Universitat Oberta de Catalunya

Orquestación de aplicaciones corporativas mediante virtualización de contenedores usando Docker

José Angel Fernández Ameijeiras
Grado de ingeniería informática

José Manuel Castillo Pedrosa

04/01/2016

FICHA DEL TRABAJO FINAL

Título del trabajo:Orquestación de aplicaciones corporativas mediante virtualización de contenedores usando Docker	Orquestación de aplicaciones corporativas mediante virtualización de contenedores usando Docker
Nombre del autor:	José Angel Fernández Ameijeiras
Nombre del consultor:	José Manuel Castillo Pedrosa
Fecha de entrega (mm/aaaa):	01/2017
Área del Trabajo Final:	Administración de redes y so
Titulación:	Grado de ingeniería informática

Resumen del Trabajo (máximo 250 palabras):

En este trabajo se realizará un estudio de las tecnologías de virtualización de contenedores con el fin de implementar y poner en marcha un sistema que permita orquestar el despliegue de aplicaciones sobre un entorno empresarial.

Para ello, se realizará en primera instancia un análisis de los sistemas de virtualización más habituales para continuar introduciendo los conceptos y sistemas de virtualización de contenedores. Una vez introducida la parte teórica se analizan distintas herramientas de virtualización de contenedores centrándonos en la herramienta Docker para la cual se detalla su arquitectura, funcionamiento y proceso de instalación para finalizar con un par de ejemplos prácticos de despliegue de servicios.

A continuación, una vez que ya hemos implementado y analizado un sistema de virtualización de contenedores como tecnología necesaria de base, pasamos a estudiar distintas soluciones del mercado para implementar un sistemas de orquestación basado en microservicios para el despliegue de aplicaciones de carácter corporativo. Finalizamos con la implantación, instalación y puesta en marcha del sistema estudiado acompañado de unos ejemplos de orquestación usando dos aplicaciones de código abierto que se ven bastante habitualmente en los entornos corporativos actuales para dar soporte a distintas soluciones.

Abstract (in English, 250 words or less):

In this job we do a study of container virtualization technologies will be carried out in order to implement a system that allows orchestrating the deployment of applications and services in enterprise environments.
This will be done in the first instance an analysis of the most common virtualization systems to continue introducing concepts and systems of containerization virtualization. Once the theoretical part is introduced, different container virtualization tools are analyzed, focusing on the Docker tool, which details its architecture, operation and installation process to finish with a couple of practical examples of service deployment.
Then, once we have already implemented and analyzed a container virtualization system and all necessary background technology, we started to study different solutions in the market to implement a micro-services based orchestration systems for the deployment of corporate applications.

Then finish with timplementation, installation and first steps of the studied system accompanied by some examples of orchestration using two open source applications that are quite commonly used in the current corporate environments to support different solutions.

Palabras clave (entre 4 y 8):

Virtualización
Contenedor
Orquestación
Despliegue
Microservicios
Docker

Índice

Lista de figuras

1. Introducción

1.1. Contexto y justificación del Trabajo

1.1.1. Descripción del proyecto

Actualmente, estamos asistiendo a un momento de cambio continuado en los escenarios que describen los marcos de las infraestructuras de TI, cualquiera que sea el tamaño y ámbito (público o privado) de la empresa. Aspectos como la ubicuidad de los sistemas, el procesado de grandes volúmenes de datos para su análisis o BigData[1], el Internet de las cosas o IoT[2], y los paradigmas de la virtualización de sistemas a través del cloud computing, han propiciado que aquellas infraestructuras de TI más tradicionales a las que estábamos acostumbrados no puedan ser capaces de adaptarse a estos nuevos escenarios, cada vez más exigentes, que se nos plantean.
La necesidad cada vez de mayor rapidez en el despliegue de aplicaciones y servicios, la reducción de costes, la optimización de los recursos en las soluciones de sistemas de TI en las empresas, etc. obligan al uso constante de las tecnologías más punteras en virtualización. Sin este tipo de herramientas las empresas actuales dejan de ser competitivas, debido a las situaciones de mercado tan cambiantes y volátiles que experimentamos, y las convierten en imprescindibles para enfrentarse a todos los retos a los que se exponen estas organizaciones.
La virtualización de servidores permite, entre otros, el ahorro de costes en los despliegues de servicios informáticos (tanto en el hardware como en su mantenimiento), así como una inmediatez y alta disponibilidad de los servicios desplegados. De este modo la virtualización ha permitido construir servidores y servicios tolerantes a fallos, escalables, ágiles, fáciles de mantener, y todo ello con ahorro de costes. Esto hace que las empresas en general, y más particularmente las empresas cuyos productos o servicios están más orientados al mundo digital, o empresas de base más tecnológica, necesiten un uso continuo de las tecnologías más punteras para poder conseguir una correcta alineación de su estrategia de negocio con el ámbito de TI.
Podemos afirmar que existe un antes y un después en el ámbito de las infraestructuras de TI marcado por la aparición de las tecnologías de virtualización pero hoy en día, fruto del avance continuo y de necesidades de un volumen de despliegues y servicios de forma imparable, la tecnología de virtualización clásica ya se ve superada en múltiples áreas igual que sucedió en un pasado reciente con el despliegue de infraestructuras de TI más clásicas y anteriores a la virtualización.
La tecnología que ha llegado para complementar y completar la virtualización de servidores es la virtualización de contenedores de aplicaciones. Esta tecnología va un paso mas allá en el paradigma de la virtualización, permitiendo no sólo el salto de virtualizar servidores sino también de virtualizar directamente un contenedor donde se ejecuta una aplicación, permitiendo de este modo una mayor abstracción aislando la componente "lógica de la aplicación" del componente “sistema operativo”.
De este modo, la virtualización de contenedores da un paso más allá permitiendo de un modo más sencillo y asequible proveer soluciones y servicios más completos ofreciendo una solución para dar el salto a soluciones del tipo IaaS[3] (Infrastructure as a Service), PaaS[4] (Platform as a Service) y SaaS [5](Software as a Service). Estamos en un momento en que la feroz competencia de los mercados hace que los clientes demanden soluciones tecnológicas en continuo cambio y

1 https://es.wikipedia.org/wiki/Big_data
2 https://es.wikipedia.org/wiki/Internet_de_las_cosas
3 http://www.gartner.com/it-glossary/infrastructure-as-a-service-iaas/
4 http://www.gartner.com/it-glossary/platform-as-a-service-paas/
5 http://www.gartner.com/it-glossary/software-as-a-service-saas/

con tiempos de implantación mínimos de modo que los proveedores que las ofrecen deben de adaptarse a este modelo de la mejor forma posible. Claramente este tipo de soluciones son las que permiten seguir este tipo de modelo de implantación de servicios, lo que si duda otorga una ventaja competitiva clara.

Una de las herramientas de virtualización de contenedores de código abierto que más auge ha experimentado es Docker[6]. Esta tecnología de contenedores promete cambiar la forma en la que las empresas gestionan los servicios y operaciones de TI, tal y como introducimos que hizo la tecnología de virtualización años atrás en el área de las infraestructuras de TI.
Mediante estas herramientas llegamos al objetivo final que es el proveer una arquitectura base de TI para dar soporte a la orquestación de servicios corporativos. La orquestación es un modelo de despliegue de aplicaciones que se basa en la interacción de los distintos servicios necesarios para dar un soporte funcional completo a una aplicación que necesita distintos componentes en su arquitectura funcional (por ejemplo una aplicación que requiere de un servidor web, una base de datos, un servidor de correo, etc). Así, de este modo la Orquestación permite alinear la solicitud de negocios con las aplicaciones, los datos y la infraestructura de TI

En este proyecto se va a realizar un análisis de las tecnologías de virtualización de contenedores, profundizando en la solución para proveer un sistema de orquestación de contenedores para el despliegue de aplicaciones corporativas, usando la herramienta de código abierto Docker, que es la herramienta que mayor auge ha experimentado a nivel de usuario y corporaciones, llegando a ser actualmente un estándar de referencia en el mercado. En primer lugar realizaremos una introducción sobre la situación actual de las tecnologías de virtualización tradicionales, así como una breve explicación de en qué consiste la tecnología de virtualización de contenedores y sus diferencias. Continuaremos con una comparativa de distintas tecnologías de contenedores que podemos encontrar en el mercado y realizaremos también una introducción a los conceptos de orquestación, definiendo sus requisitos y bases de funcionamiento así como las ventajas que ofrece a las organizaciones.
Proseguiremos con un análisis en profundidad de la tecnología subyacente que hace posible el funcionamiento de la virtualización de contenedores, para terminar introduciéndonos en el análisis de funcionamiento de la herramienta Docker.
Una vez estudiada la tecnología de Docker procederemos a realizar el desarrollo de un sistema de virtualización con esta tecnología, profundizando y detallando la puesta en marcha de todos los componentes.
Finalmente realizaremos una batería de ejemplos prácticos de orquestación de aplicaciones corporativas que podemos encontrar habitualmente en el mercado de las herramientas libres, como puede ser un CMS como el conocido Wordpress (que es claro ejemplo de aplicaciones que suelen encontrarse implantadas en organizaciones de distinta tipología ya que tiene una gran cuota de mercado).

1.1.2. Justificación del proyecto

Actualmente, todas las empresas usan en mayor o menor medida aplicaciones y servicios que deben su funcionamiento a los entornos de virtualización de servidores tipo cloud. Estos entornos han permitido el acceso de pequeñas y medianas empresas a las últimas tecnologías de virtualización a un coste sumamente ajustado, permitiendo de una forma casi natural la transición de los servicios informáticos más clásicos, a la modalidad cloud, o más orientadas al pago por uso. En el caso de las empresas de mayor tamaño o de las multinacionales, le ha permitido una mayor consolidación de su posición. Todo ello no habría sido posible sin la tecnología de virtualización de servidores, que en un pasado reciente fue considerada como el vehículo que

6 http://www.docker.com/

permitió el salto a las aplicaciones como servicio (SaaS) y a la tecnología asociada a la computación en la nube.
Hoy en día está emergiendo la tecnología de virtualización de contenedores incluso con más fuerza y con más auge del que tuvo en el pasado, probablemente porque pese a que al principio fue cuestionada y generó una gran controversia, hoy en día ya nadie cuestiona las bondades de la computación en la nube.

Si nos fijamos en la estela de las grandes compañías (Google, Amazon, Microsoft, etc.) veremos que actualmente todas las empresas TIC que fueron en su momento pioneras en la tecnología de virtualización, están introduciendo modelos de negocio basados en la virtualización de contenedores, señalando a esta como uno de los sistemas que estarán llamados a realizar una segunda revolución tecnológica en este área. Podemos ver proyectos como Kubernetes de Google, la posibilidad de ejecutar contenedores de aplicación Docker directamente a través de servidores de AWS (Amazon Web Services), o el soporte nativo de virtualización de contenedores en próximas versiones del buque insignia de los sistemas operativos de Microsoft “Windows Server”, como alguno de los casos más destacados.

A través del estudio realizado en este proyecto, cualquiera que desee introducir en su infraestructura de TI un sistema de orquestación de aplicaciones a través del uso de contenedores, podrá obtener un análisis claro de las tendencias y herramientas existentes en el mercado, así como un estudio completo de viabilidad y una base para la implantación de una de las herramientas de software libre que actualmente más cuota de mercado está ganando en la virtualización de contenedores, como es Docker.

1.1.3. Motivación para realizar el proyecto

En el pasado, la tecnología de virtualización de servidores dio paso a una revolución en la forma de ver el área de la infraestructura de sistemas. Hoy en día la virtualización de contenedores está llamada a realizar una nueva revolución en la forma de ver y pensar en las aplicaciones y servicios de las empresas. Por ello, una análisis temprano y en profundidad de esta tecnología asienta las bases para consolidarla y aportar un punto de claridad a aquellas organizaciones que puedan estar pensando en dar el paso y apostar por esta nueva tecnología para sus despliegues.

Como se trata de una tecnología de vanguardia y que está presente en el ámbito del despliegue de aplicaciones corporativas, hace que los fabricantes tanto de software como de hardware más punteros del mercado estén volcados en enfocar sus productos más innovadores y su estrategia de I+D entorno a ellos, por lo se generan gran cantidad de sinergias que hacen que este estudio tenga un importante componente de innovación y grandes posibilidades de aplicación directa.

Son estos dos puntos los que me han motivado a plantear este proyecto “*Orquestación de aplicaciones corporativas mediante la virtualización con Docker”* para poder realizar un estudio y análisis en profundidad .

1.1.4. Ámbito de aplicación del proyecto

El estudio de este proyecto se va a llevar a cabo desde el ámbito de una empresa que ya dispone actualmente de una infraestructura de TI , tanto de tipo tradicional como en modalidad “cloud” y contemplando tanto la posibilidad de que se trate de una infraestructura contratada a un proveedor externo tipo “cloud” como desde el punto de vista de un CPD más tradicional con infraestructura TI propia.

Se enfocará también el proyecto orientado al ámbito de organizaciones que cuenten con un departamento de TI propio, o al menos con algún técnico del área de sistemas que posibilite la implantación y puesta en marcha de esta solución interna y su posterior mantenimiento.
El ámbito general de aplicación irá enfocado también a todas aquellas organizaciones de cualquier tamaño que tengan la necesidad de ofrecer un servicio o aplicación específica, tanto para el uso interno de la organización como para formar parte de un producto de software o servicio desarrollado para ofrecer como producto a un tercero.

Debido a que a través de este proyecto estamos implantando un sistema de despliegue de aplicaciones de un modo sencillo, y que una vez implantado permitirá desplegar instancias de aplicaciones sin necesidad de tener un profundo conocimiento en el área de sistemas y de las operaciones de TI, este es también aplicable a aquellas empresas que quieran poner en marcha o potenciar un nuevo concepto de modelo de trabajo en el desarrollo de aplicaciones (que está teniendo un gran auge hoy en día) denominado “DevOps” (acrónimo del inglés de “development” (desarrollo) y “operations” (operaciones). Este modelo de desarrollo surge como una respuesta a la interdependencia del desarrollo de software y las operaciones de TI, y su principal objetivo es ayudar a una organización a producir productos y servicios software rápidamente potenciando que el propio desarrollador pueda implantar y mantener la aplicación y el despliegue de forma autónoma y permitiendo la abstracción de la arquitectura subyacente).

1.2. Objetivos del Trabajo

El objetivo estratégico que se plantea para este proyecto podemos definirlo como **“La puesta en marcha de una solución que permita la Orquestación de aplicaciones corporativas a través de la implantación de un sistema de contenedores virtuales usando la plataforma abierta Docker ”**.

Los principales objetivos operativos de este proyecto son:

- Reducción del gasto en infraestructura de la empresa para el despliegue de aplicaciones corporativas.
- Reducción de los tiempos de despliegue de aplicaciones y servicios corporativos.
- Mejora de la disponibilidad y de los niveles de servicio (SLA).
- Mejora de la productividad de toda la cadena de valor involucrada en la prestación de servicios de TI.
- Posibilidad de integrar el concepto de desarrollo de aplicaciones y mantenimiento integral de las mismas (Implantación del modelo DevOps).
- Posibilidad de implantación del concepto de “Bimodal IT[7]” en una infraestructura TI empresarial, permitiendo la gestión de 2 modelos de operación TI en la organización.
- Mejora de la imagen global de la empresa en los mercados derivado del uso de herramientas tecnológicas de última generación.

1.3. Hipótesis de trabajo, enfoque y método seguido.

Hipótesis de trabajo: Partimos de que somos los responsables de IT de un empresa de servicios informáticos. Nuestra empresa es una compañía que se dedica al desarrollo de sistemas y soluciones informáticas que, además de desarrollos informáticos propios en forma de aplicaciones orientas al consumo corporativo de terceras empresas, también ofrece desarrollos y soluciones a medida.

7 http://www.gartner.com/it-glossary/bimodal/

Esta empresa se puede considerar una empresa joven, que ha ido creciendo en volumen de facturación y tamaño auspiciada por el boom tecnológico de los últimos años. El factor clave del éxito de nuestra empresa siempre ha sido apostar por el uso de las tecnologías más vanguardistas, tanto para el desarrollo de sus productos como para ofrecerlos en forma de servicios en proyectos a medida de terceros.
Además siempre a destacado por emplear estas ventajas tecnológicas, no solo en los aspectos más técnicos sino también emplearlos para innovar en los procesos de gestión y producción.
La empresa en este último ejercicio fiscal facturó un volúmen de 5 millones de euros y al cierre del ejercicio contaba con más de 50 empleados distribuidos en distintos departamentos. Dispone con un pequeño departamento de administración que se encarga de las tareas propias de la facturación a clientes y la gestión contable diaria. Este departamento depende directamente de la dirección de la empresa encabezada por un director o CEO, un director tecnológico o CTO y un par de socios. De este mismo consejo de dirección dependen dos grandes departamentos, que vertebran por un lado el área de desarrollo de productos y gestión de proyectos, y por otro lado un departamento de TI.
El departamento de desarrollo y gestión de proyectos engloba a más del 70% de la plantilla y esta formado por analistas, programadores especializados en distintos lenguajes de programación, diseñadores y maquetadores web, jefes de proyecto, etc.
Este departamento es el que se encarga del desarrollo de las distintas aplicaciones, así como de la gestión y dirección de las soluciones y proyectos que realiza la empresa.
La empresa cuenta también con un departamento IT, que es el que se encarga de la arquitectura y mantenimiento de los distintos sistemas informáticos empleados internamente por la empresa para dar servicio a los usuarios, y del análisis , arquitectura y mantenimiento de los sistemas empleados en los productos desarrollados y en los proyectos a medida.
En este departamento de IT la empresa cuenta con profesionales con gran experiencia en arquitecturas de todo tipo de soluciones de servidor basados en software libre y en el uso de los distintos sistemas operativos de servidor basados en Linux. Trabajan fundamentalmente con el sistema operativo Ubuntu, que es con el que estos se han ido formando y por tanto se sienten más cómodos en el día a día, pero también trabajan con otros sistemas operativos Linux (RHEL, CentOS, Debian) que también tienen instalados en sus servidores para dar soporte a distintas funcionalidades.
En sus orígenes la empresa apostó por la instalación de un novedoso CPD en el sótano de sus oficinas centrales, que cuenta con un sistema de alimentación ininterrumpida, dobles acometidas eléctricas y varios proveedores de red redundantes que dan servicio de conectividad. Asimismo, también se invirtió en la compra de servidores físicos para dar servicios a la empresa y clientes a través de este CPD.
Posteriormente, hace 4 años con la aparición de la virtualización y del auge de los entornos tipo “cloud” la organización reoriento parte de los servidores de este CPD para hacer uso de esta tecnología y para emplearla para desplegar sus productos y servicios. Como la empresa ya contaba con personal de IT para poder llevar a cabo este tipo de proyectos internos se puso en marcha la instalación de una solución de código abierto sobre entornos de servidor Linux, conocida como OpenStack[8].
Este proyecto de sistema de virtualización ha resultado todo un éxito de cara a la dirección de la empresa ya que ha permitido obtener una ventaja competitiva gracias a que les permitido dotarse de un sistema de despliegue de aplicaciones ágil y permitirles una importante reducción de costes.
En las últimas fechas la empresa esta sufriendo una serie de retrasos en los nuevos despliegues de servicios, debido a la sobrecarga de trabajo que los técnicos de IT están teniendo en los continuos despliegues de nuevos sistemas de producción que están poniendo en marcha. A esta sobrecarga de despliegues también se añade el trabajo de puesta en marcha de nuevos entornos de pruebas para los equipos de desarrollo que continuamente necesitan probar nuevas herramientas para su análisis.

8 https://es.wikipedia.org/wiki/OpenStack

La dirección de la empresa ha mostrado su preocupación por perder esta ventaja competitiva sobre el mercado, por lo que ha encargado a su responsable de IT el realizar una análisis de viabilidad de un nuevo sistema de virtualización de contenedores de aplicaciones del que tanto se esta hablando entre las grandes empresas de Silicon Valley, y en cuyos círculos se especula que supondrá una nueva revolución en los entornos para despliegues de aplicaciones corporativas, como la que supuso hace un par de años los sistemas de despliegues tipo “cloud”.
Para este cometido el responsable de IT arranca un proyecto cuya estrategia pretende analizar los sistemas de virtualización de contenedores existentes actualmente en el mercado, usando tecnologías de código abierto y sobre sistemas operativos Linux, con el fin de poder reaprovechar la infraestructura del CPD con el que ya cuenta la empresa, y todo ello orientado a poder ofrecer un sistema que permita orquestar el despliegue de aplicaciones.

1.4. Requisitos

Tal y como introducimos en el supuesto anterior, partimos de una empresa que ya posee un CPD propio con toda la infraestructura necesitaría para mantenerlo, mediante la presencia de varios ISP de Internet para su uso, climatización, doble acometida eléctrica, etc así como el personal necesario para mantenerlo funcionando correctamente. Por este motivo, vamos a implantar la solución que se plantea en este proyecto dentro de este mismo CPD y aprovechando el hardware interno que la empresa posee.
Vamos a necesitar los siguientes requisitos hardware para poner en marcha el proyecto:

- 2 servidores **Dell Poweredge R430**[9] con la siguientes características:
  - Formato RACK 19” de 1U de dimensiones: 48.24 cm x 60.7 cm x 4.28 cm
  - 2 x Intel Xeon E5-2620 v4 2.1GHz 8C / 16T
  - 2 x 32GB RDIMM, 2400MT/s, Dual Rank, x4 Data Width
  - PERC H730 RAID Controller, 1GB NV Cache
  - 3 x 400GB Solid State Drive SATA tipo MLC 6Gbps 2.5 pulgadas Hot-plug en configuración en RAID 5
  - Doble fuente de alimentación con soporte Hot-plug (1+1) y 550W

  Partimos de 2 servidores para permitir dotar al sistema de alta disponibilidad y redundancia y realizaremos una arquitectura que además permita el alto rendimiento para poder combinar la potencia de cómputo de ambos servidores. Del mismo modo, la arquitectura propuesta permitirá aumentar la capacidad de cómputo simplemente escalando el número de servidores
- 2 switches gestionados red Dell **PowerConnect x1026**[10], con las siguientes características:
  - Formato RACK 19” de 1U. Dimensiones 41,25 mm x 209 mm x 250 mm
  - 24 puertos Gigabit Ethernet 10/100/1000BASE-T
  - Hasta 52 Gb/s de capacidad
  - 38,7 Mp/s de velocidad de reenvío
  - Alimentación máxima 17,5 W
  - Módulo de fibra SFP 1Gb

  Partimos de 2 switches para permitir dotar al sistema de alta disponibilidad y redundancia y realizaremos una arquitectura que permita la tolerancia a fallos.

9 http://i.dell.com/sites/doccontent/shared-content/data-sheets/en/Documents/Dell-PowerEdge-R430-Spec-Sheet.pdf
10 http://i.dell.com/sites/doccontent/business/smb/merchandizing/es/Documents/Dell_Networking_X_Series_spec_sheet_ES.pdf

- 2 routers de acceso Internet Cisco Catalyst de la serie 1900 o similar cada uno de ellos facilitado por un ISP distinto para dotar el acceso a los servicios también de alta disponibilidad.

1.5. Planificación del Trabajo

A continuación se incluye una lista resumida de tareas y un diagrama de Gantt :

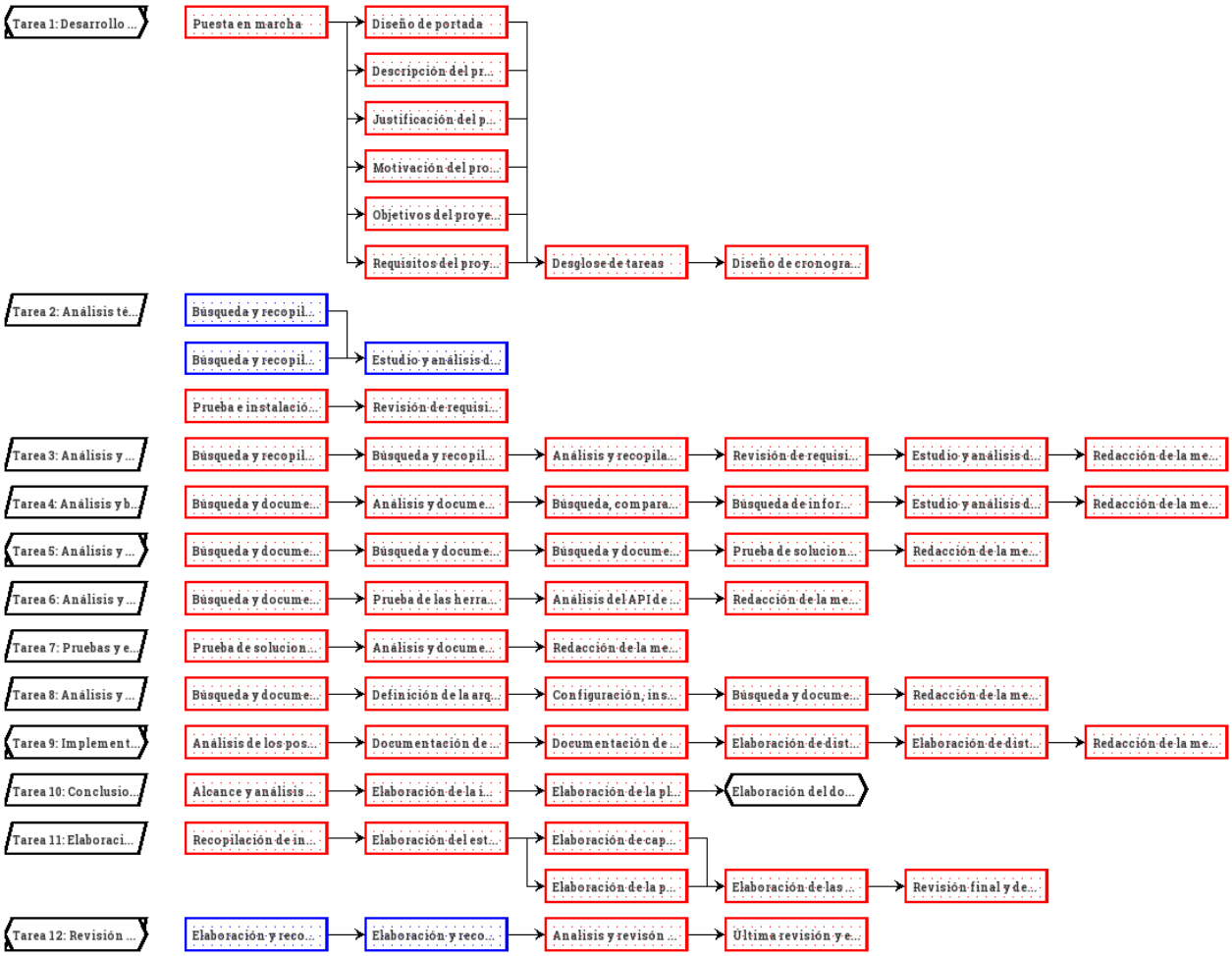


Ilustración 1: Diagrama de tareas del proyecto

Nombre	Inicio	Terminado
⊞Tarea 1: Desarrollo de un plan de trabajo	21/09/16 9:00	7/10/16 17:00
⊞Tarea 2: Análisis técnico previo de la solución global del proyecto	8/10/16 9:00	13/10/16 17:00
⊞Tarea 3: Análisis y documentación de las arquitecturas de virtualización	14/10/16 9:00	20/10/16 17:00
⊞Tarea 4: Análisis y búsqueda de información sobre virtualización de contenedores	21/10/16 9:00	26/10/16 17:00
⊞Tarea 5: Análisis y estudio de la herramienta Docker	27/10/16 9:00	11/11/16 17:00
⊞Tarea 6: Análisis y documentación del ecosistema Docker	12/11/16 9:00	18/11/16 17:00
⊞Tarea 7: Pruebas y ejemplos de funcionamiento del ecosistema Docker	19/11/16 9:00	25/11/16 17:00
⊞Tarea 8: Análisis y búsqueda de información sobre orquestación de contenedores	26/11/16 9:00	5/12/16 17:00
⊞Tarea 9: Implementación de la solución y ejemplos de uso	6/12/16 9:00	16/12/16 17:00
⊞Tarea 10: Conclusiones	17/12/16 9:00	20/12/16 17:00
⊞Tarea 11: Elaboración de la presentación	21/12/16 9:00	31/12/16 17:00
⊞Tarea 12: Revisión final del proyecto. Bibliografía y Anexos	1/01/17 9:00	4/01/17 17:00

Ilustración 2: Listado de tareas del proyecto

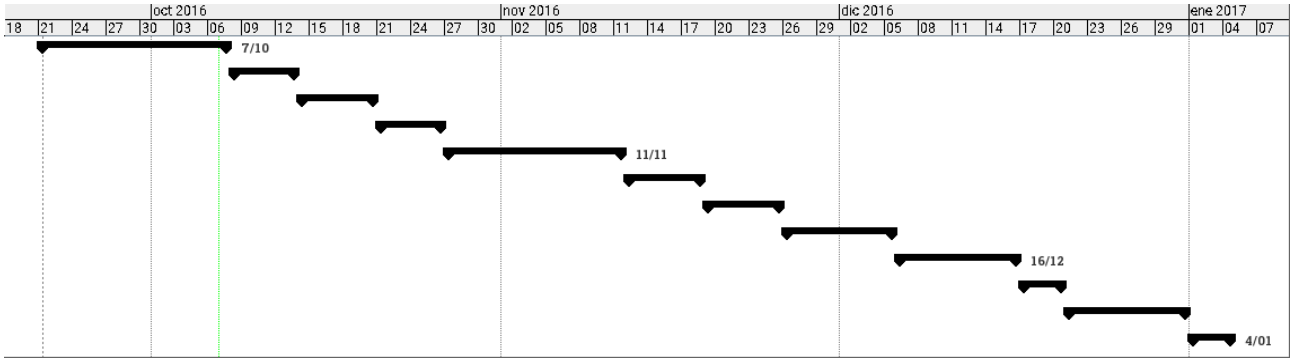


Ilustración 3: Diagrama de Gantt

En los anexos finales del proyecto puede consultarse la lista detallada de tareas y diagramas de Gantt.

1.6. Breve sumario de productos obtenidos

A partir de este trabajo se generará una documentación completa y un análisis de un sistema de virtualización de contenedores, así como un manual de instalación para la puesta en marcha de un sistema de despliegue de aplicaciones a través de un entorno de contenedores virtuales, junto con un conjunto de ejemplos de aplicaciones Opensource existentes en el mercado de distintas áreas y que suelen ser usadas frecuentemente en el área corporativa.
Adicionalmente a la documentación en formato electrónico y a las aplicaciones de ejemplo, se facilitará un enlace de descarga de una simulación de la infraestructura propuesta a través de máquinas virtuales y en un formato Open Virtual Format que permita importar el entorno para pruebas.

1.7. Breve descripción de los otros capítulos de la memoria

En los primeros capítulos, introducimos información sobre la virtualización de servidores y la virtualización de contenedores, explicando las ventajas y diferencias de ambas tecnologías e introduciendo el funcionamiento de ambas. A continuación introducimos el concepto de orquestación y microservicios para continuar con una propuesta de modelo de infraestructura para nuestro proyecto.
Posteriormente seleccionamos la herramienta de virtualización de contenedores Docker y su ecosistema de aplicaciones como solución a la orquestación estática mediante Docker Compose y a la orquestación dinámica mediante Docker Swarm. Se explica el ecosistema de aplicaciones y se estudian estas dos principales herramientas de orquestación, una vez visto el funcionamiento básico del núcleo principal: Docker Engine.
Finalmente cerramos el estudio de la virtualización de contenedores con dos ejemplos de despliegue de aplicación a través de orquestación estática y dinámica.

2. Virtualización

2.1. Introducción a la computación en la nube

El término de computación en la nube o más comúnmente conocido como «cloud computing», hace referencia a una nueva interpretación tecnológica y de modelo de negocio que engloba ideas muy diversas en cuanto al tratamiento del almacenamiento de la información, comunicación entre los ordenadores, provisión de servicios , metodologías del proceso de desarrollo, etc, todo ello englobado bajo el concepto de «nube». De este modo, podemos verlo como un nuevo paradigma que permite ofrecer servicios de computación de forma distribuida a través de Internet.

Estos servicios que se ofrecen a través de la red a través del modelo «como un servicio» podemos englobarlos en varias tipologías en función modelo de negocio asociado al modelo de cloud computing. Estas tipologías se conocen comúnmente como la generación «As a Service», del inglés «como un servicio». Haciendo referencia a su función, son básicamente 3 modelos de servicio: IaaS (*Infrastructure as a Service del inglés Infraestructura como servicio*), PaaS (*Platform as a Service de inglés plataforma como servicio*) y SaaS (*Software as a Service del inglés software como servicio*). Más delante detallaremos más en profundidad la características de estos modelos.

El concepto de «cloud computing» se hizo popular y cobró más auge a través de los grandes proveedores de servicios de Internet, como Google, Amazon o Microsoft que empezaron a construir su propia infraestructura orientada a ofrecer sus productos a través de este tipo de modelo de servicios. En el desarrollo y aumento de velocidad de Internet fueron uno de los factores claves para su consolidación.
El otro factor clave para el auge y consolidación del cloud es la tecnología de virtualización. Como hemos visto, el cloud se apoya en un sistema de infraestructura tecnológica dinámica que se caracteriza, entre otros factores, por una rápida movilización de los recursos, una elevada capacidad de adaptación para atender a una demanda variable de recursos y un alto grado de automatización. Todos ellos fueron posibles gracias a la incursión de la virtualización de infraestructuras. Es tan determinante este último factor que a veces se tiende a confundir la virtualización en si con el concepto de cloud computing.
Como veremos en la próxima sección, donde entraremos a profundizar conceptos sobre virtualización, consiste en la creación a través de software de una versión virtual de algún recurso tecnológico. De este modo, la virtualización permite crear servidores, estaciones de trabajo, almacenamiento y sistemas de red aislados de la capa de hardware físico específico subyacente.

Y una vez vistos ambos conceptos, podemos comprender más claramente, la diferencia entre virtualización y cloud, de modo que cuando hablamos de virtualización nos referimos a software que manipula o gestiona el hardware, mientras que cuando hablamos de cloud nos referimos más concretamente al servicio resultante que podemos ofrecer.

2.1.1. Características básicas del cloud computing.

Para poder entender las claves básicas del cloud computing debemos de tener en cuenta una serie de características que definen de forma distintiva los sistemas de cloud computing de los sistemas más clásicos en el despliegue de servicios de carácter corporativo. Podemos distinguir las siguientes características principales:

- **Escalabilidad o elasticidad:** Esta característica permite aumentar o disminuir las capacidades ofrecidas por el servicio de cloud en función de necesidades puntuales, con

gran celeridad y sin suponer un coste adicional para el servicio o sin incurrir en penalizaciones por el uso.

- **Autoservicio bajo demanda:** Esta característica permite a los usuarios auto abastecerse de recursos informáticos tales como capacidad de computo o almacenamiento en la medida de sus necesidades puntuales, sin que sea necesaria la intervención humana del proveedor de servicios.
- **Acceso amplio a la red**: Esta característica permite a los usuarios la posibilidad de acceder a los servicios de cloud en cualquier lugar, momento y desde cualquier tipo de dispositivo que disponga de conexión a la red (tablet, móvil, portátil, equipo de sobremesa, etc...)
- **Servicio cuantificable y asequible:** Esta característica permite dotar a los sistemas cloud de un control y optimización de recursos que permita su medición con un alto grado de abstracción.
- **Multiusuario:** Esta característica permite a los servicios cloud la capacidad de compartir a los usuarios los medios y recursos informáticos, dotándolos de un máximo aprovechamiento.

