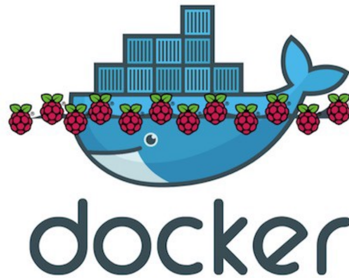
Buildare e usare container Docker
per Raspberry Pi (ARM)



I problemi legati all'utilizzo di Docker su Raspberry (ed in generale su Linux Embedded basati su tecnologia ARM), sono due:

1. Installare Docker su questi dispositivi;
2. Cross-buildare i container Docker su computer convenzionali, per semplificare e velocizzare l'intero processo di sviluppo.

Il primo problema, fortunatamente, è già risolto dagli stessi sviluppatori docker, che al momento hanno rilasciato uno script che permette, con un semplicissimo comando, di installare docker praticamente su qualsiasi ambiente linux supportato (e quindi anche ambiente ARM).

Il secondo problema è un po' più incasinato, ma per fortuna i ragazzi di hypriot.com hanno rilasciato un bellissimo post che spiega una procedura semplice da atturare per fare questa operazione.

Questo post quindi ha il triplice scopo di:

1. Insegnare come installare Docker su Raspberry Pi in modo semplice e veloce.
2. Proporre una possibile build chain per cross-compilare immagini Docker su un computer Intel.
3. Presentare una soluzione per il deploy delle applicazioni direttamente su Raspberry Pi.

Cross Build di Docker su computer Intel

La **Cross Compilazione** o **Cross Build** è un concetto un po' complesso, che provo a spiegare qui: l'idea è quella di creare un *eseguibile* (o in questo caso un *container*) pensato per funzionare su un'architettura hardware diversa rispetto a quella in cui è viene eseguita il build. Nel nostro caso, in particolare, quello che ci interessa è creare container docker per Raspberry Pi (quindi architettura ARM) su un computer classico, solitamente dotato di architettura Intel.

Il crossbuild è utile per due motivi:

1. Quando la nostra macchina target è poco performante, e quindi un build diretto sarebbe enormemente lento. Il Raspberry Pi è una macchina che, specialmente nelle versioni 3 e 3+, ha ottenuto potenze di calcolo non indifferenti. Ciò non toglie che un computer consumer è comunque più veloce in questo tipo di processi:

2. Quando (e questo è il motivo principale per cui preferisco preferisco fare il cross build) lavorare direttamente sulla macchina target è scomodo perchè non abbiamo a disposizione un buon ambiente di sviluppo.

Prima di procedere, ricordiamoci che docker ha un metodo di build piramidale. Quando si vuole creare un nuovo container, solitamente quello che si fa è partire da un container già esistente che viene customizzato come descritto nel *Dockerfile*. Se vogliamo creare un container con target ARM, dobbiamo quindi partire da un container già creato per ARM.

Container per Architetture arm32v7

Fortunatamente, anche in questo caso ci vengono in aiuto gli ideatori e mantainer di Docker, che mettono a disposizione, parallelamente alle varie immaigni ufficiali docker, anche immagini per differenti architetture. Per accedere a queste immagini, basta anteporre il tag dell'architettura che ci interessa usare (ad esempio `arm32v7` nel caso del Raspberry PI) al nome dell'immagine da scaricare.

Se, per esempio, ci interessa scaricare `ubuntu:16.04` per Raspberry Pi, useremo il nome `arm32v7/ubuntu:16.04`.

Cross Build di container per architettura arm32v7

> Nota: Ho testato questa procedura sia su macOS che su Linux, non ho avuto modo di usarla sotto Windows, che comunque sconsiglio perchè notorialmente ha un bel po' di problemi nello sviluppo.

