Assignment 2

Application Idea :

Server Application - Created as a Maven Project in NetBeans IDE with the following classes

1. DataSource.java - uses a static C3P0 ComboPoolDataSource and has the RDS instance login credentials
2. FitbitResource.java - Java class which has the code to handle the endpoints for the assignment
3. UserDAO.java - Java class to implement the logic for each endpoint
4. RESTConfig.java - This has the application path.

I used Glassfish 5.0 for deployment purpose for this assignment. Initially I deployed it with Glassfish 4.1 but this created a lot of issues for deployment of the final web server application in EC2. The problem I was facing was an exception related to the org.glassfish.jersey.containers.glassfish gf-CDI jar. Unable to resolve this issue I decided to switch to a newer version of Glassfish.

For the RDS instance I made use of t2.medium which allows 312 threads of connection at a time and EC2 instance of type t2.micro

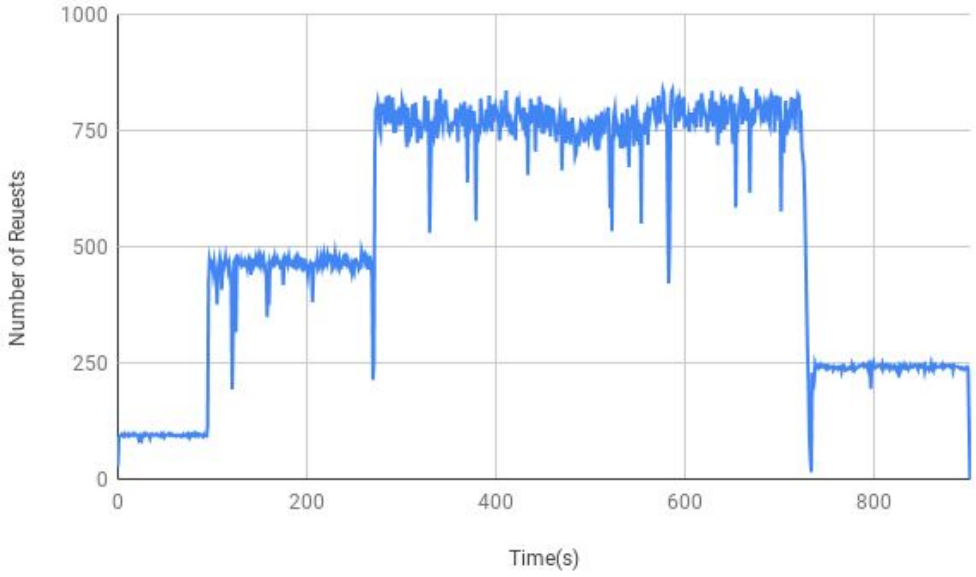
Client Application -Created as a Maven Project in Eclipse IDE with the following classes

1. FitbitClient.java - A Java class that implements the logic for a Jersey Client application which will access the web server endpoints. I have made use of Blocking Queue Data Structure to keep the count of number of responses, requests and successful requests. I have used a HashMap<Long,ArrayList<Long>> for tracking all the request latencies as well. Whenever a response for successful request is received by the client , the blocking queue adds an entry to the map for the request timestamp and request latency(both long values). Since this application sends concurrent requests, it's highly likely that any two requests have the same timestamp. Hence the value of the hashmap is a list of all latencies for that timestamp. The postprocessing of the map is to convert the map to a TreeMap(sorted on key-time in milliseconds) and then making a new TreeMap with buckets of requests latencies map to each second(function MapToSecondsBucket()). I finally write both the Original treemap and the Seconds treemap to output files. This way I was able to retrieve the data for plotting and p99 and p95 as well.

Current flaws in design : Could not implement non overlapping phases, I also have multiple blocking queues for calculating number of requests and response. I plan on reducing this number by a few. But since these are used only for post processing I decided to keep them. Interestingly I tested my application with concurrent hashmap for keeping the latencies initially instead of keeping a blocking queue but that delayed the performance of my application.

