eman ta zabal zazu
Universidad del País Vasco
Euskal Herriko Unibertsitatea

Escuela Universitaria de Ingeniería Vitoria-Gasteiz
Ingeniaritzako Unibertsitate Eskola Vitoria-Gasteiz

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

I. Introducción **1**

1. Objetivo **2**

2. Herramientas utilizadas **3**
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

II. Docker **7**

3. Introducción a Docker **8**
3.1. Instalación . . . 8
3.2. Uso básico de Docker . . . 9
3.2.1. *run* . . . 11
3.2.2. *ps* . . . 11
3.2.3. *inspect* . . . 12
3.2.4. *stop* y *kill* . . . 12
3.2.5. *rm* . . . 13
3.2.6. *images* y *rmi* . . . 13
3.3. Creación de Dockerfiles . . . 14
3.4. Profundizar en Docker . . . 16

4. Redes en Docker **17**
4.1. Redes básicas en Docker . . . 17
4.1.1. *docker0* . . . 17
4.1.2. Prueba de conexión entre contenedores . . . 18
4.1.2.1. *ping* . . . 18
4.1.2.2. *netcat* . . . 20
4.1.3. Links entre contenedores . . . 21
4.2. Redes avanzadas con Docker . . . 22
4.2.1. *network* de Docker . . . 22
4.2.1.1. Prueba de *network* . . . 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 *Swarm manager* . . . 27
4.2.2.4. Configurar el Swarm y lanzar contenedores . . . 29
4.3. Configuración manual de redes . . . 32

III. ROS **34**

5. Introducción a ROS **35**
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

6. Prueba con nodos ROS **39**
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**

7. Creación del sistema **47**
7.1. Crear el sistema con Docker . . . 49
7.2. Aplaiciónes para el sistema . . . 51
7.2.1. Visión artificial . . . 51
7.2.2. Transmisión de imágenes de una Webcam . . . 51

7.2.3. Visión artificial 51
7.2.4. Visión artificial 51
7.2.5. Visión artificial 51
7.2.6. Visión artificial 51

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

1 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.

2 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 us 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 Paspberry 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 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

3 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 las siguiente manera:

```
$ docker --help
Usage: docker [OPTIONS] COMMAND [arg...]
docker daemon [ --help | ... ]
docker [ --help | -v | --version ]

A self-sufficient runtime for containers.

Options:

--config=~/.docker                 Location of client config files
-D, --debug=false                  Enable debug mode
--disable-legacy-registry=false    Do not contact legacy registries
-H, --host=[]                      Daemon socket(s) to connect to
-h, --help=false                   Print usage
-l, --log-level=info               Set the logging level
--tls=false                        Use TLS; implied by --tlsverify
--tlscacert=~/.docker/ca.pem       Trust certs signed only by this CA
--tlscert=~/.docker/cert.pem       Path to TLS certificate file
--tlskey=~/.docker/key.pem         Path to TLS key file
--tlsverify=false                  Use TLS and verify the remote
-v, --version=false                Print version information and quit

Commands:
attach    Attach to a running container
build     Build an image from a Dockerfile
commit    Create a new image from a container's changes
cp        Copy files/folders between a container and the local filesystem
create    Create a new container
```

```
diff      Inspect changes on a container's filesystem
events    Get real time events from the server
exec      Run a command in a running container
export    Export a container's filesystem as a tar archive
history   Show the history of an image
images    List images
import    Import the contents from a tarball to create a filesystem image
info      Display system-wide information
inspect   Return low-level information on a container or image
kill      Kill a running container
load      Load an image from a tar archive or STDIN
login     Register or log in to a Docker registry
logout    Log out from a Docker registry
logs      Fetch the logs of a container
network   Manage Docker networks
pause     Pause all processes within a container
port      List port mappings or a specific mapping for the CONTAINER
ps        List containers
pull      Pull an image or a repository from a registry
push      Push an image or a repository to a registry
rename    Rename a container
restart   Restart a container
rm        Remove one or more containers
rmi       Remove one or more images
run       Run a command in a new container
save      Save an image(s) to a tar archive
search    Search the Docker Hub for images
start     Start one or more stopped containers
stats     Display a live stream of container(s) resource usage statistics
stop      Stop a running container
tag       Tag an image into a repository
top       Display the running processes of a container
unpause   Unpause all processes within a container
version   Show the Docker version information
volume    Manage Docker volumes
wait      Block until a container stops, then print its exit code

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 contendores en ejecución. Para mostrar todos, se hace uso hay que indicarselo 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.

