Documentación instalación

Comandos

A continuación, se van a detallar el conjunto de comandos e instrucciones en las que se basa el funcionamiento de Docker, con la intención de servir de guía introductoria al funcionamiento de esta herramienta.

Para poder crear las imágenes, sobre las que se ejecutarán los contenedores, se debe tener en cuenta que su creación requiere un archivo DockerFile para cada una de ellas, ubicado dentro de la carpeta donde se encuentran los .py.

Esta determinada ubicación viene relacionada con el conjunto de instrucciones que este archivo contiene, y cómo referencian a los archivos que engloba la imagen.

Una vez que el documento Dockerfile contiene todos los comandos que se quieren ejecutar, y que deben ser adaptados para cada una de las imágenes, se puede proceder con la creación de las imágenes en el ordenador, de manera local, para posteriormente subirlas a Docker Hub.

Los pasos a seguir son:

1. `docker login` → Primero de todo, y a modo de comprobación por si el registro no está bien hecho, se requiere iniciar con la cuenta de Docker Hub dentro del terminal

2. `docker build --platform /linux/arm64/v8 -t "nombre imagen":"version" .` → Este comando crea la imagen con un nombre determinado, indicando que la plataforma donde será utilizada es Linux arm64, siendo la ejecutada dentro de la RPi

3. `docker tag "nombre imagen antiguo":"versión" "nombre usuario Docker Hub"/"nuevo nombre imagen":"versión"` → Si este paso fuera necesario, adapta el nombre de la imagen antes de subirla a Docker Hub, al formato que la web requiere para su subida

4. `docker push "nombre imagen":"versión"` → Una vez la imagen ya está creada, y tiene el nombre con el formato correcto, esta puede ser subida a DockerHub para que pueda ser obtenida con el docker-compose.yml posteriormente.

Hay que tener en cuenta que tanto el paso 2 como el 4 pueden llevar su tiempo, pero sobre todo el 2, ya que hay que entender que se están instalando todas las dependencias solicitadas dentro de la imagen, para que esta pueda ser completamente operativa en su despliegue en la RPi.

La creación de imágenes de Docker, una vez la primera versión ya ha sido generada, funciona como una pila de capas, las cuales se modifican o no según

si la diferencia entre diferentes versiones es mayor o no. Por ello, el tiempo de creación de las imágenes variará según las diferencias con la imagen anterior.

Si en ningún momento se experimenta ningún error, y el conjunto de instalaciones se van ejecutando correctamente, únicamente hay que tener paciencia y esperar.

Cuando las imágenes ya están creadas y subidas a Docker Hub, se debe proceder a la configuración de la RPi, en el caso de no estar ya configurada, o a la descarga de las imágenes y despliegue de los contenedores en caso de ya estar en funcionamiento todo el entorno del dispositivo.

En el caso de que configurar la RPi sea necesario, al asumir que se inicia desde 0 este proceso, será necesario ir al documento denominado como "Transversal Project Guide", ubicado dentro del GitHub del ecosistema de drones, y realizar la instalación pertinente, ya que a lo largo de este documento se asumirá que esta configuración ya está hecha.

Instalación Docker

Para poder trabajar con Docker y Docker Compose, y así poder obtener los contenedores necesarios para la instalación de los servicios, se requiere instalar esta herramienta dentro del sistema siguiendo las siguientes instrucciones:

1. `curl -sSL https://get.docker.com | sh`
2. `sudo usermod -aG docker ubuntu`
3. `newgrp docker`
4. `sudo apt install docker-compose`
5. `sudo systemctl enable docker`

Una vez hecha esta instalación, es necesario realizar algunas comprobaciones antes de poner en marcha los contenedores utilizando Docker. Utilizando un archivo docker-compose.yml bien configurado, se sabe que los diferentes puertos de la RPi serán accesibles desde los contenedores desplegados, pero dos comprobaciones deben ser realizadas en la Raspberry Pi antes de probar el funcionamiento de la cámara, en el caso de que vaya a ser utilizada, para asegurar que el usuario tiene permiso para acceder a ella:

1. `sudo usermod -aG video ubuntu`
2. `sudo modprobe bcm2835-v4l2`

Al mismo tiempo, se debe editar el fichero /boot/firmware/config.txt y añadir la siguiente línea de configuración:

- `start_x=1`

En el caso de querer hacer una prueba de conexión con el autopiloto del dron, y dado a que las dependencias necesarias para su uso serán instaladas dentro de un contenedor de Docker, se deberían instalar las siguientes dependencias dentro del inicio de la RPi:

- pip3 install dronekit==2.9.1

- pip3 install pymavlink==2.2.10
- pip3 install mavproxy==1.6.4

Una vez instaladas este, es posible correr un código de ejemplo, como el de la Activity #18 del “Transversal Project Guide” que muestre si, como es de esperar, se tiene acceso al autopiloto del dron en /dev/ttyS0

Cuando Docker ya está instalado, se debe enviar el fichero docker-compose.yml via ftp a la RPi, o generar un archivo manualmente con este nombre dentro de ella, con la configuración necesaria, y una vez este ya se encuentre dentro del sistema, los comandos a seguir son:

1. `docker-compose up` (Arranca los contenedores) o `docker-compose up -d` (Para arrancar los contenedores sin ver los logs en el terminal)
2. `docker ps -a` (Ver estado de los contenedores arrancados, si es requerido)
3. `docker-compose stop` (Parar los contenedores)
4. `docker-compose down` (Parar los contenedores y eliminarlos, sin eliminar las imágenes asociadas a ellos)

Para identificar la estructura que debe tener un archivo docker-compose.yml, en la **Fig 1** se puede ver uno de ellos en la estructura actual de los servicios de a bordo, en el momento en el que se está redactando esta documentación.

```
version: "3.3"
services:
  monitor:
    #build: ./MonitorDEE
    image: jordillaveria/monitor_arm64:v2
    # Connection can be global, local or classpip
    command: ["python3", "monitor.py", "global"]
    # command: ["python3", "monitor.py", "classpip"]
    container_name: monitor
    networks:
      main:
        ipv4_address: 192.168.208.3
    depends_on:
      - mosquitto
    external_links:
      - mosquitto

  services:
    #build: ./DroneEngineeringEcosystemDEE
    image: jordillaveria/services_arm64:v11
    privileged: true
    container_name: services
    command: ["python3", "boot.py", "broker.hivemq.com"]
    #command: ["python3", "boot.py", "classpip.upc.edu", "dronsEETAC", "mimara1456."]
    #command: ["tail", "-f", "/dev/null"]
    networks:
      main:
        ipv4_address: 192.168.208.4
    depends_on:
      - mosquitto
    external_links:
      - mosquitto
    devices:
      - /dev/video0:/dev/video0
      - /dev/ttyS0:/dev/ttyS0
      - /dev/gpiomem:/dev/gpiomem

  restapi:
    image: jordillaveria/restapi_arm64:v2
    restart: always
    container_name: restapi
    command: ["uvicorn", "main:app", "--host", "0.0.0.0", "--port", "9000"]
    ports:
      - "9000:9000"
    networks:
      main:
        ipv4_address: 192.168.208.6

  mosquitto:
    image: eclipse-mosquitto:2.0.5
    container_name: mosquitto
    ports:
      - "1883:1883"
      - "1884:1884"
      - "8000:8000"
    volumes:
      - ./config:/mosquitto/config
      - ./data:/mosquitto/data
      - ./log:/mosquitto/log
    networks:
      main:
        ipv4_address: 192.168.208.2

  mongo:
    image: mongo:bionic
    container_name: mongo
    restart: always
    volumes:
      - mongodb_data:/data/db
    networks:
      main:
        ipv4_address: 192.168.208.5
    command: mongod --bind_ip_all

networks:
  main:
    driver: bridge
    ipam:
      config:
        - subnet: 192.168.208.0/20

volumes:
  mongodb_data:

secrets:
  dockerhub_username:
    external: true
  dockerhub_password:
    external: true
```

Fig 1. Archivo “docker.compose.yml”

Para aclarar respecto a la **Fig 1**, el servicio “mosquitto” va directamente seguido del denominado como “raspi”, aunque para no ocupar mucho espacio de esta documentación, se ha representado el documento en dos columnas separadas.

