

## Administración de Sistemas

# Virtualización ligera para sistemas embebidos con aplicaciones robóticas usando ROS y Docker



Ander Granado Masid

*17 de enero de 2016*

## Índice general

<b>I. Introducción</b>	<b>1</b>
<b>1. Objetivo</b>	<b>2</b>
<b>2. Herramientas utilizadas</b>	<b>3</b>
2.1. Hardware . . . . .	3
2.1.1. Raspberry Pi . . . . .	3
2.2. Software . . . . .	4
2.2.1. Docker . . . . .	5
2.2.1.1. Funcionamiento de Docker . . . . .	5
2.2.2. ROS . . . . .	5
<b>II. Docker</b>	<b>7</b>
<b>3. Introducción a Docker</b>	<b>8</b>
3.1. Instalación . . . . .	8
3.2. Uso básico de Docker . . . . .	9
3.2.1. <i>run</i> . . . . .	11
3.2.2. <i>ps</i> . . . . .	11
3.2.3. <i>inspect</i> . . . . .	12
3.2.4. <i>stop</i> y <i>kill</i> . . . . .	12
3.2.5. <i>rm</i> . . . . .	13
3.2.6. <i>images</i> y <i>rmi</i> . . . . .	13
3.3. Creación de Dockerfiles . . . . .	14
3.4. Profundizar en Docker . . . . .	16

<b>4. Redes en Docker</b>	<b>17</b>
4.1. Redes básicas en Docker	17
4.1.1. <i>docker0</i>	17
4.1.2. Prueba de conexión entre contenedores	18
4.1.2.1. <i>ping</i>	18
4.1.2.2. <i>netcat</i>	20
4.1.3. Links entre contenedores	21
4.2. Redes avanzadas con Docker	22
4.2.1. <i>network</i> de Docker	22
4.2.1.1. Prueba de <i>network</i>	22
4.2.2. Redes Multi-Host	24
4.2.2.1. Instalación de Swarm	24
4.2.2.2. Creación de un Swarm	25
4.2.2.3. Lanzar el <i>Swarm manager</i>	27
4.2.2.4. Configurar el Swarm y lanzar contenedores	29
4.3. Configuración manual de redes	32

### III. ROS 34

<b>5. Introducción a ROS</b>	<b>35</b>
5.1. Entorno de trabajo	35
5.2. Uso de ROS	36
5.3. Gestionar paquetes ROS	37
5.3.1. Crear paquetes	37
5.3.2. Compilar paquetes	38
5.4. Modelo distribuido Publisher-Subscriber	38
<b>6. Prueba con nodos ROS</b>	<b>39</b>
6.1. Código de la prueba	39
6.1.1. Publisher	39
6.1.2. Subscriber	40
6.1.3. CMakeLists	41
6.2. Construir el paquete	41
6.3. Ejecución	44

### IV. Implementación del sistema 46

<b>7. Creación del sistema</b>	<b>47</b>
7.1. Crear el sistema con Docker	49
7.2. Aplicaciones para el sistema	51
7.2.1. Visión artificial	51
7.2.1.1. Transmisión de imágenes de una Webcam	51

---

7.2.2. Recolección de información de sensores . . . . .	51
7.2.3. Recolección de información de sensores . . . . .	51
 <b>V. Conclusiones</b>	 <b>52</b>
 <b>8. Conclusiones</b>	 <b>53</b>
8.1. Sobre Docker y ROS . . . . .	53
8.2. Sobre la asignatura . . . . .	54

## Índice de figuras

4.1.	Imágenes GNU/Linux creadas . . . . .	27
4.2.	Imágenes GNU/Linux creadas para el Swarm . . . . .	29
5.1.	Modelo Publisher-Subscriber . . . . .	38
7.1.	Esquema del sistema en un ordenador x86 . . . . .	48
7.2.	Esquema del sistema en una Raspberry Pi . . . . .	49

# **Parte I.**

## **Introducción**

## Objetivo

En el siguiente documento tiene como objetivo desarrollar un sistema virtualizado para controlar un vehículo robótico. Este sistema dispondrá de diferentes módulos que estarán conectados entre sí e interactuarán entre ellos. Para desarrollarlo se hará uso de ROS y Docker. El sistema estará pensado para funcionar dentro de una Raspberry Pi. En el siguiente capítulo se explicará en profundidad todas las herramientas que usaremos para desarrollarlo, tanto de hardware como de software.

## Herramientas utilizadas

### 2.1. Hardware

Aunque el vehículo dispone de numeroso hardware, en esta sección solo hablaremos sobre el hardware para el cual nosotros vamos a programar. En este caso todo nuestro sistema se montará en una Raspberry pi, aunque el desarrollo del sistema lo haremos en los PCs con arquitectura x86.

#### 2.1.1. Raspberry Pi

Raspberry Pi es un ordenador de placa reducida que debido a su bajo coste (35 \$) y su pequeño tamaño, es ampliamente usado en sistemas de bajo coste, sistemas embebidos o en entornos educativos. Existen dos principales modelos, la Raspberry Pi y la Raspberry Pi 2. La Raspberry Pi a su vez cuenta con 4 diferentes submodelos, el A, el A+, el B y el B+.

Aunque cuenta con diferentes submodelos con diferentes especificaciones, las características generales de la Raspberry Pi son [Wikimedia Foundation Inc.(5 de Octubre de 2015)]:

- SoC (System on Chip) Broadcom BCM2835:
  - CPU ARM 1176JZF-S a 700 MHz single-core (familia ARM11)
  - GPU Broadcom VideoCore IV (OpenGL ES 2.0, MPEG-2 y VC-1, 1080p30 H.264/MPEG-4 AVC3)
- Memoria SDRAM: 256 MB (en el modelo A) o 512 MB (en el modelo B), compartidos con la GPU
- Puertos USB 2.0: 1 (en el modelo A), 2 (en el modelo B) o 4 (en el modelo B+)
- 10/100 Ethernet RJ-45 (en el Modelo B)



- Salidas de video:
  - Conector RCA (PAL y NTSC)
  - HDMI (rev1.3 y 1.4)
  - Interfaz DSI para panel LCD
- Salidas de audio:
  - Conector de 3.5 mm
  - HDMI
- Puertos GPIO: 8 o 17 (en el caso de las versiones +)

El segundo modelo de Raspberry Pi, conocido como Raspberry Pi 2, añade mejoras notables con respecto a la anterior generación. Sus características básicas son:

- SoC Broadcom BCM2836:
  - 900 MHz quad-core ARM Cortex A7
  - GPU Broadcom VideoCore IV (OpenGL ES 2.0, MPEG-2 y VC-1, 1080p30 H.264/MPEG-4 AVC3)
- 1GB memoria SDRAM, compartida con la GPU
- 4 puertos USB 2.0
- 10/100 Ethernet RJ-45
- Salidas de video:
  - Conector RCA (PAL y NTSC)
  - HDMI (rev1.3 y 1.4)
  - Interfaz DSI para panel LCD
- Salidas de audio:
  - Conector de 3.5 mm
  - HDMI
- 17 puertos GPIO

## 2.2. Software

Para lograr dicho objetivo anteriormente descrito, se hace uso de una serie de herramientas, entre las cuales se incluye Docker y ROS.

### 2.2.1. Docker

Docker es una plataforma abierta para aplicaciones distribuidas para desarrolladores y administradores de sistemas [Docker Inc.(29 de Septiembre de 2015)]. Docker automatiza el despliegue de contenidos de software proporcionando una capa adicional de abstracción y automatización de virtualización a nivel de sistema operativo en Linux [Wikimedia Foundation Inc.(29 de Septiembre de 2015a)]. Docker utiliza características de aislamiento de recursos del kernel de Linux,

#### 2.2.1.1. Funcionamiento de Docker

Docker se basa en el principio de los contenedores. Cada contenedor consta de una serie de aplicaciones y/o librerías que se ejecutan de manera independiente del OS (Sistema Operativo) principal, pero que usan el kernel Linux del sistema operativo anfitrión. Para hacer esto se hacen uso de diferentes técnicas tales como cgroups y espacios de nombres (namespaces) para permitir que estos contenedores independientes se ejecuten dentro de una sola instancia de Linux. De esta manera se logra reducir drásticamente el consumo de recursos de hardware, a cambio de que las librerías, aplicaciones o sistemas operativos deban ser compatibles con linux y ser compatibles con la arquitectura del hardware en la que se están ejecutando (x86, ARM, SPARC,...).

Mediante el uso de contenedores, los recursos pueden ser aislados, los servicios restringidos, y se otorga a los procesos la capacidad de tener una visión casi completamente privada del sistema operativo con su propio identificador de espacio de proceso, la estructura del sistema de archivos, y las interfaces de red. Los contenedores comparten el mismo kernel, pero cada contenedor puede ser restringido a utilizar sólo una cantidad definida de recursos como CPU, memoria y E/S. [Wikimedia Foundation Inc.(29 de Septiembre de 2015a)].

### 2.2.2. ROS

ROS (Robot Operating System) es un framework flexible para desarrollar software para robots. Es una colección de herramientas, librerías que tratan de simplificar la creación de aplicaciones complejas y robustas para todo tipo de sistemas robóticos [Open Source Robotics Foundation(29 de Septiembre de 2015)].

ROS provee los servicios estándar de un sistema operativo tales como abstracción del hardware, control de dispositivos de bajo nivel, implementación de funcionalidad de uso común, paso de mensajes entre procesos y mantenimiento de paquetes. Está basado en una arquitectura de grafos donde el procesamiento toma lugar en los nodos que pueden recibir, mandar y multiplexar mensajes de sensores, control, estados,

planificaciones y actuadores, entre otros [Wikimedia Foundation Inc.(29 de Septiembre de 2015b)].

Las áreas que incluye ROS son:

- Un nodo principal de coordinación.
- Publicación o subscripción de flujos de datos: imágenes, estéreo, láser, control, actuador, contacto, etc.
- Multiplexación de la información.
- Creación y destrucción de nodos.
- Los nodos están perfectamente distribuidos, permitiendo procesamiento distribuido en múltiples núcleos, multiprocesamiento, GPUs y clústeres. Login.
- Parámetros de servidor.
- Testeo de sistemas.

# **Parte II.**

## **Docker**

## Introducción a Docker

Tras haber explicado anteriormente a grandes rasgos el funcionamiento de Docker, es conveniente antes de lanzarnos a la creación del sistema saber como crear y configurar los contenedores de Docker que conformarán el sistema. En este capítulo se explicará como empezar a trabajar con Docker, desde instalarlo y como empezar a usarlo hasta como se usan los Dockerfiles para lanzar contenedores personalizados.

### 3.1. Instalación

Lo primero que haremos será instalar Docker en nuestro sistema. En nuestro caso tenemos un sistema Ubuntu instalado en una máquina virtual, ya que nuestro hardware cuenta con SO Windows. Instalar Docker en cualquier distribución basada en Debian es tan sencillo como seguir los siguientes pasos [Docker Inc.(10 de Octubre de 2015a)]:

1. Comprobar si tenemos curl instalado

```
$ which curl
```

En caso de que no este instalado, instalarlo mediante:

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

2. Instalar Docker mediante el siguiente comando:

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

3. Comprobar que Docker se ha instalado correctamente.

```
$ docker run hello-world
```

Si ejecutamos el comando anterior y nos muestra información sobre Docker, ya hemos terminado de instalar Docker.