Plots and Statistics:

Default Client Settings: 64 threads in peak phase and 100 iterations:

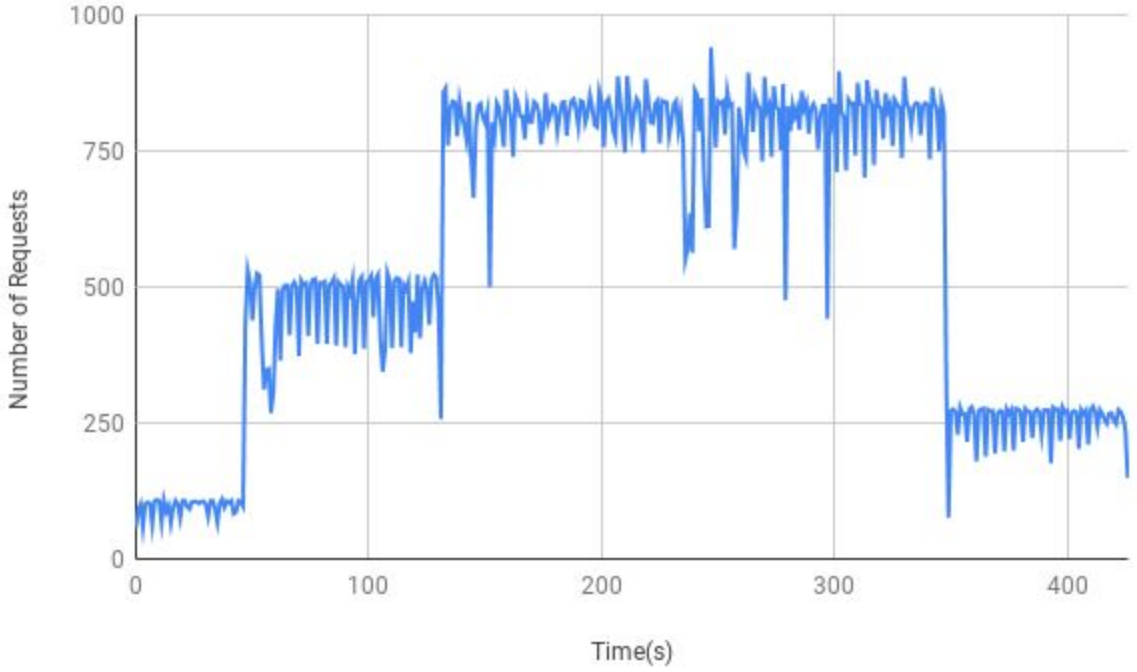



```
Client: Thread Count: 64, Iteration Count: 100
Client starting time: 1540592341095 milliseconds
Warmup phase:All threads running...
Warmup phase complete time: 96.718 seconds
Load phase:All threads running...
Load phase complete time: 176.384 seconds
Peak phase:All threads running...
Peak phase complete time: 464.644 seconds
Cooldown phase:All threads running...
Cooldown phase complete time: 167.799 seconds
Client end time: 1540593246654 milliseconds
========================================
Total requests sent: 481000
Total response received: 481000
Total wall time: 905.559 seconds
Total number of successful requests: 481000
Overall throughput across all phases: 531.1636114793614 seconds
latency of 95th Percentile: 0.067 seconds
latency of 99th Percentile: 0.055 seconds
```

