Virtualización Ligera en Sistemas Embebidos (Docker sobre Raspberry Pi)

Julen Aristimuño Arana

Índice general

Índice de figuras

1 Docker

Docker está en boca de todos y la impresión generalizada es que 2015 va a ser el año de su despegue definitivo y que los contenedores han llegado para quedarse.

Los contenedores no son una tecnología nueva pero Docker ha reunido las características necesarias para hacerla sencilla y popular en Linux. Suponen un cambio en la infraestructura de las aplicaciones con algunas ventajas sobre la virtualización y la instalación de los servicios directamente en el sistema.

1.1. ¿Qué es Docker?

Docker es un proyecto de código abierto con el que fácilmente podremos crear contenedores. Estos contenedores de Docker podríamos definirlos como máquinas virtuales ligeras, menos exigentes con los chips y memorias de los equipos donde se ejecutarán.

Docker se compone de tres elementos fundamentales:

Contenedores Docker:
Son como un directorio, contienen todo lo necesario para que una aplicación pueda funcionar sin necesidad de acceder a un repositorio externo al contenedor. Cada uno de éstos es una plataforma de aplicaciones segura y aislada del resto que podamos encontrar o desplegar en la misma máquina host.

Imágenes Docker:
La imagen Docker podríamos entenderla como un SO con aplicaciones instaladas. Sobre esta base podremos empezar a añadir aplicaciones que vayamos a necesitar en otro equipo donde tengamos intención de usar la imagen. Además Docker nos ofrece una forma muy sencilla de actualizar las imágenes que tengamos creadas, así como un sencillo método para crear nuevas imágenes.

Repositorios Docker:
También conocidos como Registros Docker, contienen imágenes creadas por los

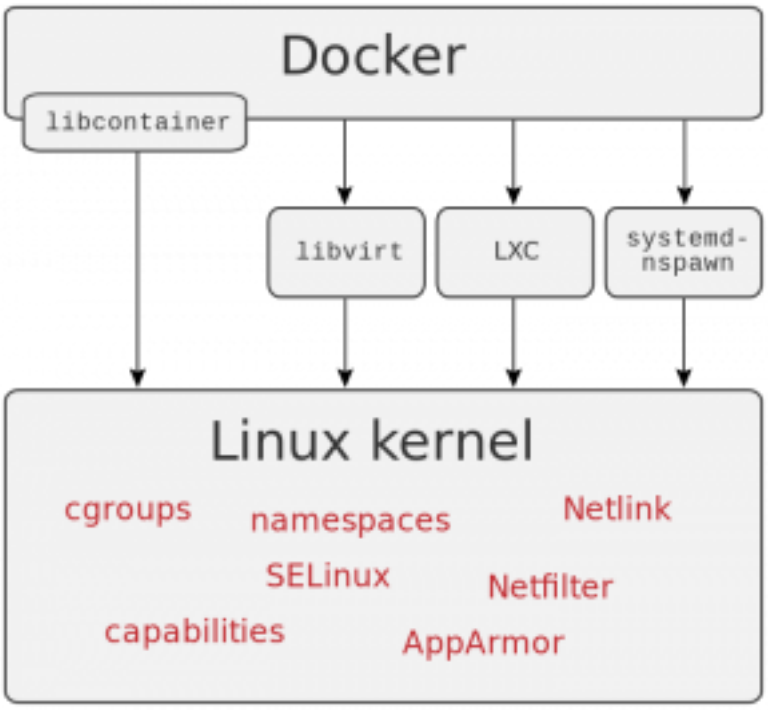

Figura 1.1: Esquema básico de Docker

usuarios y puestas a disposición del público. Podemos encontrar repositorios públicos y totalmente gratuitos o repositorios privados donde tendremos que comprar las imágenes que necesitemos. Estos registros permiten desarrollar o desplegar aplicaciones de forma simple y rápida en base a plantillas, reduciendo el tiempo de creación o implementación de aplicaciones o sistemas.

Docker implementa una API de alto nivel, para así proporcionar contenedores ligeros que ejecutan los procesos de manera aislada, basándose en el núcleo o kernel del sistema. Se basa en la funcionalidad del núcleo y utiliza aislamiento de recursos (CPU, memoria, etcétera) y los espacios de nombres por separado, para aislar así la aplicación. Otra características importante es que accede a la virtualización del kernel de linux, a través de la biblioteca libcontainer , indectamente a través de libvirt, LXC o nspawn systemd.

Mediante este sistema los recursos pueden ser aislados, los servicios restringidos. Los contenedores múltiples pueden compartir el mismo núcleo, pero se puede limitar la cantidad de su uso (CPU, memoria y E/S)1.1.

1.1.1. El concepto de Contenedor

Un contenedor es simplemente un proceso para el sistema operativo que, internamente, contiene la aplicación que queremos ejecutar y todas sus dependencias. La aplicación contenida solamente tiene visibilidad sobre el sistema de

ficheros virtual del contenedor y utiliza indirectamente el kernel del sistema operativo principal para ejecutarse.

Podemos trazar un paralelismo entre el contenedor y una máquina virtual: ambos son sistemas autocontenidos que, en realidad, utilizan un sistema superior para ejecutar sus trabajos. La gran diferencia es que una máquina virtual necesita contener todo el sistema operativo mientras que un contenedor aprovecha el sistema operativo sobre el cual se ejecuta.

1.2. ¿Cómo funciona Docker?

En un principio contamos con una imagen base, sobre la que realizaremos los diferentes cambios. Tras confirmar estos cambios mediante la aplicación Docker, crearemos la imagen que usaremos. Esta imagen contiene únicamente las diferencias que hemos añadido con respecto a la base. Cada vez que queramos ejecutar esta imagen necesitaremos la base y las 'capas' de la imagen. Docker se encargará de acoplar la base, la imagen y las diferentes capas con los cambios para darnos el entorno que queremos desplegar para empezar a trabajar.

1.3. Características de Docker

Docker tiene varias características interesantes. Es ligero ya que no hay virtualización aprovechándose mejor el hardware y únicamente necesitando el sistema de archivos mínimo para que funcionen los servicios.

