Progetto Sistemi orientati ad Internet

Jacopo Orlandini[1]

[1] Universita degli studi di Parma
[2] jacopo.orlandini@studenti.unipr.it

Abstract. Implementazione del framework Docker su sistema CloudAWM. Vengono implementate le funzionalitá di base di docker per avere un sistema dinamico in gradi di eseguire lo scaling automatico di container per eseguire un carico variabile di richieste.

Keywords: Docker · CloudAWM

1 Docker

Docker 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. Poich utilizza delle funzionalit del kernel Linux (principalmente cgroups e namespaces), un container di Docker, a differenza di una macchina virtuale, non include un sistema operativo separato. Al contrario, utilizza le funzionalit del kernel e sfrutta l'isolamento delle risorse (CPU, memoria, I/O a blocchi, rete) ed i namespace separati per isolare ci che l'applicazione pu vedere del sistema operativo. Docker accede alle funzionalit di virtualizzazione del kernel Linux o direttamente utilizzando la libreria libcontainer, che disponibile da Docker 0.9.

Un container Docker sta diventando molto popolare grazie alle sue caratteristiche uniche:

- Flessibile anche le applicazioni pi complesse possono essere conteinerizzate;
- Leggero si appoggiano sul kernel presente;
- Intercambiabile aggiornamenti on-the-fly;
- Portabile posso compilare localmente e fare deploy sul cloud;
- Scalabile posso incrementare il numero di repliche del 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, ovvero ció che l'immagine diventa in memoria nel momento dell'esecuzione.

Altri comandi docker che principalemente uso sono:

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

Fig. 1. Per verificare che docker sia installato, usare docker run hello-world (esempio di prova)

- $ docker container ls -a : visualizzare tutti container
- $ docker image ls -a : visualizzare tutte le image
- $ docker stats : visualizzare stato dei container in real-time
- $ python killContainers.py : reset della simulazione, elimina tutti i container presenti nel sistema.

Per costruire un image si necessitá di un nome da dare all'image, di un file eseguibile e di un file di configurazione Dockerfile. Il file Dockerfile, che deve

Dockerfile

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

Fig. 2. Per creare un image con la configurazione DockerFile

mantenere invariato il nome al fine del build, usa python con tag 2.7-slim. Al link [Docker_python] é possibile trovare un catalogo di versioni Python da poter utilizzare, nel caso servisse compatibilit con versioni piú recenti. Osserviamo che alla riga 14 del Dockerfile viene esposta la porta 80 sul container, che verr redirezionata sulle porte 4000 e successive. Infine l'image viene costruita con il file python chiamato app.py.

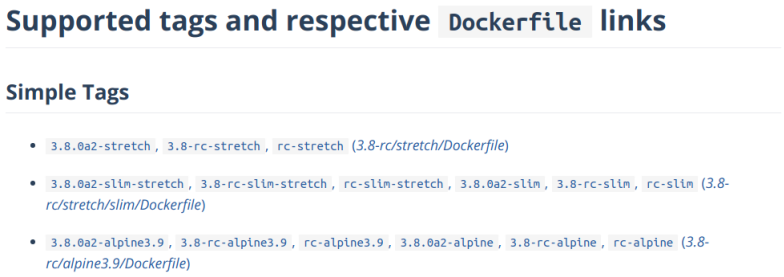

Fig. 3. Versioni di python per Dockerfile

Una volta creati i file di configurazione [”Dockerfile”, ”requirements.txt”] é possibile usare CLI per costruire un image.

Durante lo svolgimento del progetto ho implementato la costruzione dell’image con API Docker, con le stesse configurazioni. Costruita l’image é possibile avviare un container su essa. Ovviamente é possibile avviarla con CLI per una veloce fase di debug:

```
(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. 4. Per verificare che lo stato del container

Ogni qualvolta si faccia un rebuild dell’immagine con lo stesso nome, la precedente versione viene sovrascritta.

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. 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 della CPU e dello stato di utilizzo del container al fine di soddisfare le richieste del sistema. Questo wrapper mantiene traccia dell'id del container, porta del container e nome del container per facilitare la gestione del container.

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. 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 all'avvio del metodo.

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.

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 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.