## 3.2. Uso básico de Docker

Para hacer uso de Docker necesitamos trabajar desde la terminal. La forma básica para trabajar con Docker es la siguiente:

```
$ docker [subcomando de docker] [parametros]
```

De esa manera primero indicamos que queremos usar Docker y a continuación indicamos que es lo que queremos hacer. En el ejemplo anterior hemos usado *run* para ejecutar un contenedor de Docker. Por último introducimos los diferentes parámetros. La lista completa de comandos que acepta Docker se puede ver de la siguiente manera:

```
1 $ docker --help
2 Usage: docker [OPTIONS] COMMAND [arg...]
3 docker daemon [ --help | ... ]
4 docker [ --help | -v | --version ]
5
6 A self-sufficient runtime for containers.
7
8 Options:
9
10 --config=~/.docker           Location of client config files
11 -D, --debug=false           Enable debug mode
12 --disable-legacy-registry=false Do not contact legacy registries
13 -H, --host=[]              Daemon socket(s) to connect to
14 -h, --help=false           Print usage
15 -l, --log-level=info       Set the logging level
16 --tls=false                Use TLS; implied by --tlsverify
17 --tlscacert=~/.docker/ca.pem Trust certs signed only by this CA
18 --tlscert=~/.docker/cert.pem Path to TLS certificate file
19 --tlskey=~/.docker/key.pem  Path to TLS key file
20 --tlsverify=false          Use TLS and verify the remote
21 -v, --version=false        Print version information and quit
22
23 Commands:
24 attach    Attach to a running container
25 build     Build an image from a Dockerfile
26 commit    Create a new image from a container's changes
27 cp        Copy files/folders between a container and the local filesystem
28 create    Create a new container
```

```

29 diff      Inspect changes on a container's filesystem
30 events    Get real time events from the server
31 exec      Run a command in a running container
32 export    Export a container's filesystem as a tar archive
33 history   Show the history of an image
34 images    List images
35 import    Import the contents from a tarball to create a filesystem image
36 info      Display system-wide information
37 inspect   Return low-level information on a container or image
38 kill      Kill a running container
39 load      Load an image from a tar archive or STDIN
40 login     Register or log in to a Docker registry
41 logout    Log out from a Docker registry
42 logs      Fetch the logs of a container
43 network   Manage Docker networks
44 pause     Pause all processes within a container
45 port      List port mappings or a specific mapping for the CONTAINER
46 ps        List containers
47 pull      Pull an image or a repository from a registry
48 push      Push an image or a repository to a registry
49 rename    Rename a container
50 restart   Restart a container
51 rm        Remove one or more containers
52 rmi       Remove one or more images
53 run       Run a command in a new container
54 save      Save an image(s) to a tar archive
55 search    Search the Docker Hub for images
56 start     Start one or more stopped containers
57 stats     Display a live stream of container(s) resource usage statistics
58 stop      Stop a running container
59 tag       Tag an image into a repository
60 top       Display the running processes of a container
61 unpause   Unpause all processes within a container
62 version   Show the Docker version information
63 volume    Manage Docker volumes
64 wait      Block until a container stops, then print its exit code
65
66 Run 'docker COMMAND --help' for more information on a command.

```

Como se puede observar existen diferentes comandos que nos permitirán configurar Docker, obtener información sobre él y tratar con las imágenes y los contenedores de Docker. Durante todo este documento trabajaremos con la versión 1.9, liberada el 3 de noviembre de 2015.

```

$ docker --version
Docker version 1.9.0, build 76d6bc9

```

A continuación vamos a explicar algunos de ellos para poder empezar a trabajar con Docker.

### 3.2.1. *run*

El comando esencial para empezar a trabajar con Docker es el comando ***run***. Para poder lanzar directamente un contenedor de Docker, se usa el comando *run*.

```
$ docker run ubuntu:trusty
```

Con el comando anterior hemos lanzado un contenedor de Docker que lleva Ubuntu. Lo primero que hace Docker para lanzar una imagen es comprobar si ya tiene en local la imagen desde la que se va a crear el contenedor. En caso de no tenerla accederá a unos repositorios llamados Docker Hub, donde se encuentran una gran cantidad de ***Dockerfiles***. Los *Dockerfiles* son los archivos que sirven para generar esas imágenes (veremos más adelante como funcionan estos archivos especiales). Una vez Docker genere la imagen a partir del *Dockerfile* ejecutará el contenedor, que a grandes rasgos es una instancia de la imagen.

### 3.2.2. *ps*

Podremos observar los contenedores Docker que tenemos lanzados mediante el comando ***ps*** de Docker.

```
1 $ docker ps
2
```

CONTAINER ID	IMAGE	COMMAND	CREATED
	STATUS	PORTS	NAMES

Aunque hemos lanzado un contenedor, mediante el comando *ps* de Docker vemos que en realidad no hay ningún contenedor en ejecución. Si no le indicamos ningún parámetro al comando *ps*, solo nos mostrará los contenedores en ejecución. Para mostrar todos, se hace uso hay que indicárselo con *-a*.

Si nosotros hemos lanzado un contenedor, ¿Porqué no se está ejecutando? Esto es porque los contenedores de Docker solo se mantienen en ejecución mientras el comando con el que se han iniciado este activo [Docker Inc.(13 de Octubre de 2015b)].

Para poder probar el comportamiento por defecto del comando *ps*, vamos a crear un contenedor demonizado, un contenedor que se ejecutará indefinidamente hasta que lo paremos. Esto es muy habitual en Docker, ya que las aplicaciones hechas con Docker suelen diseñarse para funcionar 24/7, como por ejemplo servidores web. Lo haremos de la siguiente manera.

```
$ docker run -d ubuntu:14.04 /bin/sh -c "while true; do echo hello world;
sleep 1; done"
```



Con el **-d** lo que logramos es que se siga ejecutando el contenedor en segundo plano, como si fuera un demonio (daemon), un proceso que esta siempre en ejecución. También debemos darle algo para hacer, ya que si no, tal y como hemos comentado antes, finalizará la ejecución. Esto lo logramos mediante un bucle infinito en shell script, que es lo que pasamos como parámetro entre comillas. Si ahora ejecutamos el comando *ps*, comprobaremos que tenemos una máquina en ejecución.

```
1 $ docker ps
2 CONTAINER ID          IMAGE          COMMAND          NAMES          CREATED
3 40f5c913912e         ubuntu        "/bin/sh -c 'while tr" 2 seconds
   ago             Up 2 seconds          adoring_euclid
```

### 3.2.3. *inspect*

En caso de que queramos obtener más información sobre un contenedor de Docker, podemos usar el comando ***inspect*** de Docker. La forma más básica de trabajar con este comando es la de utilizar como parámetro el ID o nombre de la máquina. Este comando nos devolverá por la salida estándar un JSON con una gran cantidad de parámetros que nos indican diferentes aspectos sobre el contenedor. También se pueden obtener solo un parámetro o un grupo de parámetros en concreto. En el siguiente ejemplo se muestra su uso, para obtener toda la información y para obtener un dato, en este caso la dirección IP del contenedor.

```
1 $ docker inspect adoring_euclid
2 # ...
3 # ... Se omite la salida por ser demasiado grande
4 # ...
5 $ docker inspect --format='{{.NetworkSettings.IPAddress}}' adoring_euclid
6 172.17.0.3
```

### 3.2.4. *stop y kill*

En caso de que queramos matar un contenedor, podremos hacerlo mediante el comando ***stop*** o mediante el comando ***kill*** de Docker. El primero mata directamente el contenedor, de manera análoga al kill de linux, a diferencia del otro, que detiene la ejecución de una manera más segura.

```
1 $ docker ps
2 CONTAINER ID          IMAGE          COMMAND          NAMES          CREATED
3 40f5c913912e         ubuntu        "/bin/sh -c 'while tr" 2 seconds
   ago             Up 2 seconds          adoring_euclid
```

```

4 $ docker stop 40f5c913912e
5 40f5c913912e
6 $ docker ps
7 CONTAINER ID        IMAGE               COMMAND             CREATED
                        STATUS             PORTS              NAMES

```

### 3.2.5. *rm*

Hay que tener en cuenta que ni stop ni kill eliminan el contenedor, sino que detienen su ejecución. El contenedor, junto con toda la información que tiene, sigue almacenado. Si queremos eliminar un contenedor definitivamente lo que debemos hacer es usar el comando *rm* de Docker.

```

1 $ docker ps
2 CONTAINER ID        IMAGE               COMMAND             CREATED
                        STATUS             PORTS              NAMES
3 $ docker run -d ubuntu:14.04 /bin/sh -c "while true; do echo hello world;
4   sleep 1; done"
5 5abaece69cbd445e69cae61cef6de9f42e2561eacb4b8969024c922eec348a5d
6 $ docker ps
7 CONTAINER ID        IMAGE               COMMAND             CREATED
                        STATUS             PORTS              NAMES
8 5abaece69cbd        ubuntu:14.04       "/bin/sh -c 'while tr" 3 seconds
9   ago              Up 2 seconds              mad_ardinglyhelli
10 $ docker stop mad_ardinglyhelli
11 mad_ardinglyhelli
12 $ docker ps
13 CONTAINER ID        IMAGE               COMMAND             CREATED
                        STATUS             PORTS              NAMES
14 5abaece69cbd        ubuntu:14.04       "/bin/sh -c 'while tr" 5 minutes
15   ago              Exited (137) 4 minutes ago
16   mad_ardinglyhelli
17 $ docker rm mad_ardinglyhelli
18 mad_ardinglyhelli
19 $ docker ps -a
20 CONTAINER ID        IMAGE               COMMAND             CREATED
                        STATUS             PORTS              NAMES

```

### 3.2.6. *images* y *rmi*

Otro comando útil a la hora de gestionar Docker es el comando *images*. El comando *images* nos muestra todas las imágenes que tenemos en local. Si queremos

eliminar alguna, usamos el comando *rmi*.

```

1 $ docker images
2 REPOSITORY          TAG                IMAGE ID           CREATED
3 ubuntu              latest            a005e6b7dd01      18 hours ago
4                    188.4 MB
5 ros                 latest            67110eef39cf       7 weeks ago
6                    826.7 MB
7 hello-world         latest            af340544ed62       9 weeks ago
8                    960 B
9 $ docker rmi -f hello-world
10 Untagged: hello-world:latest
11 Deleted: af340544ed62de0680f441c71fa1a80cb084678fed42bae393e543faea3a572c
12 Deleted: 535020c3e8add9d6bb06e5ac15a261e73d9b213d62fb2c14d752b8e189b2b912
13 $ docker images
14 REPOSITORY          TAG                IMAGE ID           CREATED
15 ubuntu              latest            a005e6b7dd01      18 hours ago
16                    188.4 MB
17 ros                 latest            67110eef39cf       7 weeks ago
18                    826.7 MB

```

### 3.3. Creación de Dockerfiles

Hasta ahora hemos visto que podemos crear contenedores Docker de una manera sencilla, pero si queremos hacer algún tipo de cambio en la configuración de estos contenedores debemos hacerlo de manera manual, accediendo a la terminal del contenedor y usando comandos. Docker provee un potentísimo sistema que nos permite automatizar las tareas de configuración de nuestras imágenes de Docker, que es el uso de Dockerfiles. En realidad, cuando nosotros llamamos al comando run de Docker y no tenemos una imagen de Docker, lo que estamos haciendo es llamar a un Dockerfile que se encuentra en el Docker Hub, y mediante él generar la imagen desde la que se creará el contenedor. De esta manera, mediante el uso de imágenes personalizadas, crearemos contenedores personalizados, con programas instalados o diferentes configuraciones realizadas en ellos.

Los Dockerfile tienen una sintaxis especial, que nos permitirán entre otras cosas, ejecutar comandos de linux para configurar aspectos de nuestro contenedor. A continuación se muestra el Dockerfile que se usa para crear una imagen de Ubuntu [Tianon Gravi(10 de Octubre de 2015)].

```

1 FROM scratch
2 ADD ubuntu-trusty-core-cloudimg-amd64-root.tar.gz /
3
4 # a few minor docker-specific tweaks

```

```

5 # see https://github.com/docker/docker/blob/master/contrib/mkimage/
  debootstrap
6 RUN echo '#!/bin/sh' > /usr/sbin/policy-rc.d \
7 && echo 'exit 101' >> /usr/sbin/policy-rc.d \
8 && chmod +x /usr/sbin/policy-rc.d \
9 \
10 && dpkg-divert --local --rename --add /sbin/initctl \
11 && cp -a /usr/sbin/policy-rc.d /sbin/initctl \
12 && sed -i 's/^exit.*/exit 0/' /sbin/initctl \
13 \
14 && echo 'force-unsafe-io' > /etc/dpkg/dpkg.cfg.d/docker-apt-speedup \
15 \
16 && echo 'DPkg::Post-Invoke { "rm -f /var/cache/apt/archives/*.deb /var/
  cache/apt/archives/partial/*.deb /var/cache/apt/*.bin || true"; };' > /
  etc/apt/apt.conf.d/docker-clean \
17 && echo 'APT::Update::Post-Invoke { "rm -f /var/cache/apt/archives/*.deb /
  var/cache/apt/archives/partial/*.deb /var/cache/apt/*.bin || true"; };'
  >> /etc/apt/apt.conf.d/docker-clean \
18 && echo 'Dir::Cache::pkgcache ""; Dir::Cache::srcpkgcache "";' >> /etc/apt
  /apt.conf.d/docker-clean \
19 \
20 && echo 'Acquire::Languages "none";' > /etc/apt/apt.conf.d/docker-no-
  languages \
21 \
22 && echo 'Acquire::GzipIndexes "true"; Acquire::CompressionTypes::Order:: "
  gz";' > /etc/apt/apt.conf.d/docker-gzip-indexes
23
24 # enable the universe
25 RUN sed -i 's/^#\s*(deb.*universe)\$/\1/g' /etc/apt/sources.list
26
27 # overwrite this with 'CMD []' in a dependent Dockerfile
28 CMD ["/bin/bash"]

```

Se puede observar que hay diferentes comandos en mayúscula que llaman la atención. Estos comandos son los que reconoce Docker. Con el comando **FROM** se le indica a Docker otro Dockerfile sobre el que empezar, en caso de que queramos partir de una imagen ya existente. En este caso al usar el termino *scratch*, se le indica que parta desde cero. Es *obligatorio* empezar siempre un Dockerfile con este comando.

