda escribir y leer. Para ejecutar estos comandos se usa la instrucción *RUN comando*:

```
RUN mkdir -p /opt/POSTGRESDB/TS_ENEFRO_SEVILLAROCIO_TMP
/opt/POSTGRESDB/TS_ENEFRO_SEVILLAROCIO_DATOS
/opt/POSTGRESDB/TS_ENEFRO_SEVILLAROCIO_IDX
RUN chown postgres:postgres -R /opt/*
```

Por último se necesita cambiar una configuración de PostgreSQL. Por defecto esta aplicación no habilita las conexiones externas. E-nefro necesita que Tomcat pueda acceder y realizar consultas así que se añade la línea *listen_addresses='*'* en el fichero *postgresql.conf* (51):

```
RUN echo "listen_addresses='*'" >>
/var/lib/postgresql/data/postgresql.conf
```

Para construir la imagen se usará el comando *docker build* junto con la opción *–t* para nombrar a la imagen *enefro-database.*

```
$ docker build -t enefro-database contenedor-postgres/
Sending build context to Docker daemon 21.02 MB
Step 1 : FROM postgres
 ---> 54fa18d9f3b6
Step 2 : ADD scripts2/ /docker-entrypoint-initdb.d
 ---> 511c73bf9917
Removing intermediate container 35856c376a77
Step 3 : RUN echo "listen_addresses='*'" >>
/var/lib/postgresql/data/postgresql.conf
 ---> Running in eb64632b9d9a
 ---> e5ce6d68592d
Removing intermediate container eb64632b9d9a
Step 4 : RUN mkdir -p /opt/POSTGRESDB/TS_ENEFRO_SEVILLAROCIO_TMP
/opt/POSTGRESDB/TS_ENEFRO_SEVILLAROCIO_DATOS
/opt/POSTGRESDB/TS_ENEFRO_SEVILLAROCIO_IDX
 ---> Running in 900d57a49e98
 ---> 40333e0523a6
Removing intermediate container 900d57a49e98
Successfully built 40333e0523a6
```

Con la imagen construida se podría ejecutar el contenedor con *docker run*, pero antes de hacerlo se va a construir el de Tomcat.

4.1.2 Contenedor de Tomcat

Para desplegar el fichero *enefro.war* es necesario un contenedor con Tomcat. Apache tiene una imagen oficial en el registro de Docker que permite un rápido despliegue sobre contenedores.

Colocando el fichero *war* en el directorio *webapps*, Tomcat se encargará de desplegarlo automáticamente en el arranque en http://dominio:puerto/nombre_aplicación (52).

Al igual que en el contenedor anterior, se creará un directorio en el servidor donde se encuentre el *Dockerfile* y el fichero *enefro.war*. Esta carpeta será la que el cliente de Docker le pase al host para construir la imagen.

Dentro de la consola del servidor se ejecutan los siguientes comandos:

```
$ mkdir contenedor-tomcat/
$ cp enefro.war contenedor-tomcat/
$ touch contenedor-tomcat/Dockerfile
```

La carpeta *contenedor-tomcat* quedará así:

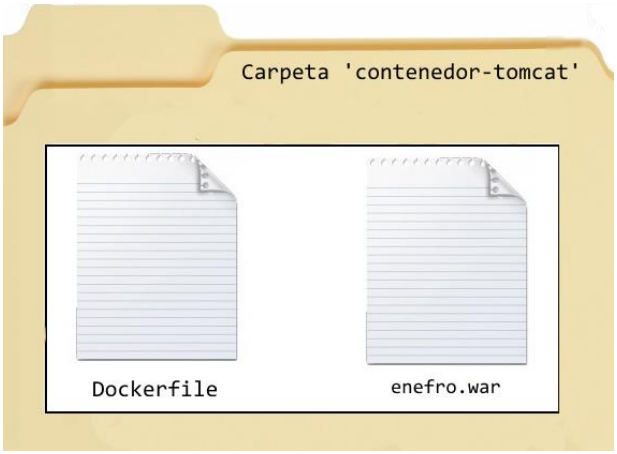


Ilustración 13 - Contenido de la carpeta 'contenedor-tomcat'

El fichero *Dockerfile* tendrá el siguiente contenido:

```
FROM tomcat:7.0.67-jre7

RUN rm -rf /usr/local/tomcat/webapps/*
ADD enefro.war /usr/local/tomcat/webapps/
```
Dockerfile

Código 6 - Dockerfile del contenedor de Tomcat

La primera línea del *Dockerfile* es la imagen base, la oficial de Tomcat, en concreto la versión 7.0.67 que es la que requiere la aplicación.

```
FROM tomcat:7.0.67-jre7
```

Antes de copiar el fichero *war* es conveniente que eliminar las aplicaciones de ejemplo de Tomcat para que no interfieran con E-nefro.

```
RUN rm -rf /usr/local/tomcat/webapps/*
```

Por último el contenedor necesita el fichero *enefro.war* para que se despliegue correctamente, por lo que se añade a la carpeta *webapps* de Tomcat

```
ADD enefro.war /usr/local/tomcat/webapps/
```

Antes de construir el contenedor es necesario editar el fichero de configuración de E-nefro con la información de conexión a la base de datos.

Este fichero se encuentra dentro del fichero *enefro.war*, concretamente en *WEB-INF/classes/resources/aplicacion.properties*. Para modificar este fichero, se descomprime el fichero *enefro.war*:

```
$ tar xzf enefro.war -C enefro_descomprimido
$ vi enefro_descomprimido/WEB-INF/classes/resources/aplicacion.properties
```

En el fichero de configuración se encuentran los siguientes parámetros entre otros:

```
[…]
database.url=
database.usuario=
database.clave=
[…]
                                                    aplicación.properties
```

Código 7 - Parámetros de configuración de la base de datos de E-nefro

En usuario y clave se introducen los datos introducidos en los ficheros *sql* anteriores. En URL se usará el conector de PostgreSQL:

```
jdbc:postgresql://alias_postgres/nombre_bbdd_aplicacion
```

alias_postgres será el alias escogido posteriormente en la conexión entre los contenedores de Tomcat y PostgreSQL. Docker se encarga automáticamente de sustituir este alias por el dominio del contenedor de base de datos.

Una vez editado el fichero con los cambios anteriores, se vuelve a comprimir el fichero *enefro.war:*

```
$ cd enefro_descomprimido
$ jar cvf enefro.war *
```

Este nuevo *enefro.war* estará preparado para conectarse con el contenedor de PostgreSQL.

Ya es posible construir la imagen con *docker build*, etiquetándola como *enefro-tomcat.*

```
$ docker build -t enefro-tomcat contenedor-tomcat
Sending build context to Docker daemon 867.6 MB
Step 1 : FROM tomcat:7.0.67-jre7
 ---> 4f703d3a81a1
Step 2 : RUN rm -rf /usr/local/tomcat/webapps/*
 ---> Using cache
 ---> 0e08289ce4e2
Step 3 : ADD enefro.war /usr/local/tomcat/webapps/
 ---> 6589d052a493
Removing intermediate container 325a358cbf1f
Successfully built 6589d052a493
```

Con la imagen construida, se pueden lanzar los contenedores de la base de datos y de Tomcat a partir de las imágenes con el comando *docker run*. Es importante abrir el puerto 8080 en el contenedor de Tomcat para que la aplicación sea accesible. Con la opción *–d* se mantiene el contenedor en ejecución en segundo plano.

```
$ docker run -d ––name=database enefro-database
$ docker run -d -p 8080:8080 ––link=database:alias ––name=aplicacion enefro-
tomcat
```

Tomcat necesita tener conexión con la base de datos, por eso se usa la opción *--link=nombre-contenedor:alias-contenedor*. Al contenedor de PostgreSQL se le ha llamado *database*, por eso será el primer parámetro. El alias deberá coincidir con el que se escribió en el fichero de configuración de E-nefro.

No se ha especificado ningún comando al contenedor ni en el *docker run* ni en el *Dockerfile* anterior por lo que el contenedor debería morir al no tener ningún proceso que lo mantenga activo. No obstante no ocurrirá ya que en el *Dockerfile* se tomó como imagen base *tomcat:7.0.67-jre7*. Esta imagen a su vez también tiene un *Dockerfile* creado por los desarrolladores de Apache cuyo contenido está disponible en Github (53). Al final de él se observa la siguiente línea:

```
[…]
CMD ["catalina.sh", "run"]
```

Dockerfile

Código 8 - Extracto del Dockerfile de la imagen de Tomcat 7 oficial

Al no escribir *CMD* en el *Dockerfile* de la imagen *enefro-tomcat* el contenedor toma el de su imagen base, que es el script de arranque de Tomcat.

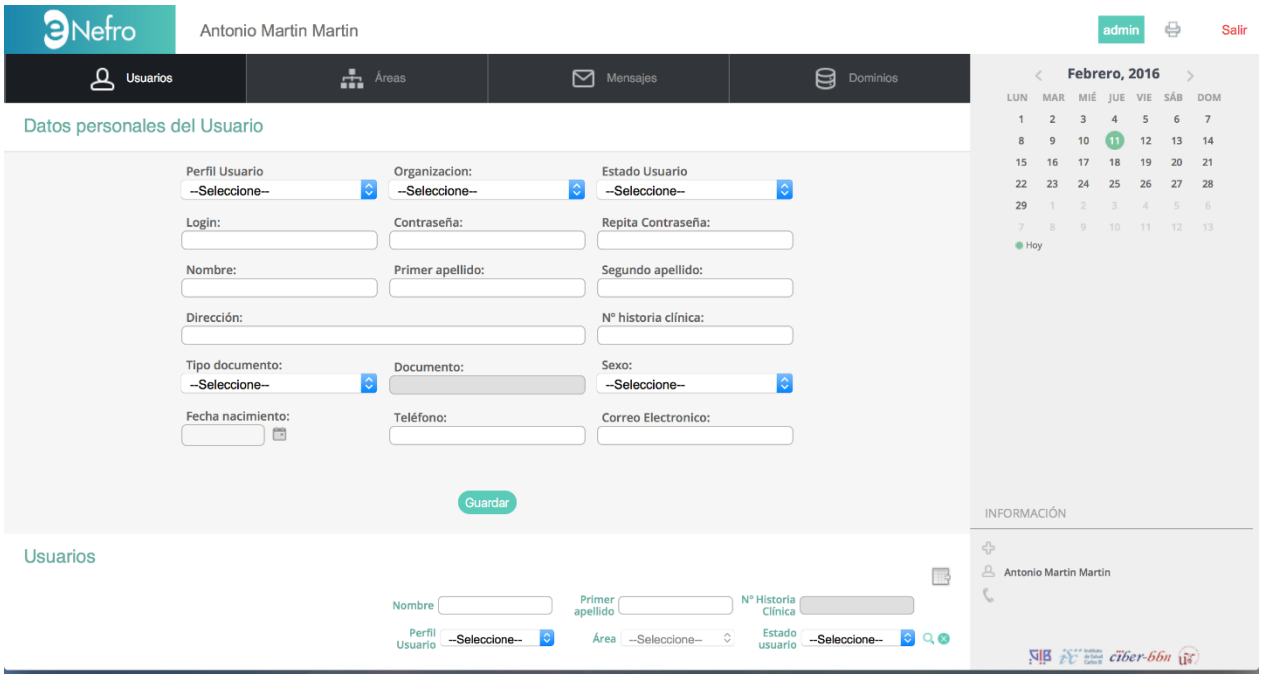
Lo mismo ocurre con el contenedor de PostgreSQL, en su *Dockerfile* accesible desde Github se puede ver el comando que lo mantiene activo (54).