2.2. Capas o Modelos de servicio del cloud computing

Tal y como introducimos al principio de este apartado, el cloud computing consiste en ofrecer servicios a modo de recursos y existen diversas capas o modelos de servicio de cloud en función de como sean estos servicios, que son :

- **IaaS** (*del inglés Infraestructura como servicio* - ***Infrastructure as a Service***).
  Este modelo de servicio se basa en poner a disposición del usuario final una infraestructura informática (capacidad de computo, espacio de almacenamiento, recursos de red, etc) como un servicio. De este modo, el usuario final en lugar de tener la necesidad de adquirir recursos para infraestructuras tradicionales adquiere recursos sin una limitación mayor que su capacidad económica ya que adquiere servicios de infraestructura escalables.
  También podremos verlo nombrado como HaaS (Hardware as a Service) en algunas publicaciones, pero se refiere a este mismo modelo. Ejemplos comerciales del modelo IaaS podrían ser los servicios Amazon Web Services o AWS[11] (EC2, S3, Route53), Google CloudPlatform[12], Rackspace Cloud[13], etc
- **PaaS** (*del inglés Plataforma como servicio* - ***Platform as a Service***).
  Este modelo de servicio se basa en poner a disposición del usuario unas herramientas que le permitan la realización de desarrollos informáticos completos, de manera que le permitirá construir sus aplicaciones completas o distintas piezas de software sin necesidad de adquirir e implantarlas en equipos tradicionales, y obviando las posibles licencias y mantenimiento de estas herramientas. Además ofrece a través de la red todos los recursos y requisitos necesarios para crear y desplegar todo lo necesario para el correcto funcionamiento de este software.
  Ejemplos comerciales del modelo PaaS podría ser Google App Engine[14] o Microsoft Azure[15] que ofrecen todo un ecosistema de aplicaciones para el desarrollo, test y despliegue de servicios de software.
- **SaaS** (*del inglés Software como servicio* - ***Software as a Service***).
  Este modelo se basa en poner a disposición del usuario aplicaciones completas de software como si se tratasen de un servicio (además de un modelo de despliegue de

11 https://aws.amazon.com/es/
12 https://cloud.google.com/
13 https://www.rackspace.com/es/cloud
14 https://cloud.google.com/appengine/
15 https://azure.microsoft.com/es-es/?WT.srch=1&WT.mc_ID=SEM_t2xPfJza

software) incluyendo las licencias que puedan ser necesarias para que puedan usarlo como un servicio bajo demanda. De este modo, podemos considerarlo como la oferta de un producto terminado que ofrece un servicio especifico para el que fue desarrollado siendo un modelo de pago por uso.
Ejemplos comerciales del modelo IaaS podrían ser Office365[16] de Microsoft, GSuite[17] de Google, Dropbox[18], etc.

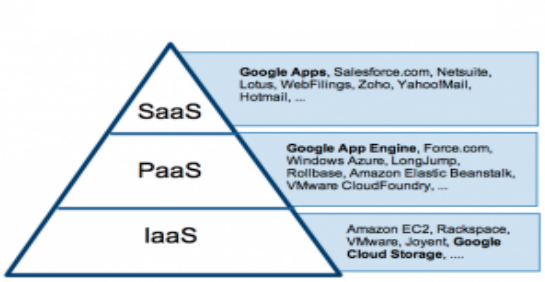


Ilustración 4: Modelo de servicios cloud según Gartner

[GART01]

Como podemos apreciar en la ilustración anterior IaaS se encuentra situado en la base de pirámide, en medio de la misma se encuentra PaaS y coronando la cúspide podemos ver el modelo SaaS. Cada nivel que ascendemos en la pirámide, vamos añadiendo capas de abstracción a nuestro modelo siendo por tanto el modelo SaaS el que mayor abstracción permite a cualquier modelo de servicios en cloud. Podemos ver que este proyecto se engloba dentro del modelo SaaS ya que se pretende implantar una plataforma completa para el despliegue de aplicaciones orquestadas como servicios (concretamente orientadas a lo que actualmente se conocen con el nombre de microservicios y que introduciremos más adelante en el capítulo de orquestación).

[WIKI1]

[TICBEAT1]

2.3. Virtualización

Una de las tecnologías principales que han propiciado la consolidación del cloud es la virtualización, consistente en la creación o simulación a través de software de una versión virtual de algún recurso tecnológico. Este recurso suele ser habitualmente CPU, Memoria, Almacenamiento y componentes de Red (o la combinación de varios) y la capa de software que los gestiona y maneja, conocida como Hypervisor o VMM (Virtual Machine Monitor), que crea una capa de abstracción entre estos recursos y el sistema operativo de las máquinas “invitado” (como se conocen los sistemas operativos virtuales que se ejecutan sobre el mismo anfitrión y sobre el que comparten recursos).
Dicho de otra manera y en un sentido más general, virtualizar es particionar un servidor físico en varios servidores virtuales compartiendo todos ellos los recursos hardware de la máquina anfitrión.

2.4. Tipos de virtualización.

16 https://products.office.com/es-es/business/explore-office-365-for-business
17 https://gsuite.google.com/intl/es/
18 https://www.dropbox.com/business

Existen varios tipos de virtualización en función de los recursos de se virtualizen y de como se realice esta virtualización. Así podemos ver los siguientes tipos de virtualización:

- **Virtualización a nivel Hardware**:
  Emular mediante máquinas virtuales distintos componentes hardware como redes, almacenamiento, etc.
- **Virtualización a nivel de Sistema Operativo**:
  No se virtualiza hardware y se ejecuta una única instancia de sistema operativo por cada servidor virtual. Existen los siguientes subtipos:
  - Virtualización Completa (Fullvirtualization):
    Consistente en ejecutar sistemas operativos virtualizados empleando un hypervisor como intermediario entre el hardware físico y los distintos sistemas operativos que se virtualizan. El hypervisor es el encargado de interactuar directamente con el servidor físico aislando los sistemas operativos virtuales entre si. Este tipo de virtualización está formado por 3 capas: la capa Hardware, el hypervisor y los distintos sistemas operativos.
    Este tipo de virtualización tiene la ventaja de que no es necesario modificar los sistemas operativos anfitriones con ninguna capa adicional de software que de soporte al hypervisor. En contrapartida, son generalmente más lentos porque el hypervisor consume recursos adicionales para la gestión del hardware.
    Ejemplos de este tipo de virtualización son KVM, VirtualBox de Oracle o VMWare.

  - Paravirtualización (Paravirtualization):
    Consistente en ejecutar sistemas operativos virtualizados denominados “guest” o “Invitados” sobre otro sistema operativo que actúa como host anfitrión y que se comunica con los sistemas operativos “guest” a través del hypervisor. Es el caso de la virtualización completa, pero en este caso se introduce software adicional sobre los sistemas operativos virtualizados que descarga al hypervisor de muchas tareas y hace que estos sistemas operativos sean conscientes de que están virtualizados y que trabajan a través del anfitrión.
    Las ventajas de este tipo de virtualización pueden considerarse desventajas para la virtualización completa, por lo que este tipo de virtualización es más ligero en gasto de CPU que el anterior pero requiere de software adicional y de modificar el sistema operativo anfitrión para su funcionamiento. Ejemplo de este tipo de virtualización es Xen Server

  - Virtualización de contenedores:
    Consistente en ejecutar sistemas operativos virtualizados denominados “guest” o “Invitados” en forma de una aplicación aislada sobre otro sistema operativo que actúa como host anfitrión, pero que a diferencia de los anteriores en este tipo, ambos sistemas operativos comparten el mismo Kernel y estos se aíslan lógicamente mediante distintas soluciones técnicas que consisten básicamente en asignar espacios de nombres distintos a cada sistema operativo invitado.

[WIKI2]

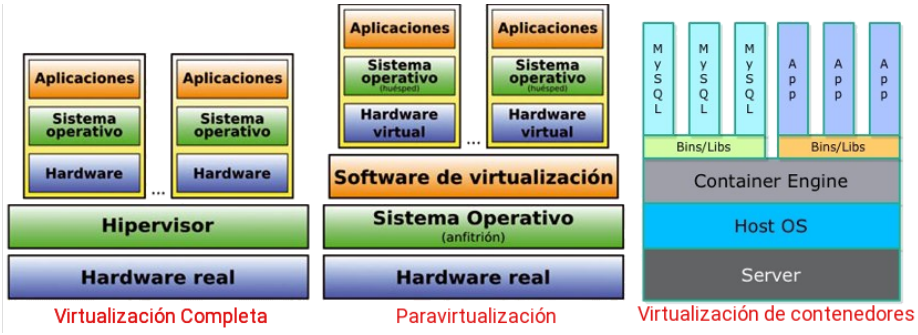


Ilustración 5: Tipos de virtualización de sistemas operativos

En este proyecto haremos uso de la virtualización de contenedores, por lo que a continuación hablaremos de ella más detalladamente en el siguiente capítulo.

3. Virtualización de contenedores

3.1. Introducción a la virtualización de contenedores. El concepto de contenedor.

La virtualización basada en contenedores es una aproximación a la virtualización en la cual se ejecuta la capa de virtualización como una aplicación aislada dentro del sistema operativo del equipo anfitrión. En este tipo de sistemas sólo se ejecuta un único núcleo del sistema operativo o Kernel[19], que es el del sistema operativo anfitrión, y este ayudado por el software de virtualización de contenedores específico es el que se encarga de crear nuevos entornos de ejecución (que podríamos comparar con las máquinas virtuales) y que reciben el nombre de contenedores.
Un contenedor es sencillamente un proceso para el sistema operativo que internamente contiene la aplicación que queremos ejecutar y todas las dependencias derivadas de la misma. Empaquetamos una aplicación en una unidad estandarizada para desempeñar un servicio que contiene lo necesario para funcionar como un todo (código, librerías, software, etc), empaquetado bajo la analogía de un contenedor.

[BLOG1]

[wiki3]

3.2. Diferencias entre virtualización clásica y virtualización de contenedores.

Existen múltiples similitudes entre ambas tecnologías de virtualización por lo que vamos a centrarnos en señalar las diferencias más importantes, pues son las que nos permitirán comprender posteriormente las ventajas e inconvenientes de cada una, así como cuando debemos usar un modelo de virtualización u otro.
Las diferencias más importantes que podemos encontrar en el ámbito de la arquitectura son las siguientes:

- La virtualización clásica conlleva la virtualización de forma obligatoria de todo un sistema operativo completo dentro de cada máquina virtual que queramos implementar, mientras que con la virtualización de contenedores sólo debemos de ejecutar un contenedor, por lo que podemos compararlo con la ejecución de un proceso aislado dentro del sistema operativo. Al estar obligados a mantener un sistema operativo por cada máquina virtual, incurrimos en una sobrecarga de cómputo y un aumento del número de horas de mantenimiento del sistema operativo de la virtualización clásica, lo cual no sucede en el caso de los contenedores donde esto no sucede. Podemos decir que la virtualización de contenedores aprovecha mejor los recursos y requiere menos capacidad de cómputo y horas de mantenimiento del sistema.
- El tiempo de puesta en marcha o arranque de un contenedor virtual se reduce a la ejecución de un nuevo proceso para el sistema operativo, mientras que con un entorno de virtualización clásico requiere el arranque normal de un sistema operativo con todos los componentes habituales y servicios que pueda tener, por lo que podemos afirmar que el arranque de un contenedor virtual es más rápido que el arranque en un entorno de virtualización clásico.

19 https://es.wikipedia.org/wiki/N%C3%BAcleo_(inform%C3%A1tica)

- La virtualización de contenedores no exige de una máquina física para poder ejecutarse mientras que un sistema de virtualización clásico necesita de un equipo físico anfitrión y un hypervisor para realizar el control. En el caso del software necesario para la virtualización de contenedores puede ejecutarse sin problemas sobre un servidor virtual como equipo anfitrión, lo cual permite una mayor versatilidad.

A continuación mostramos en la siguiente figura los distintos componentes de ambas arquitecturas de virtualización, de modo que podemos apreciar de un modo más visual las diferencias que hemos estado analizando:

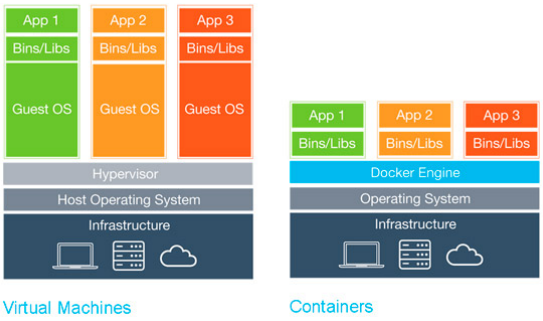


Ilustración 6: Arquitectura de máquinas virtuales clásicas frente a contenedores (Docker)

Llegados a este punto, veamos a modo de ejemplo qué componentes requeriría virtualizar un servicio, como puede ser una base de datos, usando ambas tecnologías:

- Usando una máquina virtual:
    1. Una máquina física como soporte hardware con un sistema operativo anfitrión instalado.
    2. Un hypervisor que gestione las peticiones de hardware virtual y las gestione sobre el hardware local.
    3. Un sistema operativo en el equipo virtualizado (Sistema operativo “guest” o invitado).
    4. Sobre este sistema operativo invitado tener instalado el software de gestión de bases de datos que nos proporcione el servicio final.

- Usando un software de contenedores:
    1. Una máquina física como soporte hardware con un sistema operativo anfitrión instalado.
    2. Un software de virtualización (LXC, Docker, etc) de contenedores instalado sobre el sistema operativo anfitrión.
    3. Un contenedor con el software y dependencias del sistema de gestión de bases de datos que nos proporcione el servicio.

3.3. ¿Qué tecnología debemos usar? Virtualización de contenedores o virtualización clásica.

Una vez introducidas tanto las tecnologías de virtualización clásica como las de virtualización de contenedores y haber visto como funcionan y la arquitectura básica que les permite funcionar, así como las ventajas e inconvenientes que presentan, es imposible no tratar de pensar si ambas

tecnologías se solapan en funcionalidad y hacernos la pregunta de cual de ellas es mejor para llevar a cabo el despliegue de aplicaciones.
Aunque a priori se puede considerar que ambas realizan una función similar, consistente en virtualizar un entorno para ofrecer un servicio específico, todos los expertos tienen claro que ambas tecnologías aportan valores distintivos, por lo que podemos considerar que una u otra aporta las misma funcionalidades. De modo que lo que en una son ventajas en la otra pueden considerarse inconvenientes considerándose así sistemas complementarios. Dependiendo de lo que necesitemos virtualizar o los requisitos que tenga nuestro servicio será mejor un sistema u otro.
Si necesitamos el despliegue de aplicaciones dónde sea necesaria una arquitectura de microservicios (Desarrollar aplicaciones en base a pequeños microservicios que funcionan de modo autónomo (BBDD, servidores web, etc)) o si necesitamos elasticidad o varias copias de nuestras aplicaciones y poder dimensionar nuestros servicios de forma rápida y automatizada, la mejor solución será la virtualización de contenedores. En caso de necesitar un entorno más similar a la computación clásica donde necesitamos un sistema operativo con funcionalidades más avanzadas, la solución será la virtualización clásica.
Para este proyecto vamos a orquestar el despliegue de aplicaciones usando arquitectura de microservicios y buscando la máxima rapidez y escalabilidad en el despliegue, por lo que nuestra opción pasa por el uso de la virtualización de contenedores.

3.4. Precedentes tecnológicos y anatomía de los contenedores

Las técnicas para enjaular un proceso en un espacio aislado dentro de un sistema operativo donde pueda ejecutarse no son nuevas. Este tipo de tecnología aparece por primera vez alrededor del año 1979 de la mano del comando "**chroot**"[20] que se introduce en los sistemas operativos Unix como concepto de desarrollo y que permitía aislar a un proceso del sistema operativo y a todos los procesos hijos que estuviese dentro de una ruta, de modo que la interpretasen como directorio raíz. A partir de los primeros usos en Unix fue pasando rápidamente a soportarse en otros sistemas operativos como BSD. De este modo podemos considerar a este comando como el punto de partida de la tecnología de virtualización de contenedores que van extendiendo el concepto para aumentar funcionalidades hacia los entornos aislados de ejecución.
El siguiente salto hacia lo que hoy son contenedores se produjo con la aparición del comando "jail[21]" el cual orienta y amplia las funcionalidades de chroot introduciendo, como su propio nombre indica, las jaulas que añaden el el que proceso se ejecuta de modo aislado o "enjaulado". Es muy habitual ver ambos procesos combinados de la mano con el nombre de "jaulas chroot". A partir de este momento empiezan a extender su compatibilidad y uso a otros sistemas operativos como Solaris, AIX, HP-UX hasta que finalmente (aproximadamente sobre el año 2008) aparece la primera virtualización de contenedores completa en Linux con el nombre de **LXC** (Linux Containers) y que se basa en tecnologías base implementadas en el Kernel del sistema operativo Linux para proporcionar la abstracción necesaria para crear los contenedores.
La tecnología de contenedores nace pues, de la mano del sistema operativo Unix, por lo que podemos considerar a este sistema operativo y a sus variantes (BSD, HP-UX, Linux, etc) como la solución nativa para la virtualización de contenedores.

Para este proyecto, tal y como hemos introducido en las hipótesis iniciales, buscamos una solución de contenedores de código abierto para una empresa con personal acostumbrado a trabajar en el sistema operativo Linux por lo que, junto con que es la solución nativa para el trabajo con contenedores, es el sistema operativo en el que nos vamos a centrar para nuestro

20 https://es.wikipedia.org/wiki/Chroot [wiki5]
21 https://en.wikipedia.org/wiki/FreeBSD_jail [wiki6]

análisis (A pesar de que actualmente no es un requisito obligatorio para trabajar con contenedores ya que existan soluciones para ejecutar contenedores de modo nativo en otros sistemas operativos como puede ser Microsoft Windows).

Todas la soluciones de virtualización de contenedores Linux actuales, como hemos visto, basan su funcionamiento en una serie de librerías o funcionalidades básicas del Kernel y vertebran la arquitectura que da soporte a su funcionamiento. Estos componentes son:

3.4.1. Namespaces

Los Namespaces o espacios de nombres son una funcionalidad del Kernel de Linux (Soportada a partir de la versión 2.6.23) que permite aislar y virtualizar recursos del sistema Linux como una colección de procesos garantizando el aislamiento entre los mismos.
Ejemplos de recursos que pueden aislarse son el identificador de los procesos (PID), los nombres de máquina, identificadores de usuario (UID), recursos de red, etc.
Los namespaces envuelven uno de estos recursos globales del sistema operativo en un capa de abstracción que lo aísla y que está ligada solo a los procesos que están dentro de este mismo espacio de nombres.
Actualmente existen 6 tipos distintos de namespaces, cada uno de los cuales está relacionado con el aislamiento especifico de algún aspecto del sistema. Son los siguientes:

3.4.1.1. NETWORK namespace

Es la funcionalidad que proporciona el aislamiento de los entornos a nivel de red. A través de esta característica podemos tener distintos NETWORK namespaces cada unos de ellos con su propia capa de red y configuración específica y aislada.

3.4.1.2.PID namespace

Es la funcionalidad que proporciona el aislamiento de los procesos de usuario (PID). Esto permite que los procesos de usuario sólo sean visibles por el proceso padre que los ha creado, de modo que los namespaces padres pueden ver y modificar a los namespaces hijos pero no al contrario. Esto, permite entre otras cosa que halla procesos con el mismo PID dentro de un único sistema operativo; eso sí, cada uno dentro de un PID namespace distinto que es el que les dota de aislamiento.

3.4.1.3.UTS namespace

Es la funcionalidad que proporciona el aislamiento de los identificadores de sistema relacionados con el nombre de la máquina (HOSTNAME) y dominio de la misma (DOMAINNAME) lo cual permite asociar un nombre y dominio de máquina específico para cada namespace, que serán reconocidos sólo por procesos del mismo namespace.

3.4.1.4.MOUNT namespace

Es la funcionalidad que proporciona el aislamiento a nivel de puntos de montaje del sistema de ficheros, lo cual permite la reutilización y aislamiento de los puntos de montaje del sistema de ficheros entre los distintos namespaces, y de modo que cada namespace individual tiene su propia jerarquía de ficheros y puntos de montaje.

3.4.1.5.USER namespace

Es la funcionalidad que proporciona el aislamiento de los identificadores de usuario (PID) y grupos (GID) de los namespaces de modo que cada namespace tiene sus propios usuarios y grupos con un sistema específico de privilegios. Al igual que en el caso de los PID namespaces permite que se den distintos UID y GID sobre el mismo sistema operativo ya que cada uno de ellos pertenecerá a su propio namespace donde estará confinado.

3.4.1.6.IPC (Interprocess Communication [22]) namespace

22 http://www.science.unitn.it/~fiorella/guidelinux/tlk/node56.html

Es la funcionalidad que proporciona el aislamiento en la cola de mensajes de comunicación entre procesos, semáforos y memoria compartida. Al igual que en namespaces anteriores, proporciona un ámbito aislado para la base de comunicación de procesos y sus objetos compartidos (System V IPC[23]) dentro de cada namespace particular.

[wiki7]

3.4.2. Cgroups

Los Cgroups[24] (abreviatura de Control Groups) o grupos de control son una funcionalidad del Kernel de Linux (Soportada de forma completa a partir de la versión 2.6.24) que permite gestionar el control y limitar el uso de determinados recursos hardware para un grupo de procesos específico. Los recursos hardware que permite controlar Cgroups son CPU, Memoria, entrada/salida de disco y red entre otros. Al igual que sucedían con los namespaces, cada proceso de un contenedor pertenece a un Cgroup único, lo que lo dota de abstracción y control a nivel hardware.
Los Cgroups se organizan de forma jerárquica siguiendo una estructura tipo árbol, del mismo modo que se organizan los procesos del sistema operativo, por lo que todos los Cgroups hijos dependen de un Cgroup padre del que heredan atributos (Igual que el sistema de procesos del sistema operativo Linux) y límites definidos por este Cgroup padre. Del mismo modo, pueden combinarse en nuevos grupos y subgrupos siguiendo un sistema jerárquico tipo árbol.
De esta manera, los Cgroups son una excelente herramienta para controlar la asignación de los recursos hardware con una elevada granularidad y combinabilidad a nivel de recursos hardware.
Existen dos versiones de Cgroups: Cgroups1 y Cgroups2. Una es evolución de la otra y los Kernel más actuales ya traen soporte Cgroups2. A diferencia de Cgroups1, Cgroups2 tiene sólo una jerarquía de procesos única y discrimina entre procesos (no hilos).

[kernel1]

3.4.3. Netlink

Netlink es una funcionalidad del Kernel de Linux que dota a los sistemas de virtualización de contenedores de un mecanismo de gestión y aislamiento a nivel de socket, comunicación entre procesos y objetos compartidos (System V IPC[25]).

3.4.4. Netfilter

Netfilter[26] es un funcionalidad del Kernel de Linux que dota a los sistemas de virtualización de contenedores de un mecanismo de aislamiento a nivel de red IPV4 e IPV6 gestionando a modo de firewall en filtrado de paquetes y tráfico de red entre los contenedores.

3.4.5. SELinux

SELinux[27] (Acrónimo de Security-Enhanced Linux) es una funcionalidad del Kernel de Linux que dota a los sistemas de virtualización de contenedores de un mecanismo de políticas de seguridad de control de acceso obligatorio, conocido también como MAC, (incluyendo todos los procesos que se ejecutan con el nivel máximo de privilegios de usuario(root)), permitiendo la aplicación de políticas de seguridad directamente a los contenedores. Funciona de una manera similar a Capabilities pero aporta nuevas políticas de seguridad y además refuerza el control del host anfitrión frente a determinadas acciones privilegiadas ejecutadas desde los contenedores.

23 http://www.science.unitn.it/~fiorella/guidelinux/tlk/node56.html
24 https://en.wikipedia.org/wiki/Cgroups [wiki8]
25 http://www.science.unitn.it/~fiorella/guidelinux/tlk/node56.html
26 https://www.netfilter.org/
27 https://es.wikipedia.org/wiki/SELinux [wiki8]

También permite controlar otros aspectos como el acceso a los puertos de red que pueden exponer los contenedores.

3.4.6. Capabilities

Capabilities es una funcionalidad que dota a los sistemas de virtualización de contenedores de una capa de seguridad adicional ya que permite controlar y restringir las operaciones privilegiadas que pueden ejecutar los procesos que se ejecutan dentro de los contenedores.
Existen un gran nuevo de operaciones privilegiadas que se pueden restringir y que se identifican como “capability keys”, que según se establezcan o no pueden controlar permisos como el control del acceso y establecimiento de la hora del sistema (Capability Key: SYS_TIME), control de operaciones sobre la red (Capability Key: NET_ADMIN), control del envío de señales dentro de un contenedor (Capability Key: KILL), entre otros ejemplos.
Todas las operaciones privilegiadas que pueden gestionarse usando capabilities pueden consultarse en el siguiente enlace[28] a pie de página.

Ilustración 7: Vista de las capas que recubren un contenedor

3.5. Ventajas e inconvenientes

A continuación vamos a señalar de forma detallada las ventajas e inconvenientes que podemos encontrar en la tecnología de virtualización de contenedores.

3.5.1. Ventajas:

1. Mínimo consumo de recursos:
   Es el sistema de virtualización que consume un menor número de recursos hardware, los cuales revierten directamente en la aplicación ejecutada sobre el contenedor, por lo además es el que aporta mayor productividad.
2. Máximo aprovechamiento del hardware:
   Permite un aprovechamiento máximo de los recursos hardware que ofrece la máquina.
3. Mayor densidad de servicios virtualizados por anfitrión:Al aprovechar mejor los recursos de la máquina y consumir un menor número de recursos, permite una mayor densidad de equipos ejecutándose sobre el mismo hardware.
4. Mayor agilidad en despliegue de aplicaciones:
   Los contenedores y las aplicaciones que albergan arrancan en segundos, lo que permite una mayor agilidad en el despliegue de nuevas aplicaciones a más instancias de la misma.
5. Dimensionamiento escalable:
   Al tener un consumo muy ajustado de recursos y permitir despliegues rápidos, dotan a este sistema de una arquitectura perfecta a la hora de crear un sistema de dimensionamiento escalable.

28 https://linux.die.net/man/7/capabilities [wiki9]

6. Compatibilidad multisistema:
   Al crear contenedores de servicios aislados, hacen que los servicios que albergan puedan ser desplegados en cualquier sistema de contenedores generando un elevado grado de compatibilidad.
7. Portabilidad:
   Del mismo modo que en el caso de la compatibilidad, con los sistemas de virtualización de contenedores aumenta la portabilidad de los servicios que albergan los mismos, de modo que podrán ejecutarse sin problemas en cualquier sistema a donde portemos el contenedor.
8. Multitud de servicios disponibles para desplegar:
   Existe un gran número de servicios y aplicaciones preparados, y mantenidos con sus respectivos fabricantes, disponibles en formato contenedor para desplegarse, lo que permite el arranque de nuevos servicios que pueden ser usados de forma casi instantánea.
9. Simplicidad en la operación:
   Al ser un sistema ligero y poder tener una mayor densidad de servicios virtuales por anfitrión físico, permite simplificar la operación del sistema. Además las aplicaciones de virtualización de contenedores incluyen de forma nativa un gran número de recursos para facilitar la operación de los contenedores y los servicios que albergan.
10. Facilita la automatización:
    El software de gestión de contenedores incluye aplicaciones específicas y herramientas orientadas a la automatización de tareas y servicios dentro de los contenedores del anfitrión.
11. Ahorro de costes:
    De las ventajas anteriormente descritas se deduce un ahorro claro de costes tanto a nivel de hardware como de operación y mantenimiento de los servicios albergados dentro de los contenedores.

3.5.2. Inconvenientes:

1. Aumenta los riesgos de brechas de seguridad:
   Como tratamos con un sistema donde podemos tener un sistema muy heterogéneo de aplicaciones en los distintos contenedores, se corre el riesgo de que una brecha de seguridad en un contenedor especifico dentro de un sistema anfitrión pueda dejar expuesto a riegos tanto al mismo anfitrión como al resto de contenedores vecinos dentro del mismo SO. Al aumentar la densidad de contenedores que un anfitrión puede alojar, el riesgo de una brecha en un contenedor puede aumentar significativamente y del mismo modo, provoca que esta pueda afectar a más sistemas. La mayor parte de los esfuerzos en la tecnología de contenedores va enfocada a aislar y securizar cada vez más los contenedores mediante el uso de múltiples herramientas o librerías con el fin de eliminar este problema.
2. Menor flexibilidad a nivel de sistema operativo del contenedor:
   Los sistemas de las imágenes de los contenedores virtualizados sólo pueden ser basados en versiones relativamente recientes del sistema operativo Linux, por lo que perdemos flexibilidad a la hora de poder elegir el sistema operativo que se ejecuta dentro del contenedor.
3. Posibles bugs por modelo de desarrollo de software en constante evolución:
   Existen actualmente varias herramientas de virtualización de contenedores de reciente aparición, que continuamente están siendo actualizadas para aumentar funcionalidades y con un desarrollo muy rápido lo que podría provocar la aparición de errores de código por no ser versiones completamente estables y probadas.

3.6. Análisis y comparativa de distintas herramientas de virtualización de contenedores del mercado

Existen varias soluciones de virtualización de contenedores usando software libre en el mercado. A continuación haremos una breve introducción de las opciones más destacables, y nos decantaremos por una para implementarla en este proyecto indicando los motivos que nos han llevado a eligirla frente al resto y las ventajas que nos aporta para este proyecto en concreto.

3.6.1. LXC

LXC[29] (Acrónimo en inglés de LinuX Containers) es una tecnología de virtualización en el nivel del sistema operativo que permite que a partir de servidores físicos se puedan ejecutar múltiples instancias de sistemas operativos aislados (Conocidos como VPS, acrónimo de servidores virtuales privados) o entornos virtuales aislados (EV acrónimo de Entornos Virtuales) / contenedores.
LXC surge como una mejora y aumento de funcionalidades de los primeros sistemas de virtualización de entornos como «FreBSD jail» o «Solaris Containers». Basa su funcionamiento en el uso como base de herramientas de soporte de aislamiento de procesos que incluye el propio Kernel de Linux, las cuales son fundamentalmente Cgroup y Namespaces (Comentadas ya en capítulos anteriores), y sobre las que se van añadiendo funcionalidades, nuevas herramientas, plantillas y librerías que van dotando al sistema de un conjunto de nuevas funcionalidades para la administración y control de los contenedores virtuales.
LXC es un sistema maduro y estable con más de 5 años en el mercado desde las primeras apariciones de la versión LXC 1.0, por lo que es un software que está perfectamente testeado tanto a nivel de problemas de seguridad como ajustado al máximo a nivel de rendimiento de los recursos hardware empleados.
LXC es una tecnología que como hemos indicado se basa en funcionalidades básicas del Kernel de Linux, por tanto de forma nativa sólo puede ser ejecutada en sistemas operativos Linux. Es necesario aclarar que también podemos ejecutarlo en otros sistemas operativos, como puede ser Mac o Windows, pero siempre como resultado de ejecutarlos embebidos dentro de un entorno Linux que se virtualiza de forma clásica sobre un anfitrión con otro sistema operativo. Por este motivo, se considera que no es una ejecución nativa ya que es necesario virtualizar primero el sistema operativo Linux para poder, dentro de este, virtualizar contenedores.
Este sistema, esta enfocado fundamentalmente a la virtualización de sistemas operativos completos dentro de los contenedores de modo que cada uno incluye su propia estructura de sistema de servicios de arranque, logs, procesos, etc de modo que la estructura interna del contenedor y su funcionamiento es muy similar al de un sistema operativo completo pero con el aislamiento que proporciona este tipo de virtualización.
LXC es también uno de los primeros sistemas completos de virtualización de contenedores en aparecer en el mercado, y como aparece en base a añadir funcionalidades sobre otros sistemas ya existentes, fue usado también como tecnología base para otros sistemas de contenedores que aparecieron posteriormente, por lo que podemos considerarlo como el principal precursor.

Ilustración 8: Logotipo LXC

29 https://es.wikipedia.org/wiki/LXC

Actualmente LXC esta auspiciado por el proyecto linuxcontainers.org[30], el cual a su vez está mantenido por la empresa Canonical la cual ostenta una importante posición en el mercado por ser la responsable del sistema operativo Ubuntu el cual es un sistema operativo Linux que es uno de los referentes del mercado tanto de servidor como de sistemas operativos de escritorio que más éxito y comunidad de usuarios está experimentando. El respaldo de esta empresa en su desarrollo supone una garantía de continuidad y futuro del mismo.

3.6.2. OpenVZ

OpenVZ[31] es otro sistema de software libre que se ofrece como solución de virtualización de sistema operativo para servidores privados virtuales o para soluciones de entornos virtuales o contenedores.
Este sistema de basa en el uso de un Kernel de Linux modificado especialmente y de un conjunto de utilidades de usuario que se combinan para proveer un sistema robusto y sencillo de manejar para el usuario.
Se caracteriza por ser un sistema ligero y muy optimizado de cara a un bajo consumo de recursos. Quizás su característica más distintiva es el uso de un Kernel propio, sea una de sus mayores limitaciones ya que ha provocado que su desarrollo y la aparición de nuevas funcionalidades sea más lenta que otras tecnologías que usan un Kernel estándar. Esto también limita los sistemas operativos anfitriones distintos donde puede ser instalado más que en el caso de otras herramientas de virtualización que no dependen de un Kernel realizado a medida.
OpenVZ es un software libre bajo la licencia GPL 2 desarrollado y mantenido por la empresa SWsoft, Inc y que también se vende como solución comercial dentro de un entorno de aplicaciones denominado Virtuozzo.

Ilustración 9: Logotipo OpenVZ

3.6.3. Docker

Docker[32] es otro sistema de virtualización de contenedores a nivel de SO, que tal y como vimos en las soluciones anteriores permite que a partir de servidores físicos se puedan ejecutar múltiples instancias de sistemas operativos aislados (Conocidos como VPS, acrónimo de servidores virtuales privados) o entornos virtuales aislados (EV, acrónimo de Entornos Virtuales) / contenedores. En el caso de Docker, a diferencia de las soluciones anteriores, el sistema se centra más en la virtualización de entornos (VE) y contenedores, por lo que esta más enfocado a la automatización en el despliegue de aplicaciones embebidas en estos contenedores software dotando, si cabe, a este sistema de una capa adicional de abstracción y automatización de virtualización a nivel de sistema operativo que otras soluciones. Por este motivo, podemos decir que el sistema Docker esta más enfocado y optimizado para la ejecución de una aplicación embebida dentro de un contenedor aunque también podemos embeber en el contenedor un sistema operativo completo como en otras soluciones.
A nivel de arquitectura y componentes, la solución Docker funciona de una manera muy similar a las otras soluciones vistas anteriormente, por lo que para proporcionar las características básicas de aislamiento y virtualización emplea las tecnologías que proporciona el Kernel de Linux:

30 https://linuxcontainers.org/
31 https://openvz.org/
32 https://www.docker.com/

Namespaces para proporcionar aislamiento de procesos, identificadores de usuarios y grupos, sistemas de montaje de ficheros, etc entre los distintos contenedores. También emplea Cgroups para gestionar el aislamiento de recursos hardware como red, Cpu y memoria entre otros. Además de estas funcionalidades que proporciona el Kernel también podemos destacar el uso de Capabilities, SELinux, AppArmor, Netlink y Netfilter entre otros. Docker accede a la virtualización del Kernel, y a estas funcionalidades a través de una serie de drivers o librerías especificas. Hasta la versión 1.8, Docker empleaba directamente el driver de LXC para gestionar la interacción con las herramientas del Kernel pero apartir de esta versión, y continúa en la actualidad, han pasado a usar un conjunto de librerías denominado libcontainer.
Al igual que los otros sistemas de contenedores que hemos analizado, Docker sólo puede ejecutar de modo nativo bajo un sistema operativo Linux (ya que como hemos visto se basa en funcionalidades nativas del Kernel de Linux para funcionar). Es un sistema que permite una enorme portabilidad ya que, debido al enorme auge que ha experimentado, la mayor parte de las grandes empresas comerciales de tecnologías de infraestructuras orientadas al cloud computing permiten ejecutar contenedores Docker de forma nativa desde sus plataformas cloud. Destacamos el caso de Google Cloud, Amazon AWS y Microsoft Azure que tienen plataformas nativas especializadas en la ejecución de contenedores Docker con múltiples funcionalidades y tipos de escalabilidad y alta disponibilidad.
Auspiciado por el éxito de las soluciones Docker, podemos encontrar en el mercado un gran numero de aplicaciones comerciales en formato contenedor Docker para ser usadas y arrancadas sin necesidad de conocimientos previos. Del mismo modo, podemos encontrar todo tipo de servicios que se distribuyen preparados para ser usados en formato contenedor Docker, lo que nos permite acceder a desplegar una enorme cantidad de aplicaciones y servicios sin prácticamente necesidad de un conocimiento previo de los pasos de su instalación, y lo que es más importante, con el soporte directo del fabricante de la solución.
Podemos ver a Docker además de como una herramienta para la creación de contenedores virtuales, como una capa más de abstracción y aislamiento que además aporta un sistema que facilita enormemente la gestión y administración de los contenedores.