Hay que tener en cuenta que el comando `docker-compose up -d` instalará todos los servicios indicados en él como contenedores, y al mismo tiempo, se le puede indicar qué comandos quiere que se ejecuten al arrancar estos contenedores, como parte de las propiedades de configuración. Como referencia, dos de los comandos principales al arrancar la imagen de *services* son:

- `command: [“python3”, “boot.py”, “broker.hivemq.com”]` → Arrancará automáticamente el archivo boot.py, arrancando los servicios como se hacía anteriormente manualmente
- `command: [“tail”, “-f”, “/dev/null”]` → Mantiene el contenedor en estado “UP”, para poder acceder a él de manera remota, pero sin arrancar el archivo de boot.py, que deberá ser arrancado de manera manual una vez dentro del contenedor.

Respecto al contenedor “mosquitto”, hay que saber que este arranca el bróker obteniendo la configuración de un archivo denominado como “mosquitto.conf”, que se debe encontrar dentro de la carpeta config que se crea al ejecutar por primera vez este archivo, o puede ser creada previamente de manera manual siguiendo los siguientes pasos:

1. `mkdir config` → Crea la carpeta donde ubicar el archivo

2. `nano mosquitto.conf` → Crea y permite la edición del archivo denominado como “mosquitto.conf”

3. Añadir la configuración de que se muestra en la **Fig 2**

A continuación, se muestra en la **Fig 2** la configuración requerida dentro de este archivo “mosquitto.conf”, para permitir al contenedor de Docker que contiene Mosquitto trabajar tanto con un bróker interno, como con un externo:

```
listener 1884
allow_anonymous true
listener 1883
allow_anonymous true
listener 8000
protocol websockets
allow_anonymous true
```

Fig 2. Configuración del archivo de Mosquitto

Como último comando útil para el uso de Docker, para acceder a uno de los contenedores, y poder interactuar con la información que ellos contienen, se puede ejecutar el comando:

- `docker exec -it "nombre contenedor" /bin/bash`

Hay que tener en cuenta que el contenedor debe estar en estado "UP" para poder ser accesible, si no es el caso, saltará un error de que el acceso no está permitido o similar.

Una vez ubicado dentro del contenedor ejecutando el comando previo, hay que entender que todas las librerías necesarias para la ejecución de los archivos que contiene ya han sido instaladas, tanto por las instaladas utilizando comandos en el Dockerfile, como los requerimientos necesarios que se establecen en el requirements.txt.

Por otro lado, cualquier modificación de estos que se haga de manera manual, y sin generar una versión nueva de la imagen asociada, implicará que una vez que el contenedor se pare, o se elimine, estos cambios se perderán.

Esto se debe a que Docker trabaja con imágenes de solo lectura, no editables, sino que requiere volver a generarlas ante cualquier cambio que se les realice.

Errores comunes

La intención de este apartado es dejar documentados diferentes errores que se han experimentado a lo largo del despliegue de los contenedores, que han llevado muchas horas de trabajo para resolver, y que pueden servir de ayuda a futuros compañeros o compañeras que prueben el uso de Docker.

1. Obtener los mensajes de la **Fig 3**, cuando ya se ha comprobado que la cámara funciona correctamente en el contenedor.

```
Starting camera service
[ WARN:0@10.922] global cap_v4l.cpp:982 open VIDEOIO(V4L2:/dev/video0): can't open camera by index
[ERROR:0@10.925] global obsensor_uvc_stream_channel.cpp:156 getStreamChannelGroup Camera index out of range
Camera ready
```

Fig 3. Error de no reconocimiento de la cámara

El error mostrado en la **Fig 3** es muy común, y aunque parece que el mensaje indique que no puede encontrarse la cámara, o que no puede ser abierta, al final a lo que se refiere es que, aunque se puede detectar que la cámara está conectada, esta no se puede utilizar porque está siendo utilizada por otro recurso del contenedor.

Este error principalmente se da cuando:

- El archivo boot.py se para haciendo "Ctrl + C", "Ctrl + Z" o comandos parecidos

- Se experimenta un error en la ejecución del archivo boot.py que para los servicios repentinamente, sin liberar los recursos.

