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 funzionalità di scaling automatico, sostituendo l'architettura OpenStack presente. Il sistema presentato riesce a soddisfare attraverso l'uso di Francis Controller i carici 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 conteinerizzate;
- 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 dipendenza in modo tale che l'applicazione esegua rapidamente e sia indipendente dall'ambiente [Fig. 1].

1.2 Image

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

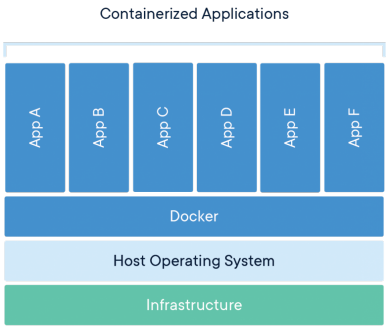

Fig. 1. Container sul sistema operativo

1.3 Costruzione di un container

Un container è 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 è un'istanza runtime di un immagine (ciò 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 è illustrata in [Fig. 2]:

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

Fig. 2. Versione Docker

Prima di vedere come costruire un container da zero, mostro i comandi principali che userò 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'image sarà basata su un semplice server Python, app.py. Per costruire un image si necessità 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 image con la configurazione DockerFile

Dockerfile rappresenta la configurazione di sistema per costruire l'image. Il nome del file "Dockerfile" non deve essere modificato in quanto nel momento della build, Docker cercherà 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 image, oppure attraverso le API Docker.

Per costruire una image, spostartsi nella cartella dove sono presenti i file descritti nella sezione precedente, poi da terminale si utilizza il comando:

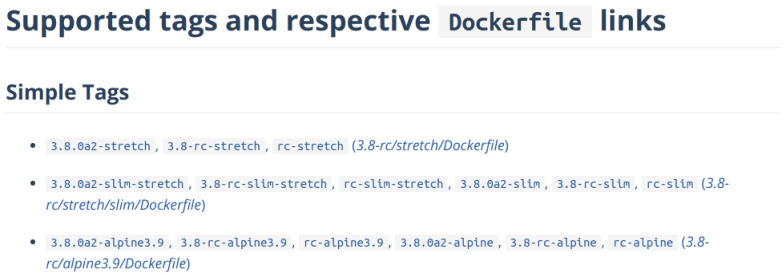

Fig. 4. Versioni di python per Dockerfile

"docker build −−tag=serverapp. Per costruirla attraverso le API di Docker si utilizza [Fig. 6]:

```
import docker as dk
client = dk.from_env()
# "./" rappresenta la cartella contenente i file
client.images.build(path="./", tag="prova")
```