Ilustración 10: Logotipo Docker

Además de la propia herramienta principal de Docker para el control y gestión de los contenedores, existe un importante ecosistema de aplicaciones y utilidades integrado en el propia Docker que permite dotar a esta solución de un gran numero de herramientas y funcionalidades adicionales que aportan distintas soluciones de valor. Docker Machine, Docker Swarm y Docker Compose son tres de las más salientables y que combinadas, permiten que los contenedores sean más portables y escalables de tal forma que pueden ser más fácilmente desplegados y administrados en conjunto por lo que combinadas permiten la orquestación de sistemas a gran escala. Así, todo el ecosistema de aplicaciones que aporta la solución Docker van orientadas a dotar de un mayor numero de funcionalidades y herramientas de gestión y automatización para un sistema de aplicaciones embebidas en un contenedor virtual.
Podemos considerar que con Docker podemos realizar empaquetados de aplicaciones como si se tratase de una unidad estandarizada y estanca (contenedor) que incluya todo lo necesario para ejecutar esa aplicación (Código, librerías, bibliotecas del sistema, herramientas, etc) y que puede ser replicado de forma rápida, fiable y sistemática.

Conclusiones:

De las tres soluciones de código abierto para virtualización de contenedores que hemos estudiado, en primer lugar hemos descartado la solución OpenVZ que pese a ser una solución

ligera y bástamente probada en entornos de producción, es la que peor se adaptada a la solución que buscamos debido a que esta fundamentalmente orientada a la ejecución en contenedores de entornos virtuales, pero muy enfocada a ejecutar sistemas operativos completos dentro de contenedor de un modo muy similar a la virtualización tradicional de máquinas. También hemos tenido en cuenta que, pese a que no podemos decir que es una solución que esté en desuso, la irrupción de las otras soluciones tecnológicas como las estamos valorando han relegado a OpenVZ a un segundo plano y a un cada vez menor uso en entornos profesionales.
El siguiente candidato que hemos descartado es LXC, por cuestiones similares a la anterior solución. Pese a que también es un entorno muy ligero y ampliamente probado consideramos que puede no ser tan flexible para nuestra solución como puede ser el candidato elegido: Docker. Además de por la flexibilidad, LXC es un sistema más complejo de manejar que Docker por lo que la curva de aprendizaje, para conseguir similares resultados a nivel de virtualización es mucho más pronunciada y requiere un mayor nivel de conocimientos técnicos.
Otra característica para decantarnos por Docker es que este implementa de forma nativa en su versión más actual soporte para orquestación de aplicaciones, que es uno de los pilares que necesitamos desarrollar para nuestro proyecto y que con LXC sólo no podríamos conseguir y necesitaríamos recurrir a soluciones de terceros.
Finalmente los dos últimos puntos que nos han hecho decantarnos por Docker han sido: está muy orientado a virtualizar aplicaciones dentro de contenedores más que a virtualizar sistemas operativos completos, por lo que el nivel de eficiencia será mayor y está experimentando un auge enorme y los propios fabricantes de aplicaciones están ofreciendo las mismas, mantenidas en formato nativo Docker, por lo que podemos poner en marcha aplicaciones sin prácticamente necesidad de conocer su funcionamiento interno, por lo que las puestas en marcha y despliegues se reducen en coste y tiempo a la mínima expresión.
Este proyecto pretende poner en marcha un sistema de orquestación de aplicaciones a través del uso de contenedores usando un sistema robusto y de código abierto, por lo que la solución que hemos visto que mejor se adapta pasa por el uso de la herramienta Docker. Por ello, estudiaremos su arquitectura y funcionamiento en los siguientes capítulos.

3.7. Orquestación de aplicaciones

Existen múltiples definiciones sobre el concepto de orquestación de aplicaciones pero de un modo simple, podemos definir la orquestación de servicios o aplicaciones como el uso de la automatización para la creación y composición de la arquitectura, herramientas y procesos utilizados por operadores humanos para entregar un servicio.
La orquestación aprovecha tareas automatizadas y procesos predefinidos para permitir la creación de infraestructura complejas y para conseguir el aprovechamiento de los recursos de forma óptima y automatizada. Podemos considerar, a modo de analogía, el concepto de orquestación como un proceso y la automatización como una tarea.

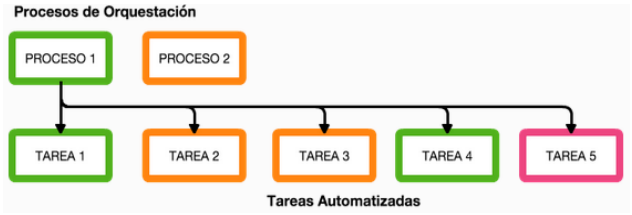


Ilustración 11: Orquestación de procesos

[orquestacion1]

De este modo, el objetivo principal de la orquestación consiste en la automatización de procesos orientados al despliegue y ciclo de vida de las aplicaciones o servicios. Y la automatización de procesos en los despliegues software se basan en el uso de algún tipo de software que facilite la

instalación, configuración y mantenimiento del servicio o aplicación con la mínima intervención humana.

[orquestacion2]

Así, la automatización en despliegue de aplicaciones y servicios debe permitirnos una gestión completa de los recursos e infraestructura, permitirnos dotar a nuestros servicios y aplicaciones de tolerancia a fallos y escalado dinámico, monitorización de recursos , aplicar un sistema de despliegue continuo, etc.

Existen dos tipos distintos de orquestación en base fundamentalmente a como se gestiona el escalado de recursos.

- Orquestación estática: El sistema requiere una configuración más manual de los recursos y no permite el escalado de forma muy eficiente.
- Orquestación Dinámica: Escala de forma sencilla y eficientes los recursos de la aplicaciones y servicios y el propio sistema toda ciertas decisiones de forma automática.

En este proyecto, se plantea una solución de orquestación con la herramienta Docker que, como veremos, incluye 2 herramientas para la orquestación de servicios y aplicaciones sobre contenedores virtuales. Una de ellas , **Docker Compose** que permite la orquestación estática orientada a un funcionamiento más centrado un solo servidor y **Docker Swarm** que permite la orquestación dinámica de servicios y aplicaciones sobre un cluster servidores.

En la orquestación nos encontramos con la necesidad de resolver procesos que para poder optimizar a un mayor nivel y escalar, deben estar formados por múltiples tareas más simples en lugar de tareas muy complejas.
Así, las arquitecturas óptimas para la orquestación y que por ese motivo suelen ir de la mano, son las arquitectura orientadas a los servicios (***SOA*** de sus siglas en Inglés Service Oriented Architecture) y que cada vez van más orientadas a distribuir las tareas en servicios más pequeños (***Microservicios***). Estás arquitecturas son un paradigma para el diseño y desarrollo de sistemas distribuidos que han sido creadas para satisfacer los objetivos de negocio que incluyen necesidades de facilidad y flexibilidad, lo que permite una mejor alineación con los procesos de negocio con las ventajas de eso conlleva.

4. Arquitectura base

Se propone la siguiente arquitectura como base para el despliegue de la solución del proyecto:

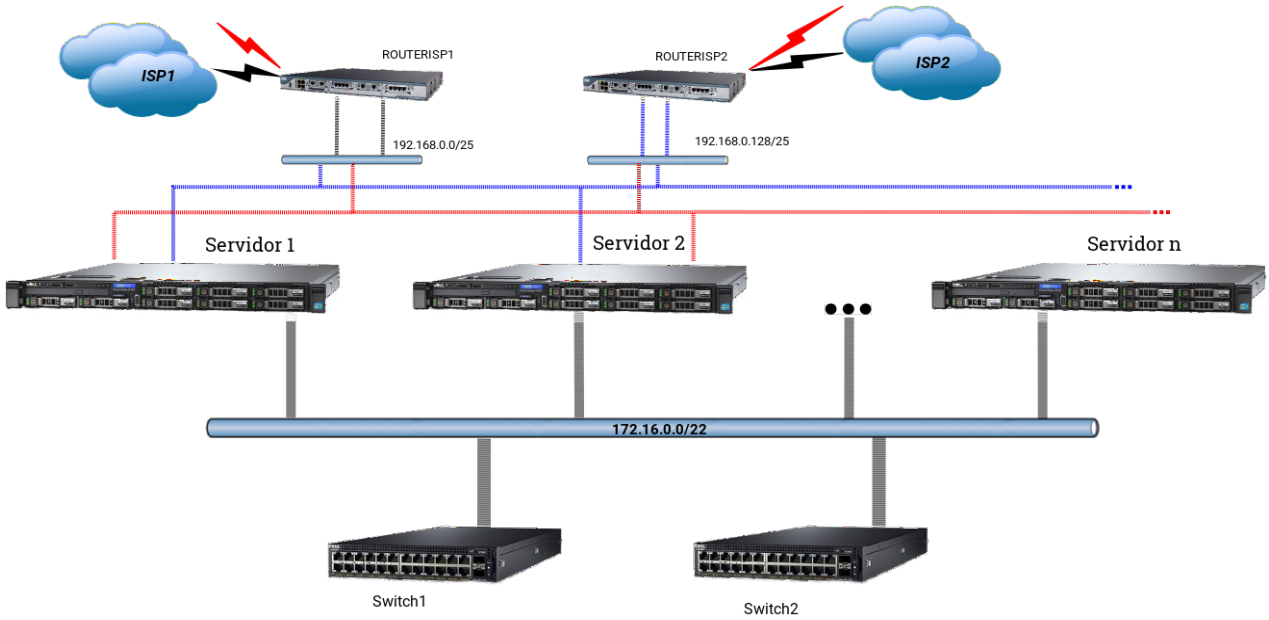


Ilustración 12: Vista Global de la arquitectura

Se ha realizado una arquitectura de base para la solución del proyecto respetando las directrices de los requisitos iniciales. Se trata de una arquitectura de cluster clásica con varios equipos para garantizar redundancia en todos los componentes que permiten dar servicio a la solución propuesta.
A nivel de hardware de cómputo, partimos de dos servidores físicos Dell PowerEdge R430, que son los equipos que albergaran el software de virtualización de contenedores además de otros servicios auxiliares que puedan ser necesarios. En este caso, partimos de dos porque es el número mínimo en el cluster para garantizar la disponibilidad de los servicios de contenedores, pero la arquitectura está diseñada para poder crecer dinámicamente simplemente con añadir nuevos servidores y conectarlos a la misma, por lo tanto, tenemos garantizada la escalabilidad horizontal de la arquitectura y de la solución propuesta.
A nivel de red, podemos ver claramente diferenciada la conectividad de red que proporciona acceso a Internet y que se realizará a través de al menos 2 ISP distintos para mantener la alta disponibilidad del servicio, y una definición de una red de área local privada que se usará para comunicar tanto los servidores anfitriones como los distintos contenedores de servicios que virtualizaremos.
A continuación profundizaremos en los distintos aspectos de la arquitectura separándolos en las distintas áreas lógicas:

4.1. Sistema operativo

Previamente a definir la arquitectura hardware para llevar a cabo el proyecto, y ahora que ya hemos aclarado que vamos a usar el software de virtualización de contenedores y orquestación Docker, debemos definir el sistema operativo que vamos a emplear en los servidores como sistema operativo anfitrión. La solución que proponemos está basada en el uso de software libre

por lo que a nivel de sistema operativo debemos de marcar esta restricción. También hemos visto que las soluciones de virtualización de contenedores que funcionan en modo nativo, usan todas ellas Linux como sistema operativo anfitrión.
Para ello, hemos probado la solución Docker que vamos a emplear para la orquestación en 2 tipos de sistemas operativos Linux distintos, para evaluar los distintos aspectos que podemos encontrarnos. Hemos elegido dos sistemas Linux el que hemos considerado más representativo de los 2 subtipos existentes de arquitecturas de paquetes (tipo RPM o DEB). Los sistemas probados han sido:

openSUSE 42.1

Ubuntu 16.04 Server LTS

Se ha probado el ecosistema de aplicaciones Docker en estos sistemas operativos ya que ambos son soportados por el fabricante mediante repositorios oficiales. Los dos sistemas operativos cumplen los objetivos iniciales se instalan de forma sencilla y funcionan sin problemas. También hemos de destacar que al poseer repositorios oficiales de Docker en ambas distribuciones las actualizaciones son sencillas de mantener ya que se gestionan directamente con el sistema propio de cada distribución.
Finalmente nos hemos decantado por el uso del **sistema operativo Ubuntu 16.04 Server** para los servidores anfitriones, porque los administradores IT de la empresa, como hemos indicado en el supuesto inicial están ya familiarizados con este sistema operativo en sus servidores, por lo que la curva de aprendizaje será mínima y además con ello nos garantizamos un mantenimiento más sencillo, unos menores costes de administración y conseguimos un menor nivel de rechazo en la implantación de este nuevo sistema.
También se tuvo en cuenta para tomar la decisión la duración del ciclo de vida de las versiones de Ubuntu, ya que al decantarnos por una versión LTS (Long Term Support) tenemos garantizado un soporte oficial de 5 años y la posibilidad de contratación de soporte técnico adicional a nivel de sistema operativo, de modo que con el sistema de soporte que ofrece Canonical (que es la empresa que está detrás de la distribución y soporte de Ubuntu) en caso de ser necesario sería posible contratar bonos de soporte específicos.

No forma parte del ámbito de este proyecto la documentación del proceso de instalación del sistema operativo Ubuntu 16.04 Server LTS, ya que es un proceso ampliamente documentado en el que no entraremos, ya que partiremos de que en ambos servidores tenemos el sistema operativo instalado y actualizado. Si bien, en los anexos del proyecto añadimos documentación sobre los pasos más importantes del proceso de instalación.

4.2. Arquitectura Hardware Base

A continuación detallamos por áreas funcionales los distintos apartados de la arquitectura propuesta.

4.2.1. Arquitectura de red

Como se puede ver en la ilustración general, en la arquitectura de red, se perfilan claramente diferenciadas dos redes redundantes para dar conectividad a la solución. Por un lado, tenemos

una red para dar conectividad con Internet y por otra lado una red local para la comunicación a los servidores y las redes empleadas los contenedores virtuales.
Ambas redes estarán presentes en todos los nodos del cluster que empleamos de modo que detallaremos individualmente el direccionamiento empleado y funcionalidad de cada una.
Descartar que los servidores Dell R430 elegidos traen de forma integrada 4 tomas de red, las cuales aprovecharemos al completo para distintas funcionalidades como veremos individualmente en cada apartado.

4.2.1.1.Internet

Para dotar al sistema de conectividad y acceso a Internet vamos a optar por un sistema con al menos 2 proveedores de Internet o ISP con una línea de al menos 100 Mbit/s simétricos cada uno. Hoy en día existen múltiples proveedores que ofrecen este servicio a un coste de línea residencial. Estas líneas se contratarán con distintos proveedores para garantizar la disponibilidad del servicio y conexión, de modo que nuestro servicio este garantizado por al menos 2 proveedores. En la medida de lo posible, sería recomendable considerar también la contratación de dos tecnologías distintas de acceso al medio por parte del proveedor, de modo que si por ejemplo un acceso es a través de fibra FTTH[33] el otro sería recomendable que fuese Docsis, ATM o a través que cualquier otra tecnología distinta de la FTTH, con el fin de diversificar tecnologías y evitar fallos comunes.
Adicionalmente cada proveedor nos entregará dos conexiones de red con un direccionamiento e ip pública propio, de modo de una sea la línea principal y la otra sea una línea de respaldo o backup que permanecerá inactiva y actuará en caso de fallo de la principal.
Identificar a cada proveedor con los nombres genéricos ISP1 e ISP2 y del mismo modo nombraremos los routers, siguiendo este mismo modelo ROUTERISP1 y ROUTERISP2 para poder identificarlos en las posteriores ilustraciones.
El esquema de conexionado sería el siguiente:

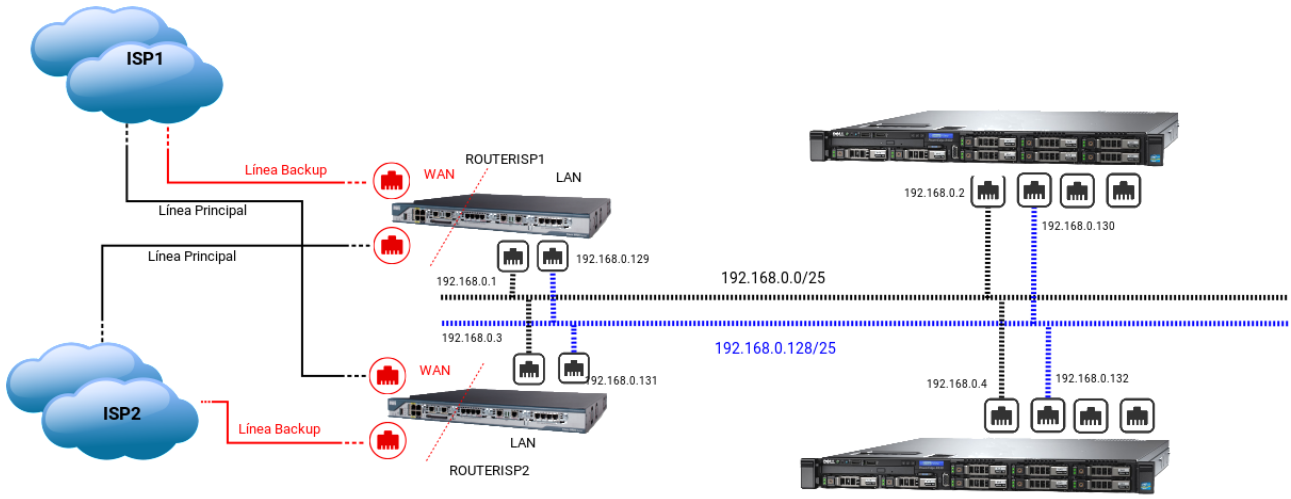


Ilustración 13: Arquitectura de Red - Internet

Como vemos, cada router tiene las dos líneas de conectividad y cada una proviene de un ISP, con una linea identificada como principal y otra de backup señalada del lado público del router identificado como WAN. De la parte interna del router, identificada esta vez como LAN, vamos a usar dos puertos ethernet del mismo los cuales van conectados cada uno a uno de los servidores de forma cruzada, como se puede ver en la ilustración anterior, para redundar la conexión y disponibilidad de los mismos.

33 https://es.wikipedia.org/wiki/Fibra_hasta_la_casa

No vamos a entrar en la configuración de la subred a nivel ip del lado público del router ya que esta consideramos que nos vendrá dada por el ISP y sólo nos ocuparemos de recibir una dirección ip pública balanceada por el interfaz primaria o por la de backup según corresponda.
Del lado interno del conexionado de cada router hacia los servidores vamos a usar también los dos interfaces de red y asignar dos subrangos de direcciones ,y mediante la técnica de subnetting, asignaremos una dirección ip al router y una a cada servidor nuevo que añadamos dentro de cada rango (Inicialmente en nuestro supuesto 2 servidores). Como se puede apreciar en la ilustración anterior la conexiones de red internas de cada servidor irán cruzadas, una a cada router de conexión a Internet para dotar el acceso a Internet de alta disponibilidad de modo que aunque uno de los routers fallase por completo o fuese necesario hacer una intervención para mantenimiento del mismo, el entorno permanecería completamente funcional y con acceso a Internet.
Si necesitásemos aprovisionar servidores adicionales, simplemente sería necesario añadirlos a la red conectándolos del mismo modo que los actuales servidores, conectando sus dos nuevas interfaces públicas a cada uno de los routers existentes.
Así, el direccionamiento del lado interno de cada router quedaría de la siguiente manera:

Dispositivo	Direccionamiento	Rango de direcciones ip
ROUTER ISP1	192.168.0.0/25	192.168.0.1 - 192.168.0.127
ROUTER ISP2	192.168.0.128/25	192.168.0.129 - 192.168.0.254

Un ejemplo de asignación de direcciones siguiendo esa distribución sería el siguiente:

Dispositivo	Dirección ip
ROUTER ISP1 - Interfaz 1	192.168.0.1
ROUTER ISP1 - Interfaz 2	192.168.0.129
SERVIDOR1 - Interfaz 1	192.168.0.2
SERVIDOR1 - Interfaz 2	192.168.0.130
ROUTER ISP2 - Interfaz 1	192.168.0.3
ROUTER ISP2 - Interfaz 2	192.168.0.131
SERVIDOR2 - Interfaz 1	192.168.0.4
SERVIDOR2 - Interfaz 2	192.168.0.132

A nivel de configuración, internamente en cada servidor configuraremos las direcciones ip de ambos routers como puerta de enlace utilizando distintas métricas para priorizar una de las salidas en cada servidor y para utilizar la siguiente en caso de fallo. También configuraremos los routers para permitir el acceso en sentido de Internet hacia los servidores a través de las ips públicas y para poder exponer los servicios en los contenedores virtuales desde Internet. Para ello, configuraremos una zona DMZ[34] (*demilitarized zone*) o zona desmilitarizada a nivel del router para redirigir todos los puertos de la ip pública a la ip privada de la red de área local de cada servidor.
Aunque no entraremos a definir la configuración, ya que no entra en el ámbito de este proyecto, es necesario destacar que se habilitará un firewall sobre cada uno de los servidores correctamente configurado para evitar exponer más servicios de los necesarios en los servidores realizando un bastionado de los mismos.
Con esta estructura hemos implementado un sistema que permitirá mantener conectividad con Internet considerando tanto la posibilidad de fallo de las línea del proveedor como de uno de los routers de salida a Internet por lo que dotamos el acceso a Internet de alta disponibilidad.

34 https://es.wikipedia.org/wiki/Zona_desmilitarizada_(inform%C3%A1tica)

4.2.1.2.Redes de Área Local y direccionamiento

Tal y como introducimos anteriormente, en esta arquitectura tenemos 2 zonas de red claramente delimitadas. En este punto vamos a detallar los aspectos más técnicos de la red privada.
La red privada, da soporte de comunicación de red interna a todos los contenedores que vamos a desplegar y a las redes virtuales que estos puedan necesitar, así como para la comunicación entre los servidores que forman parte del cluster para todo tipo de tareas administrativas y de gestión de los mismos.
Del mismo modo que para la conectividad pública diseñamos un entorno de alta disponibilidad, en el caso de la red privada este apartado es más importante de si cabe ya que todos los contenedores desplegados en forma de servicios necesitan comunicarse entre si para compartir información. De este modo, es un punto fundamental para garantizar una orquestación de servicios correcta.
Para ello planteamos la siguiente arquitectura y direccionamiento que mostramos en la siguiente ilustración:

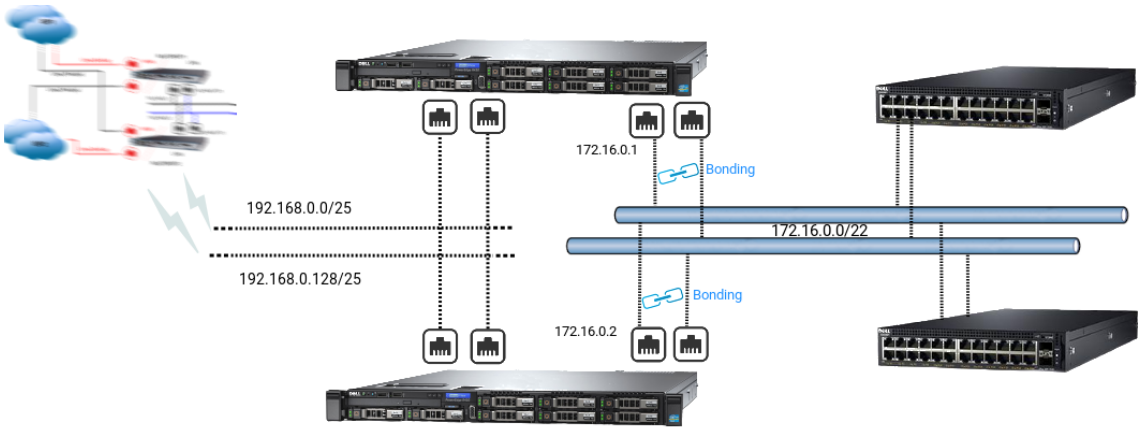


Ilustración 14: Arquitectura de la lan privada

Comenzamos por la instalación de dos switchs Dell Powerconnect X1026, que tal y como detallamos en los requisitos iniciales destaca por proporcionar 26 puertos de conectividad Gigabit. Por el numero puertos disponibles en el switch, podríamos incluir hasta 24 nuevos servidores (Sin contar los 2 servidores ya aprovisionados) por lo que reservamos bastante espacio para que el cluster de servidores pueda crecer horizontalmente sin necesidad preocuparnos a corto plazo.
De los 2 interfaces de red que quedan libres en cada servidor,después de la configuración de la conectividad a Internet, conectaremos uno de cada a cada uno de los switch anteriores de forma cruzada de modo que cada servidor tenga un puerto de red conectado a cada switch proporcionando redundancia de conectividad a nivel de este elemento de red.
Para complementarlo, vamos a implementar a nivel de configuración de sistema operativo una técnica denominada “*Bonding*”, la cual nos permitirá emplear de forma simultanea ambos interfaces de red de cada servidor para envío y recepción de tráfico de forma conjunta, además de dotarlo de tolerancia a fallos ya que el propio sistema gestiona el fallo de una interfaz de forma automática adaptando el tráfico a los interfaces enlazados en “*Bonding*”. El “*Bonding*” es un driver especifico del Kernel de Linux que tiene distintos modos de funcionamiento en función que como deseemos gestionar los interfaces de red enlazadas. En nuestro caso, usaremos el modo “*balanced-alb*” o “*modo 6*” que se basa en el balanceo del envío y recepción de tráfico mediante la manipulación automática por parte de este drivers de los replies de ARP.

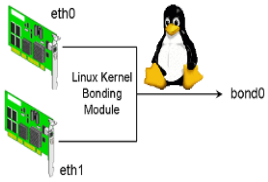


Ilustración 15: Estructura de Bonding

No vamos a especificar en este punto como sería paso a paso la configuración del "*Bonding*" de red y vamos a continuar suponiendo que ya tenemos este paso hecho en cada servidor y que contamos en cada uno con un interfaz "*bond0*" ya preconfigurado y funcionando en modo el modo "*balanced-alb*", aunque en los anexos incluiremos los pasos necesarios para su configuración.
De este modo, aunque uno de los switchs de la red fallase complemente o fuese necesario paralo por tareas de mantenimiento, el funcionamiento y comunicación de los nodos internos del cluster y los contenedores de la red permanecerían funcionando sin cortes.

A nivel de direccionamiento ip, definimos una subred partiendo de un subred clase B segmentada con subnetting de 22 bits, con posibilidad para 1022 hosts para dar cabida a todos los contenedores que podamos necesitar. Como hemos visto, de partida vamos a aprovisionar 2 servidores con 64 Gb de memoria RAM por lo que podremos dar cabida a un gran número de contenedores virtuales publicados en la misma red que los servidores anfitrión. Por ese motivo y para poder aumentar el cluster vamos a reservar este rango de red:

Direccionamiento Clase B	Direcciones IP		Número de Hosts
	Desde	Hasta	
172.16.0.0/22	172.16.0.0	172.16.3.254	1022

Adicionalmente, vamos a reservar las primeras 32 direcciones ip para los 26 posibles servidores que podemos conectar en el switch, empezando por la 172.16.0.1 y 172.16.0.2 para nuestros 2 servidores iniciales (Como se puede apreciar, a pesar de ser dos interfaces físicos comparten una misma ip debido al funcionamiento de este modo de Bonding, donde las interfaces funcionan como un único interfaz de red).

4.2.2. Almacenamiento

A nivel de almacenamiento, vamos a utilizar la capacidad interna de cada servidor para albergar los datos necesarios para el funcionamiento de todo el entorno.
Si posteriormente se necesitara más espacio de almacenamiento, se podría recurrir a volúmenes de almacenamiento en red, como por ejemplo un sistema de discos iSCSI[35] , que conectaríamos individualmente a cada servidor.
A nivel interno de cada servidor, partimos de un almacenamiento de 3 discos SSD de 400GB de capacidad por cada servidor. Estos se configurarán en RAID 5 para garantizar tolerancia a fallos.
El RAID5 o RAID de distribución de paridad proporciona una división de los datos a nivel de bloques que distribuyen la información de paridad entre los discos miembros y que se implementa con un número mínimo de 3 discos (como es el caso), y que en contraprestación nos obliga a perder el espacio de un disco en almacenamiento de información de paridad por lo que tendremos disponible para su uso, unicamente la suma del espacio de los otros dos discos.
En nuestro particular, nos permite disponer de un espacio libre de 800GB.

35 https://es.wikipedia.org/wiki/ISCSI

El modelo de servidores elegido, trae de serie un tarjeta PERC H710 que soporta este modo de RAID de forma nativa vía hardware. Además no vamos a entrar en la configuración del RAID ya que, para el modelo podemos solicitarlo preconfigurado en RAID5 de fabrica.

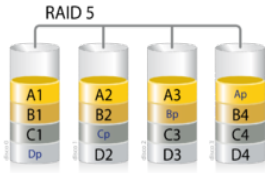


Ilustración 16: Estructura RAID5

Como la configuración de RAID es hardware, a nivel de sistema operativo durante la instalación del mismo, veremos una única unidad de 800GB que particionaremos, siguiendo la técnica de separar una partición para sistema operativo (Creando subvolumenes Btrfs para separar las carpetas "/var", "/tmp", "/usr/src", "/home", etc... como es habitual en las buenas prácticas de la instalación de los sistemas operativos Linux[36]), memoria de intercambio (Que se ha definido con un tamaño de 8G porque tenemos bastante espacio pero que por la cantidad de memoria RAM no debería de necesitarse nunca más de 4GB), y otra para los datos que usará posteriormente Docker (Ruta /var/lib/docker). De modo que tendremos una separación lógica de sistema operativo y datos.
Finalmente, detallar que a nivel de sistema de ficheros emplearemos el sistema de ficheros **Btrfs**[37] para las particiones tanto del sistema operativo como de los datos. Btrfs es un sistema nativo y soportado en la distribución Ubuntu 16.04 que usaremos en los equipos anfitriones, que aunque no viene seleccionada como formato de partición por defecto, si la usaremos porque aporta a la aplicación Docker una serie de funcionalidades (copy-on-write o instantáneas de volúmenes por ejemplo) que son aprovechadas por obtener una mejora de rendimiento y una mejora en la gestión del sistema de contenedores. No es necesario ninguna configuración adicional ya que si Docker detecta que es btrfs el sistema de ficheros donde esta instalado (/var/lib/docker) automáticamente hace uso del driver para btrfs por defecto.
Además, como usamos un sistema de ficheros Btrfs, no es necesario preocuparse por los tamaños de las particiones ya que con este sistema podemos redimensionarlas posteriormente si fuese necesario.

La estructura quería como se detalla a continuación:

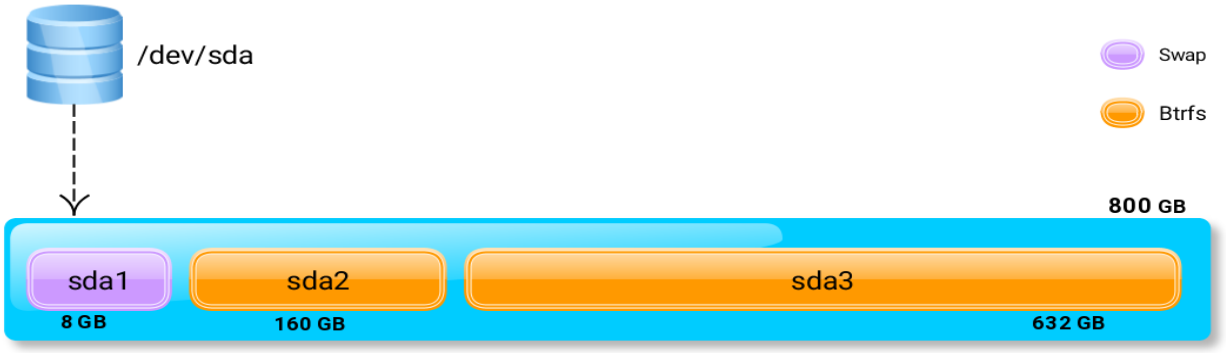


Ilustración 17: Particionado de disco

36 http://sobrebits.com/buenas-practicas-en-el-particionado-de-gnulinux-parte-2-avanzado/ [prt1]
37 https://es.wikipedia.org/wiki/Btrfs

5. Docker

[libro1] [libro2][libro3][libro4][libro5]

En capítulos anteriores hemos introducido la herramienta Docker como la solución de virtualización de contenedores que vamos a utilizar así como la arquitectura de servidores, red y almacenamiento sobre la que se implantará la solución.
En este capitulo vamos a ver los principales aspectos de Docker, su funcionamiento, modelos de uso, pasos de instalación y configuración así como los primeros ejemplos de despliegue de aplicaciones sobre esta solución y la herramienta de orquestación estática que incluye Docker.

5.1. Introducción a la arquitectura Docker

Docker es una aplicación que sigue el modelo cliente-servidor. Así, a nivel de arquitectura de la aplicación tenemos fundamentalmente una parte de servidor denominada “Docker Server” y una aplicación cliente, denominada “Docker Client” que hace uso de funcionalidades de la parte servidor, fundamentalmente a través de un cliente en línea de comandos. El modo servidor de Docker es el que se encarga de toda la gestión, mantenimiento y creación de los contenedores y su estructura de funcionamiento. La aplicación es única para ambos modos, sólo que al ejecutarla se indica si lo que queremos arrancar es el modo cliente o el modo servidor.
Existe también un tercer componente denominado, Docker Registry. Se trata de un repositorio donde se albergan las imágenes de los contenedores virtuales a emplear.
Mediante el cliente Docker se envían ordenes al servidor Docker que es el que gestiona el funcionamiento del sistema base de contenedores y este a su vez, mediante el uso de la librería libcontainer interactúa directamente con las distintas utilidades del Kernel para acceder a las utilidades de virtualización. Aunque libcontainer es actualmente el método nativo de interacción con el Kernel, Docker puede usar otras librerías o interfaces intermedias, como pueden ser libvirt o LXC.

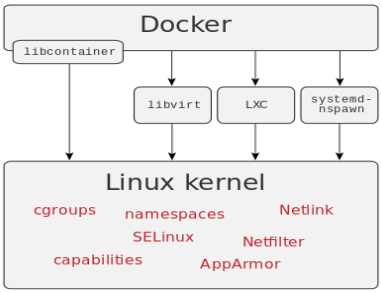


Ilustración 18: Esquema de comunicación Docker - Kernel

5.2. Modelos de uso e instalación de Docker

Una de las funcionalidades más importantes de la solución Docker es el hecho de que podemos optar por varias modalidades para el despliegue de nuestra arquitectura de contenedores, siendo compatibles entre sí. Podemos optar por un sistema de despliegue en **modalidad de uso tipo cloud**, a través del cual nos abstraemos completamente de la arquitectura hardware sobre la que instalamos nuestra solución y no tenemos que preocuparnos más que de facilitar el contenedor a desplegar. Para emplear este modo podemos contratar directamente el servicio con múltiples proveedores (Amazon AWS, Azure o el propio Cloud de Docker) y sólo debemos de facilitar los contenedores a ejecutar.