```
$ docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED
                STATUS              PORTS               NAMES
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 maquina. 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.

```
$ docker inspect adoring_euclid
# ...
# ... Se omite la salida por ser demasiado grande
# ...
$ docker inspect --format='{{.NetworkSettings.IPAddress}}' adoring_euclid
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.

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

```
$ docker stop 40f5c913912e
40f5c913912e
$ docker ps
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.

```
$ docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED
              STATUS              PORTS               NAMES
$ docker run -d ubuntu:14.04 /bin/sh -c "while true; do echo hello world;
   sleep 1; done"
5abaece69cbd445e69cae61cef6de9f42e2561eacb4b8969024c922eec348a5d
$ docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED
              STATUS              PORTS               NAMES
5abaece69cbd        ubuntu:14.04        "/bin/sh -c 'while tr"   3 seconds
    ago       Up 2 seconds                            mad_ardinghelli
$ docker stop mad_ardinghelli
mad_ardinghelli
$ docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED
              STATUS              PORTS               NAMES
$ docker ps -a
CONTAINER ID        IMAGE               COMMAND                  CREATED
              STATUS                      PORTS               NAMES
5abaece69cbd        ubuntu:14.04        "/bin/sh -c 'while tr"   5 minutes
    ago       Exited (137) 4 minutes ago
   mad_ardinghelli
$ docker rm mad_ardinghelli
mad_ardinghelli
$ docker ps -a
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***.

```
$ docker images
REPOSITORY          TAG                 IMAGE ID            CREATED
                VIRTUAL SIZE
ubuntu              latest              a005e6b7dd01        18 hours ago
            188.4 MB
ros                 latest              67110eef39cf        7 weeks ago
             826.7 MB
hello-world         latest              af340544ed62        9 weeks ago
             960 B
$ docker rmi -f hello-world
Untagged: hello-world:latest
Deleted: af340544ed62de0680f441c71fa1a80cb084678fed42bae393e543faea3a572c
Deleted: 535020c3e8add9d6bb06e5ac15a261e73d9b213d62fb2c14d752b8e189b2b912
$ docker images
REPOSITORY          TAG                 IMAGE ID            CREATED
                VIRTUAL SIZE
ubuntu              latest              a005e6b7dd01        18 hours ago
            188.4 MB
ros                 latest              67110eef39cf        7 weeks ago
             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)].

```
FROM scratch
ADD ubuntu-trusty-core-cloudimg-amd64-root.tar.gz /

# a few minor docker-specific tweaks
```

```
# see https://github.com/docker/docker/blob/master/contrib/mkimage/debootstrap
RUN echo '#!/bin/sh' > /usr/sbin/policy-rc.d \
&& echo 'exit 101' >> /usr/sbin/policy-rc.d \
&& chmod +x /usr/sbin/policy-rc.d \
\
&& dpkg-divert --local --rename --add /sbin/initctl \
&& cp -a /usr/sbin/policy-rc.d /sbin/initctl \
&& sed -i 's/^exit.*/exit 0/' /sbin/initctl \
\
&& echo 'force-unsafe-io' > /etc/dpkg/dpkg.cfg.d/docker-apt-speedup \
\
&& 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 \
&& 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 \
&& echo 'Dir::Cache::pkgcache ""; Dir::Cache::srcpkgcache "";' >> /etc/apt/apt.conf.d/docker-clean \
\
&& echo 'Acquire::Languages "none";' > /etc/apt/apt.conf.d/docker-no-languages \
\
&& echo 'Acquire::GzipIndexes "true"; Acquire::CompressionTypes::Order:: "gz";' > /etc/apt/apt.conf.d/docker-gzip-indexes

# enable the universe
RUN sed -i 's/^#\s*\(deb.*universe\)$/\1/g' /etc/apt/sources.list

# overwrite this with 'CMD []' in a dependent Dockerfile
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 me 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 ver 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 esencia para poder lanzarnos a construir nuestro sistema.

4 Redes en Docker

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 *ntework* 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 canda 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
     CREATED             STATUS              PORTS               NAMES
3 829a49bb2cfa        osrf/ros:indigo-desktop   "/ros_entrypoint.sh /"   6
     seconds ago       Up 6 seconds
     compassionate_mccarthy
4 2f3c19da0cb8        osrf/ros:indigo-desktop   "/ros_entrypoint.sh /"   16
     seconds ago      Up 16 seconds                           grave_mahavira
```