Ora è possibile avviare un container sulla image con il comando in [Fig. 5].

```
(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 prensenza del server andare su localhost:4000.

Ogni qualvolta si faccia un rebuild dell'image, la precedente versione viene sovrascritta.

2 Funzionamento del sistema

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

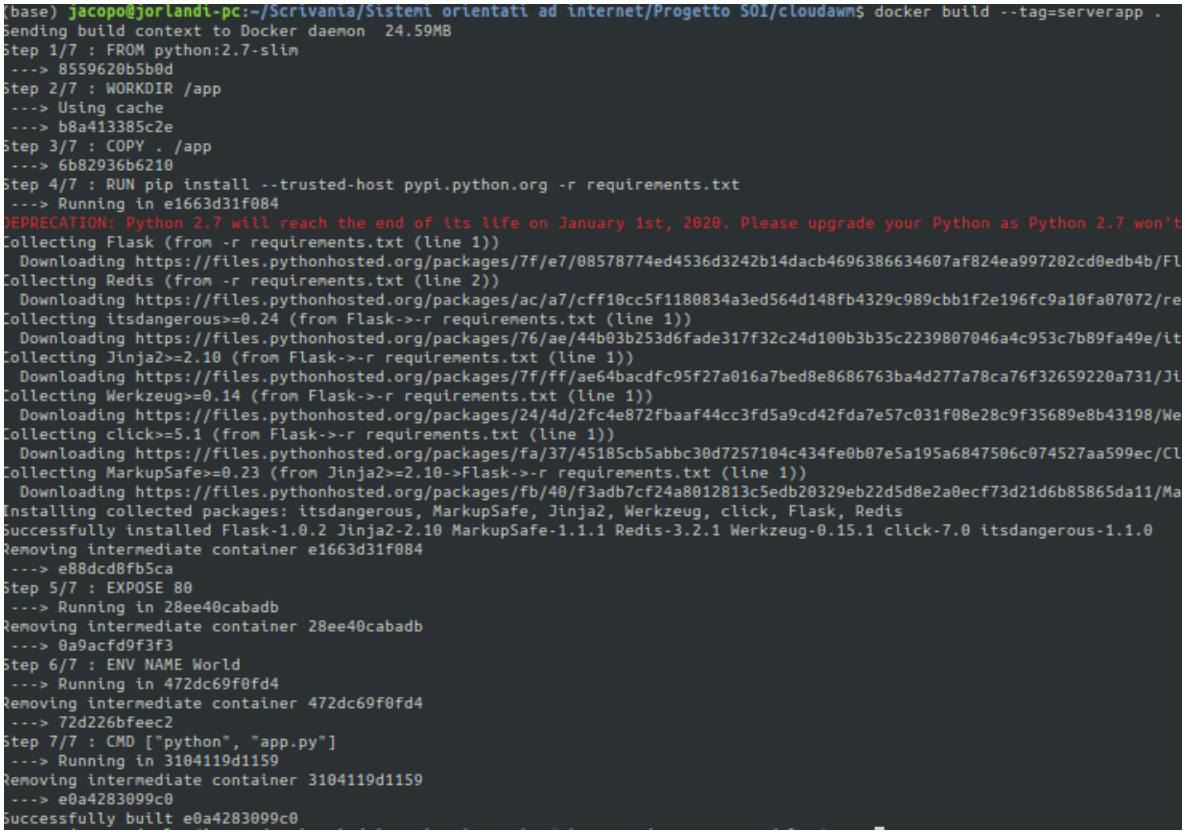

Fig. 6. Docker build

La classe AutonomicWorkloadManager inizializza alcune variabili di supporto per la gestione dei dati e imposta la porta di partenza dei server a 4000. Alla partenza del sistema vengono creati un numero minimo di container (harcoded $kmin = 2$ o 5) con l'image serverapp. In questo modo i container mi rappresentano i server su cui caricare le richieste. 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 con kMin container fissati). Il funzionamento del sistema è quello di soddisfare nel miglior modo le richieste. A questo fine sono instanziati due tipologie di carichi:

- carico statico : staticWork:int, richieste presenti all'inizio della simulazione;
- carico dinamico : dynamicWork:int, carico che viene aggiunto dopo un lasso temporale definito.

Per il carico statico vengono istanziate, una ed una sola volta, 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 l'istanza del 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;

3 Implementazione Sistema

3.1 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.

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

3.2 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.

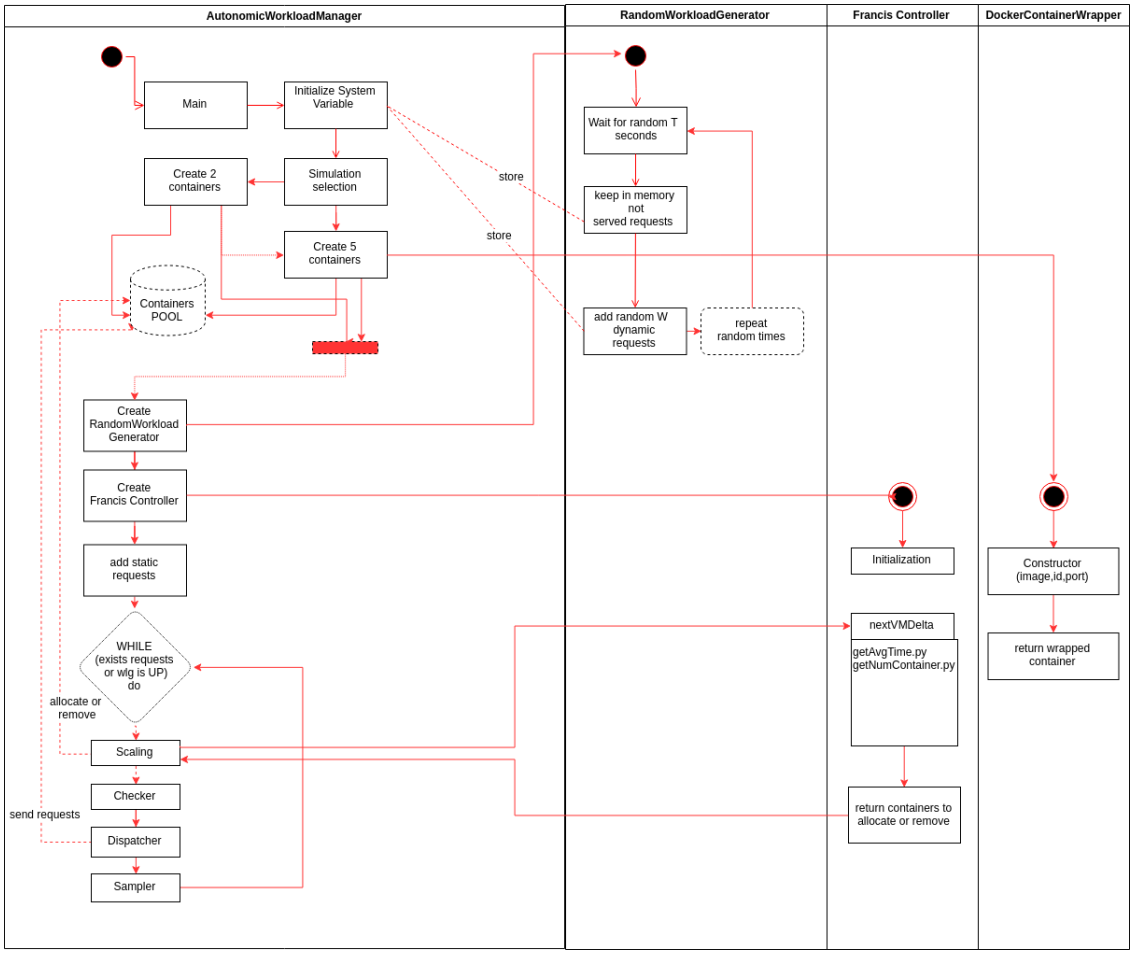

Fig. 7. Docker System Overview