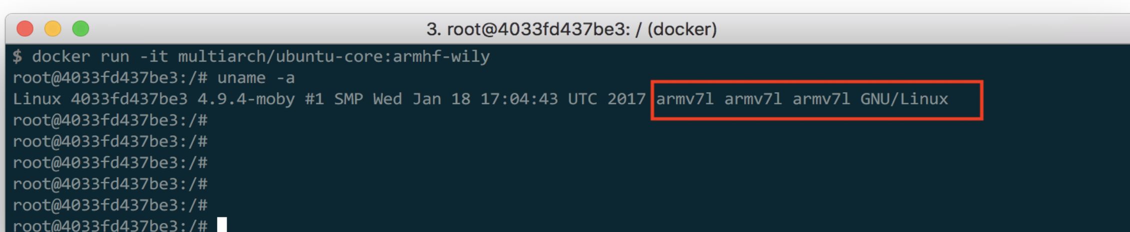
Una volta ottenuta la nostra immagine di partenza, il prossimo passo è quella di poter lanciare il container sulla nostra macchina in emulazione. Per farlo, usiamo il progetto **qemu**, e per fortuna anche in questo caso docker ci fornisce dei tools per configurarlo al meglio.

Infatti, tutto quello che serve per far gestire il sistema di cross compilazione è lanciare questo script una volta sul nostro computer:

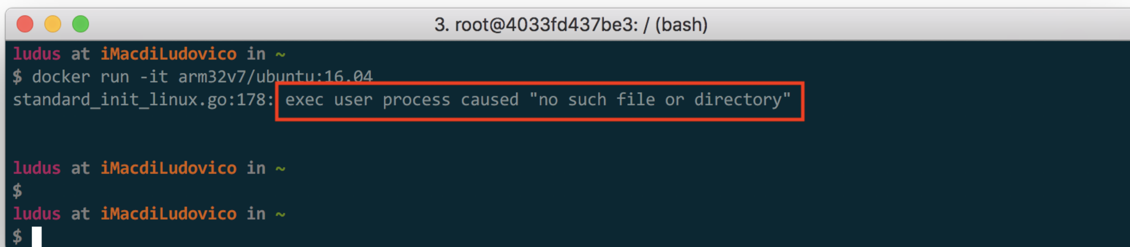
```
docker run --rm --privileged multiarch/qemu-user-static:register
```

```
Per disabilitare:
docker run --rm --privileged multiarch/qemu-user-static:register --reset
```

Fatta questa operazione, potremo utilizzare tutti i container che hanno registrato il file `qemu-*-static` per la loro architettura. Nella community docker sono già presenti un po' di container che hanno già fatto questa operazione (si veda il progetto multiarch). Ad esempio, non dovremmo avere problemi a lanciare il container `multiarch/ubuntu-core:armhf-wily`, come mostrato in figura



Però non funzionerà il container (o la classe di container) che ci interessa (`arm32v7/ubuntu:16.04`), che restituirà l'errore `standard_init_linux.go:178: exec user process caused "no such file or directory"` che essenzialmente indica che `qemu-arm-static` non è ancora stato registrato al suo interno.



Questo non è un problema quando vi interessa solo usare un container linux base: in questo caso potete spulciare la libreria **multiarch** e trovere tantissimi container già pronti all'uso.

Diventa un problema, invece, quando ci interessa utilizzare container basati su `arm32v7/ubuntu:16.04` (o simili), come ad esempio tutti i container ufficiali di ROS.

Registrare qemu nel vostro container

La registrazione di qemu nel container è un processo molto facile che va sempre eseguito all'inizio della fase di build di un container non ancora registrato. Una volta fatto, tutti i container costruiti su questo saranno automaticamente registrati, e non sarà necessario ripetere l'operazione.