El otro modo de uso, que según los requisitos iniciales planteados para este proyecto, es el modo que usaremos para este proyecto, consiste en una **modalidad de uso local** también conocida como **modalidad "on promise"**, por lo que para ello el siguiente paso será proceder a la instalación del entorno Docker sobre la infraestructura hardware y arquitectura propuesta usando como base el sistema operativo Ubuntu 16.04 LTS que hemos seleccionado.
Es importante señalar, antes de empezar con los pasos de la instalación, que ambos modos son compatibles por lo que un contenedor o incluso un sistema completo de aplicaciónes con varios contenedores podría portarse de una modalidad a otra de forma completamente transparente. Esto nos permite un grado muy alto de compatibilidad y portabilidad de las aplicaciones desplegadas de una a otra modalidad con todas las ventajas que ello supone.
Procedemos a continuación con los pasos de instalación del componente principal del sistema Docker, conocido como Docker Engine:

1. Desde la consola del sistema operativo, ejecutamos los comandos que mostramos a continuación para actualizar la información de los paquetes de los repositorios oficiales de Ubuntu y nos aseguramos de que tengamos los paquetes mínimos para esta primera etapa:

```
sudo apt update
sudo apt install apt-transport-https ca-certificates
```

2. Añadimos la clave GPG oficial del proveedor Docker a la lista de claves conocidas (ID: 58118E89F3A912897C070ADBF76221572C52609D):

```
sudo apt-key adv \
               --keyserver hkp://ha.pool.sks-keyservers.net:80 \
               --recv-keys 58118E89F3A912897C070ADBF76221572C52609D
```

3. Añadimos el repositorio de Docker para nuestro sistema operativo y actualizamos la lista de paquetes:

```
echo "deb https://apt.dockerproject.org/repo ubuntu-xenial main" \
| sudo tee /etc/apt/sources.list.d/docker.list

sudo apt update
```

4. Instalamos la última versión de Docker:

```
sudo apt-get install docker-engine
```

5. Arrancamos el servicio y lo habilitamos para que arranque de forma automática a partir de este momento en el arranque del sistema:

```
sudo systemctl enable docker
sudo systemctl start docker
```

En cuatro sencillos pasos ya tenemos el sistema de virtualización de contenedores disponible para trabajar, podemos comprobar su funcionamiento y ejecutar directamente nuestro primer contenedor (se trata de un contenedor disponible para pruebas que muestra el habitual mensaje "Hello World") simplemente con el comando:

```
sudo docker run hello-world –name prueba
```

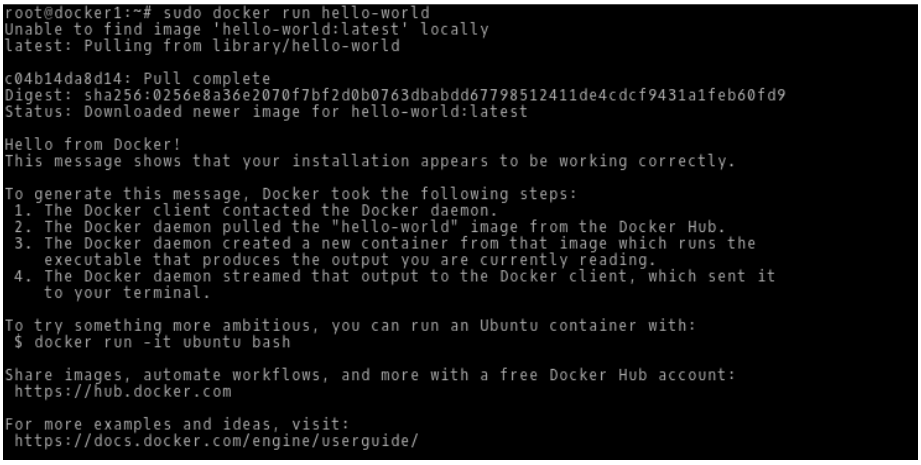


Ilustración 19: Ejecución del contenedor hello-world

Como vemos, los fabricantes de la solución Docker facilitan un repositorio propio de la herramientas Docker para el sistema operativo que hemos elegido donde mantienen la aplicación actualizada, por lo que podemos concluir que es un proceso sumamente sencillo.

5.3. Componentes y funcionamiento

Para poder entender como funciona internamente Docker, tenemos que ver previamente una serie de conceptos básicos y que a partir de este capitulo veremos muy habitualmente.

5.3.1. Imágenes de Docker (Docker Images)

Las imágenes de Docker son plantillas del sistema de ficheros completo que vamos a ejecutar. Estas imágenes son las copias en modo lectura de los datos iniciales que serán la base para los contenedores. Podríamos asimilarlo en virtualización clásica como una instantánea de una máquina virtual.
Existen de forma tanto pública como privada, múltiples imágenes de aplicaciones y servicios mantenidas por sus propios fabricantes (a modo de sistema de distribución estandarizado) ya preparadas para ser usadas en cualquier sistema de Docker en el cual pueden ser descargadas. Así, podremos encontrar este tipo de plantillas con aplicaciones ya preinstaladas como servidores web apache, servidores de bases de datos mysql, etc listas para ser usadas o modificadas por nosotros mismos para generar nuevas imágenes de Docker.

5.3.2. Registros de Docker (Docker Registries)

Los registros de Docker son servicios web donde se alojan las imágenes de Docker y funcionan a modo de centros de distribución. Pueden ser públicos y por lo tanto accesibles de forma libre o pueden ser privados y requerir un registro para acceder a los mismos. Como albergan imágenes de aplicaciones podemos verlos en ocasiones referenciados como repositorios, haciendo una analogía a los registros donde se almacena software.
Podemos usar también los registros públicos que ya facilita el propio fabricante Docker (Existe un plan completo de precios en base a distintos tipos de necesidades que puede ser contratado) o incluso es posible implantar un registro propio.
En nuestro caso, vamos a hacer uso del sistema de Docker donde se encuentran públicas todas la imágenes de servicios que vamos a emplear a modo de ejemplo.

5.3.3. Contenedores de Docker (Docker Containers)

Finalmente, nos encontramos el concepto de contenedor el cual, en Docker, podemos verlo como una instancia de una imagen que está en ejecución.
En analogía con la virtualización clásica, un contenedor es una máquina virtual en ejecución.
En el ejemplo anterior, con el que se ha probado que la instalación de Docker era correcta, estábamos ejecutando un nuevo contenedor llamado "*prueba*" a partir de una imagen del repositorio público de Docker denominada "*hello-world*".

```
sudo docker run hello-world –name prueba
```

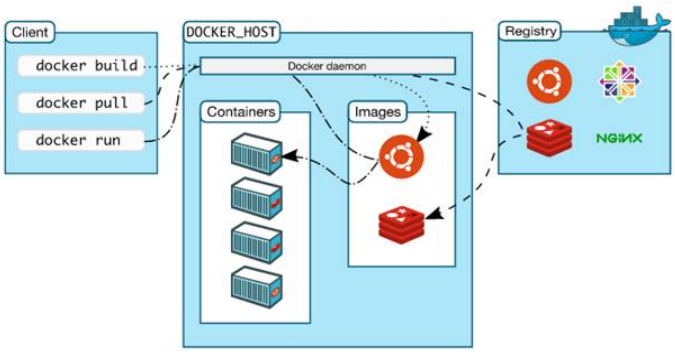


Ilustración 20: Modelo Docker

5.4. Arquitectura Docker

Una vez que ya tenemos el sistema Docker instalado, que hemos fijado los conceptos de imágenes, registros y contenedores y que hemos ejecutado las primeras pruebas, debemos de continuar viendo y configurando una serie de aspectos para adaptar la arquitectura de Docker a la arquitectura base hardware que hemos planteado como solución.

5.4.1. CPU y Memoria

A nivel de CPU y Memoria, la virtualización de contenedores permite aislar cada uno de ellos asignando individualmente los recursos necesarios a cada contenedor.
En la arquitectura propuesta se cuenta con 2 servidores con 64GB de RAM y 16 núcleos físicos de Cpu (32 Threads). Tenemos la capacidad de asignación individual de 64GB de RAM y 16 núcleos de CPU para compartir entre los contenedores que ejecutamos en cada servidor.
Si no indicamos límites, cada contenedor que se ejecute dispondrá para su uso de todos los recursos de la máquina que estén disponibles. No todos los contenedores o aplicaciones que ejecutemos tienen la misma necesidad de recursos a nivel tanto de CPU como de memoria, por lo que seremos nosotros los que definamos las necesidades específicas de recursos que tendrá cada contenedor cada vez que lo arranquemos y definamos un nuevo servicio.
Ya hemos visto como se ejecutan los contenedores en los ejemplos anteriores por lo que veremos ahora como asignarle una limitación en cuanto a recursos de CPU y memoria.
CPU:
Existen varias formas de limitar la CPU a través del arranque de un contenedor. Los modos principales son:

1. ***--cpu-share***=””. Permite asignar un valor entre 0 y 1024 que representa un porcentaje a modo de peso de recursos de CPU. Por ejemplo: --cpu-share="512" para ejecutar un contenedor con el 50% de los recursos de cpu del servidor anfitrión.
2. ***--cpuset-cpus***=””. Permite limitar el número de CPUs sobre las cuales se ejecutará el contenedor. Recibe un valor entero que representa el número de cpu o un rango. Por ejemplo: --cpu-set="0-1" para ejecutar un contenedor sobre la CPU 1 y CPU 2 del servidor.
3. ***--cpu-quota*** y ***--cpu-period***. Suelen usarse en conjunción y limitan el tiempo de acceso de contenedor al planificador (cpu-quota) durante un periodo (cpu-period). Así, --cpu-

quota=25000 y –cpu-limit=100000 limita un contenedor al 25% de tiempo de acceso al planificador.

Memoria:
Existen varias formas de limitar la memoria a través del arranque de un contenedor o a la hora de definirlo, pero el más habitual es el uso de los 2 modificadores:

1. ***--memory***=””. Permite asignar un valor de memoria al contenedor que se ejecuta introduciendo la cantidad de memoria en forma de un entero positivo y admitiendo las unidades b,k,m,g. Por ejemplo: --memory=”2g” para ejecutar un contenedor con 2Gb de memoria RAM.
2. ***--memory-swap***=””. Permite asignar un valor de memoria de intercambio al contenedor que se ejecuta, introduciendo la cantidad de memoria en forma de un entero positivo y admitiendo las unidades b,k,m,g. Por ejemplo: --memory-swap=”500m” para ejecutar un contenedor con 500Mb de memoria de intercambio.

Posibles ejemplos de ejecución:

- ✔ Ejecutar un contenedor con el nombre *prueba1*, desde la imagen de la última versión oficial de Ubuntu con un límite de 4 virtual CPUs y 2Gb de RAM:

```
sudo docker run --name prueba1 --cpuset-cpus=”0-3” --memory=”2g” \ ubuntu:latest
```

- ✔ Ejecutar un contenedor con el nombre *prueba2*, desde la imagen de la última versión oficial de Ubuntu con un límite de 25% del servidor y otro desde la misma imagen con nombre *prueba3* y 50% del servidor. Ambos con 1Gb de RAM y 256Mb de memoria de intercambio:

```
sudo docker run --name prueba2 --cpu-share=”256” --memory=1G \ --memory-swap=256m
ubuntu:latest

sudo docker run --name prueba3 –cpu-share=”512” --memory=1G \ --memory-swap=256m
ubuntu:latest
```

5.4.2. Almacenamiento en Docker. Volúmenes

[url5][url6]

En este apartado vamos a ver como gestionar el sistemas de almacenamiento en Docker para introducir su funcionamiento y ver los aspectos de configuración básicos que necesitamos conocer.
Los contenedores no forman parte del almacenamiento de datos persistentes por lo que cuando un contenedor se elimina, todos los datos que este contienen son borrados. Cuando necesitamos hacer uso de almacenamiento persistente en Docker debemos hacer uso de los volúmenes de datos (*data volume*), ya que es el mecanismo que ofrece para la gestión de los mismos.
Un volumen de datos permite fundamentalmente:

1. Guardar de forma persistente información de un contenedor
2. Compartir información entre contenedores

Internamente, un volúmen de datos es un directorio especial con metadatos[38] creado en cada servidor donde se ejecuta el servicio de Docker, y que suele estar albergado en la ruta física de cada servidor anfitrión */var/lib/docker/volumes*. Para la creación y gestión el cliente Docker emplea el comando “*docker volume*” que veremos a continuación.

Un volumen podemos usarlo de dos maneras distintas:

- **Volumen de datos:**

38 https://es.wikipedia.org/wiki/Metadato

Podemos crear y gestionar un volumen de datos de forma individual y posteriormente indicarle a un contenedor que use este volumen persistente que hemos creado cuando ejecutamos o creamos el contenedor.
Así, en primer lugar creamos un volumen denominado "volumenDeTest" con el comando que mostramos a continuación:

```
root@docker1:/# docker volume create --name volumenDeTest
volumenDeTest
```

Una vez creado, podemos ver su configuración del siguiente modo:

```
root@docker1:/# docker volume inspect volumenDeTest
[
    {
        "Name": "volumenDeTest",
        "Driver": "local",
        "Mountpoint": "/var/lib/docker/volumes/volumenDeTest/_data",
        "Labels": {},
        "Scope": "local"
    }
]
```

Podemos ver que el volumen de datos es un directorio especial, que se corresponde con el directorio "*/var/lib/docker/volumes/volumenDeTest/_data*" del equipo anfitrión.

A continuación podemos ya usar el volumen creado directamente mediante el modificador "*--volume*" o "*-v*" al crear o definir un contenedor, indicando el nombre del volumen y la dirección interna del contenedor donde irá mapeado:

```
root@docker1:/# docker run -it --name contenedor1 -v volumenDeTest:/test
ubuntu:latest /bin/bash
root@2b9f7c6cf23d:/# df -h
Filesystem      Size  Used Avail Use% Mounted on
/dev/vda1        38G  3.3G   33G  10% /
tmpfs          1000M     0 1000M   0% /dev
tmpfs          1000M     0 1000M   0% /sys/fs/cgroup
/dev/vda1        38G  3.3G   33G  10% /test
shm              64M     0   64M   0% /dev/shm
```

- **Contenedor de volúmenes de datos:**

La otra forma de usar los volúmenes de datos, es crear contenedores especiales donde creamos volúmenes de datos y después asociamos este contenedor a otros contenedores para que almacenen datos de forma persistente.
Podemos ver un ejemplo de su funcionamiento a continuación:
Con este primer comando creamos un contenedor denominado "*ContenedorDeDatos*" que contiene el volumen "*volumen_compartido*"

```
root@docker1:/# docker create -it --name ContenedorDeDatos -v /volumen_compartido
ubuntu:latest /bin/bash
4a70d3cb877f8c58e6ee799080cde2446527e941dcd7281d7d857ad841e08c57
```

A continuación creamos un contenedor denominado "*contenedor1*" y le asociamos los volúmenes del "*ContenedorDeDatos*".

```
root@docker1:~# docker run -it --name contenedor1 --volumes-from ContenedorDeDatos
ubuntu:latest /bin/bash
root@2d14604b3f5b:/# df -h
Filesystem      Size  Used Avail Use% Mounted on
/dev/vda1        38G  3.3G   33G  10% /
tmpfs          1000M     0 1000M   0% /dev
tmpfs          1000M     0 1000M   0% /sys/fs/cgroup
/dev/vda1        38G  3.3G   33G  10% /volumen_compartido
shm              64M     0   64M   0% /dev/shm
root@2d14604b3f5b:/# ls /volumen_compartido/
root@2d14604b3f5b:/#
```

Como podemos ver, el */volumen_compartido* esta mapeado en el nuevo contenedor creado y listo para ser usado.

Además de los volúmenes, existe otra forma adicional más simple de persistir datos en contenedores, que consiste en **mapear ficheros o directorios directamente** desde el equipo anfitrión. Para ello debemos de usar el simplemente el parámetro "*--volume origen:destino*" , donde el origen es un directorio o fichero del equipo anfitrión y el destino es un directorio o fichero dentro del contenedor, pero mapeado al indicado en el origen del equipo anfitrión, cuando definimos un contenedor. Podemos ver un ejemplo a continuación:

```
root@docker1:/# ls /opt
puppetlabs

root@docker1:/# docker run -it --name contenedorC -v /opt:/test ubuntu:latest /bin/bash

root@bc6c3bb4edf8:/# ls /test
puppetlabs
```

5.4.3. Red

[url7]

Docker permite crear y configurar distintas redes para la comunicación entre los contenedores de forma automática y a través de su propia linea de comandos mediante el comando principal "*docker network*". Por defecto, nada más instalarlo podemos ver las siguientes redes creadas por defecto mediante la ejecución del comando "*docker network ls*":

```
root@docker1:~# docker network ls
NETWORK ID          NAME                DRIVER              SCOPE
833c5f13ad17        bridge              bridge              local
d1200055a53b        host                host                local
6603c6d439ef        none                null                local
```

Como podemos ver, existen varios tipos de redes en Docker en función de la funcionalidad que necesitemos. Los tipos de redes más frecuentes son:

- **none**: Se trata de una zona de red sin conexión.
- **host**: Se trata de una zona de red que representa la red directamente mapeada del host anfitrión.
- **bridge**: Se trata de un puente de red, denominado por defecto "docker0" y donde se van añadiendo los contenedores que se van creando. Todos los contenedores que se van conectando en este tipo de zonas, por defecto tienen comunicación entre ellos. Docker viene preconfigurado además con la zona *bridge* con un direccionamiento 172.17.0.0/16 otorgando una clase B completa.
- **overlay o red superpuesta**: Este es el tipo de zona de red que definiremos sobre nuestra arquitectura de servidores para este proyecto. Este tipo de zonas permiten crear redes en

distintos servidores que estén configurados en cluster y que son visibles entre ellas de una manera segura (Se basan en el uso de la tecnología VXLAN[39]).

Para poder realizar las primeras pruebas sobre Docker, definiremos una red propia de tipo bridge sobre nuestra infraestructura de red, aunque como hemos indicado, la red que usaremos es la overlay ya que es la que nos permite orquestar un sistema de despliegues de servicios en distintos servidores y que estos se comuniquen sobre distintos contenedores, pero no podemos ponerla en marcha hasta que tengamos la solución de orquestación definitiva definida e instalada.

Vamos a proceder a configurar una zona de red de tipo bridge inicialmente, sobre la infraestructura de red de cada uno de los servidores y con el direccionamiento que habíamos previsto.
Direccionamiento IP: 172.16.0.0/22

Para ello ejecutamos el siguiente comando sobre cada uno de los servidores:

```
docker network create -d bridge --ip-range=172.16.1.0/24 --subnet=172.16.0.0/22
--gateway=172.16.1.1 -o parent=bond0 lan-privada
```

Con este comando, creamos una red de tipo bridge con el nombre *"lan-privada"* con el direccionamiento indicado, sobre el interfaz bond0.
Podemos consultar las múltiples opciones del comando *"docker network create"* a través del siguiente enlace.

Una vez ejecutado, podemos ver la nueva red bridge creada del siguiente modo:

```
root@docker1:~# docker network ls
NETWORK ID          NAME                DRIVER              SCOPE
833c5f13ad17        bridge              bridge              local
d1200055a53b        host                host                local
97bec1b1754b        lan-privada         bridge              local
6603c6d439ef        none                null                local
```

Y ver el detalle de la misma con el siguiente comando y salida de resultados:

```
root@docker1:~# docker network inspect lan-privada
[
    {
        "Name": "lan-privada",
        "Id": "97bec1b1754b4e3ffe51a324c64957501db54bf9aa29120a7803e78c696d3c09",
        "Scope": "local",
        "Driver": "bridge",
        "EnableIPv6": false,
        "IPAM": {
            "Driver": "default",
            "Options": {},
            "Config": [
                {
                    "Subnet": "172.16.0.0/22",
                    "IPRange": "172.16.1.0/24",
                    "Gateway": "172.16.1.1"
                }
            ]
        },
        "Internal": false,
        "Containers": {},
        "Options": {
            "parent": "bond0"
```

39 http://hispavirt.com/2013/01/02/entendiendo-vxlan/

```
        },
        "Labels": {}
    }
]
```

Como vemos, ya tenemos nuestra red creada y complemente funcional. Ahora, simplemente para crear contenedores virtuales sobre la misma, sólo tenemos que indicarlo mediante el modificador *--network="nombre de la red"*.

A continuación vamos a proceder a crear dos contenedores a modo de ejemplo sobre este bridge que hemos creado y comprobar la conectividad entre contenedores. Para ello, creamos 2 contenedores usando una imagen de la última versión del sistema operativo Ubuntu, con el comando "*docker run*" y los asignamos a nuestra red bridge de nombre "*lan-privada*" con los comandos que mostramos a continuación:

```
docker run -it --name="contenedor1" –network="lan-privada" \ ubuntu:latest /bin/bash

docker run -it --name="contenedor2" –network="lan-privada" \ ubuntu:latest /bin/bash
```

Ahora, del mismo modo que inspeccionamos antes la red, volvemos a comprobarla, y como se muestra a continuación, podemos ver los 2 contenedores que hemos creado asociados a la red y con los datos de direccionamiento ip de cada uno:

```
root@docker1:~# docker network inspect lan-privada
[
    {
        "Name": "lan-privada",
        "Id": "97bec1b1754b4e3ffe51a324c64957501db54bf9aa29120a7803e78c696d3c09",
        "Scope": "local",
        "Driver": "bridge",
        "EnableIPv6": false,
        "IPAM": {
            "Driver": "default",
            "Options": {},
            "Config": [
                {
                    "Subnet": "172.16.0.0/22",
                    "IPRange": "172.16.1.0/22",
                    "Gateway": "172.16.0.1"
                }
            ]
        },
        "Internal": false,
        "Containers": {
            "179f42baa4c97d2a85d1e9d535ef5a167fee9268422e05045841b12943925118": {
                "Name": "contenedor2",
                "EndpointID":
"714278256eef5a44b2dfb77bf554bb30bbacf37e627d5e3f52425e16733f4824",
                "MacAddress": "02:42:ac:10:00:03",
                "IPv4Address": "172.16.1.2/24",
                "IPv6Address": ""
            },
            "6c7a368449f38c0aca961ac1fb5266438067d993eb6bda0172f813323692fd42": {
                "Name": "contenedor1",
                "EndpointID":
"8701ffacb8f5173cfc25d43718bceadf034d4605500089c3fb8dd073ffb0ce50",
                "MacAddress": "02:42:ac:10:00:02",
                "IPv4Address": "172.16.1.3/24",
                "IPv6Address": ""
            }
        },
        "Options": {
```

```
            "parent": "bond0"
        },
        "Labels": {}
    }
]
```

Finalmente comprobamos como se muestra en las siguientes ilustraciones la conectividad, primero probando con un ping desde un contenedor a otro y después probando la conectividad de ambos desde el host anfitrión, también con sendos pings.

```
root@714278256eef:/# ping 172.16.1.3
PING 172.16.1.3 (172.16.1.3): 56 data bytes
64 bytes from 172.16.1.3: icmp_seq=0 ttl=64 time=0.145 ms
64 bytes from 172.16.1.3: icmp_seq=1 ttl=64 time=0.078 ms
^C--- 172.16.1.3 ping statistics ---
2 packets transmitted, 2 packets received, 0% packet loss
round-trip min/avg/max/stddev = 0.078/0.111/0.145/0.034 ms
root@714278256eef:/#
```

```
root@docker1:~# ping 172.16.1.2
PING 172.16.1.2 (172.16.1.2) 56(84) bytes of data.
64 bytes from 172.16.1.2: icmp_seq=1 ttl=64 time=0.088 ms
64 bytes from 172.16.1.2: icmp_seq=2 ttl=64 time=0.068 ms
^C
--- 172.16.1.2 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 999ms
rtt min/avg/max/mdev = 0.068/0.078/0.088/0.010 ms
root@docker1:~# ping 172.16.1.3
PING 172.16.1.3 (172.16.1.3) 56(84) bytes of data.
64 bytes from 172.16.1.3: icmp_seq=1 ttl=64 time=0.067 ms
64 bytes from 172.16.1.3: icmp_seq=2 ttl=64 time=0.074 ms
^C
--- 172.16.1.3 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 999ms
rtt min/avg/max/mdev = 0.067/0.070/0.074/0.009 ms
root@docker1:~#
```

Del mismo modo que hemos visto, usando el comando "*docker network*" podemos crear tantas redes distintas y del tipo que necesitemos de una manera rápida y realizando una importante abstracción de los comandos de configuración de red clásicos del sistema operativo, lo que dota a este sistema de una gran robustez a nivel de comunicación.
La red de tipo bridge que hemos creado nos sirve como ejemplo para la orquestación estática y para familiarizarnos con el sistema de red, ya que para nuestra arquitectura de múltiples servidores y sistema de orquestación dinámica será necesario configurar más adelante una red de tipo overlay (no puede configurarse una red de tipo overlay hasta que tengamos todos los servidores de cluster definidos y unidos y esto los haremos más adelante), que trabaja de un modo muy similar al tipo bridge pero permite la comunicación entre lo distintos servidores que formen parte del cluster, permitiendo de este modo crear una topología virtual de red que se extiende entre distintos servidores permitiendo la comunicación entre los contenedores que estén dentro de la misma y aunque esta se distribuya a múltiples servidores.

5.5. Trabajando con Docker. Gestión de imágenes

Los contenedores en Docker son instancias de imágenes. Estas imágenes por definición son plantillas del sistema de ficheros completo que vamos a ejecutar en modo lectura. Así, para continuar debemos ver los métodos que existen para crear nuestras propias imágenes o como modificar las existentes para poder adaptar su funcionamiento y estructura a nuestras necesidades. Es importante destacar que una vez creadas nuestras imágenes, estás están disponibles sólo en el servidor donde han sido compiladas y para poder usarlas desde cualquier

otro servidor como un sistema de publicación o distribución es necesario subirlas al Docker Hub o a cualquier otro registro de imágenes de Docker que tengamos.
Vemos a continuación las dos formas de crear nuevas imágenes con Docker.

5.5.1. Compilación de imágenes

La primera forma de crear un imagen propia o adaptada consiste en compilarla directamente de una imagen existente. Para ello, primero partimos de una imagen especifica que actuará como plantilla, para posteriormente arrancar una imagen en forma de contenedor e ir modificando manualmente todo lo necesario. Una vez modificado, solo será necesario compilar o construir este contenedor que tenemos para convertirlo en una nueva imagen que podrá ser usaba como base.
El comando que permite compilar un contenedor en una imagen es:

```
docker commit -m "Comentario" idcontenedor imagen:version
```

A partir del contenedor "*idcontenedor*" que hemos ejecutado previamente de una imagen y que hemos modificado, creamos una nueva imagen de nombre "*imagen*" e indicando la versión de la misma si fuese necesario.

5.5.2. Dockerfile

La segunda forma que existe para generar imágenes, además de ser la más utilizada, se trata de los ficheros conocidos como Dockerfile.
Estos ficheros funcionan a modo de guiones, que partiendo de una imagen base van ejecutando tareas específicas sobre el contenedor base modificándolo en función de nuestras necesidades. El proceso final pasa por compilar o construir la imagen definitiva a partir de nuestro Dockerfile.
La ventaja del uso de los ficheros Dockerfile es que el ciclo de vida de los contenedores está pensado para ser corto y evolucionar del mismo modo que el versionado de los servicios o aplicaciones que contienen. El uso del Dockerfile nos permite automatizar este proceso de creación de imágenes ya que podemos reutilizarlo adaptándolo tanto como sea necesario y rehusarlo, mientra que la compilación manual de imágenes requiere un trabajo manual de adaptación de la imagen que no es automatizable y por lo tanto repetible.

Las opciones completas que podemos usar para componer el fichero Dockerfile podemos verlas en el siguiente enlace.

[docker8]

Explicaremos ahora las opciones principales que tenemos en un Dockerfile, a través de un ejemplo sobre nuestro entorno de como crear una imagen propia. Vamos a suponer que necesitamos montar un entorno con 3 servidores web Apache2 a partir de una imagen desde la última versión del sistema operativo Ubuntu y usando como base nuestra infraestructura. Desde este modo vamos a crear esta nueva imagen para crear varios contenedores y para después poder usar la.
El fichero Dockerfile para crear la imagen sería el siguiente:

```
FROM ubuntu:latest

MAINTAINER angel@ameijeiras.es

RUN apt-get update && apt-get install -y apache2 && apt-get clean && rm -rf
/var/lib/apt/lists/*

ENV APACHE_SERVERNAME localhost
ENV APACHE_RUN_USER www-data
ENV APACHE_RUN_GROUP www-data
ENV APACHE_LOG_DIR /var/log/apache2
ENV APACHE_LOCK_DIR /var/run/
ENV APACHE_PID_FILE /var/run/apache2.pid
```

```
EXPOSE 80

COPY ./index.html /var/www/html/index.html

CMD ["/usr/sbin/apache2", "-D", "FOREGROUND"]
```

Usamos las direcciones *FROM* para indicar la imagen base desde la que partimos. Mediante *MAINTAINER* especificamos quién es el autor de la misma. Mediante la directiva *RUN* especificamos los comandos a ejecutar sobre la imagen, que en este particular consiste en instalar el servidor apache2 desde los repositorios oficiales de la distribución y a continuación hacer una limpieza de los datos que no sean necesarios para continuar y mantener un tamaño mínimo de imagen. Continuamos mediante la directiva *ENV* que se emplea para definir variables de entorno que serán visibles en el contenedor al arrancar y que se usan para pasar diversos comandos de configuración inicial, en este caso para el servidor web apache2.
Mediante *EXPOSE* indicamos que vamos a usar o mapear el puerto 80 para nuestra aplicación y mediante *COPY* indicamos que al compilar el Dockerfile copie un index.html del directorio en el que nos encontramos a directorio de destino indicando y ya dentro del contenedor. Se trata de un "*index.html*" que hemos preparado previamente para mostrar en el servidor web un mensaje personalizado de prueba.
Finalmente mediante la directiva CMD, indicamos que debe de ejecutarse por defecto una vez arranque el contenedor. En este caso, se trata del demonio del servidor web apache2 para que arranque en segundo plano.

Una vez listo lo compilamos y vemos la salida del proceso como se muestra a continuación:

```
root@docker1:/apache2# docker build -t mi-apache2 .
```

```
root@docker1:/apache2# docker build -t mi-apache2 .
Sending build context to Docker daemon 3.072 kB
Step 1 : FROM ubuntu:latest
 ---> 4ca3a192ff2a
Step 2 : MAINTAINER angel@ameijeiras.es
 ---> Using cache
 ---> 2d2821d86ed0
Step 3 : RUN apt-get update && apt-get install -y apache2 && apt-get clean && rm -rf /var/lib/apt/lists/*
 ---> Using cache
 ---> 3ced9fe33e26
Step 4 : ENV APACHE_SERVERNAME localhost
 ---> Running in e79c8bb6a0c0
 ---> cc8955834151
Removing intermediate container e79c8bb6a0c0
Step 5 : ENV APACHE_RUN_USER www-data
 ---> Running in dd67be481cc0
 ---> d80e2116988c
Removing intermediate container dd67be481cc0
Step 6 : ENV APACHE_RUN_GROUP www-data
 ---> Running in 589328e3cbd3
 ---> 3d3b334defa1
Removing intermediate container 589328e3cbd3
Step 7 : ENV APACHE_LOG_DIR /var/log/apache2
 ---> Running in 62acd1c29523
 ---> c7f990bc2565
Removing intermediate container 62acd1c29523
Step 8 : ENV APACHE_LOCK_DIR /var/run/
 ---> Running in bd78a96d2659
 ---> 19fae4f7e3f0
Removing intermediate container bd78a96d2659
Step 9 : ENV APACHE_PID_FILE /var/run/apache2.pid
 ---> Running in 42ad9291ae45
 ---> 728fb5096cb6
Removing intermediate container 42ad9291ae45
Step 10 : EXPOSE 80
 ---> Running in 020bb4825818
 ---> 0c6363e7ab07
Removing intermediate container 020bb4825818
Step 11 : COPY ./index.html /var/www/html/index.html
 ---> 5fb01a65aa19
Removing intermediate container edfc0a22d7d8
Step 12 : CMD /usr/sbin/apache2 -D FOREGROUND
 ---> Running in 3ff3426884b0
 ---> d89a578752f8
Removing intermediate container 3ff3426884b0
Successfully built d89a578752f8
root@docker1:/apache2#
```

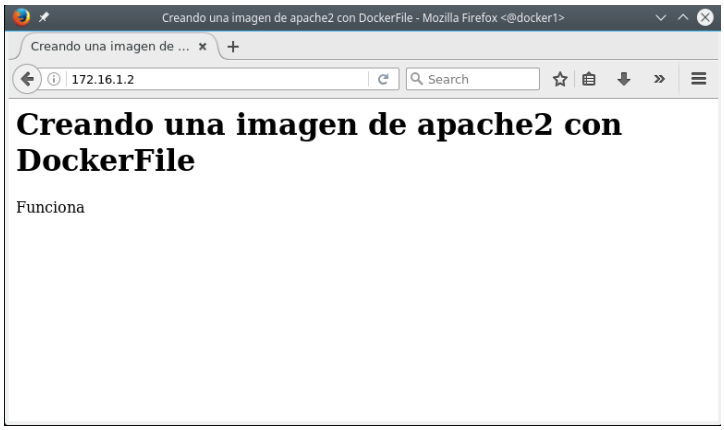
Una vez creada la imagen, simplemente necesitamos arrancar 3 contenedores denominados "*servidorWEB1*", "*servidorWEB2*" y "*servidorWEB3*" con la imagen creada "*mi-apache2*" sobre nuestra infraestructura:

```
root@docker1:/apache2# docker run -itd --name "servidorWEB1" --network=lan-privada mi-apache2
882a4a3400a69a53ed4c27da44dee838372d99bbcd63c07ddcfc4b3c7d0ac638
root@docker1:/apache2# docker run -itd --name "servidorWEB2" --network=lan-privada mi-apache2
1b39069478d8554cdc05625334aea157f408ee034791cc0902a86fceb2751613
root@docker1:/apache2# docker run -itd --name "servidorWEB3" --network=lan-privada mi-apache2
5e1067371360764d8fa22e1065d9424a38c59b8aacc468c5bdea1f00d43762af
```

Podemos ver los 2 servidores web en ejecución:

```
root@docker1:/apache2# docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS              PORTS               NAMES
5e1067371360        mi-apache2          "/usr/sbin/apache2 -D"   4 seconds ago       Up 3 seconds        80/tcp              servidorWEB3
1b39069478d8        mi-apache2          "/usr/sbin/apache2 -D"   8 seconds ago       Up 7 seconds        80/tcp              servidorWEB2
882a4a3400a6        mi-apache2          "/usr/sbin/apache2 -D"   13 seconds ago      Up 13 seconds       80/tcp              servidorWEB1
root@docker1:/apache2#
```

Y poder acceder usando un navegador web a una de las direcciones ip del mismo, para comprobar su funcionamiento:



5.6. Ecosistema Docker. Docker ToolBox

Docker no es simplemente un sistema de virtualización de contenedores clásico, ya que incluye un ecosistema completo de aplicaciones que le permiten automatizar y administrar un gran número de tareas y procesos que llevan la virtualización de contenedores a otro nivel. De este modo, Docker incluye un completo sistema de aplicaciones o herramientas adicionales, que dan soporte funcional a distintas tareas y que en combinación permiten dotarlo como hemos visto de nuevas funcionalidades convirtiéndolo en una solución completa de orquestación. Docker Compose y Docker Swarm son las herramientas que nos permitirán poner en marcha el sistema de orquestación. Docker Compose facilita la orquestación más a nivel de cada servidor de forma aislada mientras que Docker Swarm esta más orientado al nivel de cluster de orquestación de varios servidores.
A continuación introducimos brevemente todos los componentes del ecosistema y entraremos en mas detalle y ejemplos en Docker Compose y Docker Swarm:

5.6.1. Docker Hub

Docker Hub es la solución pública en formato cloud de Docker donde se mantienen las imágenes de los contenedores y que funciona a modo de repositorio de imágenes. Podemos registrarnos para usarlo de forma gratuita en el enlace : https://hub.docker.com/
Una vez registrados, podemos acceder y consultar vía navegador web todo el catalogo de aplicaciones públicas disponibles y también podemos gestionar un repositorio que incluye nuestra cuenta gratuita. Así, por defecto se incluye un repositorio (que puede ser público o privado) y un servicio de construcción o compilado de contenedores.
Adicionalmente existe una política de precios que se puede consultar en su propia página web por si necesitamos contratar más repositorios de contenedores directamente con Docker.

Para nuestro proyecto, usaremos imágenes públicas de este hub de Docker con lo que no necesitaremos servicios adicionales salvo la creación de una cuenta de tipo gratuito en Docker Hub, fundamentalmente para poder buscar y gestionar de un modo sencillo, imágenes de los contenedores públicos que aportan fabricantes de aplicaciones.

[docker1]

5.6.2. Docker Machine

Docker Machine es una herramienta que nos permite crear y administrar contenedores virtuales creados con Docker Engine. Docker Machine funciona como un componente que permite, mediante el uso de distintos drivers, crear y administrar nodos dónde está instalado Docker Engine. Así funciona de una manera similar a un entorno de centralización desde donde se controlan los servidor de contenedores que se ejecutan en otros nodos a través de Docker Engine. Un gran ventaja de Docker Machine es que existen múltiples drivers de proveedores, como puede ser Microsoft Azure o Amazon AWS, que permiten gestionar distintos Docker Engine en entornos heterogéneos desde un lugar común.