Los contenedores son autosuficientes (aunque pueden depender de otros contenedores, por ejemplo, un wordpress que necesita una base de datos mysql) no necesitando nada más que la imagen del contenedor para que funcionen los servicios que ofrece.

Las imágenes de docker son portables entre diferentes plataformas el único requisito es que en el sistema huésped esté disponible docker.

Es seguro, pudiendo hacer que los contenedores se comuniquen por un túnel solo disponible para ellos, los contenedores están aislados en el sistema mediante namespaces y control groups.

1.4. Ventajas de Docker

1. Podemos disponer de un entorno de desarrollo (devbox) o servicio en varios minutos/horas en vez de algún día. Esto es así porque la configuración y los servicios necesarios están automatizados en la construcción de las imágenes de los contenedores mediante Dockerfiles.

2. Al estar los servicios en contenedores no hace falta instalarlos en la máquina en la que son alojados, de forma que podemos disponer de los servicios y

después eliminarlos de forma sencilla sin "ensuciar" el sistema huésped.

3. Nos permite tener versiones más parecidas o iguales a las usadas en producción. Por ejemplo, en Arch Linux nos permite tener un mysql de la distribución Ubuntu usando la misma versión.

1.5. Instalación de Docker

En mi caso, cuento con un portátil con un sistema operativo anfitrión Windows 10, sobre este sistema he instalado el software VirtualBox. Dentro del VirtualBox he creado una máquina cuyo sistema operativo es un Ubuntu Server 14.04 y aquí será donde voy a instalar Docker.

Instalar Docker es muy sencillo, únicamente tenemos que seguir estos pasos:

1. Comprobar si tenemos instalado Curl en nuestro sistema:

```
$ which curl
```

2. Si no está instalado seguimos las siguientes instrucciones para instalarlo:

```
$ sudo apt-get update
$ sudo apt-get install curl
```

3. Instalamos Docker ejecutando el siguiente comando:

```
$ curl -sSL https://get.docker.com/ | sh
```

4. Comprobamos que Docker se ha instalado correctamente ejecutando el comando:

```
$ docker --version
```

El comando tiene que devolvernos la versión de docker que hemos instalado que será la última versión disponible

1.5.1. Problemas con las versiones de Docker

Durante el desarrollo del proyecto tuve un problema con las versiones de Docker. Al inicio del proyecto instalé la versión de Docker 1.8, con esta versión realicé las primeras pruebas e intenté realizar conexiones entre nodos. Al salir la versión de Docker 1.9 con nuevas funcionalidades que me hacían falta, intenté actualizar la versión de Docker.

Para actualizar Docker, básicamente tenía que borrar la versión actual y seguidamente, instalar la nueva. Pero al lanzar el comando para borrar la versión no me dejaba, la máquina me decía que no podía borrar Docker. Después de buscar información en diferentes páginas de Internet, probar mil formas... Todas fallaban. La solución final fué la de instalar una nueva máquina virtual igual que

la anterior y volver a instalar Docker desde 0. Ahora si con la versió de Docker 1.9.1 estaba listo para continuar con el proyecto y centrarme en las conexiones entre los nodos.

1.6. Primeros pasos con Docker

Existe gran cantidad de comandos que podemos utilizar dentro docker, ahora voy a proceder a explicar los más importantes.

El primer comando que voy a exponer será el comando **run** , ese comando permite crear y ejecutar un contenedor. El comando **run** permite pasarle una serie de parámetros con los que le podemos asignar memoria, cpu, variables de entorno, servidor DNS, etc...

```
$ docker run -i -t ubuntu /bin/bash
```

Pero, ¿Cómo funciona en realidad e comando **run**?

1. Comprueba que tenemos la imagen solicitada en nuestro entorno local.
1.1 Si la tenemos, la utiliza.
1.2 Si no la tenemos, la descarga del respositorio Docker Hub.

2. Crea un nuevo contenedor.

3. Le asigna espacio para el sistema de archivos y monta / añade una capa de lectura y escritura a la imágen.

4. Crea y asigna una interfaz de red / puente que permite al contenedor la comunicación con el host local.

5. Detecta y concede una dirección IP disponible desde un grupo de IPs.

6. Ejecuta el comando especificado.

Tenemos que saber también que este comando admite unos parámetros y algunos son de gran importancia:

1. - **rm** : elimina el contenedor una vez haya finalizado su ejecución.

2. - **t** : asigna un emulador de terminal al comando.

3. - **i** . hace que la ejecución de los comandos sea interactiva, mostrando por pantalla los resultados de la ejecución del comando sobre el contenedor.

Después de explicar el comando **run**, toca centrarnos en el comando **images**

```
$ docker images
```

Este comando nos permite ver las imagenes qué tenemos ejecutandose en nuestra máquina.

Otro comando importante es el comando **ps**

```
$ docker ps -a
```

Este comando sin el parámetro -a nos permite ver las instancias de los contenedores iniciadas. Con el parámetro -a podemos ver las iniciadas y las no iniciadas.

Con los dos comandos anteriores podemos ver qué tenemos en ejecución en nuestra máquina. Pero, ¿Podemos borrar estas instancias o imágenes?

Tenemos dos comandos que sirven para responder a la pregunta, el comando - **rm** y el comando **-rmi**

```
$ docker rm <contenedor_id>
```

```
$ docker rmi <imagen_id>
```

El primer comando valdrá para borrar instancias de contenedores, y el segundo para borrar las imágenes.

Seguimos con estudio de los comandos más importantes, el comando **start** me permite arrancar un contenedor que estaba parado.

```
$ docker start -a <contenedor_id>
```

Para terminar, hablaré sobre el comando **attach**. Este comando permite que nos volvamos a adjuntar a un conteneor que esté en marcha.

```
$ docker attach <contenedor_id>
```

1.6.1. Imágenes en Docker

Una imagen es una especie de plantilla, una captura del estado de un contenedor. Ya comenté que un contenedor no es una máquina virtual, pero para que te hagas una idea, podríamos decir que una imagen de un contenedor es como un snapshot de una máquina virtual, pero mucho más ligero.

Las imágenes se utilizan para crear contenedores, y nunca cambian.

Hay muchas imágenes públicas con elementos básicos como Java, Ubuntu, Apache... etc, que se pueden descargar y utilizar. Normalmente cuando creas imágenes, partimos de una imagen padre a la que le vamos añadiendo cosas.

