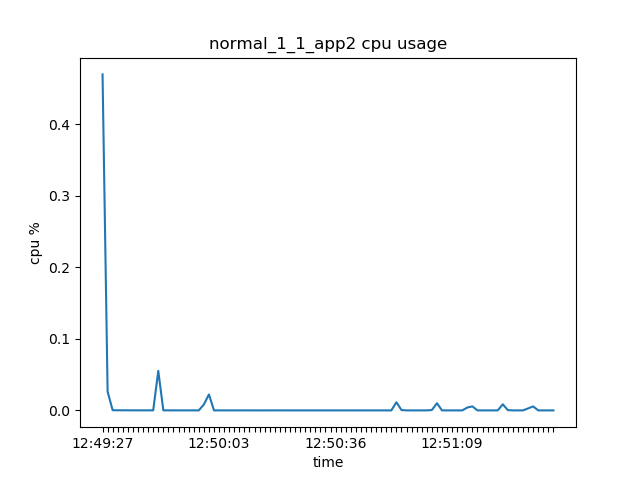


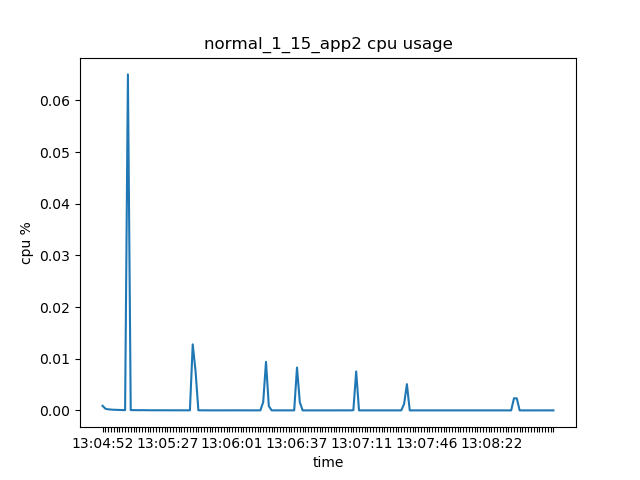
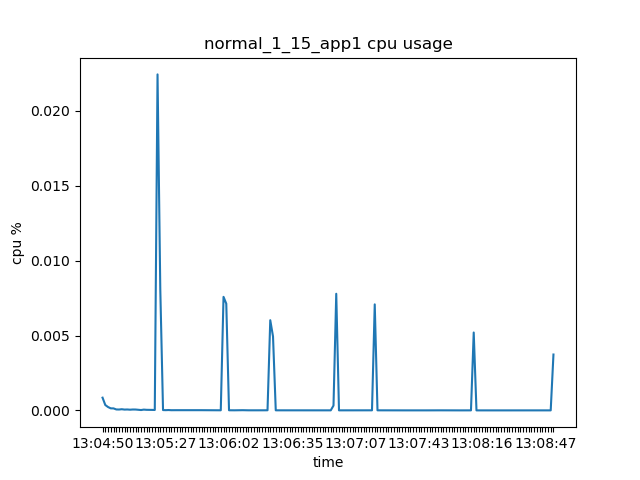
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | | | | | | | | | | | | | | | | |
|  | | | CLOUD COMPUTINGCOURSEWORK REPORT | | | | | | | | | |  | | | |
|  | |  | | | | | | | | | | | |  | | |
|  | | | | | |  | | |  | | | | | | | |
|  | | | | | | ALEXANDROS **RANTOS**  Student Number  **190587101** | | |  | | | | | | | |
|  | | | | | | DATE 16/12/2019 | | |  | | | | | | | |
|  | | | | |  | | | | |  | | | | | | |
|  | | | | DESIGN & CODE ANALYSIS | | | | | | |  | | | | | | |
|  | | | |  | | |  |  | | |
|  | The technologies used for this coursework are docker, docker swarm and Amazon Web services. Docker is a tool that makes it possible to run and automate applications in the desired environment. It’s a lightweight software package which acts as a very efficient virtualization tool since there is almost no overhead. However, you can run only one image of an application in docker hence using Docker Swarm was mandatory in order to overcome this obstacle. Docker Swarm is a tool that allows the user to group and manage multiple containers in his host environment. It follows a decentralized design and provides auto-balancing within the environment. Finally, AWS was also used to run the above solution in the cloud and not locally.PART 1 Firstly, docker needs to pull the image from the registry and this is done by running this command from the command line:  Afterwards, in order to run that image, the following line had to be typed:    The publish flag is added to run the container on the port:  (publish port)8080:8080(container port)  By that time the container is running locally and can be accessed this URL  <http://localhost:8080/primecheck> as shown below:    The same can be done using the Docker Engine SDK running the python script    The detach parameter is optional in the occasion that the container has to run in the background. | | | | | | | | | | | | | | |  | |
|  | PART 2 A multi-service application had to be deployed in a docker Environment thus Docker Swarm was used. The above could be achieved via the command line interface by running the following command:  the .yalm file defines the services, their volumes and networks as it is shown here:  As seen in .yaml it contains the image of the containers, the published port:container port, the files that each container can read(ro) and in the situation of our application how many times should be virtually redeployed (replicas).  Same can be achieved by running this python script:    No volumes have been specified for mongo dB since there was no such specification in the assignment description. In that way, if the docker stops or restarts the database loses its data.  A UML of the above architecture is shown below:    Made in <https://www.draw.io/>  Another detail that should be pointed out is that cAdvisor is being deployed in this part as well and not separately in Part 4. | | | | | | | | | | | | | | |  | |
|  | PART 3 + 4 In order to generate load to the web application, urllib.request python library was used so URLs can be visited via the python script.  The correct arguments’ values and format for the script to run correctly are shown below:    For normal and Poisson distribution, numpy functions (numpy.random.normal & numpy.random.poisson) were used to calculate the inter-request time xi.  In general, the load generator script is calling the web application and calculates the xi accordingly with the input values. If any negative values occurred they are inverted to positives ones with the abs() function and are inputted in the time.sleep() function to simulate idle activity between requests. In addition, cAdvisor API is called to fetch the JSON that contains all the data related with the application performance and specifically the values of CPU, memory and network I/O utilization, all of which are stored in the mongo database via pymongo library. As far as database design is concerned, each collection is created within the database named “benchmarks” and has the following name pattern {name\_of\_distribution}\_{distribution\_values}\_{appname} so easier conclusions could be drawn between the given input & the load which was generated.  Lastly, 12 different loads generated for the web application and the motif was an increment by 5 per distribution value. For example, 8 different inputs were created from normal distribution (Normal μ = 1, 5, 10, 15 with σ = 1 and μ = 1 with σ = 1, 5, 10, 15 and one with μ = σ = 10) and accordingly for 5 different Poisson distribution (λ = 1, 5, 10, 15). This load pattern was automated and can be reproduced running this script: run\_preset\_load.py. In this fashion, a variety of virtual activity was produced and help to gather data under different circumstances.  Visual representation of the generated request time is provided in the following histograms:  Poisson Distribution      Normal Distribution incrementing μ      Normal Distribution incrementing σ | | | | | | | | | | | | | | |  | |
|  | PART 5 Last part of this project is to visually present the data that was gathered during the load of part 4. Again, pymongo was used to gain access to the mongo database through URI to fetch all the stored data. Firstly, all records from each collection are being fetched and sorted by ascending timestamp. Afterwards, the data is manipulated in order to get similar graphs with the ones of the cAdvisor. In detail:   * Network I/O difference is calculated * Memory is transformed into Megabytes * CPU percentage is calculated by the following formula | | | | | | | | | | |  | | |

