EECS6432 - Adaptive Software Systems (Fall 2018-2019)
Daniel Marchena Parreira - 216181497

Part 1

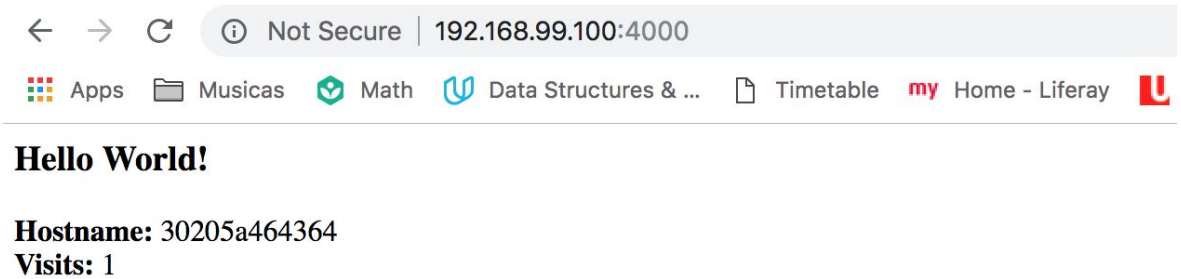
I've completed tutorial 1 to 4 but using a different approach. On that note, the tutorials covered virtual machine creation, network interface setup and docker daemon installation in a truly old fashion. Although there is nothing wrong with this methodology, I decided to check Dockers website and found an end to end tutorial on how to build a Docker image, publish it to docker hub, orchestrate services with docker-compose, set up virtual machines with docker-machine, create a docker-swarms and much more. The link for this official tutorial can be found here, and I also decided to write my own step by step which can be accessed here.

For this reason, my first step when doing this assignment was to install both Docker for Mac and Oracle's Virtualbox in my personal machine and run the command `docker-machine create --driver virtualbox my_vm_name` in order to create my virtual machines. This command spawns a virtual machine with boot2docker which is a Lightweight Linux for Docker distro that has Docker Daemon pre-installed and configured. Also, this request creates and configures a TLS certificate in order to secure the communication between the virtual machine and other machines (other VMS or the host itself).

Furthermore, after setting up two virtual machines using docker-machine, I connected to both by SSH and did a setup in order that one would act as a master node (`docker swarm init`) and the other would act as a standard node (`docker swarm join`). After establishing this relationship between both Docker Daemon clients, the next step was to deploy an application. Having said that, I wrote a simple web app using Python 3 and Flask that outputs the server hostname and the total number of visits the web app has. To store this information, I connected the web app to a Redis instance and incremented a counter by one whenever someone visited the page. The code for the app can be found here.

Subsequently, I developed a Dockerfile to provision the Flask application and a docker-compose.yml file to orchestrate all services (Flask app, Redis, etc..) on the stack. Therefore the last step was to deploy those services to my Docker Swarm, and this was done by running `docker stack deploy -c docker-compose.yml getstartedlab` inside the Swarm master node. This decides where each container should be deployed depending on the number of resources of each VM.

Finally, after the deploy, I was able to reach the application on the web browser by accessing any of my VMs IP address followed by the application port (all this information was mapped inside the docker-compose.yml file). It is also important to mention that the Flask app container was communicating with the Redis instance container by using Docker's overlay network. Further details about the setup process and the code can be found in the README inside the repository mentioned in the first paragraph.



Part 2

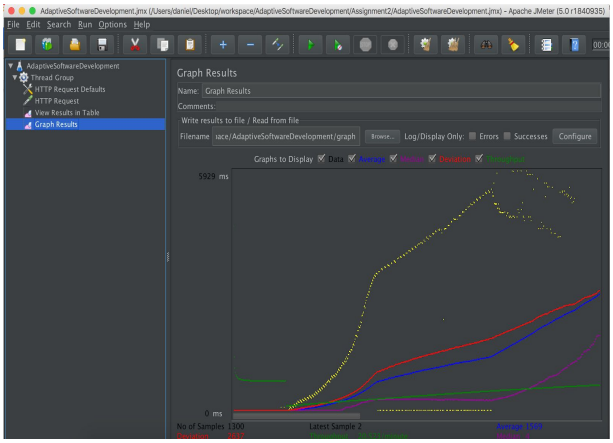
The program that reads the performance from the containers, that computes the average utilization of the containers CPU, displays it for end users and scales the application depending on the setpoints can be found here. This Python script was developed based on the one available at the end of Tutorial 4, and it was modified to support end-users to provide setpoints (CPU threshold) and TLS certificates for a secure communication with the VMs.

The experiment was recorded and can be found here. Having said that, during the demo, I initially establish a load of 100 requests to my web application. That load was not sufficient to hit the PID controller threshold (10% min and 40% max CPU). However, right after that, a test with a batch of 600 requests was done, and that was able to overshoot the setpoints extremely fast.

Consequently, the load balancer noticed that the average CPU usage (our process value) was way higher than the established setpoints (10% min and 40% max CPU) and started scaling up the application. It is important to mention that the rising time was in question of seconds after the batch of requests was sent, the average CPU usage came from a steady 17.16% to 291.29% in less than 5 seconds. The scale-up function started to provision more container for the web app service in order to respond to every single request.

As a result, the average CPU usage started going down as the requests were answered by the swarm of growing containers, and eventually, it starts decreasing to the point our process value was lower than 10% SP. That lead the PID controller to scale down the application since we didn't need all that processing power anymore.

Finally, a test was done using new setpoints (10% min and 100% max CPU), and I could clearly see that the steady-state error was reached faster compared to the previous test. The reason why is part related to the fact that the difference between the process variable and the setpoint is smaller in this second scenario. In this test, we also reach a similar extreme of 300% PV, but the scaling-up process is more efficient since achieving the 100% SP is easier compared to 40% SP. This reduces the processing needed for spawning and destroying containers.




```
uvn
<<------------>>
Analyzing getstartedlab_web service
<<------------>>
CPU Usage (avg): 2.33%
CPU Setpoints are currently from 10.00% to 100.00%
-------------
CPU Usage (avg): 2.01%
CPU Setpoints are currently from 10.00% to 100.00%
-------------
CPU Usage (avg): 169.32%
CPU Setpoints are currently from 10.00% to 100.00%
-------------
scaling triggered...
getstartedlab_web tasks:
        ContainerID: f3a237e0729cb846477afca1af3f440c0402f8cb63a8b
uvn
CPU Usage (avg): 206.08%
CPU Setpoints are currently from 10.00% to 100.00%
-------------
```