

Progetto Sistemi orientati ad Internet

Jacopo Orlandini¹

¹ Università degli studi di Parma

² `jacopo.orlandini@studenti.unipr.it`

Abstract. Implementazione del framework Docker su sistema CloudAWM. Vengono implementate le funzionalità di scaling automatico, sostituendo l'architettura OpenStack presente. Il sistema presentato riesce a soddisfare attraverso l'uso di Francis Controller i carichi dinamici del sistema.

Keywords: Docker · CloudAWM

1 Docker

Docker è un progetto open-source che automatizza il deployment (consegna o rilascio al cliente, con relativa installazione e messa in funzione o esercizio, di una applicazione o di un sistema software tipicamente all'interno di un sistema informatico aziendale) di applicazioni all'interno di contenitori software, fornendo un'astrazione aggiuntiva grazie alla virtualizzazione a livello di sistema operativo di Linux.

Docker implementa API di alto livello per gestire container che eseguono processi in ambienti isolati. Poichè utilizza delle funzionalità del kernel Linux, un container di Docker, a differenza di una macchina virtuale, non include un sistema operativo separato. Al contrario, utilizza le funzionalità del kernel e sfrutta l'isolamento delle risorse (CPU, memoria, I/O a blocchi, rete) ed i namespace separati per isolare ciò che l'applicazione può vedere del sistema operativo. Docker accede alle funzionalità di virtualizzazione del kernel Linux o direttamente utilizzando la libreria libcontainer, che è disponibile da Docker 0.9.

Docker è diventato molto popolare grazie alle sue caratteristiche uniche:

- Flessibilità : applicazioni più complesse possono essere containerizzate;
- Leggerezza : utilizza kernel Linux presente;
- Intercambiabilità : aggiornamento del container on-the-fly;
- Portabilità : compilazione locale e deploy su Cloud;
- Scalabilità : possibilità di incrementare il numero di repliche del container.

1.1 Container

Un container è una unità standard software che impacchetta il codice e tutte le sue dipendenze in modo tale che l'applicazione esegua rapidamente e sia indipendente dall'ambiente [figura 1].

1.2 Image

Un image per docker container é leggera e indipendente. É un software eseguibile che include tutto ciò che serve per far girare l'applicazione: codice, runtime, strumenti di sistema, librerie di sistema e impostazioni.

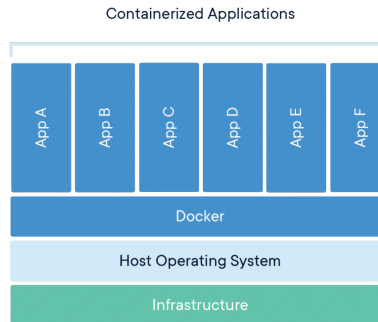


Fig. 1. Container sul sistema operativo

1.3 Costruzione di un container

Un container é lanciato facendo girare un "image". Un image identifica un pacchetto eseguibile che include tutto il necessario in termini di requisiti per farlo funzionare (librerie, variabili d'ambiente,...). Un container é un'istanza runtime di un immagine (ció che l'immagine diventa in memoria nel momento dell'esecuzione). Per verificare che docker sia installato, usare `docker run hello-world` e seguire le istruzioni. La versione utilizzata di Docker é la seguente:

```
(base) jacopo@jorlandi-pc:~$ docker --version
Docker version 18.09.2, build 6247962
(base) jacopo@jorlandi-pc:~$
```

Fig. 2.

Prima di vedere come costruire un container da zero, mostro i comandi principali che useró durante l'esecuzione del progetto per verificare lo stato del sistema.

- `$ docker container ls -a` : visualizzare tutti container presenti in locale;
- `$ docker image ls -a` : visualizzare tutte le image in locale;

- \$ docker stats : visualizzare stato dei container in real-time;
- \$ python killContainers.py : reset della simulazione, elimina tutti i container presenti nel sistema.