Per eseguire la procedura, quindi, dobbiamo creare un nuovo ambiente docker in cui eseguire il build della nostra nuova immagini. Creiamo una cartella `docker-img-rpi` e creiamo, al suo interno, il file `Dockerfile` in cui andremo a lavorare. Dobbiamo quindi utilizzare un file di qemu chiamato `qemu-arm-static`. Questo file si trova nel vostro computer (è uno dei file registrati dalla procedure fatta sopra), ma per semplicità potete scaricarlo cliccando qui

Scaricatelo e mettetelo nella cartella di lavoro. A questo punto, è anche necessario dargli i permessi di esecuzione con il comando:

```
$ chmod +x qemu-arm-static
```

Andiamo quindi a creare il `Dockerfile`:

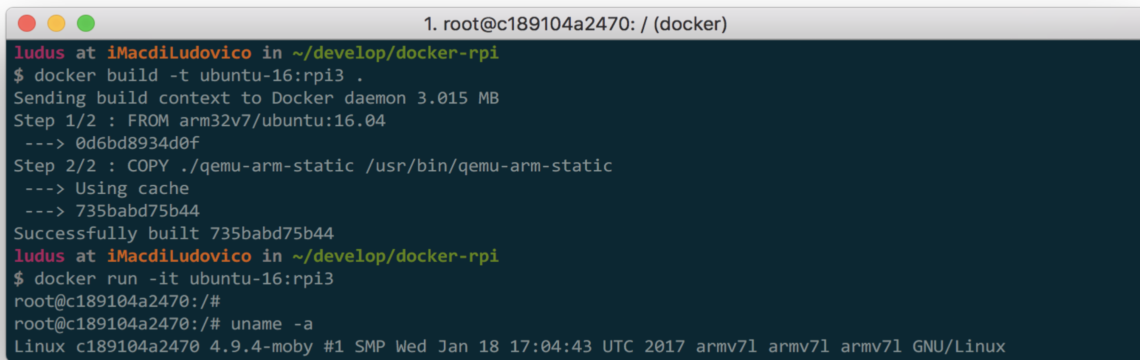
```
FROM  arm32v7/ubuntu:16.04
COPY ./qemu-arm-static /usr/bin/qemu-arm-static
```

Dove la prima riga `FROM arm32v7/ubuntu:16.04` dice di creare la nuova immagine a partire da `arm32v7/ubuntu:16.04`, mentre la seconda è quella che si occupa (effettivamente) di eseguire la registrazione.

A questo punto, buildiamo il tutto e testiamo se tutto funziona:

```
$ docker build -t ubuntu-16:rpi3 .
$ docker run -it ubuntu-16:rpi3
```

E come vedete, adesso l'immagine adesso viene eseguita correttamente.



Build e Deploy con Docker Compose

Una volta abilitata la fase di cross compilazione, non ci resta altro che trovare un modo per passare l'immagine dal nostro computer verso il raspberry in cui vogliamo che questa venga eseguita.

Ovviamente, ci sono diversi modi per farlo: il modo più semplice che ho trovato è quello di usare il progetto **dockehub** per condividere le immagini (utilissimo a meno che non volete creare delle immagini private) e **docker-compose** per il build e il deploy.

Come avete visto se avete smanettato un po' con docker, la gestione da linea di comando di docker è un po' pesantuccia, ed ci troviamo spesso a dover scrivere comandi molto lunghi su shell. L'errore e i typos sono quindi un grosso problema che spesso ci costringono a riscrivere molte volte lo stesso comando. Docker-compose è un progetto che, sebbene porti tantissimi vantaggi nella gestione dei container, utilizzo spesso perchè semplifica enormemente la gestione da linea di comando di docker.

Vi ricordo che per installare docker-compose basta eseguire il comando:

```
$ sudo pip install docker-compose
```

Il concetto di questo tool è molto semplice: viene creato un nuovo file (chiamato `docker-compose.yml`) che contiene alcune informazioni del progetto docker che vogliamo creare (come il nome da dare all'immagine del container, la posizione dei Dockerfile, etc.), grazie a questo file, docker-compose semplifica le api dal shell di docker enormemente.