Para comprobar si los contenedores están en ejecución se puede usar *docker ps*, que imprime una lista de los contenedores activos:

```
$ docker ps
CONTAINER ID     IMAGE          COMMAND     STATUS      PORTS         NAMES
8295a94816ca enefro-tomcat "catalina.sh run" Up 0.0.0.0:8080->8080/tcp
application
732m219ao2jf enefro-database "docker-entrypoint.sh" Up 0.0.0.0/tcp database
```

Entrando desde el navegador a http://dominio:8080/enefro se verá lo siguiente:

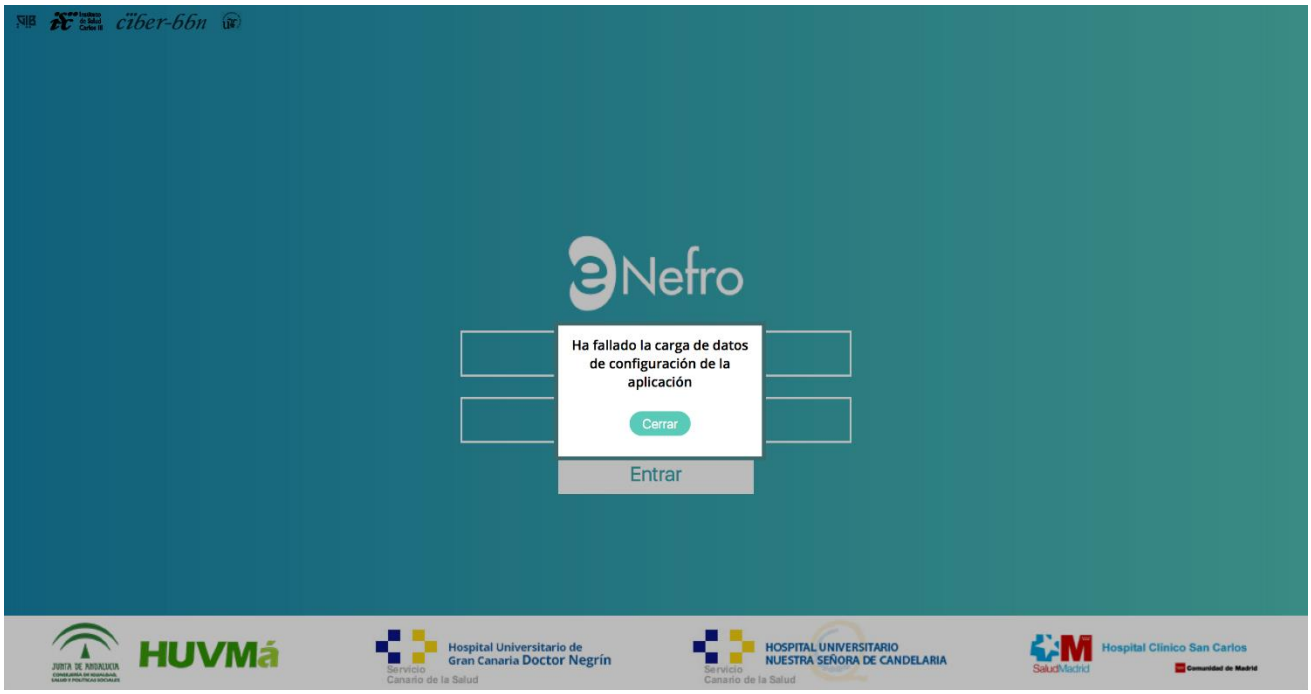
La aplicación funciona correctamente sobre contenedores. Se puede entrar con el usuario que se crea por defecto, *admin* y contraseña *admin10*.



Si el contenedor de la base de datos no estuviese activo la aplicación no dejaría entrar. Se puede probar parándolo con el siguiente comando:

```
$ docker stop database
```

Al acceder por el navegador se muestra un error:



Como el contenedor de Tomcat está vivo la aplicación se muestra por pantalla, sin embargo no puede establecer conexión con la base de datos mostrando el error de carga de datos.

Se puede volver a activar el contenedor de la base de datos con el comando *docker start*:

```
$ docker start database
```

Aunque la aplicación funcione los datos introducidos en PostgreSQL no persisten.

4.1.3 Persistencia de la información

Una de las principales características de un contenedor es su ciclo de vida. En cuanto este se destruye todo el sistema de ficheros en él se pierde y aunque es fácilmente reemplazable lanzando otro contenedor de la misma imagen la información no persistirá. Esta volatilidad se soluciona utilizando volúmenes y contenedores de datos, definidos en el capítulo 3.2.

La aplicación de E-nefro almacena la información persistente en la base de datos, por lo que es necesario que esta se guarde en otro sitio aunque el contenedor de PostgreSQL se destruya.

Se creará un contenedor que guarde la información de las tablas de la base de datos. Para saber dónde se guardan en el sistema de ficheros se ejecuta el comando *show data_directory* en la base de datos (55). Mediante *docker exec NOMBRE_CONTENEDOR comando* es posible ejecutar cualquier comando dentro de un contenedor*:*

```
$ docker exec -it database psql -U postgres -c "show data_directory”

      data_directory
--------------------------
 /var/lib/postgresql/data
(1 row)
```

Por lo tanto el volumen estará en */var/lib/postgresql/data*. El contenedor de datos se lanzará con un volumen en este directorio y el de la base de datos compartirá este volumen con la opción *--volumes-from*. El contenedor de datos tiene la única función de crear el volumen, que no se borrará aunque este contenedor o el de PostgreSQL se borren o reinicien. De esta manera la información siempre estará en el volumen, pase lo que le pase a los contenedores. La única manera de borrar la información sería con el comando *docker rm –v*.

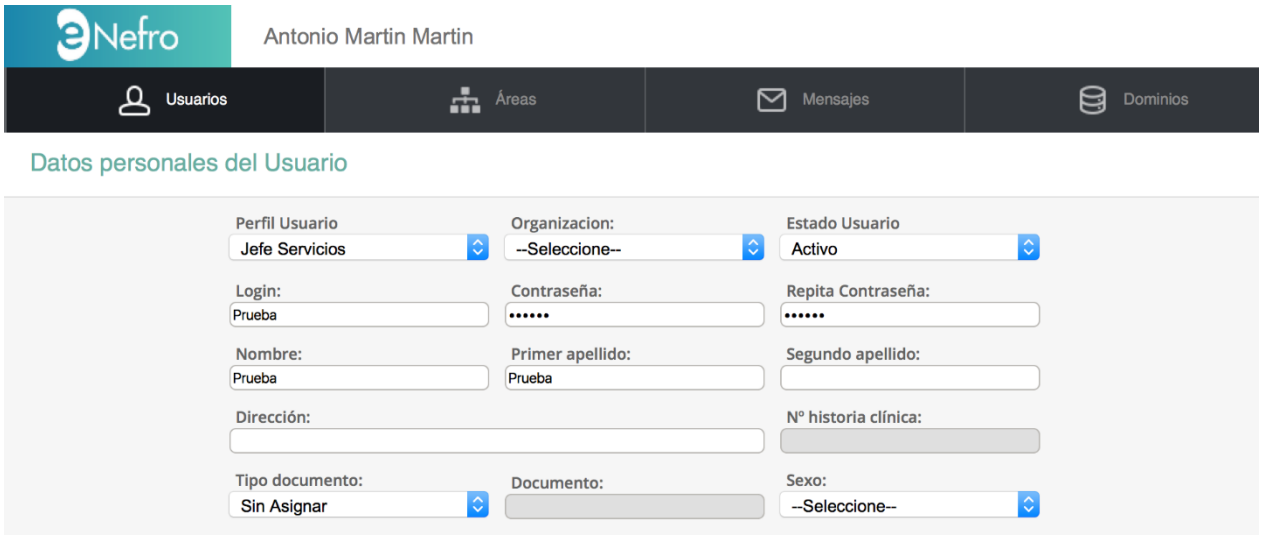
El comando para lanzar el contenedor de datos será:

```
$ docker run -—name=datos -v /var/lib/postgresql/data ubuntu:14.04 echo
"Contenedor de datos”
```

Ahora habrá que eliminar el contenedor de postgres y relanzarlo compartiendo el volumen con *–volumes-from*:

```
$ docker rm -f database
$ docker run --volumes-from=datos -d --name=database enefro-database
```

Se puede probar la persistencia de datos creando un usuario nuevo en la aplicación y reiniciando el contenedor.



El usuario se ha guardado correctamente:

A continuación se borra e inicia de nuevo el contenedor de la base de datos:

```
$ docker rm -f database
$ docker run -d --volumes-from=datos --name=database enefro-database
```

Se comprueba que el usuario continúa en la base de datos lanzando una consulta SQL a la tabla donde se encuentran los usuarios, *e_usuario*:

```
$ docker exec database psql -U postgres enefro_sevillarocio -c "select
usua_id, usua_codigo from e_usuario"
 usua_id | usua_codigo
---------+-------------
       1 | admin
       2 | enefro
       3 | Prueba
(3 rows)
```

4.1.4 Proxy

En este momento las peticiones a la aplicación van directas al contenedor de Tomcat por el puerto 8080. En esta sección se utilizará un contenedor con una aplicación llamada Nginx (56) que se encargará de procesar todo el tráfico entrante para después redirigirlo a Tomcat. Nginx es un servidor web similar a Apache aunque más ligero, lo que lo convierte en el favorito para ser usado como proxy inverso (57).

Usar un proxy inverso como intermediario del tráfico supone muchas ventajas:

- Se puede distribuir el tráfico entre varios contenedores idénticos de Tomcat corriendo simultáneamente para evitar sobrecarga y proveer un servicio elástico que adapte sus recursos. En la siguiente sección se habilitará este contenedor de Nginx para que funcione como balanceador de carga entre varios contenedores de Tomcat
- En cuanto a la seguridad, se evita tener el contenedor de Tomcat expuesto al exterior, pudiendo establecer un punto único de autenticación y autorización para todas las peticiones entrantes.
- El proxy puede servir contenido estático evitando carga innecesaria para la aplicación. Adicionalmente, puede funcionar como caché.

El nuevo sistema quedará como se muestra en la siguiente figura, con el contenedor de Nginx recibiendo todo el tráfico y redirigiéndolo entre varios contenedores de Tomcat:

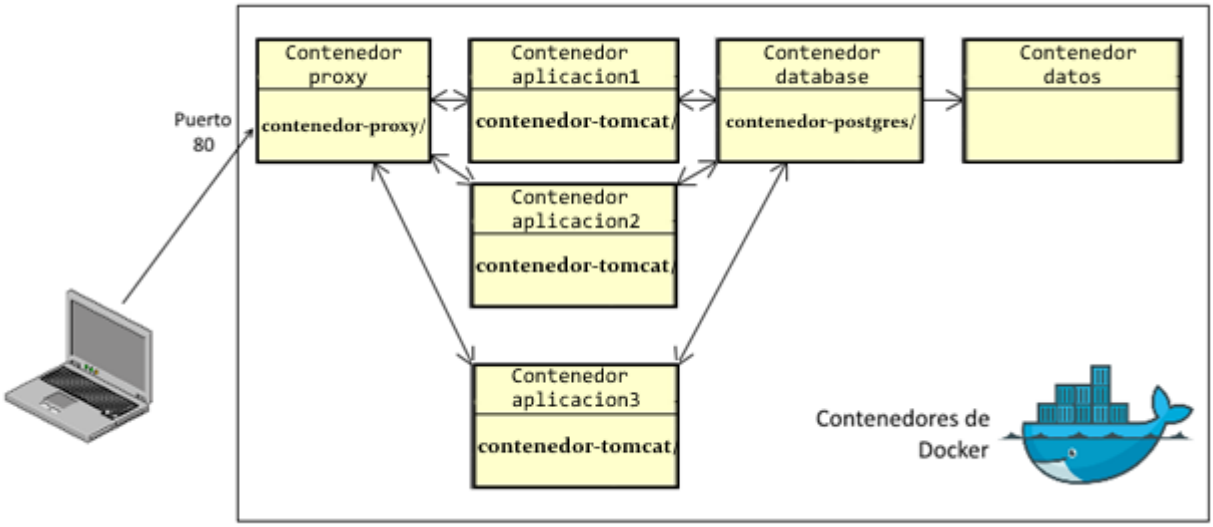


Ilustración 14 - Sistema utilizando Nginx como Proxy/Balanceador

Al igual que anteriormente con los contenedores de Tomcat y PostgreSQL, se usará la imagen oficial de Nginx y se adaptará a través de un *Dockerfile*. En la consola del servidor se crea la carpeta proxy con el *Dockerfile* y dentro la carpeta *conf* donde estarán los ficheros de configuración de Nginx:

```
$ mkdir contenedor-proxy
$ touch contenedor-proxy/Dockerfile
$ mkdir contenedor-proxy/conf
$ touch contenedor-proxy/enefro-nginx.conf
```

Nginx guarda sus ficheros de configuración en */etc/nginx/conf.d/*, que se llaman con un *include* desde */etc/nginx/nginx.conf*. La configuración para el proxy de este sistema se copiará en un fichero llamado *enefro-nginx.conf* que se redactará posteriormente.