Las imágenes se identifican por un ID, y un par nombre-versión.

La mayoría de las imágenes base se encuentran dentro del registro público **Docker Hub**. La que yo he utilizado para desarrollar este proyecto es la imagen osrf/ros:indigo-desktop-full. Para encontrarla, basta con entrar en la web de DockerHub, introducir ros en el buscador y entrar dentro del bloqueosrf/ros. Después de hacer esto basta con hacer un pull de la imagen dentro de Docker, lo explicaré más adelante.

1.6.2. El concepto de DockerFile

Es un archivo de configuración que se utiliza para crear imágenes. En dicho archivo indicamos qué es lo que queremos que tenga la imagen, y los distintos comandos para instalar las herramientas.

Los DockerFiles tienen algunas instrucciones:

FROM: indica la imagen base a partir de la cual crearemos la imagen que construirá el Dockerfile.

MAINTAINER: documenta el creador de la imagen.

RUN: permite ejecutar una instrucción en el contenedor, por ejemplo, para instalar algún paquete mediante el gestor de paquetes (apt-get, yum, rpm, ...).

A mi en este proyecto, especialmente me interesa una imagen de el sistema operativo ROS, que será el que utilizaré para crear mis nodos.

2 ROS

ROS (Robot Operating System) provee librerías y herramientas para ayudar a los desarrolladores de software a crear aplicaciones para robots. ROS provee abstracción de hardware, controladores de dispositivos, librerías, herramientas de visualización, comunicación por mensajes, administración de paquetes y más. ROS está bajo la licencia open source, BSD.

2.1. ¿Qué es ROS?

ROS (Robot Operative System) es un meta sistema operativo de código abierto mantenido por la Open Source Robotics Foundation (OSRF). Proporciona los servicios que caben esperar de un sistema operativo incluyendo las abstracciones de hardware, control de dispositivos a bajo nivel, implementación de utilizades comunes, paso de mensajes entre procesos y gestión de paquetes. Tambien proporciona herramientas y librerías para obtener, compilar, escribir y ejecutar código a través de multiples ordenadores. ROS se compone de un número de nodos independientes, cada nodo se comunica con el resto de nodos utilizando el modelo publicador/subscriptor.

2.2. Conceptos básicos de ROS

2.2.1. Master

El ROS Master proporciona el registro de nombre y consulta el resto del grafo de computación. Sin el maestro, los nodos no serían capaces de encontrar al resto de nodos, intercambiar mensajes o invocar servicios.

2.2.2. Nodos

Los nodos son procesos que realizan la computación. ROS está diseñado para ser modular en una escala de grano fino; un sistema de control del robot comprende por lo general muchos nodos. Por ejemplo, un nodo controla un láser, un nodo controla los motores de las ruedas, un nodo lleva a cabo la localización,

un nodo realiza planificación de trayectoria, un nodo proporciona una vista gráfica del sistema, y así sucesivamente. Un nodo de ROS se escribe con el uso de las librerías cliente de ROS, roscpp y rospy.

2.2.3. Mensajes

Los nodos se comunican mediante el paso de mensajes. Un mensajes es una estructura simple con campas tipados. Los tipos primitivos de datos (integer, floating point, boolean, etc.) están soportados, como los arrays de tipo primitivo. Los mensajes pueden incluir estructuras y arrays (como las estructuras de C).

2.2.4. Tópicos

Los mensajes son enrutados por un sistema de transpote que utiliza semántica publicardor/subscriptor. Un nodo envía mensajes publicando en un tópico. El tópico es un nombre que se usa para identificar el contenido del mensajes. Un nodo que esta interesado en cierto tipo de datos se subscribirá al tópico apropiado. Pueden existir muchos publicadores concurrentemente para un solo tópico y un nodo puede publicar y/o subscribirse a múltiples tópicos.

2.3. Crear un ROS Package

La aplicación que creemos dentro de ROS será un paquete instalable. ROS dispone de algunas herramientas para crear y compilar los paquetes. Yo utilizaré la herramiento **catkin**. El paquete catkin poseerá todo el código que vayamos desarrollando.

Para que un paquete sea considerado como un paquete catkin tiene que cumplir una serie de requisitos:

1. El paquete debe contener un fichero catkin **packkage.xml**, este fichero contendrá la meta información sobre el paquete

2. El paquete debe contener un fichero **CMakeLists.txt**, este archivo será el que use catkin a la hora de compilar el paquete.

3. No puede haber más de un paquete por carpeta.

Después de explicar estos conceptos nos toca crear el paquete. Para crear un paquete catkin lo primero que tenemos que hacer es crear un workspace de catkin que será quien contenga los paquetes. Los comandos que debemos utilizar son los siguientes

```
$ mkdir -p ~/catkin_ws/src
$ cd ~catkin_ws/src
$ catkin_init_workspace
$ cd ~catkin_ws/
$ source devel/setup.bash
```

Con esto tendríamos creado y compilado nuestro workspace de catkin. Ahora para crear el paquete introducimos los siguientes comandos.

```
$ cd ~catkin_ws/src
$ catkin_create_pkg paqueteCatkin std_msgs rospy roscpp
```

Hemos creado un paquete llamado paqueteCatkin. También le hemos indicado que nuestro paquete va a tener dependencias para compilar código fuente de Python y de C++

Por último, para compilar el paquete necesitamos el comando **catkin-make**. Este comando se fija en nuestro archivo CMakeLists.txt para saber cómo tiene que compilar el paquete y lo compila.

```
$ cd ~catkin_ws/
$ catkin_make
```

2.4. Comandos básicos de ROS

El primer comando es **roscore**. Este comando ejecuta todo lo necesario para que dar soporte de ejecución al sistema completo de ROS. Siempre tiene que estar ejecutándose para permitir que se comuniquen los nodos. Permite ejecutarse en un determinado puerto.

Otro comando importante es **rosrun**. Permite ejecutar cualquier aplicación de un paquete sin necesidad de cambiar a su directorio.Podemos pasarle una serie e parámetros.

El tercer comando es **rosnode**. Este comando nos proporciona información sobre un nodo.