3.3 DockerContainerWrapper

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

3.4 FrancisController

La classe FrancisController rappresenta una particolare istanza di controller che permette di conoscere il numero di istanze da allocare attraverso il metodo "nextVMDelta". FrancisController viene invocato nel main della classe AutonomicWorkloadGenerator. La classe preesistente del progetto CloudAWM, viene modificata solamente nel metodo nextVMDelta() (metodo della classe Francis Controller). 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. Il metodo nextVMDelta necessità di sapere il numero di container attivi al momento dell'invocazione. Per questo motivo la seconda modifica è l'acquisizione del numero di container attivi al momento dell'invocazione. È stato creato lo script getNumContainer.py, per ottenere il numero di container attivi nel sistema. Terza ed ultima modifica al metodo è il parametro etak[0][0]. Ad esso viene assegnato il numero totale di richieste presenti nel sistema. Il responso di questo metodo è un numero intero che rappresenta il numero di container da allocare o rimuovere.

3.5 app.py

Lo script app.py rappresenta il file eseguibile che rappresenta il server, basato su Flask. L'image Docker chiamata "serverapp", viene costruita su lo script app.py. 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.

3.6 startContainer.py

Lo script helloContainer.py rappresenta lo script di avvio del container con image "serverapp". Facilmente implementabile la possibilità di avviare container con image differenti, occorre passare come argomento il nome dell'image e costruirla