[docker2]

5.6.3. Docker Registry

Docker Registry es una aplicación que permite crear un Docker Hub para uso personal. Como comentamos en capítulos anteriores, donde hablamos del concepto de registro de Docker, debemos de recordar que existe la posibilidad de implantar de forma privada un sistema completo de registro usando el modelo "on promise"[40] como este que facilita Docker mediante esta solución mediante el modelo de cloud.
Es una solución muy sencilla de usar, porque se distribuye como una imagen de un contenedor, por lo que podemos desplegar un Docker Registry para uso propio de la siguiente manera (creamos un contenedor con nombre "*OwnRegistry*", que almacena los datos en el *Volumen1* creado previamente y que hace uso del puerto 5000 para el uso de la aplicación):

```
docker run -d -p 5000:5000 --name OwnRegistry --volume \ Volumen1:/var/lib/registry
registry:2
```

A continuación podríamos ya subir (PUSH) o descargar (PULL) imágenes a nuestro propio registro privado (la imagen "ubuntu" en el ejemplo que mostramos a continuación) de la siguiente manera:

```
docker push localhost:5000/ubuntu
docker pull localhost:5000/ubuntu
```

[docker3]

5.6.4. Docker Universal Control Plane

Docker Universal Control Plane, también conocido por sus siglas UCP, es un sistema de gestión integral que permite la administración centralizada de todas la herramientas Docker.
Se basa en una instalación en modalidad "on promise" que permite monitorizar y gestionar una arquitectura de nodos de Docker mediante una interfaz GUI a través de la cual podemos crear o administrar contenedores, volúmenes, red, etc.

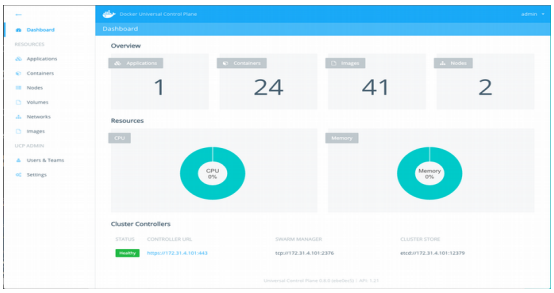


Ilustración 21: Cuadro de mando de Docker UCP

[docker5]

5.6.5. Docker Compose

40 https://en.wikipedia.org/wiki/On-premises_software

Docker Compose es una herramienta que permite ejecutar aplicaciones complejas que necesitan una solución multicontenedor. Permite crear a través de un fichero (*docker-compose.yml* por defecto) a modo guión, todos los requisitos y recursos necesarios para poder ejecutar una aplicación. Esta solución permite dar un paso más para simplificar la orquestación de una aplicación que requiera varios contenedores para ejecutarse, de modo que con un simple fichero podemos componer varias piezas o contenedores de un servicio. La limitación Compose a nivel de orquestación es que no nos permite usar múltiples máquinas para realizar el orquestado, permitiendo sólo el funcionamiento acotado al propio servidor anfitrión. La aplicación Docker Compose es un script escrito en lenguaje de programación Python, que se instala desde línea de comandos de forma muy sencilla:

```
apt install docker-compose
```

Una vez instalada del repositorio de Docker, es recomendable actualizarla a la última versión (2.0), que incluye múltiples nuevas mejoras y está disponible a través del instalador oficial de plugins de Python:

```
pip install docker-compose --upgrade
```

Así, Docker Compose permite orquestar de forma estática todos los componentes que una aplicación puede necesitar para funcionar correctamente mediante un único comando y en base a una fichero plantilla. Este fichero mediante formato YAML. Mediante el fichero se definen los servicios que tenemos (componentes de la aplicación), redes donde van conectados, volúmenes necesarios, etc.
Todas las opciones del fichero docker-compose podemos verlas en el siguiente enlace.

Vemos un ejemplo mediante supuesto práctico de docker-compose.yml, donde se crea una aplicación simulada que necesita ejecutar un servicio que será una aplicación Python y una base de datos Postgresql. Como vemos en el propio fichero, creamos dos servicios *db* y *web* donde a su vez creamos un contenedor de base de datos desde la imagen oficial de Postgresql, y creamos otro contenedor de la imagen oficial de Python versión 2.7. Definimos también una sección *volumes* , donde declaramos que usamos un directorio del equipo anfitrión para mapear los datos del código de ejemplo de Python. Incidamos una sección *ports* donde declaramos que exponemos el puerto 8000 donde se ejecutará la aplicación Python y finalmente otra seccion *depend_on* donde indicamos que para que el servicio *web* se ejecute, primero debe de crearse el servicio *db* del cual depende.
El fichero *docker-compose.yml* sería:

```
version: '2'
services:
  db:
    image: postgres
  web:
    image: python:2.7
    command: python manage.py runserver 0.0.0.0:8000
    volumes:
      - .:/code
    ports:
      - "8000:8000"
    depends_on:
      - db
```

Una vez creado el docker-compose.yml sólo es necesario arrancarlo con el comando "*docker-compose up*" o pararlo mediante "*docker-compose down*".
También podemos usar la ordenes habituales que se usan de forma individual con el cliente Docker, pero estas se ejecutan sólo sobre los servicios individuales definidos dentro de este proyecto Docker Compose.

Vemos el resultado de la ejecución del fichero anterior y como se crean y enlazan ambos contenedores (con nombre *zzz_db_1*$_1$ y *zzz_web_1*), volúmenes de datos, etc. en un sólo paso:

```
root@docker1:/zzz# docker-compose up
Creating network "zzz_default" with the default driver
Pulling db (postgres:latest)...
latest: Pulling from library/postgres
386a066cd84a: Pull complete
e6dd80b38d38: Pull complete
9cd706823821: Pull complete
40c17ac202a9: Pull complete
7380b383ba3d: Pull complete
538e418b46ce: Pull complete
c3b9d41b7758: Pull complete
dd4f9522dd30: Pull complete
920e548f9635: Pull complete
628af7ef2ee5: Pull complete
004275e6f5b5: Pull complete
Digest: sha256:e761829c4b5ec27a0798a867e5929049f4cbf243a364c81cad07e4b7ac2df3f1
Status: Downloaded newer image for postgres:latest
Pulling web (python:2.7)...
2.7: Pulling from library/python
386a066cd84a: Already exists
75ea84187083: Pull complete
88b459c9f665: Pull complete
1e3ee139a577: Pull complete
729baab2cba2: Pull complete
eda53d7523c4: Pull complete
52e71dbe4c63: Pull complete
Digest: sha256:f8763014855f9bad3607d9309a248439052d2d4a000ace3f207a2df63a83f238
Status: Downloaded newer image for python:2.7
Creating zzz_db_1
Creating zzz_web_1
Attaching to zzz_db_1, zzz_web_1
```

```
root@docker1:/zzz# docker ps -a
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS                      PORTS               NAMES
b65a698f8de1        python:2.7          "python manage.py run"   33 seconds ago      Exited (2) 32 seconds ago                       zzz_web_1
14ca296bcd21        postgres            "/docker-entrypoint.s"   34 seconds ago      Exited (0) 16 seconds ago                       zzz_db_1
```

Cuando arrancamos un contenedor es habitual que definamos parámetros específicos del mismo como la red en la que irá conectado, los volúmenes de datos que contendrá, puertos de red que necesitamos mapear, etc. Si necesitamos arrancar un servicio o un entorno de varios contenedores, este proceso se va complicando y vamos necesitando varias líneas de comando con su configuración específica.
Así con Docker Compose podemos crear sistemas complejos con múltiples servicios, volúmenes, redes, etc de una manera sencilla y escalable, con una configuración sencilla y escasa intervención manual humana, por lo que podemos considerarlo herramienta de orquestación estática.

Al final veremos un ejemplo más completo de orquestación estática con Docker Compose y escalado, sobre nuestra infraestructura para demostrar su funcionamiento en un sistema de servicios web y base de datos más completo.

[docker6]

5.6.6. Docker Swarm

Docker Swarm es la herramienta del ecosistema Docker que permite crear un cluster de alta disponibilidad de virtualización de contenedores y lo dota se un sistema de orquestación dinámica de servicios a nivel de múltiples servidores, funcionando como un enjambre de nodos de provisión de servicios.
En el siguiente capítulo vemos la solución Docker Swarm en detalle y su aplicación sobre nuestra arquitectura.

[docker7]

6. Docker Swarm. Orquestación dinámica de aplicaciónes.

Docker Swarm es la herramienta del ecosistema Docker, que permite crear un cluster de alta disponibilidad de virtualización de contenedores, y lo dota de un sistema de orquestación dinámica de servicios a nivel de múltiples servidores, funcionando como un enjambre de nodos de provisión de servicios.

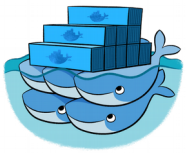


Ilustración 22: Logotipo Docker Swarm

[docker7] [libro1] [libro2] [libro3] [libro4] [libro5]

Desde la versión 1.12, Docker Swarm ya no se considera como una aplicación del ecosistema Docker que sea necesario instalar, ya que debido a su enorme funcionalidad la han integrado en Docker Engine. Por este motivo, con la instalación base de Docker Engine que hemos realizado ya tenemos Docker Swarm instalado y funcional para empezar a trabajar y a configurar un entorno de cluster.
Antes de continuar, es necesario definir un par de conceptos empleados en Docker Swarm y que encontraremos referenciados a los largo del proyecto:

- **Tarea:** Una tarea (*"task"*) en un contenedor asignado a un nodo de trabajo. Si este contenedor falla o sufre un problema, el nodo se encarga de recrearlo y si el nodo y la tarea fallan se crea una nueva tarea en un nodo.
- **Servicio:** Un servicio (*"service"*) es un ente que define el estado de un conjunto de tareas, indicando la imagen a la que pertenece, funcionando como identificador para balanceo de carga e indicando el número de replicas o instancias que existen del mismo.

Docker Swarm incluye por defecto en su solución una serie de funcionalidades necesarias cuando implementamos un sistema de cluster de servidores, y que otras soluciones no incluyen, y por lo tanto, fuerzan la necesidad de software adicional para permitir implementar estas funcionalidades. Estos elementos son:

- **Balanceador de carga**. Por defecto, Docker Swarm provee una solución de balanceo de carga implementada de forma nativa. Incluye un servicio que usando la tecnología IPVS[41] (De las siglas IP Virtual Server), crea de forma automática un balanceador de carga de capa 4 en el stack TCP/IP para cada servicio que creamos a nivel de cluster. Simplemente con hacer uso del nombre con el que se crea el servicio, si este tiene múltiples instancias (con distintas direcciones ip), Docker Swarm resuelve automáticamente las direcciones de red de los servicios de forma balanceada usando distintos algoritmos preconfigurables.
- **Servicio de descubrimiento clave-valor**. Otras soluciones de orquestación necesitan que se instale adicionalmente un servicio de descubrimiento de tipo clave-valor. Suelen usarse herramientas como *consul*[42], *etcd*[43] o *ZooKeeper*[44]. Con Docker Swarm este servicio viene incluido, con lo cual no necesitamos ninguna herramienta ni servicio adicional para apoyar la solución de orquestación propuesta.

41 https://en.wikipedia.org/wiki/IP_Virtual_Server [url8]
42 https://www.consul.io/ [consul]
43 https://coreos.com/etcd/ [etcd1]
44 https://zookeeper.apache.org/

- **Servicio de tolerancia a fallos y replicación**. Incluye un sistema automático de *scheduling* que le permite tomar ciertas decisiones de forma automatizada en caso de problemas en algún nodo. Es decir, podemos indicarle el número de instancias de un servicio que necesitamos tener disponibles, por ejemplo de un servidor web, y este se encarga de distribuirlas de forma automática entre los nodos disponibles. Si posteriormente un nodo con un número X de instancias sufre un problema y esas X instancias dejan de estar disponibles, automáticamente el Swarm se encarga de arrancarlas en otro nodo para compensar el problema y cumplir con el número de instancias de un servicio que le habíamos indicado.
- **Servicios de comprobación de salud**. Incluye de forma nativa, sistemas para chequear la salud de los nodos y de los contenedores para la monitorización y la toma de decisiones.

Además de estas ventajas que hemos destacado, es la solución nativa de Docker para la orquestación dinámica y por tanto será la herramienta que usaremos.

6.1. Otras herramientas de orquestación dinámica

Existen actualmente dos herramientas más de orquestación que son competidoras directas y que permiten realizar funciones similares a Docker Swarm. Estas herramientas son Kubernetes y Marathon MESOS.

6.1.1. Kubernetes

Kubernetes es una herramienta de código abierto para la orquestación de contenedores, desarrollada íntegramente por Google y presentada en 2014. Se desarrolla inicialmente para el propio uso interno de Google como orquestador de su sistema de recursos en los datacenters. Esta centrado en el despliegue de servicios, actualización y escalado sin necesidad de interrupciones en el mismo. La mayor ventaja de este sistema es que esta diseñado para ser portable y funcionar en múltiples infraestructuras tanto local, soluciones en la nube o sistema de cloud híbrido. El mayor inconveniente es que comporta una gran complejidad de implementación por lo que es un sistema con una importante curva de aprendizaje para su implantación en entornos complejos.

6.1.2. MARATHON MESOS

Es una solución que aparece publicada en el año 2009 y que se basa en una combinación de dos herramientas (MARATHON y MESOS) que se complementan para formar una solución de orquestación. Por un lado MESOS es un sistema de administración de clusters que permite desplegar aplicaciones sobre máquinas físicas o virtuales, mientras que MARATHON es un framework[45] que funciona sobre MESOS y que hace que la infraestructura de MESOS funcione con contenedores, con lo cual también es compatible con Docker. El mayor problema de este sistema es que la configuración de los entornos de red y de descubrimiento de servicios (Implementado con Zookeeper[46]) es sumamente complejo.

45 https://es.wikipedia.org/wiki/Framework
46 https://zookeeper.apache.org/ [zoo]

6.2. Definición de la arquitectura Swarm

Docker Swarm emplea una arquitectura clásica de cluster, donde existe uno o múltiples nodos responsables de la coordinación del cluster e identificados como nodos “Manager“, y nodos donde se ejecutan los procesos de trabajo, que en este caso son contenedores virtuales y que son conocidos como nodos “Worker”. Los nodos Manager, pueden compartir también el role de “Worker” al mismo tiempo.

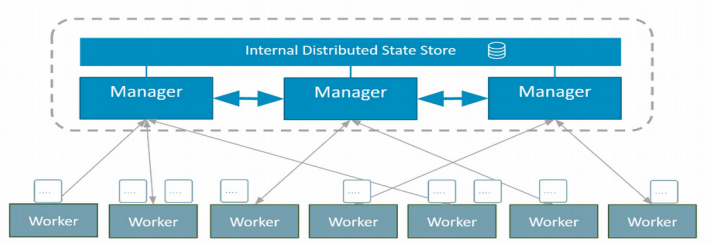


Ilustración 23: Arquitectura Docker Swarm

El sistema Swarm implementa la posibilidad de tener múltiples nodos Manager, para dotar al entorno de orquestación de alta disponibilidad. Para ello emplea el algoritmo de consenso y elección de líder Raft[47]. El funcionamiento se basa en que sólo hay un nodo Manager principal o líder. Si este se cae o sufre un problema se elije otro nodo Manager como nuevo líder.
Señalar que en nuestra arquitectura comenzamos con 2 servidores físicos, con lo cual ambos serán Manager y Workers para poder usarlos para desplegar contenedores y para que si un Manager sufre algún problema el otro asuma el liderato. Señalar que no es la solución ideal, ya que el número de nodos Manager para garantizar una alta disponibilidad completa debería ser impar y empezar en un mínimo de 3 servidores, pero sí debe tenerse en cuenta si incluimos más de 3 nodos mánager. Destacar también que el sistema Swarm está pensado de igual modo para tolerancia a fallos y disponibilidad, por lo que si un sistema completo se queda sin Managers, el entorno que exista en ese momento continúa funcionando sin ningún problema hasta que un nuevo Manager vuelva a estar disponible. Lo único que se perdería es la posibilidad de orquestación de forma automatizada, pero todos los servicios (volúmenes, redes, etc...) y contenedores continuarían plenamente funcionales.

Swarm Size	Majority	Fault Tolerance
1	1	0
2	2	0
3	2	**1**
4	3	1
5	3	**2**
6	4	2
7	4	**3**
8	5	3
9	5	**4**

Ilustración 24: Tolerancia a Fallos

6.3. Instalación y puesta en marcha. Configuración del sistema

Con la instalación de Docker Engine, realizada en capítulos anteriores, ya tenemos disponible y completamente funcional el sistema Swarm. Para empezar a trabajar sólo necesitamos crear un

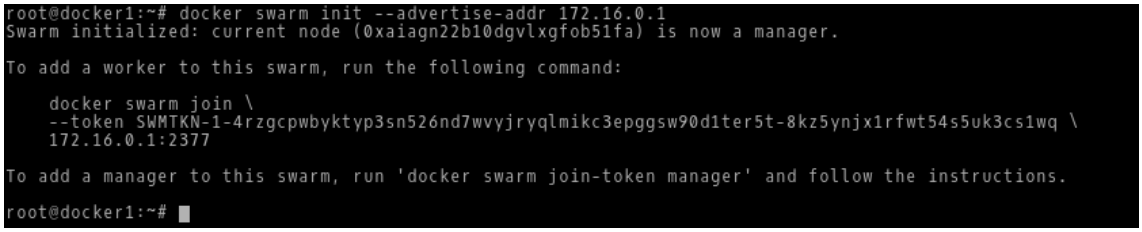
47 https://raft.github.io/ [raft]

cluster con al menos un nodo Manager y un nodo Worker. En nuestro entorno vamos a crear ambos servidores como nodos Manager y Workers.

Vamos a adoptar la convención de nombrar como *docker1*, a nuestro primer servidor físico con ip 172.16.0.1, que será también el primer nodo del cluster. El segundo servidor físico, lo nombraremos *docker2, con* ip 172.16.0.2 que será configurado como segundo nodo del cluster y se configurará con el rol de worker

Para crear una configuración del cluster con los dos primeros servidores, sobre el primer nodo docker1 ejecutamos el siguiente comando al que sólo es necesario indicarle la dirección ip (y sobre esta interfaz de red, recordemos que definiéramos un bond0 como interfaz de tipo bonding para usar dos interfaces de red físicos y que estos no supongan un cuello de botella, irá toda la comunicación del cluster y de las redes overlay que crearemos).

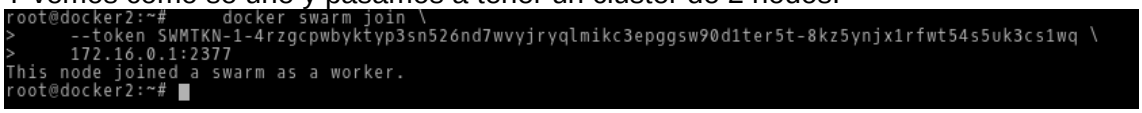
```
root@docker1:~# docker swarm init --advertise-addr 172.16.0.1
```



La salida que nos muestra, indica que se ha inicializado correctamente el nodo donde lo hemos ejecutado (docker1) y que se ha convertido en el Manager del cluster de Docker Swarm. Además nos indica ya directamente los dos posibles comandos que debemos de ejecutar sobre otros nodos para unirlos con el rol de Manager o Worker en este cluster.
Ejecutamos a continuación, sobre el nodo2 o docker2, el comando para añadir este servidor como nodo con el role de Worker al cluster:

```
root@docker2:~#docker swarm join  \
--token SWMTKN-1-4rzgcpwbyktyp3sn526nd7wvyjryqlmikc3epggsw90d1ter5t-
8kz5ynjx1rfwt54s5uk3cs1wq 172.16.0.1:2377
```

Y vemos como se une y pasamos a tener un cluster de 2 nodos.



Ahora que ya tenemos 2 nodos, todos los comandos de administración y orquestación sólo podremos ejecutarlos sobre el nodo Manager. Si no lo hacemos así y tratamos de ejecutarlos, Docker nos lo indicará con un error.
Finalmente, sobre el nodo con rol de Manager que a partir de ahora será el servidor1 identificado como *nodo1* o *docker1*, ejecutamos el comando "*docker info"* y "*docker node ls"* y comprobamos el estado del cluster y los roles de los nodos:

```
root@docker1:~#docker info
```

```
 Network: null overlay bridge host
Swarm: active
 NodeID: 0xaiagn22b10dgvlxgfob51fa
 Is Manager: true
 ClusterID: 0um2z7qts7hvt4sn04erdhwyk
 Managers: 1
 Nodes: 2
 Orchestration:
  Task History Retention Limit: 5
 Raft:
  Snapshot Interval: 10000
  Heartbeat Tick: 1
  Election Tick: 3
 Dispatcher:
  Heartbeat Period: 5 seconds
 CA Configuration:
  Expiry Duration: 3 months
 Node Address: 172.16.0.1
Runtimes: runc
```

```
root@docker1:~#docker node ls
```

```
root@docker1:~# docker node ls
ID                            HOSTNAME  STATUS  AVAILABILITY  MANAGER STATUS
0xaiagn22b10dgvlxgfob51fa *   docker1   Ready   Active        Leader
9j49b6fixcxtlc605tcl2cxm5     docker2   Ready   Active
root@docker1:~#
```

Ya podemos desplegar contenedores en el cluster de Swarm que hemos implementado en cualquiera de los nodos, por lo que el último punto que nos falta es configurar una red de overlay para crear una red interna, que permita comunicar todos los contenedores de un servicio independientemente de en que nodo o servidor físico se encuentren desplegados. Por defecto, cuando inicializamos un cluster de Swarm, se crean automáticamente dos nuevas redes en todos los nodos del cluster, además de las redes por defecto que vimos en capítulos anteriores. Una de ella es la red overlay por defecto “ingress” de Docker Swarm y la otra es una red de tipo bridge *“docker_gwbridge”*. La red *“docker_gwbridge*” se utiliza para comunicar los contenedores de las redes overlay con el exterior y para la comunicación entre los nodos de Swarm en los diferentes hosts, por lo que funciona a modo de gateway interno para el uso de Docker.

Para crear una nueva red overlay, sólo tenemos que usar desde el nodo “Manager” el comando “*docker network*” que ya conocemos, indicando que es de tipo overlay mediante el drivers, indicando un rango de red (para este caso en concreto declaramos una subred de clase c para el uso en los contenedores) y le indicamos un nombre identificativo “*lan_proyecto*”

```
root@docker1:~# docker network create --driver overlay --subnet 10.0.2.0/24 lan_proyecto
```

```
root@docker1:~# docker network create --driver overlay --subnet 10.0.2.0/24 lan_proyecto
6tkq3053ffv6ezm87wry5rb1d
root@docker1:~# docker network ls
NETWORK ID          NAME                DRIVER              SCOPE
0fcfde31168b        bridge              bridge              local
765e5e96f6a4        docker_gwbridge     bridge              local
83af1be0e92b        host                host                local
3o5bezajoc2x        ingress             overlay             swarm
6tkq3053ffv6        lan_proyecto        overlay             swarm
a173ffe625a5        none                null                local
root@docker1:~#
```

```
root@docker1:~# docker network inspect lan_proyecto
```

```
root@docker1:~# docker network inspect lan_proyecto
[
    {
        "Name": "lan_proyecto",
        "Id": "6tkq3053ffv6ezm87wry5rb1d",
        "Scope": "swarm",
        "Driver": "overlay",
        "EnableIPv6": false,
        "IPAM": {
            "Driver": "default",
            "Options": null,
            "Config": [
                {
                    "Subnet": "10.0.2.0/24",
                    "Gateway": "10.0.2.1"
                }
            ]
        },
        "Internal": false,
        "Containers": null,
        "Options": {
            "com.docker.network.driver.overlay.vxlanid_list": "257"
        },
        "Labels": null
    }
]
```

Al igual que sucedía cuando creábamos redes en Docker Compose sobre un sólo nodo, aquí podemos proceder del mismo modo y crear tantas redes de tipo overlay como necesitemos y todos los contenedores que unamos a esta red estarán comunicados, independientemente del nodo físico en donde se desplieguen.
Cuando inicializamos un cluster de Docker Swarm, este incluye automáticamente un servicio de almacenamiento clave-valor donde se registran todas las redes overlay que vamos creando en el nodo *Manager* (que recordemos que es el único nodo que nos permite la creación de estas redes y comandos del cluster) y su topologías. Cada nuevo nodo del cluster, consulta este servicio de forma automática y va replicando y propagando estas redes de overlay. Para la segregación interna de las redes overlay, Docker usa la tecnología VXLAN en base a la estructura de redes creadas en el repositorio interno de almacenamiento clave-valor.

[dockernt]

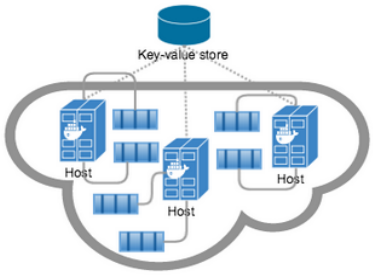


Ilustración 25: Arquitectura de redes Overlay

Una vez que tenemos el entorno de red funcional sólo nos queda empezar a desplegar servicios. A diferencia de cuando teníamos un sólo nodo con Docker Compose, donde ejecutábamos los mismos con el comando "*docker run*", en un entorno con varios nodos en cluster con Docker Swarm, ahora lo que creamos son servicios que ejecutan tareas.
Recordamos la definición de servicio: "Un servicio (*"service"*) es un ente que define el estado de un conjunto de tareas, indicando la imagen a la que pertenece, funcionando como identificador para balanceo de carga e indicando el número de réplicas o instancias que existen del mismo".
De este modo, el comando principal usado en Docker Swarm para crear servicios es: "***docker service***", que se combina con las directivas:

- *create*: Crea un servicio en el entorno de cluster.
- *ls* : Lista los servicios disponibles en un entorno de cluster.
- *rm* : Borra un servicio.
- *scale*: Permite escalar el tamaño de un servicio.
- *inspect*: Inspecciona a fondo un servicio.
- *ps*: Muestra las tareas pertenecientes a un servicio.

Vamos a detallar las opciones más importantes que permite el comando “docker service create”, que es el más complejo al ser en la creación del servicio donde se definen sus principales opciones de funcionamiento. Se pueden consultar todas las opciones en detalle a través del siguiente enlace.

[dockerservice]

El comando “*docker service create”* admite, entre otros, los siguientes modificadores:

- --*endpoint-mode* (*vip* o *dnsrr*): Cada vez que creamos un servicio en Docker Swarm, este crea automáticamente un balanceador de cargar para distribuir de forma automática las peticiones a las distintas tareas o instancias del servicio. Si especificamos dnsrr, el sistema integrado de balanceador de carga funcionará a modo de round robin dns [48] (Resolviendo cada vez una dirección ip de alguno de los servicios activos). Si especificamos vip (Opción por defecto), con cada servicio Docker Swarm creará un balanceador de carga 4 usando la tecnología IPVS[49] que incluye el Kernel de Linux. En este modo, Swarm crea una dirección ip con cada servicio que automáticamente balancea todas las peticiones que recibe sobre todas las instancias de cada servicio.
- --*env*: Permite enviar variables de entorno a la imagen del contenedor que funcionará como base de este servicio.
- --*network*: Permite indicar el nombre de la red de overlay donde se crearán los contenedores de este servicio.
- --*replicas*: Permite indicar el número de tareas (instancias) que definimos que son necesarias para poder ofrecer este servicio.
- --*mode* (*replicated* o *global*): Permite indicar el modo de funcionamiento del servicio. Si especificamos *replicated* (Modo por defecto) el servicio ejecuta el número de tareas en cualquiera de los nodos del cluster, mientras que si especificamos *global*, el servicio ejecuta una tarea en cada nodo del cluster de forma automática. Con este último modo, lo que conseguimos es que este servicio se ofrezca en alta disponibilidad porque automáticamente estamos indicando que necesitamos una instancia del mismo en cada nodo que tengamos en el cluster. Si posteriormente añadiésemos más nodos a nuestro cluster, con sólo unirlos, automáticamente de desplegaría una instancia de este tipo de servicios en los nuevos nodos para cumplir la restricción de servicio en modo *global*.
- --*name*: Permite indicar el nombre identificativo del servicio.

Y por último se indica la imagen del contenedor a usar para implementar el servicio. La estructura del comando al completo sería: “*docker service create [MODIFICADORES] IMAGEN”*

Veamos ahora el funcionamiento de los servicios con varios ejemplos sobre nuestra infraestructura:

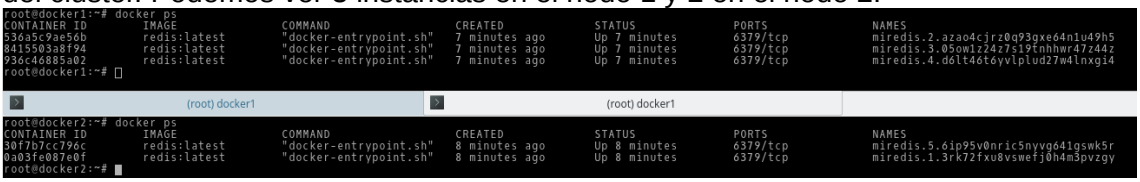
Vamos a crear un servicio con 5 instancias, basándonos en una imagen de un servidor Redis[50] (Redis es un motor de base de datos en memoria, basado en el almacenamiento en tablas de hashes (clave/valor) que vamos a usar como ejemplo), desplegandolo sobre la red de overlay lan_proyecto:

```
root@docker1:~#docker service create --name miredis --network lan_proyecto  --replicas 3
redis:latest
```

```
root@docker1:~# docker service create --name miredis --network lan_proyecto  --replicas 5 redis:latest
36moqtxcmba5ssfgp05cqemtc
root@docker1:~# docker service ls
ID            NAME     REPLICAS  IMAGE         COMMAND
36moqtxcmba5  miredis  0/5       redis:latest
root@docker1:~#
```

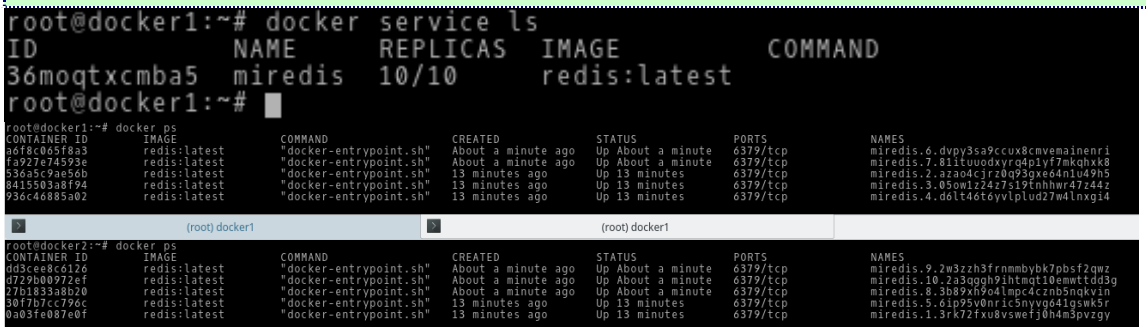
48 https://es.wikipedia.org/wiki/Dns_round_robin [dnsrr]
49 http://www.linuxvirtualserver.org/software/ipvs.html [ipvs]
50 https://es.wikipedia.org/wiki/Redis

Comprobamos como se crean automáticamente 5 instancias de forma repartida entre los nodos del cluster. Podemos ver 3 instancias en el nodo 1 y 2 en el nodo 2.



Si ahora necesitamos escalar el servicio, a 10 instancias del mismo, procedemos con el comando que se muestra a continuación y comprobamos el estado del servicio y como se crean las tareas en ambos nodos:

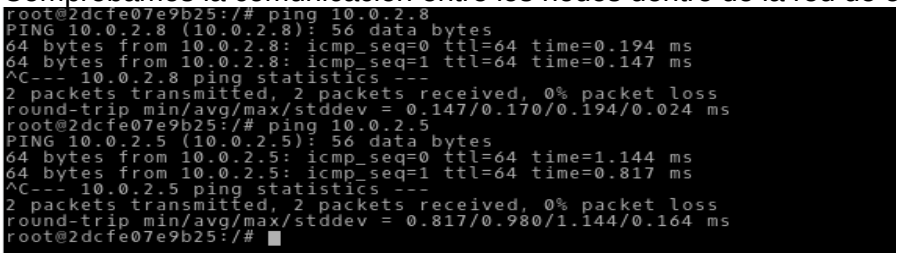
```
root@docker1:~#docker service scale miredis=10
```



Si ahora necesitamos reducir el servicio, a 2 instancias, procedemos con el comando que se muestra a continuación y comprobamos el estado del servicio y como se crean las tareas en ambos nodos:

```
root@docker1:~#docker service scale miredis=10
```

Comprobamos la comunicación entre los nodos dentro de la red de overlay:



Comprobamos cual es la dirección ip creada de forma automática por Docker Swarm para balancear las peticiones a este servicio, mediante el siguiente comando:

```
root@docker1:~#docker service inspect miredis
```

```
… (omitido)
            "VirtualIPs": [
                {
                    "NetworkID": "6tkq3053ffv6ezm87wry5rb1d",
                    "Addr": "10.0.2.2/24"
```

```
                }
… (omitido)
```

Comprobamos la comunicación con el balanceador de carga para este servicio desde uno de los nodos de la red:

```
root@2dcfe07e9b25:/# ping miredis
PING miredis (10.0.2.2): 56 data bytes
64 bytes from 10.0.2.2: icmp_seq=0 ttl=64 time=0.088 ms
64 bytes from 10.0.2.2: icmp_seq=1 ttl=64 time=0.123 ms
^C--- miredis ping statistics ---
2 packets transmitted, 2 packets received, 0% packet loss
round-trip min/avg/max/stddev = 0.088/0.105/0.123/0.000 ms
root@2dcfe07e9b25:/#
```

7. Ejemplos de despliegues de aplicaciones orquestadas con contenedores virtuales Docker

Veamos finalmente dos ejemplos de orquestación estática (sobre un sólo nodo) y dinámica (sobre un cluster de nodos) usando las herramientas que nos ofrece Docker – Docker Compose y Docker Swarm – y todo ello sobre la infraestructura propuesta.

Para estos ejemplos, vamos orquestar la instalación y escalado de la aplicación WordPress. WordPress es un CMS (de sus siglas en inglés, Content Management System) o sistema de gestión de contenidos orientado a la creación de sitios web. Hoy en día, es uno de los principales sistemas comerciales para la creación de páginas web de ámbito corporativo y su éxito se basa en ser un sistema robusto y que aporta una gran adaptabilidad a nivel de formatos y una gran facilidad de administración.
Por este motivo, es muy habitual verlo instalado en ámbitos corporativos de distinta índole y por su enorme implantación empresarial hemos decidido usarlo como ejemplo.
Está escrito en lenguaje PHP[51] y necesita como componentes básicos para su funcionamiento un servidor web con capacidad para ejecutar PHP y una base de datos.

[wordpress1]

[wordpress2]

7.1. Ejemplo completo de despliegue de un sistema CMS usando Worpress a través de orquestación estática sobre un nodo y usando Docker Compose

Vamos a emplear la herramienta Docker Compose para crear un servicio orquestado estáticamente para el despliegue automático y con posibilidades de escalado sobre la arquitectura y servidores propuestos. Aclarar que este ejemplo sólo permite la orquestación y escalado sobre un único servidor físico, que hemos visto que es la limitación de la herramienta Docker Compose.

