Universidad de Alcalá
Escuela Politécnica
Superior

Grado en Ingeniería en
Tecnologías de la
Telecomunicación

Trabajo Fin de Grado

Realización de
escenarios para ilustrar
el funcionamiento de
Docker

Autor: Samuel Morillo
Hernangómez

Tutor/es: José Manuel
Arco Rodríguez

2022

Universidad de Alcalá

ESCUELA POLITÉCNICA SUPERIOR

Grado en Ingeniería en Tecnologías de Telecomunicación

Trabajo Fin de Grado

Realización de escenarios para ilustrar el funcionamiento de Docker

Autor: Samuel Morillo Hernangómez

Tutor: José Manuel Arco Rodríguez

Tribunal:

Presidente: Jaime José García Reinoso

Vocal 1º: Antonio García Herráiz

Vocal 2º: José Manuel Arco Rodríguez

Fecha de depósito: 15-09-2022

Resumen

El trabajo desarrollado consiste en la realización de escenarios prácticos para ilustrar el funcionamiento de la plataforma Open Source, Docker, que sirve para desarrollar y ejecutar aplicaciones. El principal objetivo es mostrar el funcionamiento de Docker y su despliegue en clusters, llamado Docker Swarm, mediante tutoriales prácticos a personas en el en progreso de la enseñanza de ingenierías relacionadas con las telecomunicaciones.

Docker es una tecnología de creación de contenedores que permite la creación y el uso de contenedores bajo un determinado sistema operativo.

Palabras clave: Docker, nodo, manager y worker.

Abstract

The work developed consists of the realization of practical scenarios to illustrate the operation of the Open Source Docker platform that is used to develop and execute applications. The main objective is to show the operation of Docker and its deployment in clusters, called Docker Swarm, through practical tutorials to people in the process of teaching engineering related to telecommunications.

Docker is a containerization technology that allows the creation and use of containers under a certain operating system

Key words: Docker, nodo, manager y worker.

Índice de contenido

Índice de figuras

Índice de códigos

1. Introducción

El trabajo se centra en la realización de escenarios para ilustrar el funcionamiento de Docker, esto puede ser un gran aporte para la comunidad debido al auge de la tecnología Docker. Se centra en una demostración práctica y guiada, lo cual sirve como herramienta para conocer los conceptos y características del software Docker.

Docker es una tecnología Open Source, esto hace que las colaboraciones de desarrolladores lo hagan estar en constante mejora, además es muy atractivo debido a que su uso mediante la suscripción más simple no conlleva gastos.

En cuanto a una definición que se nos ofrece Docker es una tecnología de creación de contenedores que permite la creación y el uso de contenedores bajo el sistema operativo Linux principalmente [1]. Los contenedores son parecidos a una VM (Virtual Machine) en el sentido de que son aplicaciones virtualizadas que se ejecutan sobre un programa especial, llamado en general “engine”. La diferencia fundamental es que no hay que virtualizar todo el sistema operativo si no solo lo necesario para ejecutar la aplicación, esta característica lo hace extremadamente más ligero [2]. Es más sencilla su ejecución reduciendo el coste computacional necesario para implementarlo, de nuevo esto es una característica que atrae a muchos usuarios a hacer uso de Docker. Otra ventaja importante es que las aplicaciones ejecutadas en contenedores Docker disponen de un entorno tan aislado como se desee.

Durante la redacción de este trabajo se muestra el funcionamiento mediante la realización de escenarios de uso de los contenedores Docker. Se comienza con muestras de su funcionamiento básicas, no se requiere conocimientos previos sobre esta tecnología.

En los inicios de la redacción de este trabajo de fin de grado se exponen conceptos básicos de Docker, tras ello se configura un formato de redacción en modo tutoriales que sirven como guía para ir conociendo el uso práctico. Cada tutorial expone características de Docker y demuestran la teoría expuesta de manera práctica y muy visual.

1.1 Objetivos

El objetivo del trabajo es la realización de escenarios para ilustrar el funcionamiento de Docker, esto se desarrolla en un formato que pueda servir como tutorial para personas que estén interesadas en conocer el funcionamiento de Docker sin conocimientos previos; por ello irá acompañado de explicaciones del desarrollo de cada escenario de la manera más explícita posible. Dividiéndose por secciones cada una de ellas aporta un elemento o característica nueva de manera sencilla. Los ítems que se han abordado durante el desarrollo son:

- Descarga de las imágenes públicas y oficiales de repositorio público Docker Hub.
- Sistema de ficheros en contenedores Docker.
- Persistencia de los datos entre distintas sesiones de los contenedores.
- Comunicaciones en las redes internas de Docker.
- Implementar un gestor o manager en varios nodos.

- Control de los comandos Docker en una terminal Linux.
- Ejecución de contenedores en varias plataformas/infraestructuras.
- Control de versiones de las aplicaciones desarrolladas para ser ejecutadas en contenedores Docker.
- Servicio de directorio para localizar contenedores y servicios en un clúster.
- Composición de microservicios.
- Presentación de ejemplos reales de uso de Docker.
- Balanceo de carga en nodos Swarm y disponibilidad ante fallos de los contenedores.
- Comprobar comportamiento ante la pérdida de datos con ejecución en varios nodos.

1.2 Estructura de la memoria

En esta sección se puede encontrar de manera resumida como está organizada la memoria del trabajo:

- **Introducción**: En esta primera parte se describe una introducción del proyecto desarrollado y se describe la jerarquía de los principales puntos desarrollados.

- **Estado del arte**: Se detallan las características de Docker de manera teórica y se compara con su antecesor LXC. Además, se describe Docker Swarm enfrentando una comparativa de este con Kubernetes debido a su similitud. Se explican las soluciones que aporta el uso de Docker y Docker Swarm, comentando las limitaciones que ambos poseen.

- **Tutoriales**: Se describen paso a paso las ejecuciones que se deben realizar. Esto sirve para demostrar la teoría propuesta sobre Docker de manera práctica, además de aprender su uso real.

    - Tutorial 1: **Instalación de Docker Engine y creación de usuario en Docker Hub.** En este primer tutorial se redacta como se debe instalar Docker Engine y como crear un usuario gratuito en Docker Hub para permitirnos adquirir, compartir y almacenar imágenes de Docker.

    - Tutorial 2: **Primeros pasos con Docker.** Utilizando un Dockerfile sencillo se explican los distintos comandos que lo componen y como compilar y ejecutar la imagen resultante. Esta imagen obtenida de Docker se almacena en el Docker Hub asociado al usuario del tutorial anterior.

    - Tutorial 3: **Servicios Docker definidos en docker-comopose.yml.** En este apartado se comienza a utilizar un docker-compose.yml basado en la imagen alojada en Docker Hub en del tutorial dos. Se explica los distintos elementos utilizados para definir el servicio y además se despliegan con Docker Swarm, acompañado de una muestra como este Docker Swarm realizado un escalado de réplicas.

- Tutorial 4: **Docker swarm para presentar los servicios.** Se hace uso de la herramienta docker-machine para alojar los distintos nodos del Docker Swarm en host virtuales distintos. Se inician y se unen al Swarm varios nodos.

- Tutorial 5: **Cooperación de varios servicios alojados en Docker Swarm.** Se despliegan varios nodos en Docker Swarm que contiene varios servicios, hay un servicio que necesita de los datos persistentes para cumplir su función y se demuestra la persistencia de los datos.

- Tutorial 6: **Uso de volúmenes de almacenamiento en Docker.** Muestra los distintos modos de almacenamiento en Docker, redacción de sus características teóricas, comparación y demostración de uso.

- Tutorial 7: **Comunicaciones en Docker.** Ilustración con distintas imágenes de Docker de como se comunican entre los distintos contenedores del Docker Swarm y el exterior de la red virtual creada para el enjambre.

- Tutorial 8: **Servicios en Docker Swarm con host en distintas redes.** Demostración del funcionamiento del Docker Swarm que aloja sus nodos en host que residen en redes físicamente distintas.

- Tutorial 9: **Ejemplo real de servicios globales en internet con uso Docker.** Se presenta un uso de Docker en el mundo real para alojar un servicio web en una plataforma de computación en la nube, en este caso se utiliza Google Cloud Platform.

- **Conclusiones**: En el conjunto de la memoria se descubren los aspectos básicos de Docker y Docker Swarm. Se hace un revisión teórica y comparativa con tecnologías similares en los primeros capítulos. Tras ello se ilustra mediante distintos escenarios y pruebas prácticas las características desarrolladas de manera teórica, además, se observan casos de uso real de Docker y Docker Swarm.

- **Bibliografía**: Se detallan las fuentes de información consultadas.

2. Estado del arte

2.1 Orígenes de Docker y su antecesor LXC

Los orígenes de Docker residen en la empresa antes llamada dotCloud Inc, esta misma debe tanto su éxito actual a la tecnología Docker que el nombre en este momento de la compañía es Docker Inc. El software de Docker nace como un proyecto interno en dotCloud que se dedicaba principalmente a ofrecer un servicio PaaS (Platform-as-a-Service).
Con la intención de mejorar sus servicios los ingenieros de la compañía trabajaban en el lanzamiento de tareas desplegadas sobre el kernel de Linux que incluyesen funcionalidades de LXC (LinuX Containers) y cgroups [4].

Los cgroups de GNU/Linux o grupos de control, son una característica del kernel Linux que permite que los procesos se organicen en grupos jerárquicos. El objetivo de agrupar los procesos de esta manera es limitar y monitorear el uso de varios tipos de recursos. Con la ayuda de los cgroups cada proceso corre en su propio espacio del kernel y de la memoria. Cuando se tiene la necesidad, se puede configurar cgroups para limitar los recursos de los puede disponer para utilizar un proceso [4].

LXC es previo a Docker, pero comparten varias similitudes. LXC es una plataforma de contenedores Open Source que proporciona un conjunto de herramientas, plantillas, bibliotecas y enlaces entre lenguajes. Ofrece un entorno de virtualización que puede instalarse en varios sistemas basados en Linux [5]. En ambos casos, Docker y LXC requiere la instalación de este software Open Source para hacer uso de ellos.

Como se comentaba LXC es el antecesor de Docker, pero estos ahora encuentran diferencias entre sí como las siguientes:

- La herramienta Docker no limita su instalación únicamente a los sistemas operativos basados en Linux, ya que puede ser instalado en otros sistemas operativos como MacOS (con una máquina virtual Linux) y Windows. Además, los contenedores Docker disponen de imágenes basadas en Linux y otras basadas en Windows, estas ofrecidas por Microsoft.

- LXC es un contenedor de sistema operativo, mientras que Docker es un contenedor de aplicaciones; esto se refleja en que cada contendor LXC suele tener varios servicios ejecutándose y en cada contenedor de Docker se suele aislar un único servicio, esto es lo habitualmente recomendado, aunque no es un imperativo de sus características. De este modo, un contenedor LXC se comporta como un sistema operativo funcional, incluyendo de este modo como mínimo todas las funcionalidades básicas de un sistema operativo Linux. Por el contrario, Docker solo virtualiza lo necesario haciendo que este sea extremadamente ligera su ejecución y tamaño de las imágenes. Esta característica mencionada hace que Docker se más ligero en ejecución y tamaño de sus imágenes que su antecesor LXC.

- Una ventaja de Docker es que, debido a una gran comunidad que se le permite alojar sus imágenes de Docker en un repositorio de manera gratuita y a las grandes

distribuidoras publicar imágenes oficiales, es muy sencillo obtener imágenes configuradas para distintas aplicaciones o usos. Las imágenes LXC también pueden compartirse de igual manera en un repositorio, pero su uso no esta tan extendido como con Docker Hub, el cuál es gratuito y de uso muy sencillo para almacenar, compartir u obtener imágenes.

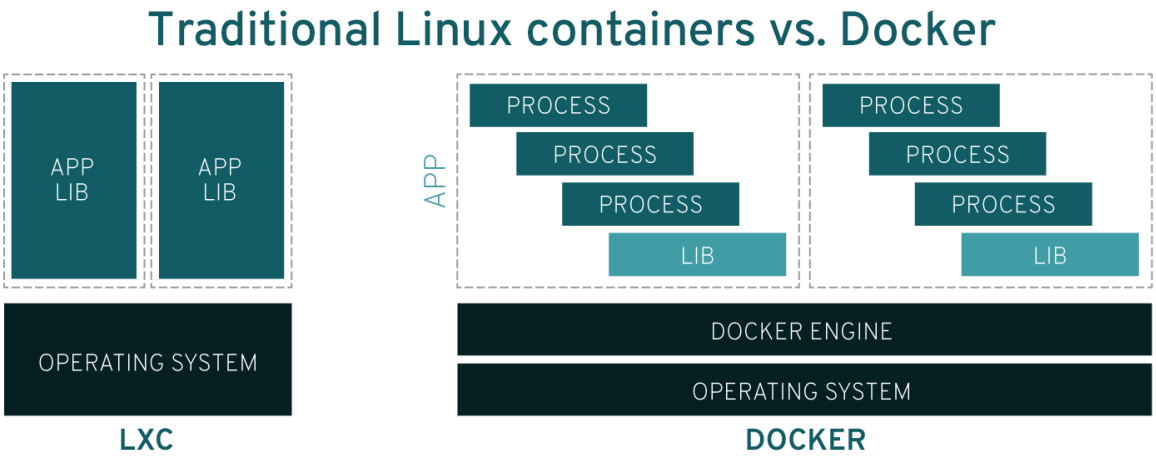

Figura 1: Diferencias entre Docker y LXC [1]

Continuando con la historia de los orígenes de Docker, fue en 2013 cuando la empresa dotCloud presentó de manera pública la herramienta Docker y ante la buena acogida de la comunidad tecnológica se definió como un proyecto Open Source. Esto significa que cualquiera podría de manera gratuita descargar, ejecutar y modificar el código de Docker. Al igual que en otros proyectos Open Source, que se expusiese el código y se ofreciera la posibilidad de mejorarlo hizo que Docker obtuviera una gran fama y ventajas.

En cuanto al uso de Docker la empresa Apium Hub [6] nos ofrece los siguientes datos:

- 2/3 de las empresas que prueban Docker, lo adoptan. La mayoría de las compañías que adoptaron Docker ya lo hicieron dentro de los 30 días posteriores al uso inicial de la producción, y casi todos los restantes se convierten dentro de los 60 días.
- La adopción de Docker real aumentó un 30% en un año.
- PHP, Ruby, Java y Node son los principales frameworks de programación utilizados en contenedores.

2.2 ¿Qué es Docker?

Docker es un proyecto Open Source que desarrolla la tecnología software de creación de contenedores que permite la creación y el uso de contenedores [1].

La tecnología Docker usa el kernel de Linux y las funciones que este ofrece, como pueden ser los cgroups y namespaces, para segregar los procesos.

Los namespaces de GNU/Linux permiten encapsular recursos globales del sistema de forma aislada, evitando que puedan interferir con procesos que estén fuera del namespace. Los

cambios en el recurso global son visibles para otros procesos que son miembros del namespace, pero son invisibles para otros procesos fuera del mismo namespace [7]. Ejemplo de uso de los namespace es la configuración de los network namespace que nos permiten aislar la configuración de red, IP, firewall o más elementos similares entre distintos namespaces.

Es la combinación del uso de cgropus y namespaces lo que enmarca la definición de un contenedor en concreto, produciendo que cada uno de los contenedores se puedan ejecutar de manera independiente. Uno de los propósitos de los contenedores Docker es lograr esta independencia. Esto aporta la capacidad de ejecutar varios procesos y aplicaciones por separado mejorando el uso de su infraestructura y, al mismo tiempo, conservando la seguridad como si se tratase de sistemas aislados, como se observa en la siguiente imagen:

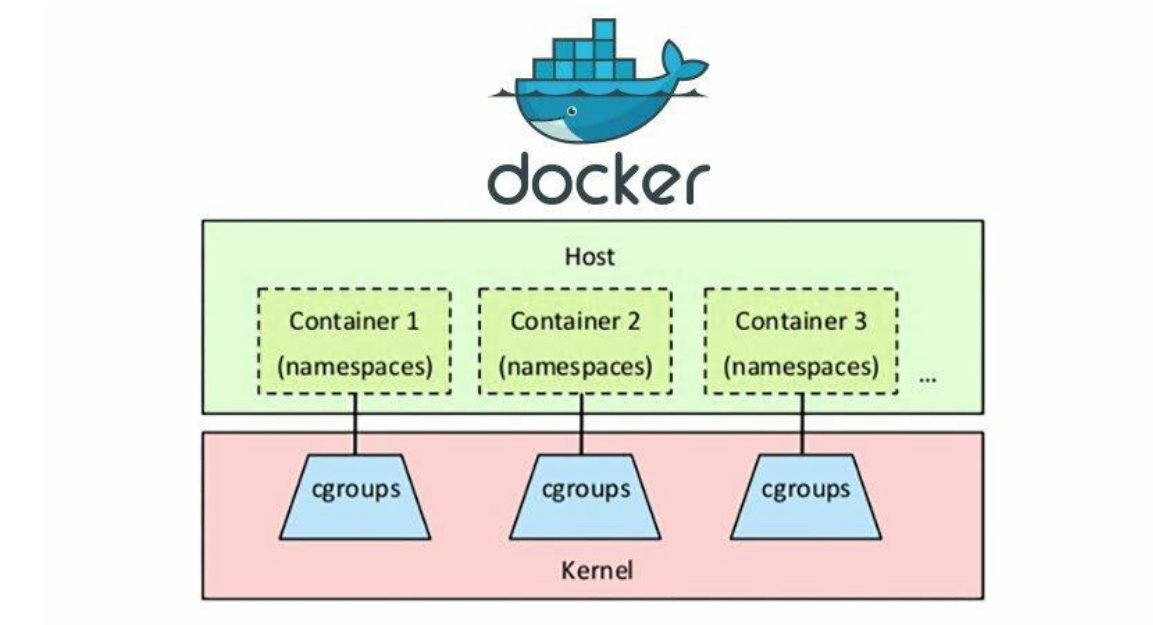

Figura 2: Estructura simbólica de la herramienta Docker [8]

Las herramientas del contenedor basado en kernel Linux, como Docker, ofrecen un modelo de implementación basado en imágenes. Esto permite compartir una aplicación, o un conjunto de servicios, con todas sus dependencias en varios entornos. Docker también automatiza la implementación de la aplicación (o conjuntos combinados de procesos que constituyen una aplicación) en este entorno de contenedores. Es decir, otra de las ventajas que se suma a la independencia y aislamiento es la gran portabilidad de su uso mediante la posibilidad de compartir imágenes con control de versiones. La distribución de las imágenes de estos contenedores es similar a la de las imágenes de las VMs (Virtual Machine); la diferencia fundamental es que no hay que virtualizar todo el sistema operativo si no solo lo necesario para ejecutar la aplicación, esta característica lo hace extremadamente más ligero [2].

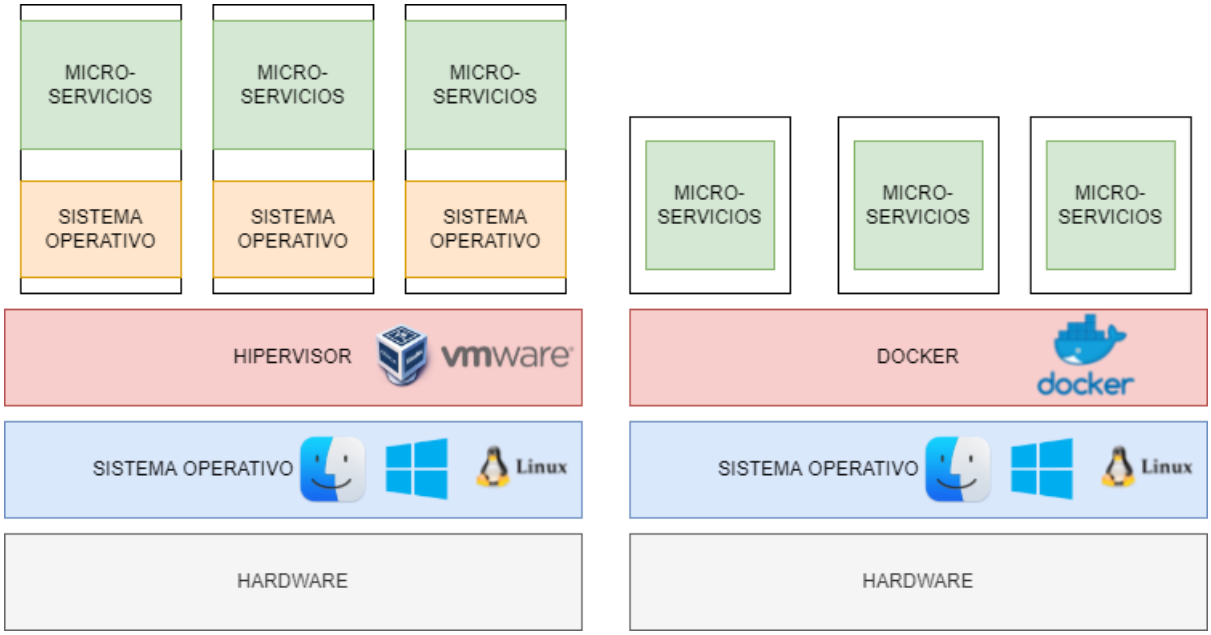

Figura 3: Estructura de una máquina virtual frente a un contenedor Docker

La ejecución de un contenedor Docker está asociado a una máquina, ya sea virtual o física. Esta máquina puede ser que sea accedida por distintos usuarios, por ello se debe limitar los permisos de la ejecución de Docker a únicamente los usuarios necesarios para evitar creaciones o destrucciones de los contenedores, volúmenes o más elementos de Docker que no sean correctas.

Además, se debe poner especial atención en proteger los procesos de Docker, ya que, si un usuario lo finaliza, desencadena la destrucción de todos los contenedores alojados en este host e incluso si se estos están cooperando con otros contenedores en distinto host corrompiendo el funcionamiento.

2.2.1 Docker Windows, alternativa a Docker convencional

Desde Microsoft se ha trabajado en una alternativa para Docker en la cual el sistema operativo que lo controla no sea Linux sino Windows o Windows Server. Para ello, se ofrecen de manera oficial cuatro imágenes de Docker:

- Windows Server Core
- Nano Server
- Windows
- Windows Server

Estas imágenes pueden ser obtenidas desde Docker Hub o desde el repositorio oficial de Microsoft el cuál es alojado por ellos mismos en Azure.

Como se dispone de un catálogo de opciones tan reducido, es sencillo crear una guía de recomendación de que imagen escoger, las siguientes descripciones son ofrecidas por Microsoft:

- Nano Server es una oferta de Windows ultraligera para el desarrollo de nuevas aplicaciones.

- Server Core tiene un tamaño medio y es una buena opción para migrar aplicaciones de Windows Server mediante "lift-and-shift".

- Windows es la imagen más grande y tiene compatibilidad completa con las API de Windows para cargas de trabajo.

- Windows Server es ligeramente más pequeña que la imagen de Windows, tiene compatibilidad completa con las API de Windows y le permite usar más características de servidor.

Además de estas descripciones Microsoft ofrece preguntas globales que pueden plantearse los usuarios sobre que imagen escoger y así poder elegir la imagen de Docker Windows que más se ajusta a su objetivo.

El uso de estas imágenes no es muy extendido y suele utilizarse para casos específicos. En caso de ser necesario la ejecución de estas imágenes se pueden considerar seguras y robustas, debido a que tienen el soporte de las imágenes oficiales de la comunidad de Microsoft.

2.3 ¿Qué es Docker Swarm?

Docker Swarm es una herramienta software que permite ejecutar los contenedores en una granja de nodos, esto implica uno o varios balanceadores de carga implementados en uno o varios nodos maestros y los nodos que prestan el servicio, implementados en nodos trabajadores. Los contenedores que se ejecutan en modo Swarm se les denomina en ocasiones como modo enjambre.

La referencia a que pueda existir uno varios nodos maestros es debido a que debe existir al menos un nodo maestro, pero podría haber más de uno sin que esto sea un conflicto.

Ejecutar Docker en modo Swarm permite gestionar varios clusters de manera descentralizada. Esto es una orquestación de contenedores los cuales no tienen por qué estar situados en la misma máquina física ni virtual. Crea así una independencia del sistema que aloja los contenedores, aumentando la seguridad ante pérdida del servicio si se distribuye los nodos en distintas máquinas.

Docker Swarm se basa en una arquitectura maestro-esclavo o también referido en inglés manager-worker. Cada enjambre está formado al menos por un nodo maestro (también llamado administrador o manager) y tantos nodos esclavos (llamados trabajadores o workers) como se desee. El maestro de Swarm es responsable de la gestión del clúster y la delegación de tareas, el esclavo se encarga de ejecutar dichas tareas.

Existen tres conceptos que se deben diferenciar: servicio, aplicación y tarea. Los servicios definidos para Docker Swarm pueden contener varias tareas o una única. Las tareas hacen referencia a las réplicas de un mismo servicio y puede existir varias tareas de un mismo servicio. El encargado de esta distribución de tareas entre los nodos como ya se ha mencionado es el manager del enjambre.
Las aplicaciones hacen referencia a los distintos softwares que puede contener un mismo contenedor que ejecuta una imagen especifica de Docker.

Una imagen que puede ilustrar las diferencias entre estos conceptos es la siguiente:

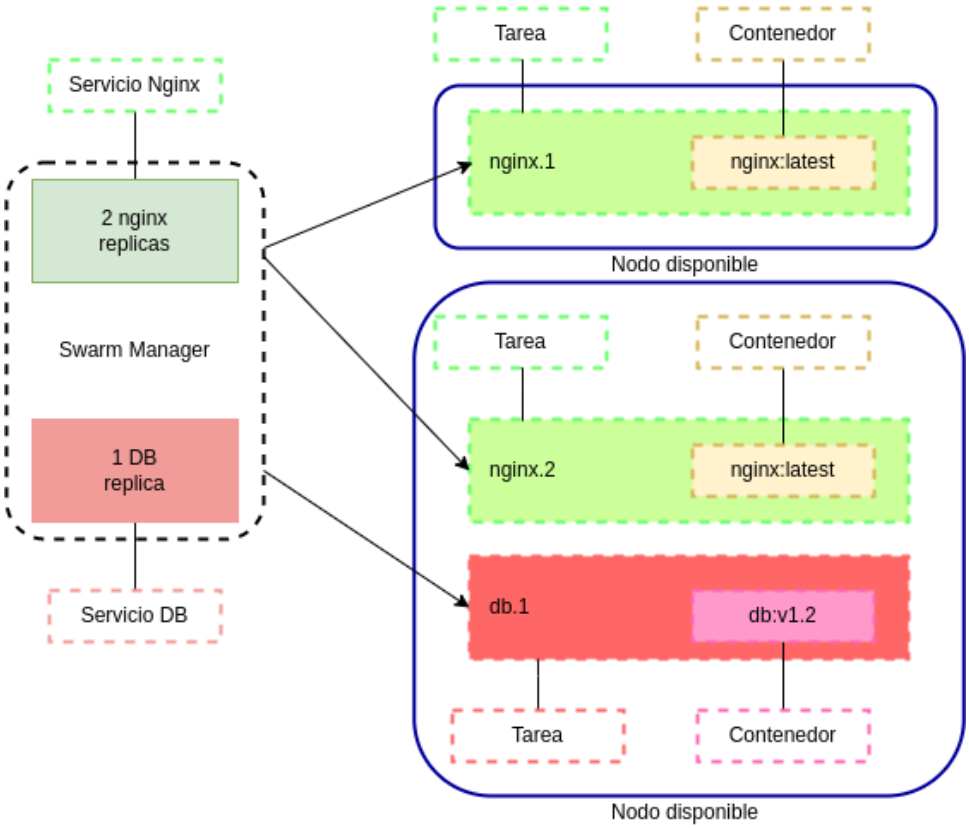

Figura 4: Diferenciar servicio y tarea

Se distinguen dos maneras de trabajar con Docker Swarm en cuanto a cómo es la disponibilidad de las réplicas de los servicios definidos:

- Servicios globales: Swarm ejecutará una tarea en todos y cada uno de los nodos del clúster que cumpla con las restricciones establecidas como la ubicación del servicio o recursos.

- Servicios replicados: Se especifica el número de tareas idénticas que se desea ejecutar. Swarm creará tantas réplicas de la tarea especificada como se ha determinado. Por ejemplo, si creamos un servicio con dos réplicas, Swarm creará dos tareas que alojará en los nodos del enjambre [3].