La forma más sencilla de solucionar este error es reiniciar los servicios que están ubicados dentro del contenedor, para que estos se restauren y la cámara se libere. Para realizar esto, simplemente hace falta ubicarse en el directorio de la Raspberry Pi donde se encuentra el archivo docker-compose yml, y ejecutar:

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

Estos dos comandos, que se han presentado en el apartado anterior, paran e inician los contenedores, permitiendo volver a poder utilizar la cámara que estaba bloqueada en otro uso.

En el caso de que este error no se solucione al reiniciar los contenedores, y en ningún momento se haya conseguido hacer funcionar la cámara en el contenedor, se puede deber casi con total seguridad a que esta debe estar mal conectada. La cámara de la RPi es muy sensible, y debe estar correctamente conectada para que esta funcione.

2. Mismo error que el caso anterior, pero desde un primer momento, sin llegar a poder comprobar que la cámara está bien conectada

Cuando estos mensajes son recibidos al intentar interactuar con la cámara, por ejemplo, al tomar una foto o simplemente al realizar una prueba, desde el primer momento, implica que la cámara está mal conectada a la Raspberry Pi, ya que no puede ser debido a que la cámara está bloqueada en un proceso anterior.

En este caso, lo que hay que comprobar es que las conexiones entre la cámara y la RPi estén bien hechas, ya que, aunque es una conexión aparentemente sencilla, si no está hecha correctamente impiden que la cámara pueda ser detectada y utilizada como se espera.

La configuración correcta es siguiendo las imágenes de la **Fig 4**:

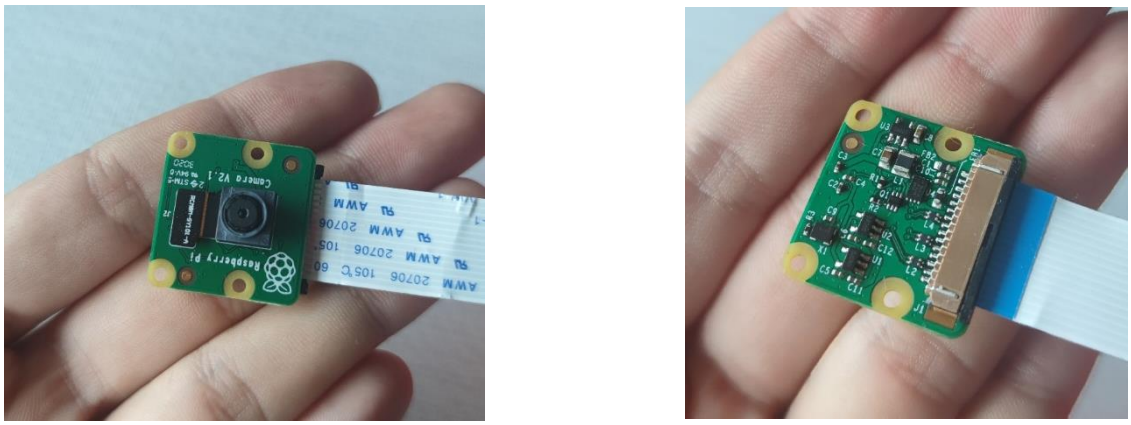

Fig 4. Correcta conexión del cable a la cámara de la RPi

Al conectar el cable a la cámara, hay que hacer especial hincapié en varios puntos:

1. Las letras del cable siempre apuntarán hacia arriba de la RPi, conectándose a la cámara
2. La parte del cable que debe conectarse por la misma cara que la cámara es la que tiene los pines visibles, no la conexión que tiene un parche azul, esta debe quedar por la parte posterior de la cámara, tal y como se muestra en la **Fig 4**

En la conexión del cable que se realiza con la RPi, el único punto importante a tener en consideración es que, como ya se ha comentado, las letras del cable queden visibles, no en la parte posterior de este mirando hacia el interior de la placa base, tal y como se muestra en la **Fig 5**.