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
```

```
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.

```
root@2f3c19da0cb8:/# ping 172.17.0.5
PING 172.17.0.5 (172.17.0.5) 56(84) bytes of data.
64 bytes from 172.17.0.5: icmp_seq=1 ttl=64 time=0.085 ms
64 bytes from 172.17.0.5: icmp_seq=2 ttl=64 time=0.058 ms
64 bytes from 172.17.0.5: icmp_seq=3 ttl=64 time=0.061 ms
64 bytes from 172.17.0.5: icmp_seq=4 ttl=64 time=0.060 ms
64 bytes from 172.17.0.5: icmp_seq=5 ttl=64 time=0.106 ms
64 bytes from 172.17.0.5: icmp_seq=6 ttl=64 time=0.135 ms
^C
--- 172.17.0.5 ping statistics ---
6 packets transmitted, 6 received, 0% packet loss, time 4997ms
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.

```
root@829a49bb2cfa:/# ping grave_mahavira
PING grave_mahavira (172.17.0.4) 56(84) bytes of data.
64 bytes from grave_mahavira.bridge (172.17.0.4): icmp_seq=1 ttl=64 time
    =0.087 ms
64 bytes from grave_mahavira.bridge (172.17.0.4): icmp_seq=2 ttl=64 time
    =0.066 ms
64 bytes from grave_mahavira.bridge (172.17.0.4): icmp_seq=3 ttl=64 time
    =0.066 ms
64 bytes from grave_mahavira.bridge (172.17.0.4): icmp_seq=4 ttl=64 time
    =0.067 ms
64 bytes from grave_mahavira.bridge (172.17.0.4): icmp_seq=5 ttl=64 time
    =0.066 ms
64 bytes from grave_mahavira.bridge (172.17.0.4): icmp_seq=6 ttl=64 time
    =0.064 ms
^C
--- grave_mahavira ping statistics ---
6 packets transmitted, 6 received, 0% packet loss, time 5001ms
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 cuanta a la hora de renombrar los contenedores. Tampoco podemos cambiar el nombre de un contenedor durante su ejecucíon, 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 contendores el 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*.

```
$ docker inspect -f "{{ .HostConfig.Links }}" ros2
[/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 verión 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 contedores 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 contendores 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.

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

```
root@969d7f0c6a5b:/# ping ubuntu1
PING ubuntu1 (172.19.0.2) 56(84) bytes of data.
64 bytes from ubuntu1 (172.19.0.2): icmp_seq=1 ttl=64 time=0.062
    ms
64 bytes from ubuntu1 (172.19.0.2): icmp_seq=2 ttl=64 time=0.077
    ms
64 bytes from ubuntu1 (172.19.0.2): icmp_seq=3 ttl=64 time=0.060
    ms
64 bytes from ubuntu1 (172.19.0.2): icmp_seq=4 ttl=64 time=0.067
    ms
^C
--- ubuntu1 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 2999ms
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 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 virtualizacián para crear más VMs dentro de ella es un proceso redundate 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
```

```
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*.

```
$ docker-machine create -d virtualbox local
Running pre-create checks...
Creating machine...
(local) Copying C:\Users\ander\.docker\machine\cache\boot2docker.iso
   to C:\Users\ander\.docker\machine\machines\local\boot2docker.iso
   ...
(local) Creating VirtualBox VM...
(local) Creating SSH key...
(local) Starting the VM...
(local) Waiting for an IP...
Waiting for machine to be running, this may take a few minutes...
Machine is running, waiting for SSH to be available...
Detecting operating system of created instance...
Detecting the provisioner...
Provisioning with boot2docker...
Copying certs to the local machine directory...
Copying certs to the remote machine...
Setting Docker configuration on the remote daemon...
Checking connection to Docker...
Docker is up and running!
To see how to connect 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.

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

4. Generamos un *discovery token* mediante la imagen de Swarm de Docker. Este token nos servira 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.