Por último tenemos **roswtf**. Permite chequear si algo va mal. Ejecutamos roscd y después roswtf.

3

Desarrollo principal del proyecto

El objetivo final de nuestro proyecto es el de controlar un robot motorizado mediante un sistema virtualizado. El sistema estará compuesto por diferentes elementos que se comunicarán entre sí para lograr el objetivo. Utilizaremos Docker y ROS como software y el sistema que desarrollaremos lo integraremos dentro de una Raspberry PI.

3.1. Redes en Docker

Como ya sabemos, Docker nos permite crear multitud de contenedores que funcionan simultáneamente, es probable que a la hora de desarrollar un sistema con Docker nos interese que estos contenedores se comuniquen entre sí y ejecuten tareas por separado o compartan datos entre ellos. En concreto nuestro sistema va a contar con un intercambio de mensajes ente los diferentes contenedores ROS. Por lo tanto, tenemos la necesidad de configurar una red efectiva para la comunicación entre los contenedores que necesitemos.

3.1.1. Docker0

Por sí mismo, Docker proporciona los fundamentos necesarios para la creación de redes y comunicación de contenedor-a-contenedor y de contenedor-a-host.

Cuando el proceso de Docker nace, configura una nueva interfaz puente virtual llamada docker0 en el sistema host. Esta interface permite a Docker crear una sub-red virtual para el uso de los contenedores que se ejecutarán. Este puente funciona como el punto o interfaz principal entre la creación de redes en el contenedor y el host.

Cuando un contenedor es inicializado por Docker, una nueva interfaz virtual se crea y se le proporciona una dirección en el rango de la sub-red. La dirección IP es asignada al intervalo de la red del contenedor, proporcionando a la red del contenedor una ruta al puente docker0 en el sistema host. Docker automáticamente configura las reglas en iptables que permitirán redirigir y configurar la

máscara NAT para el tráfico originado en la interfaz docker0 hacia el resto del mundo.

¿Cómo los contenedores exponen servicios a los consumidores?

Otros contenedores en el mismo host podrán acceder a los servicios proporcionados por sus vecinos sin configuraciones adicionales. El sistema host simplemente enruta las peticiones originadas por y destinadas a la interfaz docker0 a la ubicación adecuada.

Los contenedores pueden exponer sus puertos al host, donde es que estos reciben el tráfico redirigido hacia el mundo exterior. Los puertos expuestos pueden ser mapeados al sistema host, tan solo con seleccionar un puerto específico o dejando a Docker seleccionar un puerto al azar, alto y sin usar. Docker se encarga de todas la configuración en las reglas de redirección e iptables en estas situaciones.

3.1.2. Ping entre contenedores

Cuando tenemos un par de contenedores creados y se encuentran en la misma subred, obviamente, podemos efectuar una comunicación entre ellos. Hay diferentes métodos para comunicarse pero ahora vamos a centrarnos en el ping entre ellos.

El ping es una forma muy sencilla de probar la comunicación entre diferentes contenedores.

Para efectuar esta prueba vamos a arrancar dos contenedores, da igual que imagen tengan nuestros contenedores pero para especificar aún más nuestros contenedores contendrán la imagen osrf/ros:indigo-desktop.

El primer paso es crear contenedores de la siguiente forma:

```
$ docker run -it osrf/ros:indigo-desktop /bin/bash
```

Comprobamos que están las dos máquinas creadas y ejecutándose

```
$ docker ps
 CONTAINER ID          IMAGE                          COMMAND
                     CREATED
           STATUS          PORTS                          NAMES
7ca83c68acal           osrf/ros:indigo-desktop     "/ros_entrypoint
    .sh /"   18 seconds
ago        up 18 seconds                                  sad_poitras
878a8aa8e598           osrf/ros:indigo-desktop     "/ros_entrypoint
    .sh /"   45 seconds
ago        up 45 seconds                                  happy_mclean
```

Para efectuar esta comunicación necesitamos obtener la dirección IP de los contenedores y hace ping a la dirección IP del contenedor con el que queremos

conectar y viceversa. Para obtener la dirección IP del contenedor vamos a utilizar el comando inspect de Docker de la siguiente forma:

```
$ docker inspect --format='{{.NetworkSettings.IPAddress}}'
    sad_poitras
172.17.0.3
$ docker inspect --format='{{.NetworkSettings.IPAddress}}'
    happy_mclean
172.17.0.2
```

Una vez que hemos obtenido las direcciones IP que genera docker0 ahora basta con hacer ping entre los contenedores. Vamos a realizar la conexión desde el contenedor llamado sad_poitras al contenedor happy_mclean. Para ello basta con usar el comando ping desde dentro del contenedor sad_poitras de la siguiente forma:

```
root@878a8aa8e598:/# ping 172.17.0.3
PING 172.17.0.3 (172.17.0.3) 56(84) bytes of data.
64 bytes from 172.17.0.3: icmp_seq=1 ttl=64 time=0.079 ms
64 bytes from 172.17.0.3: icmp_seq=2 ttl=64 time=0.053 ms
64 bytes from 172.17.0.3: icmp_seq=3 ttl=64 time=0.057 ms
64 bytes from 172.17.0.3: icmp_seq=4 ttl=64 time=0.066 ms
64 bytes from 172.17.0.3: icmp_seq=5 ttl=64 time=0.054 ms
64 bytes from 172.17.0.3: icmp_seq=6 ttl=64 time=0.060 ms
64 bytes from 172.17.0.3: icmp_seq=7 ttl=64 time=0.056 ms
64 bytes from 172.17.0.3: icmp_seq=8 ttl=64 time=0.060 ms
64 bytes from 172.17.0.3: icmp_seq=9 ttl=64 time=0.054 ms
64 bytes from 172.17.0.3: icmp_seq=10 ttl=64 time=0.050 ms
64 bytes from 172.17.0.3: icmp_seq=11 ttl=64 time=0.054 ms
--- 172.17.0.3 ping statistics ---
11 packets transmitted, 11 received, 0% packets loss, time 9996ms
rtt min/avg/max/mdev = 0.050/0.058/0.079/0.010 ms
```