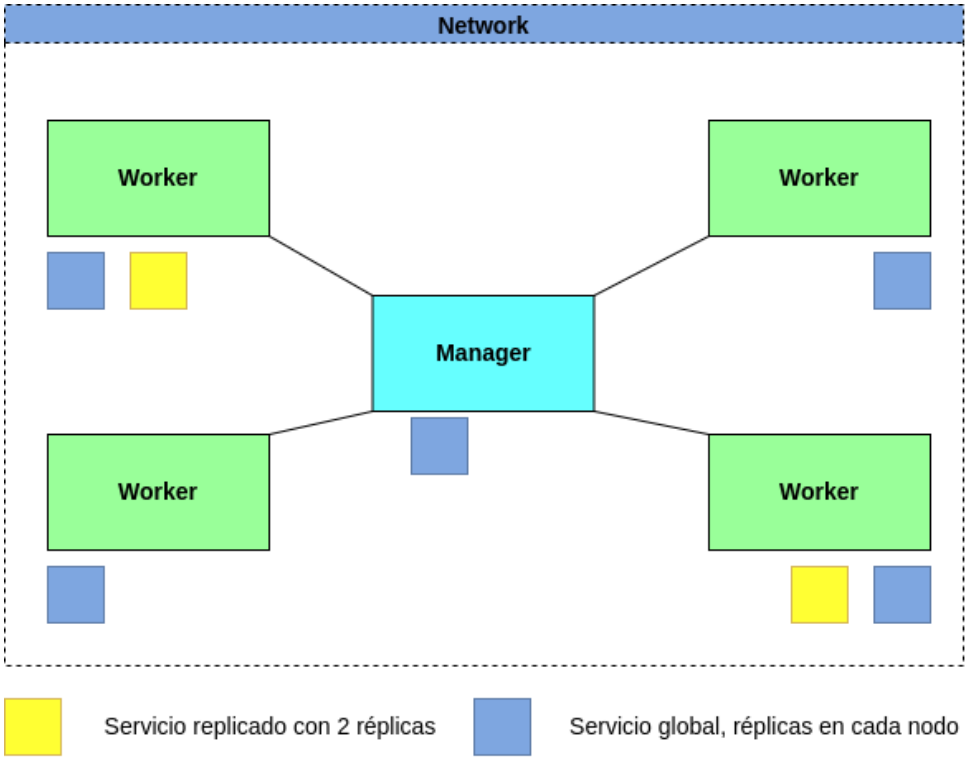

Figura 5: Ilustración sobre servicios replicados y servicios globales

La imagen anterior muestra como existe un servicio replicado con dos réplicas, alojado en dos nodos distintos. Existe también un servicio global, de este segundo se encuentra una réplica en cada nodo disponible.

La manera de trabajar con un servicio global o un servicio replicado se establece mediante el número de replicas a ejecutar de un servicio o indicando que se desea ejecutar el servicio en el modo global. Se puede indicar mediante las sentencias del Docker CLI o mediante el docker-compose.yml

Para el ejemplo de servicios replicados ambos ejemplos serían los siguientes:

- A través de la ejecución iniciada con un comando similar al siguiente:

```
docker service create \
  --name my_website \
  --replicas 3 \
  nginx
```

- Mediante docker-compose.yml:

```
version: "3"
services:
  my_website:
    image: nginx:1.20.2
    deploy:
      replicas: 3
      restart_policy:
        condition: on-failure
      resources:
        limits:
          cpus: "1"
          memory: 128M
    ports:
      - "80:80"
    networks:
      - webnet
networks:
  webnet:
```

Código 1: Archivo docker-compose.yml ejemplo de servicios replicados.

Se observa que en ambos ejemplos las réplicas establecidas a generar son tres del servicio my_website.

Por otro lado, siguiendo este mismo ejemplo, pero en el caso de servicios globales sería de la siguiente manera:

- Inicio del servicio mediante un comando parecido al siguiente:

```
docker service create \
  --name my_website \
  --mode global \
  nginx
```

- Mediante docker-compose.yaml

```
version: "3"
services:
  my_website:
    image: nginx:1.20.2
    deploy:
      mode: global
      restart_policy:
        condition: on-failure
      resources:
        limits:
          cpus: "1"
          memory: 128M
    ports:
      - "80:80"
    networks:
      - webnet
networks:
  webnet:
```

Código 2: Archivo docker-compose.yml ejemplo de servicios globales

En este ejemplo el servicio my_website se está ejecutando en modo global, es decir una réplica por cada nodo disponible.

Los servicios pueden ser accedidos por cualquier nodo del swarm, que como se verá más adelante, usa una red ovelay interna (Ingress network) que conecta todos sus nodos y la petición llega al nodo donde se esté ejecutando el servicio.

Cada servicio se registra automáticamente en un DNS interno del Swarm. Si un servicio está replicado en varios nodos el nodo gestor balancea las peticiones usando el DNS. Por ejemplo, si hay tres réplicas de my_website, el DNS podría tener registrado lo siguiente
my_website 172.17.0.5
my_website 172.17.0.6
my_website 172.17.0.7
y cada vez que se pide el servicio el DNS responde usando las tres direcciones de forma circular.

Si hay varios nodos gestores funcionan en modo Activo-Pasivo, uno es el “leader” y los demás están de respaldo.

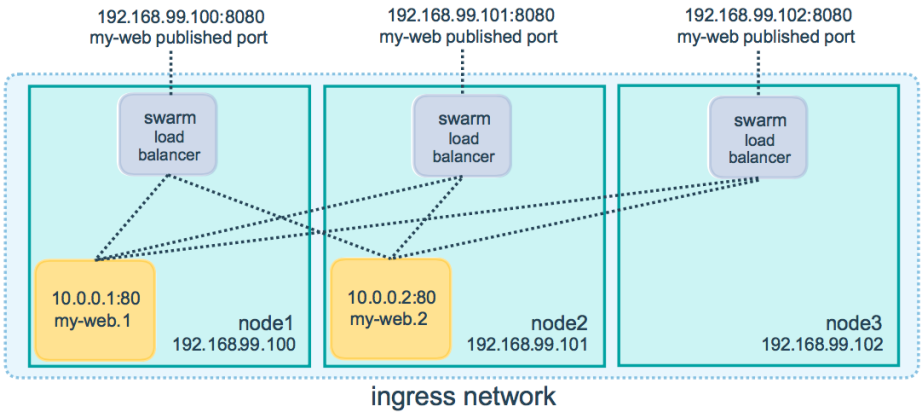

Figura 6: Gestión interna de los servicios Docker Swarm

2.3.1 Características de Docker Swarm

Docker Swarm se caracteriza por las siguientes ventajas principalmente:

- Se encuentra integrado con la API Docker Engine.
- Se organiza de manera jerárquica muy simple, existen únicamente dos roles. El rol de worker y el rol de manager. Los managers pueden actuar también como workers.
- Redistribución de las cargas de trabajo si algún nodo falla asegurando una alta disponibilidad.

- Administración de los grupos de contenedores, ofreciendo la posibilidad de agregar, eliminar, balancear la carga entre ellos, etc.

- Funcionalidades de escalado manual, permitiendo aumentar y disminuir recursos según necesidad, y rolling updates (gestión de actualizaciones sobre los servicios) integrado.

- No es necesaria ninguna instalación extra para tenerlo disponible con Docker.

2.4 Diferencias entre Docker Swarm y Kubernetes

Existen puntos en común que llevan a que esta comparación pueda ser de interés para comprender las diferencias y casos de uso recomendables de cada uno.

2.4.1 Breve definición y características de Kubernetes

Como definición breve de Kubernetes (K8s) es una plataforma de código libre cuyo objetivo es el de administrar y organizar clusters de contenedores, evitando los procesos manuales relacionados para escalar e implementar aplicaciones en contenedores. Cuando se trabaja con grupos de contenedores, Kubernetes facilita su administración de forma rápida y efectiva [9].

Las principales características de Kubernetes son:

- Gestión de las cargas de trabajo críticas, permite optimizar los recursos y aumentar el rendimiento.

- Añadir contenedores de manera automática sin que afecte a la disponibilidad.

- Eliminar, reiniciar o reemplazar los contenedores que fallen. Kubernetes con esta opción busca garantizar una alta disponibilidad.

- Facilita la orquestación de almacenamiento para crear sistemas de almacenamiento.

- Ejecución de despliegues automatizados monitorizando el proceso y permitiendo revertir los cambios (roll-backs automáticos).

- Escalado automático o manual de aplicaciones, incrementando los recursos necesarios o reduciéndose en caso de infrauso.

- Utiliza su propia API que está en constante mejora debido a la inmensa comunidad que hace uso de Kubernetes.

- Soporta una gran variedad de contenedores como Docker, RKT, cri-o o frakti.

2.4.2 Comparativa entre tecnologías

Construcciones de las aplicaciones

En Kubernetes, cada nivel de aplicación se define como un pod y se puede escalar cuando se administra mediante una implementación, que se especifica en archivos YAML. Como se ha especificado en las características de Kubernetes el escalado puede realizarse de manera manual o automática, en los entornos de servicios públicos en producción suele producirse un auto-escalado. Los pods se pueden usar para ejecutar pilas de aplicaciones integradas verticalmente, aplicaciones compartidas y coadministradas.

En cuanto a Docker Swarm los servicios se pueden escalar utilizando plantillas Docker Compose YAML. Los servicios pueden ser globales o replicados. Los globales se ejecutan en todos los nodos, y los replicados ejecutan réplicas de un mismo servicio distintos nodos. Las tareas se pueden ampliar o reducir, y desplegar en paralelo o en secuencia.

En la comparativa de la manera de escalar, Docker Swarm tiene una gran desventaja ya que no ofrece la opción de un escalado automático en función de las necesidades.

Alta disponibilidad de los servicios

Las implementaciones en Kubernetes permiten que los pods se distribuyan entre los nodos para proporcionar HA (High Availability), tolerando así los fallos en distintos puntos de la aplicación. Los servicios de balanceo de carga detectan pods no saludables, en estado erróneo, y los eliminan. Se admite alta disponibilidad de Kubernetes. Además, se pueden cargar múltiples nodos maestros y nodos de trabajo para solicitudes de kubectl y clientes.

A diferencia de los primeros, los servidores se pueden replicar entre los nodos de Swarm. Los administradores de Swarm son responsables de todo el clúster y administran los recursos de los nodos de trabajadores. Los gerentes utilizan el equilibrio de carga de entrada para exponer los servicios externamente.

Los administradores de Swarm usan el algoritmo de consenso de Raft para garantizar que tengan información de estado consistente. Se recomienda asimismo un número impar de gerentes, y la mayoría de los gerentes deben estar disponibles para un clúster Swarm en funcionamiento.

Balanceo de Carga

Los pods de Kubernetes se exponen a través de un servicio, que se puede usar como un equilibrador de carga dentro del clúster.

Docker Swarm gestiona el balanceo de carga con ayuda de la creación de un DNS interno que distribuye solicitudes entrantes a un nombre de servicio. Los servicios pueden ejecutarse en los puertos especificados por el usuario o pueden asignarse automáticamente en el rango de puertos 30000-32767.

Escalado automático de la aplicación

El escalado automático en Kubernetes se implementa de manera simple en función de los valores declarados para la redimensión y métricas de los pods. Los valores definidos como límites para el autoescalados pueden ser diversos, un ejemplo de los valores más utilizados en sistemas de producción son los de uso de CPU y/o uso de memoria.

Docker Swarm no ofrece la posibilidad de realizar el autoescalado, estas redimensiones deben realizarse de manera manual. Esta limitación supone una gran desventaja en la comparativa realizada.

Chequeos de salud

En Kubernetes se ofrecen distintas posibilidades para asegurar el correcto funcionamiento de los pods. Se permite realizar comprobaciones durante el inicio del pod para verificar cuando está completado su despliegue y también verificaciones tanto de estado, por si alguno fallase y fuese necesario reiniciarlo u otra operación, como de carga; en algunas ocasiones el pod puede estar sobrecargado y no permite aceptar más carga de la que ya está gestionando.

Los chequeos de salud en Docker Swarm están limitados a los servicios. Si un contenedor que aloja un servicio no responde durante su estado de ejecución, se reemplaza contenedor. Se puede implementar de manera sencilla la funcionalidad de verificación de estado en las imágenes Docker usando la instrucción HEALTHCHECK o en el archivo docker-compose.yml.

```yaml
version: '3.1'

services:
  web:
    image: docker-flask
    ports:
      - '8212:5000'
    healthcheck:
      test: curl --fail -s http://localhost:8212/ || exit 1
      interval: 1m30s
      timeout: 10s
      retries: 3
      start_period: 50s
```

Código 3: Archivo docker-compose.aml ejemplo de uso de healthcheck

Este ejemplo de código para el healthcheck está basado en un ejemplo que ofrece la documentación [10].

En este ejemplo se observa la implementación de un chequeo de seguridad del servicio configurado en el docker-compose.yml. Se basa en realizar una petición así mismo en el puerto expuesto de manera exterior, se realiza esta petición cada minuto y medio declarando un timeout de diez segundos. Si esta comprobación detecta error en tres ocasiones consecutivas considera que el contenedor se encuentra en un estado *unhealthy*.

En ocasiones es necesario generar un periodo en el inicio en el cual no se comiencen a realizar las comprobaciones para dar tiempo suficiente al despliegue, en este ejemplo este periodo de gracia es de cincuenta segundos.

Redes

Kubernetes crea una conexión entre iguales para conectar los pods y los agentes de nodo para una red eficiente entre clústeres. Esta conexión incluye políticas de red que regulan la comunicación entre pods y asignan direcciones IP distintas a cada uno de ellos. Para definir la subred, el modelo de red de Kubernetes requiere dos Classless Inter-Domain Routers (CIDRs):

- Una dirección para el direccionamiento interno del nodo
- Una dirección para exponer el servicio alojado en el pod

Docker Swarm crea dos tipos de redes para cada nodo que se une a un Swarm:

- Un tipo de red overlay de todos los servicios dentro de la red.
- El puente que conecta la red global con la red interna del docker Swarm

Se genera una red overlay que permite las conexiones peer-to-peer entre los servicios, estas comunicaciones son seguras y cifradas.
Con una red superpuesta de varias capas, se logra una distribución de igual a igual entre todos los hosts que permite comunicaciones seguras y cifradas. [11]

Descubrimiento de servicios

Los servicios se pueden encontrar en Kubernetes utilizando variables de entorno o DNS. Kubelet agrega un conjunto de variables de entorno cuando se ejecuta un pod. Además, admite variables {SVCNAME_SERVICE_HOST} Y {SVCNAME_SERVICE_PORT} simples, así como también variables compatibles de Docker. El servidor DNS está disponible como un complemento. Para cada servicio de Kubernetes, el servidor DNS crea un conjunto de registros DNS. Si DNS está habilitado en todo el clúster, los pods podrán usar nombres de servicio que se resolverán automáticamente

El nodo de Swarm Manager asigna a cada servicio un nombre DNS único y equilibrios de carga que ejecutan contenedores. Las solicitudes a los servicios se equilibran hacia los contenedores individualmente a través del servidor DNS integrados en el enjambre. Docker Swarm tiene las siguientes opciones de descubrimiento de los servicios que aloja:

- Por defecto, la manera en la que se asocia un servicio a una dirección interna a la que distribuir el tráfico es de manera automática y sin ser explicita por parte del desarrollador.
- Se puede usar un archivo estático o una lista de nodos como backend de descubrimiento. El archivo debe almacenarse en un host al que se pueda acceder desde Swarm Manager. También puede proporcionar una lista de nodos como opción cuando Docker Swarm ya se haya iniciado.

2.4.3 Conclusión

En conclusión, el uso de Kubernetes requiere mayor conocimiento y complejidad, pero el uso de esta tecnología es más potente que Docker Swarm. La popularidad de Kubernetes es mayor que la de Docker Swarm además de que sus funciones se ajustan más a la hora de elaborar un servicio y exponerlo de manera pública.

Docker Swarm también es una herramienta útil pero sus características hacen que sea más deseable únicamente cuando se desea montar de manera sencilla entornos de pruebas.

Un artículo web [12] nos ofrece la estadística de que a principios del año 2020 el 51,8% de las organizaciones tecnológicas analizadas había comenzado a usar Kubernetes en los últimos dos años, el 21,6% utiliza esta tecnología desde hace tres años o más. Por otra parte, las organizaciones que no han implementado ya su uso declaran dos opciones: el 15,1% planea adoptar su uso a lo largo del año y únicamente el 11,5% no contempla su uso.

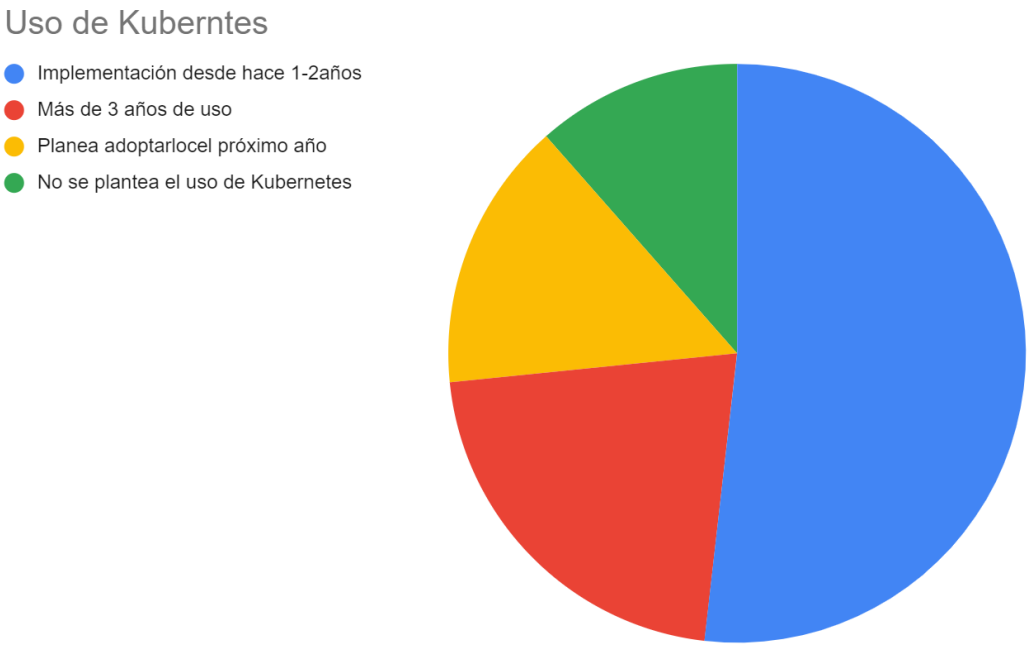

Figura 7: Proporciones de uso de la herramienta Docker en las empresas

3. Desarrollo de escenarios Docker

Esta parte del trabajo desarrolla distintos escenarios Docker con el fin de ilustrar su uso práctico y demostrar las características que esta tecnología ofrece. Está dividido en tutoriales para su fácil seguimiento, incluso sin conocimientos previos sobre Docker.

3.1 Tutorial 1: Instalación de Docker Engine y creación de usuario en Docker Hub

Se puede instalar Docker en los tres principales sistemas operativos de hoy en día MacOS, Windows y Linux. En entornos reales de producción se trata en la mayoría de los casos de sistemas operativos Linux y Windows, esto lidia perfectamente con el uso de Docker que permite su ejecución en ambos.

En los casos que Windows y MacOS ejecutan contenedores basados en Linux, se trata de este caso el más habitual, de manera transparente para el usuario actúa la herramienta Docker Machine que simula que el contenedor está ejecutándose es un sistema operativo Linux, esto no requiere ninguna acción por parte del usuario.

Docker Machine es una herramienta para aprovisionar y administrar hosts dockerizados (hosts con Docker Engine instalados en ellos). Docker Machine tiene su propio CLI "docker-machine" y el cliente Docker Engine. Se puede usar Docker Machine para instalar Docker Engine en uno o más sistemas virtuales. Estos sistemas virtuales pueden ser locales (como cuando usa Docker Engine en Mac o Windows, aunque también se puede usar docker-machine en Linux como se ve en otros tutoriales de este mismo trabajo) o en host remotos (como cuando usa Docker Machine para aprovisionar hosts dockerizados en proveedores cloud). Los propios hosts dockerizados se pueden considerar, y a veces se denominan, "máquinas" gestionadas [13].

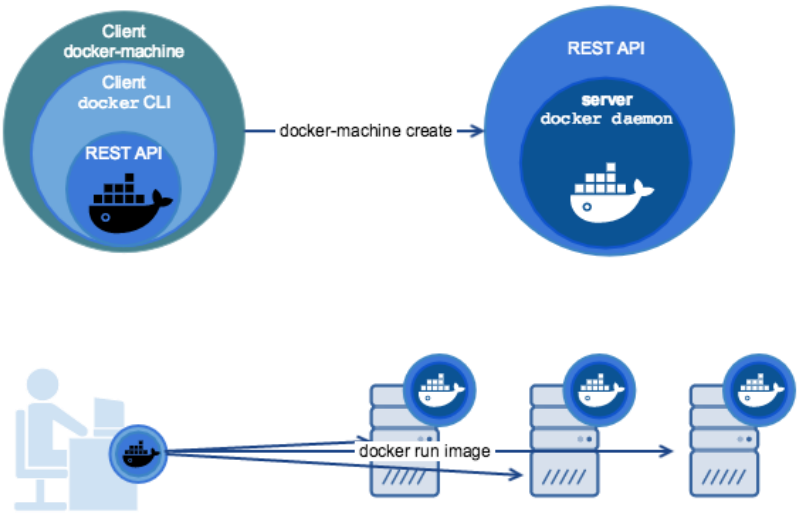

Figura 8: Ilustración de docker-machine [13]

Para el caso de contenedores basados en imágenes Windows, estas imágenes son ofrecidas por Microsoft y están disponibles en Docker Hub, solo pueden ser ejecutados en sistemas operativos Windows. Una manera de evadir esta restricción en los otros sistemas operativos es mediante la instalación de máquinas virtuales Windows.

En los casos que un sistema operativo Windows ejecuta contenedores basados en imágenes Windows de Microsoft existe la posibilidad de que este orquestado por el hipervisor 'Hyper-V' o no. La siguiente imagen muestra como sería la posibilidad de ejecutar un contenedor nativo Windows sobre un Kernel Windows de manera esquemática.

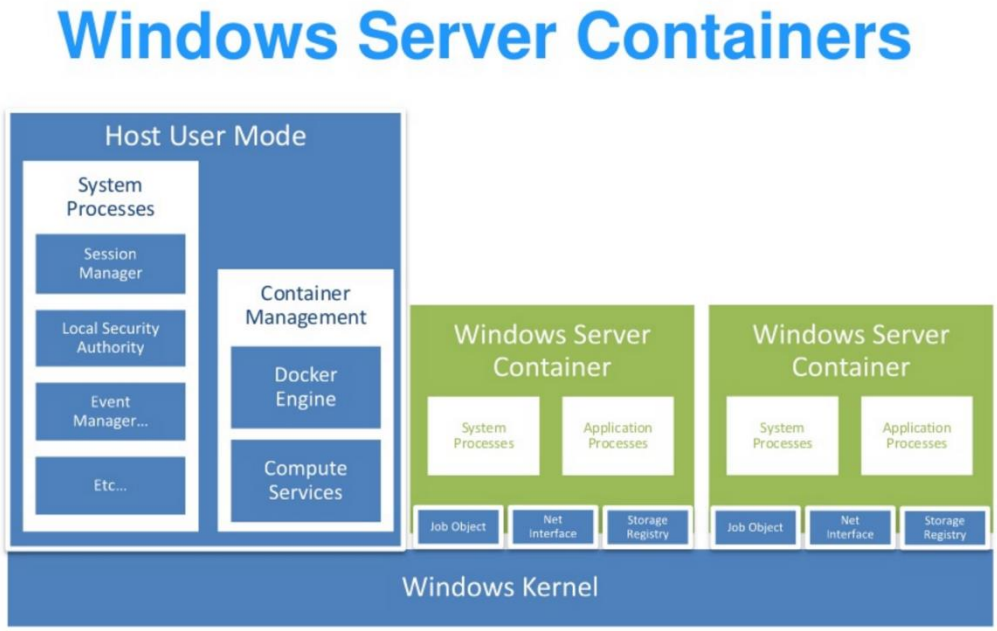

Figura 9: Contendores nativos Windows sobre Kernel Windows [14]

Por otro lado, una imagen con el mismo formato muestra como se ejecutaría en el caso de orquestar esta ejecución mediante Hyper-V, para solicitar esta opción se debe ejecutar Docker con el parámetro *–isolation=hyperv* :

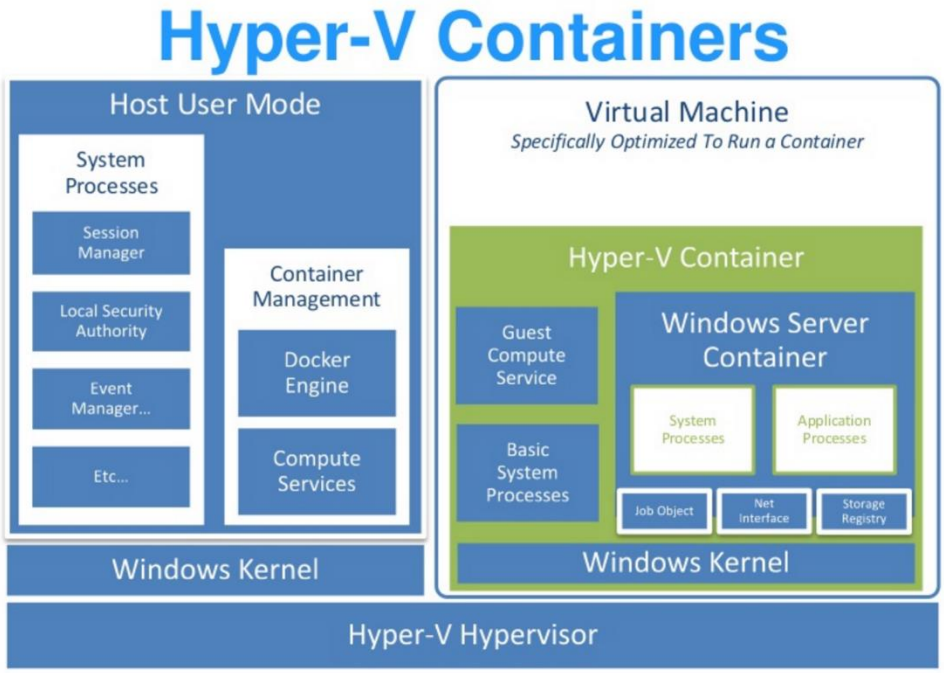

Figura 10: Contendores nativos Windows orquestados con Hyper-V sobre Kernel Windows [14]

Ambas maneras brindan el mismo comportamiento y experiencia al usuario, simplemente se diferencian en el modo de aislamiento. La primera opción, sin intervención de Hyper-V, utiliza el aislamiento de procesos, mientras que el contenedor de Hyper-V utiliza el aislamiento de máquinas virtuales. Aun que exista la intervención del hipervisor el inicio de estos contenedores es más ágil que el de una máquina virtual [15].

MacOS dispone de la opción de ejecutar tanto contenedores Docker basados en imágenes Linux como los basados en imágenes Windows, pero para este segundo caso debe crear una máquina virtual basada en Windows para evitar la limitación mencionada, no le sirve con Docker machine al usuario como se ha mencionado para el caso de la ejecución de contenedores Linux. A continuación, se muestra de manera visual cuando los contenedores son alojados en MacOS.

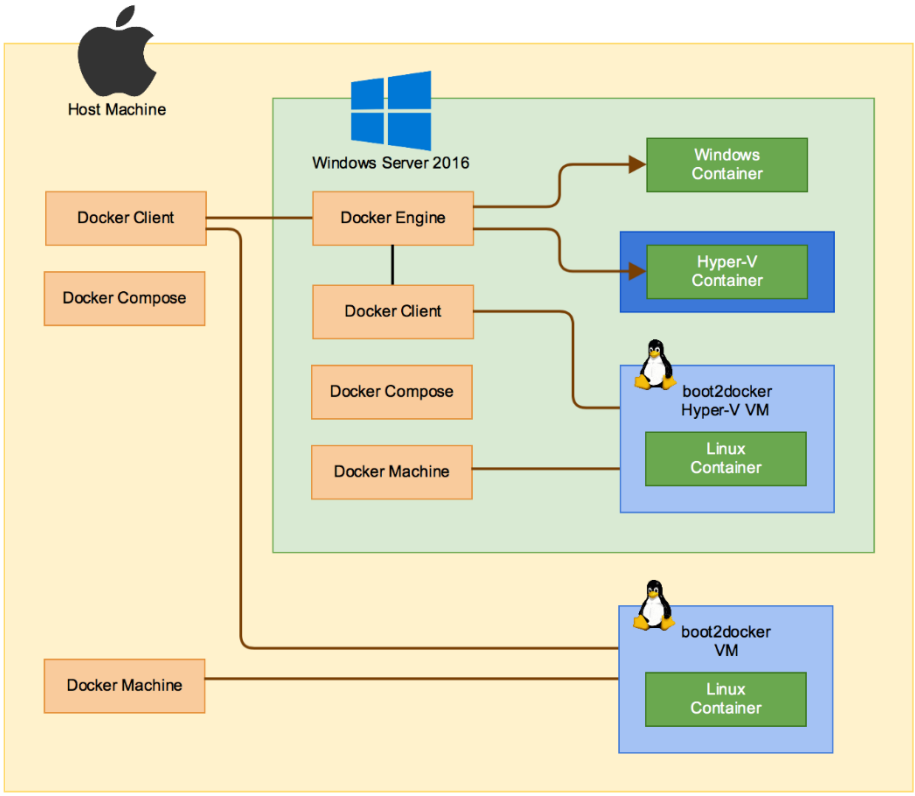

Figura 11: Esquema de la ejecución de contenedores Docker en MacOS [16]

En el desarrollo de este trabajo se ha optado por la implementación en Linux debido a que es el sistema operativo que mejor compatibilidad ofrece con Docker sin excepciones.