La carpeta del contenedor del proxy tendrá el siguiente contenido:

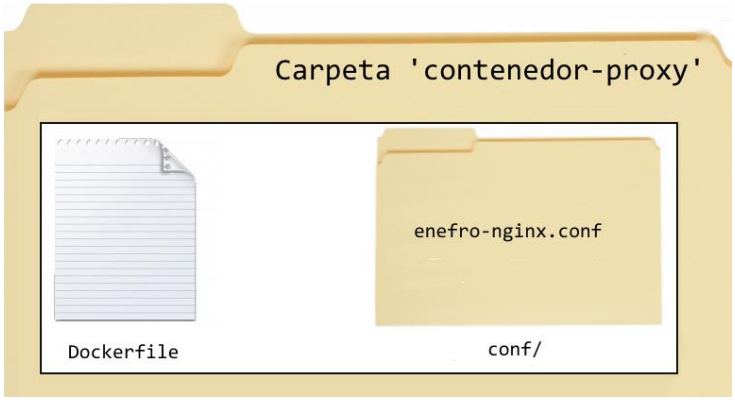


Ilustración 15 - Contenido de la carpeta 'contenedor-proxy'

El *Dockerfile* se muestra a continuación:

```
FROM nginx

RUN rm /etc/nginx/conf.d/default.conf
COPY conf/enefro-nginx.conf /etc/nginx/conf.d/enefro-nginx.conf
                                                              Dockerfile
```

Código 9 - Dockerfile del contenedor de Nginx

En la primera línea se establece como imagen base la oficial de Nginx (58)

```
FROM nginx
```

Observando la documentación, por defecto se coloca un fichero llamado *default.conf* que es necesario, por lo que se borrará y copiará *enefro-nginx.conf*:

```
RUN rm /etc/nginx/conf.d/default.conf

COPY conf/enefro-nginx.conf /etc/nginx/conf.d/enefro-nginx.conf
```

En el *Dockerfile* de la imagen base *nginx* se ve que la instrucción *CMD* arranca el contenedor con el servicio de Nginx en segundo plano: *CMD ["nginx", "-g", "daemon off;"]* (59).

En cuanto al fichero *enefro-nginx.conf*, Nginx ofrece muchísimas opciones en su funcionalidad como proxy inverso, detallado en su documentación oficial (60). El contenido con la configuración mínima es el siguiente:

```
server {
    listen       80;
    location / {
        proxy_pass         http://enefro;

        proxy_set_header   Host             $host;
        proxy_set_header   X-Real-IP        $remote_addr;
        proxy_set_header   X-Forwarded-For  $proxy_add_x_forwarded_for;
        proxy_set_header   X-Forwarded-Proto http;
    }
}
```

enefro-nginx.conf

Código 10 - Fichero de configuración de Nginx

En él se declara que se quieren escuchar las peticiones entrantes al puerto 80. Las peticiones que lleguen a / se pasan a http://enefro al especificarlo en *location* / y en *proxy_pass*. Este enefro es un alias que se definirá en la llamada a *docker run* del contenedor de Nginx con *--link aplicacion:enefro*, para que Docker sustituya bien la IP del contenedor de Tomcat.

Las *proxy header* son cabeceras necesarias para que el contenedor de E-nefro pueda interpretar correctamente las peticiones.

Una vez definidos el *Dockerfile* y el fichero de configuración de Nginx es posible construir la imagen del contenedor con *docker build*. En el comando se aclara el nombre con *–t* y la carpeta donde está el *Dockerfile*:

```
$ docker build -t enefro-proxy contenedor-proxy/
```

Tras unos segundos de espera, la imagen estará disponible para poder lanzarla. Se quiere acceder a través del puerto 80 por lo que en el comando de *docker run* se abrirá este puerto con *-p*. Además, se enlazará con el contenedor principal mediante *--link*:

```
$ docker run -p 80:80 --name=proxy --link aplicacion:enefro -d enefro-proxy
```

Ahora la aplicación estará accesible desde el puerto 80. Como ya no haría falta el acceso directo al contenedor de Tomcat que existía antes a través del 8080, se puede relanzar evitando exponer el puerto si se elimina la opción de *–p 8080:8080*:

```
$ docker rm -f aplicación
$ docker run -d ––link database:alias ––name=aplicacion enefro-tomcat
```

La aplicación sigue disponible en el puerto 80 ya que el proxy se encarga de redirigir las peticiones de manera interna.

4.1.5 Balanceador de carga

Nginx ofrece muchas posibilidades para balanceo de carga entre varios servidores.

En este nuevo escenario se van a lanzar varios contenedores de la imagen *enefro-tomcat*, todos enlazados con la misma base de datos y con diferentes nombres. Se probará con tres diferentes:

```
$ docker run -d ––link=database:alias ––name=aplicacion1 enefro-tomcat
$ docker run -d ––link=database:alias ––name=aplicacion2 enefro-tomcat
$ docker run -d ––link=database:alias ––name=aplicacion3 enefro-tomcat
```

En el fichero de configuración de Nginx de E-nefro llamado *enefro-nginx.conf* que se copió en la creación de la imagen se va a añadir la configuración necesaria para que este también balancee entre los tres contenedores de Tomcat. El fichero *enefro-nginx.conf* quedará así:

```
upstream enefroservers {
        server aplicacion1;
        server aplicacion2;
        server aplicacion3;
    }

server {
    listen       80;

    location / {
        proxy_pass         http://enefroservers;

        proxy_set_header   Host             $host;
        proxy_set_header   X-Real-IP        $remote_addr;
        proxy_set_header   X-Forwarded-For  $proxy_add_x_forwarded_for;
        proxy_set_header   X-Forwarded-Proto http;
    }
}
```

enefro-nginx.conf

Código 11 - Fichero de configuración de Nginx como balanceador de carga

Se ha añadido la directiva *upstream*, que declara varios servidores llamados como los alias de los contenedores de Tomcat para que Docker se encargue de traducirlo. En *proxy_pass* se declara el nombre del *upstream* anterior para que se balancee entre los servidores.

Al no definir ningún método de balanceo Nginx toma por defecto *round-robin*, el cual va rotando el tráfico por orden a cada petición que va llegando. Se pueden definir otros métodos como redirigir al contenedor con menos conexiones activas o usando IP *hash* (61).

Como se ha modificado el fichero de configuración, se reconstruye la imagen del proxy y se relanza enlazando con los tres contenedores

```
$ docker build -t enefro-proxy contenedor-proxy/
$ docker run -p 80:80 --name=proxy --link aplicacion1:enefro --link
aplicacion2:enefro –link aplicacion3:enefro -d enefro-proxy
```

4.1.6 Automatización multi-contenedor

Hasta ahora se ha trabajado con contenedores de manera individual, construyéndolos y lanzándolos uno a uno. Gracias a opciones como *--link* o los volúmenes se crean vínculos entre contenedores donde se comparten variables de entorno o la persistencia de datos es posible.

Sin embargo, existen herramientas más complejas como Docker Compose que permiten el manejo de varios contenedores a la vez con pocos comandos, configurando previamente los enlaces y volúmenes de cada contenedor.

Anteriormente se ha visto que manejar un sistema distribuido compuesto de cuatro contenedores puede causar más de un problema a la hora de desplegarlo, teniendo en cuenta además que el orden en el que los

vamos arrancando importa. Por ello, Docker Compose ofrece una manera fácil de administrar una aplicación multi-contenedor.

Si se quisiera reconstruir el sistema anterior se tendrían que ejecutar estos comandos:

1. Parar y eliminar los contenedores

```
$ docker rm -f aplicacion
$ docker rm -f database
$ docker rm -f proxy
```

2. Reconstruirlos

```
$ docker build -t enefro-database contenedor-postgres/
$ docker build -t enefro-tomcat contenedor-tomcat
$ docker build -t enefro-proxy contenedor-proxy/
```

3. Relanzarlos

```
$ docker run -d --volumes-from=datos --name=database enefro-database
$ docker run -d ––link database:alias ––name=aplicacion enefro-tomcat
$ docker run -p 80:80 --name=proxy --link aplicacion:enefro -d enefro-proxy
```

Mientras que con Docker Compose se ejecutan los siguientes comandos:

```
$ docker-compose stop
$ docker-compose build
$ docker-compose up -d
```

Para hacerlo funcionar se usa un fichero llamado *docker-compose.yml* donde se guarda la información de cada uno de los contenedores y el orden de ejecución. En la documentación oficial se encuentran todas las opciones posibles (62).

El fichero *docker-compose.yml* se crea en el directorio de trabajo usado durante el desarrollo de los apartados anteriores:

```
$ touch docker-compose.ml
```

Este sería el *docker-compose.yml* final:

```
enefro:
  build: tomcat
  links:
    - db
dbdata:
  image: ubuntu:14.04
  volumes:
    - /var/lib/postgresql/data
  command: "echo Datos"
db:
  build: postgres
  volumes_from:
    - dbdata
proxy:
  build: proxy
  links:
    - enefro:enefro-tomcat
  ports:
    - "80:80"
```

docker-compose.yml

Código 12 - Fichero de configuración de Compose del sistema

Se irá completando la configuración para cada contenedor:

1. El contenedor de Tomcat y la aplicación E-nefro:

```
aplicacion:
  build: contenedor-enefro
  links:
    - database
```

Es necesario recordar que se había creado una carpeta *contenedor-enefro* donde se escribía un *Dockerfile* definiendo al contenedor. Con estas líneas se crea un contenedor llamado *aplicacion* que se va a construir a partir del *Dockerfile* contenido en la carpeta *contenedor-enefro*. Además, como se hacía anteriormente con la opción *--link*, va a enlazarlo con el contenedor definido a continuación llamado *database*.

No se especifica ningún puerto ya que se entra a la aplicación a través del puerto 80 del proxy de Nginx.

2. Contenedor de la base de datos:

```
database:
  build: contenedor-database
  volumes_from:
    - datos
```

Al igual que antes, con la directiva *build* va a proceder a construir el contenedor definido por el *Dockerfile* contenido en el directorio *contenedor-database.* Por otra parte va a tomar los volúmenes del contenedor llamado datos.

3. Contenedor de datos:

```
datos:
  image: ubuntu:14.04
  volumes:
    - /var/lib/postgresql/data
  command: echo "Contenedor de datos"
```

Como este contenedor no se ha construido con un Dockerfile sino que utiliza una imagen del registro, se indica mediante *image* qué imagen base utilizará. Con *command* se especifica qué comando va a ejecutar el contenedor, un texto que imprime *Contenedor de datos*.

4. Proxy

```
proxy:
  build: contenedor-proxy
  links:
    - aplicacion
  ports:
    - "80:80"
```

Por último, se construye el proxy con el *Dockerfile* definido en la carpeta *contenedor-proxy*. Después se enlaza al contenedor principal y se abre el puerto 80 para que se pueda acceder a la aplicación.
Ya es posible ejecutar los dos comandos que lanzarán la aplicación multi-contenedor. Se comienza con *build*, que construirá los contenedores que tienen un *Dockerfile.*

```
$ docker-compose build
```

Una vez construidos, se pueden lanzar con el comando *up*. La opción *-d* hace que funcionen en segundo plano:

```
$ docker-compose up -d
Recreating contenedor_dbdata_1
Recreating contenedor_db_1
Recreating contenedor_enefro_1
Recreating contenedor_proxy_1
```

Se han ejecutado los contenedores en el orden perfecto. La aplicación estará disponible en http://IP-HOST, por el puerto 80 gracias al proxy de Nginx.

Con el comando *docker-compose ps* se imprimen los contenedores activos:

```
$ docker-compose ps
       Name                     Command               State
Ports
-------------------------------------------------------------------------------
--------------
contenedor_db_1       /docker-entrypoint.sh postgres   Up       5432/tcp
contenedor_dbdata_1   echo Contenedor de Datos         Exit 0
contenedor_enefro_1   catalina.sh run                  Up       8080/tcp
contenedor_proxy_1    nginx                            Up       443/tcp,
0.0.0.0:80->80/tcp
```

Cada contenedor está funcionando correctamente, excepto el de datos que está parado sirviendo como volumen persistente la información de la base de datos.

Si se quisiera editar algún *Dockerfile* o alguna configuración de algún contenedor, basta con reconstruir y lanzar con:

```
$ docker-compose build
$ docker-compose up -d
```

Para parar el sistema se usa el comando *docker-compose stop*:

```
$ docker-compose stop
```

4.2 Despliegue de E-nefro en la nube

En este apartado se va a tomar como base el entorno de contenedores del apartado anterior para desplegar la aplicación E-nefro en una máquina virtual en la nube de Azure.

Dentro de la instancia que se creará por medio de la herramienta de línea de comandos de Azure se lanzarán los contenedores. Para mostrar las posibilidades de la nube se modificarán los recursos destinados a esta instancia, se configurará el cortafuegos y se creará una imagen a partir de la máquina virtual con E-nefro funcionando.

Al contrario que en la sección anterior donde el cliente y el host de Docker estaban en el mismo servidor, ahora el host estará en la instancia de Azure mientras que el cliente estará en una máquina local con conexión a internet. El cliente necesitará tener instalada la herramienta de Azure (63) y el cliente de Docker, explicado en la sección 3. Además también deberá tener los ficheros de E-nefro y las carpetas de los contenedores creadas con sus correspondientes *Dockerfiles*.

En el siguiente esquema se muestra como a través del cliente se ejecutan comandos, tanto usando la *API* de Azure para crear la máquina virtual o editar los *endpoints*, como con el host de Docker dentro de la instancia para construir las imágenes y lanzar contenedores.