De la misma manera podemos hace un ping desde el contenedor happy_mclean al contenedor sad_poitras como vemos a continuación:

```
root@7ca83c68acal:/# ping 172.17.0.2
PING 172.17.0.2 (172.17.0.2) 56(84) bytes of data.
64 bytes from 172.17.0.2: icmp_seq=1 ttl=64 time=0.062 ms
64 bytes from 172.17.0.2: icmp_seq=2 ttl=64 time=0.052 ms
64 bytes from 172.17.0.2: icmp_seq=3 ttl=64 time=0.050 ms
64 bytes from 172.17.0.2: icmp_seq=4 ttl=64 time=0.053 ms
64 bytes from 172.17.0.2: icmp_seq=5 ttl=64 time=0.054 ms
--- 172.17.0.2 ping statistics ---
5 packets transmitted, 5 received, 0% packets loss, time 3998ms
rtt min/avg/max/mdev = 0.050/0.054/0.062/0.006 ms
```

Observamos en los dos ejemplos del ping que no hemos perdido ningún paquete, por lo que podemos asegurar que la conexión esta correctamente establecida.

Para terminar, he de añadir que no también podemos hacer ping entre los contenedores indicando el nombre del contenedor al que vamos a hacer el ping

en vez de su dirección IP, por eso tenemos que saber que los nombres de los contenedores han de ser únicos.

3.1.3. Link entre contenedores

Por temas de seguridad, vamos a necesitar que la conexión entre nuestros contenedores sea lo más privada posible. Como ya sabemos el uso del ping esta reducido a comprobar la conexión entre los contenedores dentro del docker0 al que se conectan todos los contenedores creados de manera automática. El link se utiliza para privatizar aún más la conexión entre diferentes contenedores.

Este link básicamente lo que hace es crear un puente virtual entre los contenedores que nosotros queremos comunicar y así esta comunicación se abstrae del resto de contenedores por lo que creamos una red"privada entre los contenedores linkados.

Para realizar la prueba vamos a crear un contenedor sencillo y seguidamente crearemos un segundo contenedor con el flag -link al contenedor primero.

Creamos el primer contenedor:

```
$ docker run -it --name cont1 osrf/ros:indigo-desktop /bin/bash
```

Ahora creamos el segundo contenedor con el link;

```
$ docker run -it --name cont2 --link cont1 osrf/ros:indigo-desktop
    /bin/bash
```

Para comprobar que todo ha salido correctamente, únicamente tenemos que observar los links que tiene nuestro contenedor con el comando inspect que hemos mencionado anteriormente. Al ejecutar el comando se abrirá dentro del documento un JSON con todos los datos y dentro de este aparecerá un apartado llamado LINKS:

```
$ docker inspect -f "{{ .HostConfig.Links }}" cont2
"LINKS": ["/cont1:/cont2/cont1"]
```

3.1.4. Network en Docker

A mitad del desarrollo de nuestro proyecto, se lanzó una nueva versión de Docker, la versión 1.9. Esta versión traía consigo la nueva funcionalidad de crear todo tipo de redes de manera realmente sencilla sin ningún tipo de configuración. Esta nueva funcionalidad permite crear redes virtuales enteras no solo realizar conexiones entre contenedores como habíamos hecho hasta ahora.

Para poder utilizar esta funcionalidad es necesario utilizar el comando network dentro de Docker encargado de crear una red Single-Host.

Con el fin de probar esta nueva funcionalidad, vamos a crear una red y conectarle un par de contenedores para comprobar que funcionan correctamente.

```
$ docker network create red_docker
be76e3d58a0ffd12e7828d87eb7dbad24c065e7ebc2d19c793e3b13d73c5cb5b
```

Creamos ahora dos contenedores Docker:

```
$ docker run -it --name cont3 osrf/ros:indigo-desktop /bin/bash
root@c913f0114b24:/#
$ docker run -it --name cont4 osrf/ros:indigo-desktop /bin/bash
root@b82139eab9b3:/#
```

Después de crear los dos contenedores nos tocar conectarlos a la red que hemos creado, este paso lo podemos realizar de forma sencilla utilizando el comando connect que tiene network desde el Host:

```
$ docker network connect red_docker cont3
$ docker network connect red_docker cont4
```

Para comprobar que hemos realizado las conexiones correctamente vamos a realizar un ping entre los contenedores:

```
root@c913f0114b24:/# ping cont 4
PING cont4 (172.17.0.3) 56(84) bytes of data.
64 bytes from 172.17.0.3: icmp_seq=1 ttl=64 time=0.079 ms
64 bytes from 172.17.0.3: icmp_seq=2 ttl=64 time=0.054 ms
64 bytes from 172.17.0.3: icmp_seq=3 ttl=64 time=0.059 ms
64 bytes from 172.17.0.3: icmp_seq=4 ttl=64 time=0.053 ms
--- con4 ping statistics ---
4 packets transmitted, 4 received, 0% packets loss, time 2998ms
rtt min/avg/max/mdev = 0.053/0.061/0.079/0.011 ms
```

```
root@b82139eab9b3:/# ping cont 3
PING cont3 (172.17.0.2) 56(84) bytes of data.
64 bytes from 172.17.0.2: icmp_seq=1 ttl=64 time=0.056 ms
64 bytes from 172.17.0.2: icmp_seq=2 ttl=64 time=0.050 ms
64 bytes from 172.17.0.2: icmp_seq=3 ttl=64 time=0.054 ms
64 bytes from 172.17.0.2: icmp_seq=4 ttl=64 time=0.050 ms
--- con3 ping statistics ---
4 packets transmitted, 4 received, 0% packets loss, time 2997ms
rtt min/avg/max/mdev = 0.050/0.052/0.056/0.007 ms
```

No se ha perdido ningún paquete por lo que la conexión esta efectuada de forma correcta.