Docker ofrece una guía oficial de como instalar Docker Engine en los distintos sistemas operativos, en el caso de este trabajo se ha seguido la opción que ofrecen para sistemas operativos basados en Ubuntu [17].

Para verificar la compatibilidad del sistema operativo Linux, Docker Inc ofrece una ilustración que relata para que distribuciones Linux y arquitecturas se encuentran disponibles paquetes *.deb* y *.rpm* .

Platform	x86_64 / amd64	arm64 / aarch64	arm (32-bit)	s390x
CentOS	✓	✓		
Debian	✓	✓	✓	
Fedora	✓	✓		
Raspbian			✓	
RHEL				✓
SLES				✓
Ubuntu	✓	✓	✓	✓
Binaries	✓	✓	✓	

Figura 12: Compatibilidad de arquitecturas y sistemas operativos con Docker Engine [18]

Se puede instalar de distintas maneras, es recomendable seguir alguna de las distintas guías que ofrece Docker Inc. Existe la posibilidad de en el momento de la descarga escoger una versión específica para ser instalada, no es necesario que sea la más reciente.

El siguiente paso es crear una cuenta en Docker Hub donde se comparten imágenes de manera pública, a niveles personales y oficiales, y se puede tener un repositorio privado de manera gratuita.
Desde este enlace Sign up for Docker Hub se puede dar de alta una cuenta, una vez se crea esa cuenta se debe asociar a nuestra herramienta local Docker. Si no realizamos esta asociación no podremos compartir ni descargar imágenes.
Para vincular nuestra cuenta en Docker Hub usamos el comando en terminal *docker login*.

Tras realizar los pasos sugeridos se puede realizar la comprobación para percibir si se ha instalado correctamente con el comando destinado a ello, *sudo docker run hello-world* . Este comando nos devuelve algo similar a lo siguiente:

```
i2smorillo@i2smorillo-XPS:~$ sudo docker run hello-world

Hello from Docker!
This message shows that your installation appears to be working correctly.

To generate this message, Docker took the following steps:
 1. The Docker client contacted the Docker daemon.
 2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
    (amd64)
 3. The Docker daemon created a new container from that image which runs the
    executable that produces the output you are currently reading.
 4. The Docker daemon streamed that output to the Docker client, which sent it
    to your terminal.

To try something more ambitious, you can run an Ubuntu container with:
 $ docker run -it ubuntu bash

Share images, automate workflows, and more with a free Docker ID:
 https://hub.docker.com/

For more examples and ideas, visit:
 https://docs.docker.com/get-started/
```

Figura 13: Comprobación de la instalación de Docker Engine

Tras la confirmación de la instalación de Docker, se procede a instalar Docker Compose. De nuevo Docker Inc., nos ofrece una guía de instalación oficial: Install Docker Compose
Para la comprobación de esta nueva instalación se puede usar el comando *docker-compose -v* que muestra la versión de Docker Compose instalada.

Esta otra guía nos muestra como configurar todos los permisos que pueden ser necesarios en referencia al Docker CLI: Post-installation steps for Linux | Docker Documentation

3.2 Tutorial 2: Primeros pasos con Docker

3.2.1 Definir un contenedor a través de Dockerfile

Dockerfile define lo que sucede en el entorno dentro de su contenedor. El acceso a recursos como interfaces de red y unidades de disco está virtualizado dentro de este entorno, que está aislado del resto del sistema, por lo que debe asignar puertos de manera exterior y ser específico sobre qué archivos se desea "copiar" en el contenedor.
La ventaja de esta definición del dockerfile es que en dondequiera que se ejecute, mantendrá el mismo comportamiento. Este aspecto ofrece portabilidad de las imágenes de Docker además de su reducido tamaño.

Dockerfile:
El Dockerfile que va a definir el comportamiento del contenedor para esta actividad es el siguiente, contiene comentarios de que acción realiza cada comando:

```
# Specify a base image
FROM python:2.7-slim

# Set the working directory
WORKDIR /usr/app

# Copy my local directory into the path container /usr/app
COPY . /usr/app

# Install any needed packages specified in requirements.txt
RUN pip install --trusted-host pypi.python.org -r requirements.txt

# Expose the container port 80
EXPOSE 80

# Define environment variable
ENV NAME World

# Run app.py when the container launches
CMD ["python","app.py"]
```

Código 4: Dockerfile para el desarrollo de la imagen friendlyhello

En este archivo Dockerfile se hace referencia a dos archivos (app.py y requirements.txt) que se definen a continuación. Estos archivos deben estar en el mismo directorio que el Dockerfile según se define el path que debe copiar en el contenedor y las instrucciones ejecutadas.

A continuación, se presentan algunas instrucciones de Dockerfile, algunas se encuentran en este archivo Dockerfile, código 4, y otras no pero son de gran utilidad y uso frecuente [19].

La primera sentencia, FROM, escoge la imagen base a partir de la cual se va a construir la nueva imagen de Docker, se puede hacer uso de imágenes locales u otras obtenidas de manera online. Si la imagen no se encuentra en local, se intenta obtener la mencionada imagen en Docker Hub, a no ser que se le indique un repositorio distinto.

Estas imágenes base suelen ocupar poco tamaño. Esto se consigue con incluyendo el software mínimo necesario, drivers imprescindibles, comandos más usados y solo las opciones más usadas de estos comandos.

En Docker Hub se obtienen descripciones de las imágenes que aloja donde se puede consultar que software o aplicaciones contiene la imagen. A partir de estas imágenes se debe adecuar la construcción de las imágenes propias para ajustarlo a nuestro objetivo.

En el caso de este ejemplo que se usa la imagen python:2.7-slim, y se trata de una imagen oficial alojada en Docker Hub se puede obtener información de ella y el esto de versiones en el repositorio publico Docker Hub.

A continuación, se escoge cual va a ser el directorio de trabajo inicial con WORKDIR y en la tercera sentencia se copia, COPY, todo el directorio local actual en la ubicación */usr/app* del contenedor.

Una función similar a la de la instrucción COPY es la que realiza la instrucción ADD, esta permite no solo agregar archivos a la imagen que se encuentren en local, sino que puede añadir archivos descargados que proporcionemos con una URL.

La instrucción RUN ejecuta el comando que le acompaña, en este caso se usa para instalar en la imagen que construye las necesidades de la aplicación cuando esta se ejecute.

EXPOSE sirve para definir que puertos expone el contenedor que ejecuta la imagen resultante de este Dockerfile, por defecto EXPOSE asume que se hace uso del protocolo TCP, pero es posible indicar el uso de UDP.

La instrucción ENV realiza una asociación clave-valor en las variables de entorno en la imagen resultante.

Una instrucción que no aparece en este Dockerfile pero puede ser de utilidad es ARG, es la única que puede preceder a FROM. La instrucción ARG define una variable que los usuarios pueden entregar y establecer en el momento de la compilación al constructor con el comando de compilación Docker mediante el indicador --build-arg <varname>=<value>.
Si se entrega un clave-valor que no se ha definido con ARG en el Dockerfile se genera una advertencia.

CMD proporciona unos valores por defecto al contenedor, estos valores se pueden tratar de unas sentencias ejecutables. En el caso de este Dockerfile se utiliza para que se ejecute de manera predeterminada la sentencia *python app.py.*
El comando CMD debe ser único en el Dockerfile y en el caso de existir más de uno solo se seguirán las directivas del último.

Otra instrucción importante sobre todo en los casos que se tiene control de permisos es USER. Esta instrucción establece el nombre de usuario (o UID) y, opcionalmente, el grupo de usuarios (o GID) que se usará al ejecutar la imagen o que ejecutará las posteriores instrucciones RUN, CMD y ENTRYPOINT del mismo Dockerfile.

En este ejemplo no se hace uso de ello, pero es habitual el uso del comando ENTRYPOINT, este comando se usa para establecer sentencias ejecutables que siempre se ejecutaran al iniciar el contenedor. De manera diferente al comando CMD, este no puede ser ignorado o sustituido aun indicando otros parámetros como entrada mediante el CLI de Docker.

Se hace uso habitualmente de la combinación entre los comandos CMD y ENTRYPOINT para establecer comandos ejecutables de inicio del contenedor en el cual se establecen valores por defecto, pero pueden ser modificados al inicio de la ejecución. En modo resumen, se

establece la parte fija y no editable con el comando ENTRYPOINT y os valores por defecto que podrían ser modificados con CMD.

Un pequeño ejemplo de esto es el siguiente:

```
FROM alpine
ENTRYPOINT [ "echo", "Hello"]
CMD ["World!"]
```

Código 5: Dockerfile muestra de diferencia entre CMD y ENTRYPOINT

La ejecución de la imagen resultante de este pequeño Dockerfile devuelve por pantalla: *Hello World!* Pero si le indicamos un parámetro distinto en el arranque de la ejecución, como por ejemplo Samuel, esto modifica el CMD intercambiando el valor por defecto por el indicado.

```
i2smorillo@i2smorillo-XPS:~/example_entrypoint_cmd$ docker build -t example_entrypoint_cmd .
Sending build context to Docker daemon  2.048kB
Step 1/3 : FROM alpine
 ---> e66264b98777
Step 2/3 : ENTRYPOINT [ "echo", "Hello"]
 ---> Using cache
 ---> 0ec300d7d4cd
Step 3/3 : CMD ["World!"]
 ---> Using cache
 ---> 3f4f064a3bfc
Successfully built 3f4f064a3bfc
Successfully tagged example_entrypoint_cmd:latest
i2smorillo@i2smorillo-XPS:~/example_entrypoint_cmd$ docker run example_entrypoint_cmd
Hello World!
i2smorillo@i2smorillo-XPS:~/example_entrypoint_cmd$ docker run example_entrypoint_cmd Samuel
Hello Samuel
```

Figura 14: muestra de diferencia entre CMD y ENTRYPOINT

Se realizará otro ejemplo relacionado más adelante, a continuación, se continua con la tarea principal del apartado en curso.

requirements.txt:
Define los paquetes Python que van a ser instalados en el contenedor, antes incluso del inicio de su ejecución. En este caso el contenido de este archivo es:

```
Flask
Redis
```

Instala las librerías Flask y Redis para Python, debido a que Redis no se está ejecutando (hemos instalado la librería de Python, pero no Redis en sí), se debe esperar que el intento de usarlo produzca un fallo.

app.py:
Es la definición en sí de la aplicación en lenguaje Python:

```
from flask import Flask
from redis import Redis, RedisError
import os
import socket

# Connect to Redis
redis = Redis(host="redis", db=0, socket_connect_timeout=2,
socket_timeout=2)

app = Flask(__name__)

@app.route("/")
def hello():
    try:
        visits = redis.incr("counter")
    except RedisError:
        visits = "<i>cannot connect to Redis, counter disabled</i>"

    html = "<h3>Hello {name}!</h3>" \
           "<b>Hostname:</b> {hostname}<br/>" \
           "<b>Visits:</b> {visits}"
    return html.format(name=os.getenv("NAME", "world"),
hostname=socket.gethostname(), visits=visits)

if __name__ == "__main__":
    app.run(host='0.0.0.0', port=80)
```

Código 6: Aplicación app.py que se usa en el desarrollo de la imagen friendlyhello [2]

Una vez creados estos archivos es momento de comenzar la compilación de la imagen para su posterior ejecución.

3.2.2 Compilación de la imagen Docker

En este momento debe asegurarse de que está en el directorio correcto y todos los archivos en este mismo:

```
i2smorillo@i2smorillo-XPS:~/Tutorial_2$ ls
app.py  Dockerfile  requirements.txt
```

Figura 15: Archivos necesarios tutorial 3.2

A continuación, se hace uso del comando *docker build* para construir una imagen de Docker que a través de la opción *-t* vamos a etiquetar como *friendlyhello*:

```
i2smorillo@i2smorillo-XPS:~/Tutorial_2$ docker image ls
REPOSITORY   TAG       IMAGE ID   CREATED   SIZE
i2smorillo@i2smorillo-XPS:~/Tutorial_2$ docker build -t friendlyhello .
Sending build context to Docker daemon  4.608kB
Step 1/7 : FROM python:2.7-slim
2.7-slim: Pulling from library/python
123275d6e508: Pull complete
dd1cd6637523: Pull complete
0c4e6d630f2c: Pull complete
13e9cd8f0ea1: Pull complete
Digest: sha256:6c1ffdff499e29ea663e6e67c9b6b9a3b401d554d2c9f061f9a45344e3992363
Status: Downloaded newer image for python:2.7-slim
 ---> eeb27ee6b893
Step 2/7 : WORKDIR /usr/app
 ---> Running in d13105a7b809
Removing intermediate container d13105a7b809
 ---> 665ccdd13e4e
Step 3/7 : COPY . /usr/app
 ---> 34e9079b83eb
Step 4/7 : RUN pip install --trusted-host pypi.python.org -r requirements.txt
 ---> Running in 7b56a10bc90b
DEPRECATION: Python 2.7 reached the end of its life on January 1st, 2020. Please upgrade your Python as Python 2.7 is no longe
s://pip.pypa.io/en/latest/development/release-process/#python-2-support
Collecting Flask
  Downloading Flask-1.1.4-py2.py3-none-any.whl (94 kB)
Collecting Redis
  Downloading redis-3.5.3-py2.py3-none-any.whl (72 kB)
Collecting click<8.0,>=5.1
  Downloading click-7.1.2-py2.py3-none-any.whl (82 kB)
Collecting Werkzeug<2.0,>=0.15
  Downloading Werkzeug-1.0.1-py2.py3-none-any.whl (298 kB)
Collecting Jinja2<3.0,>=2.10.1
  Downloading Jinja2-2.11.3-py2.py3-none-any.whl (125 kB)
Collecting itsdangerous<2.0,>=0.24
  Downloading itsdangerous-1.1.0-py2.py3-none-any.whl (16 kB)
Collecting MarkupSafe>=0.23
  Downloading MarkupSafe-1.1.1-cp27-cp27mu-manylinux1_x86_64.whl (24 kB)
Installing collected packages: click, Werkzeug, MarkupSafe, Jinja2, itsdangerous, Flask, Redis
Successfully installed Flask-1.1.4 Jinja2-2.11.3 MarkupSafe-1.1.1 Redis-3.5.3 Werkzeug-1.0.1 click-7.1.2 itsdangerous-1.1.0
WARNING: You are using pip version 20.0.2; however, version 20.3.4 is available.
You should consider upgrading via the '/usr/local/bin/python -m pip install --upgrade pip' command.
Removing intermediate container 7b56a10bc90b
 ---> c91e8e3d8cc2
Step 5/7 : EXPOSE 80
 ---> Running in 9cbbc16019ff
Removing intermediate container 9cbbc16019ff
 ---> 52963a61dc5a
Step 6/7 : ENV NAME World
 ---> Running in d13ee3fa5709
Removing intermediate container d13ee3fa5709
 ---> 1c2dabcf430a
Step 7/7 : CMD ["python","app.py"]
 ---> Running in b90060081e4f
Removing intermediate container b90060081e4f
 ---> 447d2eda16b7
Successfully built 447d2eda16b7
Successfully tagged friendlyhello:latest
i2smorillo@i2smorillo-XPS:~/Tutorial_2$ docker image ls
REPOSITORY      TAG         IMAGE ID       CREATED          SIZE
friendlyhello   latest      447d2eda16b7   19 seconds ago   159MB
python          2.7-slim    eeb27ee6b893   18 months ago    148MB
i2smorillo@i2smorillo-XPS:~/Tutorial_2$
```

```
i2smorillo@i2smorillo-XPS:~/Tutorial_2$ docker image ls
REPOSITORY   TAG       IMAGE ID   CREATED   SIZE
i2smorillo@i2smorillo-XPS:~/Tutorial_2$ docker build -t friendlyhello .
Sending build context to Docker daemon  4.608kB
Step 1/7 : FROM python:2.7-slim
2.7-slim: Pulling from library/python
123275d6e508: Pull complete
dd1cd6637523: Pull complete
0c4e6d630f2c: Pull complete
13e9cd8f0ea1: Pull complete
Digest: sha256:6c1ffdff499e29ea663e6e67c9b6b9a3b401d554d2c9f061f9a45344e3992363
Status: Downloaded newer image for python:2.7-slim
 ---> eeb27ee6b893
Step 2/7 : WORKDIR /usr/app
 ---> Running in d13105a7b809
Removing intermediate container d13105a7b809
 ---> 665ccdd13e4e
Step 3/7 : COPY . /usr/app
 ---> 34e9079b83eb
Step 4/7 : RUN pip install --trusted-host pypi.python.org -r requirements.txt
 ---> Running in 7b56a10bc90b
DEPRECATION: Python 2.7 reached the end of its life on January 1st, 2020. Please upgrade your Python as Python 2.7 is no lor
details about Python 2 support in pip, can be found at https://pip.pypa.io/en/latest/development/release-process/#python-2-s
Collecting Flask
  Downloading Flask-1.1.4-py2.py3-none-any.whl (94 kB)
Collecting Redis
  Downloading redis-3.5.3-py2.py3-none-any.whl (72 kB)
Collecting click<8.0,>=5.1
  Downloading click-7.1.2-py2.py3-none-any.whl (82 kB)
Collecting Werkzeug<2.0,>=0.15
  Downloading Werkzeug-1.0.1-py2.py3-none-any.whl (298 kB)
Collecting Jinja2<3.0,>=2.10.1
  Downloading Jinja2-2.11.3-py2.py3-none-any.whl (125 kB)
Collecting itsdangerous<2.0,>=0.24
  Downloading itsdangerous-1.1.0-py2.py3-none-any.whl (16 kB)
Collecting MarkupSafe>=0.23
  Downloading MarkupSafe-1.1.1-cp27-cp27mu-manylinux1_x86_64.whl (24 kB)
Installing collected packages: click, Werkzeug, MarkupSafe, Jinja2, itsdangerous, Flask, Redis
Successfully installed Flask-1.1.4 Jinja2-2.11.3 MarkupSafe-1.1.1 Redis-3.5.3 Werkzeug-1.0.1 click-7.1.2 itsdangerous-1.1.0
WARNING: You are using pip version 20.0.2; however, version 20.3.4 is available.
You should consider upgrading via the '/usr/local/bin/python -m pip install --upgrade pip' command.
Removing intermediate container 7b56a10bc90b
 ---> c91e8e3d8cc2
```

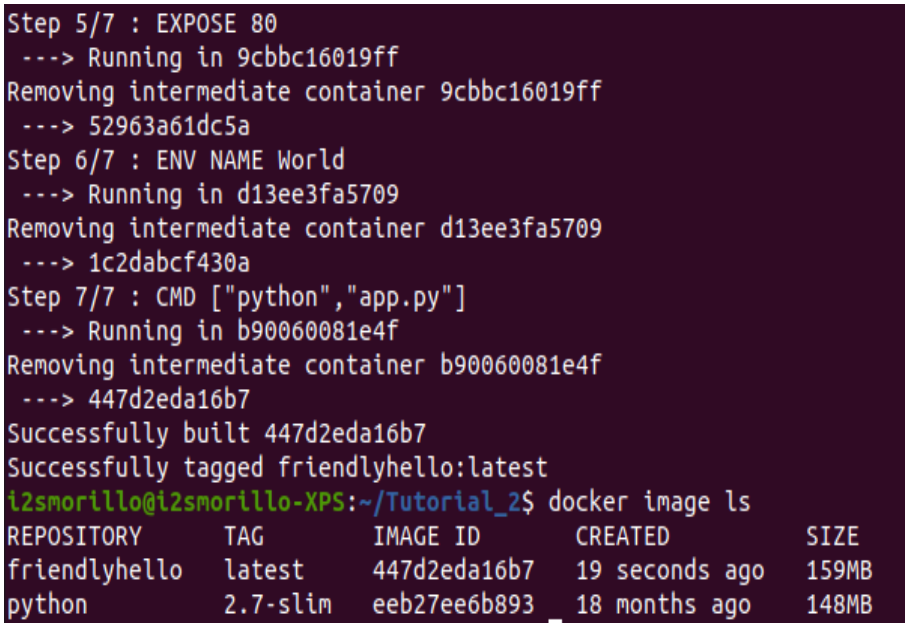

Figura 16: Construcción con docker build en el tutorial 3.2

Como se puede observar con la sentencia *docker image ls* la imagen se encuentra construida en nuestro sistema de manera local. Para la construcción de esta imagen ha sido necesario primero que se descargue la imagen base que se ha especificado en el Dockerfile, tras ello se ha ido construyendo contenedores temporales para ir cumpliendo con los objetivos de cada paso hasta que se obtiene una imagen final con el tag que se especifica.
Esta construcción ha requerido de un contenedor temporal en cada paso ya que es la primera vez que se realiza. Si en algún momento se decide modificar el Dockerfile la nueva construcción de la imagen intentará hacer uso de la caché, es decir, solo construirá los contenedores temporales a partir del paso que se ha modificado y por tanto modifica el comportamiento de la imagen.

Nota: Aparecen errores relacionados con las versiones de Python y pip que no son críticos y se puede continuar con el tutorial a pesar de ello.

3.2.3 Ejecución de la imagen Docker

Se va a ejecutar la aplicación con el formato de sentencia:

```
docker run -p
<puerto_expuesto_exteriormente>:<puerto_redirigido_al_contenedor>
<imagen_docker>
```

En este caso se expondrá el puerto 1913 conectado al puerto 80 que es el cual se encuentra escuchando el contenedor y la aplicación que se ejecuta.

```
i2smorillo@i2smorillo-XPS:~/Tutorial_2$ docker run -p 1913:80 friendlyhello
 * Serving Flask app "app" (lazy loading)
 * Environment: production
   WARNING: This is a development server. Do not use it in a production deployment.
   Use a production WSGI server instead.
 * Debug mode: off
 * Running on http://0.0.0.0:80/ (Press CTRL+C to quit)
172.17.0.1 - - [25/Oct/2021 15:50:26] "GET / HTTP/1.1" 200 -
172.17.0.1 - - [25/Oct/2021 15:50:26] "GET /favicon.ico HTTP/1.1" 404 -
172.17.0.1 - - [25/Oct/2021 15:50:57] "GET / HTTP/1.1" 200 -
172.17.0.1 - - [25/Oct/2021 15:50:57] "GET /favicon.ico HTTP/1.1" 404 -
172.17.0.1 - - [25/Oct/2021 15:51:00] "GET / HTTP/1.1" 200 -
```

Figura 17: Ejecución de la imagen friendlyhello construida en el tutorial 3.2

Una vez se encuentra ejecutándose la aplicación, se puede verificar con la ayuda de un buscador el funcionamiento de la aplicación web. Se debe hacer una búsqueda a localhost indicando el puerto que se ha expuesto de manera exterior. Se puede comprobar también el funcionamiento de la aplicación mediante curl (*Client URL*), el cual es un software libre que se emplea como navegador web de texto.

Figura 18: Comprobación con un navegador del servicio web alojado en el contenedor Docker con imagen friendlyhello almacenada localmente

```
i2smorillo@i2smorillo-XPS:~/Tutorial_2$ curl http://0.0.0.0:1913/
<h3>Hello World!</h3><b>Hostname:</b> a27590536a71<br/><b>Visits:</b> <i>cannot connect to Redis, counter disabled</i>
```

Figura 19: Comprobación con un curl del servicio web alojado en el contenedor Docker con imagen friendlyhello almacenada localmente

La aplicación web muestra el hostname del contenedor Docker que responde a la petición, se muestra el ID del contenedor como se puede comprobar con *docker ps* . El Redis como hemos anticipado aún no está configurado su funcionamiento, se hace uso de las funciones de redis más adelante.

Se va a proceder a ejecutar la aplicación de otro modo, para ello primero se debe detener el contenedor con CTRL+C en la propia terminal donde está corriendo o en otra terminal distinta del host que aloja el contenedor. Siguiendo la segunda opción, se detiene el contenedor mediante el comando:

```
docker container stop <ID_CONTAINER o NAME_CONTAINER>
```

```
i2smorillo@i2smorillo-XPS:~/Tutorial_2$ docker ps
CONTAINER ID   IMAGE          COMMAND           CREATED          STATUS          PORTS                                       NAMES
a27590536a71   friendlyhello  "python app.py"   11 minutes ago   Up 11 minutes   0.0.0.0:1913->80/tcp, :::1913->80/tcp       dazzling_shannon
i2smorillo@i2smorillo-XPS:~/Tutorial_2$ docker container stop a27590536a71
a27590536a71
```

Figura 20: Comandos docker ps y docker container stop para detener un contenedor

Una vez el contenedor de Docker se ha detenido, devuelve como parámetro el ID del contenedor detenido.

Se va a ejecutar de igual modo la imagen, pero esta vez se va a introducir en el comando *docker run* la opción *-d* , está opción hace que el contenedor sea independiente a la terminal en la que se inicia Docker, es decir, el contenedor va a ejecutar en segundo plano.

```
i2smorillo@i2smorillo-XPS:~/Tutorial_2$ docker ps
CONTAINER ID   IMAGE     COMMAND   CREATED   STATUS    PORTS     NAMES
i2smorillo@i2smorillo-XPS:~/Tutorial_2$ docker run -d -p 1913:80 friendlyhello
d359ff5159f01190fb199f56585c38803e9bdae996f316bfecc56551d017c522
i2smorillo@i2smorillo-XPS:~/Tutorial_2$ docker ps
CONTAINER ID   IMAGE          COMMAND           CREATED          STATUS          PORTS                                       NAMES
d359ff5159f0   friendlyhello  "python app.py"   11 seconds ago   Up 10 seconds   0.0.0.0:1913->80/tcp, :::1913->80/tcp       practical_thompson
```

Figura 21: Ejecución de un contenedor en modo detached

Esta vez la sentencia que inicia el contenedor Docker devuelve como parámetro el ID del contenedor completo sin simplificar. En cuanto al comportamiento del contenedor es exactamente igual.

3.2.4 Compartir las imágenes de Docker en un repositorio

Esta es una buena demostración de la gran portabilidad que proporciona la herramienta Docker. Es habitual que las imágenes no estén almacenadas en local sino en un repositorio remoto en el que se tiene acceso a su descarga desde cualquier dispositivo.

Se debe tener una cuenta registrada de manera gratuita en hub.docker.com, con dicha cuenta se debe iniciar sesión en el host local con *docker login* .

Asociar una etiqueta a una imagen es opcional, este etiquetado es de gran ayuda para que sea más sencillo llevar a cabo un control de versiones. La sintaxis más habitual y útil es:

```
docker tag <nombre de la imagen> <username hub
docker>/<repositorio>:<tag>
```

```
i2smorillo@i2smorillo-XPS:~/Tutorial_2$ docker image ls
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
friendlyhello       latest              447d2eda16b7        43 minutes ago      159MB
python              2.7-slim            eeb27ee6b893        18 months ago       148MB
i2smorillo@i2smorillo-XPS:~/Tutorial_2$ docker tag friendlyhello smorilloh/get-started:tutorial2
i2smorillo@i2smorillo-XPS:~/Tutorial_2$ docker image ls
REPOSITORY                TAG                 IMAGE ID            CREATED             SIZE
friendlyhello             latest              447d2eda16b7        46 minutes ago      159MB
smorilloh/get-started     tutorial2           447d2eda16b7        46 minutes ago      159MB
python                    2.7-slim            eeb27ee6b893        18 months ago       148MB
```

Figura 22: Crear una imagen con nombre y tag con el comando docker tag

En la imagen se puede ver como existe la imagen con el tag especificado con el mismo tamaño que esta misma imagen que solo difiere en nombre y tag.