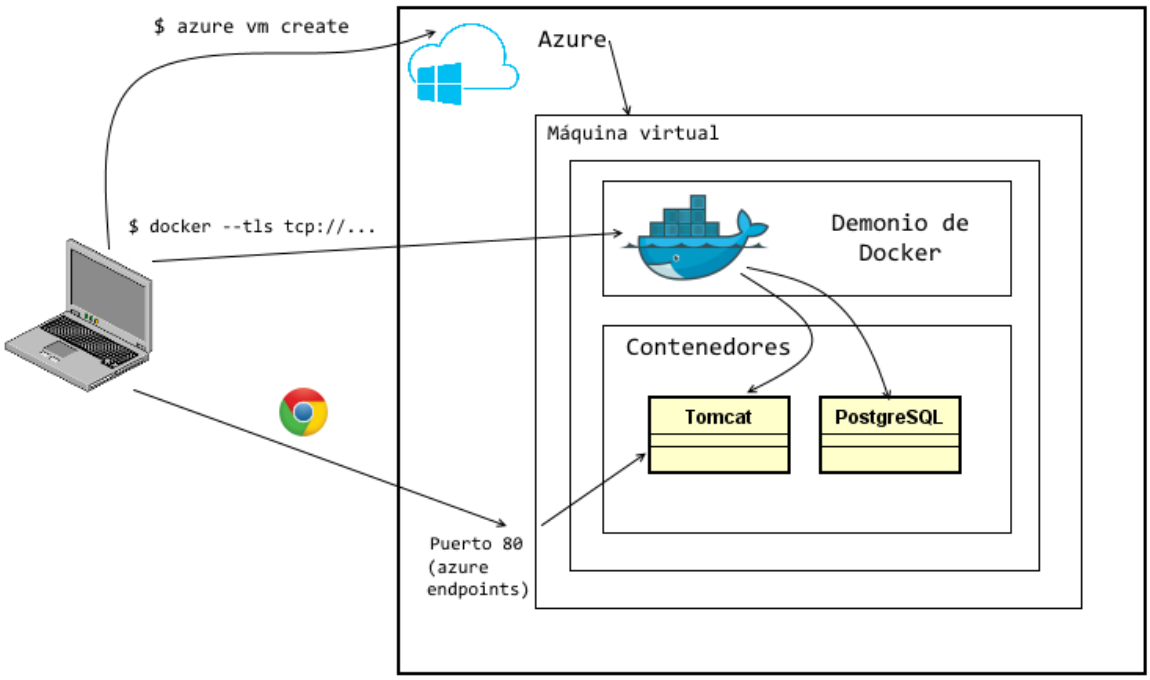


Ilustración 16 - Docker desplegado en Azure

En la consola de la máquina cliente será necesario introducir las credenciales de Azure con el comando *azure login*:

```
$ azure login
```

La aplicación E-nefro funcionará en una máquina virtual Ubuntu en la que esté instalado Docker. Los contenedores se lanzarán siguiendo el procedimiento del apartado anterior.

Azure tiene una imagen preparada para funcionar como un host de Docker, siendo capaz de alojar contenedores y de entender los comandos que ejecutemos desde un cliente remoto sin necesidad de conexión por SSH a él. Los comandos se mandarán a través de un nuevo puerto, por defecto 2376 (64).

Para lanzar un servidor hay que especificar la imagen, en este caso una de Ubuntu que pueda funcionar como host de Docker. Se pueden ver todas las imágenes ofrecidas por Azure con los siguientes comandos, el primero para ver toda la lista y el segundo filtrando por las imágenes de Ubuntu 14-04:

```
$ azure vm image list
$ azure vm image list | grep Ubuntu-14_04
```

La siguiente referencia es la de una imagen de Ubuntu 14.04:

```
b39f27a8b8c64d52b05eac6a62ebad85__Ubuntu-14_04_4-LTS-amd64-server-20160222-
en-us-30GB
```

Para que incluya el host de Docker al comando de creación de la máquina virtual se le añade la palabra *docker*. La herramienta en línea de comando de Azure tiene el comando *azure vm create*:

```
$ azure vm docker create -l "North Europe" nombre_instancia
"b39f27a8b8c64d52b05eac6a62ebad85__Ubuntu-14_04_4-LTS-amd64-server-20160222-
en-us-30GB" usuario password
```

Con la opción *-l* se indica la región, en *nombre_instancia* irá el identificador de la máquina y el usuario y contraseña serán los que se usen para acceder. Al escribir *docker* entre *vm* y *create* se incluye el host.

Una vez finalizado se puede comprobar que la instancia se ha creado correctamente ejecutando el comando *azure vm list*. El nombre de la instancia es *enefro* ya que el comando anterior se ha llamado sustituyendo *nombre_instancia* por *enefro*.

```
$ azure vm list
info:    Executing command vm list
+ Getting virtual machines
data:    Name    Status Location      DNS Name             IP Address
data:    enefro  Ready  North Europe  enefro.cloudapp.net  100.77.218.66
info:    vm list command OK
```

Con el comando *list* se ve la URL y dirección IP asignadas a la instancia.

El *plugin* de Docker que Azure ha incluido en la instancia ha hecho que se creen unas credenciales en el directorio /*.docker* del cliente. Esto permitirá la comunicación de comandos de Docker por medio del puerto 2376, asignado por defecto. Para mostrar la información del host de Docker se ejecuta el comando *docker info*. La opción *--tls* habilita una comunicación segura y *-H* es para especificar un host de Docker remoto, en este caso estará en la URL de la instancia lanzada y el puerto 2376.

```
$ docker --tls -H tcp://enefro.cloudapp.net:2376 info
```

Por tanto a partir de ahora se usará la siguiente estructura para ejecutar comandos de Docker

```
$ docker --tls -H tcp://enefro.cloudapp.net:2376 COMANDO-DOCKER
```

El procedimiento de despliegue es similar al seguido en el capítulo 4. Sin embargo Docker Compose no es compatible con el uso remoto, por lo que hay que lanzar cada contenedor de forma individual.

Los siguientes pasos se ejecutan desde la consola de la máquina local cliente:

1. Construir las imágenes a partir de los *Dockerfile*.

```
$ docker --tls -H tcp://enefro.cloudapp.net:2376 build -t enefro-tomcat
contenedor-tomcat/
$ docker --tls -H tcp://enefro.cloudapp.net:2376 build -t enefro-postgres
contenedor-database
$ docker --tls -H tcp://enefro.cloudapp.net:2376 build -t enefro-proxy
contenedor-proxy
```

2. Lanzar los contenedores de las imágenes construidas indicando los enlaces y los puertos que se quieran abrir.

```
$ docker --tls -H tcp://enefro.cloudapp.net:2376 run -–name=datos -v
/var/lib/postgresql/data ubuntu:14.04 echo "Contenedor de datos”
$ docker --tls -H tcp://enefro.cloudapp.net:2376 run -d-volumes-from=datos –
name=database enefro-database
$ docker --tls -H tcp://enefro.cloudapp.net:2376 run -d --link database:db --
name=aplicacion enefro-tomcat
$ docker --tls -H tcp://enefro.cloudapp.net:2376 run -p 80:80 --name=proxy --
link aplicacion:enefro -d enefro-proxy
```

Con *docker ps* se comprueba que los contenedores están corriendo:

```
$ docker --tls -H tcp://enefro.cloudapp.net:2376 ps
CONTAINER ID        IMAGE               COMMAND                  CREATED
STATUS              PORTS                          NAMES
b0fc11dc3c36        enefro-tomcat       "catalina.sh run"        15 minutes
ago      Up 15 minutes       8080/tcp                       aplicacion
1deb3b7a8333        enefro-proxy        "nginx"                  18 minutes
ago      Up 18 minutes       0.0.0.0:80->80/tcp, 443/tcp   proxy
05ca74c21ecd        enefro-database     "/docker-entrypoint.s"   21 minutes
ago      Up 21 minutes       5432/tcp                       database
```

Sin embargo cuando se intenta acceder a través del navegador a la URL del servidor se recibe un error 500. Esto es debido a que como medida de seguridad el corta fuegos de Azure cierra todos los puertos por defecto a sus instancias.

Se pueden ver los puertos abiertos de una instancia con el comando *azure vm endpoint list nombre_instancia*:

```
$ azure vm endpoint list enefro
info:    Executing command vm endpoint list
+ Getting virtual machines
data:    Name       Protocol  Public Port  Private Port  Virtual IP
data:    ---------  --------  -----------  ------------  ------------
data:    docker     tcp       2376         2376          40.77.57.118
```

En el servidor de Ubuntu lanzado solo está abierto el puerto 2376 de Docker. Si se quiere acceder por medio del navegador a la aplicación hay que crear un *endpoint* en el puerto 80. El puerto 80 de la máquina se traduce por otro exterior de manera similar a NAT. En este caso se quiere abrir el puerto 80 interior al 80 exterior:

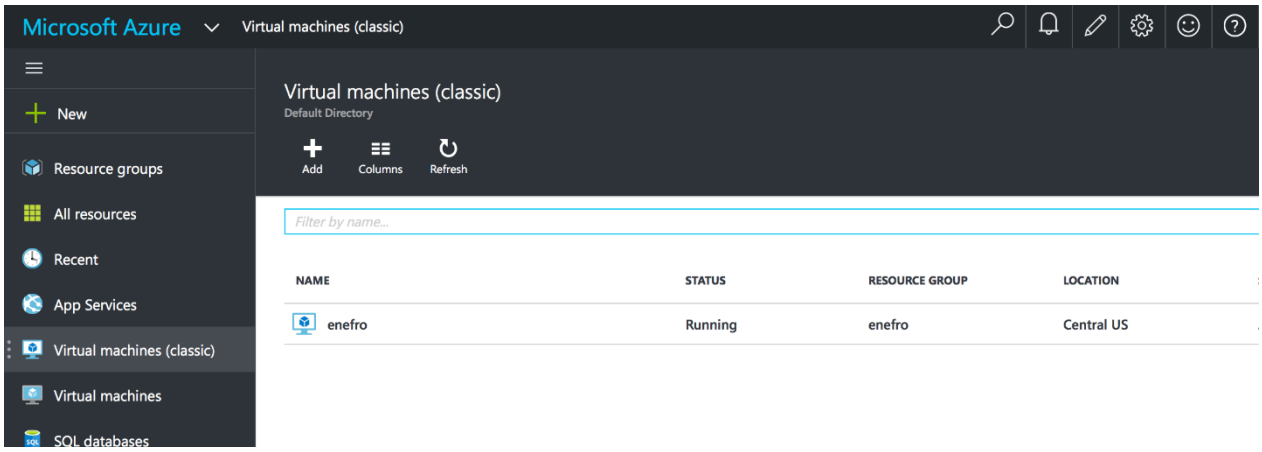
```
$ azure vm endpoint create enefro 80 80
```

Tras ejecutar este comando ya se tiene acceso a la aplicación desde el navegador.

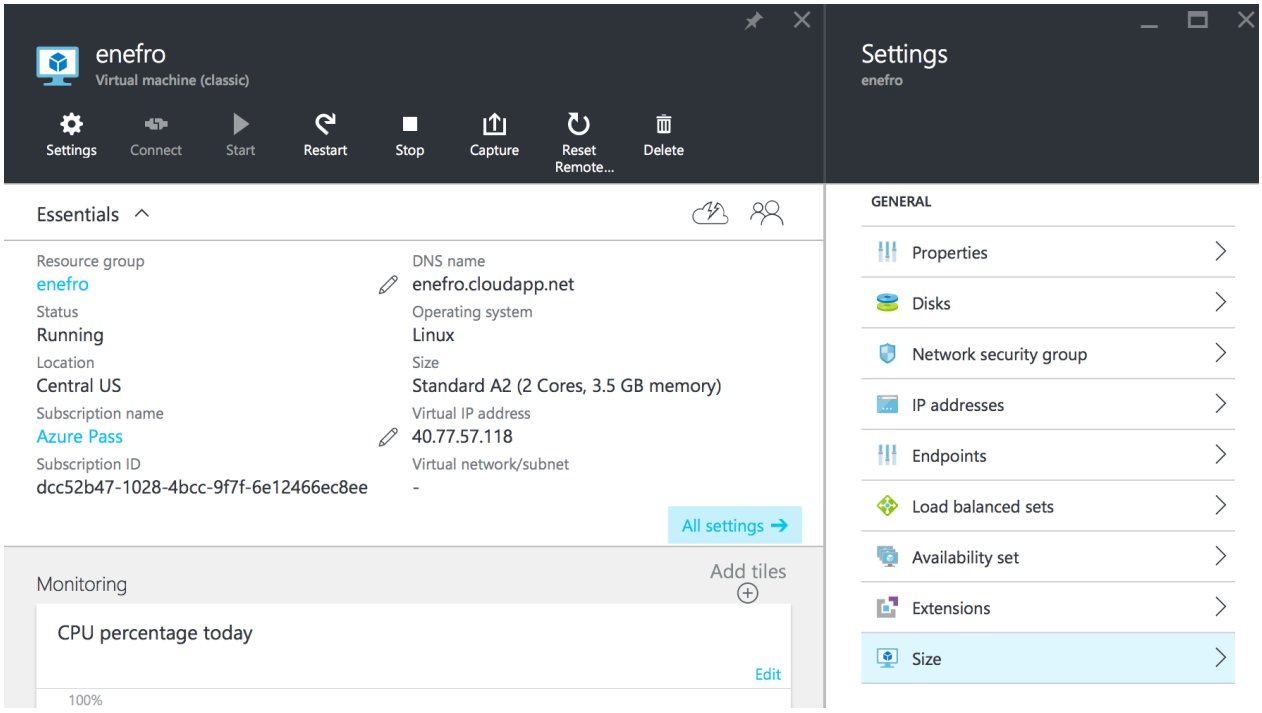
El tipo de instancia que se ha creado por defecto es una A1 Standard, que tiene un núcleo y 1.75 GB de memoria RAM. Azure proporciona herramientas de autoescalado del servidor, lo que permitirá que las especificaciones de CPU y memoria se adapten en tiempo real en función del tráfico.

