

CS352: Cloud Computing Jan-May 2019

SelfieLess Acts.

CONTAINER ORCHESTRATION & CLOUD WEB APP.

CS352 | 04-05-2019

|  |  |  |  |
| --- | --- | --- | --- |
| SNo | Name | USN | Class/Section |
| 1 | HRISHIKESH S. | 01FB16ECS139 | C |
| 2 | KARTIK A. NARAVAGOL | 01FB16ECS159 | C |
| 3 | KARTHIK R. | 01FB16ECS158 | C |
| 4 | HARSHITH H. K . | 01FB17ECS713 | C |

# Introduction

For our Cloud Computing Project, we are working on a web application called SelfieLessActs. In this cloud based-web application we are able to upload acts of people doing Selfless acts, assign a category , caption to it.

Users are also able to upvote each act via front-end.

We are using EC2 instances of A.W.S. Cloud services and to make the REST APIs, we are using Flask. The Web application is divided into two parts – The users and acts, each of which is a disk image which we have created and stored in Docker hub. These images are run on Docker Containers and the traffic to these containers are balanced by Load balancers. The Container Orchestration consists of Fault tolerance and Auto Scaling which will be explained in detail in the next sections.

## Related work

1. Cloud Computing course and slides

2. Docker Volumes and Container official documentation

3. Flask official Documentation

4. Docker sdk official documentation

## EXPERIMENTS/DESIGN.

**Programming Language** : Python

JavaScript (for frontend)

**Modules & Tools used** : Flask

requests library

threading library

docker SDK for container management in python

HTML/CSS (frontend)

AJAX (frontend-backend communication)

For managing container orchestration, we have Implemented Fault Tolerance and Auto-Scaling.

In our implementation, we have three threads.

1. Running the flask application where the requests are serviced in a round-robin manner.

2. Fault tolerance

3. Auto scaling thread supported by upscaling and downscaling functions.

We will be attaching docker volumes as our implementation of a database is with a file system, while starting the containers.

We are maintaining a dictionary that contains all ports of the active containers and their corresponding docker container ID.

Health check and crash API are managed with a variable ‘healthy’ which indicate the health of the container.

During fault tolerance, we iterate over the list of active ports and send a request to the health check API that has been implemented in each container. If the health check API returns a status of code of 200, it indicates that the container is healthy, or else it will return 500, which forces the container to restart.

During auto scaling, which is called every 2 minutes, the fault tolerance thread is disabled using a semaphore and active\_ports variable is locked using an RLock from python threading library. If we did not implement this, fault tolerance would give a run-time error, stopping the thread.

For calculating the scale factor for auto-scaling, we have a formula.

scale\_factor = r – n +1

where ‘r’ is the number of containers that should exist after auto scaling and ‘n’ is the number of currently existing containers.

During upscaling, we will iterate over the expected number of containers and if any port is not active, then we start a container at the given port.

During downscaling, we will iterate over the containers that are expected to be disabled, and if any port here is active, we will stop those containers immediately.

We have to specify timeout as 0 seconds, as by default, docker sdk takes 10 seconds to stop a container.

Initially, we had automatically deployed our images by linking Github to DockerHub and enabling automated build. The images acts:latest and users:latest are available publicly.

Our entire project has been version controlled using Git and each branch represents our project for each assignment & project evaluations.

**PSEUDO CODE FOR FAULT TOLERANCE.**

faultTolerance():

for port in list of active ports:

send get requests

get response status code

if(response code == 500)

restart container

else

return “no fault found” , 200

**PSEDUO CODE FOR AUTO-SCALING**

auto\_scaling():

In every 120 seconds,

scale\_factor = expected number of containers – number of container + 1

if containers\_to\_be\_created > no\_of\_active\_ports :

up\_scale(scale\_factor)

elif the containers\_to\_be\_created < no\_of\_active\_ports :

down\_scale(scale\_factor)

else:

print("No scaling")

**PSEUDO-CODE FOR DOWN-SCALING**

down\_scale(scale\_factor):

for port\_i in range(act\_port\_end +scale\_factor, act\_port\_end):

if(port\_i in active\_ports):

stop container at port\_i

**PSEUDO-CODE FOR UP-SCALING**

up\_scale(scale factor):

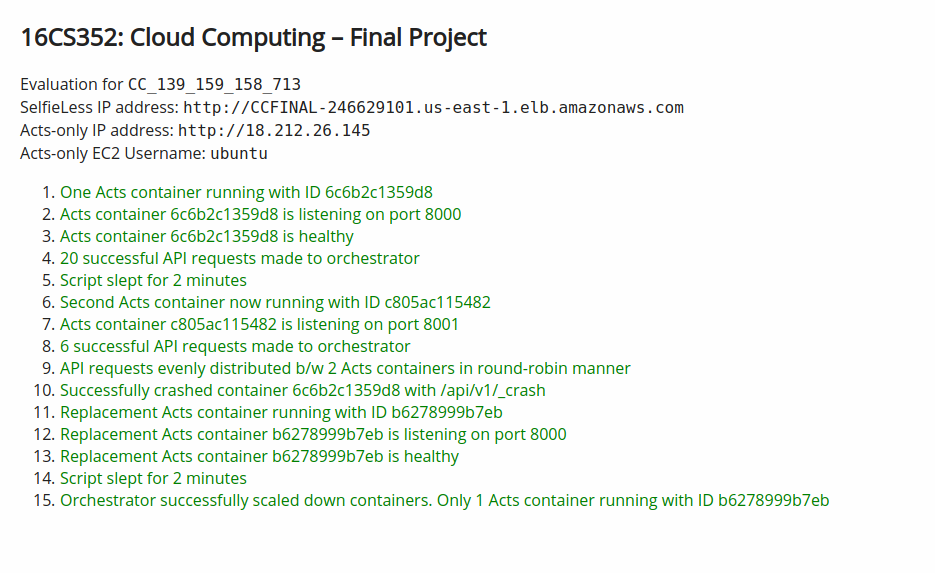
act\_port\_end += scale\_factor -1

for port\_i in range(ports):

if(port\_i not in active\_ports):

run container at port\_i

## TESTING/RESULTS



## REFERENCES

1. Cloud Computing course and slides

2. Docker Volumes and Container official documentation

3. Flask official Documentation

4. Docker official documentation

## EVALUATIONS (Leave this for the faculty)

|  |  |  |  |
| --- | --- | --- | --- |
| Date | Evaluator | Comments | Score |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

## CHECKLIST

|  |  |  |
| --- | --- | --- |
| S.No. | Item | Status |
| 1. | Source code documented | Yes |
| 2. | Source code uploaded to CCBD server | -- |
| 3. | Source code in GitLab. Please do not upload your source code to GitHub where it can be seen by everyone. | https://gitlab.com/redlegblackarm/SelfieLessActs |