A continuación, debe subirse la imagen etiquetada al repositorio. Para ello se usa el comando con la siguiente estructura:

```
docker push <username hub docker>/<repositorio>:<tag>
```

Una vez subida la imagen en el repositorio se puede observar mediante la consola web en hub.docker.com, se observará algo similar a la siguiente imagen:

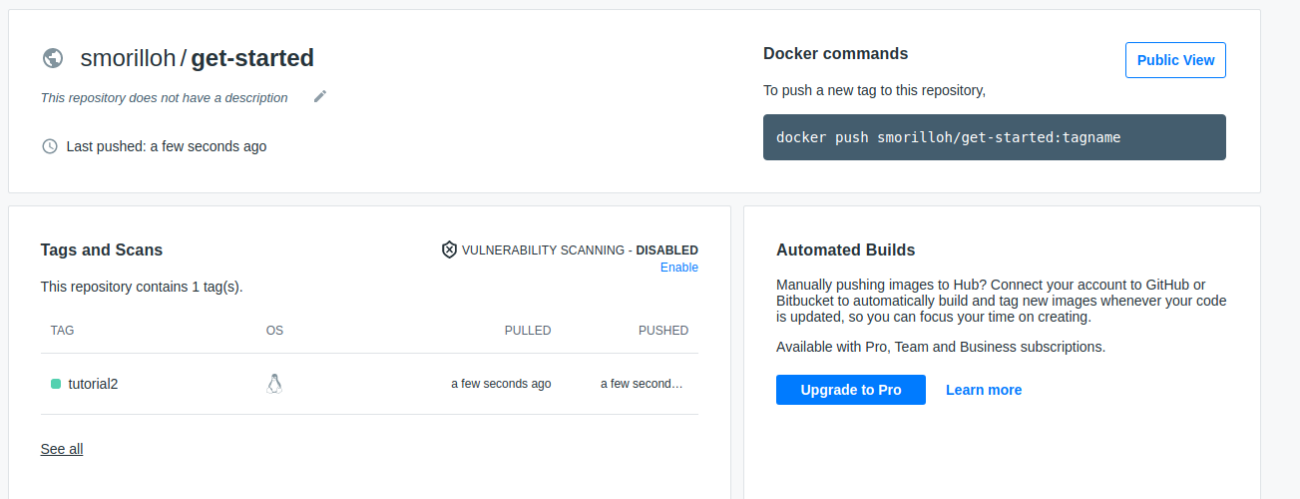

Figura 23: Repositorio asignado a un usuario en Docker Hub

3.2.5 Ejecución de imágenes obtenidas desde el repositorio remoto

El repositorio remoto permite hacer uso de la imagen almacenada en cualquier equipo con el comando que sigue la siguiente sintaxis:

```
docker run -p
<puerto_expuesto_exteriormente>:<puerto_redirigido_al_contenedor>
<username hub docker>/<repositorio>:<tag>
```

Si la imagen no existe de manera local en el equipo esté la descarga del repositorio indicado. Para eliminar la imagen local y comprobar como se desarrolla la descarga desde el repositorio remoto se puede hacer uso de la sentencia:

```
docker system prune -a
```

Se ejecuta la sentencia que inicia la ejecución de docker a partir de la imagen almacenada en el repositorio. Se puede verificar que el funcionamiento es exactamente igual a cuando esta imagen se encontraba almacenada de manera local.

```
i2smorillo@i2smorillo-XPS:~/Tutorial_2$ docker run -p 2017:80 smorilloh/get-started:tutorial2
Unable to find image 'smorilloh/get-started:tutorial2' locally
tutorial2: Pulling from smorilloh/get-started
123275d6e508: Pull complete
dd1cd6637523: Pull complete
0c4e6d630f2c: Pull complete
13e9cd8f0ea1: Pull complete
45d1e499e43c: Pull complete
fced74fc3210: Pull complete
2aacabad3a14: Pull complete
Digest: sha256:aebe46b979a955eeeb05311295a76b1ef710e56178f991fc55bef913efbc201f
Status: Downloaded newer image for smorilloh/get-started:tutorial2
 * Serving Flask app "app" (lazy loading)
 * Environment: production
   WARNING: This is a development server. Do not use it in a production deployment.
   Use a production WSGI server instead.
 * Debug mode: off
 * Running on http://0.0.0.0:80/ (Press CTRL+C to quit)
172.17.0.1 - - [25/Oct/2021 16:40:37] "GET / HTTP/1.1" 200 -
172.17.0.1 - - [25/Oct/2021 16:40:37] "GET /favicon.ico HTTP/1.1" 404 -
```

Figura 24: Ejecución de la imagen friendlyhello obtenida del repositorio remoto en el tutorial 3.2

Esta vez se ha escogido exponer otro puerto de manera exterior al contenedor para demostrar que no influye en el comportamiento del contenedor. Verifica que esto es así, solo se debe indicar de manera correcta el puerto en la búsqueda.

Figura 25: Comprobación con un navegador del servicio web alojado en el contenedor Docker con imagen friendlyhello descargada del repositorio

3.3 Tutorial 3: Servicios Docker definidos en docker-comopose.yml

3.3.1 Definición de servicios

En una aplicación distribuida, diferentes partes de la aplicación se denominan "servicios".
Los servicios son en realidad "contenedores". Un servicio ejecuta una imagen, pero define la forma en que se ejecuta la imagen: puertos a usar, cuántas réplicas del contenedor deben ejecutarse para que el servicio tenga la capacidad que necesita, etc.
El número de réplicas de un servicio puede cambiar el número de contenedores que se ejecutan, esto se denomina escalar el servicio. El escalado puede ser tanto aumentando

como disminuyendo el número de réplicas. Esta definición de los servicios y escalados se define en el archivo docker-compose.yml.

3.3.2 Definición del archivo docker-compose.yml

Es un archivo tipo YAML en el que se define el comportamiento y escalado de los servicios Docker.

Continuando a partir de la imagen almacenada en el repositorio en el tutorial anterior se define el siguiente docker-compose.yml:

```
version: "3"
services:
  web:
    image: smorilloh/get-started:tutorial2
    deploy:
      replicas: 3
      restart_policy:
        condition: on-failure
      resources:
        limits:
          cpus: "0.1"
          memory: 50M
    ports:
      -"4000:80"
    networks:
      -webnet
networks:
  webnet:
```

Código 7: Definición del docker-compose.yml con imagen smorilloh/get-started:tutorial2 y tres réplicas

Define que la imagen a usar es "*smorilloh/get-started:tutorial2"*, almacenada en Docker hub. Ejecutando tres réplicas de dicha imagen, cada una de ellas tiene limitado el uso de CPU al 10% de todos los núcleos disponibles y 50MB de memoria RAM.
Las definiciones del docker-comopose.yml declaran que se interconectan el puerto 4000 de manera exterior con el puerto 80 interior del contenedor. El puerto 80 se comparte entre los distintos contenedores a través de la red *webnet* con balanceo de carga.
La política de reinicio en este caso es on-failure, es decir, se reinicia el contenedor si sale debido a un error, que se manifiesta como un código de salida distinto de cero.

3.3.3 Políticas de reinicio del servicio en docker-compose.yml

Existen cuatro posibilidades para definir las condiciones de la política de reinicio:

Flag restart_policy	Descripción de la política de reinicio
`always`	Siempre se reinicia el contenedor si se detiene. Si se detiene manualmente, solo se reinicia cuando el demonio de Docker se reinicia o el propio contenedor se reinicia manualmente.

`no`	Es el valor por defecto si no se le indica otro. Implica que no se debe reiniciar automáticamente el contendor.
`on-failure`	Se reinicia el contenedor si este sale debido a un error.
`unless-stopped`	Similar a always, excepto que cuando el contenedor se detiene (manualmente o de otro modo), no se reinicia, aunque se reinicie el demonio Docker.

Figura 26: Tabla de posibilidades del flag restart_policy [20]

3.3.4 Docker swarm

En esta parte se va a utilizar Docker swarm para orquestar los distintos contendores, swarm también es conocido como enjambres. Docker swarm permite agrupar varios clusters y gestionarlos de manera centralizada.

Docker Swarm se basa en el modelo maestro-esclavo. Cada clúster de Docker está formado por al menos un nodo maestro, también referido como administrador o manager, y el número de nodos esclavos, también llamados workers, que se quiera. Es incluso posible que no existan nodos esclavos y solo exista un nodo manager que este si es necesario. El maestro del enjambre es responsable de la gestión del clúster y la delegación de tareas, el esclavo se encarga de ejecutar las tareas que el maestro asigna [21].

Los administradores del Swarm tienen varias estrategias para realizar el balanceo de carga entre los workers, esto se define en el docker-compose.yml. El modo Swarm tiene un componente DNS interno que asigna automáticamente a cada servicio del enjambre una entrada de DNS. El administrador de enjambres utiliza el equilibrio de carga interno para distribuir las solicitudes entre los servicios dentro del clúster según el nombre DNS del servicio.

3.3.5 Ejecución de la aplicación en el swarm

El primer paso para la ejecución del swarm es iniciarlo con el comando:

```
docker swarm init [OPCIONES]
```

En este caso no se introducirá ningún parámetro opcional, iniciará el Swarm como manager y devuelve un token que se debe introducir si se desea generar en otra instancia un Swarm esclavo de este manager.

```
i2smorillo@i2smorillo-XPS:~/Tutorial_3$ docker swarm init
Swarm initialized: current node (vipwqv1yfp4o77g5runybejjz) is now a manager.

To add a worker to this swarm, run the following command:

    docker swarm join --token SWMTKN-1-63det6whadueaqsnl4uwfgwygbr6uyllwh00kjku8aha1q41i0-5rgtbtmk4vtzubtclu2wmthv6 192.168.0.18:2377

To add a manager to this swarm, run 'docker swarm join-token manager' and follow the instructions.

i2smorillo@i2smorillo-XPS:~/Tutorial_3$ docker node ls
ID                            HOSTNAME          STATUS    AVAILABILITY   MANAGER STATUS   ENGINE VERSION
vipwqv1yfp4o77g5runybejjz *   i2smorillo-XPS    Ready     Active         Leader           20.10.10
```

Figura 27: Inicio de un swarm de docker con docker swarm init

Para ejecutar los servicios definidos en el docker-compose.yml se utiliza el comando:

```
docker stack deploy -c docker-compose.yml getstartedlab
```

```
i2smorillo@i2smorillo-XPS:~/Tutorial_3$ docker stack deploy -c docker-compose.yml getstartedlab
Creating network getstartedlab_webnet
Creating service getstartedlab_web
i2smorillo@i2smorillo-XPS:~/Tutorial_3$ docker service ls
ID            NAME              MODE         REPLICAS   IMAGE                             PORTS
091bn67ftmht  getstartedlab_web  replicated   3/3        smorilloh/get-started:tutorial2   *:4000->80/tcp
i2smorillo@i2smorillo-XPS:~/Tutorial_3$ docker service ps getstartedlab_web
ID            NAME                IMAGE                             NODE           DESIRED STATE  CURRENT STATE
o0dqrh7of91u  getstartedlab_web.1  smorilloh/get-started:tutorial2   i2smorillo-XPS  Running        Running about a mi
zdis3z7qamvd  getstartedlab_web.2  smorilloh/get-started:tutorial2   i2smorillo-XPS  Running        Running about a mi
4ot8t3z5gc68  getstartedlab_web.3  smorilloh/get-started:tutorial2   i2smorillo-XPS  Running        Running about a mi
```

Figura 28: Creación de servicios en un Swarm a partir de un docker-compose.yml

Una vez levantado el servicio se puede observar sus características y tal como estaba definido en este único Swarm se ejecutan las tres réplicas indicadas. Si se realiza una búsqueda en el navegador con la dirección localhost al puerto expuesto exteriormente como en ejemplos anteriores se verifica que muestra la web con el ID del contenedor Docker que ha respondido a la petición. Se puede comprobar realizando varias búsquedas como el balanceo de carga produce que cada vez se reciba la respuesta de uno de los contenedores.

3.3.6 Escalado del número de contenedores

Se debe modificar el número de réplicas en el archivo docker-compopse.yml. Un ejemplo podría ser el escalado hacia arriba levantando ahora cinco réplicas en total. Para dicho ejemplo se usa el docker-compose.yml siguiente:

```yaml
version: "3"
services:
  web:
    image: smorilloh/get-started:tutorial2
    deploy:
      replicas: 5
      restart_policy:
        condition: on-failure
      resources:
        limits:
          cpus: "0.1"
          memory: 50M
    ports:
      - "4000:80"
    networks:
      - webnet
networks:
  webnet:
```

Código 8: Definición del docker-compose.yml con imagen smorilloh/get-started:tutorial2 y cinco réplicas

Tras la modificación, sin ser necesario modificar el Swarm, se redespliega lo definido en el docker-compose.yml con el mismo comando que se hizo la primera vez:

```
docker stack deploy -c docker-compose.yml getstartedlab
```

Esto escalará el número de réplicas sin destruir las anteriores.

```
i2smorillo@i2smorillo-XPS:~/Tutorial_3$ docker ps
CONTAINER ID   IMAGE                              COMMAND           CREATED          STATUS
1d0be2f4f204   smorilloh/get-started:tutorial2   "python app.py"   29 minutes ago   Up 29 minute
61cbfb8355fa   smorilloh/get-started:tutorial2   "python app.py"   29 minutes ago   Up 29 minute
aba56212d5f1   smorilloh/get-started:tutorial2   "python app.py"   29 minutes ago   Up 29 minute
i2smorillo@i2smorillo-XPS:~/Tutorial_3$ docker stack deploy -c docker-compose.yml getstartedlab
Updating service getstartedlab_web (id: 091bn67ftmht1d20mnmard9ru)
i2smorillo@i2smorillo-XPS:~/Tutorial_3$ docker ps
CONTAINER ID   IMAGE                              COMMAND           CREATED          STATUS
70a670ba31a5   smorilloh/get-started:tutorial2   "python app.py"   4 seconds ago    Up 1 second
95389f55a0da   smorilloh/get-started:tutorial2   "python app.py"   4 seconds ago    Up 1 second
1d0be2f4f204   smorilloh/get-started:tutorial2   "python app.py"   29 minutes ago   Up 29 minute
61cbfb8355fa   smorilloh/get-started:tutorial2   "python app.py"   29 minutes ago   Up 29 minute
aba56212d5f1   smorilloh/get-started:tutorial2   "python app.py"   29 minutes ago   Up 29 minute
```

Figura 29: Muestra del escalado de tres a cinco réplicas en la pila de servicios

Se puede observar como se han generado dos réplicas nuevas, pero el parámetro que relata el tiempo que lleva levantado un contenedor nos aclara que los tres contenedores anteriores permanecieron sin ser destruidos.

3.3.7 Eliminar la aplicación y el swarm

Para eliminar la aplicación se hace uso del comando:

```
docker stack rm [stacks]
```

A continuación, se elimina el enjambre con:

```
docker swarm leave --force
```

Es necesario usar el parámetro *--force* debido a que este nodo del enjambre está actuando como manager y es requisito obligatorio abandonar el enjambre de este modo.

```
i2smorillo@i2smorillo-XPS:~/Tutorial_3$ docker stack rm getstartedlab
Removing service getstartedlab_web
Removing network getstartedlab_webnet
i2smorillo@i2smorillo-XPS:~/Tutorial_3$ docker swarm leave --force
Node left the swarm.
```

Figura 30: Eliminación de los servicios y abandono de un manager del docker swarm

3.4 Tutorial 4: Docker swarm para ejecutar los servicios

3.4.1 Enjambres Docker distribuidos

En los ejemplos anteriores se ha utilizado un solo nodo en el enjambre, en este caso se muestra como pueden trabajar varios nodos maestros y esclavos alojados en distintas máquinas. Las distintas máquinas que alojan los nodos pueden ser físicas o virtuales. Este hecho es transparente para el enjambre que trabaja de igual modo.

3.4.2 Herramienta docker-machine

Docker Machine es una herramienta que nos ayuda a crear, configurar y manejar máquinas tanto virtuales como físicas, con Docker Engine. Esta herramienta se encarga de crear la máquina con Docker Engine en el entorno indicado y generar los certificados TLS para una comunicación segura con dicha máquina.

Se puede instalar la versión 0.7.0 que ha sido verificado su funcionamiento. Para instalar docker-machine version 0.7.0 y conceder permisos de ejecución en la máquina se puede usar el siguiente comando:

```
curl -L
https://github.com/docker/machine/releases/download/v0.7.0/docker-
machine-`uname -s`-`uname -m` > /usr/local/bin/docker-machine && \
chmod +x /usr/local/bin/docker-machine
```

Para comprobar su instalación se puede ejecutar el comando mostrado en la imagen y esperar la misma respuesta:

```
i2smorillo@i2smorillo-XPS:~/Tutorial_4$ docker-machine -version
docker-machine version 0.7.0, build a650a40
```

Figura 31: Comprobación de la instalación de la herramienta docker-machine

Docker-machine ofrece comandos de manera que ayudan a realizar pruebas de manera muy rápida y eficiente. Se puede consultar las distintas opciones con *docker-machine –help* .

3.4.3 Crear máquinas virtuales con docker-machine

En este tutorial se va a indicar que docker-machine realice la creación de las máquinas virtuales en un entorno de VirtualBox. Para crear las máquinas de dicha manera se usa el siguiente formato de comando:

```
docker-machine create --driver virtualbox <nombre_maquina_virtual>
```

Después de crearlas se puede visualizar estas máquinas en VirtualBox y con el comando:

```
docker-machine ls
```

```
i2smorillo@i2smorillo-XPS:~/Tutorial_4$ docker-machine create --driver virtualbox maquina-virtual-2
Running pre-create checks...
Creating machine...
(maquina-virtual-2) Copying /home/i2smorillo/.docker/machine/cache/boot2docker.iso to /home/i2smorillo/.docke
(maquina-virtual-2) Creating VirtualBox VM...
(maquina-virtual-2) Creating SSH key...
(maquina-virtual-2) Starting the VM...
(maquina-virtual-2) Check network to re-create if needed...
(maquina-virtual-2) Waiting for an IP...
Waiting for machine to be running, this may take a few minutes...
Detecting operating system of created instance...
Waiting for SSH to be available...
Detecting the provisioner...
Provisioning with boot2docker...
Copying certs to the local machine directory...
Copying certs to the remote machine...
Setting Docker configuration on the remote daemon...
Checking connection to Docker...
Docker is up and running!
To see how to connect your Docker Client to the Docker Engine running on this virtual machine, run: docker-ma
i2smorillo@i2smorillo-XPS:~/Tutorial_4$ docker-machine ls
NAME                ACTIVE   DRIVER       STATE     URL                         SWARM   DOCKER      ERRORS
maquina-virtual-1   -        virtualbox   Running   tcp://192.168.99.111:2376           v19.03.12
maquina-virtual-2   -        virtualbox   Running   tcp://192.168.99.112:2376           v19.03.12
```

Figura 32: Creación de una máquina virtual y listado de máquinas virtuales con la herramienta docker-machine

Se puede observar que han sido creadas dos máquinas con el nombre maquina-virtual-1 y maquina-virtual-2. Se puede acceder a su terminal si se desea mediante SSH (secure shell) o con el uso de la interfaz gráfica de VirtualBox.

Figura 33: Muestra de las máquinas virtuales creadas con docker-machine en la interfaz visual de VirtualBox

Si se desea usar la terminal local con las variables de entorno y funcionamiento de alguna de las máquinas virtuales se debe ejecutar el comando:

```
docker-machine env <nombre_maquina_virtual>
```

Tras ello, introducir el comando que devuelve, sigue una estructura como la siguiente:

```
eval $(docker-machine env <nombre_maquina_virtual>)
```

Para deshacer esta configuración y recuperar la ejecución habitual del terminal se debe ejecutar:

```
eval $(docker-machine env -u)
```

3.4.4 Inicializar los enjambres

Se inicia primero el enjambre en maquina-virtual-1, se ha seleccionado esta máquina como manager. Se usa la sentencia:

```
docker-machine ssh maquina-virtual-1 "docker swarm init --advertise-addr
<maquina-virtual-1 IP>"
```

De esta manera nos devuelve el token necesario para introducir en las otras máquinas que trabajan como workers a la orden de este administrador.

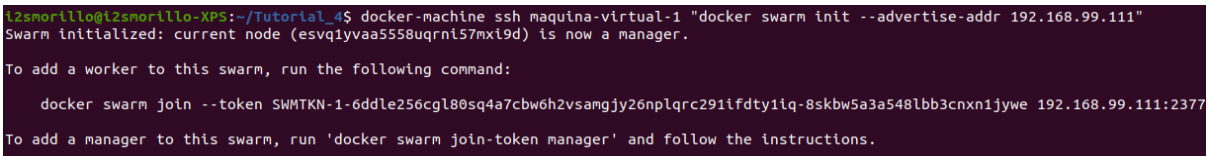

Figura 34: Inicio del Docker Swarm en una máquina virtual

Se introduce el token necesario en las otras máquinas virtuales que se desea que trabajen como workers de este administrador. Es importante que la sentencia para unirse como worker lleve dirección IP del manager acompañada del puerto 2377, el puerto mostrado por *docker-machine ls* es el 2376 y este es el puerto del demonio Docker. Se puede introducir con el puerto de manera correcta con el puerto 2377, que es el puerto asignado a la administración de enjambres, o no indicar puerto y que obtenga el predeterminado.
Se ha hecho uso de la siguiente sentencia para unir la otra máquina virtual como worker:

```
docker-machine ssh maquina-virtual-2 "docker swarm join --token SWMTKN-1-6ddle256cgl80sq4a7cbw6h2vsamgjy26nplqrc291ifdty1iq-8skbw5a3a548lbb3cnxn1jywe 192.168.99.111:2377"
```

Eso ha servido para que la segunda máquina-virtual-2 se una como worker, con el comando

```
docker node ls
```

En la máquina del administrador se puede verificar que los dos forman parte del mismo enjambre y comprobar cual es el administrador. Se debe recordar que estas sentencias solo pueden ser ejecutadas en nodos que sean administradores, en este caso se usa:

```
docker-machine ssh maquina-virtual-1 "docker node ls"
```

Se observa la demostración de como un worker no puede ejecutar estas sentencias en la siguiente imagen

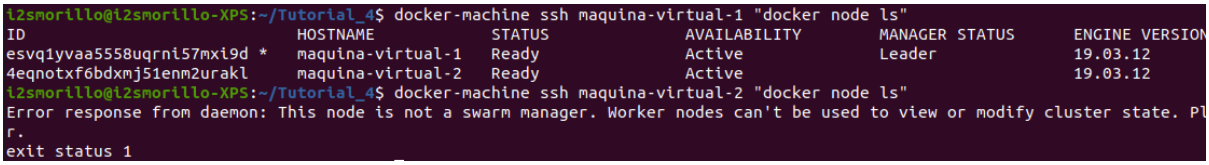

Figura 35: Listado de los nodos de un Docker Swarm

3.4.5 Ejecución de la aplicación en el Swarm con varios nodos

Se va a ejecutar la aplicación en Docker con contenedores distribuidos en los distintos nodos del enjambre, para ello debe ser el manager del enjambre quien lo ordene.

Se hace uso del docker-compose.yml configurado en el tutorial tres. Es necesario que la máquina, sea virtual o física, que contiene el nodo manager tenga acceso al archivo docker-compose.yml. En el caso de trabajar con una máquina virtual se puede enviar este archivo mediante scp, se puede usar de manera simple el comando docker-machine scp, o configurar con docker-machine la terminal local para poder usar las variables de entorno de la máquina virtual. Se ha escogido la segunda opción y se ejecuta un comando similar al siguiente:

```
eval $(docker-machine env maquina-virtual-1)
```

A continuación, se usa el comando que levanta el servicio, esto únicamente lo puede ordenar un nodo que tenga asignado el rol de manager:

```
docker stack deploy -c docker-compose.yml getstartedlab
```

En este momento el archivo docker-compose.yml tiene el contenido reflejado en el *código 7*.

```
i2smorillo@i2smorillo-XPS:~/Tutorial_4$ docker stack deploy -c docker-compose.yml getstartedlab
Creating network getstartedlab_webnet
Creating service getstartedlab_web
i2smorillo@i2smorillo-XPS:~/Tutorial_4$ docker stack ps getstartedlab
ID             NAME                  IMAGE                          NODE              DESIRED STATE   CURRENT STATE           ERROR     PORTS
vilnu1gikff8   getstartedlab_web.1   smorilloh/get-started:tutorial2   maquina-virtual-2   Running         Running 4 seconds ago
wdn2oxgyex5l   getstartedlab_web.2   smorilloh/get-started:tutorial2   maquina-virtual-1   Running         Running 5 seconds ago
w84ziej14uhd   getstartedlab_web.3   smorilloh/get-started:tutorial2   maquina-virtual-2   Running         Running 4 seconds ago
0sld512fbwrv   getstartedlab_web.4   smorilloh/get-started:tutorial2   maquina-virtual-2   Running         Running 4 seconds ago
4ym7jydi6av0   getstartedlab_web.5   smorilloh/get-started:tutorial2   maquina-virtual-1   Running         Running 5 seconds ago
```

Figura 36: Despliegue de los servicios en Docker Swarm distribuido en dos máquinas virtuales

Se observa que se han ejecutado todos los contenedores, están distribuidos entre los nodos del enjambre. Se puede ver que contenedores están en cada nodo con ayuda de docker-machine en cada máquina ejecutando el comando *docker ps* en las máquinas virtuales.

```
i2smorillo@i2smorillo-XPS:~/Tutorial_4$ docker-machine ssh maquina-virtual-1 "docker ps"
CONTAINER ID        IMAGE                             COMMAND             CREATED
96d0d8681ba2        smorilloh/get-started:tutorial2   "python app.py"     2 minutes ago
35bba1d084e0        smorilloh/get-started:tutorial2   "python app.py"     2 minutes ago
i2smorillo@i2smorillo-XPS:~/Tutorial_4$ docker-machine ssh maquina-virtual-2 "docker ps"
CONTAINER ID        IMAGE                             COMMAND             CREATED
058011cd9c4a        smorilloh/get-started:tutorial2   "python app.py"     2 minutes ago
be058b9b983d        smorilloh/get-started:tutorial2   "python app.py"     2 minutes ago
33f41e817c6c        smorilloh/get-started:tutorial2   "python app.py"     2 minutes ago
```

Figura 37: Distribución de los contenedores entre las máquinas virtuales

Como ya se ha visto en tutoriales anteriores se puede verificar el balanceo de carga y el comportamiento de los contenedores con un navegador web o la herramienta curl realizando búsquedas a la dirección IP de las máquinas virtuales con el puerto expuesto de manera exterior, se puede obtener la dirección IP como ya se ha mostrado con

```
docker-machine ls
```

En este caso se realiza un balanceo de carga round-robin entre los cinco contenedores, también se podría escalar el número de réplicas de igual modo que en el tutorial tres.

3.4.6 Eliminar las máquinas virtuales creadas con docker-machine

Si se decide eliminar los contenedores y abandonar el swarm se hace de igual modo que en el redactado en el tutorial tres. Para destruir las máquinas generadas con docker-machine se debe usar la sentencia:

```
docker-machine rm <nombre_maquina_virtual>
```

```
i2smorillo@i2smorillo-XPS:~/Tutorial_4$ docker-machine ls
NAME                ACTIVE   DRIVER       STATE     URL                         SWARM   DOCKER      ERRORS
maquina-virtual-1   -        virtualbox   Running   tcp://192.168.99.111:2376           v19.03.12
maquina-virtual-2   -        virtualbox   Running   tcp://192.168.99.112:2376           v19.03.12
i2smorillo@i2smorillo-XPS:~/Tutorial_4$ docker-machine rm maquina-virtual-1
About to remove maquina-virtual-1
Are you sure? (y/n): y
Successfully removed maquina-virtual-1
i2smorillo@i2smorillo-XPS:~/Tutorial_4$ docker-machine rm maquina-virtual-2
About to remove maquina-virtual-2
Are you sure? (y/n): y
Successfully removed maquina-virtual-2
i2smorillo@i2smorillo-XPS:~/Tutorial_4$ docker-machine ls
NAME   ACTIVE   DRIVER   STATE   URL   SWARM   DOCKER   ERRORS
i2smorillo@i2smorillo-XPS:~/Tutorial_4$
```

Figura 38: Eliminación de las máquinas virtuales con la herramienta docker-machine

3.5 Tutorial 5: Cooperación de varios servicios alojados en Docker Swarm

3.5.1 Stack con varios servicios

La parte que abarca esta sección de los tutoriales es la configuración de varios servicios bajo el mismo stack. Un stack es un grupo de servicios relacionados que comparten dependencias entre sí, además estos se pueden dirigir y escalar de manera conjunta. Un solo stack puede definir y coordinar una aplicación completa, aunque en los casos de aplicaciones más complejas esta orquestación puede darse a través de varios stacks.

La definición de estos stacks se realiza en docker-compose.yml, realmente se ha hecho uso de estos stacks en anteriores, pero con un único servicio. En este tutorial se definen varios servicios relacionados entre ellos.

3.5.2 Definición del archivo docker-compose.yml

Continuando a partir del código de tutoriales anteriores se define el siguiente docker-compose.yml:

```yaml
version: "3"
services:
  web:
    image: smorilloh/friendlyhello_correct:part2
    deploy:
      replicas: 5
      restart_policy:
        condition: on-failure
      resources:
        limits:
          cpus: "0.1"
          memory: 50M
    ports:
      - "80:80"
    networks:
      - webnet
  visualizer:
    image: dockersamples/visualizer:stable
    ports:
      - "8080:8080"
    volumes:
      - "/var/run/docker.sock:/var/run/docker.sock"
    deploy:
      placement:
        constraints: [node.role == manager]
    networks:
      - webnet
networks:
  webnet:
```

Código 9: Definición del docker-compose.yml con servicio web y visualizer

Respecto al docker-compose.yml visto anteriormente, agrega el servicio denominado visualizer.