All the above is represented via plots in correlation with time. All plots can be found at “./graphs/results/{distribution\_used}/{file\_name}. Each name’s file specifies the distribution, the distribution values and the application (for simplicity app1 & app2 was used to refer to the each of the replicas).

In this paragraph, the most noticeable trends, which were observed analyzing the above produces plots, are going to be discussed. First of all, as it was expected the overall application load that was produced by low values of mu’s and sigma’s on normal distribution or low lambda’s value on poisons accordingly was more demanded opposed to the one that was produced by high values on both formula’s. The apparent reason is that as long as the inter-request time is rising the time between requests is also increasing hence the overall traffic is decreasing. The difference can be observed in the following examples.



As it can be observed the maximum CPU load is 0.4% on both applications in the span of approximately two and a half minutes. On the other hand, the maximum CPU usage percentage if the same requests are expanding in the span of 4 minutes is 0.06 which is greatly less than the 0.4 that was observed in the above examples. Not only that the overall CPU use is higher but also the spikes that were occurred during the requests are sharper and longer. The same graphs when sigma = 15 can be seen here:



However, the same cannot be said for memory usage. It is important to note that sometimes the memory was steady while the applications were on load. In the following cAdvisor screenshots, the memory usage is steady for both applications at approximately 116MB and 119MB correspondingly but when the load starts (CPU & Network I/O spikes) the memory usage stays the same.