Comenzamos por crear dentro de uno de los servidores anfitriones, una carpeta de trabajo que para este ejemplo llamaremos “*wordpress*”. Dentro de la misma, creamos el siguiente fichero *docker-compose.yml*:

```
version: '2'

services:
   db:
     image: mysql:5.7
     volumes:
       - db_data:/var/lib/mysql
     restart: always
     environment:
       MYSQL_ROOT_PASSWORD: wordpress
       MYSQL_DATABASE: wordpress
       MYSQL_USER: wordpress
       MYSQL_PASSWORD: wordpress

   wordpress:
     depends_on:
       - db
     image: wordpress:latest
```

51 http://php.net/manual/es/intro-whatis.php

```
    restart: always
    environment:
      WORDPRESS_DB_HOST: db:3306
      WORDPRESS_DB_PASSWORD: wordpress
volumes:
    db_data:

networks:
  default:
    external:
      name: lan-privada
```

Este fichero, podemos descomponerlo en 3 grandes bloques:
Primero vemos el bloque mas importante que es "*services:*"
Dentro de este bloque es donde definimos los servicios que vamos a orquestar. En este ejemplo, necesitamos definir 2 servicios porque como hemos introducido, para ejecutar WordPress se necesita un servidor web que permita ejecutar código PHP (servicio *wordpress*) y una base de datos mysql (servicio *db*). Este primer servicio,de nombre *db*, se basa en la imagen oficial de la base de datos mysql "mysql:5.7" y define un de un volumen de datos de nombre "db_data" que será lo que haga persistente el almacenamiento de ficheros para la base de datos y finalmente también define una serie de variables de entorno que se usan para preconfigurar los datos de acceso a la base de datos que usará la aplicación WordPress.
El siguiente servicio que se declara en el fichero docker-*compose.yml* es un servicio denominado *wordpress*, el cual parte de la imagen oficial de la última versión de WordPress (que ya incluye un servidor web Apache 2.4 preconfigurado) y que define a través de variables de entorno los datos de configuración de la base de datos que hemos definido y que además se encarga en enlazarlos para su permitir su comunicación.
En los 2 últimos bloques se establecen, por un lado los volúmenes de datos persistente que vamos usar (*db_data* - en el servicio de base de datos en este ejemplo) y la configuración de la red que vamos a emplear. A nivel de red, vamos a usar la red *lan-privada* que definimos para usar en nuestra infraestructura de red en capítulos anteriores, así como su direccionamiento y que ahora declaramos su uso mediante la directiva *networks*.
Recordamos que la definíamos con el siguiente comando:

```
docker network create -d bridge --ip-range=172.16.1.0/24 --subnet=172.16.0.0/22
--gateway=172.16.1.1 -o parent=bond0 lan-privada
```

Destacar en este punto, que si no indicásemos nada en la directiva *networks*, Docker Compose al iniciar los contenedores nos crearía una nueva red virtual propia en la que sólo estaría accesibles los contenedores que forman parte de esta orquestación de este servicio con su propio direccionamiento de red. Esta también sería una buena arquitectura de red para estos servicios porque todos los contenedores estarían en su propia red protegidos de accesos externos y sólo tendríamos que exponer el puerto 80 del servidor web mapeando sobre la dirección ip del servidor anfitrión.
En este ejemplo, hemos decidido hacerlo haciendo uso del rango y red principal del anfitrión para poder comprobar la orquestación de *Docker Compose,* haciendo uso de una red previamente creada. En el siguiente ejemplo sobre la arquitectura que propondremos lo haremos creando una red propia para los contenedores con lo que de este modo vemos ambas posibilidades cada una de ellos en un ejemplo.
Como podemos comprobar, sobre una arquitectura fija de red que se ha propuesto, podemos diseñar sistemas o micro-arquitecturas autónomas que se adaptan a los distintos requisitos que puede necesitar una aplicación al orquestarla sin necesidad de costosos cambios hardware.

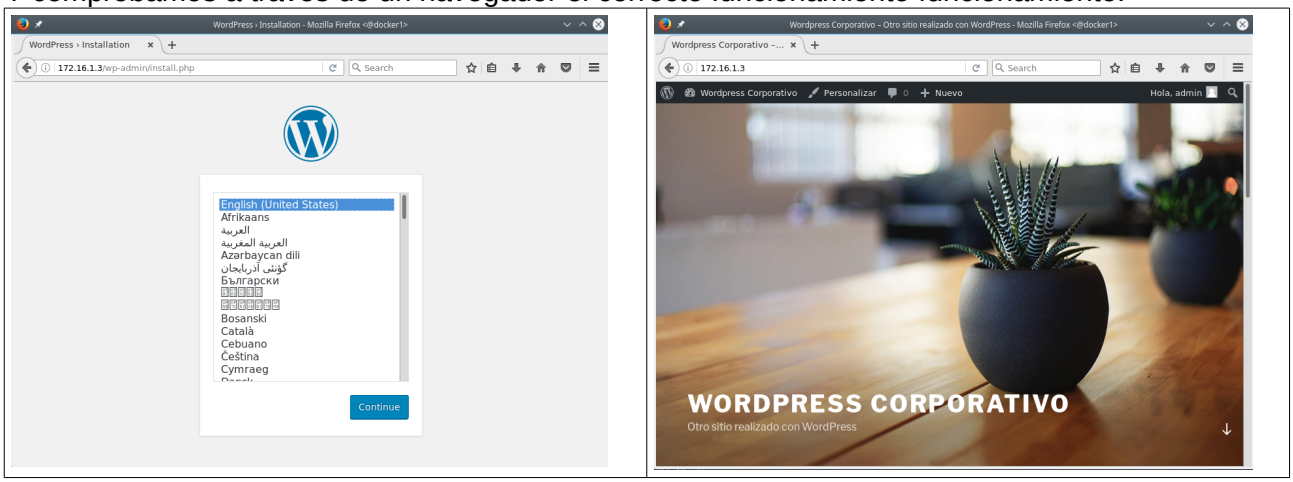
Ahora, para ejecutar nuestro despliegue de Wordpress sólo necesitamos invocarlo desde el directorio donde se encuentra el fichero *docker-compose.yml* del siguiente modo:

```
docker-compose up -d
```

Y comprobamos como se crea un contenedor para el servidor web de Wordpress y otro para la base de datos mysql, con su correspondiente volumen de datos y su red de comunicaciones:

```
root@docker1:/wordpress# docker-compose up -d
Creating volume "wordpress_db_data" with default driver
Creating wordpress_db_1
Creating wordpress_wordpress_1
root@docker1:/wordpress# docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS              PORTS               NAMES
c900e8e0d8d6        wordpress:latest    "docker-entrypoint.sh"   About a minute ago   Up About a minute   80/tcp              wordpress_wordpress_1
7f2be39af36a        mysql:5.7           "docker-entrypoint.sh"   About a minute ago   Up About a minute   3306/tcp            wordpress_db_1
root@docker1:/wordpress# docker network inspect lan-privada
[
    {
        "Name": "lan-privada",
        "Id": "36fdb3337e000d025ff7bca9f904d5ae5f49f021424d7c0a9087915e43afb9de",
        "Scope": "local",
        "Driver": "bridge",
        "EnableIPv6": false,
        "IPAM": {
            "Driver": "default",
            "Options": {},
            "Config": [
                {
                    "Subnet": "172.16.1.0/24",
                    "Gateway": "172.16.1.1"
                }
            ]
        },
        "Internal": false,
        "Containers": {
            "7f2be39af36a347432765e4203c7dd4fad821ed5b0f9f822625fece3359e0009": {
                "Name": "wordpress_db_1",
                "EndpointID": "cbf370bd96a0d674bb67418a33805740bff5ad7d57647c950017766c9e4dc50b",
                "MacAddress": "02:42:ac:10:01:02",
                "IPv4Address": "172.16.1.2/24",
                "IPv6Address": ""
            },
            "c900e8e0d8d682792fd2465ffb1e3435be5b4dbc2095245f4660cd7c7ce7896d": {
                "Name": "wordpress_wordpress_1",
                "EndpointID": "526e945cdf55680942f7bec4888735978cd7bb641da7792459f5b939e86b6e5a",
                "MacAddress": "02:42:ac:10:01:03",
                "IPv4Address": "172.16.1.3/24",
                "IPv6Address": ""
            }
        },
        "Options": {
            "parent": "bond0"
        },
        "Labels": {}
    }
]
```

Y comprobamos a través de un navegador el correcto funcionamiento funcionamiento:



Para parar la aplicación y todos los servicios orquestados que la componen, podemos hacerlo mediante el siguiente comando:

```
docker-compose down -d
```

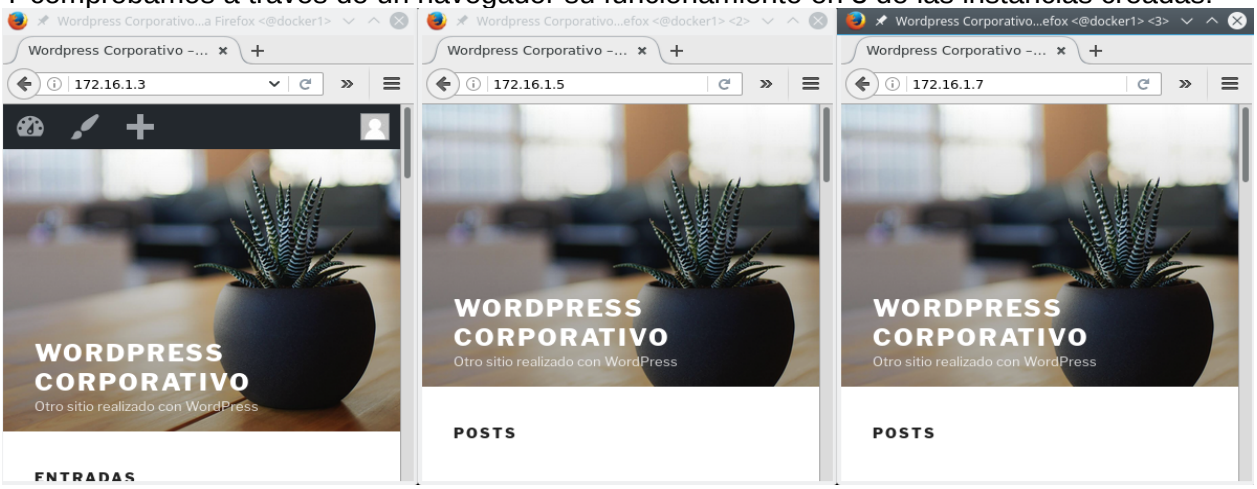
Si ahora, necesitamos escalar el servicio "*wordpress*" (que recordemos que era el que incorporaba el servidor web), podríamos crear hasta 7 nuevas instancias del servidor web de la aplicación Wordpress con la base de datos común con el siguiente comando, simplemente indicando mediante la directiva *scale "nombre del servicio"=numero de instancias*:

```
docker-compose scale wordpress=8
```

Como vemos se crean de forma casi automática 7 nuevas instancias del servicio "*wordpress*" (Para atender, por ejemplo, un pico de demanda de accesos web que pudiese haber):

```
root@docker1:/wordpress# docker-compose scale wordpress=8
Creating and starting wordpress_wordpress_2 ... done
Creating and starting wordpress_wordpress_3 ... done
Creating and starting wordpress_wordpress_4 ... done
Creating and starting wordpress_wordpress_5 ... done
Creating and starting wordpress_wordpress_6 ... done
Creating and starting wordpress_wordpress_7 ... done
Creating and starting wordpress_wordpress_8 ... done
root@docker1:/wordpress# docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS              PORTS               NAMES
e2412db5150a        wordpress:latest    "docker-entrypoint.sh"   47 seconds ago      Up 44 seconds       80/tcp              wordpress_wordpress_8
903177055ed0        wordpress:latest    "docker-entrypoint.sh"   47 seconds ago      Up 44 seconds       80/tcp              wordpress_wordpress_6
e87216a2bd88        wordpress:latest    "docker-entrypoint.sh"   47 seconds ago      Up 45 seconds       80/tcp              wordpress_wordpress_7
c00fbe024492        wordpress:latest    "docker-entrypoint.sh"   47 seconds ago      Up 45 seconds       80/tcp              wordpress_wordpress_4
007f0d6dc49a8       wordpress:latest    "docker-entrypoint.sh"   48 seconds ago      Up 45 seconds       80/tcp              wordpress_wordpress_5
051f80c8aa0d        wordpress:latest    "docker-entrypoint.sh"   48 seconds ago      Up 45 seconds       80/tcp              wordpress_wordpress_2
e82d676b0385        wordpress:latest    "docker-entrypoint.sh"   48 seconds ago      Up 46 seconds       80/tcp              wordpress_wordpress_3
c900e8e0d8d6        wordpress:latest    "docker-entrypoint.sh"   7 minutes ago       Up 7 minutes        80/tcp              wordpress_wordpress_1
7f2be39af36a        mysql:5.7           "docker-entrypoint.sh"   7 minutes ago       Up 7 minutes        3306/tcp            wordpress_db_1
root@docker1:/wordpress#
```

Y comprobamos a través de un navegador su funcionamiento en 3 de las instancias creadas:



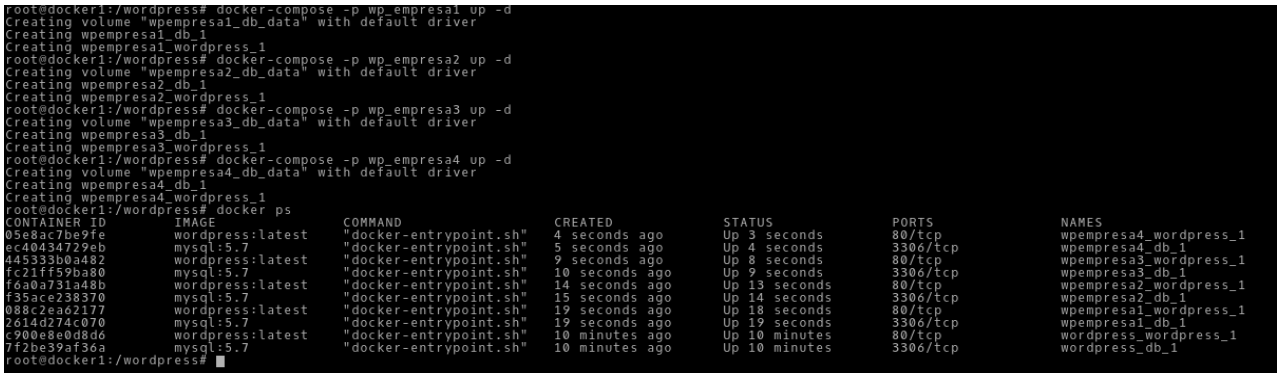
Destacar en este punto, que cada servidor web que se crea para atender peticiones mediante este servicio, se crea como un nuevo servidor web con una dirección ip distinta.
Por este motivo, para que el servicio web tuviese alta disponibilidad y alto rendimiento sería necesario disponer adicionalmente de un balanceador web, un servicio de DNS configurado en Round Robin o una solución de proxy inverso http para tener un punto de entrada de todas la peticiones web y que estas fuesen redirigidas internamente a las direcciones ip de estos servicios nuevos que estamos aprovisionando.
No forma parte del ámbito de este proyecto entrar a definir este tipo de solución y para este ejemplo vamos a partir de que ya disponenos de un servidor proxy inverso http(**) preconfigurado escuchando en el puerto 80 de cada servidor anfitrión (y que recordemos que estaba configurado para recibir las peticiones a través de un DNAT de la direcciones ips públicas que nos facilite cada ISP contratado) de modo que esta preconfigurado para redirigir el trafico web a las direcciones ip internas (de la 172.16.0.3 a la 172.16.0.10 en este ejemplo de escalado).
(**) En los anexos del proyecto incluimos, aprovechando que ya existen soluciones de contenedor preconfiguradas para actuar como proxy inverso y que además estamos desarrollando una solución de contenedores, una aplicación que realiza la función de proxy inverso http que se podría usar para esta arquitectura como una posible solución.

Podríamos además replantear el ejemplo de forma sencilla aprovechando la orquestación.
Si ahora suponemos que en lugar de escalar necesitamos nuevas aplicaciones Wordpress individuales (por ejemplo, que cada departamento de la empresa necesite su propia solución WordPress completa o que formemos parte de un grupo de empresas cada una con su formato comercial), procedemos a partir del mismo fichero *docker-compose.yml* que define nuestro sistema de aplicación orquestada de WordPress, procediendo a crear nuevos servicios WordPress (cada uno con su aplicación WordPress, base de datos, volumen de almacenamiento, etc) indicando simplemente el nombre de cada proyecto con la directiva "*-p nombreproyecto*" :

```
$ docker-compose -p wp_empresa1 up -d
$ docker-compose -p wp_empresa2 up -d
```

```
$ docker-compose -p wp_empresa3 up -d
$ docker-compose -p wp_empresa4 up -d
```

Y como comprobamos, de forma muy sencilla, se crean 4 aplicaciones WordPress distintas cada una de ellas con todos los servicios necesarios para funcionar en sus correspondientes contenedores:



7.2. Ejemplo completo de despliegue de un sistema CMS usando Worpress a través de orquestación dinámica sobre un cluster de múltiples nodos usando Docker Swarm

Vamos a emplear la herramienta Docker Swarm para crear una aplicación orquestada dinámicamente para el despliegue y escalado de los distintos servicios que necesite la aplicación sobre la arquitectura de cluster definida para Docker Swarm.

Dado que en este caso vamos a emplear un cluster de nodos para crear una instalación de la aplicación WordPress, vamos a completar un poco más la instalación básica de WordPress. Como ya hemos visto en el ejemplo anterior, los componentes mínimos para el funcionamiento de WordPress son un servidor Web con capacidad para ejecutar PHP y una base de datos Mysql. Para este ejemplo, vamos a añadir un Servidor Redis[52] (Redis es un motor de base de datos en memoria, basado en el almacenamiento en tablas de hashes (clave/valor)) para que pueda ser usado como servidor de almacenamiento de objetos en memoria de WordPress y vamos a añadir y configurar además (por defecto) un servicio de Memoria Cache Distribuida mediante la implementación de un servidor Memcached[53] y su configuración para que todos los objectos que usa WordPress puedan usar esta memoria cache y el rendimiento del servicio de WordPress aumente.

En primer lugar, creamos una red de overlay propia para usar en los contenedores de este ejemplo, denominada *red_wordpress*, reservando una subred de clase C con el direccionamiento 10.0.5.0/24 :

```
root@docker1:~# docker network create --driver overlay --subnet 10.0.5.0/24
red_wordpress
```

A continuación creamos un volumen de datos para almacenar los datos persistentes que usará el servicio de base de datos Mysql. Como los volúmenes sólo se crean sobre un único nodo del sistema, lo crearemos en el nodo con nombre docker1 (Destacar en este punto que como el volumen con los datos sólo esta en un único nodo, el servicio de base de datos sólo podría estar

52 https://redis.io/
53 https://memcached.org/

sobre este mismo nodo con lo cual tenemos un punto de fallo para el servicio de base de datos que no esta replicado con alta disponibilidad con esta solución).

```
root@docker1:~# docker volume create db_datawp
```

A continuación vamos a definir los siguientes servicios en Docker Swarm, con los nombres que indicaremos a continuación:

- *wpredis*: Servicio que se encarga de gestionar los servidores de Redis que estarán disponibles para su uso por parte de WordPress.
- *dbwordpress*: Servicio que se encarga de gestionar los servidores de base de datos Mysql Versión 5.7
- *wordpress:* Servicio que se encarga de gestionar los servidores de web donde se ejecuta el código WordPress y preconfigurados para usar el servicio de Memcache.
- balancer: Servicio que se encarga de gestionar un balanceador web ,usando el servidor HAProxy[54],para dotar de acceso externo a nuestro balanceador de contenedores hacia los servidores web del WordPress.
- wpmemcached: Servicio que se encarga de gestionar los servidores de Memcached que estarán disponibles para su uso por parte de WordPress.

Veamos ahora los comandos para crear los distintos servicios que hemos usado para este ejemplo:

Creamos primero el servicio *dbwordpress*. Mediante el modificador –mount indicamos que use el volumen previamente creado con el nombre *db_datawp* y que lo mapee internamente en la ruta */var/lib/mysql* del contenedor que se cree para este servicio. Le indicamos tambiñen mediante el envío de variables de entorno, el nombre de la base de datos a crear así como un usuario administrador y su correspondiente contraseña inicial. Mediante el modificador *--network* le indicamos la red de overlay que usaremos para este ejemplo y mediante *--replicas* 1, le indicamos que necesitamos 1 instancia de este servicio. A continuación, mediante la directiva *–constraint*, establecemos que este servicio sólo se pueda ejecutar en el nodo Manager, ya a que es el único nodo donde esta creado en volumen de datos que alberga la base de datos y por lo tanto el único que tendría el mismo y los datos. Finalmente indicamos también el uso de la imagen oficial de Docker para la base de datos Mysql 5.7:

```
root@docker1:~#docker service create --name dbwordpress --mount
type=volume,source=db_datawp,destination=/var/lib/mysql --env
MYSQL_ROOT_PASSWORD=wordpress --env MYSQL_DATABASE=wordpress --env MYSQL_USER=wordpress
--env MYSQL_PASSWORD=wordpress --network red_wordpress --replicas 1 --constraint
'node.role == manager' mysql:5.7
```

Creamos el servicio *wpredis*, indicando la red empleada para este ejemplo e indicando, mediante *–replicas* que necesitamos 4 instancias de este servicio:

```
root@docker1:~# docker service create --name wpredis --network red_wordpress --replicas
4 redis:alpine
```

Creamos el servicio *wpmemcached*, indicando la red empleada para la comunicación e indicando, mediante *–replicas* que necesitamos 4 instancias de este servicio:

```
root@docker1:~# docker service create --name wpmemcached --network red_wordpress
--replicas 4 memcached:alpine
```

A continuación, creamos el servicio *balancer*, que como hemos visto será nuestro punto de entrada al balanceador de carga del servicio *wordpress* que crearemos justo a continuación. Para este servicio, usamos el modificador *–global* para indicar que nos cree este servicio en alta

54 http://www.haproxy.org/

disponibilidad por lo que nos creará una instancia del mismo en cada nodo del cluster y usaremos el modificador *–publish* para publicar en el puerto 80 del servidor anfitrión el puerto 80 de este contenedor (Funcionando como punto de entrada a la red de overlay). Finalmente indicamos que usaremos la imagen docker *jpetazzo/hamba* (Servidor *HAProxy*) y que arrancara escuchando en el puerto 80 y reenviara las peticiones al balanceador de carga del servicio *wordpress.*

```
root@docker1:~#docker service create --name balancer --network red_wordpress --publish
80:80 --mode global jpetazzo/hamba 80 wordpress:80
```

A continuación, necesitamos una imagen con un servidor Web que permita ejecutar PHP y que realice una instalación de WordPress. Existe una imagen oficial de contenedor usando WordPress (Que hemos usado en ejemplos anteriores), pero en este caso queremos configurar el servidor de WordPress para que use nuestro servicio de instancias *Memcached* para cachear objetos y mejorar el rendimiento del entorno, por lo que vamos a componer una imagen nueva de un contenedor a partir de un fichero Dockerfile.
El fichero es el siguiente Dockerfile:

```
FROM php:5.6-apache
MAINTAINER Angel Fernandez
# install the PHP extensions we need
RUN set -ex; \
	\
	apt-get update; \
	apt-get install -y \
		libjpeg-dev \
		libmemcached-dev \
		nfs-common \
		unzip \
		libpng12-dev \
	; \
	rm -rf /var/lib/apt/lists/*; \
	pecl install memcached;\
	pecl install memcache;\
	\
	docker-php-ext-configure gd --with-png-dir=/usr --with-jpeg-dir=/usr; \
	docker-php-ext-enable memcached; \
	docker-php-ext-enable memcache; \
	docker-php-ext-install gd mysqli opcache
# TODO consider removing the *-dev deps and only keeping the necessary lib* packages

# set recommended PHP.ini settings
# see https://secure.php.net/manual/en/opcache.installation.php
RUN { \
		echo 'opcache.memory_consumption=128'; \
		echo 'opcache.interned_strings_buffer=8'; \
		echo 'opcache.max_accelerated_files=4000'; \
		echo 'opcache.revalidate_freq=2'; \
		echo 'opcache.fast_shutdown=1'; \
		echo 'opcache.enable_cli=1'; \
	} > /usr/local/etc/php/conf.d/opcache-recommended.ini

RUN a2enmod rewrite expires

VOLUME /var/www/html

ENV WORDPRESS_VERSION 4.7
ENV WORDPRESS_SHA1 1e14144c4db71421dc4ed22f94c3914dfc3b7020

RUN set -ex; \
	curl -o wordpress.tar.gz -fSL "https://wordpress.org/wordpress-$
{WORDPRESS_VERSION}.tar.gz"; \
	echo "$WORDPRESS_SHA1 *wordpress.tar.gz" | sha1sum -c -; \
# upstream tarballs include ./wordpress/ so this gives us /usr/src/wordpress
```

```
        tar -xzf wordpress.tar.gz -C /usr/src/; \
        rm wordpress.tar.gz; \
        chown -R www-data:www-data /usr/src/wordpress

COPY object-cache.php /usr/src/wordpress/wp-content/
RUN chown -R www-data:www-data /usr/src/wordpress

COPY docker-entrypoint.sh /usr/local/bin/

ENTRYPOINT ["docker-entrypoint.sh"]
CMD ["apache2-foreground"]
```

Una vez creado el fichero Dockerfile y los ficheros adicionales *docker-entrypoint.sh* (donde configuramos que use el balanceador del servicio de memcached: *wpmemcached*) y *object-cache.php, (*plugin WordPress para cachear objectos usando servidores memcached*)* compilamos la imagen y la subimos a un repositorio personal Docker Hub público *ameijeiras/wordpressmemcached* para poder usarla para desplegar nuevos servicios de este imágen.
Ejecutamos en la ruta donde se encuentra el Dockerfile y resto de ficheros necesarios para componer la imagen:

```
root@docker1:~#docker build -t ameijeiras/wordpressmemcached .
root@docker1:~#docker push ameijeiras/wordpressmemcached
```

Tras este proceso ya tenemos la imagen *wordpressmemcached* compilada, publicada y lista para ser instanciada.
Sólo nos falta crear el servicio de web de WordPress con esta imagen. Para ello, creamos el servicio indicando mediante las variables de entorno, los datos de la base de datos del servicio *wpwordpress* creada anteriormente, el número de instancias para este servicio (que empezaremos con 2 instancias) y finalmente indicamos la imagen de WordPress que hemos creado como ejemplo de despliegue:

```
root@docker1:~#docker service create --name wordpress --env
WORDPRESS_DB_HOST=dbwordpress:3306 --env WORDPRESS_DB_PASSWORD=wordpress --network
red_wordpress --replicas 2 ameijeiras/wordpressmemcached:latest
```

A continuación mostramos en resultado de la creación de estos servicios en el cluster de Docker Swarm:

Como de puede apreciar en las capturas de pantalla, podemos ver que se han creado de forma automática todos los servicios indicados y con el número de réplicas que hemos indicado repartidas entre los dos nodos del sistema.

```
root@docker1:~# docker service ls
ID             NAME          REPLICAS  IMAGE                                    COMMAND
12bloky2k94u   wpredis       4/4       redis:alpine
1h2cu2jxqlkd   wordpress     2/2       ameijeiras/wordpressmemcached:latest
3mhhlmzr0199   balancer      global    jpetazzo/hamba                           80 wordpress:80
69ytgdia69nx   wpmemcached   4/4       memcached:alpine
99k4fxzqsaik   dbwordpress   1/1       mysql:5.7
```

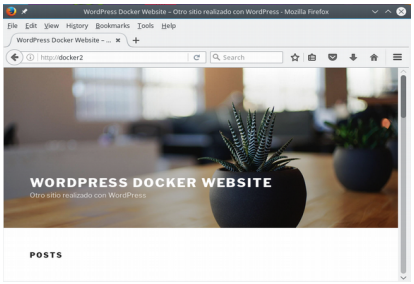
Y comprobamos como los contenedores de los servicios se reparten de forma automática entre los nodos del cluster:
docker1 – nodo1:

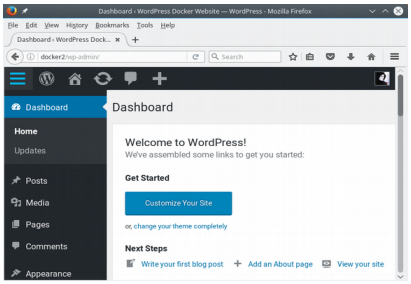
```
root@docker1:~# docker ps
CONTAINER ID        IMAGE                          COMMAND                  CREATED             STATUS              PORTS               NAMES
08d78b00f2d3        jpetazzo/hamba:latest          "hamba 80 wordpress:8"   3 minutes ago       Up 2 minutes                            balancer.0.4nl0qijp3juxbmzmuc18mpgqe
7a3c378fb7d4        redis:alpine                   "docker-entrypoint.sh"   3 minutes ago       Up 2 minutes        6379/tcp            wpredis.3.e60n0hh55mfrpacz98ôyj11vu
f046f00992b4        redis:alpine                   "docker-entrypoint.sh"   3 minutes ago       Up 2 minutes        6379/tcp            wpredis.4.6ao5seo5t1q70zphq7xc3mo5l
680a3ed08edb        memcached:alpine               "docker-entrypoint.sh"   3 minutes ago       Up 2 minutes        11211/tcp           wpmemcached.4.a0dvo82w9zu4mw2rxrem81pbs
3d977e8404ec        memcached:alpine               "docker-entrypoint.sh"   3 minutes ago       Up 2 minutes        11211/tcp           wpmemcached.3.2espsinkxoqzz5gn4il73f9wz
885a47e85fea        mysql:5.7                      "docker-entrypoint.sh"   3 minutes ago       Up 2 minutes        3306/tcp            dbwordpress.1.95vf15n3n5cmtgwaleo2k44r5
root@docker1:~#
```

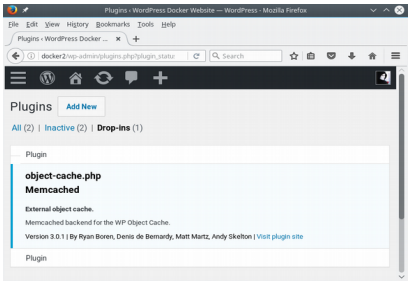
docker2 – nodo2:

```
root@docker2:~# docker ps
CONTAINER ID        IMAGE                                  COMMAND                  CREATED             STATUS              PORTS               NAMES
d5fadfa129e2        jpetazzo/hamba:latest                  "hamba 80 wordpress:8"   About a minute ago  Up About a minute                       balancer.0.5vxzy27x06t9c580c9hevfblv
6dc1648f5b3c        memcached:alpine                       "docker-entrypoint.sh"   About a minute ago  Up About a minute   11211/tcp           wpmemcached.1.2q4m83i2tmwen20kw7jtg7yy9
145fe336f38c        memcached:alpine                       "docker-entrypoint.sh"   About a minute ago  Up About a minute   11211/tcp           wpmemcached.2.4xnsf4ddmue0nnw00cuzmyeje
3dabef226ja5        ameijeiras/wordpressmemcached:latest   "docker-entrypoint.sh"   About a minute ago  Up About a minute   80/tcp              wordpress.1.48dbw1uefq36w4hy1n3m04nlf
0f35ebfe2692        redis:alpine                           "docker-entrypoint.sh"   About a minute ago  Up About a minute   6379/tcp            wpredis.2.1euh9rzbck8wlgzmzuf1py2tm
2943e81d3502        redis:alpine                           "docker-entrypoint.sh"   About a minute ago  Up About a minute   6379/tcp            wpredis.1.7fj0gi0es3h9al4oved62a0wr
ad6ca6648383        ameijeiras/wordpressmemcached:latest   "docker-entrypoint.sh"   About a minute ago  Up About a minute   80/tcp              wordpress.2.5f8q11k06i7r8fm4ammf3ppz1
root@docker2:~#
```

Comprobamos también que podemos acceder a las instancias de wordpress de forma balanceada a través de la ip de uno de los nodos y que la aplicación con todos sus componentes orquestados como servicios funciona correctamente:







Y finalmente, podemos comprobar también como los servidores de memcached están albergando datos de los objetos empleados por WordPress:

```
STAT items:33:crawler_reclaimed 0
STAT items:33:crawler_items_checked 0
STAT items:33:lrutail_reflocked 0
END
stats cachedump 3 100
ITEM 1482708681069273:wp_:transient:doing_cron [33 b; 0 s]
ITEM _:wp_:WP_Object_Cache_global:flush_number [16 b; 0 s]
ITEM 1482708681069273:wp_:transient:feed_mod_b9388c83948825c1edaef0d856b7b109 [10 b; 1482751965 s]
ITEM 1482708681069273:wp_:transient:feed_mod_d117b5738fbd35bd8c0391cda1f2b5d9 [10 b; 1482751964 s]
ITEM 1482708681069273:wp_:transient:feed_mod_ac0b00fe65abe10e0c5b588f3ed8c7ca [10 b; 1482751962 s]
ITEM 1482708681069273:wp_:comment:last_changed [21 b; 0 s]
ITEM 1482708681069273:wp_:category_relationships:1 [14 b; 0 s]
ITEM 1482708681069273:wp_:posts:last_changed [21 b; 0 s]
ITEM 1482708681069273:wp_:terms:last_changed [21 b; 0 s]
```

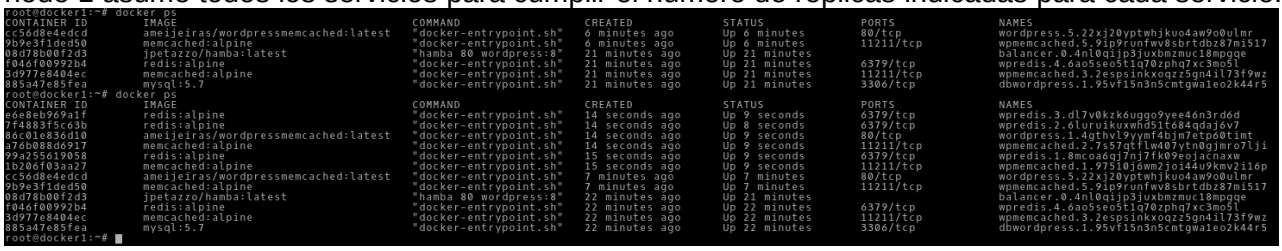
Para finalizar el ejemplo, vamos a realizar la comprobación de las ventajas de este sistema de orquestación mediante dos nuevos escenarios:

Escenario 1: Fallo completo de un nodo

En caso de un fallo completo de un nodo, todas las instancias de los servicios que se estén ejecutando en ese nodo también fallarán y por lo tanto, el número de replicas que nosotros hemos definido para nuestro servicio, en el momento en que el nodo deje de funcionar dejará de cumplirse. Lo que sucede de forma automática, siempre que haya recursos disponibles en alguno de los nodos que continúan levantados, es que Docker Swarm arranca nuevos servicios en los nodos del cluster que continúen funcionando para volver a cumplir el número de instancias de cada servicio que nosotros le hemos indicado.

Para ello, simulamos un fallo completo del nodo 2 o *docker2*, (Apagándolo) y comprobamos como se crean en forma automática en el nodo 1 todos los servicios que estaban ejecutándose en el nodo2.

Podemos ver el la captura de pantalla que se muestra a continuación como, al fallar el nodo 2, el nodo 1 asume todos los servicios para cumplir el número de replicas indicadas para cada servicio:



```
root@docker1:~# docker ps
CONTAINER ID        IMAGE                                      COMMAND                  CREATED             STATUS              PORTS               NAMES
cc56d8e4edcd        ameijeiras/wordpressmemcached:latest       "docker-entrypoint.sh"   6 minutes ago       Up 6 minutes        80/tcp              wordpress.5.22xj20yptwhjkuo4aw9o0ulmr
9b9e3f1ded50        memcached:alpine                           "docker-entrypoint.sh"   6 minutes ago       Up 6 minutes        11211/tcp           wpmemcached.5.9ip9runfwv8sbrtdbz87mi517
08d78b00f2d3        jpetazzo/hamba:latest                      "hamba 80 wordpress:8"   21 minutes ago      Up 21 minutes                           balancer.0.4nl0qijp3juxbnzmuc18mpgqe
f046f00992b4        redis:alpine                               "docker-entrypoint.sh"   21 minutes ago      Up 21 minutes       6379/tcp            wpredis.4.6ao5yeo5t1q70zphq7xc3mo5l
3d977e8404ec        memcached:alpine                           "docker-entrypoint.sh"   21 minutes ago      Up 21 minutes       11211/tcp           wpmemcached.3.2espsinkxoqzz5gn4il73f9wz
885a47e85fea        mysql:5.7                                  "docker-entrypoint.sh"   21 minutes ago      Up 21 minutes       3306/tcp            dbwordpress.1.95vf15n3n5cmtgwa1eo2k44r5
root@docker1:~# docker ps
CONTAINER ID        IMAGE                                      COMMAND                  CREATED             STATUS              PORTS               NAMES
e6e8eb969a1f        redis:alpine                               "docker-entrypoint.sh"   14 seconds ago      Up 9 seconds        6379/tcp            wpredis.3.dl7v0kzk6uggo9yee46n3rd6d
7f4883f5c63b        redis:alpine                               "docker-entrypoint.sh"   14 seconds ago      Up 8 seconds        6379/tcp            wpredis.2.6luruikuxwhd51tt684qdaj6v7
86c01e836d10        ameijeiras/wordpressmemcached:latest       "docker-entrypoint.sh"   14 seconds ago      Up 9 seconds        80/tcp              wordpress.1.4gthvl9yvmf4bjm7ctpo0timt
a76b088d6917        memcached:alpine                           "docker-entrypoint.sh"   14 seconds ago      Up 9 seconds        11211/tcp           wpmemcached.2.7s57qtflw407vtn0gjmro7lji
99a255619058        redis:alpine                               "docker-entrypoint.sh"   15 seconds ago      Up 9 seconds        6379/tcp            wpredis.1.8mcoa6gj7nj7fk09eojacnaxw
1b206f03aa27        memcached:alpine                           "docker-entrypoint.sh"   15 seconds ago      Up 9 seconds        11211/tcp           wpmemcached.1.97510j6wm2joi44u9kmv2ii6p
cc56d8e4edcd        ameijeiras/wordpressmemcached:latest       "docker-entrypoint.sh"   7 minutes ago       Up 7 minutes        80/tcp              wordpress.5.22xj20yptwhjkuo4aw9o0ulmr
9b9e3f1ded50        memcached:alpine                           "docker-entrypoint.sh"   7 minutes ago       Up 7 minutes        11211/tcp           wpmemcached.5.9ip9runfwv8sbrtdbz87mi517
08d78b00f2d3        jpetazzo/hamba:latest                      "hamba 80 wordpress:8"   22 minutes ago      Up 21 minutes                           balancer.0.4nl0qijp3juxbnzmuc18mpgqe
f046f00992b4        redis:alpine                               "docker-entrypoint.sh"   22 minutes ago      Up 22 minutes       6379/tcp            wpredis.4.6ao5yeo5t1q70zphq7xc3mo5l
3d977e8404ec        memcached:alpine                           "docker-entrypoint.sh"   22 minutes ago      Up 22 minutes       11211/tcp           wpmemcached.3.2espsinkxoqzz5gn4il73f9wz
885a47e85fea        mysql:5.7                                  "docker-entrypoint.sh"   22 minutes ago      Up 22 minutes       3306/tcp            dbwordpress.1.95vf15n3n5cmtgwa1eo2k44r5
root@docker1:~#
```

Escenario 2: Necesidad de reescalar los servicios

En caso de que las necesidades de nuestros servicios cambien, el escalado de los mismos es sumamente sencillo y basta con cambiar sobre el mismo servicio la configuración del número de tareas o instancias que necesitamos.