Este servicio usa una imagen, *dockersamples/visualizer:stable*, que es accesible a todo el mundo en Docker Hub.

Se usa un volumen */var/run/docker.sock* , los volúmenes son el mecanismo predefinido para conservar los datos generados y que los contenedores Docker puedan hacer uso de ellos. Esto permite al contenedor almacenar datos de manera persistente. Estos datos se guardan en el host local que almacena el contenedor Docker, en este caso lo almacena en /var/run/docker.sock . Se hará un desarrollo más extendido de esta parte en otro tutorial.

En lo definido en deploy-placement nos ayuda a controlar la ubicación del servicio en los distintos nodos, en este ejemplo está obligando a que el servicio visualizer esté alojado en el nodo Swarm manager. Esto se debe a que es un servicio que hace muestra de los distintos servicios que están alojados en el Swarm y es el manager quien puede controlar y obtener información de todos los workers. Se puede hacer referencia de distintas formas al nodo, al

nombre del nodo o hostname o incluso a un nodo de una determinada región, si se utilizan debidamente las etiquetas, para definir donde ejecutar el servicio.

3.5.3 Ejecución de la aplicación en el Swarm distribuido con varios servicios en varios nodos

Para ejecutar varios nodos Swarm se hace uso de docker-machine, como se explica en el tutorial cuatro. Recordando que “maquina-virtual-1” es el nodo manager. Se configura la terminal local con las variables de entorno que dicha máquina con:

```
eval $(docker-machine env maquina-virtual-1)
```

Ahora en mismo directorio que el docker-compose.yml definido se ejecuta el siguiente comando para desplegar el stack:

```
docker stack deploy -c docker-compose.yml getstartedlab
```

Este comando crea una sola red, llamada webnet, que es compartida por los dos servicios definidos, llamados web y visualizer.

Se puede observar en el nodo manager que tiene dos contenedores para el servicio web y una réplica para el servicio visualizer que estaba obligado a alojarse en este nodo. También se observan los puertos expuestos de manera interna por los contenedores.

```
i2smorillo@i2smorillo-XPS:~/Tutorial_5$ docker stack ps getstartedlab
ID             NAME                          IMAGE                                NODE              DESIRED STATE
1aupm8rp2txv   getstartedlab_visualizer.1    dockersamples/visualizer:stable      maquina-virtual-1 Running
j5s7biacp3vf   getstartedlab_web.1           smorilloh/friendlyhello_correct:part2 maquina-virtual-2 Running
gixaiysc3o2h   getstartedlab_web.2           smorilloh/friendlyhello_correct:part2 maquina-virtual-1 Running
nxkv3rms3xdp   getstartedlab_web.3           smorilloh/friendlyhello_correct:part2 maquina-virtual-2 Running
wzdhv24cq2ak   getstartedlab_web.4           smorilloh/friendlyhello_correct:part2 maquina-virtual-1 Running
h67oqx72pdxa   getstartedlab_web.5           smorilloh/friendlyhello_correct:part2 maquina-virtual-2 Running
i2smorillo@i2smorillo-XPS:~/Tutorial_5$ docker ps
CONTAINER ID   IMAGE                                COMMAND           CREATED         STATUS         PORTS      NA
392c912be061   dockersamples/visualizer:stable      "npm start"       5 minutes ago   Up 5 minutes   8080/tcp   ge
6afdc577b878   smorilloh/friendlyhello_correct:part2 "python app.py"  5 minutes ago   Up 5 minutes   80/tcp     ge
f205471f930e   smorilloh/friendlyhello_correct:part2 "python app.py"  5 minutes ago   Up 5 minutes   80/tcp     ge
```

Figura 39: Distribución de los contenedores en ejecución del Swarm

Si se realiza una búsqueda a través de un navegador web a dirección con la dirección IP de alguna de las máquinas virtuales y puerto 8080, se observa el nuevo servicio configurado visualizer.

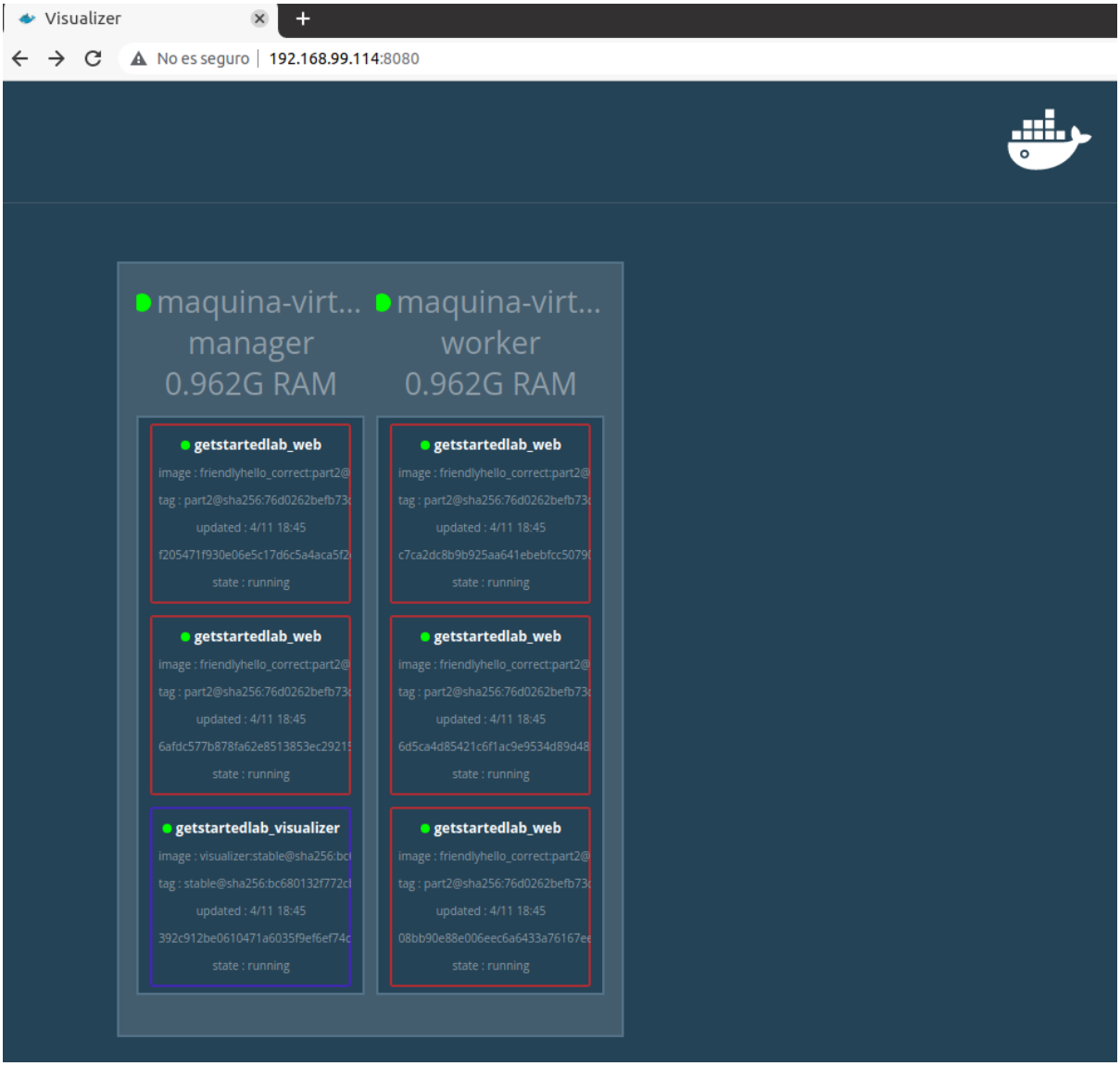

Figura 40: Interfaz gráfica de control que ofrece el servicio visualizer

Muestra información de todos los contenedores alojados en los distintos nodos del Swarm de una manera más visual que mediante la terminal.

3.5.4 Datos persistentes para su uso en un servicio por ejemplo Redis

En este apartado se hace uso de datos persistentes, para ello se ejemplifica con el uso de Redis que necesita que la información que maneja sea almacenada. Para incluir Redis entre los servicios se debe modificar docker-compose.yml, se hace uso de la siguiente configuración:

```yaml
version: "3"
services:
  web:
    image: smorilloh/friendlyhello_correct:part2
    deploy:
      replicas: 5
      restart_policy:
        condition: on-failure
      resources:
        limits:
          cpus: "0.1"
          memory: 50M
    ports:
      - "80:80"
    networks:
      - webnet
  visualizer:
    image: dockersamples/visualizer:stable
    ports:
      - "8080:8080"
    volumes:
      - "/var/run/docker.sock:/var/run/docker.sock"
    deploy:
      placement:
        constraints: [node.role == manager]
    networks:
      - webnet
  redis:
    image: redis
    ports:
      - "6379:6379"
    volumes:
      - "/home/docker/data:/data"
    deploy:
      placement:
        constraints: [node.role == manager]
    command: redis-server --appendonly yes
    networks:
      - webnet
networks:
  webnet:
```

Código 10: Definición del docker-compose.yml con servicio web, visualizer y redis

Sobre la parte añadida respecto al *código 9*, se añade el servicio de Redis a partir de una imagen de Redis oficial alojada en Docker Hub; es un servicio de uso frecuente por eso se le ha concedido un alias con denominación tan simple y así poder acceder de manera tan sencilla a la imagen almacenada en el repositorio Docker Hub.
El puerto preconfigurado para Redis es el 6379, esto nos obliga a que el puerto interior del contenedor sea 6379 pero el exterior puede ser cualquiera.
El servicio Redis accede a un directorio del sistema de archivos del host que aloja el contenedor, esto es contrario al aislamiento del contenedor, y proporciona esos datos dentro

del contenedor en el directorio */data*. Además, el comando que se solicita que se ejecute permite que el Redis haga uso de los datos y los modifique de forma que a cada reinicio pueda continuar usando estos. Es decir, el servicio redis definido necesita para su correcto funcionamiento datos persistentes para su uso. Por último, al igual que los otros servicios hace uso de la red webnet y de este modo puede tener conexión directa con los otros contenedores.

Se hace un despliegue del nuevo docker-compose.yml para que se inicie el servicio de redis. Se hace desde el nodo master que es el cual tiene autoridad para iniciar servicios.

```
i2smorillo@i2smorillo-XPS:~/Tutorial_5$ docker-machine ls
NAME                ACTIVE   DRIVER       STATE     URL                         SWARM   DOCKER
maquina-virtual-1   *        virtualbox   Running   tcp://192.168.99.116:2376           v19.03.12
maquina-virtual-2   -        virtualbox   Running   tcp://192.168.99.117:2376           v19.03.12
i2smorillo@i2smorillo-XPS:~/Tutorial_5$ docker stack deploy -c docker-compose.yml getstartedlab
Creating network getstartedlab_webnet
Creating service getstartedlab_visualizer
Creating service getstartedlab_redis
Creating service getstartedlab_web
```

Figura 41: Redespliegue de los servicios incluyendo el servicio redis

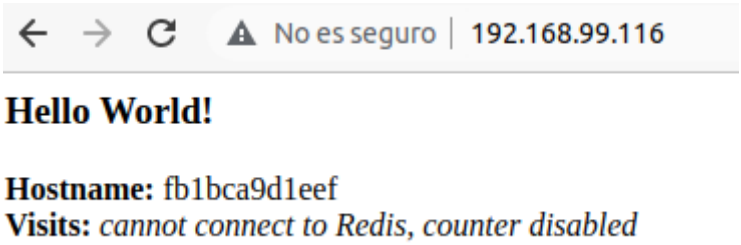

Figura 42: Servicio web con despliegue de redis incompleto

Tras crear los servicios aún no está disponible la función del Redis porque está intentando hacer uso del directorio *data* y este aún no existe, se debe crear en el host que está alojado el contenedor de redis. Observando el código se refleja que el Redis estará alojado en el nodo master, por ello creamos el directorio de la siguiente manera:

```
docker-machine ssh maquina-virtual-1 "mkdir ./data"
```

La configuración completa y las opciones que ofrecen los volumenes de Docker se reflejan en el capítulo “3.6 Uso de volúmenes de almacenamiento en Docker”. En el capitulo actual se muestra como los datos permanecen de manera persistente aunque se detenga el contenedor que hace uso del volumen creado.

Ahora se puede ver reflejado el uso del Redis que muestra información en la búsqueda web, está configurado en este caso para contabilizar las visitas a la web.

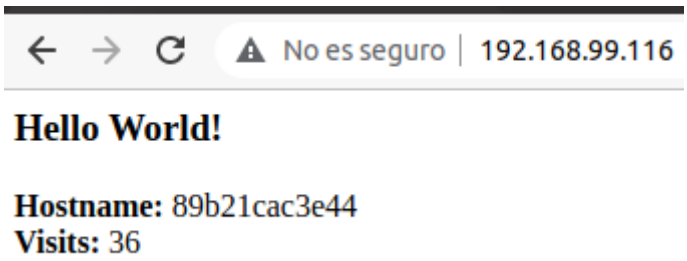

Figura 43: Servicio web con el servicio redis en ejecución completa

También se observa su funcionamiento y datos a través del servicio visualizer:

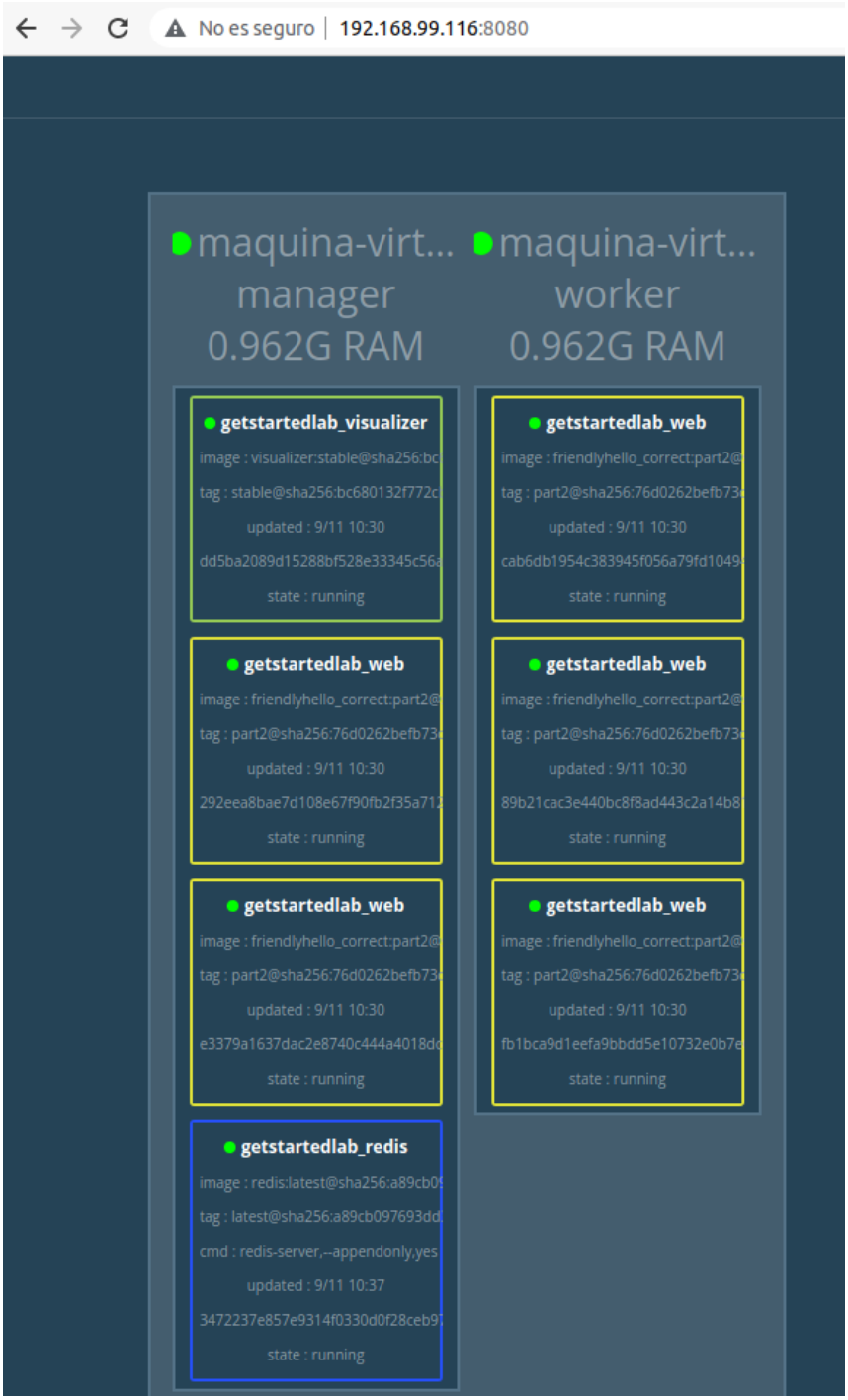

Figura 44: Interfaz gráfica de control que ofrece el servicio visualizer con el nuevo servicio redis

Para comprobar la persistencia de los datos que accede el Redis se va a destruir su contenedor, tras ello se vuelve a levantar el servicio y necesita hacer uso de los datos almacenados anteriormente. Debemos destruir su contenedor desde el terminal de maquina-virtual-1 que es el master y tiene los permisos necesarios para detener un contenedor con la ejecución de la sentencia:

```
docker stop <ID-Contenedor-Redis>
```

```
i2smorillo@i2smorillo-XPS:~/Tutorial_5$ docker container ls
CONTAINER ID   IMAGE                                   COMMAND
3472237e857e   redis:latest                            "docker-entrypoint.s…"
e3379a1637da   smorilloh/friendlyhello_correct:part2   "python app.py"
292eea8bae7d   smorilloh/friendlyhello_correct:part2   "python app.py"
dd5ba2089d15   dockersamples/visualizer:stable         "npm start"
i2smorillo@i2smorillo-XPS:~/Tutorial_5$ docker stop 3472237e857e
3472237e857e
```

Figura 45: Detención del contenedor que aloja la imagen redis

Una vez se ha vuelto a iniciar el contenedor de servicio redis, realizando búsquedas con el navegador se comprueba que el Redis continua con los datos anteriormente almacenados.

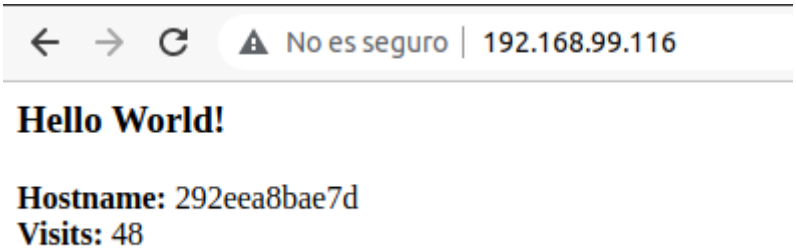

Figura 46: Muestra vía navegador web de la persistencia de los datos del volumen de redis

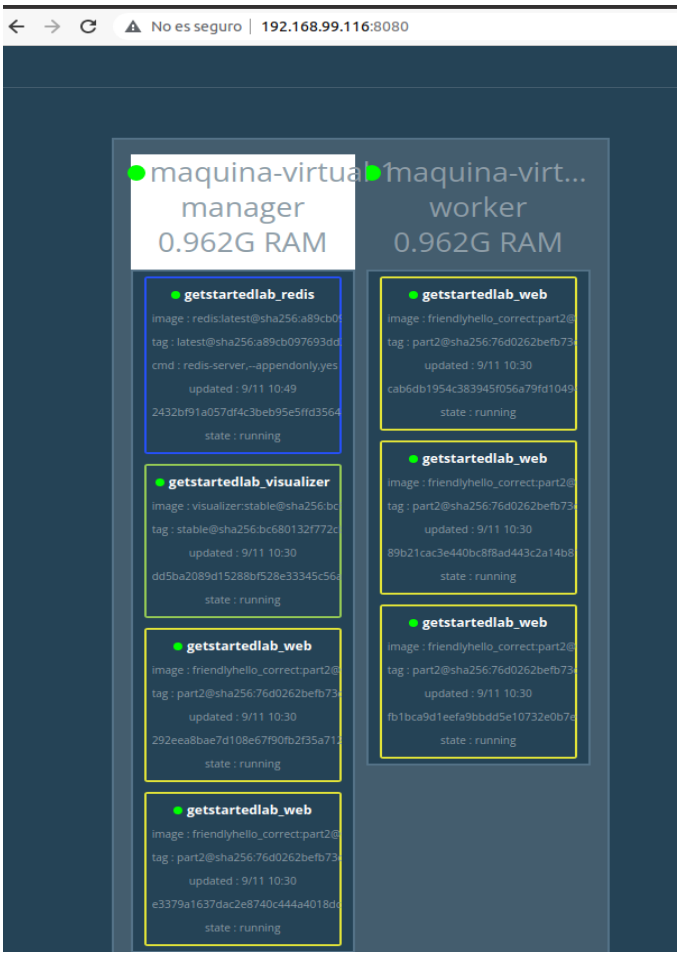

Figura 47: Interfaz gráfica de control que ofrece el servicio visualizer

Los datos persistentes es una opción necesaria en la mayoría de los servicios hoy en día configurados, sin esta opción Docker no tendría el valor que tiene y uso frecuente entre desarrolladores.

Una vez realizadas las comprobaciones de que efectivamente se almacenan los datos de manera correcta con persistencia se debe eliminar las máquinas virtuales como anteriormente se ha presentado.

3.6 Tutorial 6: Uso de volúmenes de almacenamiento en Docker

Los volúmenes de almacenamiento que utiliza un contenedor o sistemas de ficheros, como se ha visto son una parte básica de composición de un contenedor Docker con el parámetro “volumenes”. No siempre es necesario, pero en la mayoría de los casos si es interesante o necesario hacer uso de ello, ya que la mayor parte de las aplicaciones hacen uso de datos persistentes y deben alojar estos de algún modo.

Para definir qué tipo de volumen usar existen tres opciones:

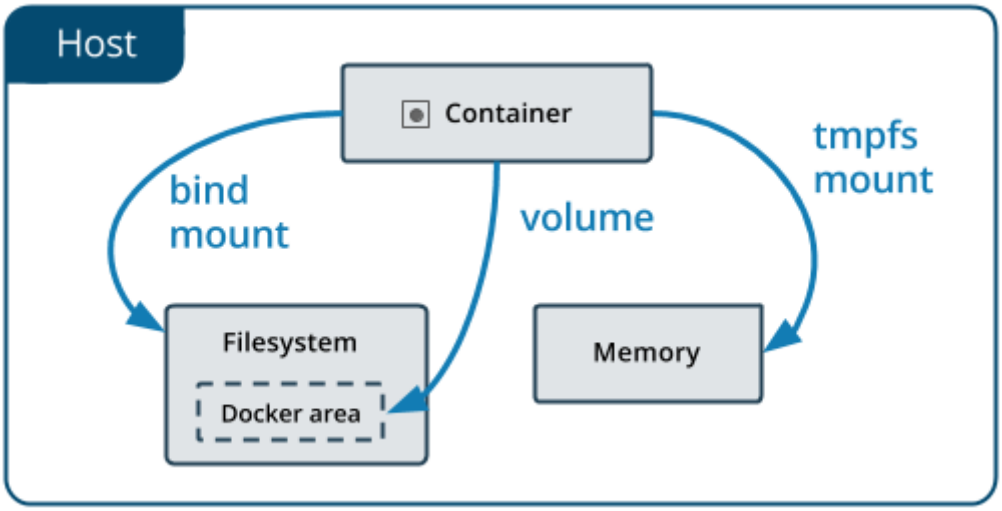

Figura 48: Distintos tipos de volúmenes de almacenamiento Docker [22]

Por defecto si no se indica lo contrario los datos son gestionados con el modelo TMPFS, abreviatura de temporal file system, y como indica su nombre mantiene los datos persistentes durante el tiempo de ejecución del propio contenedor. Una vez el contenedor finaliza su ejecución por cualquier motivo los datos dejan de estar disponibles destruyéndose, por ello si los datos que se manejan tienen necesidad de ser usados más allá de este ciclo de vida se debe hacer uso de los modelos bind mount o volumes. Para usar bind mount o volumes hay que indicarlo con un comando o en el docker-compose.yaml como veremos después.
En los siguientes ejemplos se hace una demostración de la persistencia de los datos en los modelos bind mount y volumes.

3.6.1 Volume

Los volúmenes son parte del sistema de archivos del host nativo en el cual está desplegado el contenedor. El contenedor Docker tiene acceso a esa parte del sistema de archivos y puede almacenar datos; esta manera permite que esos datos sean persistentes ya que si el contenedor se detiene o se elimina los datos no desaparecen con el mismo. A pesar de que estos datos están almacenados en el sistema de archivos local se mantiene el aislamiento, están únicamente administrados por Docker. Se puede montar un mismo volumen en varios contenedores a la vez y que estos compartan los archivos.

Las opciones que nos permite configurar los volumes son:

Opciones volume	Descripción de las opciones
`source o src`	Para volúmenes con nombre, este es el nombre del volumen. Para volúmenes anónimos, este campo se omite.
`destination, dst o target`	Ruta del directorio o archivo montado en el contenedor.
`readonly o ro`	Si se especifica, fija el volume montado con permisos de solo lectura
`volume-opt`	Se puede especificar más de una vez, sirve para asignar clave-valor, consta del nombre de la opción a determinar y su valor.

Figura 49: Tabla de opciones de los volumes [23]

La opción volume-opt puede ser utilizada para indicar parámetros de montaje del volumen como por ejemplo tipo, tamaño, uid u otros parámetros. También pueden utilizarse algunos clave-valor que no modifiquen su comportamiento, podría ser un tipo de etiquetado o comentarios.

Los volúmenes anónimos crean un volume sin especificar el *source*, Docker lo crea con un identificador único y se comporta como un volume con *source*. Docker desaconseja su uso ya que al no corresponder con un *source* los datos almacenados no se tiene referencia a ellos y una vez se detiene el contenedor que creó este volume desaparecen.

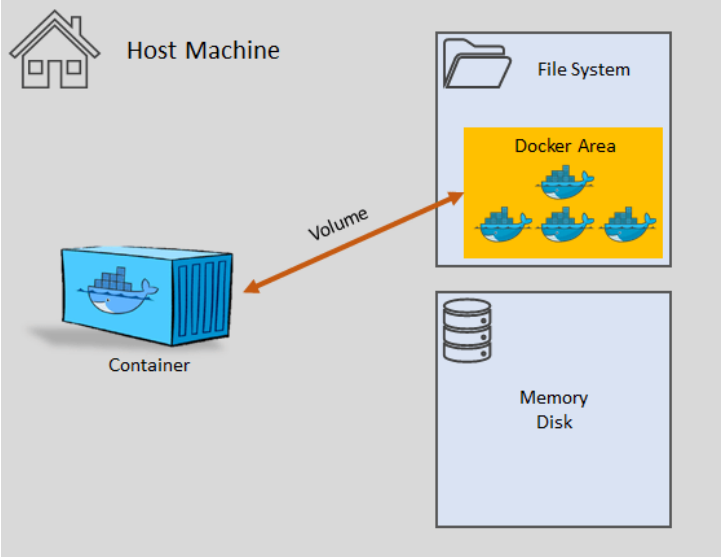

Figura 50: Esquema de uso de los volumes [24]

3.6.2 Bind Mount

Permite almacenar datos de manera persistente del contenedor Docker en el sistema de archivos local del host que aloja al mismo, al igual que los volumes. A diferencia de los volumes los archivos almacenados en esta ruta construida con bind pueden ser gestionados tanto por Docker como por el host que almacena el sistema de archivos local. Si el archivo o carpeta definida con bind mount sufre algún cambio desde cualquier punto, tanto por Docker como el sistema de archivos, se verá reflejado en el otro punto.

Bind mount, como en el caso de los volumes es utilizado cuando es necesario almacenar o hacer uso de los datos fuera del tiempo del ciclo de vida de cada contenedor Docker.

Las opciones que nos brindan los montajes bind son:

Opciones bind mounts	Descripción de las opciones
`source o src`	Para volúmenes con nombre, este es el nombre del volumen. Para volúmenes anónimos, este campo se omite.
`destination, dst o target`	Ruta del directorio o archivo montado en el contenedor.
`readonly o ro`	Si se especifica, se fijan los permisos de solo lectura
`bind-propagation`	Si se especifica modifica el modo de propagación. Las diferentes opciones de este campo se detallan en la *figura 52.*
`selinux`	Existen las opciones *z* o *Z*. La opción *z* indicada en minúscula indica que el contenido del montaje se comparte entre varios contenedores. La opción *Z* indica que el contenido del montaje de enlace es privado y no se comparte.