También es posible fijar el tipo de instancia de forma manual sin necesidad de detener la instancia. La herramienta de línea de comando de Azure no permite modificar los recursos de una máquina virtual, sin embargo sí se puede hacer desde el panel de administración web accesible desde https://portal.azure.com

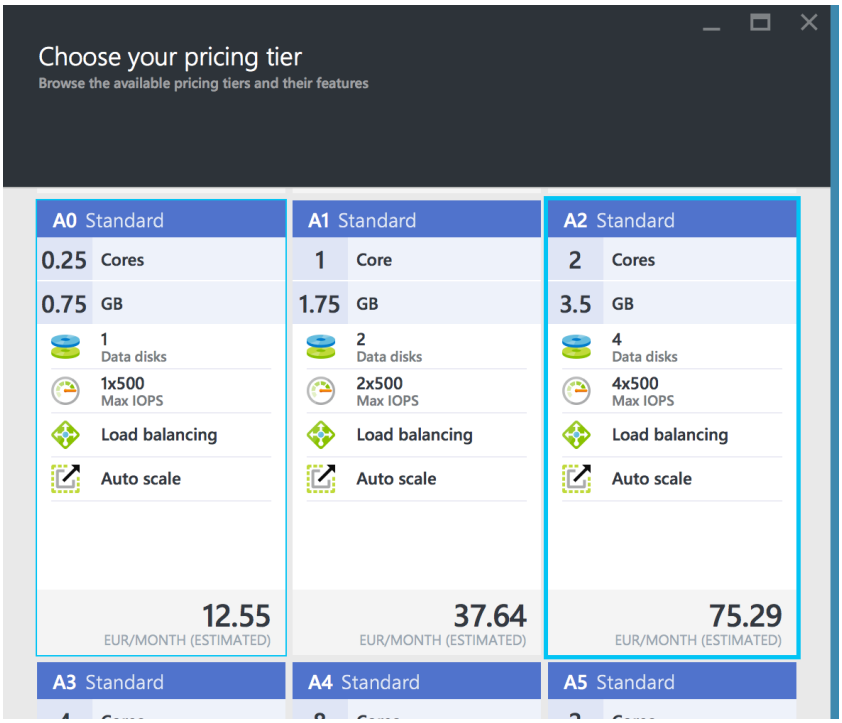
Una vez autenticados en el panel de Azure, se accede a la sección de máquinas virtuales:



Haciendo click sobre la instancia, se desplegará un menú con información sobre el servidor.



En *Settings->Size*, se listarán todos los tamaños de máquina virtual disponibles. Seleccionando cualquier otro tipo de instancia las especificaciones del servidor cambiarán pasados unos minutos y sin necesidad de reiniciar.



Ahora que el servidor está desplegado con la configuración correcta sería conveniente crear una imagen de él para poder replicarlo en una nueva instancia. Esta operación es posible desde la interfaz web o la línea de comando. Haciéndolo desde esta última, primero se para la instancia con *azure vm shutdown nombre_instancia* y después capturando la imagen con *azure vm capture nombre_instancia nombre_imagen*:

```
$ azure vm shutdown enefro
$ azure vm capture enefro enefro-docker-image
```

Pasados unos minutos la imagen estará disponible en el registro de la cuenta:

```
$ azure vm image list | grep enefro
data:    enefro-docker-image        User      Linux    undefined
```

Como al principio, usando el comando de creación de instancias se puede lanzar un servidor utilizando esta imagen en lugar de la de Ubuntu original:

```
$ azure vm docker create -l "North Europe" nombre_instancia "enefro-docker-
image" usuario password
```

La instancia contiene todas las imágenes de los contenedores pero no estarán en ejecución por defecto, al igual que los *endpoints*. Habría que ejecutar *docker run* para cada contenedor y volver a habilitar el *endpoint* en el puerto 80 para tener la aplicación funcionando de nuevo en esta réplica. Para facilitar esta tarea se puede crear un script que inicialice la instancia:

```
!/bin/bash

# Se lanzan todos los contenedores

docker --tls -H tcp://enefro.cloudapp.net:2376 run -–name=datos -v
/var/lib/postgresql/data ubuntu:14.04 echo "Contenedor de datos”
docker --tls -H tcp://enefro.cloudapp.net:2376 run -d-volumes-from=datos
–name=database enefro-database
docker --tls -H tcp://enefro.cloudapp.net:2376 run -d --link database:db
--name=aplicacion enefro-tomcat
docker --tls -H tcp://enefro.cloudapp.net:2376 run -p 80:80 --name=proxy
--link aplicacion:enefro -d enefro-proxy

# Se abre el Puerto 80
azure vm endpoint create enefro 80 80
```

init-enefro.sh

Código 13 - Script de inicialización de la imagen de E-nefro

4.3 Despliegue de E-nefro sobre Kubernetes

Hasta ahora se ha desplegado la aplicación E-nefro en un único servidor utilizando contenedores, primero de forma local. Posteriormente este entorno ha sido reproducido en un servidor en la nube, lo que aporta ventajas como la elasticidad al modificar los recursos de la máquina virtual. Esto implicaría una escalabilidad vertical, donde se añaden más recursos a un solo nodo.

En la última solución se desarrollará la escalabilidad horizontal, que consiste en añadir más instancias de los servicios al sistema con el fin de mejorar el rendimiento, disponibilidad y fiabilidad. Para ello se hará uso de la herramienta de orquestación de contenedores Kubernetes vista en la sección 3.4. Por medio de la replicación de pods se podrán reducir o aumentar en tiempo real el número de instancias de Tomcat o de PostgreSQL que se quieren en funcionamiento.

Inicialmente se comenzará con un escenario que no tendrá en cuenta la persistencia formado por un master y dos nodos. El segundo escenario sí tendrá en cuenta la persistencia pero se reducirá al master y a un único nodo. De esta manera durante el desarrollo se verán varias funcionalidades de Kubernetes, en la primera la distribución de las réplicas de pods entre los nodos y en la segunda el funcionamiento de los volúmenes.

La limitación en el primer escenario ocurre por la elección de Azure como servicio en la nube. Aunque está previsto para el futuro, Kubernetes no ha implementado aún ninguna forma de lanzar volúmenes en Azure (65) como sí ocurre con la nube de Google o Amazon. Otras alternativas de volúmenes son montar un servidor con tecnologías NFS, iSCSI o RBD, que se escapan del ámbito de este proyecto (66).

La solución que se tomará para persistir la información en Azure será usar un volumen de tipo *HostPath* (67), que monta un directorio del nodo dentro del pod. Por tanto si se persisten las tablas de la base de datos dentro de un directorio del nodo y este se monta cuando se inicializa un pod de PostgreSQL, se conservará la información. El inconveniente que acarrea esta implementación es que sólo se puede usar en un único nodo, teniendo que reducir el sistema de dos a uno.

El primer escenario sin persistencia consistirá en tres máquinas virtuales en Azure, una funcionando como master de Kubernetes y las otras dos como nodos. Desde el master se controlan todas las acciones ya que allí está la aplicación por línea de comandos cliente de Kubernetes llamada *kubectl*. A través de la API el master mandará instrucciones a los *kubelet* de los dos nodos, y estos crearán o destruirán los pods correspondientes. Todos los conceptos de Kubernetes están definidos en el capítulo 3.4.

En el siguiente esquema se muestra la infraestructura que se va a desplegar:

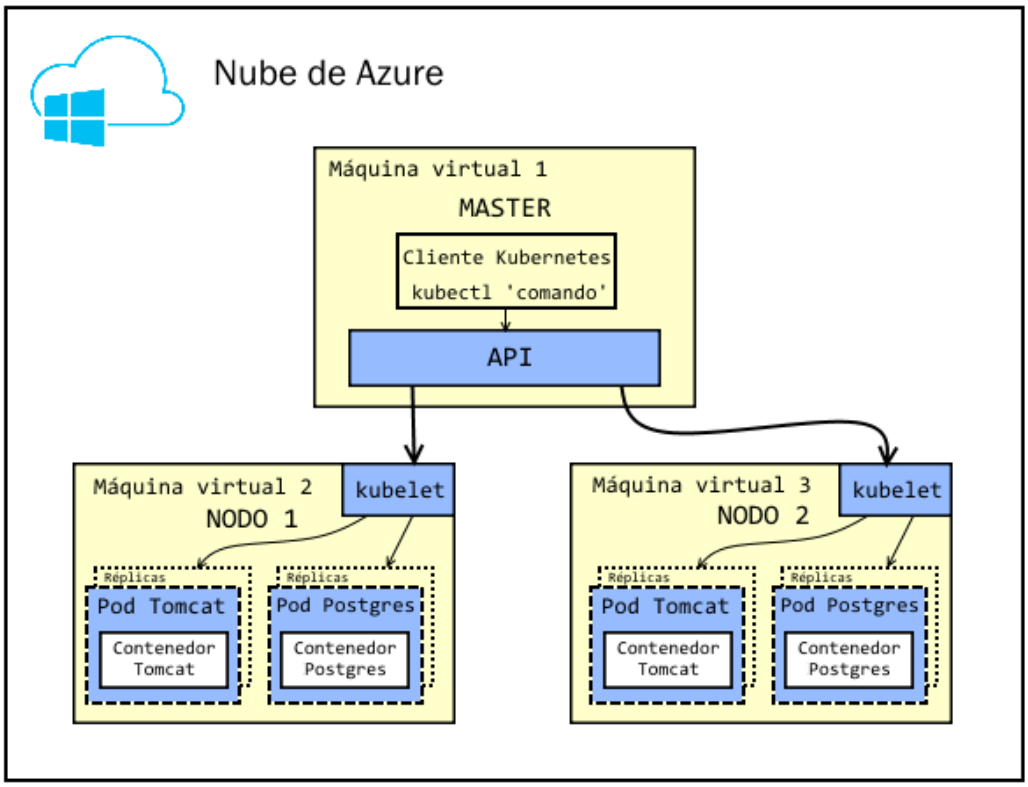


Ilustración 17 – Esquema primer escenario de E-nefro desplegado en Azure y Kubernetes

El sistema parte de los dos servicios de E-nefro: el servidor de aplicaciones Tomcat y la base de datos PostgreSQL. Para cada uno de ellos se creará un pod compuesto por su correspondiente contenedor. Haciendo uso de los *replication controller* se podrá modificar de forma dinámica cuántas réplicas se quieren de cada pod. El proxy definido en los apartados anteriores no será necesario ya que Kubernetes ofrece como funcionalidad el balanceo entre réplicas de pods.

Las réplicas de los pods están representados por cajas con línea discontinua ya que estarán distribuidos entre los dos nodos de manera indiferente.