Con el comando **ADD**, se añade un archivo, indicándole dónde queremos añadirlo. En este caso añade un *tarball* en el que se encuentra Ubuntu. Con el comando **RUN**, se ejecuta un comando de linux. El Dockerfile de Ubuntu utiliza una serie de comandos para realizar diversas tareas, como definir repositorios. Con el comando **CMD**, se define un comportamiento por defecto a la hora de lanzar la imagen, en el caso de Ubuntu se lanza una terminal en bash.

Existen más comandos que soportan los Dockerfiles. La lista de todos los comandos que permite Un Dockerfile es la siguiente [Docker Inc.(10 de Octubre de 2015b)]:

- **FROM**: Indica que Dockerfile tomar como base (*scratch* para no usar ninguno)

- **MAINTAINER**: Indica quien es el encargado de mantener el Dockerfile. Normalmente se usa un nombre o una dirección de correo electrónico
- **RUN**: Sirve para ejecutar comandos
- **CMD**: Sirve para establecer la acción por defecto al lanzar un contenedor. Solo se puede usar una vez en un Dockerfile
- **LABEL**: Sirve para añadir metadatos a una imagen
- **EXPOSE**: Sirve para indicar al contenedor que puertos tiene que estar escuchando
- **ENV**: Sirve para crear variables de entorno
- **ADD**: Sirve para copiar archivos al contenedor. Permite usar URLs externas y descomprime archivos automáticamente
- **COPY**: Permite copiar archivos en local al contenedor.
- **ENTRYPOINT**: Permite configurar un contenedor para ejecutarlo como un ejecutable
- **VOLUME**: Sirve para crear puntos de montaje dentro de un contenedor
- **USER**: Sirve para configurar el nombre de usuario o UID que se va a usar para ejecutar las instrucciones que le suceden en el Dockerfile
- **WORKDIR**: Sirve para configurar el directorio con respecto al que se van a ejecutar las instrucciones que le suceden en el Dockerfile
- **ONBUILD**: Sirve para definir instrucciones que se van a ejecutar en caso de usarse el Dockerfile como base para otro Dockerfile

### 3.4. Profundizar en Docker

Aunque hemos explicado lo básico sobre docker, no es el objetivo de este documento explicar el funcionamiento al detalle de Docker ni ser una guía de referencia a la hora de empezar a usarlo. En caso de que se quiera conocer el funcionamiento de todos los comandos de docker, la gestión de las imágenes de Docker, o se quiera obtener más información del Docker Hub, en la documentación oficial de Docker [Docker Inc.(13 de Octubre de 2015a)] se puede encontrar todo lo necesario para comprender al detalle el funcionamiento de Docker.

Tras haber explicado el funcionamiento básico de Docker y algunos puntos para poder iniciarnos con él, a continuación profundizaremos en el tema de las redes en Docker, un tema esencial para poder lanzarnos a construir nuestro sistema.

Con Docker podemos crear una gran cantidad de contenedores diferentes que se ejecuten de manera simultánea. Es lógico que a la hora de crear un sistema nos interese comunicar los contenedores entre ellos para que puedan transmitirse información. Como vamos a construir un sistema de paso mensajes entre contenedores con ROS (cómo usaremos ROS lo veremos en el siguiente capítulo) necesitamos crear una red entre esos contenedores. Para ello, en este capítulo se explicaran diferentes conceptos sobre configuración de redes en Docker. Por una parte se explican cosas básicas sobre redes con Docker y por otra parte herramientas más complejas que provee Docker (como *network* o *swarm*) para crear redes con topologías más complejas o redes multihost.

## 4.1. Redes básicas en Docker

### 4.1.1. *docker0*

Lo primero que hay que saber es que al iniciarse Docker, por defecto, se crea en el anfitrión (host) una interfaz virtual que tiene como nombre ***docker0*** [Docker Inc.(23 de Octubre de 2015)]. Docker coge de manera aleatoria una dirección IP y una subred de rango privado y se la asigna a *docker0*. Las direcciones MAC de los contenedores se asignan usando la dirección IP de cada contenedor, para evitar de esta manera colisiones ARP.

Lo que hace especial a *docker0*, es que no solo es una interfaz, sino que es un puente Ethernet virtual que redirige automáticamente los paquetes entre cualquier otra interfaz que esté conectada a él. De esta manera se pueden comunicar tanto los contenedores entre ellos como con el host.

Además desde un contenedor también se puede acceder a internet. En el capítulo anterior lanzamos contenedores que se creaban mediante los Dockerfiles que se

obtenían del Docker Hub, que es un servidor web que se encuentra en internet.

Sin embargo, **no** podemos acceder a los contenedores desde fuera, desde internet. Por defecto está establecido así, principalmente por temas de seguridad, aunque obviamente se puede cambiar.

## 4.1.2. Prueba de conexión entre contenedores

Si todos los contenedores que creamos se encuentran en una misma subred, podemos comunicarnos entre ellos simplemente con sus direcciones IP privadas o sus nombres de red. Vamos a hacer varias pruebas para comprobar que los contenedores se comunican bien entre ellos.

### 4.1.2.1. *ping*

La forma más sencilla para probar la comunicación entre dos sistemas es el uso de la herramienta *ping* de linux.

Vamos a lanzar por una parte dos contenedores Docker en dos terminales separadas. Para esta prueba usaremos la misma imagen que vamos a usar para crear nuestro sistema, que es la imagen *osrf/ros:indigo-desktop*, a la que previamente hemos hecho un *pull* para tenerla generada, ya que ocupa alrededor de 1,6 GB. Creamos los contenedores de la siguiente manera.

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

Desde fuera comprobamos que tenemos los contenedores en ejecución.

```
1 $ docker ps
2
```

CONTAINER ID	IMAGE	COMMAND	PORTS	NAMES
829a49bb2cfa	osrf/ros:indigo-desktop	"/ros_entrypoint.sh /"	6	compassionate_mccarthy
2f3c19da0cb8	osrf/ros:indigo-desktop	"/ros_entrypoint.sh /"	16	grave_mahavira

```
3
```

Podemos obtener la dirección IP de un contenedor tanto desde fuera como desde dentro de Docker. En este caso lo haremos desde fuera mediante el *inspect* de Docker.

```
1 $ docker inspect --format='{{.NetworkSettings.IPAddress}}'
   compassionate_mccarthy
2 172.17.0.5
3 $ docker inspect --format='{{.NetworkSettings.IPAddress}}' grave_mahavira
```

```
4 172.17.0.4
```

Ya tenemos las direcciones IP privadas que genera *docker0* para los dos contenedores. Ahora probamos a hacer un ping desde un contenedor a otro. Desde el contenedor *grave\_mahavira* con IP 172.17.0.4 al contenedor *compassionate\_mccarthy* con IP 172.17.0.5 se haría así.

```
1 root@2f3c19da0cb8:/# ping 172.17.0.5
2 PING 172.17.0.5 (172.17.0.5) 56(84) bytes of data.
3 64 bytes from 172.17.0.5: icmp_seq=1 ttl=64 time=0.085 ms
4 64 bytes from 172.17.0.5: icmp_seq=2 ttl=64 time=0.058 ms
5 64 bytes from 172.17.0.5: icmp_seq=3 ttl=64 time=0.061 ms
6 64 bytes from 172.17.0.5: icmp_seq=4 ttl=64 time=0.060 ms
7 64 bytes from 172.17.0.5: icmp_seq=5 ttl=64 time=0.106 ms
8 64 bytes from 172.17.0.5: icmp_seq=6 ttl=64 time=0.135 ms
9 ^C
10 --- 172.17.0.5 ping statistics ---
11 6 packets transmitted, 6 received, 0% packet loss, time 4997ms
12 rtt min/avg/max/mdev = 0.058/0.084/0.135/0.028 ms
```

Se puede hacer exactamente lo mismo con los nombres de los contenedores *docker* ya que esto son los nombres que se le dan en la red *docker0* a la que están conectados. En este caso haremos un ping desde *compassionate\_mccarthy* a *grave\_mahavira* usando para ello el nombre del contenedor.

```
1 root@829a49bb2cfa:/# ping grave_mahavira
2 PING grave_mahavira (172.17.0.4) 56(84) bytes of data.
3 64 bytes from grave_mahavira.bridge (172.17.0.4): icmp_seq=1 ttl=64 time
  =0.087 ms
4 64 bytes from grave_mahavira.bridge (172.17.0.4): icmp_seq=2 ttl=64 time
  =0.066 ms
5 64 bytes from grave_mahavira.bridge (172.17.0.4): icmp_seq=3 ttl=64 time
  =0.066 ms
6 64 bytes from grave_mahavira.bridge (172.17.0.4): icmp_seq=4 ttl=64 time
  =0.067 ms
7 64 bytes from grave_mahavira.bridge (172.17.0.4): icmp_seq=5 ttl=64 time
  =0.066 ms
8 64 bytes from grave_mahavira.bridge (172.17.0.4): icmp_seq=6 ttl=64 time
  =0.064 ms
9 ^C
10 --- grave_mahavira ping statistics ---
11 6 packets transmitted, 6 received, 0% packet loss, time 5001ms
12 rtt min/avg/max/mdev = 0.064/0.069/0.087/0.010 ms
```

Debido a esto, los nombres que se usan en los contendores deben ser **únicos**. Debemos tenerlo en cuenta a la hora de renombrar los contenedores. Tampoco podemos cambiar el nombre de un contenedor durante su ejecución, solo podremos nombrarlo al lanzarlo.



#### 4.1.2.2. *netcat*

Otra forma de probar conexiones algo más versátil es el uso de *netcat*. Netcat permite probar conexiones con cualquier tipo de puerto. Para probar que podemos usar cualquier puerto de los que no están predefinidos, vamos a usar la herramienta usando un puerto cualquiera de los que tenemos disponibles. En este caso lo haremos usando el puerto 1234. Para probarlo haremos lo siguiente.

1. Ejecutaremos en uno de los contenedores (da igual cual, la comunicación que se establecerá será bidireccional) netcat en modo escucha. En este caso lo haremos en *grave\_mahavira*.

```
root@2f3c19da0cb8:/# netcat -l 1234
```

2. Desde el otro contenedor (*compassionate\_mccarthy*) nos intentaremos conectar al primero.

```
root@829a49bb2cfa:/# netcat grave_mahavira 1234
```

3. Si todo ha salido bien, podremos escribir desde cualquiera de los dos terminales y aparecerá lo introducido en el otro.

a) Escribimos en *grave\_mahavira*.

```
root@2f3c19da0cb8:/# netcat -l 1234
Hola!
```

```
root@829a49bb2cfa:/# netcat grave_mahavira 1234
Hola!
```

b) Y ahora en *compassionate\_mccarthy*

```
root@2f3c19da0cb8:/# netcat -l 1234
Hola!
Hola de nuevo!
```

```
root@829a49bb2cfa:/# netcat grave_mahavira 1234
Hola!
Hola de nuevo!
```

### 4.1.3. Links entre contenedores

Como hemos visto, a causa de tener los contenedores dentro de una red privada, comunicarlos entre ellos es algo trivial. El problema de transmitir información de esta manera es que el interfaz *docker0* se usa para **todos** los contenedores que estén en ejecución en ese host. Si queremos realizar una comunicación privada entre dos contenedores, que sea invisible para el resto de contenedores, debemos usar el mecanismo que provee Docker, el linkado de contenedores [Docker Inc.(24 de Octubre de 2015)].