Figura 51: Tabla de opciones de los bind mounts [25]

Las opciones selinux y readonly son ignoradas cuando se usan los montajes como parte de un servicio.

La opción de propagación de un montaje bind mount es como los cambios efectuados en el volumen de almacenamiento repercute en los contenedores y en el sistema de archivos del host que los aloja.

Para los volumes y montajes bind la opción de propagación por defecto es *rprivate*. Los otros modos de propagación pueden ser configurados únicamente cuando se hace uso de un montaje bind y el sistema operativo del host que aloja el contenedor es Linux. La documentación de Docker destaca que modificar el modo de propagación del montaje bind es complejo y avanzado, en la mayoría de los casos no es necesario modificar este parámetro. Las diferentes opciones son:

Opciones bind-propagation	Descripción de las opciones
`shared`	Los sub-montajes del montaje original se exponen a los montajes de réplica, y los sub-montajes de los montajes de réplica también se propagan al montaje original.
`slave`	Si el montaje original expone un sub-montaje, el montaje réplica se puede verlo. Sin embargo, si el montaje réplica expone un sub-montaje, el montaje original no puede verlo.
`private`	Los sub-montajes son privados en ambas direcciones. Los sub-montajes en el montaje original no están expuestos a los montajes de réplica, y los sub-montajes de los montajes de réplica no están expuestos al montaje original.
`rshared`	Igual que en la opción *shared*, pero la propagación también se extiende desde y hacia los puntos de montaje anidados dentro de cualquiera de los puntos de montaje originales o réplicas. Es decir, la r inicial implica recursividad.
`rslave`	Igual que en la opción *slave*, pero la propagación también se extiende hacia y desde los puntos de montaje anidados dentro de cualquiera de los puntos de montaje originales o réplicas. Es decir, la r inicial implica recursividad.
`rprivate`	Igual que en la opción *private*, lo que significa que ningún punto de montaje en ningún lugar dentro de los puntos de montaje original o réplica se propaga en ninguna dirección.

Figura 52: Tabla de los distintos modos de propagación de los bind mounts [25]

Se hace referencia al montaje original como el sistema de archivo local del host y los montajes réplicas son ese mismo montaje construido dentro del contenedor Docker.

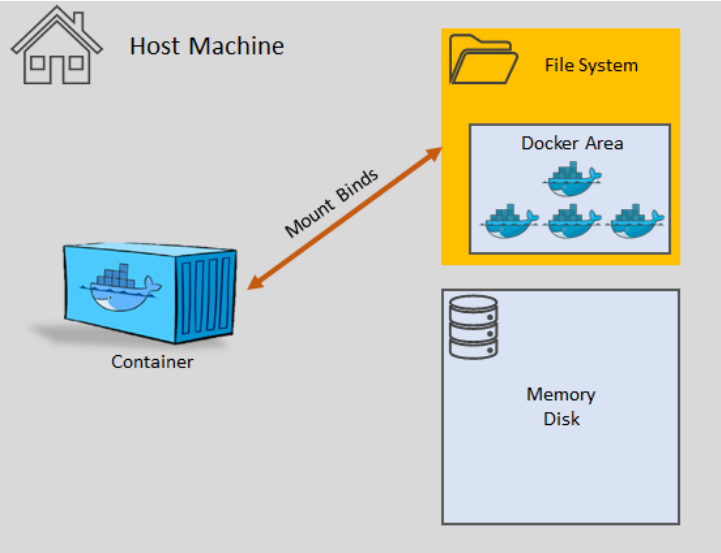

Figura 53: Esquema de uso de los bind mount [24]

3.6.3 TMPFS

Es una opción que permite la construcción de un sistema de archivos temporales, una vez se detiene el contenedor el montaje se elimina y los datos almacenados en el mismo desaparecen. A diferencia de la opción volume y bind, no puede compartir montajes tmpfs entre contenedores.

Una limitación a tener en cuenta es que el montaje tmpfs solo está disponible si el contenedor está corriendo sobre un sistema operativo Linux.

Existen únicamente dos opciones de configuración en los montajes tmpfs, estas son opcionales:

Opciones tmpfs	Descripción de las opciones
`tmpfs-size`	Tamaño del montaje tmpfs, por defecto su valor es ilimitado.
`tmpfs-mode`	Permisos en los directorios del montaje tmpfs recopilados con valor octal, por defecto el valor es 1777.

Figura 54: Tabla de opciones de los TMPFS [26]

Nota: El valor de permisos 1777 brinda permisos a absolutos a todos los usuarios, pero previene que el directorio y su contenido sea borrado por un usuario distinto al *owner*.

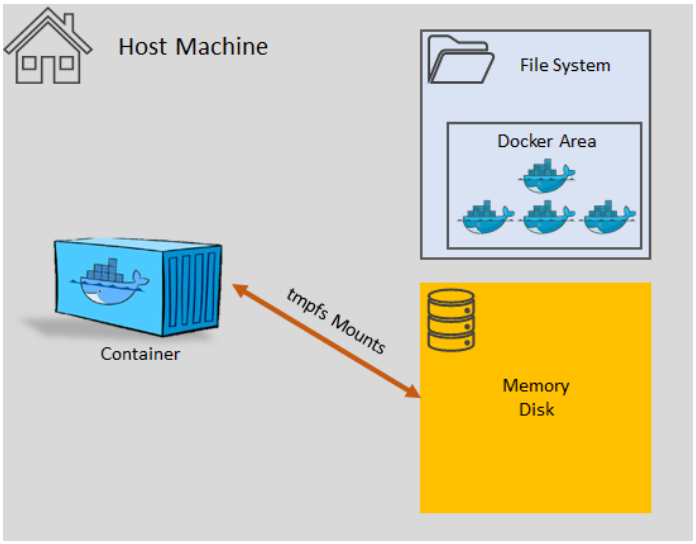

Figura 55: Esquema de uso de TMPFS [24]

3.6.4 Comparación entre los tipos de volumenes

Las comparaciones existen sobre todo entre estos los volumes y bind mounts debido a sus similitudes.

	TMPFS	Volumes	Bind Mount
Como indicar el uso de cada tipo	Por defecto	Docker CLI o docker-compose	Docker CLI o docker-compose
Persistencia de los datos fuera del ciclo de vida del contenedor	NO	SI	SI
Compartir sistema de datos	NO	SI	Si
Gestionado por parte de:	Docker	Docker	Docker y sistema de archivos local del host que lo aloja.
Externalizar del host el almacenamiento de datos:	NO	SI	NO
Opciones configuración	Muy reducido	Amplio	Muy amplio

Figura 56: Tabla comparativa TMPFS-Volumes-Bind Mounts

En cuanto a las opciones de configuración son muy distintas entre ellos, por eso conviene observar las tablas anteriores que se analiza individualmente cada opción.

En relación con esta comparativa es la propia documentación oficial de Docker la que ofrece una serie de ventajas para el uso de volumes respecto al de bind mounts [23]:

- Los volumes son más fáciles de gestionar que los bind mount por parte de los contenedores Docker.

- Se permite gestionar volumes con comandos de Docker CLI o a través de APIs Docker.

- Los volumes funcionan sin excepciones en contenedores alojados sobre host con sistemas operativos Linux y Windows.

- Los volumes se pueden compartir de manera más segura y eficiente entre contenedores Docker.

- Los controladores de volumes le permiten almacenar volúmenes en hosts remotos o proveedores en la nube, esta opción es extremadamente interesante para proteger tus datos incluso no depender de limitaciones de tamaño del host que aloja el contenedor Docker.

- Los nuevos volumes pueden tener su contenido precargado por un contenedor.

- El aumento de tamaño de un volume no aumenta el tamaño del contenedor que hace uso de este.

3.6.5 Demostración de la persistencia de los volumes

Se va a hacer uso de un Dockerfile como el siguiente, para crear una imagen de Docker con la cual hacer demostraciones sobre los volumes y montajes bind.

```
# Specify a base image
FROM alpine
#Install bash
RUN apk add --no-cache bash
#Run next commands
ENTRYPOINT ["sleep"]
CMD ["2500"]
```

Código 11: Dockerfile base para la imagen bash-examples

En este caso usamos imagen base de alpine, se le instala bash y usamos entrypoint para ejecutar un comando al inicio de la ejecución de la imagen en un contenedor y poder controlar que no se cierre inmediatamente.
Como se mencionaba en apartados anteriores, la instrucción RUN ejecuta el comando que le acompaña, en este caso se usa para instalar en bash que es necesario su uso para completar el ejemplo.
Se separa en dos pasos ENTRYPOINT y CMD el comando de entrada para que sea posible modificar el parámetro de cuánto tiempo ejecutar sleep evitando que finalice la ejecución de la imagen generada, en caso de desear iniciar el contenedor Docker con una sentencia que modifique ese valor de segundos “2500” que ejecutará el comando sleep, la estructura del comando sería:

```
docker run -it <image-name> <tiempo-sleep>
```
En caso contrario tomará como valor por defecto los 2500 segundos.

Se crea una imagen llamada bash-examples con el comando:
```
docker build -t bash-examples .
```
Se le hace referencia a dicha imagen en el docker-compose.yml que se muestra a continuación:

```
version: "3"
services:
  bash:
    image: bash-examples
    deploy:
      resources:
        limits:
          cpus: "0.1"
          memory: 50M
    volumes:
      - "volume-1:/var/example"
volumes:
  volume-1:
```
Código 12: Archivo docker-compose.yml con la imagen bash-example y uso de un volume

En este caso se crea un volume con el nombre “volume-1” que se encuentra dentro del contenedor en el path */var/example* . Para corroborar el funcionamiento de los volumes se comienza levantando un contenedor con el servicio bash con la sentencia:
```
docker-compose up
```
Una vez levantado el servicio se puede ejecutar la sentencia:
```
docker volume ls
```
Esto muestra el volume creado. Tras ello se accede al contendor con el comando:
```
docker exec -it <CONTAINER-ID> bash
```

```
i2smorillo@i2smorillo-XPS:~/Tutorial_6$ docker volume ls
DRIVER    VOLUME NAME
local     tutorial_6_volume-1
i2smorillo@i2smorillo-XPS:~/Tutorial_6$ docker container ls
CONTAINER ID   IMAGE          COMMAND        CREATED              STATUS              PORTS     NAMES
43102d3e2c01   bash-examples  "sleep 2500"   About a minute ago   Up About a minute             tutorial_6_bash_1
i2smorillo@i2smorillo-XPS:~/Tutorial_6$ docker exec -it 43102d3e2c01 bash
bash-5.1# cd /var/
cache/   empty/   example/ lib/     local/   lock/    log/     mail/    opt/     run/     spool/   tmp/
bash-5.1# cd /var/example/
bash-5.1# ls
bash-5.1#
```
Figura 57: Muestra del volume-1 y su montaje inicial

Se comprueba que el directorio */var/example* existe y está vacío. Se crea un archivo a modo de ejemplo, por ejemplo, ejecutando:
```
cat "Hello world!" > saludo
```
Tras crear el archivo de ejemplo eliminamos el servicio levantado con:
```
docker-compose rm bash-examples
```
Tras eliminarlo para comprobar que una vez se reinicia el contenedor tiene acceso a los datos volvemos a recrear el servicio con la sentencia:
```
docker-compose up
```

Se muestra exactamente el mismo contenido que fue creado, se demuestra así la persistencia de los datos haciendo uso de volumes.

```
i2smorillo@i2smorillo-XPS:~/Tutorial_6$ docker container ls
CONTAINER ID   IMAGE           COMMAND        CREATED          STATUS          PORTS     NAMES
d89fc17aa505   bash-examples   "sleep 2500"   46 seconds ago   Up 45 seconds             tutorial_6_bash_1
i2smorillo@i2smorillo-XPS:~/Tutorial_6$ docker exec -it d89fc17aa505 bash
bash-5.1# cd /var/example
bash-5.1# cat saludo
Hello World!
```

Figura 58: Uso del volume-1 para almacenar un archivo persistente

Se puede modificar el path donde montar el volume dentro del contendor y éste sigue persistiendo los mismos datos, esto es posible debido a que los datos que están disponibles lo hacen en función del nombre del volume. Por ello el nombre de un volumen en un mismo host debe de ser único.

Los volúmenes pueden ser eliminados una vez no está ningún contenedor haciendo uso de este con:

```
docker volume rm <nombre-volume>
```

3.6.6 Demostración de la persistencia de los bind mounts

Esta demostración usa la misma imagen de Docker que en ejemplo anterior, simplemente existe una variación en docker-compose.yml que nos permite crear el bind mount:

```yaml
version: "3"
services:
  bash:
    image: bash-examples
    deploy:
      resources:
        limits:
          cpus: "0.1"
          memory: 50M
    volumes:
      - "/var/data-docker:/var/example"
```

Código 13: Archivo docker-compose.yml con la imagen bash-example y uso de un bind mount

El directorio compartido por el host donde se aloja el contenedor y el directorio que posee el propio contenedor se sitúa en rutas absolutas diferentes. En el host se encuentra este directorio en el path */var/data-docker* , mientras que en el interior del contenedor este mismo se monta en el path */var/example* . Se levanta el servicio para hacer las pertinentes pruebas.

Se accede al contenedor con el comando bash y se verifica que inicialmente el directorio compartido está vacío, tanto en el contenedor como en el host. De igual manera que en el ejemplo anterior se crea un archivo llamado saludo.

```
i2smorillo@i2smorillo-XPS:/var/data-docker$ ls
i2smorillo@i2smorillo-XPS:/var/data-docker$ docker exec -it 497997df26b7 bash
bash-5.1# cd /var/example/
bash-5.1# ls
bash-5.1# echo Hello world! > saludo
bash-5.1# cat saludo
Hello world!
bash-5.1# exit
exit
i2smorillo@i2smorillo-XPS:/var/data-docker$ ls
saludo
i2smorillo@i2smorillo-XPS:/var/data-docker$ cat saludo
Hello world!
```

Figura 59: Muestra de funcionamiento de los bind mount

Se observa como un archivo creado en el interior del contenedor tenemos acceso al mismo desde el host en el path indicado para el montaje bind. Para verificar que esto es bidireccional se va a modificar un archivo en el host y este también se modificará en el contenedor. Incluso si se elimina esto se produce en ambas localizaciones.

```
root@i2smorillo-XPS:/var/data-docker# ls
saludo
root@i2smorillo-XPS:/var/data-docker# cat saludo
Hello World!
root@i2smorillo-XPS:/var/data-docker# echo Bye! >> saludo
root@i2smorillo-XPS:/var/data-docker# cat saludo
Hello World!
Bye!
root@i2smorillo-XPS:/var/data-docker# docker exec -it e70eb780f9fa bash
bash-5.1# cd /var/example/
bash-5.1# cat saludo
Hello World!
Bye!
bash-5.1# rm saludo
bash-5.1# ls
bash-5.1# exit
exit
root@i2smorillo-XPS:/var/data-docker# ls
root@i2smorillo-XPS:/var/data-docker#
```

Figura 60: Persistencia y modo por defecto de los bind mounts

Nota: Esta parte se ha ejecutado como root por los permisos necesarios para modificar archivos en los directorios */var*

3.7 Tutorial 7: Comunicaciones dentro Docker de un host

Esta sección pretende demostrar de que manera hacen uso de la red creada por Docker los contenedores generados de manera independiente en un mismo host. La imagen expuesta a continuación muestra como es esta red mencionada:

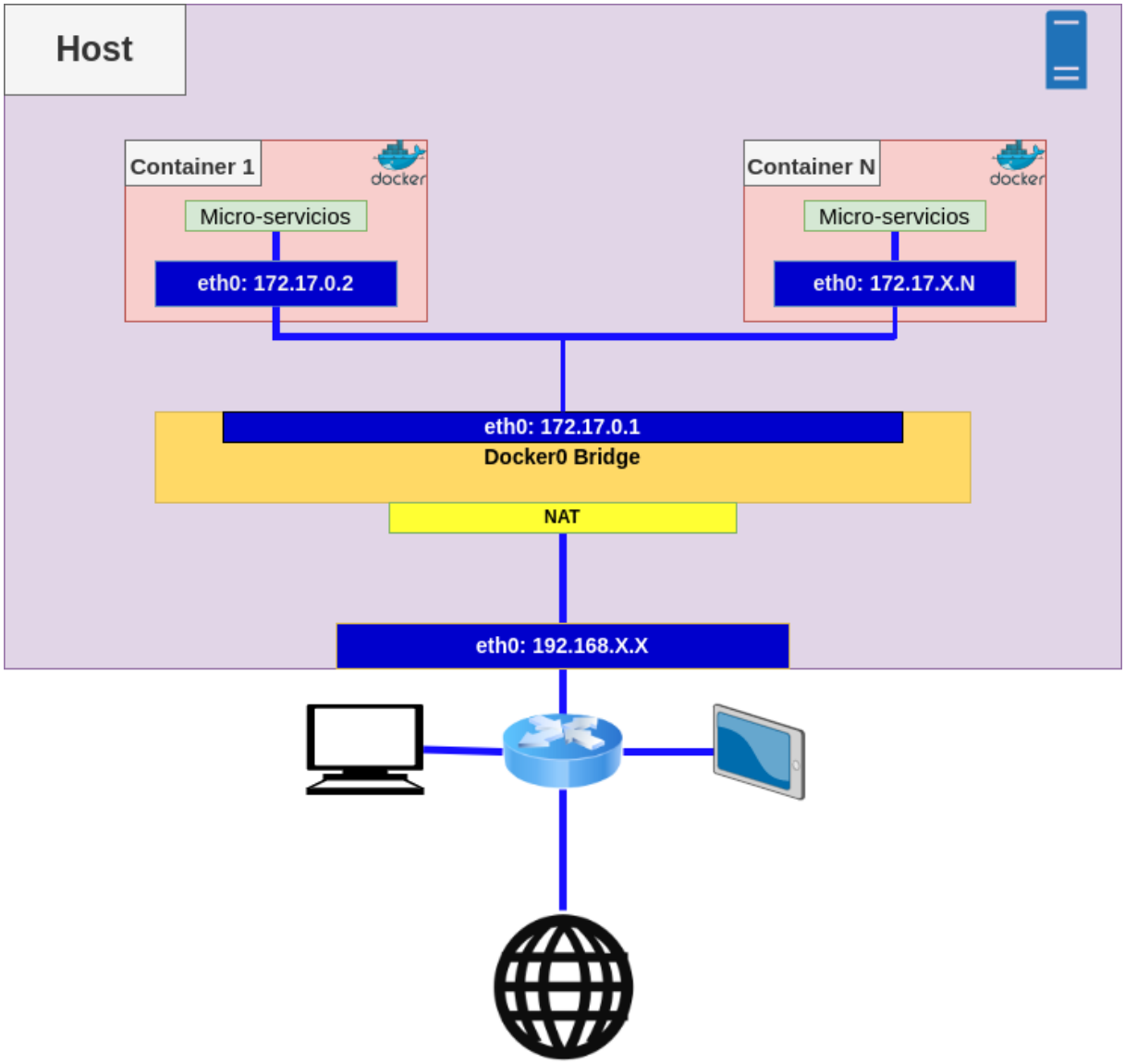

Figura 61: Esquema de la organización de la red docker0

En este ejemplo se observa que la subred creada para Docker es la red 172.17.0.0/16, para verificar el uso de dicha red y los puertos expuestos de manera interna/externa se realizan las siguientes pruebas de este apartado.

3.7.1 Modificaciones en Dockerfile para instalar paquetes

Se modifica el Dockerfile del tutorial dos para así instalar el paquete iproute2, esto permite que durante las demostraciones se pueda mostrar la dirección IP asignada al contenedor. También se instala el paquete net-tools para hacer muestra de los puertos de escucha del mismo.

El Dockerfile modificado es el siguiente:

```
# Specify a base image
FROM python:2.7-slim

# Set the working directory
WORKDIR /usr/app

# Copy my local directory into the container /app
COPY . /usr/app

# Install any needed packages specified in requirements.txt
RUN pip install --trusted-host pypi.python.org -r requirements.txt

# Install iproute-2 and net-tools
RUN apt-get update && apt-get install -y \
    iproute2 \
    net-tools \
 && rm -rf /var/lib/apt/lists/*


# Expose the container port 80
EXPOSE 80

# Run app.py when the container launches
CMD ["python","app.py"]
```

Código 14: Dockerfile para la creación de la imagen demo_addr

Creamos una imagen local con el comando:

```
docker build -t demo_addr .
```

Esta imagen se usará para crear un contenedor que nos muestra las llegadas de peticiones de otros contenedores.

3.7.2 Comprobación de direcciones IPv4 y puertos asignados

Para poder realizar las comprobaciones necesarias se muestra un terminal dentro del contenedor que está corriendo con el comando:

```
docker run -p 4000:80 demo_addr
```

Se puede comprobar que se encuentra funcionando correctamente realizando una búsqueda con alguno de los distintos navegadores web a la dirección 0.0.0.0:4000. Esta búsqueda muestra la siguiente web ya que se ha modificado el archivo app.py.

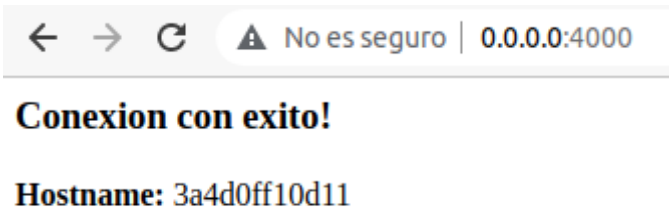

Figura 62: Muestra con un navegador del servicio web ofrecido por el contenedor demo_addr

Tras realizar esta búsqueda se observa como la petición recibida en el contenedor tiene la dirección fuente del docker bridge, esto se produce porque las peticiones pasan a través del mismo para balancearlas a los contenedores.

```
i2smorillo@i2smorillo-XPS:~/Tutorial_7$ docker run -p 4000:80 demo_addr
 * Serving Flask app "app" (lazy loading)
 * Environment: production
   WARNING: This is a development server. Do not use it in a production deployment.
   Use a production WSGI server instead.
 * Debug mode: off
 * Running on http://0.0.0.0:80/ (Press CTRL+C to quit)
172.17.0.1 - - [24/Nov/2021 09:02:29] "GET / HTTP/1.1" 200 -
```

Figura 63: Ejecución del contenedor demo_addr

Tras ello se muestran las estadísticas del contenedor con:

```
docker stats
```

Es una manera de recopilar información sobre la carga que está asimilando el contenedor en comparación a sus límites, además nos sirve para descubrir el ID asociado al contenedor.

```
CONTAINER ID   NAME            CPU %     MEM USAGE / LIMIT     MEM %     NET I/O         BLOCK I/O   PIDS
3a4d0ff10d11   sweet_davinci   0.03%     17.61MiB / 15.33GiB   0.11%     16kB / 3.76kB   0B / 0B     1
^C
i2smorillo@i2smorillo-XPS:~/Tutorial_7$ docker exec -it 3a4d0ff10d11 sh
# ip -c a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
164: eth0@if165: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default
    link/ether 02:42:ac:11:00:02 brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 172.17.0.2/16 brd 172.17.255.255 scope global eth0
       valid_lft forever preferred_lft forever
# netstat -putna
Active Internet connections (servers and established)
Proto Recv-Q Send-Q Local Address           Foreign Address         State       PID/Program name
tcp        0      0 0.0.0.0:80              0.0.0.0:*               LISTEN      1/python
#
```

Figura 64: Dirección IPv4 y puertos de escucha en el contenedor demo_addr en ejecución

Se accede a una terminal del contenedor y con la instrucción:

```
ip -c a
```

Se muestran las direcciones IP asignadas. Se encuentra la dirección de loopback (127.0.0.1) y la asignada a eth0 es la dirección IPv4 172.17.0.2. Esa dirección se encuentra en la subred docker0 creada en el host, por tanto, sólo es accesible con la dirección IPv4 172.17.0.2 desde el interior de la red docker0.

Con el comando netstat se comprueba que el puerto 80 está a la escucha para poder servir la web programada. No se refleja el puerto 4000 porque este puerto no es el contenedor el que lo expone.

3.7.3 Peticiones desde otro contenedor dirigidas a docker0 bridge

En esta sección se realizarán peticiones a la dirección IP del docker0 bridge para analizar su resultado. Se hace uso de una imagen alojada en Docker hub con el nombre asignado curlimages/curl y la versión con tag 7.79.1 : https://hub.docker.com/r/curlimages/curl
Se puede hacer uso de otras imágenes mientras estas tengan instalada la paquetería necesaria para realizar las pruebas, con el objetivo de mostrar las ventajas de hacer uso de imágenes ya creadas en Docker Hub y lo sencillo que esto resulta, se ha escogido realizar las pruebas con la ayuda de la imagen curlimages/curl:7.79.1
Esta imagen crea un contenedor con el que realiza la petición curl a la dirección indicada y muestra la respuesta recibida. En la respuesta podemos verificar si se ha establecido conexión o no con el contenedor que sirve la web de ejemplo.

```
i2smorillo@i2smorillo-XPS:~/Tutorial_7$ docker run --rm curlimages/curl:7.79.1 -L -v http://172.17.0.1:4000
Unable to find image 'curlimages/curl:7.79.1' locally
7.79.1: Pulling from curlimages/curl
a0d0a0d46f8b: Pull complete
26e35fdbe697: Pull complete
c61479116531: Pull complete
e747c307f5be: Pull complete
5f47b037be16: Pull complete
5398d7c0aa83: Pull complete
382ec23d0cb7: Pull complete
aedc28606492: Pull complete
110e7f874674: Pull complete
Digest: sha256:1a2209a10a11295c3ab6952d1278a9ea2ce0d20e33fdeaeb24d7a4586767c825
Status: Downloaded newer image for curlimages/curl:7.79.1
*   Trying 172.17.0.1:4000...
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
  0     0    0     0    0     0      0      0 --:--:-- --:--:-- --:--:--     0* Connected to 172.17.0.1 (17
> GET / HTTP/1.1
> Host: 172.17.0.1:4000
> User-Agent: curl/7.79.1-DEV
> Accept: */*
>
* Mark bundle as not supporting multiuse
* HTTP 1.0, assume close after body
< HTTP/1.0 200 OK
< Content-Type: text/html; charset=utf-8
< Content-Length: 62
< Server: Werkzeug/1.0.1 Python/2.7.18
< Date: Wed, 24 Nov 2021 09:07:47 GMT
<
{ [62 bytes data]
100    62  100    62    0     0    946      0 --:--:-- --:--:-- --:--:--   953
* Closing connection 0
<h3>Conexion con exito!</h3><b>Hostname:</b> 3a4d0ff10d11<br/>i2smorillo@i2smorillo-XPS:~/Tutorial_7$
```

Figura 65: Conexión con el contenedor demo_addr a través de la dirección IPv4 del docker0 bridge

Antes de comenzar a analizar la petición y la respuesta recibida a esta misma, se observa como al no tener acceso a esa imagen de manera local Docker realiza una búsqueda en Docker Hub y encuentra una imagen con ese mismo nombre y tag. Tras localizar la imagen solicitada en Docker Hub se descarga de manera local en el host y se ejecuta acto seguido debido a que la descarga viene ocasionada por el comando *docker run* . Si se quiere obtener la imagen y no ejecutar la descarga automáticamente como consecuencia del comando *docker run* se propone el uso de *docker pull* , en este ejemplo sería con el comando:

```
docker pull curlimages/curl:7.79.1
```

La petición se hace desde otro contenedor a la dirección IPv4 asignada al docker0 bridge, se dirige la petición al puerto 4000 ya que es el que se ha expuesto. Se puede ver en la respuesta obtenida al curl que la conexión se ha tramitado con éxito y contesta el contenedor con la imagen *demo_addr*.

En la terminal que muestra la salida del contenedor que está contestando se ve registrada la recepción de la petición desde la dirección del docker0 bridge. La petición que corresponde a esta demostración es la segunda registrada en la imagen siguiente:

```
i2smorillo@i2smorillo-XPS:~/Tutorial_7$ docker run -p 4000:80 demo_addr
 * Serving Flask app "app" (lazy loading)
 * Environment: production
   WARNING: This is a development server. Do not use it in a production deployment.
   Use a production WSGI server instead.
 * Debug mode: off
 * Running on http://0.0.0.0:80/ (Press CTRL+C to quit)
172.17.0.1 - - [24/Nov/2021 09:02:29] "GET / HTTP/1.1" 200 -

172.17.0.1 - - [24/Nov/2021 09:07:47] "GET / HTTP/1.1" 200 -
```

Figura 66: Recepción de la petición a través de la dirección IPv4 de docker0 bridge

Para verificar que el puerto expuesto internamente en el contenedor de Docker con la imagen *demo_addr* no está expuesto en el docker0 bridge generamos una petición igual que la anterior modificando el puerto. Esta petición está dirigida a la dirección IPv4 del docker0 bridge y el puerto 80.

```
i2smorillo@i2smorillo-XPS:~/Tutorial_7$ docker run --rm curlimages/curl:7.79.1 -L -v http://172.17.0.1:80
*   Trying 172.17.0.1:80...
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
  0     0    0     0    0     0      0      0 --:--:-- --:--:-- --:--:--     0* connect to 172.17.0.1 port
* Failed to connect to 172.17.0.1 port 80 after 0 ms: Connection refused
  0     0    0     0    0     0      0      0 --:--:-- --:--:-- --:--:--     0
* Closing connection 0
curl: (7) Failed to connect to 172.17.0.1 port 80 after 0 ms: Connection refused
```

Figura 67: Conexión fallida con el contenedor demo_addr a través de la dirección IPv4 del docker0 bridge

Como se esperaba de manera teórica, esta conexión ha sido rechazada ya que el docker0 bridge no expone el puerto 80. Expone el puerto 4000 y redirige las peticiones al contenedor que sirve la web al puerto 80.