El segundo escenario será el siguiente:

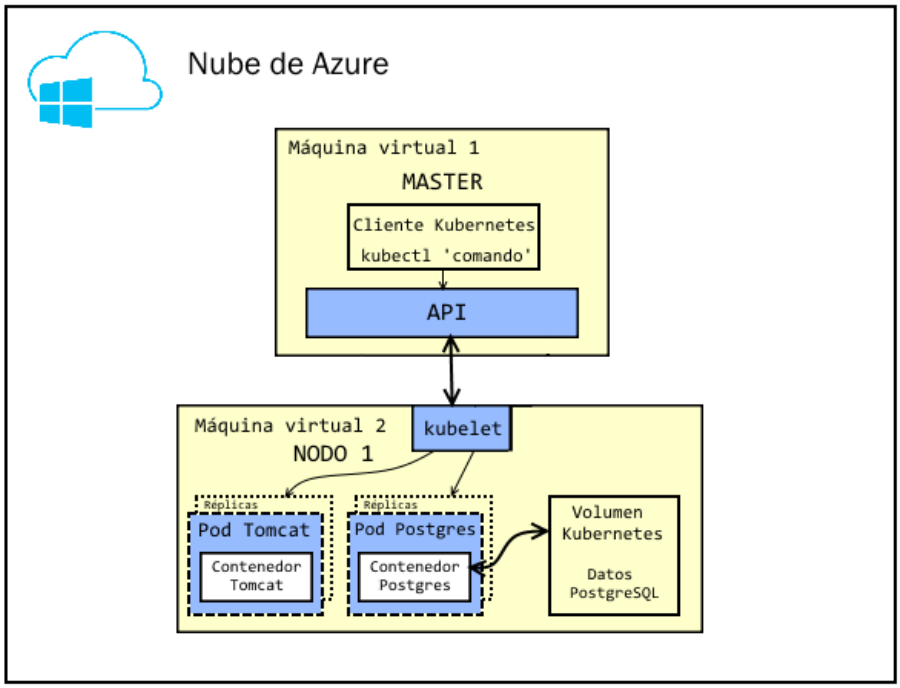


Ilustración 18 - Esquema segundo escenario de E-nefro desplegado en Azure y Kubernetes

El esquema es similar al anterior salvo que solo existe un nodo y se definirá un volumen dentro de la máquina virtual con la información de la base de datos.

Este apartado se organizará de la siguiente manera:

1. Primero se creará el entorno de máquinas virtuales en Azure compuesto por el master y los dos nodos.
2. Sobre esta infraestructura en Azure se crearán los pods, *replication controllers* y servicios a partir de sus plantillas. Esta implementación funcionará sobre dos nodos pero sin tener en cuenta la persistencia al no usar volúmenes.
3. Por último se tendrá en cuenta la persistencia añadiéndole el volumen y reduciendo el sistema a un nodo.

4.3.1 Creación del entorno de Kubernetes en Azure

Kubernetes incluye una guía para desplegar automáticamente en Azure un sistema de máquinas virtuales configuradas con todos los servicios necesarios. El primer paso es clonar el repositorio de Kubernetes desde la consola de una máquina local:

```
$ git clone https://github.com/kubernetes/kubernetes
$ cd kubernetes/docs/getting-started-guides/coreos/azure/
```

En este directorio hay un script llamado *create-kubernetes-cluster.js*. El contenido de este script está disponible en el repositorio de GitHub (68). Este va a lanzar varias instancias en Azure con los paquetes necesarios para que Kubernetes funcione y se pueda desplegar E-nefro. En total serán el master (llamado *kube-00*) y los dos nodos (*kube-01* y *kube-02*).

Se ejecuta el script:

```
$ ./create-kubernetes-cluster.js
```

En la siguiente figura se muestra el cluster que se ha creado.

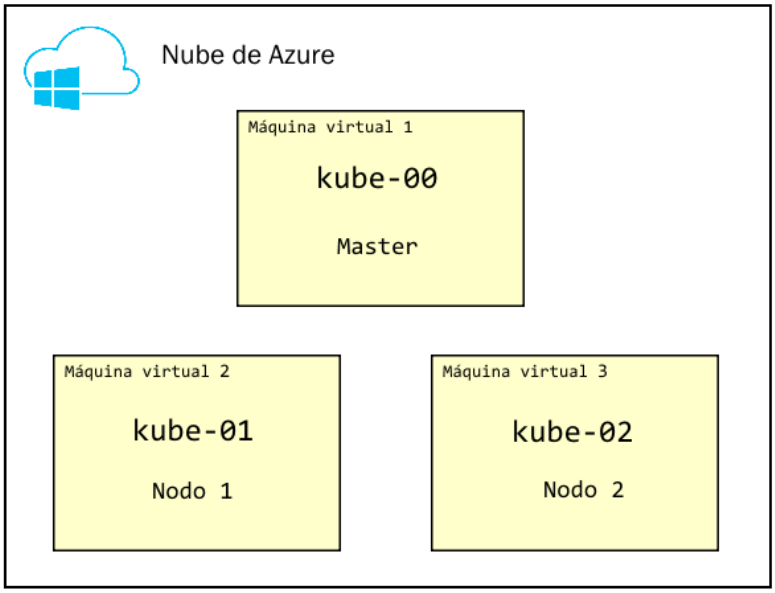


Ilustración 19 - Entorno de Kubernetes creado en Azure

Es posible establecer una conexión SSH con cualquier nodo o el master por medio del siguiente comando:

```
$ ssh -F ./output/kube_***_ssh_conf kube-0*
```

Sustituyendo las referencias que el script *create-kubernetes-cluster.js* ha creado.

4.3.2 Despliegue sin persistencia

El sistema va a estar formado por dos pods independientes, uno para Tomcat y otro para la base de datos sobre PostgreSQL. Se descarta el contenedor de proxy/balanceador de carga debido a que Kubernetes suple esta funcionalidad.

Los dos pods se definen a través de un *replication controller*, que permitirá escalar fácilmente los contenedores.

Además, será necesario definir un servicio por cada pod, el cual expone los puertos y permite acceso desde fuera a la aplicación.

Para comenzar el despliegue es necesario que los nodos *kube-01* y *kube-02* tengan las imágenes de los contenedores de Tomcat y PostgreSQL construidas. En este caso se envían las imágenes y se construyen dentro de los nodos aunque se podrían subir a algún registro para evitar tener que conectarse y hacer todo este proceso de manera manual. Primero se copian a los nodos los *Dockerfiles* y ficheros necesarios de los dos contenedores:

```
$ scp -F ./output/kube_***_ssh_conf ./enefro-tomcat.tar.gz kube-01:~
$ scp -F ./output/kube_***_ssh_conf ./enefro-database.tar.gz kube-01:~
```

Una vez copiados, se establece una conexión SSH para descomprimirlos y construir las imágenes:

```
$ ssh -F ./output/kube_***_ssh_conf kube-01
$ tar xzf enefro-tomcat.tar.gz
$ tar xzf enefro-database.tar.gz
$ docker build -t enefro-database contenedor-postgres/
```

```
$ docker build -t enefro-tomcat contenedor-tomcat/
```

Con los dos nodos listos, hay que definir los pods y servicios en el master por medio de plantillas. Una plantilla define los metadatos, etiquetas o especificaciones de un elemento de Kubernetes. Para E-nefro se hará una plantilla de los *replication controller*, donde también se define el pod y otra para los servicios.

Las plantillas utilizan el formato *yaml*, definiendo distintos aspectos del elemento. El yaml del *replication controller* del pod de tomcat será el siguiente:

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: tomcat
spec:
  replicas: 1
  selector:
    app: enefro
  template:
    metadata:
      labels:
        app: enefro
    spec:
      containers:
      - name: tomcat
        image: enefro-tomcat
        ports:
        - containerPort: 8080
```

rc-tomcat.yaml

Código 14 - Plantilla del Replication Controller de Tomcat

Primero se define el tipo de elemento con *kind*. A continuación en la metadata se nombra al contenedor como tomcat. En *spec* se especifica el número de réplicas del pod, es decir, el número de veces que se va a replicar. Se utiliza como selector la etiqueta *app: enefro*, que es necesario para enlazarlo con el servicio como se verá posteriormente. En *template* se define el pod, compuesto por un contenedor llamado tomcat con la imagen enefro-tomcat que se construyó antes en los nodos. Por último, el contenedor va a usar el puerto 8080.

Se despliega el pod de Tomcat desde el nodo master *kube-00* y usando *kubectl* para crear la plantilla anterior:

```
$ ssh -F ./output/kube_***_ssh_conf kube-00
$ kubectl create -f rc-tomcat.yaml
replicationcontroller "tomcat" created
```

Ahora mismo un pod con el contenedor de Tomcat comenzará a correr en el cluster. Se pueden listar los pods activos de la siguiente manera:

```
$ kubectl get pods -o wide
NAME           READY     STATUS    RESTARTS   AGE       NODE
tomcat-wc4vd   1/1       Running   0          1m        kube-02
```

Un pod está corriendo en el nodo *kube-02*. Gracias al servicio, el nodo en el que se despliegue el pod es completamente invisible para el usuario, de una forma similar a la que actúa un balanceador de carga entre varias instancias.

Sin la plantilla del servicio el pod no podrá ser accesible desde el exterior:

```
apiVersion: v1
kind: Service
metadata:
  name: tomcat
spec:
  ports:
  - port: 8080
    targetPort: 8080
    protocol: TCP
  selector:
    app: enefro
  type: NodePort
```

sv-tomcat.yaml

Código 15 - Plantilla del servicio de Tomcat

Con *kind* se especifica el tipo servicio, llamado en la metadata tomcat como el *replication controller*. En las especificaciones se necesita definir el mapeo de puertos 8080:8080 junto con el selector *app: enefro*. Por último, el tipo de servicio será de puerto *NodePort*. Esto quiere decir que cuando se cree el servicio, la aplicación estará expuesta al exterior en un puerto asignado aleatoriamente.

Se despliega el servicio indicándole la plantilla anterior:

```
$ kubectl create -f sv-enefro.yaml
You have exposed your service on an external port on all nodes in your
cluster.  If you want to expose this service to the external internet, you
may
need to set up firewall rules for the service port(s) (tcp:31838) to serve
traffic.
service "enefro" created
```

Se ha abierto la aplicación en el puerto 31838. Como sugiere el mensaje se debe abrir ese puerto en el corta fuegas de Azure, y para que la aplicación sea accesible desde el puerto 80 se redirige. Este comando se ejecuta desde la máquina local:

```
$ azure vm endpoint create kube-00 80 31838
```

Ahora se podrá acceder a la aplicación usando la URL que se ha asignado a la instancia kube-00. Esta URL se puede obtener mediante el siguiente comando en la máquina local:

```
$ azure vm show kube-00
info:    Executing command vm show
+ Getting virtual machines
data:    DNSName "kube-ea8a60491827be.cloudapp.net"
[…]
```

En el siguiente esquema se muestra como una conexión de un usuario pasa por el *endpoint* de Azure al servicio, que redirigirá al pod sin importar en qué nodo esté. En caso de que hubiera varias réplicas, se encargará del balanceo:

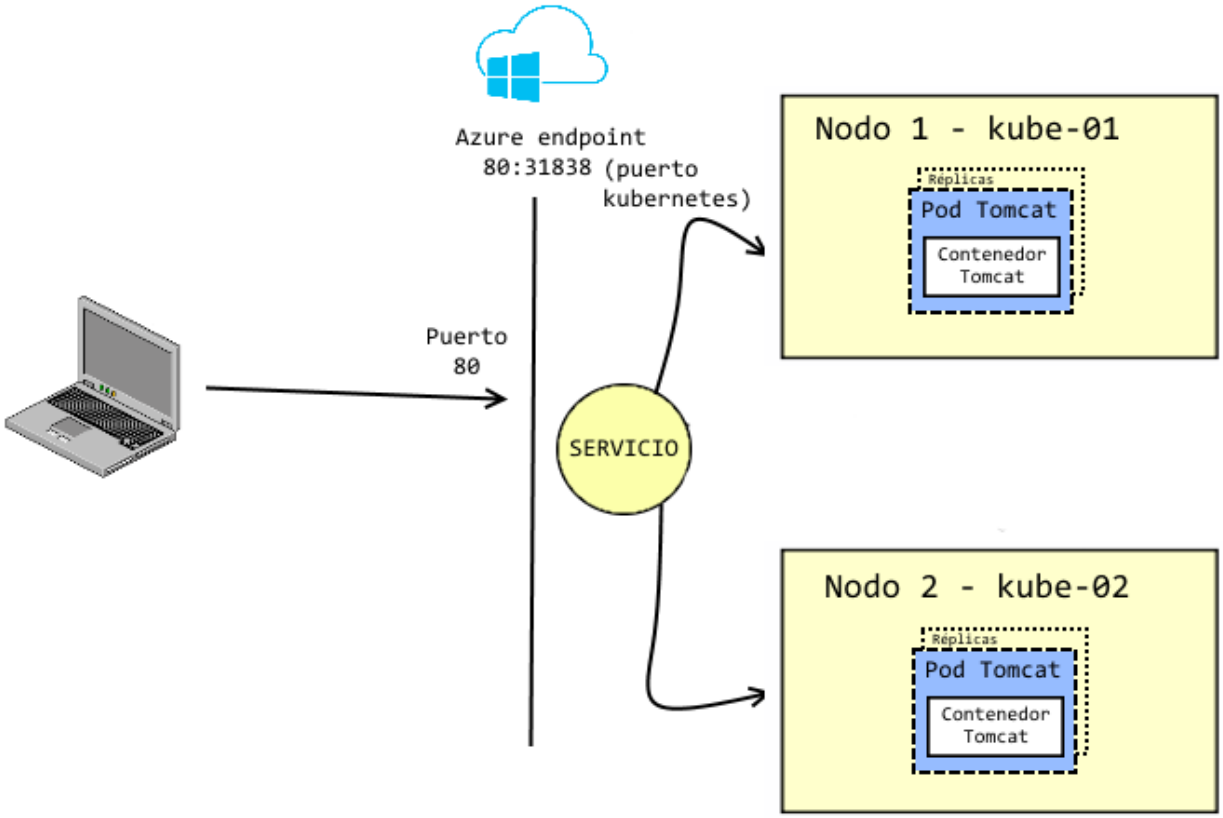


Ilustración 20 - Funcionamiento de Kubernetes en Azure

Accediendo a http://kube-ea8a60491827be.cloudapp.net/enefro estará la aplicación disponible, aunque mostrará el error de que no se puede conectar a la base de datos ya que aún no se ha desplegado su contenedor:

Para la base de datos se sigue el mismo procedimiento, definir las plantillas del *replication controller*-pod y del servicio para poder crearlas con *kubectl*.

La primera plantilla es similar a la anterior de Tomcat, tan solo haciendo referencia a la imagen de postgres construida con *docker build*.

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: postgres
spec:
  replicas: 1
  selector:
    name: postgres
  template:
    metadata:
      labels:
       name: postgres
    spec:
      containers:
      - name: postgres
        image: enefro-database
        ports:
        - containerPort: 5432
```

rc-database.yaml

Código 16 - Plantilla del Replication Controller de PostgreSQL

Se crea desde la consola del master *kube-00:*

```
$ kubectl create -f rc-database.yaml
replicationcontroller "postgres" created
```