Así, vamos a suponer que tras la definición inicial, vemos que para el servicio *wpredis*, para el cual habíamos definido 4 instancias, no estamos haciendo uso de las mismas por lo que queremos

reducir el número de 2. Además de esto, hemos realizado una campaña de marketing sobre la plataforma de WordPress instalada y comenzamos a recibir un gran número de peticiones web.
Para poder hacer frente a este incremento que se prevé puntual de peticiones, sobre el servicio wordpress, que habíamos definido inicialmente en 2 instancias vamos a redimensionarlo a 6 instancias y sobre el servicio memcached, inicialmente en 4, vamos a redimensionarlo también a 6 servidores para poder cachear más objectos y de forma más distribuida.

Escalamos el entorno con los siguientes comandos:

```
root@docker1:~#docker service scale wpredis=2
root@docker1:~#docker service scale wpmemcached=6
root@docker1:~#docker service scale wordpress=6
```

Comprobamos como los servicios cambian como hemos indicado:

```
root@docker1:~# docker service ls
ID            NAME          REPLICAS  IMAGE                                      COMMAND
12bloky2k94u  wpredis       2/2       redis:alpine
1h2cu2jxqlkd  wordpress     6/6       ameijeiras/wordpressmemcached:latest
3mhhlmzr0199  balancer      global    jpetazzo/hamba                             80 wordpress:80
69ytgdia69nx  wpmemcached   6/6       memcached:alpine
99k4fxzqsaik  dbwordpress   1/1       mysql:5.7
root@docker1:~#
```

Y comprobamos como los contenedores de los servicios se escalan y reparten de forma automática entre los nodos del cluster:

docker1 – nodo1:

```
root@docker1:~# docker ps
CONTAINER ID        IMAGE                                      COMMAND                  CREATED             STATUS              PORTS               NAMES
4c2cd9119065        ameijeiras/wordpressmemcached:latest       "docker-entrypoint.sh"   About a minute ago  Up About a minute   80/tcp              wordpress.6.774sp7n6uwh5todjo4b5pyxp3
cc56d8e4edcd        ameijeiras/wordpressmemcached:latest       "docker-entrypoint.sh"   About a minute ago  Up About a minute   80/tcp              wordpress.5.22xj20yptwhjkuo4aw9o0ulmr
7c9791c87876        memcached:alpine                           "docker-entrypoint.sh"   About a minute ago  Up About a minute   11211/tcp           wpmemcached.6.9tzkoesmrzayfjd4vr97gmnlr
9b9e3f1ded50        memcached:alpine                           "docker-entrypoint.sh"   About a minute ago  Up About a minute   11211/tcp           wpmemcached.5.9ip9runfwv8sbrtdbz87mi517
08d78b00f2d3        jpetazzo/hamba:latest                      "hamba 80 wordpress:8"   16 minutes ago      Up 16 minutes                           balancer.0.4nl0qijp3juxbmzmuc18mpgqe
f046f00992b4        redis:alpine                               "docker-entrypoint.sh"   16 minutes ago      Up 16 minutes       6379/tcp            wpredis.4.6ao5seo5t1q70zphq7xc3no5l
680a3ed08edb        memcached:alpine                           "docker-entrypoint.sh"   16 minutes ago      Up 16 minutes       11211/tcp           wpmemcached.4.a0dvo82w9zu4mw2rxrem81pbs
3d977e8404ec        memcached:alpine                           "docker-entrypoint.sh"   16 minutes ago      Up 16 minutes       11211/tcp           wpmemcached.3.2espsinkxoqzz5gn4il73f9wz
885a47e85fea        mysql:5.7                                  "docker-entrypoint.sh"   16 minutes ago      Up 16 minutes       3306/tcp            dbwordpress.1.95vf15n3n5cmtgwa1eo2k44r5
root@docker1:~#
```

docker2 – nodo2:

```
root@docker2:~# docker ps
CONTAINER ID        IMAGE                                      COMMAND                  CREATED             STATUS              PORTS               NAMES
01f85b1da999        ameijeiras/wordpressmemcached:latest       "docker-entrypoint.sh"   About a minute ago  Up About a minute   80/tcp              wordpress.4.cw1h719ifxvpalnibv6l903fw
84798f37d70d        ameijeiras/wordpressmemcached:latest       "docker-entrypoint.sh"   About a minute ago  Up About a minute   80/tcp              wordpress.3.6rgfec45igsmcpu8h5joqbxfq
d5fadfa129e2        jpetazzo/hamba:latest                      "hamba 80 wordpress:8"   16 minutes ago      Up 16 minutes                           balancer.0.5vxzy27x06f9c580c9hevfblv
6dc1648f5b3c        memcached:alpine                           "docker-entrypoint.sh"   16 minutes ago      Up 16 minutes       11211/tcp           wpmemcached.1.2q4m83i2tmwen20kw7jtg7yy9
145fe336f38c        memcached:alpine                           "docker-entrypoint.sh"   16 minutes ago      Up 16 minutes       11211/tcp           wpmemcached.2.4xnsf4ddmue0nnwb0cuznveje
3dabef2263a5        ameijeiras/wordpressmemcached:latest       "docker-entrypoint.sh"   16 minutes ago      Up 16 minutes       80/tcp              wordpress.1.48dbw1uefq36w4hyin3m04nlf
0f35ebfe2692        redis:alpine                               "docker-entrypoint.sh"   16 minutes ago      Up 16 minutes       6379/tcp            wpredis.2.1euh9rzbck8wlgzmzuf1py2tm
ad0ca6648383        ameijeiras/wordpressmemcached:latest       "docker-entrypoint.sh"   16 minutes ago      Up 16 minutes       80/tcp              wordpress.2.5f8q11k0i17r8fm4ammf3ppz1
root@docker2:~#
```

Como hemos podido comprobar, Docker Swarm ofrece un sistema de orquestación dinámica de aplicaciones donde, una vez configurados los servicios que necesitamos integrar para nuestra aplicación, garantiza la escalabilidad y alta disponibilidad de un sistema de cluster.

8. Conclusiones

Como hemos podido comprobar a lo largo de es proyecto, partimos de los requisitos base de una empresa que tenia un centro de datos con servidores propios, con personal para mantenerlo y que pretendía encontrar y validar un sistema de código abierto que les permitiese implantar un sistema de orquestación de aplicaciones.
Como ha quedado patente, se han conseguido todos los objetivos iniciales que se planteaban al inicio del mismo.
Para ello, en primer lugar se ha realizado un análisis de las distintas tecnologías de virtualización clásica y de virtualización de contenedores, comparándolas y analizando cual era la solución por la que debíamos optar para la orquestación de aplicaciones. Una vez visto que la solución pasaba por el uso de la virtualización de contenedores, se estudio la tecnología subyacente que la hacia posible, para poder entender su funcionamiento y posibles limitaciones. A continuación se analizaron varias alternativas de aplicaciones de virtualización y orquestación, con la premisa inicial del uso de software de código abierto, que nos permitieran acometer una solución, hasta decantarnos finalmente por la herramienta Docker y su ecosistema de aplicaciones como solución para conseguir los objetivos planteados.
Al mismo tiempo que se selecciono la aplicación, se planteo una arquitectura complementaria para el despliegue de la solución Docker.
Y llegados a este punto, se realizo un análisis exhaustivo del funcionamiento de todo el ecosistema de aplicaciones de Docker, analizando y estudiando su funcionamiento y documentándolo con ejemplos prácticos.
Considerando que existen dos tipos de orquestaciones, la orquestación estática y la orquestación dinámica, analizamos en profundidad dos de las herramientas del ecosistema Docker que permiten dar solución a los objetivos que nos planteábamos: Comprobamos como Docker Compose nos permitía orquestar servicios de manera estática y como Docker Swarm nos permitía lo mismo pero esta vez para orquestados dinámicamente con lo cual, la solución propuesta cumple todas la condiciones iniciales que nos marcamos.
Finalmente se realizaron dos ejemplos prácticos completos de cada una de las distintas orquestaciones, cada una con su herramienta particular (Docker Compose y Docker Swarm) pudiendo verificar que la elección de la solución Docker nos permitió dar una solución completa que incluso dotaba a nuestro proyecto de posibilidades de escalado y orquestación con una sencillez más allá de las esperadas personalmente al comienzo del mismo.

8.1. Continuidad del proyecto

Al haber analizado un sistema para la orquestación estática y otro para la orquestación dinámica de aplicaciones, con todas las tecnologías necesarias para su funcionamiento, el ámbito del proyecto abarca un importante número de áreas donde se podrían introducir nuevos estudios para futuras mejoras del proyecto.
Algunas de ellas que podemos citar sin entrar en mayor detalle podrían ser:

Alta disponibilidad del nodo mánager:
En nuestro proyecto, dado que partimos de dos nodos o servidores físicos para la implantación del sistema y no formaba parte del ámbito estricto del proyecto, no se planteo una solución de alta

disponibilidad a nivel del entorno de orquestación por lo que este apartado podría suponer un área de trabajo de cara al crecimiento futuro de nodos en la solución.

Alta disponibilidad de los volúmenes de datos:
Como vimos en los ejemplos prácticos, actualmente los volúmenes de datos son una importante limitación a la hora de implantar soluciones escalables y de alta disponibilidad ya que, sólo están disponibles en el nodo físico en donde son creados, por lo que para garantizar alta disponibilidad de algunos servicios se debería de plantear una solución que permita a los volúmenes de datos de Docker existir en múltiples nodos del cluster.

Sistema de monitorización y backups de contenedores y servicios:
No hemos comentado ningún aspecto especifico sobre la monitorización y sistema de backup de los contenedores y servicios orquestados, pero al tratarse de aspectos claves para cualquier aplicación corporativa sería un tema de estudio importante. Dado que la orquestación de un servicio conlleva muchos procesos automáticos es sistema de backup y monitorización de los contenedores y servicios implicaría un estudio más en profundidad.

8.2. Valoración económica

Para el cálculo del coste hora de los perfiles TI empleados, hemos aplicado los rangos salariales anuales en base a la guía salarial del Sector TI 2015-2016 elaborada por una reconocida consultora de RRHH especializada en el ámbito de las tecnologías de la información (*Vitae Consultores*). Guía Salarial 2015-2016

[Vitae]

Se ha realizado una breve estimación económica, separada en 3 bloques principales. Por un lado, hemos realizado una estimación económica del coste del hardware inicial planteado en los requisitos. En otro bloque hemos calculado el coste de implantar la solución y la puesta en marcha de la misma, incluyendo la instalación y configuración de los dos primeros nodos del cluster, así como el coste adicional de aprovisionar nuevos nodos al sistema. En el último bloque hemos recogido una valoración económica de la operación y mantenimiento del servicio que al igual que en el bloque anterior incluye el coste de los dos primeros nodos del sistema y el coste de mantenimiento de nodos adicionales.

Los cálculos se han realizado en base al precio coste para los perfiles de Técnico de sistemas Medio y Técnico de sistemas Experto (En calidad de Analista de sistemas), según la guía salarial indicada, sin Impuestos ni costes estructurales de la empresa y teniendo en cuenta el calendario laboral Español, resultando la siguiente base de coste/hora:

Categoria	Coste Anual	Días/Año	Horas/Día	Coste/Hora
Analista de sistemas	36000	225	8	20
Técnico de sistemas Senior	22000	225	8	12,222222222

Y en base a este coste/hora, resultan los costes finales como se muestran a continuación:

Coste Hardware			
Descripción	Coste	Unidades	Total
Dell Poweredge R430	2500	2	5000
Dell PowerConnect x1026	320	2	640
			5640

Coste de instalación y puesta en marcha:

Puesta en marcha (Incluyendo 2 nodos)			
Categoria	Coste/hora	Horas	Total
Analista de sistemas	20	30	600
Técnico de sistemas Senior	13	120	1560
			2160

Instalación Nodo adicional al cluster			
Categoria	Coste/hora	Horas	Total
Analista de sistemas	20	1	20
Técnico de sistemas	13	10	130
			150

Coste de Operación y mantenimiento:

Operación y mantenimiento (Incluyendo 2 nodos)			
Categoria	Coste/hora	Horas	Total
Analista de sistemas	20	1	20
Técnico de sistemas	13	8	104
			124

Cada Nodo adicional			
Categoria	Coste/hora	Horas	Total
Analista de sistemas	20	0,5	10
Técnico de sistemas	13	2	26
			36

8.3. Acceso a descargas del entorno de demostración.

Como complemento al proyecto, se ha creado una simulación con maquinas virtuales (Usando Oracle VirtualBox) de los dos nodos principales del sistema, así como de la infraestructura de red (Bridge, Bonding, etc.) de modo que se ajustase en la medida de lo posible a la infraestructura propuesta. Este entorno podría emplearse también como entorno de preproducción donde probar cambios, versiones distintas para probar actualizaciones, etc.
Esta simulación se ha exportado como un servicio virtualizado usando el formato Open Virtualization Format (OVF) y para emplearla sólo es necesario importarla a través de la aplicación Oracle VirtualBox.
Incluye dos servidores virtuales que conforman dos nodos de un cluster de Docker Swarm y están identificados como *docker1* (Nodo Manager) y *docker2* (Nodo Worker).

La credenciales de acceso a los nodos son las siguientes:

Usuario del sistema	Contraseña
root	administrador
administrador	administrador

Dentro del nodo *Manager* en la ruta */home/administrador/docker* se pueden encontrar todos los ficheros de Docker Compose, Docker Swarm y Dockerfile empleados en el proyecto, listos para ser probados.

Podemos descargar el entorno a través del siguiente enlace:

Enlace a la descarga del fichero OVF
https://drive.google.com/a/ameijeiras.es/file/d/0B2Chj9SEWmHOcXdkemEzdnNsU1k/view?usp=sharing

Adicionalmente, incluimos también un enlace para la descarga de los ficheros de código Dockerfile, docker-compose, etc. usados en este proyecto como ejemplos prácticos:

Enlace a la descarga de los ejemplos prácticos
https://drive.google.com/a/ameijeiras.es/file/d/0B2Chj9SEWmHObzNLVm9TN0Zob1k/view?usp=sharing

9. Anexos

9.1. Especificaciones técnicas del los servidores propuestos

[url3]

PowerEdge R430

Powerful, two-socket entry rack server delivers outstanding performance, configuration flexibility, high availability and intuitive management in a short-height (1U), short-depth (24-inch) chassis.

Designed for rack environments requiring peak two-socket performance, sizeable internal storage capacity and short chassis depth to overcome space constraints, the PowerEdge R430 rack server is an excellent fit for high-performance computing (HPC), web tech and infrastructure scale-out. The R430 also serves well for collaboration and productivity applications, surveillance and site security, and is likewise attractive as a dedicated backup or development server.

Deliver peak performance

Drive powerful performance across a wide range of workloads with the latest Intel® Xeon® processor E5-2600 v4 product family. Accelerate performance and grow memory capacity throughput with 12 DIMM slots and DDR4 memory. Boost I/O performance with up to 10 high-IOPS hard drives and two PCIe 3.0 I/O slots, driving 2x data throughput compared to previous generations.

Maximize operational efficiency

Accelerate time to production and drive better ROI by automating deployment with integrated Dell Remote Access Controller 8 (iDRAC8) with Lifecycle Controller. Save time for IT administrators with intuitive, automatable tools for monitoring and update. Control energy budgets with energy-efficient processors, memory, power supplies and Fresh Air 2.0 capability.

Discover greater versatility

Deploy powerful performance into space-constrained environments with short-height (1U), short-depth (24-inch) chassis. Harness data explosion with scalable internal storage capacity — up to 10 x 2.5" hard drives. Adapt flexibly to changing workload conditions with an expandable platform ready for virtualization and high-availability clustering.

Innovative management with intelligent automation

The Dell OpenManage systems management portfolio includes innovative solutions that simplify and automate essential server lifecycle management tasks – making IT operations more efficient and Dell servers the most productive, reliable and cost effective. Leveraging the incomparable agent-free capabilities of the PowerEdge embedded iDRAC with Lifecycle Controller technology, server deployment, configuration and updates are streamlined across the OpenManage portfolio and through integration with third-party management solutions.

Monitoring and control of Dell and third-party data center hardware is provided by OpenManage Essentials and with anytime, anywhere mobile access, through OpenManage Mobile. OpenManage Essentials now also delivers Server Configuration Management capabilities that automate one-many PowerEdge bare-metal server and OS deployments, quick and consistent replication of configurations and ensure compliance to a predefined baseline with automated drift detection.

PowerEdge R430

- Short-height, short-depth chassis
- Latest Intel Xeon E5-2600 v4 processors
- Up to 12 x DDR4 DIMMs
- 2 x PCIe 3.0 I/O slots
- Up to 10 x 2.5" HDDs

Feature	PowerEdge R430 technical specification
Form factor	1U rack server
Processor	Intel® Xeon® processor E5-2600 v4 product family **Processor sockets**: 2 **Chipset**: C610 **Internal interconnect**: Up to 9.6GT/s **Cache**: 2.5MB per core; core options: 4, 6, 8, 10, 12, 14, 16, 18, 20
Memory	DDR4 DIMMs at up to 2400MT/s 12 DIMM slots: 4GB/8GB/16GB/32GB
I/O slots	2 PCIe 3.0
Storage	SAS, SATA, nearline SAS, SSD Multiple R430 chassis available: • 10 x 2.5" hot-plug drives • 8 x 2.5" hot-plug drives • 4 x 3.5" hot-plug drives • 4 x 3.5" cabled drives
RAID controllers	**Internal controllers**: PERC S130 (SW RAID), PERC H330, PERC H730, PERC H730P **External HBAs (RAID)**: PERC H830
Communications	4 x 1GbE LOM Click here for R430 supported network interface cards (NICs) and host bus adapters (HBAs) and scroll to "Additional Network Cards" section.
Power supplies	450W, 550W PSU
Systems management	**Systems management:** IPMI 2.0 compliant; Dell OpenManage Essentials; Dell OpenManage Mobile; Dell OpenManage Power Center **Remote management** iDRAC8 with Lifecycle Controller, iDRAC8 Express (default), iDRAC8 Enterprise (upgrade) 8GB vFlash media (upgrade), 16GB vFlash media (upgrade) **Dell OpenManage Integrations:** • Dell OpenManage Integration Suite for Microsoft® System Center • Dell OpenManage Integration for VMware® vCenter™ **Dell OpenManage Connections:** • HP Operations Manager, IBM Tivoli® Netcool® and CA Network and Systems Management • Dell OpenManage Plug-in for Oracle® Database Manager
Optional supported hypervisors	Citrix® XenServer, VMware vSphere® ESXi®, Red Hat® Enterprise Virtualization®
Operating systems	Microsoft Windows Server® 2008 R2 Microsoft Windows Server 2012 Microsoft Windows Server 2012 R2 Novell® SUSE® Linux Enterprise Server Red Hat Enterprise Linux VMware vSphere ESXi For more information on the specific versions and additions, visit Dell.com/OSsupport.
Rack support	ReadyRails™ II sliding rails for tool-less mounting in 4-post racks with square or unthreaded round holes or tooled mounting in 4-post threaded hole racks, with support for optional tool-less cable management arm.
OEM-ready version available	From bezel to BIOS to packaging, your servers can look and feel as if they were designed and built by you. For more information, visit Dell.com/OEM.
Recommended support	Dell ProSupport Plus for critical systems or Dell ProSupport for premium hardware and software support for your PowerEdge solution. Consulting and deployment offerings are also available. Contact your Dell representative today for more information. Availability and terms of Dell Services vary by region. For more information, visit Dell.com/ServiceDescriptions.

End-to-end technology solutions

Reduce IT complexity, lower costs and eliminate inefficiencies by making IT and business solutions work harder for you. You can count on Dell for end-to-end solutions to maximize your performance and uptime. A proven leader in Servers, Storage and Networking, Dell Enterprise Solutions and Services deliver innovation at any scale. And if you're looking to preserve cash or increase operational efficiency, Dell Financial Services™ has a wide range of options to make technology acquisition easy and affordable. Contact your Dell Sales Representative for more information.**

Learn More at Dell.com/PowerEdge.

April 2016 | Version 2.0
Dell_PowerEdge_R430_SpecSheet

9.2. Especificaciones técnicas del los switch propuestos

Dell Networking serie X

Switches de 1 o 10 GbE con una interfaz de usuario intuitiva diseñada para optimizar las aplicaciones de red en nube e in situ

La serie X de Dell Networking es una familia de switches de Ethernet de 1 GbE y 10 GbE de gestión inteligente diseñados para empresas pequeñas y medianas que requieren un control de red de clase empresarial fusionado con una facilidad de uso tipo usuario. Los switches de la serie X cuentan con una gran variedad de números de puertos, opciones de alimentación por Ethernet (PoE) y de implementación. La configuración y la gestión se simplificaron de forma significativa con una interfaz gráfica de usuario (GUI) y un diseño de hardware intuitivos. Una amplia gama de modelos representa una capacidad de implementación hecha a su medida, que incluye la unidad compacta de 8 puertos diseñada para un montaje en el techo, la pared o un escritorio y que goza de un diseño inteligente.

Innovaciones prácticas para redes pequeñas

Herramientas eficaces dentro de una interfaz elegante con una funcionalidad similar a la de las aplicaciones hacen que usar los switches de la serie X sea un placer. Los comandos y alertas conocidos similares a los de las computadoras y servidores hacen que haya menos términos por aprender y que se adquiera más conocimiento. Conecta, configura automáticamente y alimenta teléfonos VoIP y puntos de acceso inalámbricos con las opciones de PoE.

Navegación elegante con un flujo de trabajo eficiente e intuitivo

Todos los diseños, desde la navegación y accesos hasta las estructuras de los menús y los consejos de ayuda fueron inspirados en la forma en que los profesionales de la tecnología de la información piensan y trabajan. Las herramientas fluidas, los asistentes que brindan ayuda paso a paso y un panel personalizable hacen que la configuración y el calibrado de los switches sean rápidos y exactos. Las tareas comunes, las alertas, los estatus de los puertos y la visualización de la red se muestran en la hermosa pantalla del panel.

Visibilidad de tráfico y control en tiempo real únicos

Optimice los servicios de nube y las aplicaciones de red in situ con las funciones de prioridad de seguridad y tráfico. Vea el tráfico de red y luego de supervisar, empiece a resolver en una secuencia continua. La selección única de puertos múltiples para rutinas por lotes y los perfiles de puertos para dispositivos frecuentes eliminan la necesidad de pasos adicionales y los errores de configuración.

Características clave

- Familia de switches de nivel 2+ de 1 GbE y 10 GbE con compatibilidad para alimentación por Ethernet (PoE/PoE+) opcional
  - » Switch de 8 puertos de 1 GbE compacto y sin ventiladores
  - » Un diseño compacto con PoE, 8 puertos y de 1 GbE para una colocación flexible en oficinas (modelo sin PoE)
  - » Switches de 26 y 18 puertos con un ancho de rack medio y con dos puertos de enlace ascendente SFP de 1 GbE
  - » Switches de 52 puertos con ancho de rack y con cuatro puertos de enlace ascendente SFP+ de 10 GbE
  - » Modelo de 12 puertos de 10 GbE para una conexión a servidor y almacenamiento de alta velocidad, o agregado de redes
- Diseño de interfaz gráfica de usuario revolucionario para una configuración y una "supervisión práctica" fáciles
  - » Herramientas eficaces dentro de una interfaz elegante con una funcionalidad similar a la de las aplicaciones
  - » Herramientas fluidas, asistentes que brindan ayuda paso a paso y un panel personalizable
  - » Tareas comunes, alertas, estatus de puertos y visualización de la red en un solo panel
  - » Optimice los servicios de nube y las aplicaciones de red in situ con las funciones de prioridad de seguridad y tráfico.
  - » Vea el tráfico de red y luego de supervisar, empiece a resolver en una secuencia continua.
  - » La selección de puertos múltiples para rutinas por lotes y los perfiles de puertos para dispositivos frecuentes eliminan la necesidad de pasos adicionales y los errores de configuración.
- Bastidor en paralelo para dos modelos intercambiables de 26 puertos en un rack
- Fresh Air 2.0 de Dell brinda un gran rendimiento y un uso eficiente de energía
- Conector de cierre y puerto de la consola

Explicación: **E** - estándar, **D** - disponible, **N** - no disponible

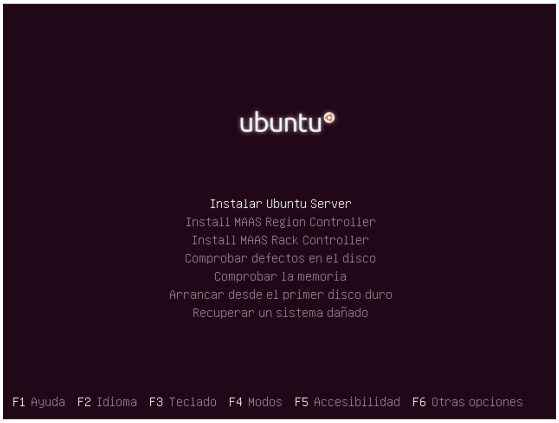
Atributos de puertos	X1008/P	X1018/P	X1026/P	X1052/P	X4012
Switches de GbE de detección automática de 10/100/1000BASE-T	8	16	24	48	N
Puertos de fibra SFP de 1 Gb	N	2	2	N	N
Puertos de fibra SFP+ de 10 Gb	N	N	N	4	12
Interfaces con PoE	8 PoE, con hasta un total de 123 W (X1008P)	16 PoE, con hasta un total de 246 W (X1018P)	24 PoE/PoE, con hasta un total de 369 W (X1026P)	24 PoE/PoE, con hasta un total de 369 W (X1052P)	N
Con alimentación PoE	S (X1008)	N	N	N	N
Reducción de alimentación para cables cortos o conexiones inactivas	S	S	S	S	N
Negociación automática velocidad, modo dúplex y control de flujo	S	S	S	S	N
Modo MDI/MDIX automático y control de flujo	S	S	S	S	N
Rendimiento	**X1008/P**	**X1018/P**	**X1026/P**	**X1052/P**	**X4012**
Capacidad de la estructura del switch	Hasta 16 Gbps	Hasta 36 Gbps	Hasta 52 Gbps	Hasta 176 Gbps	Hasta 240 Gbps
Tasa de reenvío	11,9 Mpps	26,8 Mpps	38,7 Mpps	131 Mpps	178,6 Mpps
Direcciones MAC	16.000	16.000	16.000	16.000	16.000
Memoria de búfer de paquetes	8 Mb	8 Mb	8 Mb	8 Mb	8 Mb
Calidad del servicio	**X1008/P**	**X1018/P**	**X1026/P**	**X1052/P**	**X4012**
Colas de prioridad por puerto	4	4	4	8	8
Administración	**X1008/P**	**X1018/P**	**X1026/P**	**X1052/P**	**X4012**
Supervisión SNMP limitada y administración CLI. Para obtener más información, consulte la guía del usuario	S	S	S	S	Supervisión SNMP completa
Chasis	**X1008/P**	**X1018/P**	**X1026/P**	**X1052/P**	**X4012**
Dimensiones (alto x ancho x profundidad)	42,5 x 151,13 x 151,13 mm (1,67 x 5,95 x 5,95 pulgadas)	X1018: 41,25 x 209,0 x 250,0 mm (1,62 x 8,23 x 9,84 pulgadas) X1018P: 41,25 x 209,0 x 450,0 mm (1,62 x 8,23 x 17,72 pulgadas)	X1026: 41,25 x 209,0 x 250,0 mm (1,62 x 8,23 x 9,84 pulgadas) X1026P: 41,25 x 209,0 x 450,0 mm (1,62 x 8,23 x 17,72 pulgadas)	X1052: 43,5 x 434,0 x 270,0 mm (1,71 x 17,1 x 10,63 pulgadas) X1052P: 43,5 x 434,0 x 407,0 mm (1,71 x 17,1 x 16,0 pulgadas)	41,25 x 209,0 x 250,0 mm (1,62 x 8,23 x 9,84 pulgadas)
Montaje en rack	N	1 RU, ancho medio	1 RU, ancho medio	1 RU	1 RU, ancho medio
Peso de las unidades	X1008: 0,80 kg X1008P: 0,83 kg	X1018: 1,76 kg X1018P: 3,21 kg	X1026: 1,88 kg X1026P: 3,80 kg	X1052: 3,80 kg X1052P: 6,00 kg	2,03 kg
Ventiladores	Diseño >	X1018: diseño sin ventiladores X1018P: 2 (parte posterior)	X1026: diseño sin ventiladores X1026P: 2 (parte posterior)	X1052: 2 (parte posterior) X1052P: 4 (parte posterior)	2 (parte posterior)
Condiciones ambientales de funcionamiento	**X1008/P**	**X1018/P**	**X1026/P**	**X1052/P**	**X4012**
100% libre de plomo	Si	Si	Si	Si	Si
Temperatura en estado operativo	0 a 50 °C (32 a 122 °F)	0 a 50 °C (32 a 122 °F)	0 a 50 °C (32 a 122 °F)	0 a 50 °C (32 a 122 °F)	0 a 50 °C (32 a 122 °F)
Temperatura de almacenamiento	-20 a 70 °C (-4 a 158 °F)	-20 a 70 °C (-4 a 158 °F)	-20 a 70 °C (-4 a 158 °F)	-20 a 70 °C (-4 a 158 °F)	-20 a 70 °C (-4 a 158 °F)
Humedad relativa en funcionamiento	10 a 90% sin condensación	10 a 90% sin condensación	10 a 90% sin condensación	10 a 90% sin condensación	10 a 90% sin condensación
Humedad relativa en almacenamiento:	10 a 80% sin condensación	10 a 80% sin condensación	10 a 80% sin condensación	10 a 80% sin condensación	10 a 80% sin condensación
Alimentación	**X1008/P**	**X1018/P**	**X1026/P**	**X1052/P**	**X4012**
Fuente de alimentación	X1008: 24 W (externa) X1008P: 150 W (externa)	X1018 (40 W) X1018P: 280 W	X1026 (40 W) X1026P: 450 W	X1052: 100 W X1052P 525 W	100 W
Alimentación (máx.)	X1008: 9,7 W X1008P: 13,8 W (PoE de +124 W)	X1018: 21,5 W X1018P: 27,3 W (PoE de +247 W)	X1026 (30 W) X1026P: 37 W (PoE de +370 W)	X1052: 75 W X1052P: 82 W (PoE de +370 W)	81 W
BTU/h	X1008: 33,1 BTU/h X1008P: 47,1 BTU/h	X1018: 74 BTU/h X1018P: 93,2 BTU/h	X1026: 102 BTU/h X1026P: 126,3 BTU/h	X1052: 256 BTU/h X1052P: 279,8 BTU/h	276,4 BTU/h

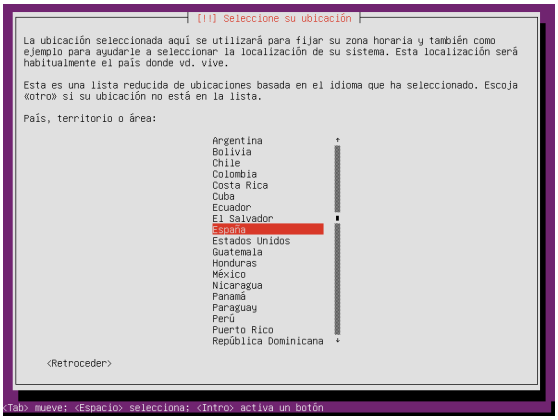
[url2]

9.3. Instalación sistema operativo Ubuntu 16.04 LTS Server

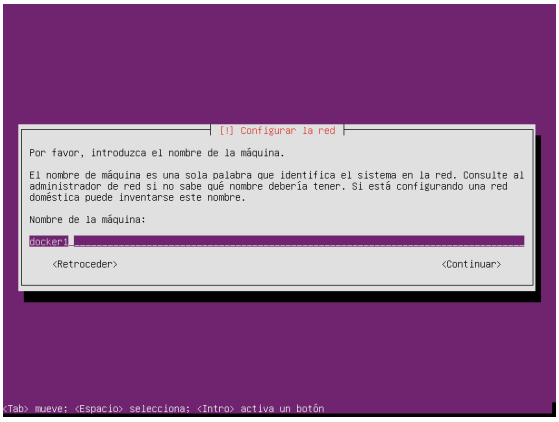
[url1]

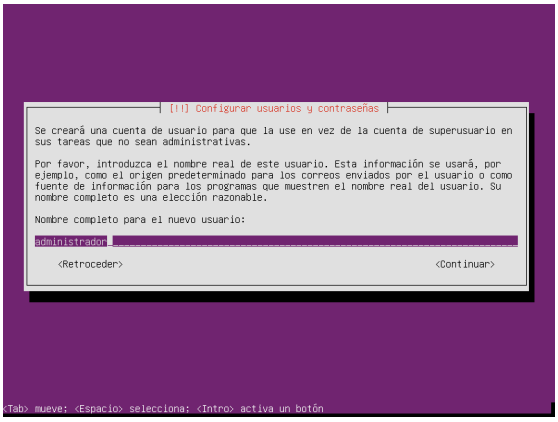
La instalación del sistema operativo Ubuntu 16.04 Server sobre un servidor es un proceso ampliamente documentado. En el siguiente enlace podemos encontrar la documentación completa e información del proceso, facilitada por el fabricante (Canonical).
Indicamos a continuación cuales serían los pasos principales e imágenes del proceso, sobre la instalación de los servidores virtuales usados para las pruebas en este proyecto.
Primero sólo hay que arrancar el sistemas desde su medio de instalación (cdrom, usb, etc), y hacer la primeras selecciones básicas de idioma y teclado:



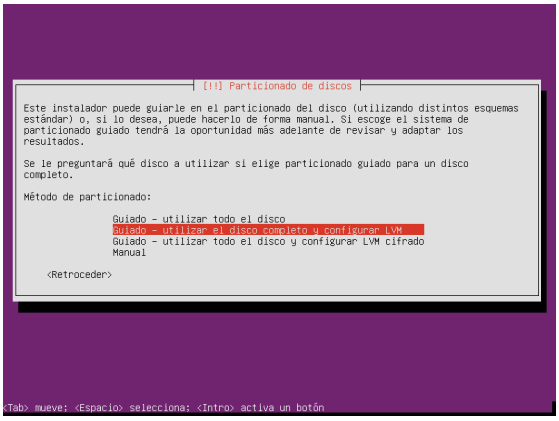


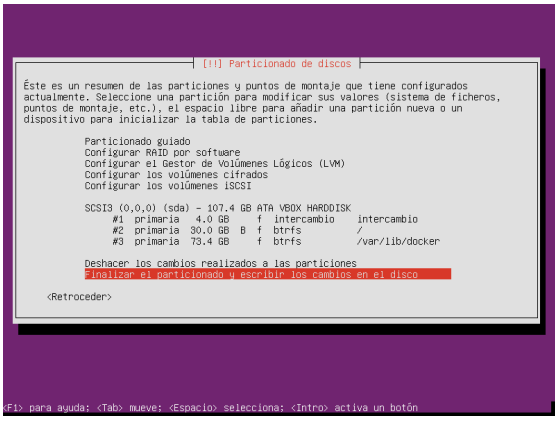
A continuación configuramos las opciones de red, nombre de hosts y los datos del usuario administrador:



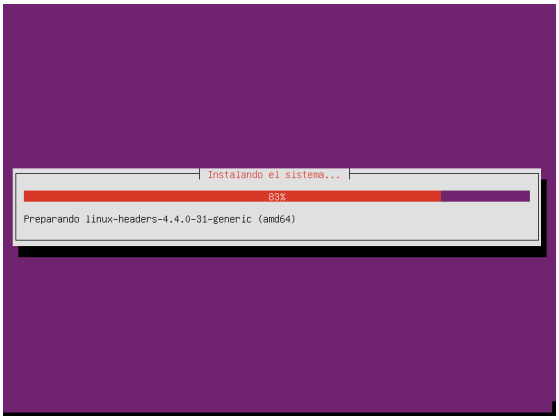


Particionamos el disco según las especificaciones definida previamente para esta arquitectura:





Y finalmente, tras aproximadamente 8 minutos de instalación del software base preconfigurado de fábrica, el sistema termina de instalar y se reinicia. Tras esto, sistema operativo arranca ya instalado y listo para empezar a configurar.