```
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. 8. startContainer.py

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

3.7 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. 9. getStatus.py

Lo script getStatus.py viene eseguito dalla classe AutonomicWorkloadManager nella sezione denominata "Checker". Lo script ritorna lo stato dei container. Le API docker non permettono di ottenere facilemente la CPU utilizzata (non è presente un campo). Sono presenti però l'utilizzo della CPU in millisecondi. Nel file è proposto una soluzione parziale per calcolare l'utilizzo della CPU in percentuale con API Docker. La soluzione più semplice è stata quella di prendere le informazioni da terminale attraverso l'istruzione Docker presentata nel file.

3.8 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. 10. getAvgTime.py

Lo script getAvgTime.py viene invocato dal Francis controller nel metodo nextVMDelta(). Lo script controlla da quanto sono attive tutti i container e restituisce una semplice media in secondi. Adotta la stessa logica del getAvg-WindowedCloudTime nella classe Qos. Lo script restituisce la media dei tempi di esecuzione dei container in una finestra temporale di 4 minuti precedenti.

3.9 client.py

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

Fig. 11. client.py

Lo script client.py viene eseguito nel AutonomicWorkloadManager nella sezione denominata "Dispatcher". Lo script client esegue una semplice richiesta HTTP che avvia start_task su server. In questo modo eseguo del carico di lavoro sul server.

3.10 killContainers.py

Lo script killContainers viene utilizzato all'interno del AWM come procedura di reset per il sistema. In questo modo posso reinizializzare la simulazione.

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 è quello con l'utilizzo del controller Francis mentre la seconda viene fatta senza nessun controller. Quello che ci si aspetta è 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 nè di allocare nè 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	random
Carico dinamico	random
Tempo carico dinamico	random
Controller	Francis o Nessuno