A screenshot of a computer

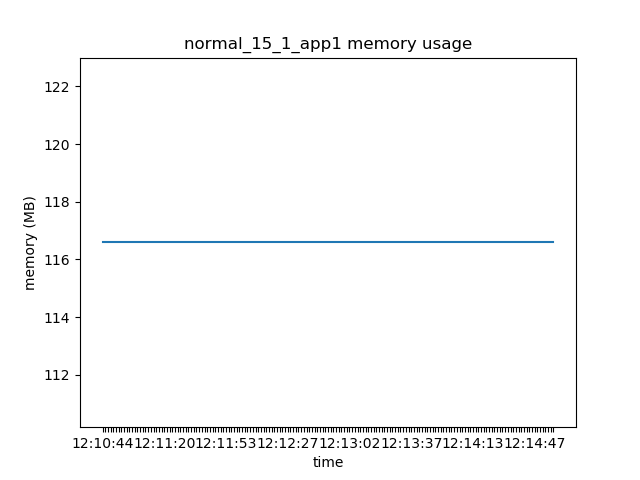
Description automatically generated

However, at the end of another load the memory drops a bit:



The application has reserved around ~115+ΜΒ of memory (specific number cannot be given because a lot of unknown factors need to be taken into consideration hence this number varies) so maybe the applications had already reserved enough memory to run which may be caused by the high traffic.

It is worth to point out that, few rises of memory usage were observed only in one of the two applications:



However, the memory of normal distribution with mu = sigma = 1 (roughly 115.8MB) is the less compared to the one with the same distribution with mu = 15 and sigma = 1 (around 116.7MB) in application 1:

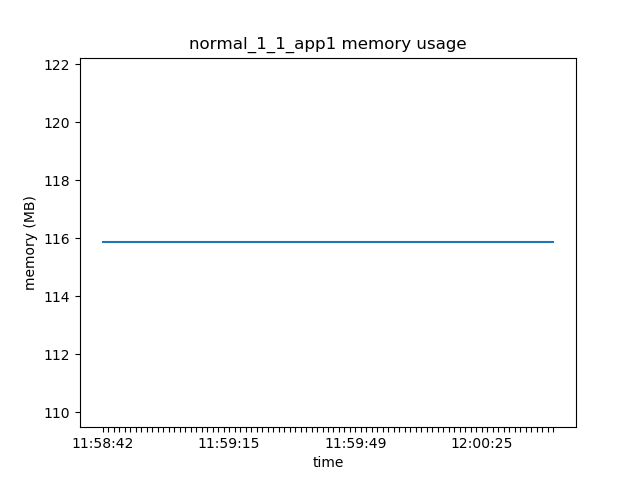
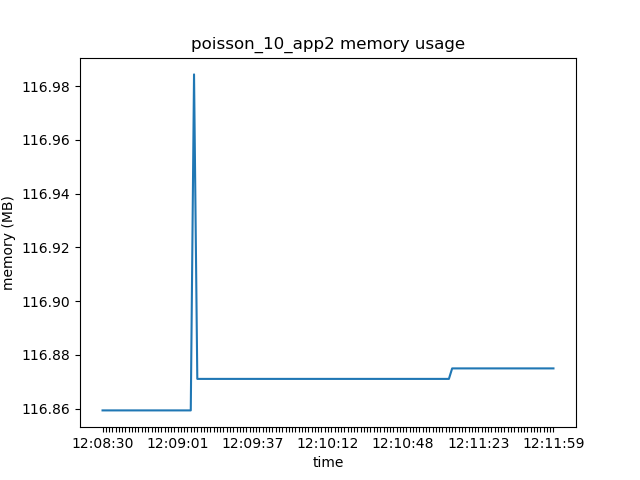
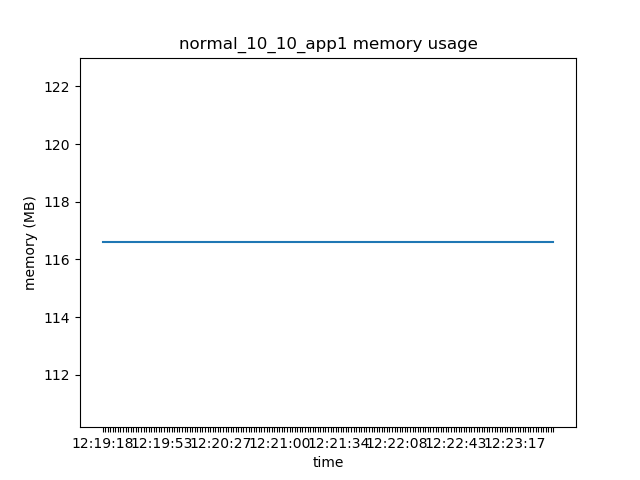