Docker provee también un sistema para mapear puertos entre dos contenedores, aunque el mejor sistema que podemos usar para conectar contenedores es el linkado, ya que abstrae todo el sistema de puertos, y crea un puente virtual que permite una comunicación segura entre los contenedores.

Para usar el sistema de links de Docker, debemos usar el flag **-link** a la hora de lanzar el contenedor. Primero vamos a crear un contenedor al que llamaremos *ros1*.

```
$ docker run -it --name ros1 osrf/ros:indigo-desktop /bin/bash
```

A continuación vamos a crear otro contenedor, al que llamaremos *ros2*, que este linkado a *ros1*.

```
$ docker run -it --name ros2 --link ros1 osrf/ros:indigo-desktop /bin/bash
```

Ahora desde fuera de los contenedores, miramos los links que tiene *ros2* mediante *inspect*.

```
1 $ docker inspect -f "{{ .HostConfig.Links }}" ros2
2 [/ros1:/ros2/ros1]
```

Ahora desde *ros2* podemos acceder a la información de *ros1*.

Para lograr este enlace Docker usa dos sistemas diferentes:

- Variables de entorno
- Actualizar el fichero */etc/hosts*

Todo esto lo realiza de manera automática a la hora de enlazar dos contenedores.

## 4.2. Redes avanzadas con Docker

### 4.2.1. *network* de Docker

En la versión 1.9 de Docker (la que se usa en este documento) se implementó una nueva funcionalidad que llevaba gestándose desde que salió la versión 1.7. Esta nueva funcionalidad permite crear redes con diferentes topologías de una manera sencilla, abstrayendo de configuraciones, al igual que los links de Docker. A diferencia de los links de Docker, que está más orientados a conectar directamente contenedores, esta nueva funcionalidad permite crear redes virtuales enteras.

La forma de trabajar con esta funcionalidad es mediante el uso del comando `network`.

```
$ docker network --help

Usage:  docker network [OPTIONS] COMMAND [OPTIONS]

Commands:
disconnect      Disconnect container from a network
inspect         Display detailed network information
ls              List all networks
rm              Remove a network
create          Create a network
connect         Connect container to a network

Run 'docker network COMMAND --help' for more information on a command.

--help=false    Print usage
```

#### 4.2.1.1. Prueba de *network*

Vamos a crear una red a la que posteriormente vamos a unir dos contenedores cualquiera (Ubuntu mismo) y haremos pruebas como las anteriores para comprobar que funciona la red que hemos creado.

1. Creamos la red.

```
$ docker network create red-prueba
6a9b1bceb5e8033744d4220da1c2e144aea9543bdea0804af0261d973b7bd57e
```

2. Creamos dos contenedores Docker.

```
$ docker run -it --name ubuntu1 ubuntu /bin/bash
root@f5225f49f6a9:/#
```

```
$ docker run -it --name ubuntu2 ubuntu /bin/bash
root@969d7f0c6a5b:/#
```

3. Añadimos desde el host los dos contenedores Docker a la red. Esto se puede hacer de dos maneras. Se puede indicar la red a la que conectarse en el momento en el que creamos el contenedor o se puede hacer después mediante el subcomando *connect* que tiene *network*. Lo haremos de la segunda manera ya que nos permite usar contenedores ya creados.

```
$ docker network connect red-prueba ubuntu1
$ docker network connect red-prueba ubuntu2
```

4. Probamos la conexión desde los contenedores

- a) Hacemos *ping* entre los contenedores.

```
1 root@f5225f49f6a9:/# ping ubuntu2
2 PING ubuntu2 (172.19.0.3) 56(84) bytes of data.
3 64 bytes from ubuntu2 (172.19.0.3): icmp_seq=1 ttl=64 time=0.144
  ms
4 64 bytes from ubuntu2 (172.19.0.3): icmp_seq=2 ttl=64 time=0.066
  ms
5 64 bytes from ubuntu2 (172.19.0.3): icmp_seq=3 ttl=64 time=0.066
  ms
6 ^C
7 --- ubuntu2 ping statistics ---
8 3 packets transmitted, 3 received, 0% packet loss, time 1998ms
9 rtt min/avg/max/mdev = 0.066/0.092/0.144/0.036 ms
```

```
1 root@969d7f0c6a5b:/# ping ubuntu1
2 PING ubuntu1 (172.19.0.2) 56(84) bytes of data.
3 64 bytes from ubuntu1 (172.19.0.2): icmp_seq=1 ttl=64 time=0.062
  ms
4 64 bytes from ubuntu1 (172.19.0.2): icmp_seq=2 ttl=64 time=0.077
  ms
5 64 bytes from ubuntu1 (172.19.0.2): icmp_seq=3 ttl=64 time=0.060
  ms
6 64 bytes from ubuntu1 (172.19.0.2): icmp_seq=4 ttl=64 time=0.067
  ms
7 ^C
8 --- ubuntu1 ping statistics ---
9 4 packets transmitted, 4 received, 0% packet loss, time 2999ms
10 rtt min/avg/max/mdev = 0.060/0.066/0.077/0.010 ms
```

- b) Usamos *netcat* con un puerto mayor que el 1024.

```
root@f5225f49f6a9:/# netcat -l 1234
```

```
Hola  
que tal?
```

```
root@969d7f0c6a5b:/# netcat ubuntu1 1234  
Hola  
que tal?
```

### 4.2.2. Redes Multi-Host

Para crear redes Multi-Host con Docker lo mejor es usar Docker Swarm. Docker Swarm es una herramienta que provee Docker que sirve para crear clusters de contenedores Docker. Para crear redes Multi-Host con esta herramienta necesitamos instalar en el host lo siguiente:

1. **VirtualBox.**
2. **Docker Machine.** Docker Machine es un software proporcionado por los creadores de Docker que nos permite crear diferentes hosts en un mismo sistema. [Docker Inc.(29 de Noviembre de 2015a)] Permite lanzar demonios del proceso *docker* independientes entre sí.

En caso de usar Docker desde OS X o desde Windows deberíamos tener ya VirtualBox instalado.

#### 4.2.2.1. Instalación de Swarm

Para instalar las herramientas necesarias (en un sistema GNU-Linux) haremos lo siguiente:

- Instalamos VirtualBox desde el gestor de paquetes. Para Debian y derivados usamos apt-get.

```
$ sudo apt-get install virtualbox
```

- Para instalar Docker Machine hay que seguir los siguientes pasos: [Docker Inc.(29 de Noviembre de 2015b)]

1. Instalamos Docker Machine con el siguiente comando (instala la versión v0.5.0, que al escribir este documento es la última versión).

```
1 $ curl -L https://github.com/docker/machine/releases/download/v0  
   .5.0/docker-machine_linux-amd64.zip >machine.zip && \  
2 unzip machine.zip && \  
3 rm machine.zip && \  
4
```

```
4 sudo mv docker-machine* /usr/local/bin
```

2. Comprobamos que se ha instalado correctamente mediante el siguiente comando.

```
1 $ docker-machine -v
2 docker-machine version 0.5.0 (04cfa58)
```

- Para instalar Swarm simplemente hacemos un pull de la imagen de Swarm mediante Docker, la cual posteriormente instanciaremos y le añadiremos nodos con sus respectivos contenedores dentro de ellos.

```
$ docker pull swarm
```

#### 4.2.2.2. Creación de un Swarm

Hay que distinguir tres conceptos para comprender el funcionamiento de Docker Swarm. Son los siguientes:

- **Docker Node (Nodo Docker):** una máquina ejecutando el demonio de Docker (la cual puede tener diferentes contenedores)
- **Swarm Host (Anfitrión Swarm):** Una máquina ejecutando el demonio de Swarm
- **Swarm (Enjambre):** Una serie de Nodos Docker

La creación del Swarm se llevara a cabo desde el host Windows y no desde la VM con GNU/Linux. Esto es debido a que es necesario que el SO soporte tecnología de virtualización, y hacer que una VM soporte virtualización para crear más VMs dentro de ella es un proceso redundante e innecesario.

De todas maneras el proceso de creación del Swarm es análogo, y en el caso de Windows, Docker Machine ya viene instalado como dependencia del Docker Toolbox, que es la herramienta mediante la que se instala todo el entorno de Docker en sistemas operativos Windows.

Docker Toolbox se puede obtener para plataformas Mac y Windows desde el siguiente enlace: <https://www.docker.com/docker-toolbox>.

Los pasos para crear un Swarm son los siguientes: [Docker Inc.(17 de Enero de 2016b)]

1. Vemos los host Docker de nuestro sistema.

```
1 $ docker-machine ls
2 NAME          ACTIVE  URL                               STATE  URL
                                SWARM  DOCKER  ERRORS
```

```

3 default * virtualbox Running tcp://192.168.99.100:2376
  v1.9.1

```

En este caso vemos uno llamado *default*. Ese es el que usa Docker para ejecutar Docker en Windows. En caso de usar Docker desde GNU/Linux ese host no se mostraría.

## 2. Creamos una máquina virtual en VirtualBox llamada *local*.

```

1 $ docker-machine create -d virtualbox local
2 Running pre-create checks...
3 Creating machine...
4 (local) Copying C:\Users\ander\.docker\machine\cache\boot2docker.iso
   to C:\Users\ander\.docker\machine\machines\local\boot2docker.iso
   ...
5 (local) Creating VirtualBox VM...
6 (local) Creating SSH key...
7 (local) Starting the VM...
8 (local) Waiting for an IP...
9 Waiting for machine to be running, this may take a few minutes...
10 Machine is running, waiting for SSH to be available...
11 Detecting operating system of created instance...
12 Detecting the provisioner...
13 Provisioning with boot2docker...
14 Copying certs to the local machine directory...
15 Copying certs to the remote machine...
16 Setting Docker configuration on the remote daemon...
17 Checking connection to Docker...
18 Docker is up and running!
19 To see how to connect Docker to this machine, run: C:\Program Files\
   Docker Toolbox\docker-machine.exe env local

```

## 3. Cargamos la configuración de la maquina en la shell.

```

1 $ eval "$(docker-machine env local)"

```

## 4. Generamos un *discovery token* mediante la imagen de Swarm de Docker. Este token nos servirá para gestionar el Swarm más adelante.

El comando de abajo ejecuta el comando *create* de Swarm en un contenedor. Descargará la imagen automáticamente en caso de no tenerla, como de costumbre.

```

1 $ docker run swarm create
2 Unable to find image 'swarm:latest' locally
3 latest: Pulling from swarm
4 d681c900c6e3: Pull complete
5 188de6f24f3f: Pull complete
6 90b2ffb8d338: Pull complete
7 237af4efea94: Pull complete

```

```

8 3b3fc6f62107: Pull complete
9 7e6c9135b308: Pull complete
10 986340ab62f0: Pull complete
11 a9975e2cc0a3: Pull complete
12 Digest: sha256:
    c21fd414b0488637b1f05f13a59b032a3f9da5d818d31da1a4ca98a84c0c781b
13 Status: Downloaded newer image for swarm:latest
14 dc60acd12fc3a6b14754abef91501be2

```

5. Guardamos el token que nos devuelve ya que lo necesitaremos para más adelante. En nuestro caso es dc60acd12fc3a6b14754abef91501be2.

Si abrimos VirtualBox podremos observar que tenemos dos máquinas de tipo GNU/Linux, la que acabamos de crear y la que usa Docker para ejecutarse.

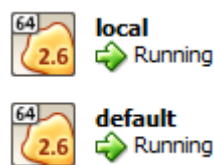


Figura 4.1.: Imágenes GNU/Linux creadas

#### 4.2.2.3. Lanzar el *Swarm manager*

Para crear un Swarm, primero es necesario crear una máquina llamada *Swarm manager*. El *Swarm manager* dirige y maneja todos los contenedores del clúster, es decir, todos los Nodos Docker.