5 Grafici

In [Fig. 12] 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 pre-

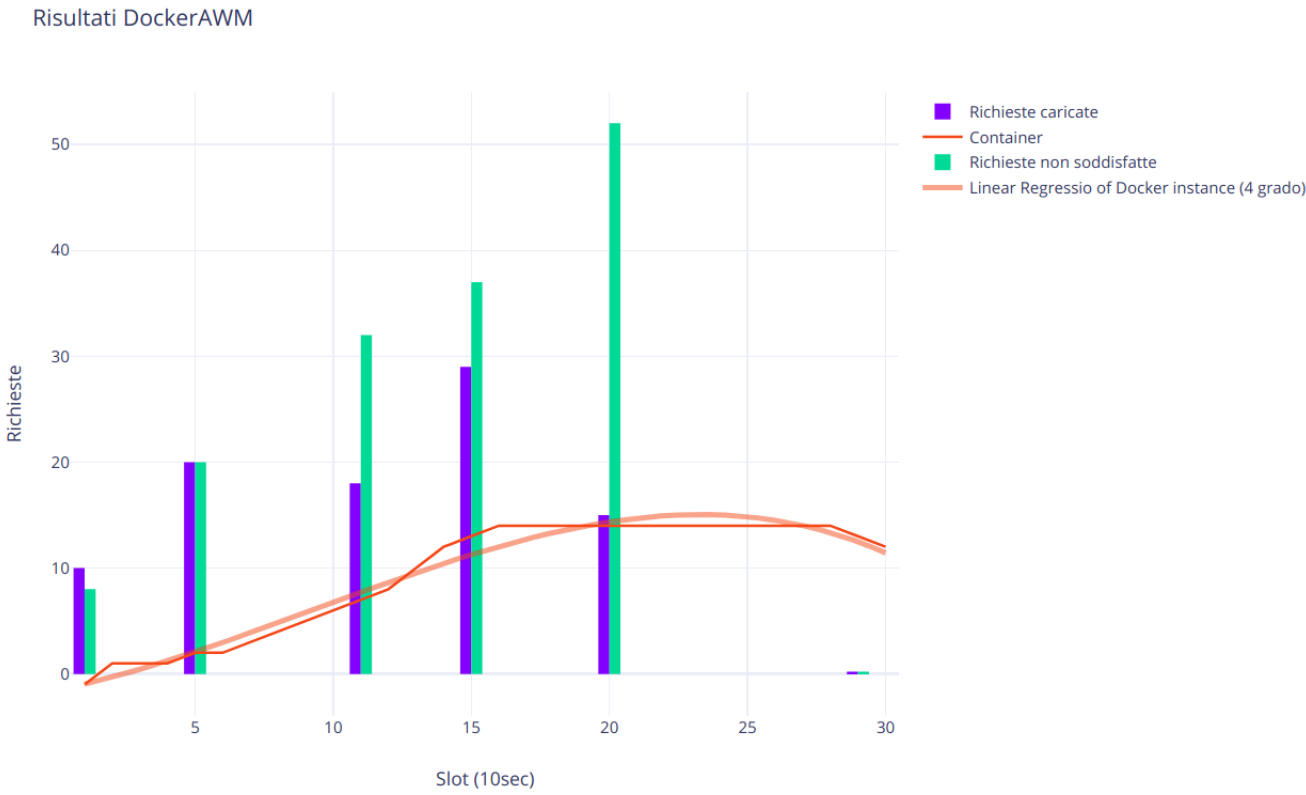

Fig. 12. Grafico allocazione container

sentate le configurazioni reali:
La simulazione con controller è stata inizializzata con questi parametri:

- staticWork = 10 (richieste)
- dynamicWork = 20 (richieste) * 5 (volte)
- delay = 50 (secondi)
- kMin = 2 (container)

La simulazione senza controller è stata inizializzata con questi parametri:

- staticWork = 10 (richieste)

- dynamicWork = 20 (richieste) * 5 (volte)
- delay = 50 (secondi)
- kMin = 5 (container)

Nella versione aggiornata di DockerAWM si presenta la soluzione randomica sia per i carichi dinamici sia per tempi di inserimento delle richieste. Inoltre viene aggiunta randomicità su quante volte avviare il generatore di carico. Per il caso presentato una stima del tempo totale di esecuzione di tutte le richieste è uguale al calcolo del tempo di lavoro atteso per ogni richieste (circa 13 secondi) moltiplicato per le richieste totali, prese a carico. Le richieste vengono eseguite in modo sequenziale: $(20 * 5 + 10) * (13sec) = 148,50sec$

Table 2. Comparazione dei risultati delle simulazioni

Indice	Risultato
0	101,70 sec
1	134,04 sec
2	148,50 sec

Nella tabella sono rappresentati il tempo di esecuzione del sistema con francis Controller rappresentato dall'indice 0 e il tempo di esecuzione del sistema DockerAWM senza l'uso di un controller per allocare dinamicamente i container. In quest'ultimo caso vengono istanziati alla partenza del sistema un numero minimo di container superiore al caso con controller, simulando un approccio statico di allocazione massiccia iniziale. I Risultati confermano che un approccio di scaling dinamico dei container favorisce a mantenere minimo i tempi di esecuzione di tutte le richieste. All'indice 2 viene presentato il caso peggiore di esecuzione delle richieste, ovvero il caso sequenziale. Nel caso presentato in [Fig. 12] il tempo di esecuzione del sistema è di 140.20 sec e il sistema si adatta ottimamente al numero di richieste variabili del sistema.

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 è 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].