```
$ docker run swarm create
Unable to find image 'swarm:latest' locally
latest: Pulling from swarm
d681c900c6e3: Pull complete
188de6f24f3f: Pull complete
90b2ffb8d338: Pull complete
237af4efea94: Pull complete
```

```
3b3fc6f62107: Pull complete
7e6c9135b308: Pull complete
986340ab62f0: Pull complete
a9975e2cc0a3: Pull complete
Digest: sha256:
   c21fd414b0488637b1f05f13a59b032a3f9da5d818d31da1a4ca98a84c0c781b
Status: Downloaded newer image for swarm:latest
dc60acd12fc3a6b14754abef91501be2
```

5. Guarmamos el token que nos devuelve ya que lo necesitaremos para más adelante. En nuetro 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.

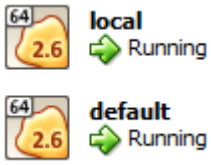

Figura 4.1.: Imágenes GNU/Linux creadas

4.2.2.3. Lanzar el *Swarm manager*

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

Despues 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.

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

```
(swarm-master) Creating SSH key...
(swarm-master) Starting the VM...
(swarm-master) Waiting for an IP...
Waiting for machine to be running, this may take a few minutes...
Machine is running, waiting for SSH to be available...
Detecting operating system of created instance...
Detecting the provisioner...
Provisioning with boot2docker...
Copying certs to the local machine directory...
Copying certs to the remote machine...
Setting Docker configuration on the remote daemon...
Configuring swarm...
Checking connection to Docker...
Docker is up and running!
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*.

```
$ docker-machine create -d virtualbox --swarm --swarm-discovery token
    ://dc60acd12fc3a6b14754abef91501b
e2 swarm-agent-00
Running pre-create checks...
Creating machine...
(swarm-agent-00) Copying C:\Users\ander\.docker\machine\cache\
    boot2docker.iso to C:\Users\ander\.docker\machine\machines\swarm-
    agent-00\boot2docker.iso...
(swarm-agent-00) Creating VirtualBox VM...
(swarm-agent-00) Creating SSH key...
(swarm-agent-00) Starting the VM...
(swarm-agent-00) Waiting for an IP...
Waiting for machine to be running, this may take a few minutes...
Machine is running, waiting for SSH to be available...
Detecting operating system of created instance...
Detecting the provisioner...
Provisioning with boot2docker...
Copying certs to the local machine directory...
Copying certs to the remote machine...
Setting Docker configuration on the remote daemon...
Configuring swarm...
Checking connection to Docker...
Docker is up and running!
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*.

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

```
e2 swarm-agent-01
Running pre-create checks...
Creating machine...
(swarm-agent-01) Copying C:\Users\ander\.docker\machine\cache\
   boot2docker.iso to C:\Users\ander\.docker\machine\machines\swarm-
   agent-01\boot2docker.iso...
(swarm-agent-01) Creating VirtualBox VM...
(swarm-agent-01) Creating SSH key...
(swarm-agent-01) Starting the VM...
(swarm-agent-01) Waiting for an IP...
Waiting for machine to be running, this may take a few minutes...
Machine is running, waiting for SSH to be available...
Detecting operating system of created instance...
Detecting the provisioner...
Provisioning with boot2docker...
Copying certs to the local machine directory...
Copying certs to the remote machine...
Setting Docker configuration on the remote daemon...
Configuring swarm...
Checking connection to Docker...
Docker is up and running!
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.