Per prima cosa dobbiamo decidere il fine della nostra applicazione. Dobbiamo provvedere a scrivere un file eseguibile da caricare su un container per svolgere un compito. Nel nostro caso, l'immagine sarà basata su un semplice server Python, app.py. Per costruire un'immagine si necessita di un nome ("serverapp"), di un file eseguibile ("app.py") e di un file di configurazione Docker ("Dockerfile"). Il file

Dockerfile

```

1# Use an official Python runtime as a parent image
2FROM python:2.7-slim
3
4# Set the working directory to /app
5WORKDIR /app
6
7# Copy the current directory contents into the container at /
  app
8COPY . /app
9
10# Install any needed packages specified in requirements.txt
11RUN pip install --trusted-host pypi.python.org -r
    requirements.txt
12
13# Make port 80 available to the world outside this container
14EXPOSE 80
15
16# Define environment variable
17ENV NAME World
18
19# Run app.py when the container launches
20CMD ["python", "app.py"]
21

```

Fig. 3. Per creare un'immagine con la configurazione DockerFile

Dockerfile rappresenta la configurazione di sistema per costruire l'immagine. Il nome del file "Dockerfile" non deve essere modificato in quanto nel momento della build, Docker cerca nella cartella di build (".") un file con nome "Dockerfile". Nel mio caso Dockerfile usa python con tag 2.7-slim. Al link [Docker-python] è possibile trovare un catalogo di versioni Python da poter utilizzare, nel caso servisse compatibilità con versioni più recenti. Osserviamo che alla riga 14 del Dockerfile viene esposta la porta 80 sul container, che verrà redirezionata sulle porte 4000 e successive sulla macchina locale. In "requirements.txt" vengono dichiarate le librerie necessarie al file python per poter eseguire.

Una volta creati i file di configurazione ["Dockerfile", "requirements.txt"] è possibile usare CLI per costruire un'immagine, oppure attraverso le API Docker.

Costruita l'immagine è possibile avviare un container su essa.

Supported tags and respective Dockerfile links

Simple Tags

- 3.8.0a2-stretch, 3.8-rc-stretch, rc-stretch (3.8-rc/stretch/Dockerfile)
- 3.8.0a2-slim-stretch, 3.8-rc-slim-stretch, rc-slim-stretch, 3.8.0a2-slim, 3.8-rc-slim, rc-slim (3.8-rc/stretch/slim/Dockerfile)
- 3.8.0a2-alpine3.9, 3.8-rc-alpine3.9, rc-alpine3.9, 3.8.0a2-alpine, 3.8-rc-alpine, rc-alpine (3.8-rc/alpine3.9/Dockerfile)

Fig. 4. Versioni di python per Dockerfile

```
(base) jacopo@jorlandi-pc:~$ docker run serverapp
* Serving Flask app "app" (lazy loading)
* Environment: production
  WARNING: Do not use the development server in a production environment.
  Use a production WSGI server instead.
* Debug mode: on
* Running on http://0.0.0.0:80/ (Press CTRL+C to quit)
* Restarting with stat
* Debugger is active!
* Debugger PIN: 333-477-761
```

Fig. 5. Per verificare che il server container sia Up. Da notare come il container esponga la porta 80 mentre il sistema locale fa rebind sulla porta 4000. Per verificare(da browser) la presenza del server andare su localhost:4000.

Ogni qualvolta si faccia un rebuild dell'immagine, la precedente versione viene sovrascritta. Per costruire un image avviare il comando:
"docker build --tag=serverapp ." [figura 6].

2 Implementazione Sistema

In questa sezione fornisco i dettagli implementativi delle classi utilizzate nel progetto.

2.1 RandomWorkLoadGenerator

La classe RandomWorkLoadGenerator ha il compito di generare un nuovo carico di richieste per il sistema. La classe genera W richieste dopo un ritardo T in secondi. Ha il compito di tenere traccia di quante richieste non soddisfatte sono presenti al momento del nuovo carico di richieste. Deve sincronizzarsi con la classe AWM per tenere traccia degli istanti di arrivo delle richieste.