Después de crear el *Swarm manager*, hay que crear una serie de Nodos Docker, que son los que se encargaran de ejecutar instancias de Docker y de manejar contenedores.

A continuación vamos a crear un *Swarm manager* y dos Nodos Docker. Para ello usamos el comando *create* de Docker Machine, usando el token anotado anteriormente y diferentes flags que veremos a continuación.

1. Creamos un Swarm manager bajo VirtualBox.

```

1 $ docker-machine create -d virtualbox --swarm --swarm-master --swarm-
    discovery token://dc60acd12fc3a6b
2 14754abef91501be2 swarm-master
3 Running pre-create checks...
4 Creating machine...
5 (swarm-master) Copying C:\Users\ander\.docker\machine\cache\
    boot2docker.iso to C:\Users\ander\.docker\machine\machines\swarm-
    master\boot2docker.iso...
6 (swarm-master) Creating VirtualBox VM...

```



```

7 (swarm-master) Creating SSH key...
8 (swarm-master) Starting the VM...
9 (swarm-master) Waiting for an IP...
10 Waiting for machine to be running, this may take a few minutes...
11 Machine is running, waiting for SSH to be available...
12 Detecting operating system of created instance...
13 Detecting the provisioner...
14 Provisioning with boot2docker...
15 Copying certs to the local machine directory...
16 Copying certs to the remote machine...
17 Setting Docker configuration on the remote daemon...
18 Configuring swarm...
19 Checking connection to Docker...
20 Docker is up and running!
21 To see how to connect Docker to this machine, run: C:\Program Files\
    Docker Toolbox\docker-machine.exe env swarm-master

```

Para indicar que es el Swarm manager usamos el flag `--swarm-master`.

2. Creamos un nodo Swarm. Lo llamaremos *swarm-agent-00*.

```

1 $ docker-machine create -d virtualbox --swarm --swarm-discovery token
  ://dc60acd12fc3a6b14754abef91501b
2 e2 swarm-agent-00
3 Running pre-create checks...
4 Creating machine...
5 (swarm-agent-00) Copying C:\Users\ander\.docker\machine\cache\
  boot2docker.iso to C:\Users\ander\.docker\machine\machines\swarm-
  agent-00\boot2docker.iso...
6 (swarm-agent-00) Creating VirtualBox VM...
7 (swarm-agent-00) Creating SSH key...
8 (swarm-agent-00) Starting the VM...
9 (swarm-agent-00) Waiting for an IP...
10 Waiting for machine to be running, this may take a few minutes...
11 Machine is running, waiting for SSH to be available...
12 Detecting operating system of created instance...
13 Detecting the provisioner...
14 Provisioning with boot2docker...
15 Copying certs to the local machine directory...
16 Copying certs to the remote machine...
17 Setting Docker configuration on the remote daemon...
18 Configuring swarm...
19 Checking connection to Docker...
20 Docker is up and running!
21 To see how to connect Docker to this machine, run: C:\Program Files\
    Docker Toolbox\docker-machine.exe env swarm-agent-00

```

3. Creamos otro nodo Swarm. Lo llamaremos *swarm-agent-01*.

```

1 $ docker-machine create -d virtualbox --swarm --swarm-discovery token
  ://dc60acd12fc3a6b14754abef91501b

```

```

2 e2 swarm-agent-01
3 Running pre-create checks...
4 Creating machine...
5 (swarm-agent-01) Copying C:\Users\ander\.docker\machine\cache\
   boot2docker.iso to C:\Users\ander\.docker\machine\machines\swarm-
   agent-01\boot2docker.iso...
6 (swarm-agent-01) Creating VirtualBox VM...
7 (swarm-agent-01) Creating SSH key...
8 (swarm-agent-01) Starting the VM...
9 (swarm-agent-01) Waiting for an IP...
10 Waiting for machine to be running, this may take a few minutes...
11 Machine is running, waiting for SSH to be available...
12 Detecting operating system of created instance...
13 Detecting the provisioner...
14 Provisioning with boot2docker...
15 Copying certs to the local machine directory...
16 Copying certs to the remote machine...
17 Setting Docker configuration on the remote daemon...
18 Configuring swarm...
19 Checking connection to Docker...
20 Docker is up and running!
21 To see how to connect Docker to this machine, run: C:\Program Files\
   Docker Toolbox\docker-machine.exe env swarm-agent-01

```

4. Si miramos de nuevo en VirtualBox, veremos que tenemos las 3 nuevas máquinas que acabamos de crear junto a las dos que teníamos anteriormente.



Figura 4.2.: Imágenes GNU/Linux creadas para el Swarm

#### 4.2.2.4. Configurar el Swarm y lanzar contenedores

1. Apuntamos nuestro entorno Docker a la máquina que está ejecutando el Swarm master.

```
1 $ eval $(docker-machine env --swarm swarm-master)
```

2. Obtenemos información sobre el Swarm creado mediante el comando *info* de Docker.

```
1 $ docker info
2 Containers: 4
3 Images: 3
4 Role: primary
5 Strategy: spread
6 Filters: health, port, dependency, affinity, constraint
7 Nodes: 3
8   swarm-agent-00: 192.168.99.103:2376
9     Status: Healthy
10    Containers: 1
11    Reserved CPUs: 0 / 1
12    Reserved Memory: 0 B / 1.021 GiB
13    Labels: executiondriver=native-0.2, kernelversion=4.1.13-
           boot2docker, operatingsystem=Boot2Docker 1.9.1 (TCL 6.4.1);
           master : cef800b - Fri Nov 20 19:33:59 UTC 2015, provider=
           virtualbox, storagedriver=aufs
14   swarm-agent-01: 192.168.99.104:2376
15     Status: Healthy
16    Containers: 1
17    Reserved CPUs: 0 / 1
18    Reserved Memory: 0 B / 1.021 GiB
19    Labels: executiondriver=native-0.2, kernelversion=4.1.13-
           boot2docker, operatingsystem=Boot2Docker 1.9.1 (TCL 6.4.1);
           master : cef800b - Fri Nov 20 19:33:59 UTC 2015, provider=
           virtualbox, storagedriver=aufs
20   swarm-master: 192.168.99.102:2376
21     Status: Healthy
22    Containers: 2
23    Reserved CPUs: 0 / 1
24    Reserved Memory: 0 B / 1.021 GiB
25    Labels: executiondriver=native-0.2, kernelversion=4.1.13-
           boot2docker, operatingsystem=Boot2Docker 1.9.1 (TCL 6.4.1);
           master : cef800b - Fri Nov 20 19:33:59 UTC 2015, provider=
           virtualbox, storagedriver=aufs
26 CPUs: 3
27 Total Memory: 3.064 GiB
28 Name: swarm-master
```

Se puede observar que tanto el manager como los nodos tienen el puerto 2376 expuesto. Cuando se crea un Swarm se puede usar el puerto que se quiera, e incluso usar diferentes puertos para diferentes nodos. Cada nodo del Swarm ejecuta un gestor de contenedores Docker.

En el caso del master, ejecuta tanto el gestor de contenedores como el Swarm manager. Esto no suele ser recomendable en entornos de producción.

### 3. Consultamos las imágenes en ejecución en el Swarm.

```

1 $ docker ps -a
2 CONTAINER ID          IMAGE                COMMAND              NAMES
3 3f228154f8ff          swarm:latest        "/swarm join --advert" 20
   minutes ago         Up 20 minutes
   agent-01/swarm-agent
4 e9df36929545          swarm:latest        "/swarm join --advert" 22
   minutes ago         Up 22 minutes
   agent-00/swarm-agent
5 a18b76c0013d          swarm:latest        "/swarm join --advert" 27
   minutes ago         Up 27 minutes
   master/swarm-agent
6 35fcf51df08a          swarm:latest        "/swarm manage --tlsv" 27
   minutes ago         Up 27 minutes
   master/swarm-agent-master

```

### 4. Lanzamos una imagen en el Swarm. Vamos a usar la ya conocida *hello-world*.

```

1 $ docker run hello-world
2
3 Hello from Docker.
4 This message shows that your installation appears to be working
   correctly.
5
6 To generate this message, Docker took the following steps:
7 1. The Docker client contacted the Docker daemon.
8 2. The Docker daemon pulled the "hello-world" image from the Docker
   Hub.
9 3. The Docker daemon created a new container from that image which
   runs the
10 executable that produces the output you are currently reading.
11 4. The Docker daemon streamed that output to the Docker client, which
   sent it
12 to your terminal.
13
14 To try something more ambitious, you can run an Ubuntu container with
   :
15 $ docker run -it ubuntu bash
16
17 Share images, automate workflows, and more with a free Docker Hub
   account:
18 https://hub.docker.com
19
20 For more examples and ideas, visit:
21 https://docs.docker.com/userguide/

```

### 5. Vamos a usar el comando *ps* de Docker para ver en que nodo se ha ejecutado el contenedor.

```

1 $ docker ps -a
2 CONTAINER ID          IMAGE                NAMES
3 91a89b22a8b1         hello-world         swarm-agent-00/sleepy_perlman
4 3f228154f8ff         swarm:latest       swarm-agent-01/swarm-agent
5 e9df36929545         swarm:latest       swarm-agent-00/swarm-agent
6 a18b76c0013d         swarm:latest       swarm-master/swarm-agent
7 35fcf51df08a         swarm:latest       swarm-master/swarm-agent-
   master

```

Como se puede observar se ha ejecutado en el nodo *swarm-agent-00*.

Una vez visto el funcionamiento de Swarm y las redes Multi-Host en Docker, podemos complicar nuestro sistema tanto como queramos. Podemos lanzar diferentes nodos en diferentes sistemas separados dentro de una misma red o entre diferentes redes conectadas entre sí. En la documentación de Swarm [Docker Inc.(17 de Enero de 2016a)] se puede encontrar más información al respecto.

### 4.3. Configuración manual de redes

Aunque en este capítulo se han enseñado varios mecanismos que provee Docker para administrar redes de contenedores, también podemos configurar toda nuestra red de una manera más tradicional, mediante la modificación de archivos como */etc/hosts* o */etc/interfaces* en nuestros contenedores, el uso de *iptables*, configuración de DNS,...

Docker mediante estos mecanismos busca abstraer parte de la configuración para hacerla mas sencilla de cara al desarrollador o al administrador.

Prácticamente cualquier aspecto relacionado con las redes se puede configurar en Docker mediante una serie de flags especiales a la hora de lanzar el servicio de Docker, por lo que no se pueden modificar mientras Docker esté en ejecución (no confundir con que un contenedor esté en ejecución). Algunos de esos comandos con flags especiales solo se pueden ejecutar con el servicio de Docker parado. Varios de los mas importantes son.

```

1 --default-gateway=IP_ADDRESS # Define la IP a la que se conectaran los
   contenedores de Docker al crearse, por defecto se usa la de docker0
2 --icc=true|false # Indica si se permite la comunicacion entre contenedores
   , por defecto true
3 --ipv6=true|false # Define si se usa IPv6, por defecto false
4 --ip-forward=true|false # Indica si esta activada la comunicacion entre
   los contenedores y el exterior, por defecto true
5 --iptables=true|false # Define si se permite el uso de iptables (filtra
   direcciones y puertos, se usa como firewall en sistemas tipo UNIX)

```

---

En la documentación de Networking avanzado de Docker [Docker Inc.(23 de Octubre de 2015)] se puede encontrar mucha más información de como hacer esto.

## **Parte III.**

### **ROS**

## Introducción a ROS

Anteriormente se ha explicado qué es ROS y que características tiene. En este capítulo se pasará a profundizar en su funcionamiento, y mostraremos como se pueden programar los sistemas de paso de mensajes que pasaremos a implementar en un sistema de contenedores Docker.