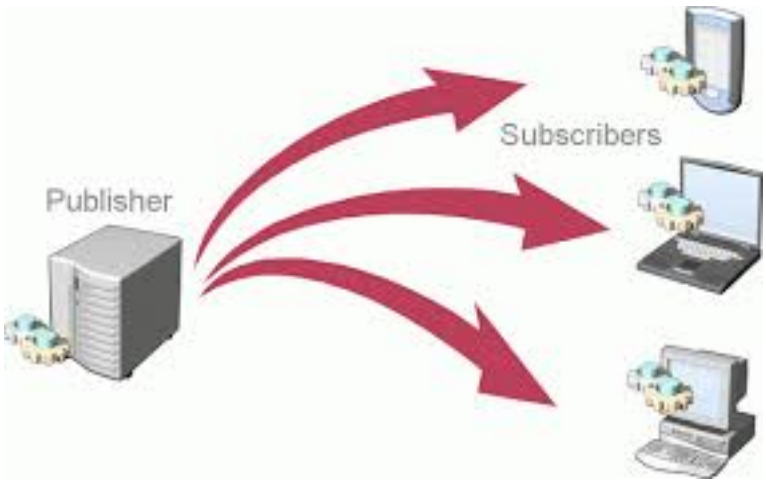

Figura 3.1: Modelo Publisher-Subscriber

3.1.5. Otras conexiones

Debido a la falta de tiempo no he podido profundizar más sobre otras conexiones pero en este documento voy a mencionar de manera sencilla otro tipo de conexiones que podemos efectuar.

Redes Multi-Host

En esta última versión de Docker como hemos mencionado anteriormente se ha impulsado mucho la conexión entre contenedores y entre ellos se ha lanzado el concepto de Muli-Host network.

Dentro de estas redes la más famosa es la Swarm. DockerSwarm es una herramienta que provee Docker que sirve para crear clusters de con-tenedores Docker.

3.2. Modelo Publisher-Subscriber

Este modelo de redes es el que vamos a utilizar para desarrollar la comunicación entre nuestros nodos.

El modelo publicado-suscriptor es muy sencillo de entender. Uno de nuestros nodos va a actuar como nodo publicador, como su nombre indica este nodo va a publicar mensajes dentro de su red que será recibidos o escuchados por los diferentes nodos suscriptores. Estos nodos suscriptores tienen la obligación de suscribirse al nodo publicador para recibir sus mensajes, si no lo hacen los mensajes del nodo publicador nunca llegarán.3.1

4 Ejemplo conexión de nodos ROS

Después de asimilar todos los contenidos mencionados hasta ahora, ha llegado el momento de ejecutar la prueba de conexión de nodos ROS dentro de Docker utilizando el modelo distribuido de Publisher-Subscriber.

En esta prueba crearemos un nodo publicador que enviará mensajes que serán recibidos por los suscriptores que previamente se habrán suscrito al tópico del nodo publicador. Para realizar este ejemplo vamos a apoyarnos en la página de wiki.ros.

4.1. Publisher

```
#include "ros/ros.h"
#include "std_msgs/String.h"

#include <sstream>

intmain(intargc,char**argv)
{
  ros::init(argc, argv, "talker");
  ros::NodeHandle n;
  ros::Publisher chatter_pub = n.advertise<std_msgs::String>("
      chatter",1000);
  ros::Rate loop_rate(10);

  intcount = 0;
  while(ros::ok())
  {
     std_msgs::String msg;
     std::stringstream ss;
     ss << "Hola (Mensaje numero: " << count << ")";
     msg.data = ss.str();
     ROS_INFO("%s", msg.data.c_str());
     chatter_pub.publish(msg);
     ros::spinOnce();
     loop_rate.sleep();
     ++count;
```

```
  }
  return0;
}
```

4.2. Subscriber

```
#include "ros/ros.h"
#include "std_msgs/String.h"

voidchatterCallback(conststd_msgs::String::ConstPtr& msg)
{
   ROS_INFO("I heard: [%s]", msg->data.c_str());
}

intmain(intargc,char**argv)
{
  ros::init(argc, argv, "listener");
  ros::NodeHandle n;
  ros::Subscriber sub = n.subscribe("chatter", 1000,
      chatterCallback);
  ros::spin();
  return0;
}
```

4.3. CMakeLists

```
cmake_minimum_required(VERSION 2.8.3)
project(beginner_tutorials)

find_package(catkin REQUIRED COMPONENTS roscpp rospy std_msgs
    genmsg)

add_message_files(FILES Num.msg)
add_service_files(FILES AddTwoInts.srv)

generate_messages(DEPENDENCIES std_msgs)

catkin_package()

include_directories(include ${catkin_INCLUDE_DIRS})

add_executable(talker src/talker.cpp)
target_link_libraries(talker ${catkin_LIBRARIES})
add_dependencies(talker prueba_generate_messages_cpp)

add_executable(listener src/listener.cpp)
target_link_libraries(listener ${catkin_LIBRARIES})
add_dependencies(listener prueba_generate_messages_cpp)
```