2.2 DockerContainerWrapper

La classe DockerContainerWrapper svolge il ruolo di contenitore di informazioni riguardo al container. Usato principalmente per tenere traccia dello stato della

```
(base) jacob@jorlandi-pc:~/Scrivania/Sistent orientati ad internet/Progetto SOI/cloudawms$ docker build --tag=serverapp .
Sending build context to Docker daemon  24.59MB
Step 1/7 : FROM python:2.7-slim
----> 8559620b5b0d
Step 2/7 : WORKDIR /app
----> Using cache
----> b8a413385c2e
Step 3/7 : COPY ./app
----> 6b82936b6210
Step 4/7 : RUN pip install --trusted-host pypi.python.org -r requirements.txt
----> Running in e1663d31f084
DEPRECATION: Python 2.7 will reach the end of its life on January 1st, 2020. Please upgrade your Python as Python 2.7 won't
Collecting Flask (from -r requirements.txt (line 1))
  Downloading https://files.pythonhosted.org/packages/7f/e7/08578774ed4536d3242b14dadb4696386634607af824ea997202cd0edb4b/Fl
Collecting Redis (from -r requirements.txt (line 2))
  Downloading https://files.pythonhosted.org/packages/ac/a7/cff10cc5f1180834a3ed564d148fb4329c989cbb1f2e196fc9a10fa07072/re
Collecting itsdangerous==0.24 (from Flask->-r requirements.txt (line 1))
  Downloading https://files.pythonhosted.org/packages/76/ae/44b03b253d6fade317f32c24d100b3b35c2239807046a4c953c7b89fa49e/lt
Collecting Jinja2==2.10 (from Flask->-r requirements.txt (line 1))
  Downloading https://files.pythonhosted.org/packages/7f/ff/ae64bacdfc95f27a016a7bed8e8686763ba4d277a78ca76f32659220a731/3l
Collecting Werkzeug==0.14 (from Flask->-r requirements.txt (line 1))
  Downloading https://files.pythonhosted.org/packages/24/4d/2fc4e872fbaf44cc3fd5a9cd42fda7e57c031f08e28c9f35689e8b43198/We
Collecting click==5.1 (from Flask->-r requirements.txt (line 1))
  Downloading https://files.pythonhosted.org/packages/fa/37/45185cb5abbc30d7257104c434fe0b07e5a195a6847506c074527aa599ec/CL
Collecting MarkupSafe==0.23 (from Jinja2==2.10->Flask->-r requirements.txt (line 1))
  Downloading https://files.pythonhosted.org/packages/fb/40/f3adb7cf24a8012813c5edb20329eb22d5d8e2a0ecf73d21d6b85865da11/Ma
Installing collected packages: itsdangerous, MarkupSafe, Jinja2, Werkzeug, click, Flask, Redis
Successfully installed Flask-1.0.2 Jinja2-2.10 MarkupSafe-1.1.1 Redis-3.2.1 Werkzeug-0.15.1 click-7.0 itsdangerous-1.1.0
Removing intermediate container e1663d31f084
----> e88dc8fb5ca
Step 5/7 : EXPOSE 80
----> Running in 28ee40cabdb
Removing intermediate container 28ee40cabdb
----> 0a9acfd9f3f3
Step 6/7 : ENV NAME World
----> Running in 472dc69f0fd4
Removing intermediate container 472dc69f0fd4
----> 72d226bfeec2
Step 7/7 : CMD ["python", "app.py"]
----> Running in 3104119d1159
Removing intermediate container 3104119d1159
----> e0a4283099c0
Successfully built e0a4283099c0
```

Fig. 6. Docker build

CPU e al fine di inviare le richieste pervenute. Questo wrapper mantiene traccia dell'id del container, porta del container e nome del container per facilitare la gestione tra progetto Java e API docker in python.

2.3 FrancisController

La classe FrancisController viene modificata solamente nel metodo nextVMDelta(). La prima modifica riguarda l'acquisizione del tempo medio della vita dei container. Il processo di acquisizione viene ottenuto con lo script getAvgTime.py che restituisce la media della vita in secondi dei container in una finestra temporale. Altra modifica l'acquisizione del numero di container attivi al momento dell'avvio del metodo. Per questo motivo eseguo lo script getNumContainer.py, per ottenere il numero di container attivi. Viene modificato il parametro etak[0][0], assegnandoli il numero totale di richieste nel sistema presenti nel sistema. Il responso di questo metodo un numero intero che rappresenta il numero di container da aggiungere o rimuovere.

2.4 AutonomicWorkloadManager

La classe AutonomicWorkloadManager viene usata per avviare la simulazione, come in precedenza in CloudAWM. Avvia i thread principali, setta le variabili di supporto per il sistema.

2.5 app.py

Lo script app.py viene utilizzato per la costruzione dell'immagine docker. Rappresenta un semplice server basato su Flask. Il server presenta due pagine :

- root : localhost:4000/, dove si restituisce il nome del container.
- start del task : localhost:4000/start_task, dove si avvia un generatore di numeri di fibonacci fino al numero 35.

Il server viene avviato in locale localhost, "0.0.0.0" sulla porta 80. Ricordo che dal pc per accedere alla pagina deve andare sulla porta 4000 o successive in base al container selezionato.