El servicio no definirá ningún tipo ni puerto externo ya que la base de datos funcionará internamente en el cluster.

```
apiVersion: v1
kind: Service
metadata:
  name: database
spec:
  ports:
  - port: 5432
    protocol: TCP
  selector:
    name: postgres
```

sv-database.yaml

Código 17 - Plantilla para el servicio de PostgreSQL

```
$ kubectl create -f sv-database.yaml
service "database" created
```

Se comprueba el estado de los pods:

```
$ kubectl get pods -o wide
NAME             READY     STATUS    RESTARTS   AGE       NODE
```

```
tomcat-wc4vd     1/1       Running   0          10m       kube-02
postgres-c3eu8   1/1       Running   0          1m        kube-01
```

En la última columna se ve que el pod anterior de Tomcat se arrancó en el nodo 2 pero la base de datos se ha arrancado en el 1. Esto no va a afectar el funcionamiento de la aplicación ya que los servicios de Kubernetes se encargan de que las comunicaciones entre contenedores no tengan barreras entre nodos.

Si ahora se entra a la aplicación a través del navegador no aparecerá el error de la base de datos y se puede acceder sin problema, por lo que la base de datos funciona correctamente.

Una de las funcionalidades más útiles de Kubernetes es la facilidad de escalar una aplicación. Gracias a haber definido los pods como *replication controllers* se puede aumentar el número de réplicas con mucha facilidad. Para escalar el pod de Tomcat con cuatro réplicas se ejecuta el siguiente comando:

```
$ kubectl scale --replicas=4 rc tomcat
replicationcontroller "tomcat" scaled
$ kubectl get pods -o wide
NAME             READY     STATUS    RESTARTS   AGE       NODE
tomcat-17daq     1/1       Running   0          9s        kube-01
tomcat-dd3ol     1/1       Running   0          9s        kube-01
tomcat-j4raj     1/1       Running   0          9s        kube-02
tomcat-wc4vd     1/1       Running   0          19m       kube-02
postgres-c3eu8   1/1       Running   0          9m        kube-01
```

Con un solo comando se ha indicado pasar de una réplica a cuatro, y se han generado tres nuevas réplicas repartidas por los dos nodos. Si por cualquier razón una de ellas se detuviera, Kubernetes se encargará de lanzar una nueva para llegar al estado deseado de 4 réplicas corriendo a la vez.

Cuando se acceda a la aplicación por el navegador el servicio va a balancear a uno de los cuatro pods de manera completamente transparente.

Se puede hacer lo mismo con la base de datos:

```
$ kubectl scale --replicas=2 rc postgres
replicationcontroller "postgres" scaled
$ kubectl get pods -o wide
NAME             READY     STATUS    RESTARTS   AGE       NODE
tomcat-17daq     1/1       Running   0          4m        kube-01
tomcat-dd3ol     1/1       Running   0          4m        kube-01
tomcat-j4raj     1/1       Running   0          4m        kube-02
tomcat-wc4vd     1/1       Running   0          24m       kube-02
postgres-8vofk   1/1       Running   0          13s       kube-01
postgres-c3eu8   1/1       Running   0          13m       kube-01
```

De forma similar, se ha generado un nuevo pod de base de datos en el nodo 1. El servicio se encargará de balancear el tráfico entre los dos pods

4.3.3 Despliegue con persistencia

La persistencia de la información en la base de datos se hará utilizando un volumen externo al pod de PostgreSQL. Las soluciones de volúmenes que propone Kubernetes son múltiples, destacando las de lanzar un disco en la nube o montar un servidor NFS distribuido.

Por las limitaciones de Azure, finalmente la solución será crear un volumen local en uno de los nodos. En Kubernetes los volúmenes locales en un único nodo se denominan *HostPath*. Este volumen es para los pods que se lancen en este nodo, así que habrá que limitar el escenario a un único nodo. Se pierde la escalabilidad horizontal a nivel nodo para asegurar la persistencia.

Por tanto se eliminará el nodo *kube-02* quedando el master y el nodo *kube-01*. En este se creará un volumen permanente que aloja los datos de PostgreSQL y que se montará cada vez que se inicialice el pod. Si se lanzan nuevas réplicas, estas también montarán el volumen y compartirán los datos.

Para comenzar con este escenario hay que eliminar los pods y parar el nodo *kube-02*.

Desde la consola de la máquina local, se apaga la instancia *kube-02* usando el cliente de Azure:

```
$ azure vm shutdown kube-02
```

Desde la consola de la máquina master *kube-00* se pueden lanzar las órdenes con el cliente kubectl. Primero se establece la conexión SSH a la máquina:

```
$ ssh -F ./output/kube_***_ssh_conf kube-00
```

Y se eliminan los pods:

```
$ kubectl delete -f rc-enefro.yaml
$ kubectl delete -f rc-database.yaml
```

Para definir el volumen hay que utilizar los conceptos del capítulo 3.4 de volumen persistente y petición de volumen. Los pasos son los siguientes:

- Primero será necesario declarar el volumen y el directorio donde se montará dentro del pod utilizando la plantilla del replication controller.
- Después se creará una nueva plantilla de petición de volumen con el mismo nombre que la declarada en la plantilla anterior.
- Por último, se usará una plantilla de volumen donde se especifica que será un volumen local de tipo *HostPath* y donde se montará dentro del nodo.

En la plantilla del replication controller de PostgreSQL definida anteriormente se añadirá al final la información referente al volumen:

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: postgres
spec:
  replicas: 1
  selector:
    name: postgres
  template:
    metadata:
      labels:
       name: postgres
    spec:
      containers:
      - name: postgres
        image: enefro-database
        ports:
        - containerPort: 5432
        volumeMounts:
        - name: postgres-data
          mountPath: /var/lib/postgresql/data
      volumes:
      - name: postgres-data
        persistentVolumeClaim:
          claimName: postgres-volume-claim
```

rc-database.yaml

Código 18 - Plantilla del Replication Controller de PostgreSQL con un volumen

Dentro de la definición del contenedor se define un volumen que se montará en el directorio donde se almacena la información de PostgreSQL:

```
volumeMounts:
  - name: postgres-data
    mountPath: /var/lib/postgresql/data
```

Al final del fichero es necesario hacer una petición de volumen, con el nombre *peticion-volumen-postgres*:

```
volumes:
  - name: postgres-data
    persistentVolumeClaim:
      claimName: peticion-volumen-postgres
```

En un nuevo fichero llamado *pvc-database.yaml* se define la petición de volumen:

```
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: peticion-volumen-postgres
spec:
 resources:
    requests:
      storage: 1Gi
```

pvc-database.yaml

Código 19 - Plantilla de la petición de volumen de PostgreSQL

En esta plantilla se pide un volumen que tenga un giga de capacidad. Esta petición buscará entre los volúmenes definidos que cumplan las características pedidas, por lo que se tendrá que definir un volumen con 1GB. Esto se hará en la siguiente plantilla llamada *pv-database.yaml*.

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: volumen-postgres
spec:
  capacity:
    storage: 1Gi
  hostPath:
    path: /mnt/postgres
```

pv-database.yaml

Código 20 - Plantilla del volumen de PostgreSQL

El volumen tendrá 1GB y se montará en el directorio */mnt/postgres* del nodo *kube-01*.

Una vez escritas todas las plantillas se pueden crear:

```
$ kubectl create -f pv-database.yaml
$ kubectl create -f pvc-database.yaml
$ kubectl create -f rc-database.yaml
$ kubectl create -f rc-enefro.yaml
```

Tras unos segundos la aplicación estará disponible. Se puede comprobar que se está guardando la información en el volumen inspeccionando el directorio */mnt/postgres* del nodo *kube-01*. Para ello se ejecuta el comando *ls* en ese directorio:

```
$ ssh -F ./output/kube_***_ssh_conf kube-01 ls /mnt/postgres
base   pg_commit_ts  pg_ident.conf  pg_notify    pg_snapshots  pg_subtrans
PG_VERSION      postgresql.conf
global       pg_dynshmem   pg_logical     pg_replslot  pg_stat     pg_tblspc
pg_xlog               postmaster.opts
pg_clog  pg_hba.conf   pg_multixact   pg_serial    pg_stat_tmp    pg_twophase
postgresql.auto.conf  postmaster.pid
```

Se ha guardado correctamente la información de la base de datos. La aplicación ahora funciona con persistencia, y cuando se lancen nuevos pods porque se quiera escalar o por errores, conservarán la información guardada anteriormente.

5 CONCLUSIONES

En este proyecto se partía de una aplicación web desplegada en una máquina virtual tradicional con una distribución clásica de servidor de aplicación junto a una base de datos donde almacenar la información. Sobre esta aplicación se han diseñado y probado varios escenarios de despliegue basados en contenedores de Docker y en la infraestructura en la nube de Microsoft Azure.

Con estos escenarios se consiguen múltiples ventajas. Al separar cada servicio de la aplicación en un contenedor se favorece la administración individual de cada uno de ellos, permitiendo modificar los recursos destinados a nivel de servicio según la demanda necesaria. La volatilidad en la vida de los contenedores hace que sean fácilmente reemplazables si aparece un error, evitando que se arrastre en el resto del sistema. Cambiando el despliegue de una máquina virtual monolítica a uno en varios contenedores se consigue un sistema más ligero al tener tan solo las librerías y recursos que necesite cada servicio. Con contenedores se pueden añadir nuevas funcionalidades de manera sencilla como el contenedor de proxy inverso que al final de la sección 4.1 permitía el balanceo entre varios contenedores de Tomcat.

Cuando se traslada el sistema de contenedores a una instancia en la nube se delega el mantenimiento de la infraestructura. Por otra parte la instancia donde está alojada la aplicación se beneficia de la elasticidad que da la nube, pudiendo modificar sus recursos como CPU o memoria en tiempo real en función del tráfico. Creando una imagen cuando el sistema está en funcionamiento se puede clonar la aplicación en una nueva instancia.

Con una herramienta de orquestación como Kubernetes se aumenta el control sobre el sistema de contenedores. Este se encarga de monitorizar el estado de todos los servicios y mantener siempre la aplicación en funcionamiento. Cuando se despliega la aplicación en varias máquinas virtuales Kubernetes distribuye los contenedores entre ellas de manera transparente y si los servicios están duplicados replicados, se encarga de balancear.

Sin embargo durante el desarrollo de estos escenarios aparecen algunos problemas. El principal que aparece en un despliegue con contenedores es como persistir la información. Los contenedores están planeados para ser reemplazados sin afectar en absoluto el comportamiento de la aplicación por lo que supone un problema la pérdida de la información almacenada en la base de datos. Para ello se tomó un contenedor aparte encargado de persistir esta información.