```
import docker as dk
import sys

# client = Docker Engine similar to CLI docker
client = dk.from_env()

dict_port = {'80/tcp': int(sys.argv[2])}

logContainer = client.containers.run(
"serverapp",
name=sys.argv[1],
detach=True,
auto_remove=True,
ports=dict_port )

```

Fig. 5. helloContainer.py

2.6 helloContainer.py

Lo script helloContainer.py rappresenta il metodo di avvio di un container con un immagine fissata a serverapp. Facilmente implementabile avviare container con image differenti, occorre passare come argomento il nome dell'image. 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'image
- name : nome del container, devono essere univoci.
- detach : gira in background
- auto_remove : quando il container termina si elimina
- port : redirezione della porta del container.

2.7 getStatus.py

```
import docker
# import datetime
import os

cmd = "docker ps -q | xargs  docker stats --no-stream"
returned_value = os.system(cmd)  # returns the exit code in unix
```

Fig. 6. getStatus.py

Lo script getStatus.py ritorna lo stato dei container versione terminale. Le API docker non permettono di ottenere facilemente la CPU utilizzata, non presente un campo. Sono presenti i campi di utilizzo della cpu in millisecondi. Nel

file proposto una soluzione parziale per calcolare l'utilizzo della CPU in percentuale con API Docker. La soluzione piú semplice stata quella di prendere da CLI le informazioni.

2.8 getAvgTime.py

```python
import docker as dk
import datetime
from datetime import timedelta
client = dk.from_env()

d = {"status":"running"}
containers = client.containers.list(filters=d)

avgTime = 0
# Finestra di tempo 3 minuti
WindowInMinutes = 4 * 60
sum = 0
counter = 0

new = datetime.datetime.now()
d = timedelta(seconds=(WindowInMinutes+60*60))    # fuso orario +1, devo togliere un ora
new = new-d # finestra temporale del passato

for c in containers:
    creation_date = c.attrs["Created"]
    new1 = datetime.datetime.now()
    d1 = timedelta(seconds=(60*60))    # fuso orario +1, devo togliere un ora
    new1 = new1 -d1
    old = datetime.datetime.strptime(creation_date[:-4], '%Y-%m-%dT%H:%M:%S.%f')    # da quanto e' attivo un container
    Te = new1 - old
    if old > new:
        sum += Te.seconds
        counter +=1
if counter == 0:
    print(0)
else:
    print (sum/counter)
```

Fig. 7. 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 secondo una finestra temporale passata di 4 minuti .

2.9 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.

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

Fig. 8. client.py

2.10 killContainers.py

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

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 (harcoded $kmin = 1$) con l'image serverapp. E' possibile impostare kmin in modo tale da far partire il sistema sia con 0 container sia con un numero piú alto. Utente ora 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 instanziati 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 possibilitá 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 nota che il sistema riesce ad adeguare il numero di container con l'arrivo delle nuove richieste in maniera dinamica seguendo il francis controller.

La simulazione con controller stata inizializzata con questi parametri:

- staticWork = 10
- dynamicWork = 20 * 5 volte
- delay = 50 sec
- kMin = 2

La simulazione senza controller stata inizializzata con questi parametri:

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

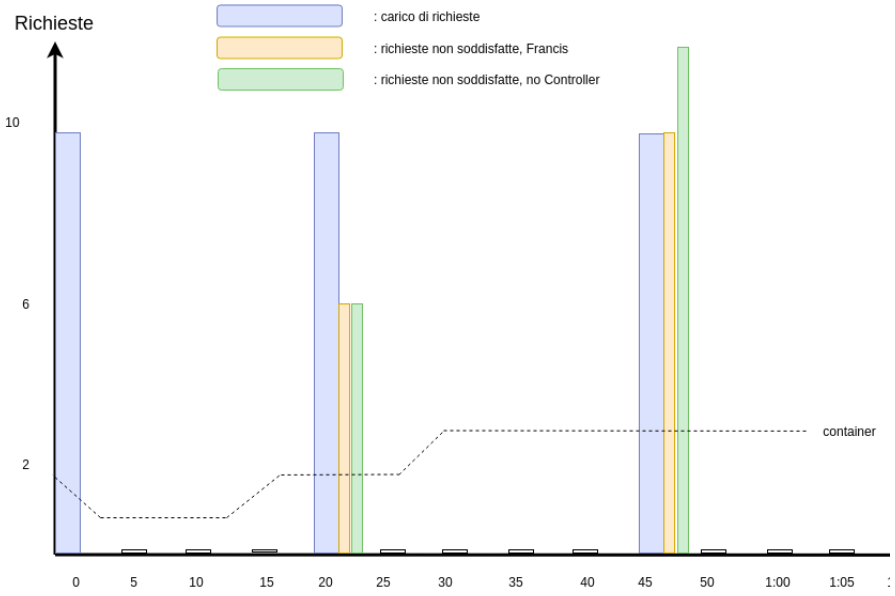

Fig. 9. Caption

– kMin = 5

La stima del tempo totale é uguale al calcolo del tempo di lavoro atteso moltiplicato per le richieste totali. $(20 * 5 + 10) * (13sec) = 148,50sec$

Table 2. Comparision of estimated vs actual link residual time

Controller	Risultato
0	101,70 sec
1	134,04 sec

6 Conclusioni

In questo progetto si presenta la possibilitá 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].