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

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

```
$ docker info
Containers: 4
Images: 3
Role: primary
Strategy: spread
Filters: health, port, dependency, affinity, constraint
Nodes: 3
 swarm-agent-00: 192.168.99.103:2376
   Status: Healthy
   Containers: 1
   Reserved CPUs: 0 / 1
   Reserved Memory: 0 B / 1.021 GiB
   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
 swarm-agent-01: 192.168.99.104:2376
   Status: Healthy
   Containers: 1
   Reserved CPUs: 0 / 1
   Reserved Memory: 0 B / 1.021 GiB
   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
 swarm-master: 192.168.99.102:2376
   Status: Healthy
   Containers: 2
   Reserved CPUs: 0 / 1
   Reserved Memory: 0 B / 1.021 GiB
   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
CPUs: 3
Total Memory: 3.064 GiB
Name: swarm-master
```

Se puede observar que tanto el manager como lon 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.

```
$ docker ps -a
CONTAINER ID        IMAGE               COMMAND
   CREATED             STATUS              PORTS               NAMES
3f228154f8ff        swarm:latest        "/swarm join --advert"   20
   minutes ago      Up 20 minutes                           swarm-
   agent-01/swarm-agent
e9df36929545        swarm:latest        "/swarm join --advert"   22
   minutes ago      Up 22 minutes                           swarm-
   agent-00/swarm-agent
a18b76c0013d        swarm:latest        "/swarm join --advert"   27
   minutes ago      Up 27 minutes                           swarm-
   master/swarm-agent
35fcf51df08a        swarm:latest        "/swarm manage --tlsv"   27
   minutes ago      Up 27 minutes                           swarm-
   master/swarm-agent-master
```

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

```
$ docker run hello-world

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

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

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

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

For more examples and ideas, visit:
https://docs.docker.com/userguide/
```

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

```
$ docker ps -a
CONTAINER ID        IMAGE               NAMES
91a89b22a8b1        hello-world         swarm-agent-00/sleepy_perlman
3f228154f8ff        swarm:latest        swarm-agent-01/swarm-agent
e9df36929545        swarm:latest        swarm-agent-00/swarm-agent
a18b76c0013d        swarm:latest        swarm-master/swarm-agent
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.

```
--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
--icc=true|false # Indica si se permite la comunicacion entre contenedores
    , por defecto true
--ipv6=true|false # Define si se usa IPv6, por defecto false
--ip-forward=true|false # Indica si esta activada la comunicacion entre
    los contenedores y el exterior, por defecto true
--iptables=true|false # Define si se perminte 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

5 Introducción a ROS

Anteriormente se ha explicado qué es ROS y que características tiene. En este capitulo se pasara 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
- ***rosnode***: nos proporciona información sobre un nodo. Disponemos de las siguientes opciones:
  - *rosnode info [nodo]*: muestra información sobre el nodo
  - *rosnode kill [nodo]*: mata ese proceso
  - *rosnode list*: muestra los nodos ejecutándose
  - *rosnode machine [maquina]*: muestra los nodos que se están ejecutando en la máquina
  - *rosnode ping [nodo]*: comprueba la conectividad del nodo
- ***rosrun*** *[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)].

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

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

```
$ cd ~/catkin_ws/src
$ 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.

```
$ cd ~/catkin_ws/
$ 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.

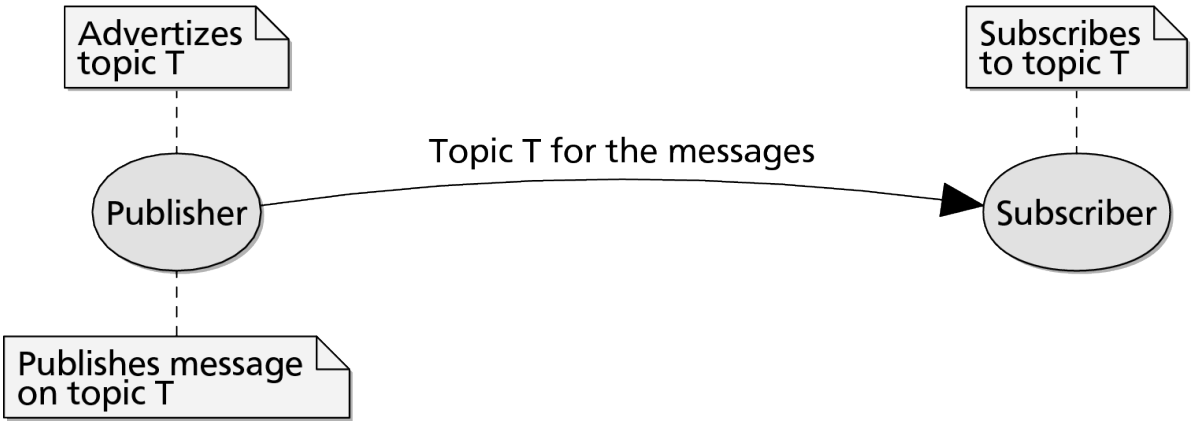

Figura 5.1.: Modelo Publisher-Subscriber

6 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 carácteres.