3.7.4 Peticiones entre contenedores en mismo host

Igual que en el apartado anterior se van a generar dos peticiones con la imagen curl para corroborar como se realizan las conexiones entre contenedores que se encuentran en la red docker0, pero esta vez la dirección IP no es la del docker0 bridge sino la del propio contenedor con la imagen demo_addr. Estas peticiones se harán desde un contenedor generado con la imagen *curlimages/curl:7.79.1* a la dirección IPv4 del contenedor con la imagen *demo_addr*.

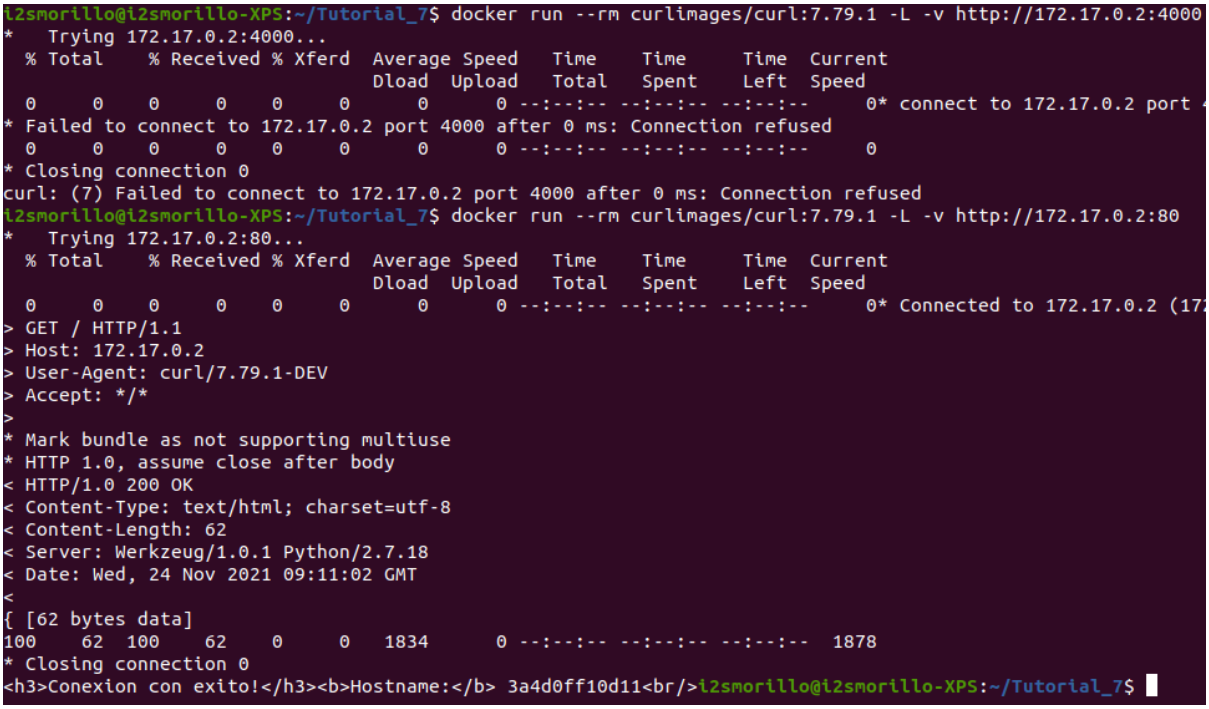

Figura 68: Conexiones con el contenedor demo_addr

Se observa como la petición realizada al puerto 4000 es rechazada, debido a que ese puerto es el que expone el dokcer0 bridge no el que expone el contenedor. Por el contrario, la petición dirigida al puerto 80 se realiza con éxito y se recibe la respuesta esperada del contenedor que aloja el servicio web. Desde la terminal que representa al contenedor con servicios web se reflejan las peticiones recibidas.

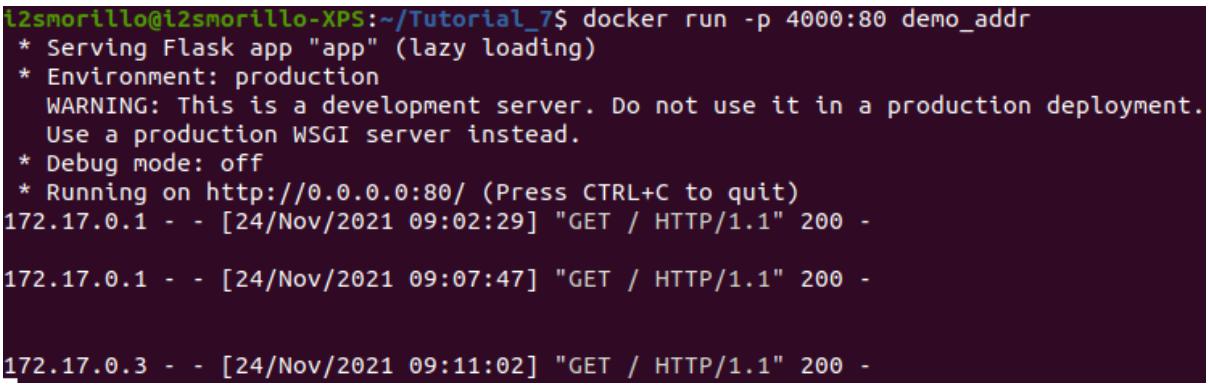

Figura 69: Recepción de la petición directa desde el contenedor con imagen curl

Relativa a esta prueba es la petición registrada en el tercer puesto de la imagen anterior, sólo se ha recibido una correctamente que es la generada contra el puerto 80. La dirección IPv4 172.17.0.3 es la relativa al contenedor que ejecuta el comando curl. Se demuestra la posibilidad de conexión directa en la subred docker0.

3.7.5 Peticiones desde un contenedor a la dirección IP de la interfaz de red de salida externa del host

Desde la terminal local al host en el que reside el contenedor con la imagen *demo_addr*, se debe verificar la dirección IP asignada y la exposición del puerto externo. En este caso el puerto es el 4000 y se realiza una búsqueda más específica de solo este:

```
i2smorillo@i2smorillo-XPS:~/Tutorial_7$ ip -c a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
2: wlp2s0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default qlen 1000
    link/ether 68:54:5a:b2:7d:f2 brd ff:ff:ff:ff:ff:ff
    inet 192.168.1.42/24 brd 192.168.1.255 scope global dynamic noprefixroute wlp2s0
       valid_lft 42027sec preferred_lft 42027sec
    inet6 fe80::1be9:2e01:5f2e:58b4/64 scope link noprefixroute
       valid_lft forever preferred_lft forever
3: docker0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default
    link/ether 02:42:53:d4:c1:04 brd ff:ff:ff:ff:ff:ff
    inet 172.17.0.1/16 brd 172.17.255.255 scope global docker0
       valid_lft forever preferred_lft forever
    inet6 fe80::42:53ff:fed4:c104/64 scope link
       valid_lft forever preferred_lft forever
```

Figura 70: Direcciones IP del host que aloja los contenedores

```
i2smorillo@i2smorillo-XPS:~/Tutorial_7$ netstat -tna | grep 4000
tcp        0      0 0.0.0.0:4000            0.0.0.0:*               ESCUCHAR
tcp6       0      0 :::4000                 :::*                    ESCUCHAR
```

Figura 71: Verificación de la escucha del puerto 4000 en el host

Se obtiene la dirección IP indicada, en este caso se trata de la 192.168.1.42 y se realizan las peticiones del mismo modo que en ejemplos anteriores.

```
i2smorillo@i2smorillo-XPS:~/Tutorial_7$ docker run --rm curlimages/curl:7.79.1 -L -v http://192.168.1.42:4000
*   Trying 192.168.1.42:4000...
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
  0     0    0     0    0     0      0      0 --:--:-- --:--:-- --:--:--     0* Connected to 192.168.1.42 (19
> GET / HTTP/1.1
> Host: 192.168.1.42:4000
> User-Agent: curl/7.79.1-DEV
> Accept: */*
>
* Mark bundle as not supporting multiuse
* HTTP 1.0, assume close after body
< HTTP/1.0 200 OK
< Content-Type: text/html; charset=utf-8
< Content-Length: 62
< Server: Werkzeug/1.0.1 Python/2.7.18
< Date: Wed, 24 Nov 2021 09:16:08 GMT
<
{ [62 bytes data]
100    62  100    62    0     0   1998      0 --:--:-- --:--:-- --:--:--  2066
* Closing connection 0
<h3>Conexion con exito!</h3><b>Hostname:</b> 3a4d0ff10d11<br/>i2smorillo@i2smorillo-XPS:~/Tutorial_7$
i2smorillo@i2smorillo-XPS:~/Tutorial_7$ docker run --rm curlimages/curl:7.79.1 -L -v http://192.168.1.42:80
*   Trying 192.168.1.42:80...
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
  0     0    0     0    0     0      0      0 --:--:-- --:--:-- --:--:--     0* connect to 192.168.1.42 port 8
* Failed to connect to 192.168.1.42 port 80 after 0 ms: Connection refused
  0     0    0     0    0     0      0      0 --:--:-- --:--:-- --:--:--     0
* Closing connection 0
curl: (7) Failed to connect to 192.168.1.42 port 80 after 0 ms: Connection refused
```

Figura 72: Conexiones con el contenedor demo_addr a través de la dirección IPv4 del host que aloja el contenedor demo_addr

Esta conexión se produce de manera externa a la red docker0, atravesando el docker0 bridge, por ello se visualiza como la conexión realizada con éxito es en la cual se dirige la petición al puerto expuesto de manera externa.

3.7.6 Peticiones desde otros dispositivos en la red local

Con otros dispositivos conectados a la red local podemos verificar el funcionamiento, esta petición debe ser a la dirección IP del host con el puerto expuesto de manera externa. Se verifica desde otro PC como se visualiza en la siguiente imagen:

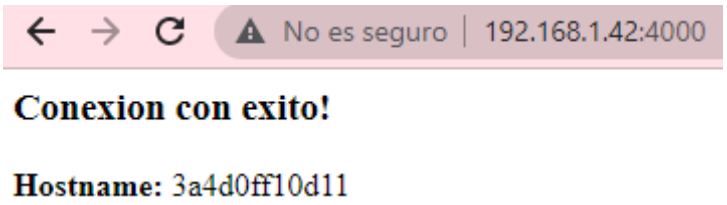

Figura 73: Servicio web del contenedor demo_addr desde otro dispositivo de la red

En la siguiente imagen quedan reflejadas las peticiones recibidas en el contenedor Docker:

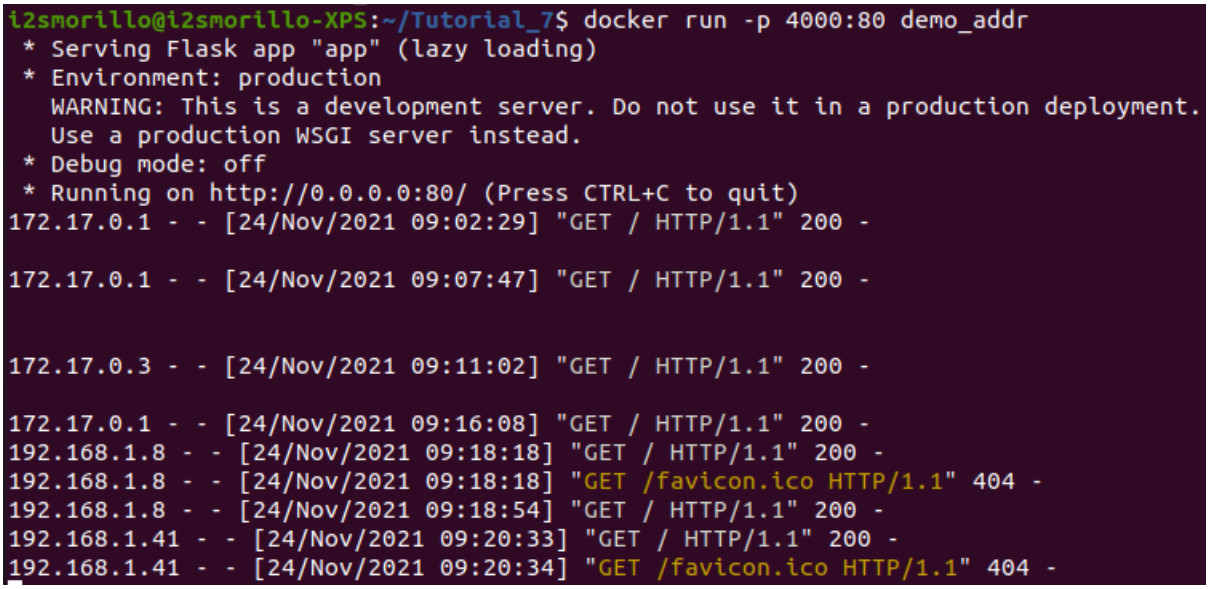

Figura 74: Recepción de las peticiones de otros dispositivos de la red

En este ejemplo las direcciones 192.168.1.8 (teléfono móvil) y 192.168.1.41 (PC portátil) son de dos dispositivos conectados en la misma red que el host que aloja el servicio. De esta manera se podría prestar servicios a los usuarios/dispositivos conectados en la misma red, para hacer este servicio global debemos abrir el puerto necesario en el router de salida a internet. De ese modo se podría lograr la conexión desde otras redes.

3.8 Tutorial 8: Servicios en Docker Swarm en distintos hosts

Este apartado ejecuta unos servicios similares a los del tutorial 3.5, pero en este caso los workers del Docker Swarm se distribuyen en distintos hosts que no pertenecen a la misma red. El hecho de que los hosts físicos se encuentren en distintas redes debe ser algo transparente para Docker Swarm que seguirá construyendo su red como en el apartado anterior.

Docker Swarm construye la red de igual manera, independientemente de donde estén alojados sus workers o master. La ilustración siguiente muestra como perciben los contenedores la red a pesar de encontrarse en redes físicamente distintas. Los diferentes contenedores Docker están, desde su punto de vista, unidos por una red local a través de la cual pueden comunicarse con sus direcciones IPs privadas.

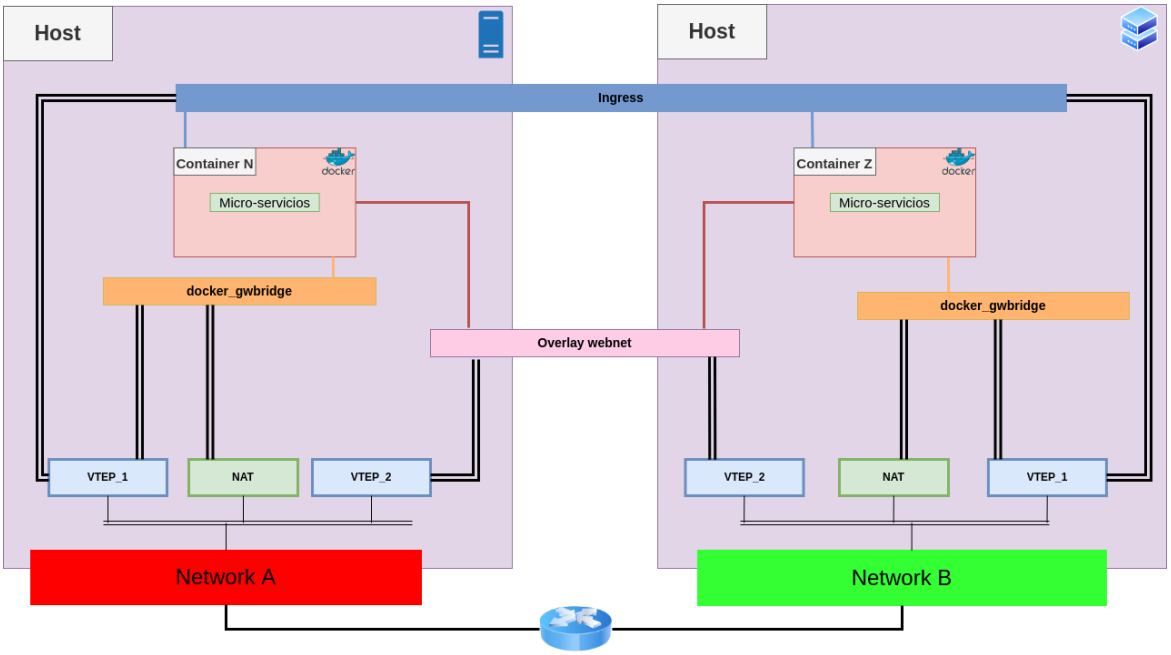

Figura 75: Percepción de la red en Docker Swarm en redes físicamente distintas ejemplo capítulo 3.8

La imagen esquemática representa la situación que se reproduce en este capítulo. En el caso del “docker_gwbridge” que construye Docker se unen todos los contenedores alojados en el mismo host para permitir su conexión con el mismo host que los alojad, que a su vez es la “puerta al exterior”.

La red *Ingress* es la cual presenta los servicios de manera externa, en el caso de este capítulo todos los contenedores exponen sus servicios al exterior, por ello todos los contenedores tienen conexión a esta red [27].

Por último la red overlay webnet , que se ve reflejada rosa, ha sido creada en docker-file.yaml (*Código 15* en el apartado 3.8.1) utilizado, esta red tiene el nombre de webnet. Esta red podría servir para aislar un servicio de otros o comunicarse entre servicios de manera aislada, en este caso no es necesario, pero se crea para mostrar la posibilidad que ofrece Docker.

Este caso redactado es la situación en que se origina en la prueba, pero puede darse la posibilidad de que algunos contenedores no necesiten estar en la red *ingress* y exponer sus servicios al exterior.
Un esquema que puede representar este ejemplo es el siguiente:

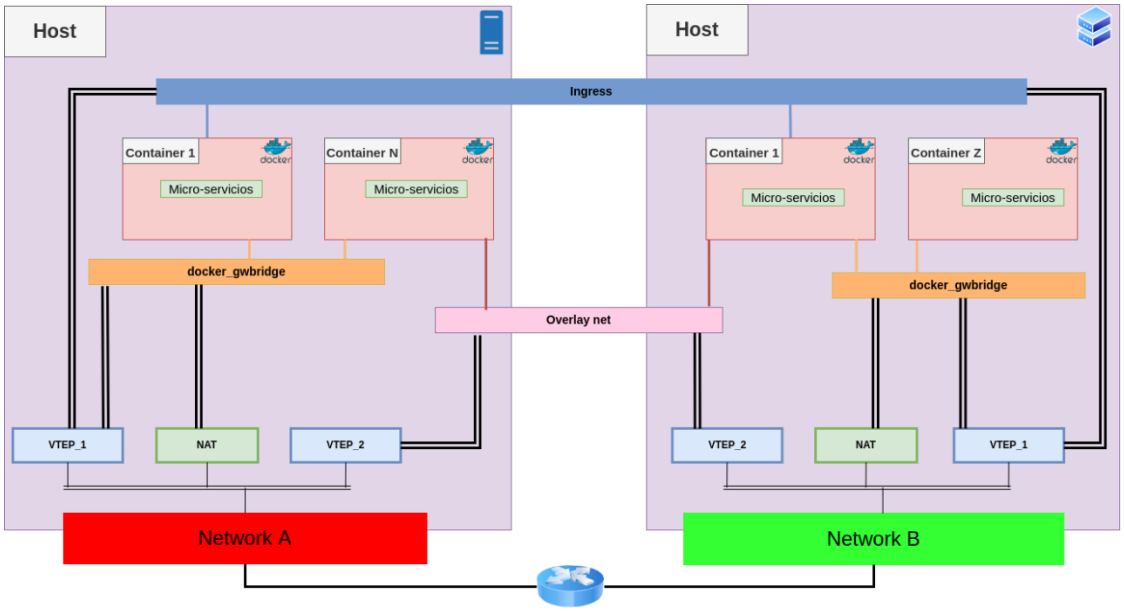

Figura 76: Percepción de la red en Docker Swarm en ejemplo en cual no todos los contenedores estas expuestos externamente

En este ejemplo se muestran contenedores que están expuestos externamente, resultante de ello estos están conectados a la red Ingress. Pero estos mismo no están conectados a la red overlay. La red overlay está en uso por otro servicio al que podrían pertenecer los contenedores N y Z, o estos podrían ser de dos servicios diferentes que se ha definido su unión a la red overlay. El servicio o servicios que están unidos a la red overlay no están expuestos externamente con la red *Ingress*.

3.8.1 Configuraciones de la imagen Docker y docker-compose

Se usa de base de código de Dockerfile el usado en el tutorial “primeros pasos con Docker”, se hará la instalación de la herramienta curl. Esto servirá para realizar comprobaciones de healthcheck sobre el contenedor y monitorizar así su estado protegiéndolo contra fallos.

El código de Dockerfile es el siguiente:

```
# Specify a base image
FROM python:2.7-slim

#Set the working directory
WORKDIR /usr/app

#Copy my local directory into the container /app
COPY . /usr/app

# Install any needed packages specified in requirements.txt
RUN pip install --trusted-host pypi.python.org -r requirements.txt

# Install iproute-2 and net-tools
RUN apt-get update && apt-get install -y \
    curl \
 && rm -rf /var/lib/apt/lists/*

#Expose the container port 80
EXPOSE 80

#Define environment variable
ENV NAME World

#Run app.py when the container launches
CMD ["python","app.py"]
```

Código 15: Dockerfile para la creación de la imagen service_web_curl

Se crea la imagen y se sube al repositorio personal con las instrucciones:

```
docker build -t service_web_include_curl .
docker tag
service_web_include_curl smorilloh/get-started:service_web_include_curl
docker push smorilloh/get-started:service_web_include_curl
```

De este modo ya es accesible la imagen desde cualquier host con acceso al hub de Docker remotamente.

A continuación, se muestra el docker-compose.yml que se usa inicialmente pero que se irá variando en el número de réplicas para las demostraciones a realizar:

```yaml
version: "3"
services:
  web:
    image: smorilloh/get-started:service_web_include_curl
    deploy:
     #placement:
       #constraints: [node.role == worker]
      replicas: 6
      update_config:
        parallelism: 2
        delay: 5s30ms
      restart_policy:
        condition: on-failure
      resources:
        limits:
          cpus: "0.1"
          memory: 50M
    healthcheck:
      test: curl --fail -s http://localhost:80/ || exit 1
      interval: 1m30s
      timeout: 25s
      retries: 3
    ports:
      - "4000:80"
    networks:
      - webnet
networks:
  webnet:
```

Código 16: Archivo docker-compose.yml con un servicio que se modificarán las réplicas

Elementos nuevos son la comprobación healthcheck explicada anteriormente y el apartado update_config en que se configura que cuando existan actualización del servicio *deploy* las réplicas se actualicen en grupos de máximo dos. Además, entre actualizaciones en grupos de dos debe haber un delay de cinco segundos y 300 milisegundos.
He de mencionar que las unidades de tiempo soportadas en la configuración docker-compose.yml son us, ms, s, m y h.

Se encuentra comentado, pero puede ser de uso la directiva *placement* que si se descomenta obliga que las réplicas de *deploy* se encuentren en nodos tipo worker.

3.8.2 Configuración de la red

Se va a situar un worker en el Docker Swarm en una red distinta en la que está alojada el master. Para que estos puedan comunicarse de manera independiente a donde esté situada la red de cada uno se deben configurar los puertos de entrada a la red local.

Los puertos que deben estar disponibles son:

- El puerto 2377 con protocolo TCP, para la comunicación de administración.

- El puerto 7046 con los protocolos TCP y UDP, para comunicación entre nodos,
- El puerto 4789 con el protocolo UDP, para la red overlay.

En el caso de este tutorial se ha desarrollado en una red con un “Router Smart WiFi” que es editable a través de una consola web alcanzable en la dirección IPv4 local 191.168.1.1.

En la sección configuración de puertos se debe configurar las siguientes reglas:

Nombre	Protocolo	Puerto/Rango Externo	Puerto/Rango Interno	Dirección IP
Docker_config_1	TCP	2377	2377	192.168.1.42
Docker_config_2	TCP	7046	7046	192.168.1.42
Docker_config_2	UDP	7046	7046	192.168.1.42
Docker_config_3	UDP	4789	4789	192.168.1.42

Figura 77: Configuración de puertos del router

Nota: 192.168.1.42 es la dirección IP del host en el que reside Docker Swarm.

Estas reglas deben estar configuradas en las distintas entradas a la red donde se aloja cada host en el que resida un nodo de Docker Swarm.

3.8.4 Ejecución de Docker Swarm nodos en distintas redes

En esta parte como indica el titulo se alojan varios nodos del Swarm en distintas redes, un ejemplo gráfico y esquemático podría ser la siguiente imagen:

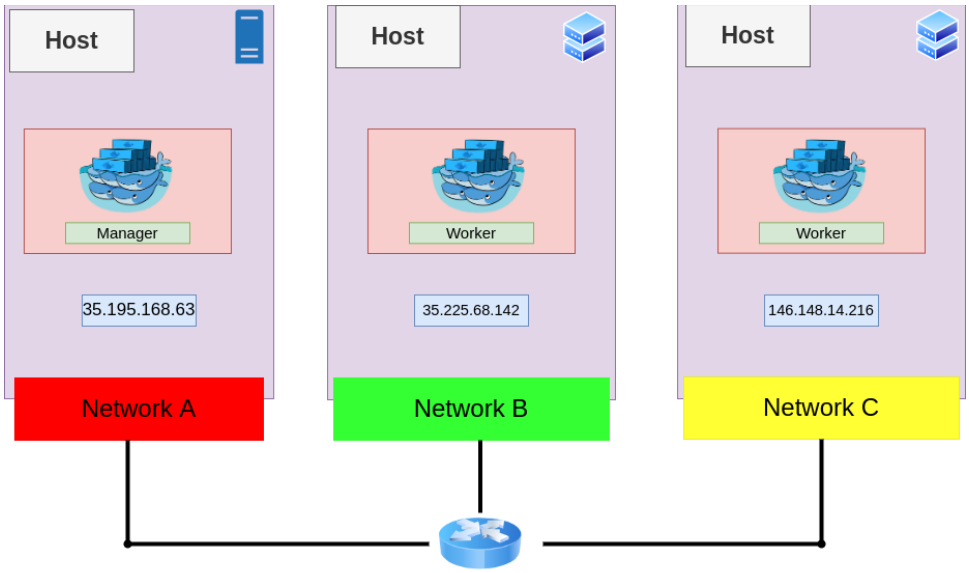

Figura 78: Ejemplo de nodos Swarm alojados en distintas redes

Desde el punto de vista de los contenedores que se ejecutan en el Swarm no distinguen si se encuentran el mismo nodo o no. En el caso de este tutorial se trata de redes y nodos distintos y esto requiere una configuración para que los hosts puedan comunicarse entre sí el tráfico e información procedente del Swarm.

Se inicia el nodo Docker Swarm que se establece como master como habitualmente con la ejecución del comando:

```
docker swarm init
```

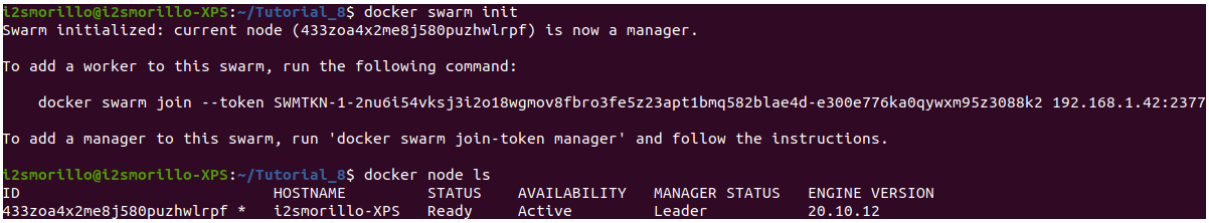

Figura 79: Inicio de Docker Swarm en host local

Se debe introducir en el otro host que se une al Swarm la instrucción que nos indica la salida del comando anterior, pero se debe modificar la dirección IP final por la dirección IP pública que alcanzará la entrada de la red y con la regla configurada correctamente hará entrega al host indicado.