4.4. Construcción del Package

Después de tener el código de nuestro sistema, ahora nos queda compilarlo y construirlo.

```
mkdir -p  /catkin_ws/src
cd /catkin_ws/src
/catkin_ws/src$ catkin_init_workspace
Creating symlink "/home/julen/catkin_ws/src/CMakeLists.txt"
    pointing to "/opt/ros/indigo/share/catkin/cmake/toplevel.cmake
    "
/catkin_ws/src$ cd ..
/catkin_ws$ source devel/setup.bash
bash: devel/setup.bash: No existe el archivo o el directorio
/catkin_ws$ cd src/
/catkin_ws/src$ catkin_create_pkg pruebaDocker std_msgs rospy
    roscpp
Created file pruebaDocker/package.xml
Created file pruebaDocker/CMakeLists.txt
Created folder pruebaDocker/include/prueba
Created folder pruebaDocker/src
Successfully created files in /home/julen/catkin_ws/src/
    pruebaDocker. Pleaseadjust the values in package.xml.
/catkin_ws/src$ ls
CMakeLists.txt  pruebaDocker
/catkin_ws/src$ cd ..
/catkin_ws$ catkin_make
Base path: /home/julen/catkin_ws
Source space: /home/julen/catkin_ws/src
Build space: /home/julen/catkin_ws/build
Devel space: /home/julen/catkin_ws/devel
Install space: /home/julen/catkin_ws/install
####
#### Running command: "cmake /home/julen/catkin_ws/src -
    DCATKIN_DEVEL_PREFIX=/home/julen/catkin_ws/devel -
        DCMAKE_INSTALL_PREFIX=/home/julen/catkin_ws/install -G
        Unix Makefiles" in "/home/julen/catkin_ws/build"
####
-- The C compiler identification is GNU 4.8.4
-- The CXX compiler identification is GNU 4.8.4
-- Check for working C compiler: /usr/bin/cc
-- Check for working C compiler: /usr/bin/cc -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Check for working CXX compiler: /usr/bin/c++
-- Check for working CXX compiler: /usr/bin/c++ -- works
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Using CATKIN_DEVEL_PREFIX: /home/julen/catkin_ws/devel
-- Using CMAKE_PREFIX_PATH: /opt/ros/indigo
-- This workspace overlays: /opt/ros/indigo
-- Found PythonInterp: /usr/bin/python (found version "2.7.6")
-- Using PYTHON_EXECUTABLE: /usr/bin/python
-- Using Debian Python package layout
-- Using empy: /usr/bin/empy
-- Using CATKIN_ENABLE_TESTING: ON
-- Call enable_testing()
-- Using CATKIN_TEST_RESULTS_DIR: /home/julen/catkin_ws/build/
    test_results
-- Looking for include file pthread.h
```

```
-- Looking for include file pthread.h - found
-- Looking for pthread_create
-- Looking for pthread_create - not found
-- Looking for pthread_create in pthreads
-- Looking for pthread_create in pthreads - not found
-- Looking for pthread_create in pthread
-- Looking for pthread_create in pthread - found
-- Found Threads: TRUE
-- Found gtest sources under '/usr/src/gtest': gtests will be
    built
-- Using Python nosetests: /usr/bin/nosetests-2.7
-- catkin 0.6.14
-- BUILD_SHARED_LIBS is on
-- traversing 1 packages in topological order:
-- - pruebaDocker
-- +++ processing catkin package: 'pruebaROS'
-- ==> add_subdirectory(pruebaROS)
-- Configuring done
-- Generating done
-- Build files have been written to: /home/julen/catkin_ws/build
####
#### Running command: "make -j1 -l1" in "/home/julen/catkin_ws/
    build"
####
/catkin_ws$ ls
build  devel  src
/catkin_ws$ cd src
/catkin_ws/src$ ls
CMakeLists.txt  pruebaDocker
/catkin_ws/src$ cd pruebaDocker/
/catkin_ws/src/pruebaDocker$ ls
CMakeLists.txt  include  package.xml  src
/catkin_ws/src/prueba$ cd src
/catkin_ws/src/prueba/src$ ls
/catkin_ws/src/prueba/src$ touch talker.cpp
/catkin_ws/src/prueba/src$ touch listener.cpp
/catkin_ws/src/prueba/src$ ls
listener.cpp  talker.cpp
```

Con este proceso hemos conseguido crear todos los archivos, ahora toca meter el código que hemos realizado dentro de su archivo mediante nano.

```
/catkin_ws/src/pruebaDocker/src$ nano listener.cpp
/catkin_ws/src/pruebaDocker/src$ nano talker.cpp
/catkin_ws/src/pruebaDocker/src$ cd ..
/catkin_ws/src/pruebaDocker$ ls
CMakeLists.txt  include  package.xml  src
/catkin_ws/src/pruebaDocker$ echo "" > CMakeLists.txt
/catkin_ws/src/pruebaDocker$ nano CMakeLists.txt
```

Ahora toca compilarlos:

```
/catkin_ws/src/pruebaDocker$ cd ..
/catkin_ws/src$ cd ..
/catkin_ws$ ls
build  devel  src
/catkin_ws$ catkin_make
/catkin_ws$ catkin_make
```

```
Base path: /home/julen/catkin_ws
Source space: /home/julen/catkin_ws/src
Build space: /home/julen/catkin_ws/build
Devel space: /home/julen/catkin_ws/devel
Install space: /home/julen/catkin_ws/install
####
#### Running command: "cmake /home/julen/catkin_ws/src -
    DCATKIN_DEVEL_PREFIX=/home/julen/catkin_ws/devel -
    DCMAKE_INSTALL_PREFIX=/home/julen/catkin_ws/install -G Unix
    Makefiles" in "/home/julen/catkin_ws/build"
####
-- Using CATKIN_DEVEL_PREFIX: /home/julen/catkin_ws/devel
-- Using CMAKE_PREFIX_PATH: /home/julenr/catkin_ws/devel;/opt/ros/
    indigo
-- This workspace overlays: /home/julen/catkin_ws/devel;/opt/ros/
    indigo
-- Using PYTHON_EXECUTABLE: /usr/bin/python
-- Using Debian Python package layout
-- Using empy: /usr/bin/empy
-- Using CATKIN_ENABLE_TESTING: ON
-- Call enable_testing()
-- Using CATKIN_TEST_RESULTS_DIR: /home/julen/catkin_ws/build/
    test_results
-- Found gtest sources under '/usr/src/gtest': gtests will be
    built
-- Using Python nosetests: /usr/bin/nosetests-2.7
-- catkin 0.6.14
-- BUILD_SHARED_LIBS is on
-- traversing 1 packages in topological order:
--  - pruebaDocker
-- +++ processing catkin package: 'pruebaDocker'
-- ==> add_subdirectory(pruebaDocker)
-- Configuring done
-- Generating done
-- Build files have been written to: /home/julen/catkin_ws/build
####
#### Running command: "make -j1 -l1" in "/home/julen/catkin_ws/
    build"
####
Scanning dependencies of target listener
[ 50%] Building CXX object pruebaDocker/CMakeFiles/listener.dir/
    src/listener.cpp.o
Linking CXX executable /home/julen/catkin_ws/devel/lib/
    pruebaDocker/listener
[ 50%] Built target listener
Scanning dependencies of target talker
[100%] Building CXX object pruebaDocker/CMakeFiles/talker.dir/src/
    talker.cpp.o
Linking CXX executable /home/julen/catkin_ws/devel/lib/
    pruebaDocker/talker
[100%] Built target talker
```

Después de terminar todo este proceso quedan generados dos ejecutables el talker y el listener.

4.5. Prueba del Sistema

Toca ahora probar el correcto funcionamiento de todo lo que hemos programado hasta ahora.