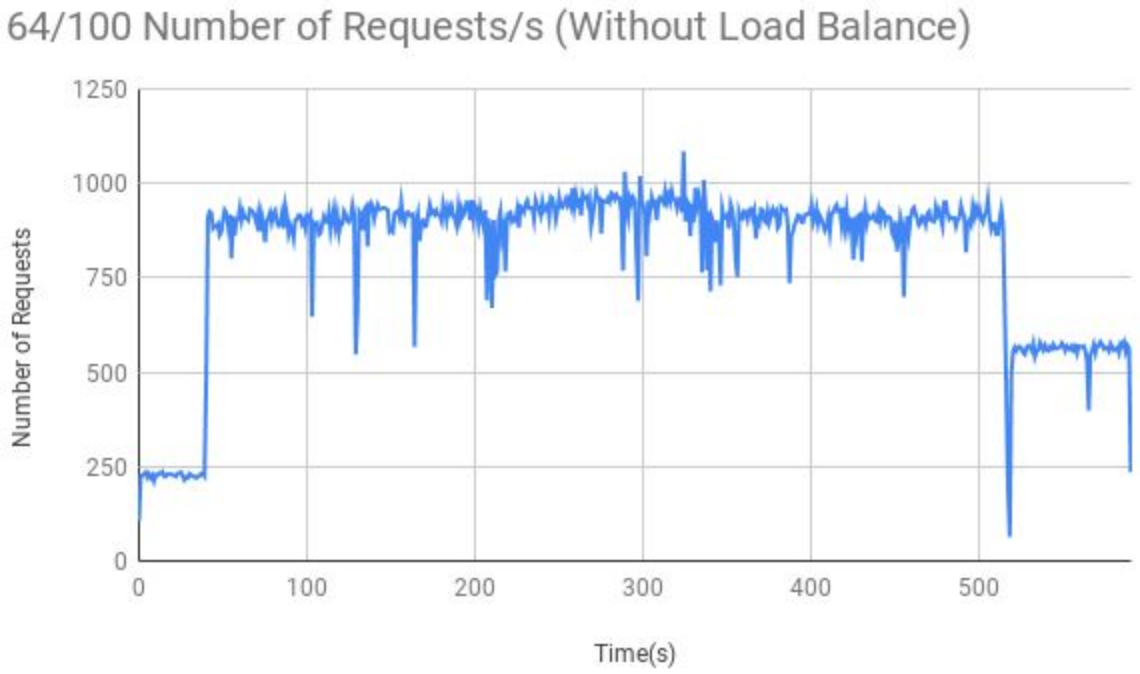
Step 3:


```
Client: Thread Count: 32, Iteration Count: 100
Client starting time: 1540796036011 milliseconds
Warmup phase:All threads running...
Warmup phase complete time: 38.380 seconds
Load phase:All threads running...
Load phase complete time: 71.060 seconds
Peak phase:All threads running...
Peak phase complete time: 193.820 seconds
Cooldown phase:All threads running...
Cooldown phase complete time: 66.863 seconds
Client end time: 1540796406140 milliseconds
========================================
Total requests sent: 240500
Total response received: 240500
Total wall time: 370.129 seconds
Total number of successful requests: 240500
Overall throughput across all phases: 649.7734612195339 seconds
latency of 95th Percentile: 0.040 seconds
latency of 99th Percentile: 0.023 seconds
```

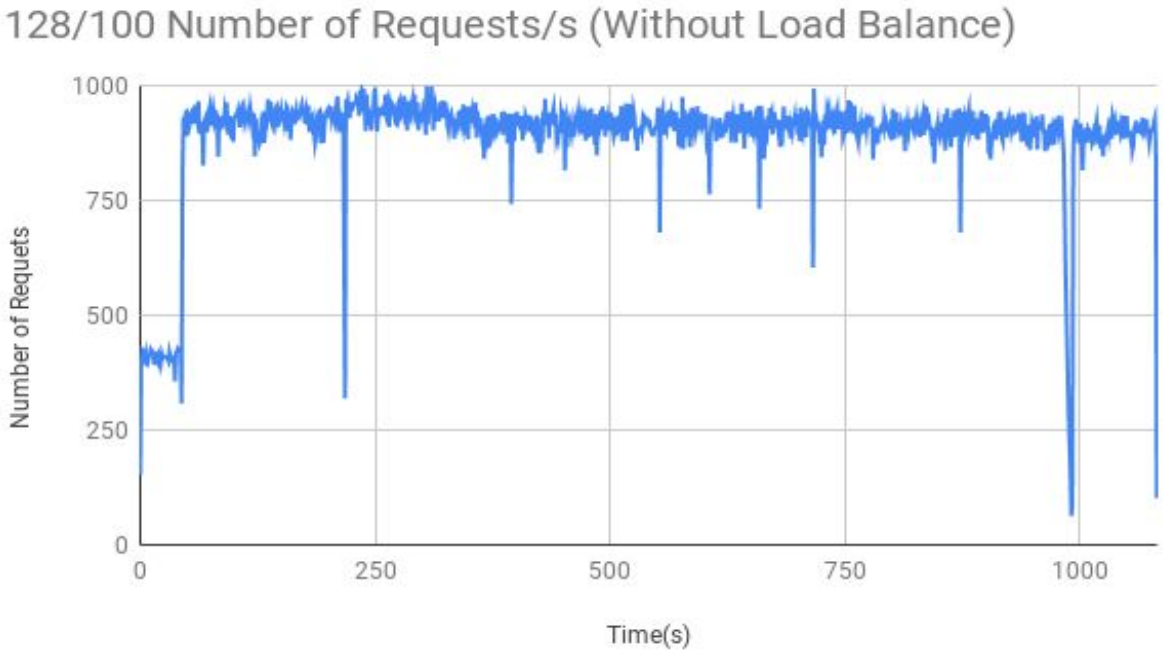
With Concurrent HashMap:




```
Client: Thread Count: 32, Iteration Count: 100
Client starting time: 1540598162892 milliseconds
Warmup phase:All threads running...
Warmup phase complete time: 47.489 seconds
Load phase:All threads running...
Load phase complete time: 85.360 seconds
Peak phase:All threads running...
Peak phase complete time: 218.599 seconds
Cooldown phase:All threads running...
Cooldown phase complete time: 77.558 seconds
Client end time: 1540598591905 milliseconds
========================================
Total requests sent: 240500
Total response received: 240500
Total wall time: 429.013 seconds
Total number of successful requests: 240500
Overall throughput across all phases: 560.5890724203575 seconds
latency of 95th Percentile: 0.034 seconds
latency of 99th Percentile: 0.026 seconds
```



