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.

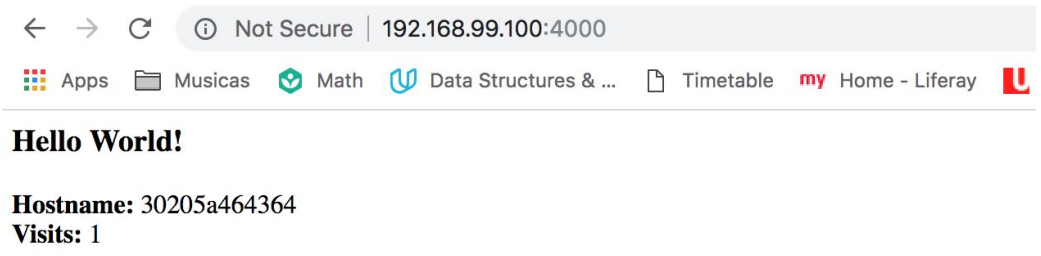
For this reason, my first step when doing this assignment was to install both Docker for Mac and Oracle's Virtualbox in my personal machine and run the command `docker-machine create --driver virtualbox my_vm_name` in order to create my virtual machines. This command spawns a virtual machine with boot2docker 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.

A demo explaining the architecture in further detail was recorded and can be found here.



Part 2

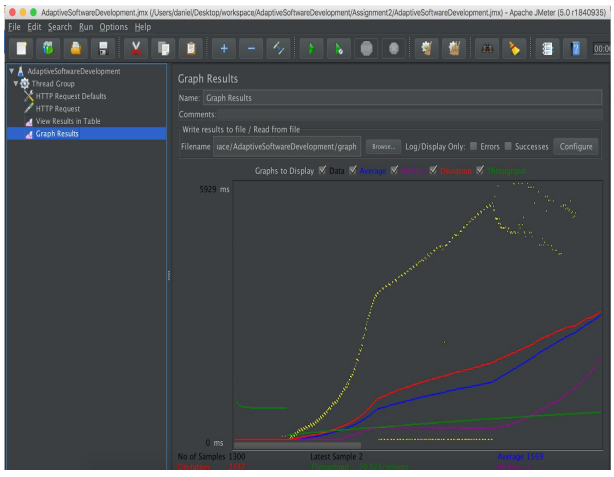
The program that reads the performance from the containers, that computes their average CPU utilization, displays it to the end user and scales the application depending on the defined setpoint can be found here. This Python script was developed based on the one available in Tutorial 4, but it was modified to support end-users to provide dynamic setpoints (CPU threshold), an interval in seconds to determine data collection frequency and Kp/Ki/Kd parameters. Also, this code supports TLS certificates for a secure communication with the VMs.

Furthermore, during my tests, I initially send a load of 100 requests with a ramp-up period of 10 seconds to my web application. Those requests were not sufficient to make the CPU average of my containers surpass the PID controller setpoint. Having said that, the balancer was set up with a CPU threshold between 10% to 40% (setpoint = (10 + 40) / 2 = 25%) and Kp, Ki and Kd equal 1.

In addition to that, I ran another test with a batch of 600 requests, and that one was able to overshoot the setpoint extremely fast. Consequently, the load balancer noticed that the average CPU usage (our process value) was way higher than the established setpoint (25% max CPU) and started scaling up the application. It is important to mention that the rising time was about 5 seconds after the batch of requests were sent, the average CPU usage came from a steady 17.16% to 291.29% in that interval. The scale-up function was called and 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 25%. That lead the PID controller to scale down the application since we didn't need all that processing power anymore.

Finally, one more test was done using the same request load (600 requests), but with a setpoint of 80% and Kp, Ki and Kd equal 2. Subsequently, I could clearly see that the steady-state error was reached faster compared to the previous test. The reason why is related to the fact that Kp, Ki and Kd impacted directly on the number of containers needed to be provisioned in order to stabilize our system. Previously the PID controller was provisioning 5 to 7 containers, with this change it started provisioning 10 to 15. Also, whenever we reached our setpoints after scaling up, the scaling down process would also happen faster because of the higher Kp, Ki and Kd values.




```
rk4cv9t4ouvn
Output Proportional Controller: 0.3324722222222222
Output Integral Controller: 0.08763888888888889
Output Derivative Controller: 0.26597777777777776
Controller final output: 1
Previous CPU Usage (avg): 1.75%
CPU Usage (avg): 1.75%
CPU Setpoints are currently from 20.00% to 50.00%
Scale parameter value:  1
-------------
getstartedlab_web tasks:
        ContainerID: 184c758f8412e42e9f7d03e631369067de1adc9
rk4cv9t4ouvn
Output Proportional Controller: -10.292669680851064
Output Integral Controller: 53.30098729314421
Output Derivative Controller: -10.359164125295509
Controller final output: 33
Previous CPU Usage (avg): 1.75%
CPU Usage (avg): 1064.27%
CPU Setpoints are currently from 20.00% to 50.00%
Scale parameter value:  15
-------------
```