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");
```

```
  ros::NodeHandle n;
  ros::Publisher chatter_pub = n.advertise<std_msgs::String>("chatter",
  ↪  1000);
  ros::Rate loop_rate(10);

  int count = 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;
  }

  return 0;
}
```

6.1.2. Subscriber

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

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

int main(int argc, char **argv)
{
  ros::init(argc, argv, "listener");
  ros::NodeHandle n;
  ros::Subscriber sub = n.subscribe("chatter", 1000, chatterCallback);
  ros::spin();
  return 0;
}
```

6.1.3. 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)
```

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.

```
~$ mkdir -p ~/catkin_ws/src
~$ cd ~/catkin_ws/src
~/catkin_ws/src$ catkin_init_workspace
Creating symlink "/home/ander/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 prueba std_msgs rospy roscpp
Created file prueba/package.xml
Created file prueba/CMakeLists.txt
Created folder prueba/include/prueba
Created folder prueba/src
```

```
Successfully created files in /home/ander/catkin_ws/src/prueba. Please
    adjust the values in package.xml.
~/catkin_ws/src$ ls
CMakeLists.txt  prueba
~/catkin_ws/src$ cd ..
~/catkin_ws$ catkin_make
Base path: /home/ander/catkin_ws
Source space: /home/ander/catkin_ws/src
Build space: /home/ander/catkin_ws/build
Devel space: /home/ander/catkin_ws/devel
Install space: /home/ander/catkin_ws/install
####
#### 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"
####
-- 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/ander/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/ander/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:
```

```
-- ~~  - prueba
-- ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
-- +++ processing catkin package: 'pruebaROS'
-- ==> add_subdirectory(pruebaROS)
-- Configuring done
-- Generating done
-- Build files have been written to: /home/ander/catkin_ws/build
####
#### Running command: "make -j1 -l1" in "/home/ander/catkin_ws/build"
####
~/catkin_ws$ ls
build  devel  src
~/catkin_ws$ cd src
~/catkin_ws/src$ ls
CMakeLists.txt  prueba
~/catkin_ws/src$ cd prueba/
~/catkin_ws/src/prueba$ 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
```

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.

```
~/catkin_ws/src/prueba/src$ nano listener.cpp
~/catkin_ws/src/prueba/src$ nano talker.cpp
~/catkin_ws/src/prueba/src$ cd ..
~/catkin_ws/src/prueba$ ls
CMakeLists.txt  include  package.xml  src
~/catkin_ws/src/prueba$ echo "" > CMakeLists.txt # Para vaciar el archivo
~/catkin_ws/src/prueba$ nano CMakeLists.txt
```

Una vez llenos podemos proceder a compilarlo.

```
~/catkin_ws/src/prueba$ cd ..
~/catkin_ws/src$ cd ..
~/catkin_ws$ ls
build  devel  src
~/catkin_ws$ catkin_make
ander@ubuntu-VirtualBox:~/catkin_ws$ catkin_make
Base path: /home/ander/catkin_ws
Source space: /home/ander/catkin_ws/src
Build space: /home/ander/catkin_ws/build
Devel space: /home/ander/catkin_ws/devel
Install space: /home/ander/catkin_ws/install
```

```
####
#### 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"
####
-- Using CATKIN_DEVEL_PREFIX: /home/ander/catkin_ws/devel
-- Using CMAKE_PREFIX_PATH: /home/ander/catkin_ws/devel;/opt/ros/indigo
-- This workspace overlays: /home/ander/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/ander/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:
-- ~~  - prueba
-- ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
-- +++ processing catkin package: 'prueba'
-- ==> add_subdirectory(prueba)
-- Configuring done
-- Generating done
-- Build files have been written to: /home/ander/catkin_ws/build
####
#### Running command: "make -j1 -l1" in "/home/ander/catkin_ws/build"
####
Scanning dependencies of target listener
[ 50%] Building CXX object prueba/CMakeFiles/listener.dir/src/listener.cpp
    .o
Linking CXX executable /home/ander/catkin_ws/devel/lib/prueba/listener
[ 50%] Built target listener
Scanning dependencies of target talker
[100%] Building CXX object prueba/CMakeFiles/talker.dir/src/talker.cpp.o
Linking CXX executable /home/ander/catkin_ws/devel/lib/prueba/talker
[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*.

```
$ roscore
```