Figure 1

It can be concluded, that since the requests happened successively the traffic kept increasing and the applications could run successfully with the already reserved memory but there were moments that more memory was needed so a sudden increase could be observed in some graphs like in this one:

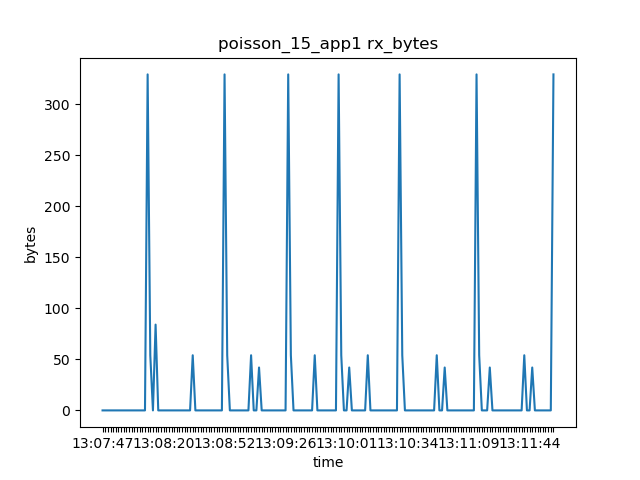


An interesting outcome can be seen if the very first request that was produced using normal distribution and mu = sigma = 1, as shown above in Figure 1, with the last one was running with normal distribution again but mu = sigma = 10:

 are compared.

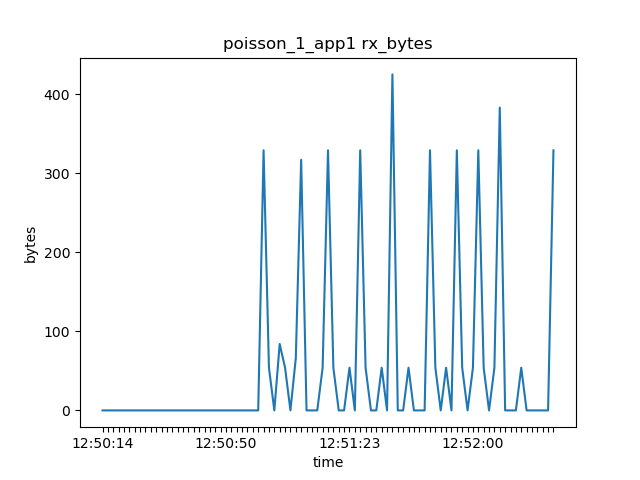
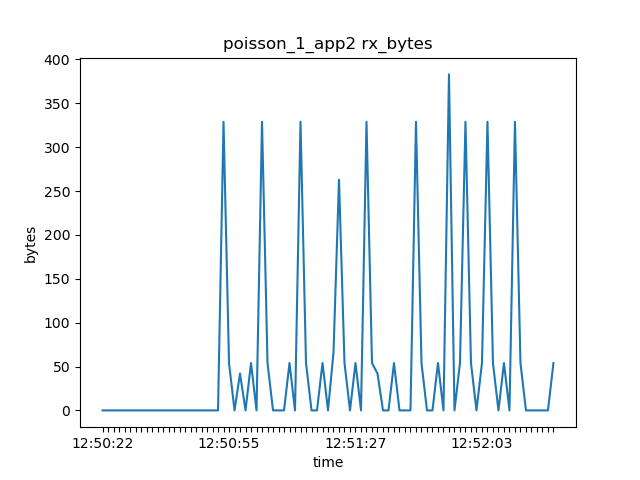
Even though both of those graphs seem identical the memory usage has slightly increased in the second one. A lot of factors can be the cause of such behaviour since a lot of ready services has been utilized. For this reason, as stated above, it is difficult to point out what exactly could cause it and only speculations can be made.

As far as network I/O is concerned, it follows a similar pattern to the CPU usage, as it can be seen below:



In general, when the request is received there is a spike, so such spikes are more scattered when the inter-time request is high but the spikes themselves are identical.

For example, if lambda = 15 for Poisson distribution the peaks are more scattered. However, the same sum of bytes is transmitted throughout any request the only difference in the tests is time-frequency that’s why the top of the graphs with high values of lambda is the same with one of the low values.



In conclusion, cloud application must take into consideration the time-frequency of the incoming requests to defend themselves from overloading the hardware and unwanted downtimes. Before any deployment, such benchmarks must be performed to define the golden ration between cost and efficiency.