9.4. Instalación y configuración de un *bonding* de red en Ubuntu.

[url4]

A continuación se detalla el proceso de instalación de bonding sobre el sistema operativo que se realizaría sobre cada uno de los servidores que forman parte de nuestro cluster de orquestación de servicios. Como hemos indicado cada servidor cuenta con 4 interfaces de red y la configuración de este bonding de red la realizaremos sobre el 3 y 4 interfaz de red que habitualmente podemos encontrarlo identificado en el sistema operativo Linux con los nombre de interfaz eth2 y eth3 y que nosotros usaremos como convención para identificarlos en los siguientes pasos.

A continuación mostramos los pasos para su configuración:

1. Se instala el paquete de bonding en el sistema:

```
sudo apt update
sudo apt install ifenslave-2.6
```

2. Se cargar el módulo de bonding en el arranque:

Se edita el fichero /etc/modules para configurar el módulo de bonding en el arranque del sistema:

```
# /etc/modules: kernel modules to load at boot time.
#
# This file contains the names of kernel modules that should be loaded
# at boot time, one per line. Lines beginning with "#" are ignored.


bonding
```

3. Se carga el módulo de bonding:

```
sudo systemctl restart networking
```

4. Se configuran los interfaces de red:

Se edita el fichero /etc/network/networking para configurar los interfaces eth2 y eth3 como el nuevo interfaz de red bond0. Una vez configurado el interface bond0, configuramos el direccionamiento ip correspondiente:

```
#eth2
auto eth2
iface eth2 inet manual
```

```
    bond-master bond0

#eth3
auto eth3
iface eth3 inet manual
    bond-master bond0

# bond0 es la nueva interfaz NIC que se us ará
# en este ejemplo usamos la dirección ip que correspondería al
# servidor 1 de las ilustraciones
auto bond0
iface bond0 inet static
    address 172.16.0.2
    netmask 255.255.255.0
    gateway 172.16.0.1
    # Configuramos bond0 en modo 6
    bond-mode 6
    bond-miimon 100
    bond-lacp-rate 1
    bond-slaves none
```

5. Se reinicia la red:

```
sudo systemctl restart networking
```

9.5. Solución Proxy inverso HTTP

Un proxy inverso http es una solución que suele implementarse en redes heterogéneas donde tenemos una zona de red expuesta a Internet y otras zonas de red internas o DMZ. Este sistema permite atender todas la peticiones de tipo http que le lleguen por la interfaz pública, tratarlas y dirigirlas hacia otros servidores http que no están expuestos.

Para este ejemplo, vamos a usar un servidor web NGINX que puede trabajar en modo de proxy inverso como una posible solución para poder exponer un servidor web en el puerto 80 de una red pública o con entrada de Internet y redirigir las peticiones a uno o varios servidores internos en otras redes no alcanzables de modo directo.
Veamos un breve descripción de los principales pasos y configuración de como se implementaría.
Partimos de un fichero *default.conf* de NGINX. Necesitamos que reciba peticiones en el puerto 80 y las reenvié a 8 servidores internos con las ips: 172.16.1.3, 172.16.1.4, 172.16.1.5, 172.16.1.6, 172.16.1.7, 172.16.1.8, 172.16.1.9 y 172.16.1.10 a sus correspondientes puertos 80. Para ello creamos el fichero que mostramos a continuación:

```
server {
    listen       80;
    server_name  localhost;

    location / {
        #root   /usr/share/nginx/html;
        #index  index.html index.htm;

        add_header X-Upstream $upstream_addr;
    }

    #error_page  404              /404.html;

    # redirect server error pages to the static page /50x.html
    #
    error_page   500 502 503 504  /50x.html;
    location = /50x.html {
```

```
        root   /usr/share/nginx/html;
    }

}
upstream wordpressapp {
    least_conn;
    server 172.16.1.3:80;
    server 172.16.1.4:80;
    server 172.16.1.5:80;
    server 172.16.1.6:80;
    server 172.16.1.7:80;
    server 172.16.1.8:80;
    server 172.16.1.9:80;
    server 172.16.1.10:80;
}
```

Este fichero es una plantilla por defecto del servicio NGINX donde, primero definimos un "*upstream wordpressapp"* con la configuración de los servidores internos hacia los que hay que redirigir e indicamos la directiva "least_conn" que se encarga de repartir las peticiones a los distintos servidores en función del que tenga menos número de conexiones, configurando un balanceador de carga. A continuación, indicamos en la configuración la directiva "*proxy_pass http://wordpressapp;*" que lo que hace es configurar el proxy inverso para redirigir ese trafico hacia el *upstream* previamente configurado.
En caso de necesitar nuevas redirecciones internas sólo sería necesario definir nuevos "*upstreams"* y su correspondiente regla de proxy inverso.

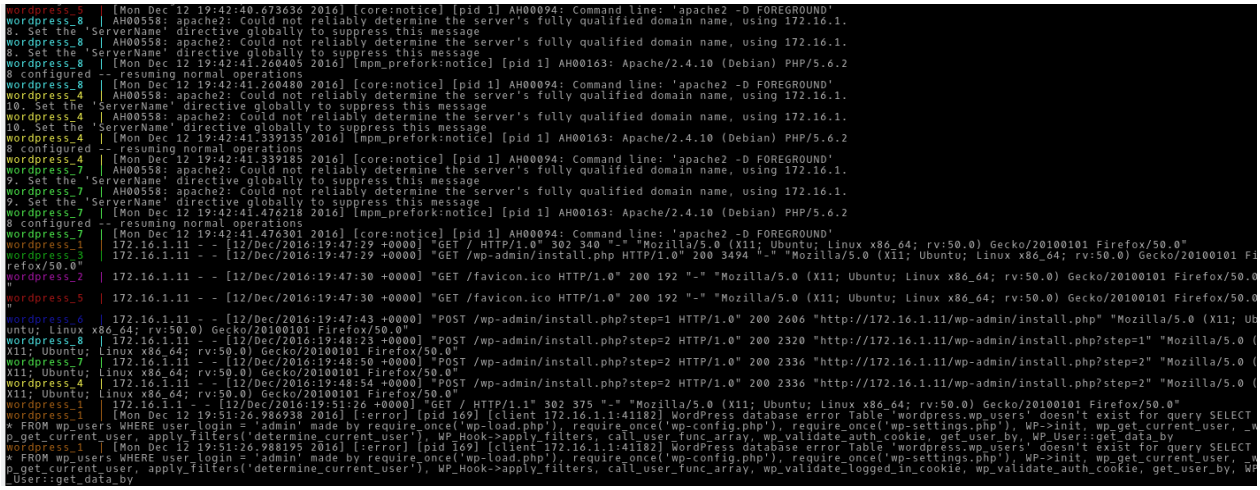
Como tenemos implementado un sistema de orquestación de contenedores, optamos por desplegar un contenedor desde la imagen oficial del servidor NGINX que ejecutamos del siguiente modo:

```
$ docker run -itd --name "ProxyHttpd" --network=lan-privada -p 80:80
-v /nginx/default.conf:/etc/nginx/conf.d/default.conf nginx:latest
```

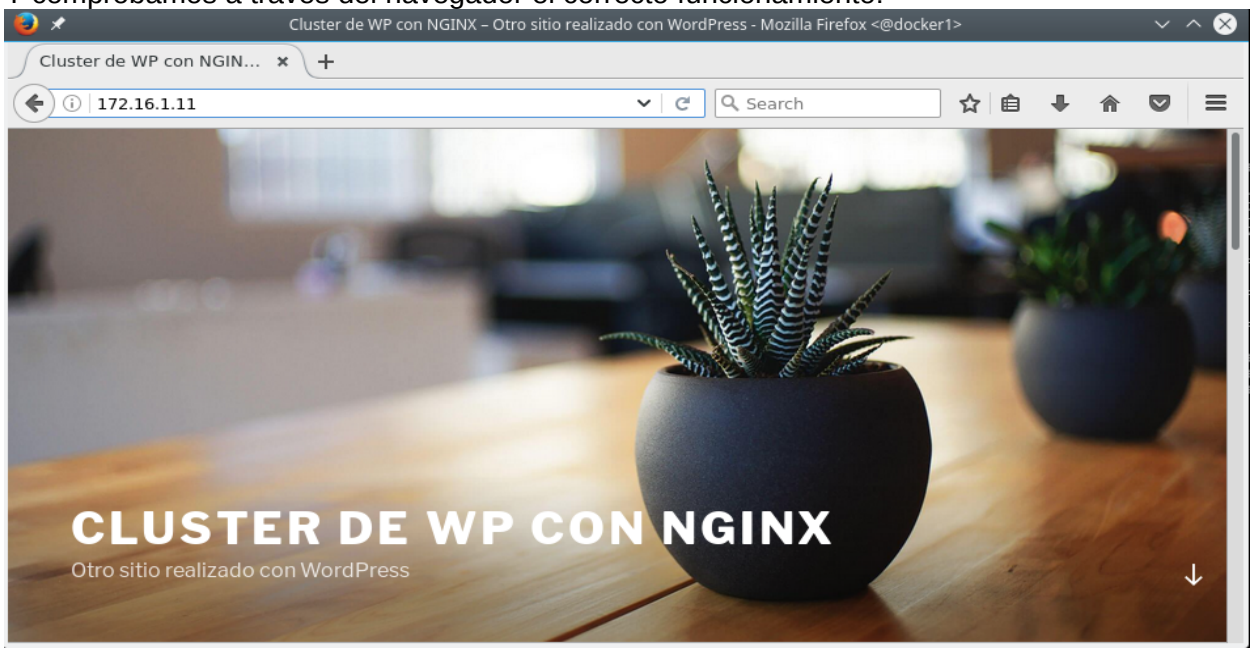
Se ejecuta el contenedor de NGINX, mapeando el puerto 80 del contenedor con el de la máquina virtual (Que a nuestra arquitectura su vez está expuesto a Internet a través de un DNAT con la ip pública de un ISP) y sobre el cual mapeamos el fichero de configuración que acabamos de crear con la configuración interna que usa de NGINX.

Vemos una captura de la ejecución del ejemplo usado en este proyecto para exponer y balancear las peticiones web hace un cluster de 8 servidores web WordPress internos:

Y a continuación vemos como los distintos contenedores reciben peticiones web a las ips balanceadas:



Y comprobamos a través del navegador el correcto funcionamiento:



[nginx1]

[nginx2]

[nginx3]

9.6. Guía rápida de comandos Docker.

En este anexo, incluimos una guía rápida con una breve explicación de los comandos principales de las aplicaciones del ecosistema Docker:

Docker management commands

Command	Description
dockerd	Launch the Docker daemon
info	Display system-wide information
inspect	Return low-level information on a container or image
version	Show the Docker version information

Image commands

Command	Description
build	Build an image from a Dockerfile
commit	Create a new image from a container's changes
history	Show the history of an image
images	List images
import	Import the contents from a tarball to create a filesystem image
load	Load an image from a tar archive or STDIN
rmi	Remove one or more images
save	Save images to a tar archive
tag	Tag an image into a repository

Container commands

Command	Description
attach	Attach to a running container
cp	Copy files/folders from a container to a HOSTDIR or to STDOUT
create	Create a new container
diff	Inspect changes on a container's filesystem
events	Get real time events from the server
exec	Run a command in a running container
export	Export a container's filesystem as a tar archive
kill	Kill a running container
logs	Fetch the logs of a container
pause	Pause all processes within a container
port	List port mappings or a specific mapping for the container
ps	List containers
rename	Rename a container
restart	Restart a running container
rm	Remove one or more containers
run	Run a command in a new container
start	Start one or more stopped containers
stats	Display a live stream of container(s) resource usage statistics
stop	Stop a running container
top	Display the running processes of a container
unpause	Unpause all processes within a container
update	Update configuration of one or more containers
wait	Block until a container stops, then print its exit code

Hub and registry commands

Command	Description
login	Register or log in to a Docker registry
logout	Log out from a Docker registry
pull	Pull an image or a repository from a Docker registry
push	Push an image or a repository to a Docker registry
search	Search the Docker Hub for images

Network and connectivity commands

Command	Description
network connect	Connect a container to a network
network create	Create a new network
network disconnect	Disconnect a container from a network
network inspect	Display information about a network
network ls	Lists all the networks the Engine `daemon` knows about
network rm	Removes one or more networks

Shared data volume commands

Command	Description
volume create	Creates a new volume where containers can consume and store data
volume inspect	Display information about a volume
volume ls	Lists all the volumes Docker knows about
volume rm	Remove one or more volumes

Swarm node commands

Command	Description
node promote	Promote a node that is pending a promotion to manager
node demote	Demotes an existing manager so that it is no longer a manager
node inspect	Inspect a node in the swarm
node update	Update attributes for a node
node ps	List tasks running on a node
node ls	List nodes in the swarm
node rm	Remove one or more nodes from the swarm

Swarm swarm commands

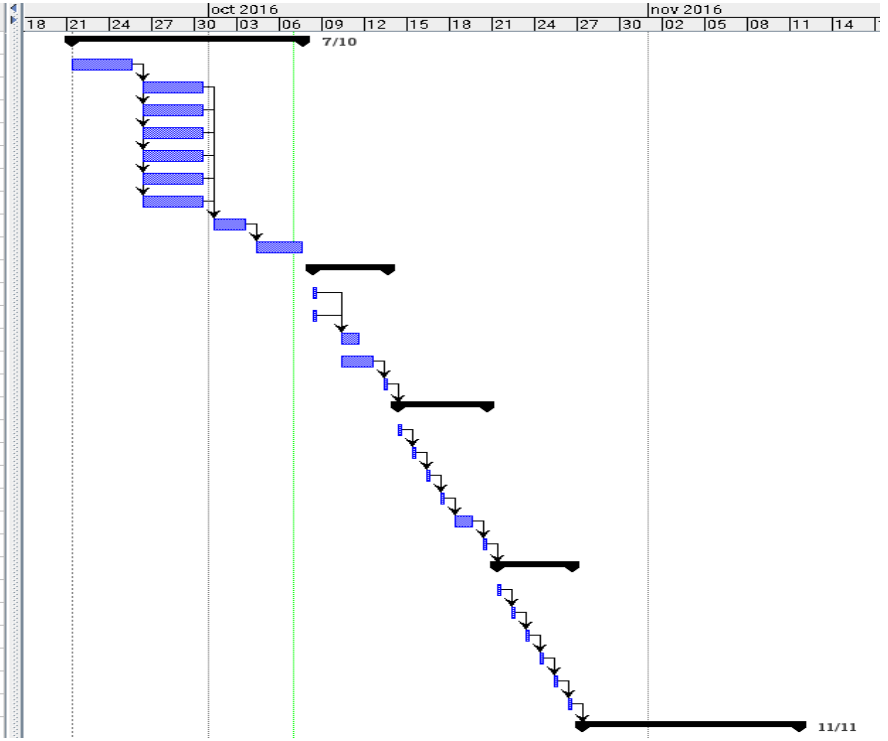
Command	Description
swarm init	Initialize a swarm
swarm join	Join a swarm as a manager node or worker node
swarm leave	Remove the current node from the swarm
swarm update	Update attributes of a swarm
swarm join-token	Display or rotate join tokens

Swarm service commands

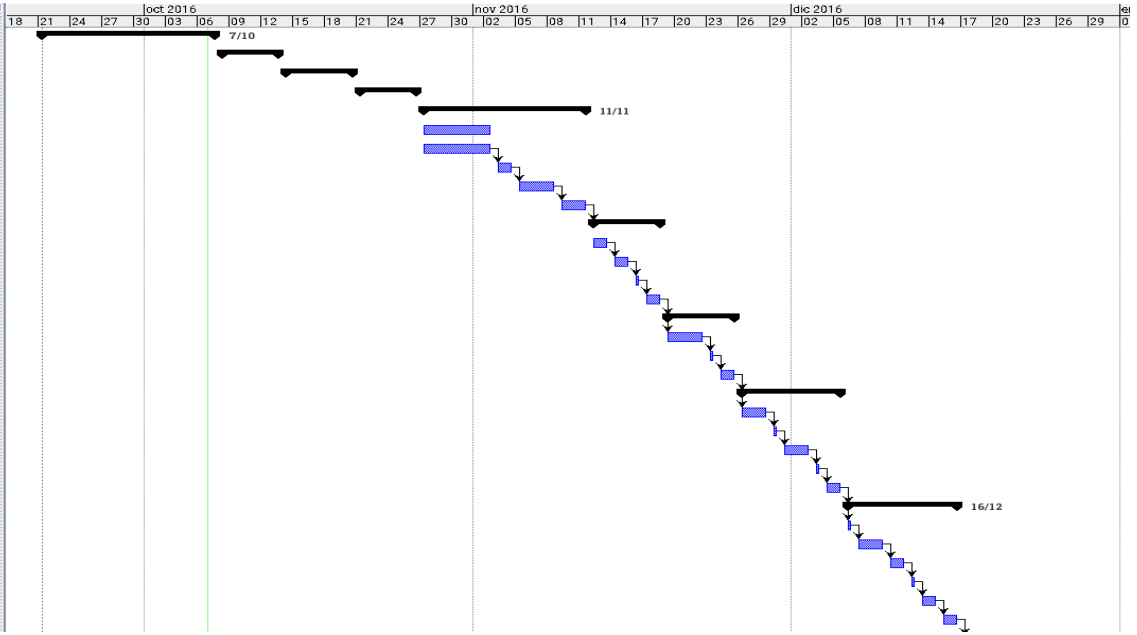
Command	Description
service create	Create a new service
service inspect	Inspect a service
service ls	List services in the swarm
service rm	Remove a service from the swarm
service scale	Set the number of replicas for the desired state of the service
service ps	List the tasks of a service
service update	Update the attributes of a service

9.7. Cronograma completo

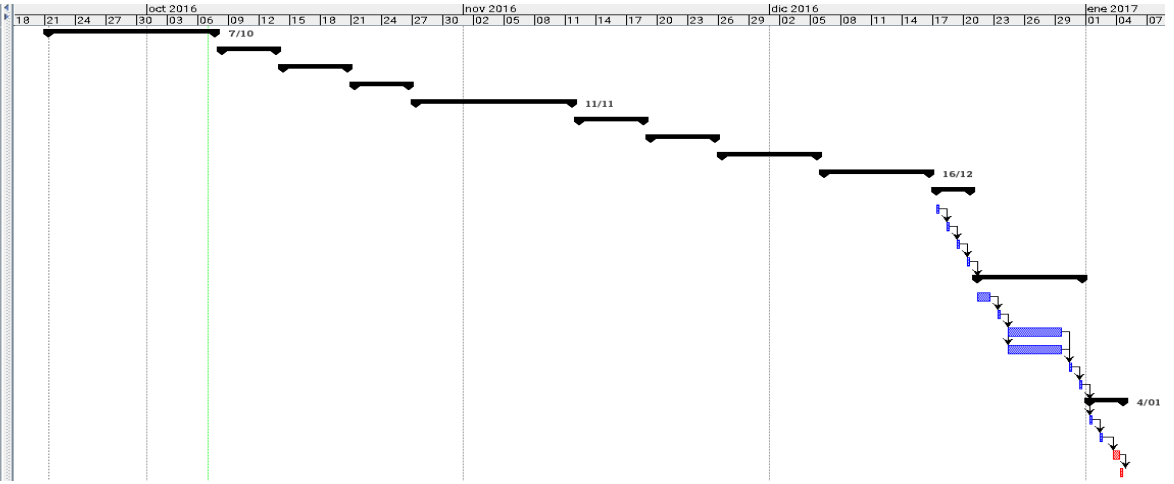
	Nombre	Inicio	Terminado
1	**Tarea 1: Desarrollo de un plan de trabajo**	**21/09/16 9:00**	**7/10/16 17:00**
2	Puesta en marcha	21/09/16 9:00	25/09/16 17:00
3	Diseño de portada	26/09/16 9:00	30/09/16 17:00
4	Descripción del proyecto	26/09/16 9:00	30/09/16 17:00
5	Justificación del proyecto	26/09/16 9:00	30/09/16 17:00
6	Motivación del proyecto	26/09/16 9:00	30/09/16 17:00
7	Objetivos del proyecto	26/09/16 9:00	30/09/16 17:00
8	Requisitos del proyecto	26/09/16 9:00	30/09/16 17:00
9	Desglose de tareas	1/10/16 9:00	3/10/16 17:00
10	Diseño de cronograma	4/10/16 9:00	7/10/16 17:00
11	**Tarea 2: Análisis técnico previo de la solución global del proyecto**	**8/10/16 9:00**	**13/10/16 17:00**
12	Búsqueda y recopilación de información general sobre virtualización de contenedores	8/10/16 9:00	8/10/16 17:00
13	Búsqueda y recopilación de información general sobre orquestación de aplicaciones	8/10/16 9:00	8/10/16 17:00
14	Estudio y análisis de la información	10/10/16 9:00	11/10/16 17:00
15	Prueba e instalación de distintas herramientas	10/10/16 9:00	12/10/16 17:00
16	Revisión de requisitos, tiempos y alcance del proyecto	13/10/16 9:00	13/10/16 17:00
17	**Tarea 3: Análisis y documentación de las arquitecturas de virtualización**	**14/10/16 9:00**	**20/10/16 17:00**
18	Búsqueda y recopilación de información sobre virtualización y cloud	14/10/16 9:00	14/10/16 17:00
19	Búsqueda y recopilación de información sobre tendencias en virtualización	15/10/16 9:00	15/10/16 17:00
20	Análisis y recopilación de cifras y diseño de histograma	16/10/16 9:00	16/10/16 17:00
21	Revisión de requisitos, tiempos y alcance del proyecto	17/10/16 9:00	17/10/16 17:00
22	Estudio y análisis de la información	18/10/16 9:00	19/10/16 17:00
23	Redacción de la memoria	20/10/16 9:00	20/10/16 17:00
24	**Tarea 4: Análisis y búsqueda de información sobre virtualización de contenedores**	**21/10/16 9:00**	**26/10/16 17:00**
25	Búsqueda y documentación de tecnologías de virtualizacion de contenedores	21/10/16 9:00	21/10/16 17:00
26	Análisis y documentacion de las tecnologías subyacentes	22/10/16 9:00	22/10/16 17:00
27	Búsqueda, comparativa y análisis sobre herramientas de virtualización	23/10/16 9:00	23/10/16 17:00
28	Búsqueda de información sobre distintas herramientas del mercado	24/10/16 9:00	24/10/16 17:00
29	Estudio y análisis de la información	25/10/16 9:00	25/10/16 17:00
30	Redacción de la memoria	26/10/16 9:00	26/10/16 17:00
31	**Tarea 5: Análisis y estudio de la herramienta Docker**	**27/10/16 9:00**	**11/11/16 17:00**



	Nombre	Inicio	Terminado
1	**Tarea 1: Desarrollo de un plan de trabajo**	**21/09/16 9:00**	**7/10/16 17:00**
11	**Tarea 2: Análisis técnico previo de la solución global del proyecto**	**8/10/16 9:00**	**13/10/16 17:00**
17	**Tarea 3: Análisis y documentación de las arquitecturas de virtualización**	**14/10/16 9:00**	**20/10/16 17:00**
24	**Tarea 4: Análisis y búsqueda de información sobre virtualización de contenedores**	**21/10/16 9:00**	**26/10/16 17:00**
31	**Tarea 5: Análisis y estudio de la herramienta Docker**	**27/10/16 9:00**	**11/11/16 17:00**
32	Búsqueda y documentación de la herramienta Docker	27/10/16 9:00	2/11/16 17:00
33	Búsqueda y documentación sobre los componentes tecnológicos de Docker	27/10/16 9:00	2/11/16 17:00
34	Búsqueda y documentación sobre la arquitectura Docker	3/11/16 9:00	4/11/16 17:00
35	Prueba de soluciones y ejemplos de uso	5/11/16 9:00	8/11/16 17:00
36	Redacción de la memoria	9/11/16 9:00	11/11/16 17:00
37	**Tarea 6: Análisis y documentación del ecosistema Docker**	**12/11/16 9:00**	**18/11/16 17:00**
38	Búsqueda y documentación sobre ecosistema de aplicaciones Docker	12/11/16 9:00	13/11/16 17:00
39	Prueba de las herramientas del ecosistema Docker e implantación	14/11/16 9:00	15/11/16 17:00
40	Análisis del API de Docker y documentacion	16/11/16 9:00	16/11/16 17:00
41	Redacción de la memoria	17/11/16 9:00	18/11/16 17:00
42	**Tarea 7: Pruebas y ejemplos de funcionamiento del ecosistema Docker**	**19/11/16 9:00**	**25/11/16 17:00**
43	Prueba de soluciones y ejemplos de uso	19/11/16 9:00	22/11/16 17:00
44	Análisis y documentación sobre aspectos de seguridad	23/11/16 9:00	23/11/16 17:00
45	Redacción de la memoria y conclusiones	24/11/16 9:00	25/11/16 17:00
46	**Tarea 8: Análisis y búsqueda de información sobre orquestación de contenedores**	**26/11/16 9:00**	**5/12/16 17:00**
47	Búsqueda y documentación sobre orquestación de aplicaciones	26/11/16 9:00	28/11/16 17:00
48	Definición de la arquitectura para la herramienta elegida	29/11/16 9:00	29/11/16 17:00
49	Configuración, instalación y puesta en marcha de la solucíon de orquestación	30/11/16 9:00	2/12/16 17:00
50	Búsqueda y documentación sobre aspectos de escalabilidad y alta disponibilidad	3/12/16 9:00	3/12/16 17:00
51	Redacción de la memoria	4/12/16 9:00	5/12/16 17:00
52	**Tarea 9: Implementación de la solución y ejemplos de uso**	**6/12/16 9:00**	**16/12/16 17:00**
53	Análisis de los posibles ejemplos de casos de uso	6/12/16 9:00	6/12/16 17:00
54	Documentación de la instalación y puesta en marcha	7/12/16 9:00	9/12/16 17:00
55	Documentación de las configuraciones especificas	10/12/16 9:00	11/12/16 17:00
56	Elaboración de distintos ejemplos de uso y despliegue	12/12/16 9:00	12/12/16 17:00
57	Elaboración de distintos ejemplos de orquestación de aplicaciones	13/12/16 9:00	14/12/16 17:00
58	Redacción de la memoria	15/12/16 9:00	16/12/16 17:00



	Nombre	Inicio	Terminado
1	**Tarea 1: Desarrollo de un plan de trabajo**	**21/09/16 9:00**	**7/10/16 17:00**
11	**Tarea 2: Análisis técnico previo de la solución global del proyecto**	**8/10/16 9:00**	**13/10/16 17:00**
17	**Tarea 3: Análisis y documentación de las arquitecturas de virtualización**	**14/10/16 9:00**	**20/10/16 17:00**
24	**Tarea 4: Análisis y búsqueda de información sobre virtualización de contenedores**	**21/10/16 9:00**	**26/10/16 17:00**
31	**Tarea 5: Análisis y estudio de la herramienta Docker**	**27/10/16 9:00**	**11/11/16 17:00**
37	**Tarea 6: Análisis y documentación del ecosistema Docker**	**12/11/16 9:00**	**18/11/16 17:00**
42	**Tarea 7: Pruebas y ejemplos de funcionamiento del ecosistema Docker**	**19/11/16 9:00**	**25/11/16 17:00**
46	**Tarea 8: Análisis y búsqueda de información sobre orquestación de contenedores**	**26/11/16 9:00**	**5/12/16 17:00**
52	**Tarea 9: Implementación de la solución y ejemplos de uso**	**6/12/16 9:00**	**16/12/16 17:00**
59	**Tarea 10: Conclusiones**	**17/12/16 9:00**	**20/12/16 17:00**
60	Alcance y análisis de viabilidad	17/12/16 9:00	17/12/16 17:00
61	Elaboración de la implantación y desarrollo	18/12/16 9:00	18/12/16 17:00
62	Elaboración de la planificación, coste y desviaciones	19/12/16 9:00	19/12/16 17:00
63	Elaboración del documento de continuidad del proyecto	20/12/16 9:00	20/12/16 17:00
64	**Tarea 11: Elaboración de la presentación**	**21/12/16 9:00**	**31/12/16 17:00**
65	Recopilación de información para la presentación	21/12/16 9:00	22/12/16 17:00
66	Elaboración del estilo maestro de la diapositiva	23/12/16 9:00	23/12/16 17:00
67	Elaboración de capturas e imágenes	24/12/16 9:00	29/12/16 17:00
68	Elaboración de la presentación	24/12/16 9:00	29/12/16 17:00
69	Elaboración de las notas de la presentación	30/12/16 9:00	30/12/16 17:00
70	Revisión final y definición de transiciones	31/12/16 9:00	31/12/16 17:00
71	**Tarea 12: Revisión final del proyecto. Bibliografía y Anexos**	**1/01/17 9:00**	**4/01/17 17:00**
72	Elaboración y recopilación de la bibliografía	1/01/17 9:00	1/01/17 17:00
73	Elaboración y recopilación de los anexos	2/01/17 9:00	2/01/17 17:00
74	Analisis y revisón completa de la memoria realizando ajustes	3/01/17 17:00	4/01/17 9:00
75	Última revisión y entrega final	4/01/17 9:00	4/01/17 17:00



10. Bibliografía

GART01: Gartner, Cloud Computing as Gartner Sees It, 2009

WIKI1: Múltiples, Computación_en_la_nube, 2016
https://es.wikipedia.org/wiki/Computaci%C3%B3n_en_la_nube

TICBEAT1: Eduardo Martín, Qué es 'cloud computing, 2014
http://www.ticbeat.com/cloud/que-es-cloud-computing-definicion-concepto-para-neofitos/

WIKI2: Múltiples Autores, Virtualización, 2016 https://es.wikipedia.org/wiki/Virtualizaci%C3%B3n

BLOG1: Gerard Vivancos, docker-y-la-era-de-los-contenedores, 2015
http://blog.celingest.com/2015/02/17/docker-y-la-era-de-los-contenedores/

wiki3: Múltiples Autores, Docker, 2016 https://es.wikipedia.org/wiki/Docker_(software)

wiki5: Múltiples Autores, Chroot, 2016 https://es.wikipedia.org/wiki/Chroot

wiki6: Múltiples Autores, ,

wiki7: Múltiples Autores, , 2016 https://en.wikipedia.org/wiki/Linux_namespaces

wiki8: Múltiples Autores, Cgroups, 2016 https://en.wikipedia.org/wiki/Cgroups

kernel1: Múltiples Autores, cGroups, 2016
https://www.kernel.org/doc/Documentation/cgroup-v1/

wiki9: Múltiples Autores, SELinux, 2016 https://es.wikipedia.org/wiki/SELinux

orquestacion1: TOOIT, Orquestacion de servicios, 2015,
http://tooit.com/es/automatizacion-y-orquestacion-de-servicios-it

orquestacion2: tuataratech, Que es la Orquestacion, 2015,
http://www.tuataratech.com/2016/03/que-es-la-orquestacion-la-proxima.html

prt1: Sobrebits, Particionado Linux, 2015, http://sobrebits.com/buenas-practicas-en-el-particionado-de-gnulinux-parte-2-avanzado/

libro1: Shrikrishna Holla, Orchestrating Docker, 2015

libro2: Sébastien Goasguen, Docker Cookbook, 2016

libro3: Deepak Vohra, Kubernetes Microservices with Docker, 2016

libro4: Oskar Hane, Build your own PaaS with Docker, 2015

libro5: Karl Matthias, Docker Up and Running, 2015

url5: Digital Ocean, Docker Data Volumes, ,
https://www.digitalocean.com/community/tutorials/how-to-work-with-docker-data-volumes-on-ubuntu-14-04

url6: Docker, Manage data in containers, 2016,
https://docs.docker.com/engine/tutorials/dockervolumes/

url7: Docker, Docker container networking, 2016,
https://docs.docker.com/engine/userguide/networking/

docker8: Docker, Dockerfile Reference, 2016, https://docs.docker.com/engine/reference/builder/

docker1: Docker, Docker Hub, 2016, https://docs.docker.com/docker-hub/

docker2: Docker, Docker Machine, 2016, https://docs.docker.com/machine/

docker3: Docker, Docker Registry, 2016, https://docs.docker.com/registry/

docker5: Docker, Docker UCP, 2016, https://docs.docker.com/datacenter/ucp/1.1/

docker6: Docker, Docker Compose, 2016, https://docs.docker.com/compose/

docker7: Docker, Docker Swarm, 2016, https://docs.docker.com/swarm/

url8: , IP Virtual Server, , https://en.wikipedia.org/wiki/IP_Virtual_Server

consul: HashiCorp, Service Discovery with Consul, , https://www.consul.io/

etcd1: CoreOS, Etcd and Service Discovery, 2016, https://coreos.com/etcd/

zoo: Apache.org, Zookeeper, 2015, https://zookeeper.apache.org/

raft: , Raft Consensus, 2015, https://raft.github.io/

dockernt: Docker, Docker Networking, 2016, https://docs.docker.com/engine/userguide/networking/

dockerservice: Docker, Docker Service Comands, 2016, https://docs.docker.com/engine/reference/commandline/service_create/

dnsrr: Wikepedia, Round RobinDNS, 2012, https://es.wikipedia.org/wiki/Dns_round_robin

ipvs: Linux Virtual Server, Linux Virtual Server, , http://www.linuxvirtualserver.org/software/ipvs.html

wordpress1: Wordpress, Wordpress, 2016, https://es.wordpress.com

wordpress2: Wikipedia, Wordpress, 2016, https://es.wikipedia.org/wiki/WordPress

Vitae: Vitae Consultores, Guia Salarial TI, 2016 http://www.vitaedigital.com/procesos-de-seleccion/guia-salarial-sector-ti-galicia-2014-2015

url3: Dell, Dell PowerEdge R430 Spec Sheet, 2016 http://i.dell.com/sites/doccontent/shared-content/data-sheets/en/Documents/Dell-PowerEdge-R430-Spec-Sheet.pdf

url2: Dell, Dell Networking serie X - hoja de especificaciones, 2016

http://i.dell.com/sites/doccontent/business/smb/merchandizing/es/Documents/Dell_Networking_X_Series_spec_sheet_ES.pdf

url1: Canonical, Guia de instalación ubuntu 16.04, , https://help.ubuntu.com/16.04/installation-guide/

url4: Canonical, UbuntuBonding, 2016, https://help.ubuntu.com/community/UbuntuBonding

nginx1: , Nginx, , https://github.com/jwilder/nginx-proxy/blob/master/Dockerfile

nginx2: NGINX, NGINX, 2016, https://www.nginx.com/resources/wiki/

nginx3: NGINX, NGINX with Docker, 2016, https://hub.docker.com/_/nginx/