### 5.1. Entorno de trabajo

Para poder trabajar con ROS lo primero que debemos tener es un entorno con las herramientas de ROS instaladas. En este caso, como vamos a usar contenedores Docker con todo lo necesario en ellos no necesitaremos instalar ningún tipo de paquete adicional. El Dockerfile que vamos a usar para generar la imagen Ubuntu con ROS ya instalado se encuentra disponible en el Docker Hub y aparece con el nombre *osrf/ros:indigo-desktop*. Esta imagen contiene [Open Source Robotics Foundation(27 de Octubre de 2015)]:

- **ros-base**, la base de ROS, que a su vez contiene:
  - **ros-core**, el núcleo de ROS
  - Librerías para construir aplicaciones
  - Librerías para comunicación
- **rqt**, framework basado en Qt para construir aplicaciones con interfaz gráfica de usuario (GUI) con ROS
- **rviz**, herramienta de visualización 3D para ROS
- Librerías genéricas para sistemas robóticos

De momento no vamos a hacer uso de ninguna herramienta gráfica. Para probar el funcionamiento de ROS antes de programar sobre nuestro sistema, vamos a elaborar



una pequeña aplicación distribuida que nos permita enviar unos datos de un nodo a otro.

## 5.2. Uso de ROS

Al igual que hacemos con Docker, manejaremos ROS desde la terminal. Existen una serie de comandos que necesitamos conocer para poder empezar a trabajar con él. Los comandos son los siguientes [Universida d'Alacant(3 de Noviembre de 2015)]:

- **roscd**: cambia a un directorio de paquete
- **roscore**: 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
- **rostopic**: nos proporciona información sobre un nodo. Disponemos de las siguientes opciones:
  - **rostopic info [nodo]**: muestra información sobre el nodo
  - **rostopic kill [nodo]**: mata ese proceso
  - **rostopic list**: muestra los nodos ejecutándose
  - **rostopic machine [maquina]**: muestra los nodos que se están ejecutando en la máquina
  - **rostopic ping [nodo]**: comprueba la conectividad del nodo
- **roslaunch [nombre\_paquete] [nombre\_nodo]**: permite ejecutar cualquier aplicación de un paquete sin necesidad de cambiar a su directorio
- **rostopic**: permite obtener información sobre un topic
  - **rostopic bw [topic]**: muestra el ancho de banda consumido por un tópico
  - **rostopic echo [topic]**: imprime datos del topic por la salida estándar
  - **rostopic find**: encuentra un topic
  - **rostopic info [topic]**: imprime información de un topic
  - **rostopic list**: imprime información sobre los topics activos
  - **rostopic pub [topic]**: publica datos a un topic activo
  - **rostopic type [topic]**: imprime el tipo de información de un topic
- **roswtf**: permite comprobar si algo va mal

## 5.3. Gestionar paquetes ROS

Cualquier aplicación que hagamos con ROS será un paquete que podremos instalar. ROS dispone de varias herramientas mediante las cuales crear y compilar esos paquetes. la más famosa de estas herramientas es **catkin**. *catkin* permite crear y compilar paquetes (que se denominan paquetes *catkin*). que contendrán todo el código que desarrollemos. Se basa en el funcionamiento de herramientas como Make o CMake, habituales para compilar aplicaciones en entornos linux.

Para que un paquete se pueda considerar un paquete *catkin* tiene que cumplir una serie de requisitos [Open Source Robotics Foundation(29 de Octubre de 2015a)]:

1. El paquete debe contener un fichero de llamado *package.xml* con un formato concreto.
  - Este archivo contendrá diferentes metadatos sobre el paquete
2. El paquete debe contener un fichero *CMakeLists.txt* que usará catkin a la hora de compilar el paquete
3. No puede haber más de un paquete por carpeta
  - No puede haber paquetes anidados en otros paquetes ni varios paquetes en una misma carpeta

### 5.3.1. Crear paquetes

Para crear un paquete con catkin primero debemos crear un workspace de catkin, que es quien contendrá los paquetes que creemos. Lo haremos de la siguiente manera [Open Source Robotics Foundation(29 de Octubre de 2015b)].

```
1 $ mkdir -p ~/catkin_ws/src
2 $ cd ~/catkin_ws/src
3 $ catkin_init_workspace
4 # Aunque no haya nada compilamos el workspace
5 $ cd ~/catkin_ws/
6 $ source devel/setup.bash
```

Una vez creado el workspace podemos crear un paquete de la siguiente manera.

```
1 $ cd ~/catkin_ws/src
2 $ catkin_create_pkg paquete std_msgs rospy roscpp
```

Como se puede observar, hemos creado un paquete llamado *paquete*. Los siguientes nombres que le hemos indicado son las dependencia que tiene el paquete, en este caso son dependencias para compilar código fuente de C++ y Python, además de herramientas para manejar mensajes.

### 5.3.2. Compilar paquetes

Para compilar paquetes generados mediante catkin, se hace uso del comando *catkin-make*. Este comando compila todo en función el archivo *CMakeLists.txt* para saber como debe compilar el paquete.

```
1 $ cd ~/catkin_ws/  
2 $ catkin_make
```

## 5.4. Modelo distribuido Publisher-Subscriber

En redes existe un tipo de comunicación conocida como modelo *Publisher-Subscriber*. Esta metodología consiste en dos nodos, uno conocido como *publisher* que va publicando mensajes. Éste nodo no decide a quién se mandan los mensajes, sino que el otro nodo, el nodo *subscriber*, tal y como su nombre indica, se suscribe al nodo que publica los mensajes, y a partir de ese momento recibe los mensajes que va publicando.

En realidad, para separar diferentes tipos de mensajes, se hace uso de *topics* (temas). Esto permite que un publisher pueda tener varios canales de envío de mensajes. Debido a esto, lo que realmente hacen los subscribers es suscribirse a esos topics, y no a los publishers. A su vez, Los publishers deciden en que topic publicar los mensajes.

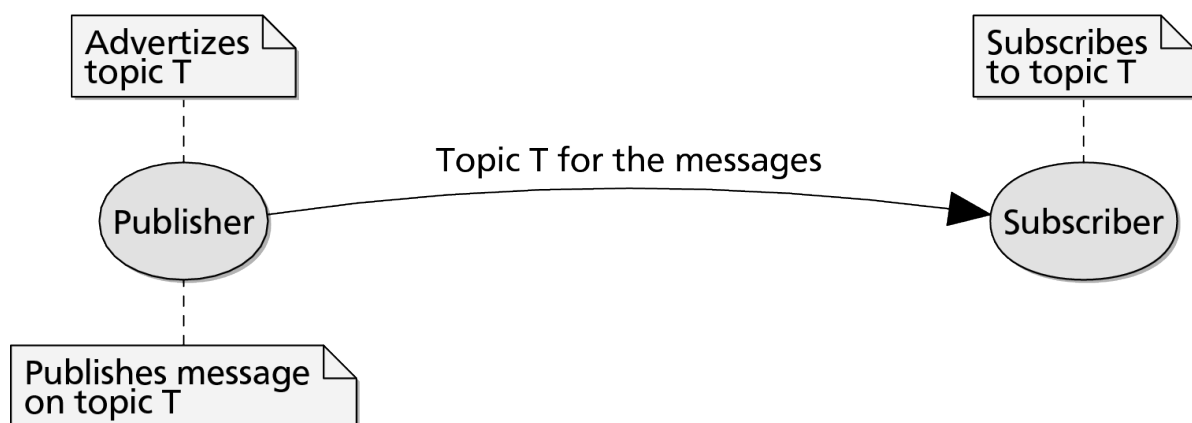


Figura 5.1.: Modelo Publisher-Subscriber

## Prueba con nodos ROS

Vamos a crear un pequeño ejemplo de este tipo para comprender mejor su funcionamiento, pero con todos los nodos en el host, para asimilar el proceso de crear y probar paquetes con ROS. En el ejemplo, El publisher publicará una serie de datos, que será recibidos por el subscriber una vez se haya suscrito al topic en el que se estén publicando. Para hacerlo lo más simple posible, los datos que se pasarán serán cadenas de caracteres.

Nos basaremos en uno de los ejemplos de los que disponemos en la Wiki de ROS [Open Source Robotics Foundation(3 de Noviembre de 2015)], al que haremos unos pequeños cambios para adaptarlo.

### 6.1. Código de la prueba

El código se divide en dos ficheros .cpp en los que tendremos por una parte el publisher y en la otra el subscriber. Tendremos un tercer archivo más, que será el archivo CMakeLists.txt que nos servirá a la hora de construir el paquete.

#### 6.1.1. Publisher

```
1  #include "ros/ros.h"
2  #include "std_msgs/String.h"
3
4  #include <sstream>
5
6  int main(int argc, char **argv)
7  {
8      ros::init(argc, argv, "talker");
```

```
9   ros::NodeHandle n;
10  ros::Publisher chatter_pub = n.advertise<std_msgs::String>("chatter",
↪   1000);
11  ros::Rate loop_rate(10);
12
13  int count = 0;
14  while (ros::ok())
15  {
16      std_msgs::String msg;
17      std::stringstream ss;
18      ss << "Hola (Mensaje numero: " << count << ")";
19      msg.data = ss.str();
20      ROS_INFO("%s", msg.data.c_str());
21      chatter_pub.publish(msg);
22      ros::spinOnce();
23      loop_rate.sleep();
24      ++count;
25  }
26
27  return 0;
28 }
```

### 6.1.2. Subscriber

```
1  #include "ros/ros.h"
2  #include "std_msgs/String.h"
3
4  void chatterCallback(const std_msgs::String::ConstPtr& msg)
5  {
6      ROS_INFO("I heard: [%s]", msg->data.c_str());
7  }
8
9  int main(int argc, char **argv)
10 {
11     ros::init(argc, argv, "listener");
12     ros::NodeHandle n;
13     ros::Subscriber sub = n.subscribe("chatter", 1000, chatterCallback);
14     ros::spin();
15     return 0;
16 }
```

### 6.1.3. CMakeLists

```
1  cmake_minimum_required(VERSION 2.8.3)
2  project(beginner_tutorials)
3
4  find_package(catkin REQUIRED COMPONENTS roscpp rospy std_msgs genmsg)
5
6  add_message_files(FILES Num.msg)
7  add_service_files(FILES AddTwoInts.srv)
8
9  generate_messages(DEPENDENCIES std_msgs)
10
11 catkin_package()
12
13 include_directories(include ${catkin_INCLUDE_DIRS})
14
15 add_executable(talker src/talker.cpp)
16 target_link_libraries(talker ${catkin_LIBRARIES})
17 add_dependencies(talker prueba_generate_messages_cpp)
18
19 add_executable(listener src/listener.cpp)
20 target_link_libraries(listener ${catkin_LIBRARIES})
21 add_dependencies(listener prueba_generate_messages_cpp)
```

## 6.2. Construir el paquete

Una vez programado el sistema, pasamos a compilarlo y construirlo. Basándonos en lo que se comentó en el capítulo anterior, lo haremos de la siguiente manera.

```
1  ~$ mkdir -p ~/catkin_ws/src
2  ~$ cd ~/catkin_ws/src
3  ~/catkin_ws/src$ catkin_init_workspace
4  Creating symlink "/home/ander/catkin_ws/src/CMakeLists.txt" pointing to "/
   opt/ros/indigo/share/catkin/cmake/toplevel.cmake"
5  ~/catkin_ws/src$ cd ..
6  ~/catkin_ws$ source devel/setup.bash
7  bash: devel/setup.bash: No existe el archivo o el directorio
8  ~/catkin_ws$ cd src/
9  ~/catkin_ws/src$ catkin_create_pkg prueba std_msgs rospy roscpp
10 Created file prueba/package.xml
11 Created file prueba/CMakeLists.txt
12 Created folder prueba/include/prueba
13 Created folder prueba/src
```

```

14 Successfully created files in /home/ander/catkin_ws/src/prueba. Please
    adjust the values in package.xml.
15 ~/catkin_ws/src$ ls
16 CMakeLists.txt  prueba
17 ~/catkin_ws/src$ cd ..
18 ~/catkin_ws$ catkin_make
19 Base path: /home/ander/catkin_ws
20 Source space: /home/ander/catkin_ws/src
21 Build space: /home/ander/catkin_ws/build
22 Devel space: /home/ander/catkin_ws/devel
23 Install space: /home/ander/catkin_ws/install
24 ####
25 ##### Running command: "cmake /home/ander/catkin_ws/src -
    DCATKIN_DEVEL_PREFIX=/home/ander/catkin_ws/devel -DCMAKE_INSTALL_PREFIX
    =/home/ander/catkin_ws/install -G Unix Makefiles" in "/home/ander/
    catkin_ws/build"
26 #####
27 -- The C compiler identification is GNU 4.8.4
28 -- The CXX compiler identification is GNU 4.8.4
29 -- Check for working C compiler: /usr/bin/cc
30 -- Check for working C compiler: /usr/bin/cc -- works
31 -- Detecting C compiler ABI info
32 -- Detecting C compiler ABI info - done
33 -- Check for working CXX compiler: /usr/bin/c++
34 -- Check for working CXX compiler: /usr/bin/c++ -- works
35 -- Detecting CXX compiler ABI info
36 -- Detecting CXX compiler ABI info - done
37 -- Using CATKIN_DEVEL_PREFIX: /home/ander/catkin_ws/devel
38 -- Using CMAKE_PREFIX_PATH: /opt/ros/indigo
39 -- This workspace overlays: /opt/ros/indigo
40 -- Found PythonInterp: /usr/bin/python (found version "2.7.6")
41 -- Using PYTHON_EXECUTABLE: /usr/bin/python
42 -- Using Debian Python package layout
43 -- Using empy: /usr/bin/empy
44 -- Using CATKIN_ENABLE_TESTING: ON
45 -- Call enable_testing()
46 -- Using CATKIN_TEST_RESULTS_DIR: /home/ander/catkin_ws/build/test_results
47 -- Looking for include file pthread.h
48 -- Looking for include file pthread.h - found
49 -- Looking for pthread_create
50 -- Looking for pthread_create - not found
51 -- Looking for pthread_create in pthreads
52 -- Looking for pthread_create in pthreads - not found
53 -- Looking for pthread_create in pthread
54 -- Looking for pthread_create in pthread - found
55 -- Found Threads: TRUE
56 -- Found gtest sources under '/usr/src/gtest': gtests will be built
57 -- Using Python nosetests: /usr/bin/nosetests-2.7
58 -- catkin 0.6.14
59 -- BUILD_SHARED_LIBS is on
60 -- ~~~~~
61 -- ~~ traversing 1 packages in topological order:

```

```

62 -- ~~ - prueba
63 -- ~~~~~
64 -- +++ processing catkin package: 'pruebaROS'
65 -- ==> add_subdirectory(pruebaROS)
66 -- Configuring done
67 -- Generating done
68 -- Build files have been written to: /home/ander/catkin_ws/build
69 ####
70 #### Running command: "make -j1 -l1" in "/home/ander/catkin_ws/build"
71 ####
72 ~/catkin_ws$ ls
73 build devel src
74 ~/catkin_ws$ cd src
75 ~/catkin_ws/src$ ls
76 CMakeLists.txt prueba
77 ~/catkin_ws/src$ cd prueba/
78 ~/catkin_ws/src/prueba$ ls
79 CMakeLists.txt include package.xml src
80 ~/catkin_ws/src/prueba$ cd src
81 ~/catkin_ws/src/prueba/src$ ls
82 ~/catkin_ws/src/prueba/src$ touch talker.cpp
83 ~/catkin_ws/src/prueba/src$ touch listener.cpp
84 ~/catkin_ws/src/prueba/src$ ls
85 listener.cpp talker.cpp

```

Ahora ya tenemos todos los archivos creados, debemos llenarlos con el contenido que hemos mostrado antes. En este caso uso *nano* para editar los archivos, pero se puede hacer con *vi* o cualquier otro editor.

```

1 ~/catkin_ws/src/prueba/src$ nano listener.cpp
2 ~/catkin_ws/src/prueba/src$ nano talker.cpp
3 ~/catkin_ws/src/prueba/src$ cd ..
4 ~/catkin_ws/src/prueba$ ls
5 CMakeLists.txt include package.xml src
6 ~/catkin_ws/src/prueba$ echo "" > CMakeLists.txt # Para vaciar el archivo
7 ~/catkin_ws/src/prueba$ nano CMakeLists.txt

```

Una vez llenos podemos proceder a compilarlo.

```

1 ~/catkin_ws/src/prueba$ cd ..
2 ~/catkin_ws/src$ cd ..
3 ~/catkin_ws$ ls
4 build devel src
5 ~/catkin_ws$ catkin_make
6 ander@ubuntu-VirtualBox:~/catkin_ws$ catkin_make
7 Base path: /home/ander/catkin_ws
8 Source space: /home/ander/catkin_ws/src
9 Build space: /home/ander/catkin_ws/build
10 Devel space: /home/ander/catkin_ws/devel
11 Install space: /home/ander/catkin_ws/install

```



```

12 #####
13 ##### Running command: "cmake /home/ander/catkin_ws/src -
    DCATKIN_DEVEL_PREFIX=/home/ander/catkin_ws/devel -DCMAKE_INSTALL_PREFIX
    =/home/ander/catkin_ws/install -G Unix Makefiles" in "/home/ander/
    catkin_ws/build"
14 #####
15 -- Using CATKIN_DEVEL_PREFIX: /home/ander/catkin_ws/devel
16 -- Using CMAKE_PREFIX_PATH: /home/ander/catkin_ws/devel;/opt/ros/indigo
17 -- This workspace overlays: /home/ander/catkin_ws/devel;/opt/ros/indigo
18 -- Using PYTHON_EXECUTABLE: /usr/bin/python
19 -- Using Debian Python package layout
20 -- Using empy: /usr/bin/empy
21 -- Using CATKIN_ENABLE_TESTING: ON
22 -- Call enable_testing()
23 -- Using CATKIN_TEST_RESULTS_DIR: /home/ander/catkin_ws/build/test_results
24 -- Found gtest sources under '/usr/src/gtest': gtests will be built
25 -- Using Python nosetests: /usr/bin/nosetests-2.7
26 -- catkin 0.6.14
27 -- BUILD_SHARED_LIBS is on
28 -- ~~~~~
29 -- ~~ traversing 1 packages in topological order:
30 -- ~~ - prueba
31 -- ~~~~~
32 -- +++ processing catkin package: 'prueba'
33 -- ==> add_subdirectory(prueba)
34 -- Configuring done
35 -- Generating done
36 -- Build files have been written to: /home/ander/catkin_ws/build
37 #####
38 ##### Running command: "make -j1 -l1" in "/home/ander/catkin_ws/build"
39 #####
40 Scanning dependencies of target listener
41 [ 50%] Building CXX object prueba/CMakeFiles/listener.dir/src/listener.cpp
    .o
42 Linking CXX executable /home/ander/catkin_ws/devel/lib/prueba/listener
43 [ 50%] Built target listener
44 Scanning dependencies of target talker
45 [100%] Building CXX object prueba/CMakeFiles/talker.dir/src/talker.cpp.o
46 Linking CXX executable /home/ander/catkin_ws/devel/lib/prueba/talker
47 [100%] Built target talker

```

Esto nos generará dos ejecutables, *talker* y *listener*, que por defecto irán al directorio `devel/lib/[nombre del paquete]` de nuestro workspace.

### 6.3. Ejecución

Probaremos el sistema para comprobar que funciona correctamente. Para ello haremos lo siguiente.

1. Por una parte lanzamos *roscore*.

```
1 $ roscore
```

2. Compilamos de nuevo el workspace que tenemos creado.

```
1 $ cd ~/catkin_ws
2 $ . devel/setup.bash
3 $ catkin_make
```

3. Lanzamos por otra parte el *listener* de la siguiente manera.

```
1 $ rosrun prueba listener
```

4. Mandamos un mensaje mediante el *talker*.

```
1 $ rostopic pub -1 /chatter std_msgs/String PruebaMensaje
2 publishing and latching message for 3.0 seconds
```

```
1 $ rosrun prueba listener
2 [ INFO] [1447683677.299227505]: I heard: [PruebaMensaje]
```

Se puede apreciar que el listener recibe el mensaje enviado por el talker.

## **Parte IV.**

# **Implementación del sistema**

## Creación del sistema

El sistema que vamos a crear constará de las siguientes partes. Primero sobre las máquinas con SO Windows se instalará una máquina virtual Ubuntu sobre Virtualbox para trabajar en un entorno linux. A día de hoy existen formas de trabajar directamente con Docker en sistemas Windows tanto por CLI (Command Line Interface) como a través de una interfaz gráfica, aunque éstas se basan en emular el kernel de linux. Para este caso se ha optado por trabajar en un entorno linux conocido para hacer más simple el despliegue del sistema.

Dentro de dicha máquina Ubuntu se instalará el propio Docker. Mediante Docker crearemos diferentes contenedores. Cada uno de esos contenedores se crearán a partir de una imagen de Ubuntu que vendrá con ROS instalado. Esa imagen sera la que aparece en el Docker Hub como *osrf/ros:indigo-desktop*. Estas máquinas se comunicarán entre ellas mediante una red creada con la herramienta *network* de Docker.

El esquema del sistema vendría a ser el que se muestra en la Figura 7.1.

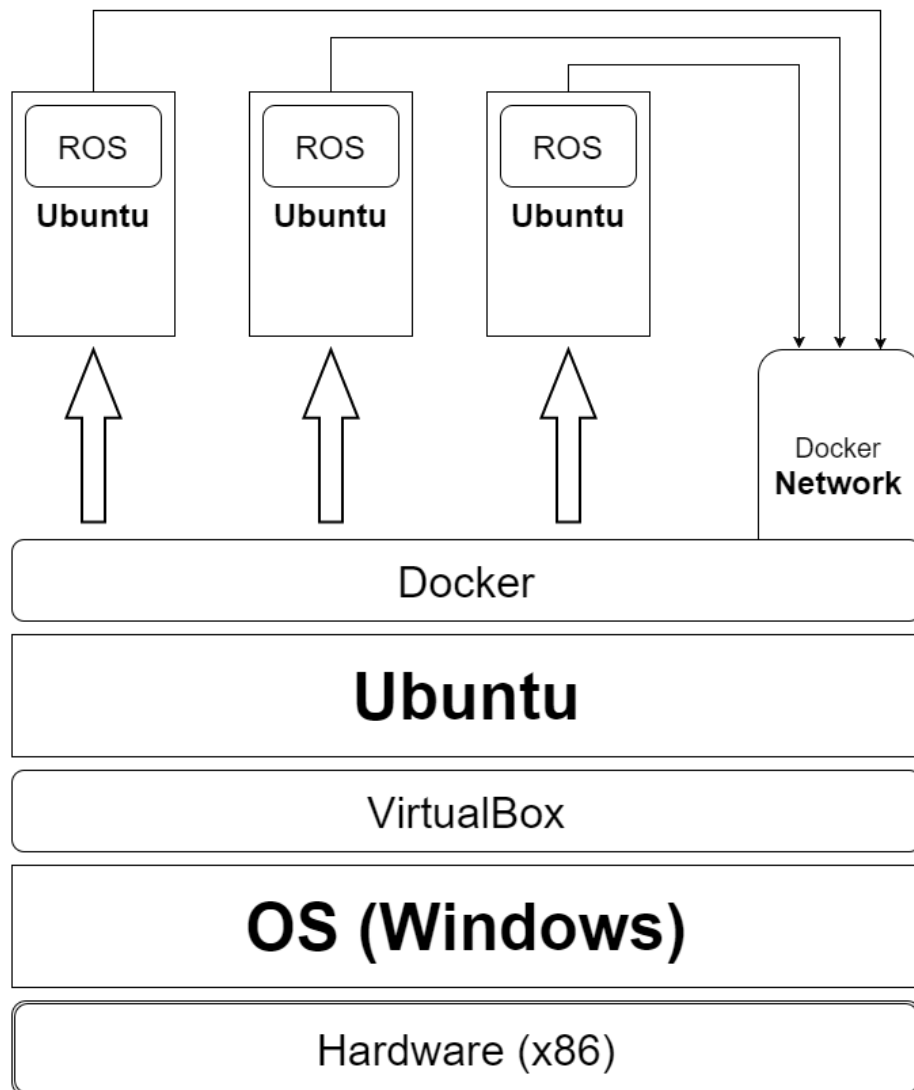


Figura 7.1.: Esquema del sistema en un ordenador x86

Posteriormente integraremos nuestro sistema en una Raspberry Pi. El esquema del sistema aplicado en una Raspberry Pi se muestra en la Figura 7.2.