La elección de Azure como proveedor ha implicado limitaciones debido al desfase respecto a sus competidores. La utilización de un disco de Azure como volumen de Kubernetes donde persistir la información no está disponible aun. La solución fue reducir el escenario a un único nodo donde albergar los contenedores y que estos pudieran guardar los datos en un volumen local llamado *HostPath*. Además, aunque Kubernetes ofrece unos *scripts* para el despliegue sencillo de un cluster sobre Azure, es complejo de adaptar a necesidades más específicas como aumentar el número de nodos.

Por otra parte, el despliegue en Azure se hizo utilizando el *plugin* de Docker en lugar de utilizar el servicio de contenedores. Esto es debido a que este servicio ha sido publicado después de que este proyecto se desarrollara (69).

Los distintos escenarios elaborados a lo largo de este proyecto presentan algunas líneas de mejora.
Respecto al despliegue con Kubernetes, una solución alternativa a la tomada y que podría mejorar las prestaciones del servicio es utilizar otra infraestructura en la nube como la de Google o Amazon. En ellas se pueden lanzar discos virtuales y usarlos como volúmenes donde persistir información de los contenedores. Otra posibilidad es desplegar un servidor NFS que albergue el volumen.

En un despliegue real el tamaño de las imágenes de Docker sería un tema importante a tener en cuenta. En las imágenes base utilizadas del repositorio oficial de Docker, como la de Tomcat o PostgreSQL, se incluyen muchas dependencias que no son necesarias para el propósito de este proyecto. Los contenedores deben ser lo más livianos posible, que solo contengan lo justo para aportar la funcionalidad esperada. En vez de confiar en sistemas operativos completos como cuando se usaban imágenes basadas en Ubuntu es mucho más ventajoso utilizar distribuciones de Linux pensadas para usos en contenedores como *CoreOS* o *Alpine Linux* (70) (71).

Otra línea de mejora es el despliegue de la aplicación usando más servicios clásicos de la nube, como el de un servidor de base de datos SQL. De esta forma se podría eliminar el contenedor de PostgreSQL, cargar los datos en una instancia de base de datos de Azure (72) y configurar la aplicación para que se integre con ella.

Esta última mejora conduce a las denominadas multi-máquinas virtuales. De manera similar a cómo funcionan los contenedores, pero utilizando máquinas virtuales tradicionales, los proveedores de nube están ofreciendo recientemente el despliegue de aplicaciones separando cada servicio en una instancia diferente. Como ocurre con las demás funcionalidades y novedades cada proveedor lo denomina de una forma diferente. Azure lo implementa por medio de plantillas y un administrador de recursos (73).

El mundo de la nube y los contenedores es un terreno de continuas innovaciones que incluso las grandes compañías muchas veces no saben predecir y es muy fácil quedarse desactualizado en cuestión de meses. Este es un gran momento ya que actualmente se están asentando las bases de la tecnología que va a utilizar cualquier desarrollador, sea grande o pequeño, durante los próximos años.

REFERENCIAS

1. Windows containers overview. [Online] https://msdn.microsoft.com/en-us/virtualization/windowscontainers/about/about_overview.

2. Linux containers isolation. [Online] https://blog.engineyard.com/2015/linux-containers-isolation.

3. Chroot concepts. [Online] https://help.ubuntu.com/community/BasicChroot.

4. Linux Vserver overview. [Online] http://linux-vserver.org/Overview.

5. FreeBSD Jails. [Online] https://www.freebsd.org/doc/handbook/jails.html.

6. cgroups. [Online] https://www.kernel.org/doc/Documentation/cgroup-v1/cgroups.txt.

7. Namespace manual. [Online] http://man7.org/linux/man-pages/man7/namespaces.7.html.

8. LXC Containers. [Online] http://www.ibm.com/developerworks/library/l-lxc-containers/.

9. LXC 1.0 security features. [Online] https://www.stgraber.org/2014/01/01/lxc-1-0-security-features/.

10. LXC Security. [Online] https://linuxcontainers.org/lxc/security/.

11. Let Me Contain That For You. [Online] https://github.com/google/lmctfy.

12. Docker 0.9: Introducing execution drivers and libcontainer. [Online] https://blog.docker.com/2014/03/docker-0-9-introducing-execution-drivers-and-libcontainer/.

13. Anuncio de LMCTFY. [Online] https://github.com/google/lmctfy/commit/0b317d7eb625d1877a8a0aaf2f46f770d9a5a50f.

14. CoreOS Rocket. [Online] https://github.com/coreos/rkt.

15. *Entrevista al fundador de CoreOS.* [Online] https://www.linux.com/news/featured-blogs/200-libby-clark/806347-collaboration-summit-keynote-alex-polvi-coreos.

16. Repositorio de APPC. [Online] https://github.com/appc/spec.

17. Docker adquiere Orchard y Fig. [Online] https://blog.docker.com/2014/07/welcoming-the-orchard-and-fig-team/.

18. Understanding modern service discovery with Docker. [Online] http://progrium.com/blog/2014/07/29/understanding-modern-service-discovery-with-docker/.

19. Consul vs Zookeeper. [Online] https://www.consul.io/intro/vs/zookeeper.html.

20. Open-Source Service Discovery. [Online] http://jasonwilder.com/blog/2014/02/04/service-discovery-in-the-cloud/.

21. Large-scale cluster management at Google with Borg. [Online] http://research.google.com/pubs/pub43438.html.

22. What is Kubernetes? [Online] http://kubernetes.io/docs/whatisk8s/.

23. Kubernetes getting started. [Online] http://kubernetes.io/docs/getting-started-guides.

24. **Carr, Nicholas.** *The Big Switch: Rewiring the World, from Edison to Google.* s.l. : W.W. Norton & Company, 2014.

25. ITU-T Study Group 13. [Online] http://www.itu.int/en/ITU-T/about/groups/Pages/sg13.aspx.

26. Y.3500 : Cloud Computing overview and vocabulary. [Online] ITU-T. https://www.itu.int/rec/T-REC-Y.3500-201408-I.

27. Recomendaciones del grupo de estudio 13 de ITU-T. [Online] http://www.itu.int/ITU-T/recommendations/index_sg.aspx?sg=13.

28. EC2 Origins. [Online] http://blog.b3k.us/2009/01/25/ec2-origins.html.

29. Docker pricing. [Online] https://www.docker.com/pricing.

30. Amazon cloud container service announcement. [Online] https://aws.amazon.com/es/blogs/aws/cloud-container-management/.

31. Google Container Engine. [Online] https://cloud.google.com/container-engine/docs/.

32. ACS Pricing. [Online] https://azure.microsoft.com/en-us/pricing/details/container-service/.

33. Pets vs Cattle. [Online] https://blog.engineyard.com/2014/pets-vs-cattle.

34. *Docker Architecture.* [Online] https://docs.docker.com/engine/understanding-docker/.

35. Hello world on Docker. [Online] https://docs.docker.com/engine/userguide/containers/dockerizing/.

36. Docker Hub. [Online] https://hub.docker.com/.

37. Docker Volumes. [Online] https://docs.docker.com/engine/tutorials/dockervolumes/.

38. What is Kubernetes? [Online] http://kubernetes.io/docs/whatisk8s/.

39. Kubernetes documentation. [Online] http://kubernetes.io/docs/.

40. Kubernetes cluster deployment solutions. [Online] http://kubernetes.io/docs/getting-started-guides/.

41. *etcd overview.* [Online] https://coreos.com/etcd/.

42. *Introduction to DHT.* [Online] https://www.ietf.org/proceedings/65/slides/plenaryt-2.pdf.

43. *Kubernetes Node definition.* [Online] http://kubernetes.io/v1.1/docs/admin/node.html.

44. Kubernetes Pod definition. [Online] http://kubernetes.io/v1.1/docs/user-guide/pods.html.

45. *Kubernetes Replication Controller definition.* [Online] http://kubernetes.io/v1.1/docs/user-guide/replication-controller.html.

46. *Kubernetes Service definition.* [Online] http://kubernetes.io/v1.1/docs/user-guide/services.htm.

47. Kubernetes Persistent Volumes. [Online] http://kubernetes.io/docs/user-guide/persistent-volumes/.

48. Kubernetes GCE Disk Volume. [Online] http://kubernetes.io/docs/user-guide/volumes/#gcepersistentdisk.

49. Kubernetes HostPath Volume. [Online] http://kubernetes.io/docs/user-guide/volumes/#hostpath.

50. PostgreSQL official Docker image. [Online] https://hub.docker.com/_/postgres/.

51. Connections and Authentification on PostgreSQL. [Online] https://www.postgresql.org/docs/9.1/static/runtime-config-connection.html.

52. Tomcat official Docker container. [Online] https://hub.docker.com/_/tomcat/.

53. *Tomcat 7 Dockerfile.* [Online] https://github.com/docker-library/tomcat/blob/master/7/jre7/Dockerfile.

54. PostgreSQL docker github. [Online] https://github.com/docker-library/postgres/tree/04b1d366d51a942b88fff6c62943f92c7c38d9b6/9.5.

55. PostgreSQL File locations. [Online] http://www.postgresql.org/docs/9.2/static/runtime-config-file-locations.html.

56. *Nginx official website.* [Online] http://nginx.org/.

57. What is a Reverse Proxy? Nginx. [Online] https://www.nginx.com/resources/glossary/reverse-proxy-server/.

58. Nginx on Docker Hub. [Online] https://hub.docker.com/_/nginx/.

59. *Nginx Dockerfile.* [Online] https://github.com/nginxinc/docker-nginx/blob/11fc019b2be3ad51ba5d097b1857a099c4056213/mainline/jessie/Dockerfile.

60. *Nginx documentation.* [Online] http://nginx.org/en/docs/.

61. *Nginx Load Balancing.* [Online] http://nginx.org/en/docs/http/load_balancing.html.

62. Docker Compose documentation. [Online] https://docs.docker.com/compose/compose-file/.

63. Guía de instalación de la herramienta de línea de comandos de Azure. [Online] https://azure.microsoft.com/en-us/documentation/articles/xplat-cli-install/.

64. *Azure Docker VM Extension.* [Online] https://azure.microsoft.com/en-us/documentation/articles/virtual-machines-linux-classic-cli-use-docker/.

65. Kubernetes Azure volume issue. [Online] https://github.com/kubernetes/kubernetes/issues/23259.

66. *Kubernetes Types of volumes.* [Online] http://kubernetes.io/docs/user-guide/persistent-volumes/#types-of-persistent-volumes.

67. Kubernetes HostPath. [Online] http://kubernetes.io/docs/user-guide/volumes/#hostpath.

68. Create a Kubernetes Cluster on Azure. [Online] https://github.com/kubernetes/kubernetes/blob/master/docs/getting-started-guides/coreos/azure/create-kubernetes-cluster.js.

69. Azure Container Service announcement. [Online] https://azure.microsoft.com/en-us/blog/azure-container-service-is-now-generally-available/.

70. CoreOS. [Online] https://coreos.com/.

71. Alpine Linux. [Online] https://www.alpinelinux.org/.

72. Azure SQL Server. [Online] https://azure.microsoft.com/es-es/services/sql-database/.

73. Azure MultiVM Templates. [Online] https://azure.microsoft.com/es-es/documentation/templates.

Otras referencias no citadas

- http://rhelblog.redhat.com/2015/08/28/the-history-of-containers/
- http://pivotal.io/platform/infographic/moments-in-container-history
- https://www.linux.com/news/featured-blogs/200-libby-clark/806347-collaboration-summit-keynote-alex-polvi-coreos
- http://cloudtweaks.com/2015/03/docker-vs-rocket-container-technology/
- https://www.flockport.com/lxc-vs-docker/
- https://coreos.com/rkt/docs/0.5.5/app-container.html
- https://www.ibm.com/cloud-computing/what-is-cloud-computing
- https://d36cz9buwru1tt.cloudfront.net/AWS_Overview.pdf
- http://blog.b3k.us/2009/01/25/ec2-origins.html
- *Docker: Up & Running*, Karl Matthias y Sean P. Kane, O'Reilly (2015)
- *Build Your Own PaaS with Docker*, Oskar Hane, Packt Publishing (2015)
- https://docs.docker.com/compose/gettingstarted/
- https://docs.docker.com/engine/userguide/
- https://docs.docker.com/engine/reference/builder
- http://engineering.riotgames.com/news/docker-jenkins-data-persists
- http://engineering.riotgames.com/news/jenkins-docker-proxies-and-compose
- https://azure.microsoft.com/es-es/documentation/articles/virtual-machines-windows-classic-setup-endpoints/
- https://azure.microsoft.com/en-us/documentation/articles/virtual-machines-linux-capture-image/
- http://kubernetes.io/docs/user-guide/walkthrough/
- http://kubernetes.io/docs/getting-started-guides/coreos/azure/
- https://github.com/kubernetes/kubernetes/tree/release-1.2/examples/mysql-wordpress-pd/
- *Kubernetes Microservices with Docker*, Deepak Vohra, Apress (2016)