Primero tenemos que lanzar el roscore:

```
$ roscore
```

Seguidamente compilamos el workspace:

```
$ cd  /catkin_ws
$ . devel/setup.bash
$ catkin_make
```

Ahora procedemos a lanzar el listener:

```
$ rosrun pruebaDocker listener
```

Para finalizar, mandamos un mensaje desde el talker:

```
$ rostopic pub -1 /chatter std_msgs/String Recibe este mensaje
publishing and latching message for 5.0 seconds
```

```
$ rosrun pruebaDocker listener
[ INFO] 21347573621.39624565]: I heard: [Recibe este mensaje]
```

Podemos concluir que el listener ha recibido el mensaje de nuestro talker, por lo tanto el sistema es correcto.

5

Creación del sistema del proyecto

En esta pequeña introducción voy a mencionar todas los elementos que vamos a necesitar para crear nuestro proyecto. Para empezar cuento con un ordenador con Windows 10 como sistema anfitrión. Sobre mi entorno de Windows instalaré el software de VirtualBox. Dentro de este VirtualBox lanzaré una máquina con Ubuntu Server como SO. En este Ubuntu Server instalaré Docker. Cada uno de los contenedores que cree dentro de Docker se crearan con una imagen de Ubuntu que trae ROS instalado, estas imagenes serán sacadas del Docker Hub, osrf/ros:indigo-desktop. Para finalizar estos nodos se comunicarán entre ellos con el comando network del que hemos hablado anteriormente.

5.1. Crear los nodos y la red con Docker

Para empezar voy a crear tres contenedores ROS:

```
$ docker run --name nodo1 -it osrf/ros:indigo-desktop
$ docker run --name nodo2 -it osrf/ros:indigo-desktop
$ docker run --name nodo3 -it osrf/ros:indigo-desktop
```

Como hemos hecho anteriormente creamos una red y conectamos los tres nodos a ella:

```
$ docker network create redProyecto
139b6860a31b3b406e5a3ea3ca336410f35le13eea40cf50fc3b3d89c3ea956fb
$ docker network connect red nodo1
$ docker network connect red nodo2
$ docker network connect red nodo3
```

Para finalizar este proyecto ejecutamos lo anterior con los nodos ROS dentro de los contenedores de Docker. Tenemos que acordarnos que la variable ROS_MASTER_URI tiene que apuntar a ros0, que es la dirección del contenedor que ejecutara roscore.

Lanzamos el roscore en el nodo1:

```
root@ea663e8a74bf:/# roscore
```

Cambiamos en los otros dos nodos la variables comentada anteriormente:

```
$ ROS_MASTER_URI=http://nodo1:11311/
```

El nodo necesita encontrar los otros dos nodos, para ello necesitamos configurar los otros dos nodos, tenemos que poner la IP del contenedor en la variable ROS_IP. Lo haremos de la siguiente manera:

```
root@a9f0f2de874b: ~/catkin_ws# export ROS_IP=172.18.0.4
root@458148c51aaf: ~/catkin_ws# export ROS_IP=172.18.0.3
```

Con todo el ejemplo compilado dentro de los nodos 2 y 3 en el nodo 2 lanzamos el listener:

```
root@a9f0f2de874b: ~/# rosrun pruebaDocker listener
```

Para finalizar lanzamos un mensaje desde el talker:

```
root@458148c51aaf: ~/# rostopic pub -1 /chatter std_msgs/String
    Recibe este mensaje
```

```
root@a9f0f2de874b: ~/# rosrun pruebaDocker listener
[ INFO] [1729998535.563515736]: I heard: [Recibe este mensaje]
```

Todo el sistema implementado esta desarrollado de forma correcta.

6 Conclusiones

Durante la realización de este proyecto hemos trabajado con una herramienta de software llamado Docker, antes de empezar el proyecto no sabía ni de la existencia de la herramienta y tanto trabajar con ella como profundizar en ella me ha resultado muy enriquecedor debido a su gran potencial en un futuro. También he aprendido a utilizar y los conceptos básicos sobre el sistema operativo de robots llamado ROS.

La asignatura en sí era demasiado profunda para el poco tiempo que teníamos para dedicarle. Me ha resultado satisfactorio el aprendizaje autónomo, pero debido a la carga de trabajo de otras asignaturas más trabajos extra escolares no he podido dedicarle tanto tiempo como me gustaría. Pero me quedan muchos temas pendientes por resolver por iniciativa propia cuando disponga del tiempo que esto requiere.

En general estoy muy satisfecho con el trabajo realizado tanto por mí como por mis compañeros y como mi profesor.

Bibliografía

1. URL: https://docs.docker.com/installation/ubuntulinux/.
2. URL: http://codehero.co/como-instalar-y-usar-docker/.
3. URL: https://www.nessys.es/docker/
4. URL: https://www.digitalocean.com/community/tutorials/docker-explicado-como-crear-contenedores-de-docker-corriendo-en–es
5. URL: http://rm-rf.es/como-instalar-configurar-usar-docker-linux-containers/
6. URL: https://www.digitalocean.com/community/tutorials/el-ecosistema-de-docker-creacion-de-redes-y-comunicacion-es
7. URL: https://loquemeinteresadelared.wordpress.com/2015/12/14/crear-una-nueva-red-en-docker-y-conectarle-dos-nuevos-contenedores/
8. URL: http://blog.docker.com/
9. URL: https://hub.docker.com/search/?isAutomated=0&isOfficial=0&page=1&pullCount=0&q=ros
10. URL: https://docs.docker.com/engine/userguide/networking/
11. URL: https://docs.docker.com/engine/userguide/networking/dockernetworks/
12. URL: http://www.ros.org/
13. URL: http://wiki.ros.org/es
14. URL: http://wiki.ros.org/ROS/Installation
15. URL: http://wiki.ros.org/ROS/Tutorials
16. URL: http://wiki.ros.org/ROS/Tutorials/CreatingPackage
17. URL: http://wiki.ros.org/ROS/Tutorials/BuildingPackages
18. URL: http://wiki.ros.org/ROS/Tutorials/WritingPublisherSubscriber%28c%2B%2B%29
19. URL: https://en.wikipedia.org/wiki/Publishsubscribe_pattern
20. URL: https://es.wikipedia.org/wiki/Sistema_Operativo_Robótico
21. URL: https://es.wikipedia.org/wiki/Docker