2.6 helloContainer.py

Lo script helloContainer.py rappresenta lo script di avvio del container con image "serverapp". Facilmente implementabile la possibilità avviare container con image differenti, occorre passare come argomento il nome dell'immagine e costruirla precedentemente. Lo script si collega all'API Docker per permettere l'avvio di un container con parametri, attraverso un client di default (Docker Engine). Da notare che posso redirezionare la porta del container attraverso un piccolo dizionario, chiamato dict_port, da 80:4000. Descrizione dei parametri:

- primo parametro : nome dell'immagine;

```

1 import docker as dk
2 import sys
3
4 # client = Docker Engine similar to CLI docker
5 client = dk.from_env()
6
7 dict_port = {'80/tcp': int(sys.argv[2])}
8
9 logContainer = client.containers.run(
10 "serverapp",
11 name=sys.argv[1],
12 detach=True,
13 auto_remove=True,
14 ports=dict_port )
15

```

Fig. 7. helloContainer.py

- name : nome del container, devono essere univoci (se no dá errore);
- detach : gira in background;
- auto_remove : quando il container termina si elimina;
- port : redirectione della porta del container.

2.7 getStatus.py

```

1 import docker
2 # import datetime
3 import os
4
5 cmd = "docker ps -q | xargs docker stats --no-stream"
6 returned_value = os.system(cmd) # returns the exit code in unix
7

```

Fig. 8. getStatus.py

Lo script getStatus.py ritorna lo stato dei container versione terminale. Le API docker non permettono di ottenere facilmente la CPU utilizzata (non presente un campo). Sono presenti però l'utilizzo della CPU in millisecondi. Nel file proposto una soluzione parziale per calcolare l'utilizzo della CPU in percentuale con API Docker. La soluzione più semplice stata quella di prendere le informazioni da CLI .

2.8 getAvgTime.py

Lo script getAvgTime controlla da quanto sono attive tutti i container e restituisce una semplice media in secondi. Adotta la stessa logica del getAvgWindowedCloudTime nella classe Qos. Lo script restituisce la media dei tempi in una finestra temporale passata di 4 minuti.

```

1  import docker as dk
2  import datetime
3  from datetime import timedelta
4  client = dk.from_env()
5
6  d = {"status": "running"}
7  containers = client.containers.list(filters=d)
8
9  avgTime = 0
10 # Finestra di tempo 3 minuti
11 WindowInMinutes = 4 * 60
12 sum = 0
13 counter = 0
14
15 new = datetime.datetime.now()
16 d = timedelta(seconds=(WindowInMinutes+60*60)) # fuso orario +1, devo togliere un ora
17 new = new-d # finestra temporale del passato
18
19 for c in containers:
20     creation_date = c.attrs["Created"]
21     new1 = datetime.datetime.now()
22     d1 = timedelta(seconds=(60*60)) # fuso orario +1, devo togliere un ora
23     new1 = new1 -d1
24     old = datetime.datetime.strptime(creation_date[:4], '%Y-%m-%dT%H:%M:%S.%f') # da quanto e' attivo un container
25     Te = new1 - old
26     if old > new:
27         sum += Te.seconds
28         counter +=1
29 if counter == 0:
30     print(0)
31 else:
32     print (sum/counter)

```

Fig. 9. getAvgTime.py

2.9 client.py

```
ip = "localhost"
port = sys.argv[1]
contents = urlopen("http://" + ip + ":" + str(port) + "/start_task").read()
```

Fig. 10. client.py

Lo script client esegue una semplice richiesta HTTP che avvia start_task su server. In questo modo eseguo del carico di lavoro sul server.

2.10 killContainers.py

Lo script killContainers non viene utilizzato all'interno del AWM ma come procedura esterna per eliminare tutti i containers. In questo modo posso reinizializzare la simulazione.

3 Funzionamento del sistema

In questa sezione viene descritto il comportamento ad alto livello del sistema implementato.

La classe AutonomicWorkloadManager inizializza alcune variabili di supporto per la gestione dei dati e imposta la porta di partenza del server a 4000. Alla partenza del sistema vengono creati un numero minimo di container (hard-coded $kmin = 2$ o 5) con l'immagine serverapp. L'utente deve inserire la tipologia di simulazione che vuole adottare; con 0 si imposta il Francis Controller, con 1 non si imposta nessun controller (il sistema funziona interamente con kmin container). Il funzionamento del sistema é quello di soddisfare nel miglior modo le richieste. A questo fine sono istanziati due tipologie di carichi:

- carico statico : presente dall'inizio della simulazione;
- carico dinamico : carico che viene aggiunto dopo un lasso temporale definito.