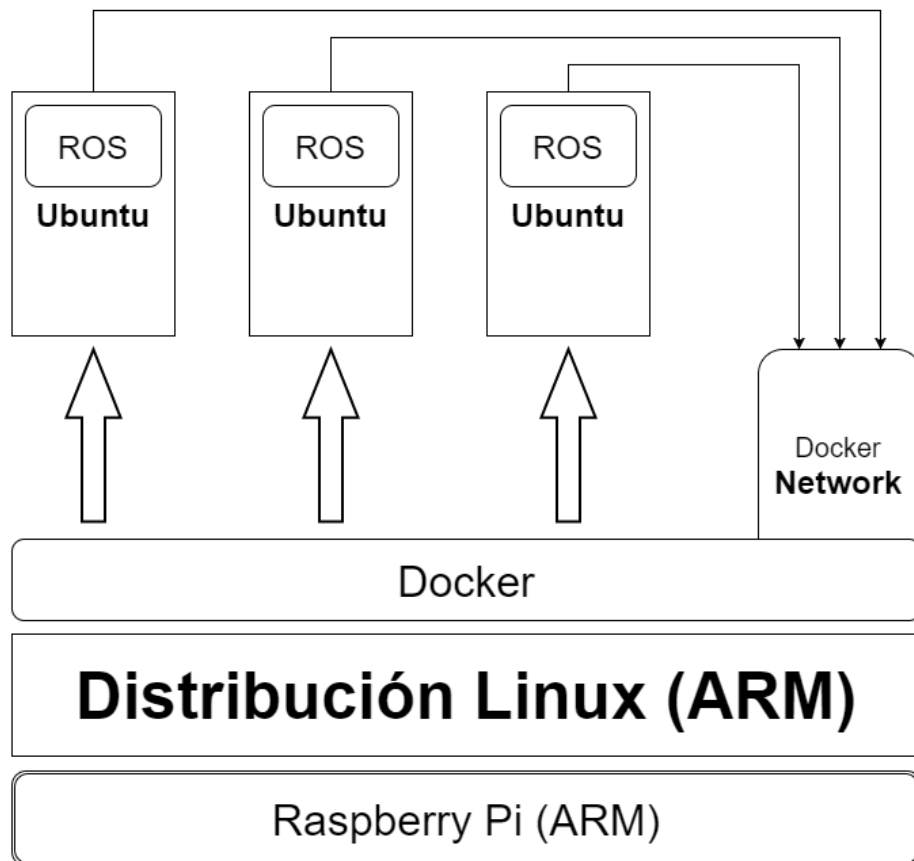


Figura 7.2.: Esquema del sistema en una Raspberry Pi

## 7.1. Crear el sistema con Docker

Lo primero que haremos será crear una serie de contenedores de docker con ROS dentro.

1. Creamos en tres terminales diferentes tres contenedores ROS.

```

1 $ docker run --name ros0 -it osrf/ros:indigo-desktop
2 $ docker run --name ros1 -it osrf/ros:indigo-desktop
3 $ docker run --name ros2 -it osrf/ros:indigo-desktop

```

2. Desde el host, creamos la red y conectamos los tres contenedores a ella.

```

1 $ docker network create red
2 a9ccfbd91df31be74881c7a7e65fbb0fdd6fec286debec6c72b1f627bb0e2ad0
3 $ docker network ls
4 NETWORK ID          NAME           DRIVER
5 a9ccfbd91df3         red            bridge

```

```

6 6fb4fab5cc04      bridge      bridge
7 a55fc7d11d74      none       null
8 2c96fadb05a4      host       host
9 $ docker network connect red ros0
10 $ docker network connect red ros1
11 $ docker network connect red ros2

```

3. Probamos el ejemplo anterior con nodos ROS pero esta vez dentro de los contenedores Docker. Simplemente hay que tener en cuenta que hay que configurar la variable *ROS\_MASTER\_URI* para que apunte a *ros0*, que es la dirección del contenedor que ejecutará *roscore*.

- a) Lanzamos *roscore* en *ros0*.

```
root@d55b47478e2c:/# roscore
```

- b) En *ros1* y *ros2* configuramos la variable que indica donde se está ejecutando *roscore*

```
$ ROS_MASTER_URI=http://ros0:11311/
```

- c) Tanto para *ros1* como para *ros2*, hace falta poner en la variable *ROS\_IP* la IP del contenedor. Esto sirve para que el *roscore* pueda encontrar los nodos. Lo haremos mirando dentro de cada contenedor la IP **correspondiente a la red creada por nosotros**. En nuestro caso se haría de la siguiente manera.

```
root@9d1dbcbf599c:~/catkin_ws# export ROS_IP=172.18.0.4
root@ee37147629e4:~/catkin_ws# export ROS_IP=172.18.0.3
```

- d) Con todo el ejemplo creado y compilado dentro de *ros1* y *ros2*, lanzamos en *ros1* el listener.

```
root@9d1dbcbf599c:~/# rosrund prueba listener
```

- e) Y probamos a escribir mediante el talker.

```
root@9d1dbcbf599c:~/# rostopic pub -1 /chatter std_msgs/String
PruebaMensaje
```

```
root@9d1dbcbf599c:~/# rosrund prueba listener
[ INFO] [1447688515.499505438]: I heard: [PruebaMensaje]
```

## 7.2. Aplicaciones para el sistema

...en desarrollo



# **Parte V.**

## **Conclusiones**

## Conclusiones

A lo largo de este documento se ha introducido al uso tanto por separado como en conjunción de las herramientas Docker y ROS. Se ha buscado darle un enfoque para hacer uso de esas herramientas en sistemas embebidos con aplicaciones robóticas. De todas maneras no ha sido ni es objetivo de este documento implementar sistemas concretos, sino servir como referencia y como guía para implementarlos.

El documento es el fruto de la investigación de ambas herramientas, apoyado con pruebas, ejemplos y explicaciones claras y comprensibles para cualquiera con conocimientos informáticos que no conozca dichas herramientas. Contiene un cuidado formato y contiene un gran número de referencias para respaldar y ampliar los datos y conceptos del documento.

### 8.1. Sobre Docker y ROS

Sobre Docker decir que, en mi opinión es el futuro de la virtualización. A diferencia de herramientas de virtualización tradicionales como VirtualBox o VMware, Docker ofrece una optimización de recursos, una simplicidad en la administración de sistemas y una escalabilidad que lo hace mucho mejor a sus competidores. Aunque quizás Docker todavía esté en un estado de desarrollo en el que aplicarlo a producción no sea todavía la mejor opción, a corto plazo pasara a ser una solución que cambiará por completo la administración de sistemas.

Sobre ROS decir que para aplicaciones robóticas es una gran solución, y mas teniendo en cuenta la proliferación en estos últimos años de microcomputadores como la Raspberry Pi, en lo que se puede por poco dinero obtener un entorno para ROS y usarlo para aplicaciones robóticas con un bajo coste. Además, teniendo en cuenta la gran cantidad de librerías y de documentación que hay al respecto, hace más sencillo meterse en un mundo tan técnico como el de la robótica.

## 8.2. Sobre la asignatura

Todo el trabajo de este documento está englobado en la asignatura de Administración de Sistemas, a la cual se le ha dado un enfoque práctico mediante un trabajo con una gran carga de investigación y de aprendizaje autónomo como éste.

por una parte trabajar con herramientas tan novedosas como Docker y poder profundizar en ellas es algo muy interesante, habiendo pasado de ni conocer la herramienta a entender su funcionamiento y comprender el enfoque de una herramienta de virtualización que difiere de las herramientas tradicionales.

Por otra parte, y en mi humilde opinión, puede echarse un poco en falta profundizar en temas como la administración de sistemas Linux tradicional o Shell Scripting, que son temas que me resultan de interés. Aunque comprendo que el tiempo es limitado y que mediante el trabajo se ha trabajado con muchas cosas como virtualización, redes, configuración de entornos, además de haber aprendido a usar una herramienta de documentación tan potente como es  $\text{\LaTeX}$ .

En definitiva puedo decir que la realización del trabajo me ha parecido una experiencia satisfactoria y con la que he aprendido mucho.

## Bibliografía

- [Docker Inc.(10 de Octubre de 2015a)] Docker Inc., 10 de Octubre de 2015a. URL <https://docs.docker.com/installation/ubuntu/linux/>.
- [Docker Inc.(10 de Octubre de 2015b)] Docker Inc., 10 de Octubre de 2015b. URL <https://docs.docker.com/reference/builder/>.
- [Docker Inc.(13 de Octubre de 2015a)] Docker Inc., 13 de Octubre de 2015a. URL <https://docs.docker.com/>.
- [Docker Inc.(13 de Octubre de 2015b)] Docker Inc., 13 de Octubre de 2015b. URL <https://docs.docker.com/userguide/dockerizing/>.
- [Docker Inc.(17 de Enero de 2016a)] Docker Inc., 17 de Enero de 2016a. URL <https://docs.docker.com/swarm/>.
- [Docker Inc.(17 de Enero de 2016b)] Docker Inc., 17 de Enero de 2016b. URL <https://docs.docker.com/swarm/install-w-machine/>.
- [Docker Inc.(23 de Octubre de 2015)] Docker Inc., 23 de Octubre de 2015. URL <https://docs.docker.com/articles/networking/>.
- [Docker Inc.(24 de Octubre de 2015)] Docker Inc., 24 de Octubre de 2015. URL <https://docs.docker.com/userguide/dockerlinks/>.
- [Docker Inc.(29 de Noviembre de 2015a)] Docker Inc., 29 de Noviembre de 2015a. URL <http://docs.docker.com/machine/>.
- [Docker Inc.(29 de Noviembre de 2015b)] Docker Inc., 29 de Noviembre de 2015b. URL <http://docs.docker.com/machine/install-machine/>.
- [Docker Inc.(29 de Septiembre de 2015)] Docker Inc., 29 de Septiembre de 2015. URL <https://www.docker.com>.
- [Open Source Robotics Foundation(27 de Octubre de 2015)] Open Source Robotics

- Foundation, 27 de Octubre de 2015. URL <http://wiki.ros.org/indigo/Installation/Ubuntu>.
- [Open Source Robotics Foundation(29 de Octubre de 2015a)] Open Source Robotics Foundation, 29 de Octubre de 2015a. URL <http://wiki.ros.org/ROS/Tutorials/CreatingPackage>.
- [Open Source Robotics Foundation(29 de Octubre de 2015b)] Open Source Robotics Foundation, 29 de Octubre de 2015b. URL [http://wiki.ros.org/catkin/Tutorials/create\\_a\\_workspace](http://wiki.ros.org/catkin/Tutorials/create_a_workspace).
- [Open Source Robotics Foundation(29 de Septiembre de 2015)] Open Source Robotics Foundation, 29 de Septiembre de 2015. URL <http://www.ros.org/>.
- [Open Source Robotics Foundation(3 de Noviembre de 2015)] Open Source Robotics Foundation, 3 de Noviembre de 2015. URL <http://wiki.ros.org/ROS/Tutorials>.
- [Tianon Gravi(10 de Octubre de 2015)] Tianon Gravi, 10 de Octubre de 2015. URL <https://github.com/tianon/docker-brew-ubuntu-core/blob/e9338b6f9ec01801bd5cc75743efe04949d123cf/trusty/Dockerfile>.
- [Universida d'Alacant(3 de Noviembre de 2015)] Universida d'Alacant, 3 de Noviembre de 2015. URL <https://moodle2014-15.ua.es/moodle/mod/wiki/view.php?pageid=701>.
- [Wikimedia Foundation Inc.(29 de Septiembre de 2015a)] Wikimedia Foundation Inc., 29 de Septiembre de 2015a. URL [https://es.wikipedia.org/wiki/Docker\\_\(Software\)](https://es.wikipedia.org/wiki/Docker_(Software)).
- [Wikimedia Foundation Inc.(29 de Septiembre de 2015b)] Wikimedia Foundation Inc., 29 de Septiembre de 2015b. URL [https://es.wikipedia.org/wiki/Sistema\\_Operativo\\_Robotico](https://es.wikipedia.org/wiki/Sistema_Operativo_Robotico).
- [Wikimedia Foundation Inc.(5 de Octubre de 2015)] Wikimedia Foundation Inc., 5 de Octubre de 2015. URL [https://es.wikipedia.org/wiki/Raspberry\\_Pi](https://es.wikipedia.org/wiki/Raspberry_Pi).