2. Compilamos de nuevo el workspace que tenemos creado.

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

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

```
$ rosrun prueba listener
```

4. Mandamos un mensaje mediante el *talker*.

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

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

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

Parte IV.

Implementación del sistema

7 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 despligue 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.

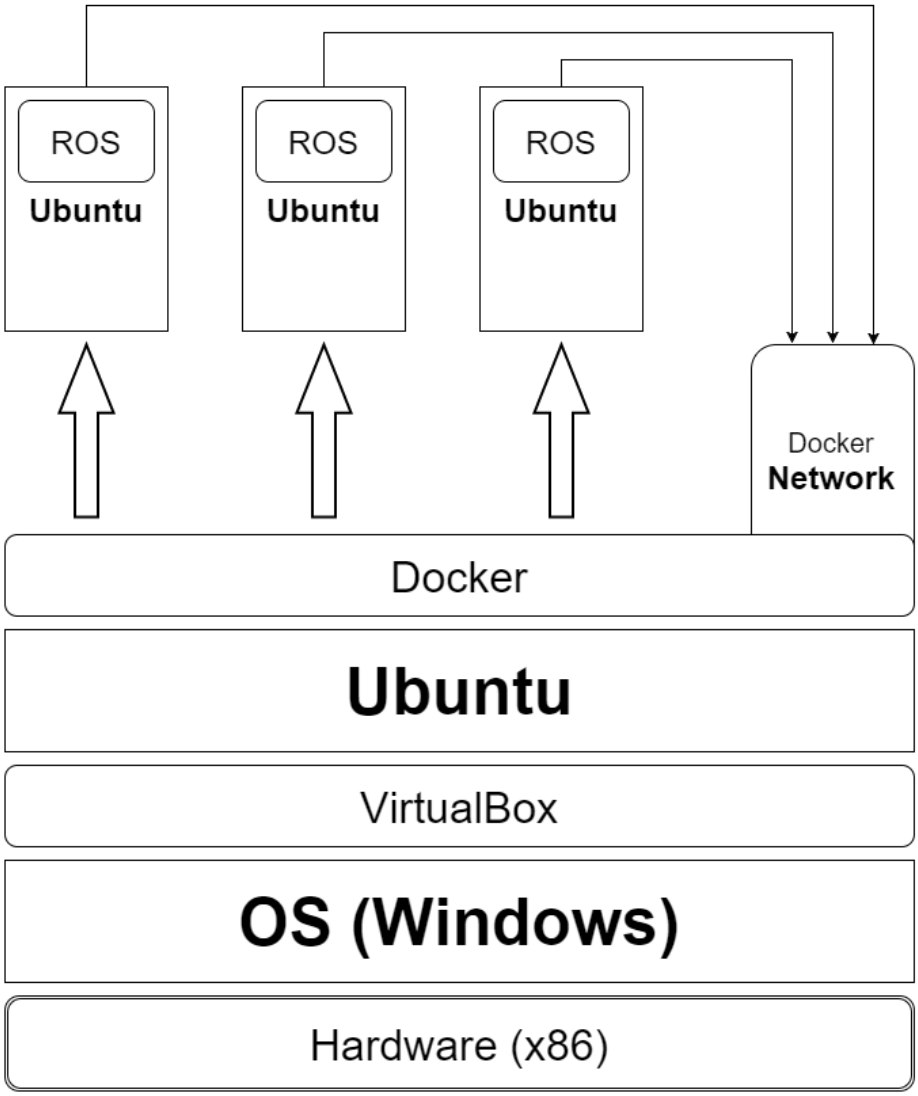

Figura 7.1.: Esquema del sistema en un ordenador x86

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

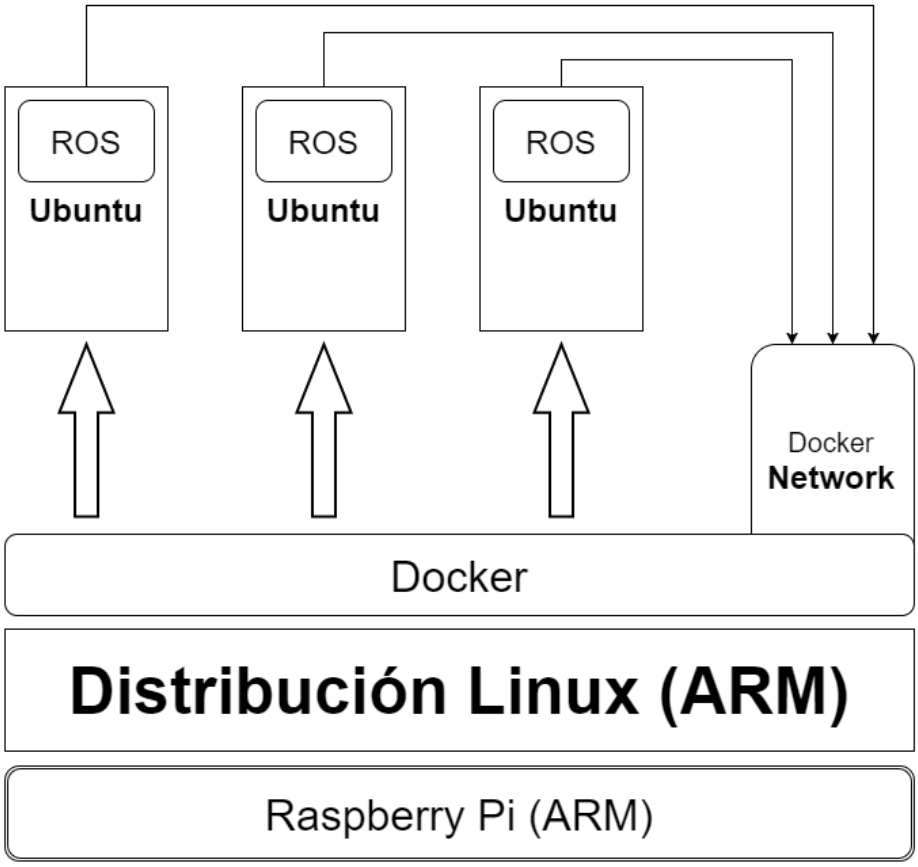

Figura 7.2.: Esquema del sistema en una Raspberry Pi

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.

```
$ docker run --name ros0 -it osrf/ros:indigo-desktop
$ docker run --name ros1 -it osrf/ros:indigo-desktop
$ docker run --name ros2 -it osrf/ros:indigo-desktop
```

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

```
$ docker network create red
a9ccfbd91df31be74881c7a7e65fbb0fdd6fec286debec6c72b1f627bb0e2ad0
$ docker network ls
NETWORK ID          NAME                DRIVER
a9ccfbd91df3        red                 bridge
```

```
6fb4fab5cc04        bridge              bridge
a55fc7d11d74        none                null
2c96fadb05a4        host                host
$ docker network connect red ros0
$ docker network connect red ros1
$ 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 conentedor. 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:~/# rosrun prueba listener
   ```

   e) Y probamos a escribir mediante el talker.

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

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

7.2. Aplaiciónes para el sistema

... en desarrollo

7.2.1. Visión artificial

7.2.2. Transmisión de imágenes de una Webcam

7.2.3. Visión artificial

7.2.4. Visión artificial

7.2.5. Visión artificial

7.2.6. Visión artificial

Bibliografía

[Docker Inc.(10 de Octubre de 2015a)] Docker Inc., 10 de Octubre de 2015a. URL https://docs.docker.com/installation/ubuntulinux/.

[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.

[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).

[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.

[Wikimedia Foundation Inc.(5 de Octubre de 2015)] Wikimedia Foundation Inc., 5 de Octubre de 2015. URL https://es.wikipedia.org/wiki/Raspberry_Pi.