Per il carico statico vengono istanziate un numero "staticWork" di richieste. Per il carico dinamico viene creato un thread che dopo un tempo "delay" carica un numero di richieste pari a "dynamicWork". Successivamente creo il thread per il controllore Francis Controller. Finita la fase di inizializzazione del sistema, si susseguono in ordine:

- Francis Response : quanti container alloco o rimuovo;
- Checker : in quale stato sono i container;
- Dispatcher : invio dei carichi ai container disponibili;
- Sampler : campionamento del sistema;

4 Risultati e Simulazioni

Qui di seguito vengono proposte le simulazioni eseguite ed i loro risultati.

4.1 Setup della simulazione

Le simulazioni sono di due tipi distinti. La prima tipologia é quello con l'utilizzo del controller Francis mentre la seconda viene fatta senza nessun controller. Quello che ci si aspetta é un notevole miglioramento delle prestazioni generali del sistema ed un migliore tempo di risposta delle richieste. La prima simulazione con francis controller viene eseguita con solo due container iniziali, il che rende leggero il sistema. La simulazione senza controller viene lanciata con 5 container ma senza la possibilità né di allocare né di rimuovere container.

Table 1. Parametri della simulazione per grafico

Parameters	Values
Campionamento	settabile sT
Porte	da 4000 in su
Cpu libera	< 3%
Carico statico	settabile
Carico dinamico	settabile
Tempo carico dinamico	settabile
Controller	Francis o Nessuno

5 Grafici

In figura 9 si presenta una simulazione di prova per evidenziare come il sistema riesca ad adeguare il numero di container con l'arrivo delle nuove richieste in maniera dinamica seguendo il francis controller. Di seguito invece vengono presentate le configurazioni reali:

La simulazione con controller é stata inizializzata con questi parametri:

- staticWork = 10
- dynamicWork = 20 * 5 volte

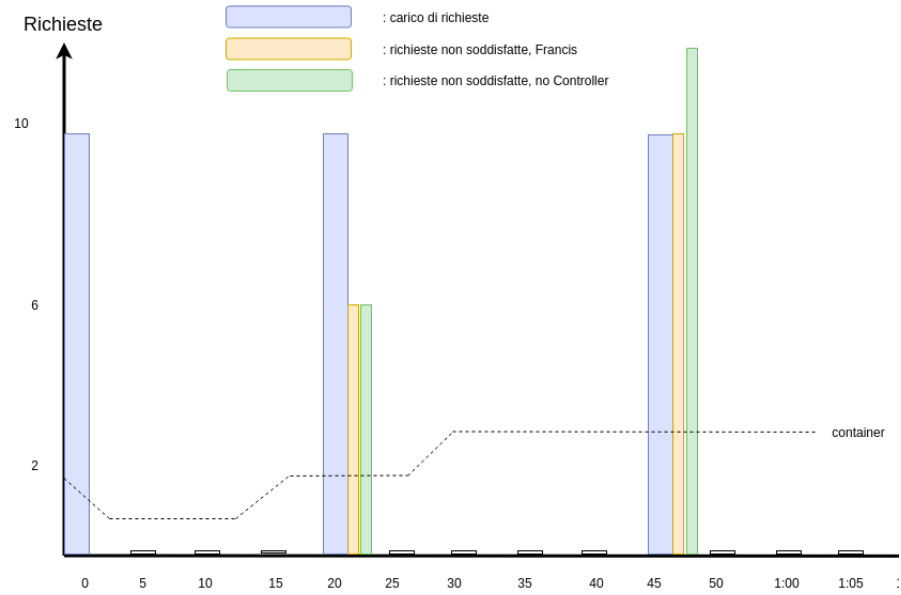


Fig. 11. Caption

- delay = 50 sec
- kMin = 2

La simulazione senza controller stata inizializzata con questi parametri:

- staticWork = 10
- dynamicWork = 20 * 5 volte
- delay = 50 sec
- kMin = 5

La stima del tempo totale é uguale al calcolo del tempo di lavoro atteso moltiplicato per le richieste totali, prese a carico in modo sequenziale.

$$(20 * 5 + 10) * (13sec) = 148,50sec$$

Table 2. Comparazione dei risultati delle simulazioni

Controller	Risultato
0	101,70 sec
1	134,04 sec
atteso	148,50 sec

6 Conclusioni

In questo progetto si presenta la possibilità di integrare Docker in un sistema di scaling automatico. Docker tuttavia presenta API ufficiali solo per Python e per Go, rendendo difficile l'implementazione con il progetto in Java. Tuttavia eseguendo gli script in maniera asincrona é stato possibile integrare il sistema CloudAWM e il framework Docker.

Bibliography

[Docker_python] Docker_python. "<https://hub.docker.com//python/>". [*Online; accessed 23–Marzo – 2019*].