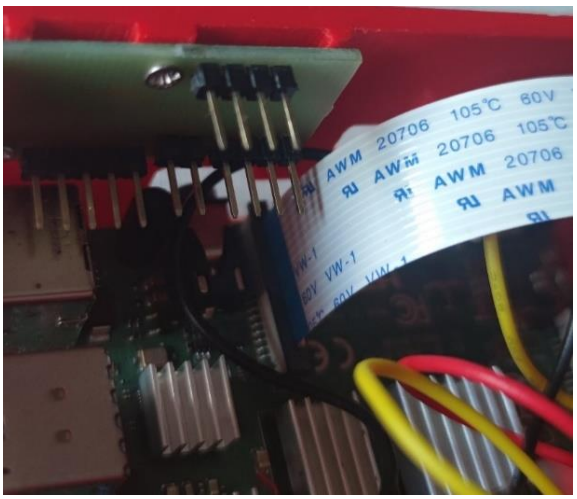

Fig 5. Conexión del cable de la cámara a la RPi

Una vez que estas conexiones se han hecho correctamente, comprobando que el cable está bien encajado en ambos casos, la recomendación es reiniciar el dispositivo, para que reinicie sus puertos de entrada y detecte la conexión de la cámara desde el inicio.

Cuando ya se ha encendido, una manera sencilla de comprobar que la cámara es detectada, antes de realizar ninguna comprobación con un código de ejemplo, es ejecutar el siguiente comando:

- `ls /dev/video*`

Al ejecutarlo, se le pide al sistema que muestre todos los dispositivos con nombre “video*”, siendo * cualquier número asociado a esta identificación. Si al ejecutar este comando se muestra que existe una identificación con el nombre de “video0”, implica que la cámara ha sido detectada, y a partir de este momento se puede trabajar con ella.

3. Problemas con plataformas amd64 o arm64

Este error tiene que ver con las características establecidas a la hora de generar una imagen, utilizando un archivo Dockerfile [22], concretamente con el parámetro *--platform.*

Hay que tener en cuenta que las imágenes utilizadas de Docker deben ser generadas para la arquitectura de Linux utilizada en la RPi. Al utilizar Ubuntu 20.04 LTS, la arquitectura debe ser arm64, característica que debe ser especificada en la construcción de las imágenes con código propio, como por ejemplo *services* o *monitor.*

Por otro lado, al utilizar imágenes ya creadas y publicadas en Docker Hub, como por ejemplo la utilizada para Mosquitto, se debe comprobar que la versión de la imagen utilizada sea compatible con Linux arm64, como por ejemplo la versión *2.0.5*, que dentro del archivo "docker-compose.yml" se identifica como eclipse-mosquitto:2.0.5.

A la hora de crear imágenes con código fuente propio, y asumiendo que el Dockerfile ya está escrito y añadido a la carpeta dónde se encuentra el código que quiere ser la base de la imagen, es importante saber que el comando a utilizar debe contener:

- `--platform /linux/arm64/v8`

Añadiendo este parámetro al ejecutar el comando docker build, se especifica que la arquitectura debe ser compatible con arm64, y permitirá que la imagen generada pueda ser utilizada en la RPi sin mayores restricciones.

4. Problemas con la librería de Neopixel y Board, relacionades con el uso de LEDs

Ambas librerías son muy importantes para poder utilizar los LEDs, tanto dentro del contenedor, como desde la propia RPi, como se ejecutaban antes de realizar las modificaciones de este trabajo. La principal diferencia entre ambos casos es que anteriormente se instalaban directamente, a través del archivo requirements.txt, y desde la introducción de Docker, estas librerías se instalan desde el archivo Dockerfile.

El error viene al utilizar contenedores, ya que la instalación de la librería de Adafruit, dentro del archivo Dockerfile, ya contiene la instalación de ambas, y en el caso de que posteriormente se instale tanto la librería de Neopixel, como Board, como parte del archivo "requirements.txt", resulta en un conflicto entre ambas instalaciones que no permite utilizar los LEDs.

Para solucionar este conflicto de una manera sencilla, lo único que hay que hacer es únicamente instalar la librería de Adafruit dentro del archivo Dockerfile, eliminando tanto Neopixel como Board de "requirements.txt", así únicamente se tendrá una instalación de estas librerías y podrán ser ejecutadas de manera correcta.