Otra opción es indicarle directamente la dirección IP pública con el parámetro *--advertise-addr*

Se introduce en el segundo host alojado en Google Cloud Platform, por tanto, en otra red el comando:

```
docker swarm join --token SWMTKN-1-
2nu6i54vksj3i2o18wgmov8fbro3fe5z23apt1bmq582blae4d-
e300e776ka0qywxm95z3088k2 <IP-externa>:2377
```

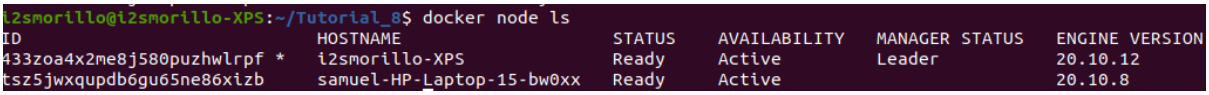

Figura 80: Comprobación del acceso al Swarm con distintas redes

Tras comprobar que los nodos se encuentran en comunicación, se procede a levantar los servicios. Para ello se ejecuta:

```
docker stack deploy -c docker-compose.yml getstartedlab
```

Se puede observar el reparto de las réplicas entre los nodos con el comando:

```
docker stack ps getstartedlab
```

Se obtiene un resultado similar al de la imagen de a continuación:

```
i2smorillo@i2smorillo-XPS:~$ docker stack ps getstartedlab
ID             NAME                  IMAGE                                NODE
r976z3ovnvru   getstartedlab_web.1   smorilloh/get-started:testing-networks   samuel-HP-Laptop-15-bw0xx
4to8htslm3d5   getstartedlab_web.2   smorilloh/get-started:testing-networks   i2smorillo-XPS
ywz6je3im53e   getstartedlab_web.3   smorilloh/get-started:testing-networks   samuel-HP-Laptop-15-bw0xx
```

Figura 81: Comprobación del reparto de réplicas en los distintos nodos del Swarm

En este momento podemos exponer el puerto 4000 del router para búsquedas externas.

Nombre	Protocolo	Puerto/Rango Externo	Puerto/Rango Interno	Dirección IP
Docker_config_1	TCP	2377	2377	192.168.1.42
Docker_config_2	TCP	7046	7046	192.168.1.42
Docker_config_2	UDP	7046	7046	192.168.1.42
Docker_config_3	UDP	4789	4789	192.168.1.42
Acceso_web	TCP	4000	4000	192.168.1.42
Acceso_web	UDP	4000	4000	192.168.1.42

Figura 82: Configuración de puertos en router Movistar

3.9 Tutorial 9: Ejemplo real de servicios globales en internet con uso Docker

3.9.1 Alojar contenedores en plataformas cloud

En esta parte se alojan los contenedores en máquinas virtuales creadas en la plataforma Google Cloud Platform (GCP).
Es habitual que un servicio se quiera ofrecer de manera global y algo muy valorado por los usuarios es la disponibilidad, con intención de no saturar una misma máquina se va a realizar una prueba en la que se crean varias instancias en GCP.
Se sitúan las instancias en diferentes localizaciones simulando así un posible problema de red o similar en una de las regiones de GCP, esto aumenta la característica de disponibilidad.

Se ha creado varias instancias para este ejemplo dos de ellas en Europa en la localización de Bélgica, zona recomendada para servicios orientados a España, y otra instancia localizada en Norte-América, Iowa.

Estado	Nombre ↑	Zona	Recomendaciones	En uso por	IP interna	IP externa
✓	docker-swarm-instance	europe-west1-b			10.132.0.2 (nic0)	35.195.168.63
✓	instance-docker-2	europe-west1-b			10.132.0.3 (nic0)	146.148.14.216
✓	instance-docker-america	us-central1-c			10.128.0.2 (nic0)	35.225.68.142

Figura 83: Instancias creadas en distintas zonas

Se va a hacer uso del mismo docker-compose.yml que en el tutorial anterior, pero aumentando el número de instancias.

3.8.3 Configuración de una instancia en Google Cloud Platform

Como se ha comentado se va a hacer uso de la infraestructura de GCP y por ello aquí se muestra como configurar máquinas virtuales en esta plataforma para la prueba.

Los servicios actualmente más usados son la computación en red, en este caso crearemos una instancia Linux en el proveedor GCP (Google Cloud Platform). Esto supone una ventaja a la hora de distribuir servicios de manera global ya que las máquinas se pueden alojar en varias localizaciones del planeta.

Las imágenes Docker se pueden implementar en distintos servicios de GCP, en este caso se aloja al worker en una máquina virtual. Se debe crear una máquina virtual en el apartado de *Compute Engine*.

Se selecciona “crear una instancia” y las características básicas sirven para un ejemplo que no es necesario optimizar al máximo.

Es necesario habilitar el tráfico HTTP para las posteriores pruebas de acceso al servicio web.

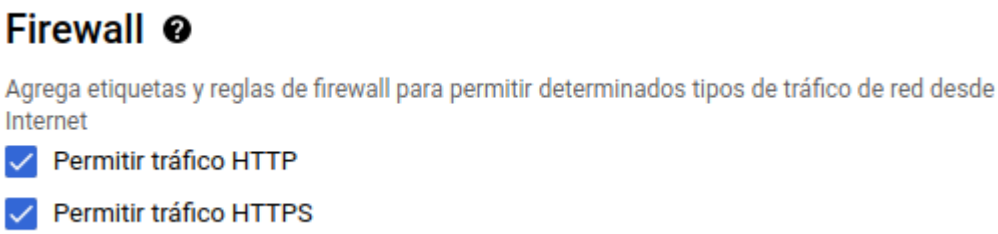

Figura 84: Habilitado el tráfico HTTP en máquina GCP (Google Compute Platform)

Una vez creada la instancia, para facilitarnos el trabajo se añade la clave de acceso SSH pública del ordenador que se usa para conectar con la instancia mediante SSH. Esto se implementa en la parte de edición de la instancia.

Una vez la máquina está iniciada genera una IP pública efímera que nos servirá para su conexión. Es importante que el firewall de GCP esté correctamente configurado para permitir las conexiones necesarias.

3.9.2 Servicios alojados en las máquinas de GCP comportamiento distinto entre nodos

Se inicia una conexión SSH con cada una de estas instancias para hacer muestra de los pasos seguidos; pero una de las ventajas es la cual nos permite centralizar la gestión de estos servicios mediante el nodo manager. Una vez configurado el nodo manager y los workers es posible controlar los servicios únicamente desde una máquina con rol de manager. En este caso se va a usar como manager la máquina con nombre “docker-swarm-instance”.

```
l2smorillo@docker-swarm-instance:~$ docker swarm init --advertise-addr 35.195.168.63
Swarm initialized: current node (jcuf6vrnrmjg408eryuw0jjmi) is now a manager.

To add a worker to this swarm, run the following command:

    docker swarm join --token SWMTKN-1-4fht0frjrcsc66ke8um2n7y6hfmhjejk3ow00xl3wkyw036zu6-200og4ez8bclza9o4nr9irsjd 35.195.168.63:2377

To add a manager to this swarm, run 'docker swarm join-token manager' and follow the instructions.
```

Figura 85: Inicio de Docker Swarm en instancia GCP con IP pública

Se debe configurar el firewall para que se puedan comunicar los servicios de Docker Swarm con destino a las máquinas alojadas en GCP. Este firewall es el implementado en la red asociada al proyecto GCP.

Nombre	Tipo	Destinos	Filtros	Protocolos/puertos	Acción	Prioridad	Red
comunicaciones-docker-swarm	Entrada	Aplicar a toda	Intervalos de IP: 0.0.0	tcp:2377, 7046 udp:4789, 7046	Permitir	1000	default

Figura 86: Configuración Firewall para comunicaciones Docker

Una vez se han unido como workers las dos instancias restantes que se han creado, se puede comprobar que el Swarm está compuesto por tres host con uno de ellos únicamente como manager:

```
l2smorillo@docker-swarm-instance:~$ docker node ls
ID                            HOSTNAME                STATUS    AVAILABILITY   MANAGER STATUS   ENGINE VERSION
jcuf6vrnrmjg408eryuw0jjmi *   docker-swarm-instance   Ready     Active         Leader           20.10.12
u35h71l2mn6o3kuk1kg81ou01     instance-docker-2       Ready     Active                          20.10.12
r4bi83e58swsgzmxrelabp8up     instance-docker-america Ready     Active                          20.10.12
```

Figura 87: Visualización de los nodos en el Swarm

Se despliegan los servicios como anteriormente se ha mencionado y posteriormente se puede visualizar la distribución de las réplicas de cada servicio:

```
i2smorillo@docker-swarm-instance:~$ docker stack deploy -c docker-compose.yml getstartedlab
Creating network getstartedlab_webnet
Creating service getstartedlab_web
i2smorillo@docker-swarm-instance:~$ docker node ls
ID                            HOSTNAME                STATUS    AVAILABILITY   MANAGER STATUS   ENGINE VERSION
jcuf6vrnrmjg408eryuw0jjmi *   docker-swarm-instance   Ready     Active         Leader           20.10.12
u35h71l2mn6o3kuk1kg81ou01     instance-docker-2       Ready     Active                          20.10.12
r4bi83e58swsgzmxrelabp8up     instance-docker-america Ready     Active                          20.10.12
i2smorillo@docker-swarm-instance:~$ docker stack ps getstartedlab
ID             NAME                  IMAGE                                          NODE                    DESIRED STATE
vkk9026517h0   getstartedlab_web.1   smorilloh/get-started:service_web_include_curl   instance-docker-2       Running
yo5mlqzj9zr0   getstartedlab_web.2   smorilloh/get-started:service_web_include_curl   docker-swarm-instance   Running
6y7kzslixr03   getstartedlab_web.3   smorilloh/get-started:service_web_include_curl   instance-docker-america Running
oa7iw7v389ll   getstartedlab_web.4   smorilloh/get-started:service_web_include_curl   instance-docker-2       Running
z7ly5o5hv9ch   getstartedlab_web.5   smorilloh/get-started:service_web_include_curl   instance-docker-america Running
q1fhp05birnm   getstartedlab_web.6   smorilloh/get-started:service_web_include_curl   instance-docker-2       Running
ls6m01vwm1id   getstartedlab_web.7   smorilloh/get-started:service_web_include_curl   docker-swarm-instance   Running
bsnmajmpb74v   getstartedlab_web.8   smorilloh/get-started:service_web_include_curl   instance-docker-america Running
```

Figura 88: Distribución de las réplicas en los distintos host GCP

Se destaca que el campo NODE contiene tres valores distintos, estos son cada una de las tres instancias que están alojando los contenedores Docker y han sido creada como máquinas virtuales en GCP.

Tras la ejecución de los pasos anteriores se pueden realizar búsquedas a cualquiera de las IPs públicas de las máquinas creadas en GCP. Estas IPs no son fijas, aunque podrían serlo mediante el pago para reservar una IP fija. En este caso las IPs de estas tres máquinas se ven reflejada en la imagen “Figura 83: Instancias creadas en distintas zonas”, son 146.148.14.216, 35.195.168.63 y 35.225.68.142. Cabe destacar la importancia de permitir el tráfico HTTP/S en el puerto que se ha escogido exponer públicamente para el servicio en el Firewall:

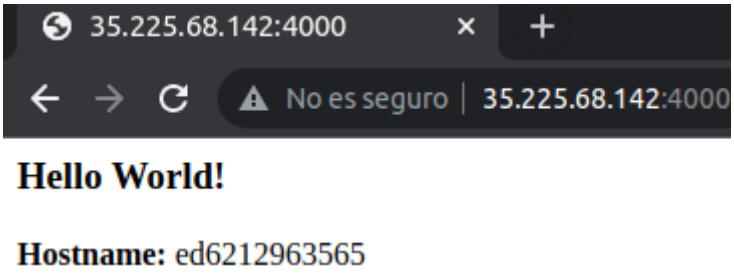

Figura 89: Distintas búsquedas de los servicios mediante IPs públicas GCP

Es destacable que la unión de los workers al nodo master se ha realizado mediante la dirección IPv4 pública que proporciona GCP (35.195.168.63), esto crea las uniones entre los hosts de manera externa sin "ver" una posible red local creada entre ellos.

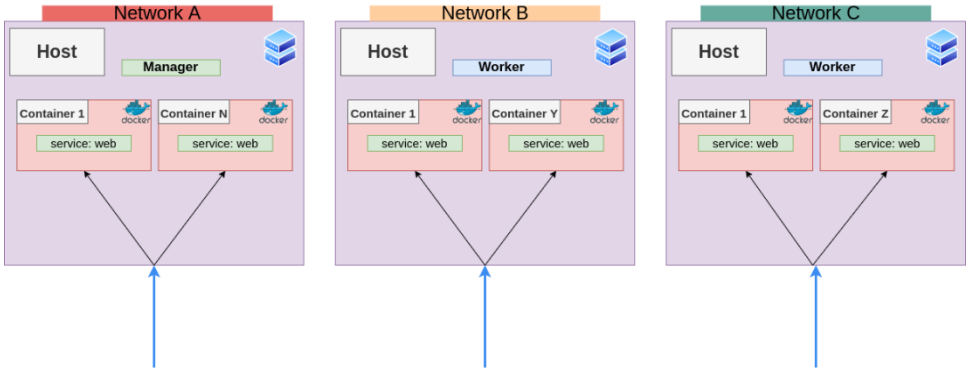

Figura 90: Modo de reparto del tráfico procedente de Internet en el capítulo 3.9.2

3.9.3 Servicio distribuido entre las diferentes máquinas de GCP

En este ejemplo se alojarán contenedores en distintas máquinas virtuales de GCP, las peticiones recibidas en una única IP la del nodo manager será distribuida entre los distintos contenedores Docker.

Se comienza iniciando las conexiones SSH como en el caso anterior, a diferencia del capítulo 3.9.2, se inicia el Docker Swarm en el host que se decida que se el manager con la dirección IPv4 local. Para esta opción se puede omitir la opción *--advertise-addr* y se seleccionará la IP de la red local.

```
i2smorillo@docker-swarm-instance:~$ docker swarm init
Swarm initialized: current node (jjvvwkvg5z1sg50it9m39uehn) is now a manager.

To add a worker to this swarm, run the following command:

    docker swarm join --token SWMTKN-1-5vgve2raj8nh65eopj04bjau0wt1z6kdcisvdnhvyb0ey64d9h-bl2gy2o8id59roximz8lwab5r 10.132.0.2:2377

To add a manager to this swarm, run 'docker swarm join-token manager' and follow the instructions.

i2smorillo@docker-swarm-instance:~$ docker node ls
ID                            HOSTNAME              STATUS    AVAILABILITY   MANAGER STATUS   ENGINE VERSION
jjvvwkvg5z1sg50it9m39uehn *   docker-swarm-instance     Ready     Active         Leader           20.10.12
v38f0orrv2set4rywnau0c7rk     instance-docker-2         Ready     Active                          20.10.12
puwxe2lu5kh8o9xj8onkjj4cr     instance-docker-america   Ready     Active                          20.10.12
```

Figura 91: Inicio del Docker Swarm en GCP con IP local

Se encuentra una situación similar a la del capítulo anterior, pero en esta ocasión se ha usado la sentencia señalada en la imagen para unir a los workers. Los otros nodos se han unido mediante la red local como se puede observar en ese comando, esto es posible debido a que independientemente de la geolocalización de nuestros servidores/host Google nos ofrece la posibilidad de crear una red local gestionada por GCP casi por completo.

Se inicia como en casos anteriores los servicios, se va a fijar el número de réplicas para que cada máquina virtual aloje una.

```
i2smorillo@docker-swarm-instance:~$ docker stack deploy -c docker-compose.yml getstartedlab
Creating network getstartedlab_webnet
Creating service getstartedlab_web
i2smorillo@docker-swarm-instance:~$ docker stack ps getstartedlab
ID             NAME                  IMAGE                                          NODE
tw4v9f2er8pq   getstartedlab_web.1   smorilloh/get-started:service_web_include_curl   instance-docker-america
cthh91i6fi89   getstartedlab_web.2   smorilloh/get-started:service_web_include_curl   instance-docker-2
wivelmklx9qx   getstartedlab_web.3   smorilloh/get-started:service_web_include_curl   docker-swarm-instance
```

Figura 92: Distribución de las réplicas en el Swarm GCP

Se puede usar cualquiera de las tres direcciones IP públicas de las máquinas y las peticiones serán balanceadas entre todas las réplicas del Swarm.

```
~
curl 35.205.218.196:4000
<h3>Hello World!</h3><b>Hostname:</b> 9c678776b461<br/><

~
curl 35.205.218.196:4000
<h3>Hello World!</h3><b>Hostname:</b> ddc3b6c86492<br/><

~
curl 35.205.218.196:4000
<h3>Hello World!</h3><b>Hostname:</b> 19f5d42ddc88<br/><
```

Figura 93: Respuesta de los distintos nodos con petición a la misma IP pública

Se observa como el hostname, que es realmente el ID del contenedor Docker que responde, es en cada petición una réplica distinta del servicio *deploy*. Esto se debe a que se realiza un balanceo de carga desde cualquiera de las direcciones IP de las máquinas.

A diferencia del capítulo 3.9.2 que cada nodo balanceaba únicamente entre las réplicas alojadas en sí mismo.

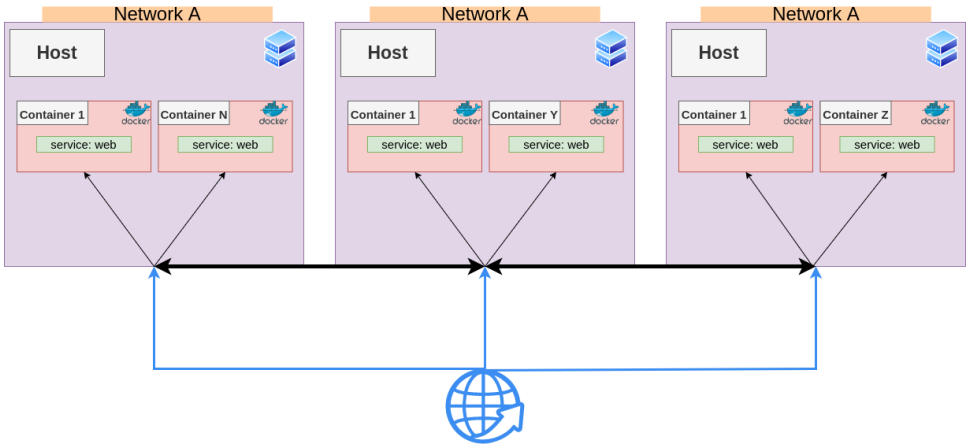

Figura 94: Modo de reparto del tráfico procedente de Internet en el capítulo 3.9.3

4. Conclusión y futuras líneas de estudio

Como parte final del trabajo se asientan unas conclusiones en lo referente al software Open Source de Docker, sobre el cual se ha expuesto conceptos teóricos y usos prácticos como se establecía en el objetivo.

Las ventajas de Docker presentadas son muy numerosas como definir en código la infraestructura, su continua evolución u optimización y portabilidad de sus imágenes. Además, al tratarse de una tecnología Open Source en continua evolución aumentaran los beneficios y rendimiento.

Estas especificaciones hacen de Docker una herramienta con gran utilidad, además encaja en la tendencia de los últimos años de alojar servicios en infraestructura *Serverless*. El concepto *Serverless* tiene la traducción estricta “sin servidor”, se conoce de esta manera al modo de trabajo en el cual la infraestructura es proveída solamente en el momento el cual la aplicación necesita ejecutarse; esto permite un ahorro de coste computacional considerable. Por esta razón se presenta en este trabajo el uso de Docker de manera práctica, el concepto de utilizar contenedores Docker en arquitecturas *Serverless* es muy útil y extendido. La intención de ofrecer a personas sin conocimientos previos de esta herramienta su uso y ventajas.

Durante los diferentes capítulos de este trabajo se muestra el uso de la herramienta Docker comenzando por un punto de partida como es la instalación. Tras obtener de manera gratuita el software se muestra su uso con ejemplos de creación de imágenes, almacenado de imágenes en registros, almacenamiento de datos persistentes o uso de redes entre contenedores.

En la parte final del trabajo se muestra un ejemplo de uso cercano a la realidad en el cual se alojan contenedores Docker en la infraestructura cloud GCP. La adopción de Docker es un hecho que está en aumento en entornos reales de virtualización. Se puede referenciar como modelo la empresa Spotify que hace uso de Docker para sus microservicios.

Como futuras líneas de trabajo existen varias posibilidades, una de las opciones más útiles podría ser el estudio de la composición de diferentes redes de comunicación entre los distintos contenedores y como se establece los nombres asociados al DNS interno que organiza el tráfico. En el caso en el cual existen varios servicios alojados en contenedores Docker se puede maximizar la seguridad creando las redes optimas que establecen las conexiones únicamente necesarias, ayudando de esta manera a maximizar el aislamiento de los contenedores y minimizando el riesgo de ataques o tráfico inservible.

5. Presupuesto

5.1 Presupuesto de recursos materiales y software

- Microsoft Word 2010: 69€
- Uso de infraestructuras Google Cloud Platform: 50€
- PC portátil HP: 500€
- PC portátil Dell: 900€
- Cuota de conexión a internet: 65€/mensuales

La suma de los diferentes gastos en recursos materiales y software acumulan **un total de 2300€.**

5.2 Presupuesto recursos humanos

El coste de la mano de obra está en referencia a un coste medio de unos 15,45€/h para un empleado con relación a una ingeniería de telecomunicaciones.

El tiempo asociado a cada trabajo es aproximado y está dividido en los distintos capítulos que hacen muestras prácticas y en la redacción de la memoria incluyendo las redacciones teóricas no asociadas a ninguna prueba.

Trabajo	Tiempo empleado (horas)	Coste
Redacciones de partes teóricas, composición de la memoria y recopilación de requisitos	180	2781€
Tutorial 1: Instalación de Docker Engine y creación de usuario en Docker Hub	25	386€
Tutorial 2: Primeros pasos con Docker	35	541€
Tutorial 3: Servicios Docker definidos en docker-comopose.yml	40	618€
Tutorial 4: Docker swarm para presentar los servicios	55	850€
Tutorial 5: Cooperación de varios servicios alojados en Docker Swarm	45	695€
Tutorial 6: Uso de volúmenes de almacenamiento en Docker	60	927€
Tutorial 7: Comunicaciones en Docker	55	850€

Tutorial 8: Servicios en Docker Swarm con host en distintas redes	75	1159€
Tutorial 9: Ejemplo real de servicios globales en internet con uso Docker	75	1159€
Total	**615**	**9966€**

Figura 95: Tabla costes asociados a recursos humanos

5.3 Presupuesto completo

Uniendo los gastos asociados a la mano de obra empleada y costes asociados a software o recursos resultan los siguientes gastos:

Concepto	Precio total
Recursos materiales y software	2300€
Recursos humanos	9966€
Total	**12266€**

Figura 96: Tabla costes totales del proyecto

6. Bibliografía

[1] Red Hat, Inc. “¿Qué es Docker?” *Red Hat*, 9 Enero 2018, https://www.redhat.com/es/topics/containers/what-is-docker. Accessed 2022.

[2] Arco, José Manuel. “Tutoriales sobre Docker.” *GP Contenedores con Teoría*, 2018.

[3] Sánchez, Ignacio. “Clúster de Docker con Swarm Mode.” En Mi Local Funciona, 2 August 2016, https://enmilocalfunciona.io/cluster-de-docker-con-swarm-mode/. Accessed 2022.

[4] Briceño, Gary. “¿Que es Docker?” *Club de Tecnología*, 20 November 2017, https://www.clubdetecnologia.net/blog/2017/que-es-docker/. Accessed 2022.

[4] Quiroga, David. “cgroups - Grupos de Control en GNU/Linux.” *clibre.io*, 30 September 2020, https://clibre.io/blog/por-secciones/hardening/item/425-cgroups-grupos-de-control-en-gnu-linux. Accessed 2022.

[5] Red Hat, Inc. “¿Qué son los contenedores Linux (LXC)?” *Red Hat*, 31 January 2018, https://www.redhat.com/es/topics/containers/whats-a-linux-container#%C2%BFen-qu%C3%A9-consiste-lxc. Accessed 2022.

[6] Novoseltseva, Ekaterina. “Utilizar Docker: beneficios, estadísticas.” Apiumhub, 3 April 2020, https://apiumhub.com/es/tech-blog-barcelona/beneficios-de-utilizar-docker/. Accessed 2022.

[7] Quiroga, David. “Namespaces - Aíslar los procesos GNU/Linux en sus propios entornos de sistema.” clibre.io, 16 April 2020, https://clibre.io/blog/por-secciones/hardening/item/384-namespaces-aislar-los-procesos-de-linux-en-sus-propios-entornos-de-sistema. Accessed 2022.

[8] Mishra, Pratik. “Docker Engine - Deep Dive into Docker. - DEV Community.” *DEV Community* , 15 April 2021, https://dev.to/pratik6217/docker-engine-deep-dive-into-docker-4hig. Accessed 2022.

[9] Ilimit Comunicacions S.L. “Kubernetes y Swarm ¿Cuándo usarlos?” *Ilimit Comunicacions*, 7 July 2020, https://www.ilimit.com/blog/kubernetes-vs-swarm/. Accessed 2022.

[10] Howchoo. “How to Add a Health Check to Your Docker Container.” *Howchoo*, 12 April 2021, https://howchoo.com/devops/how-to-add-a-health-check-to-your-docker-container#an-example-using-flask. Accessed 2022.

[11] Sengupta, Sudip. “Kubernetes vs Docker Swarm: Comparing Container Orchestration Tools.” BMC Software, 11 November 2020, https://www.bmc.com/blogs/kubernetes-vs-docker-swarm/. Accessed 2022.

[12] L. Garcia. “Unas estadísticas sobre Kubernetes – Un poco de Java.” Un poco de Java, 17 January 2020, https://unpocodejava.com/2020/01/17/unas-estadisticas-sobre-kubernetes/. Accessed 2022.

[13] Docker, Inc. “Docker Machine Overview.” Docker Documentation, https://docker-docs.netlify.app/machine/overview/#where-to-go-next. Accessed 2022.

[14] Starks John, and Taylor Brown. “Windows Server and Docker - The Internals Behind Bringing Docker and ….” *SlideShare*, https://www.slideshare.net/Docker/windows-server-and-docker-the-internals-behind-bringing-docker-and-containers-to-windows-by-taylor-brown-and-john-starks. Accessed 2022

[15] *What are Windows Containers?*, 21 May 2017, http://passionatecoder.ca/Article/What-are-Windows-Containers. Accessed 2022.

[16] Scherer, Stefan. “StefanScherer/docker-windows-box: Various Vagrant envs with Windows 2019/10 and Docker, Swarm mode, LCOW, WSL2, ...” *GitHub*, 2020, https://github.com/StefanScherer/docker-windows-box. Accessed 2022.

[17] Docker, Inc. “Install Docker Engine on Ubuntu.” Docker Documentation, https://docs.docker.com/install/linux/docker-ce/ubuntu/. Accessed 2022.

[18] Docker, Inc. “Install Docker Engine.” Docker Documentation, https://docs.docker.com/engine/install/#server. Accessed 2022.

[19] Docker, Inc. “Dockerfile reference.” Docker Documentation, https://docs.docker.com/engine/reference/builder/. Accessed 2022.

[20] Docker, Inc. “Start containers automatically.” *Docker Documentation*, https://docs.docker.com/config/containers/start-containers-automatically/. Accessed 2022.

[21] 1&1 Ionos. “Docker Compose y Swarm: gestión multicontenedor - IONOS.” Ionos, 9 July 2019, https://www.ionos.es/digitalguide/servidores/know-how/docker-compose-y-swarm-gestion-multicontenedor/. Accessed 2022.

[22] Docker, Inc. “Manage data in Docker.” Docker Documentation, https://docs.docker.com/storage/. Accessed 2022.

[23] Docker, Inc. “Use volumes.” Docker Documentation, https://docs.docker.com/storage/volumes/. Accessed 11 June 2022.

[24] Rajagopal, Vignesh Kumar. “Docker Volume vs Bind Mounts vs tmpfs mount.” Digital Varys, https://digitalvarys.com/docker-volume-vs-bind-mounts-vs-tmpfs-mount/. Accessed 2022.

[25] Docker, Inc. “Use bind mounts.” Docker Documentation, https://docs.docker.com/storage/bind-mounts/. Accessed 2022.

[26] Docker, Inc. “Use tmpfs mounts.” Docker Documentation, https://docs.docker.com/storage/tmpfs/. Accessed 2022

[27] Duan, Gary. “How Docker Swarm Container Networking Works – Under the Hood.” Blog NeuVector, 5 January 2017, https://blog.neuvector.com/article/docker-swarm-container-networking. Accessed 2022

Universidad de Alcalá
Escuela Politécnica
Superior

ESCUELA POLITECNICA
SUPERIOR