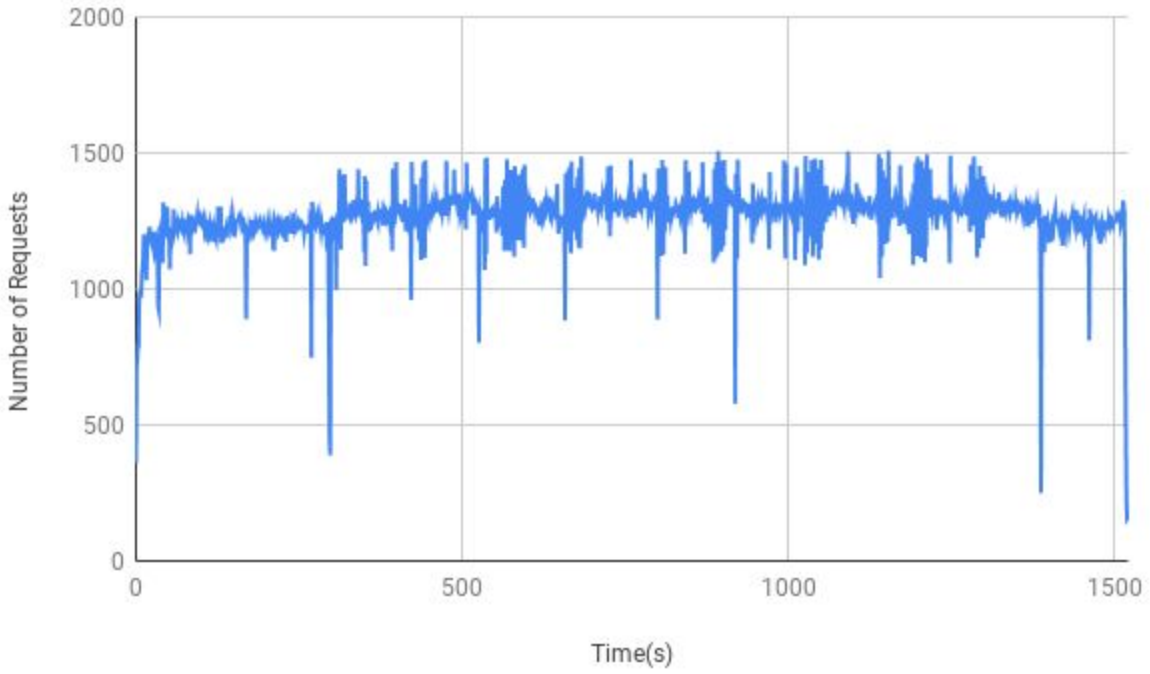

```
Client: Thread Count: 64, Iteration Count: 100
Client starting time: 1540796970362 milliseconds
Warmup phase:All threads running...
Warmup phase complete time: 40.566 seconds
Load phase:All threads running...
Load phase complete time: 89.838 seconds
Peak phase:All threads running...
Peak phase complete time: 390.421 seconds
Cooldown phase:All threads running...
Cooldown phase complete time: 71.792 seconds
Client end time: 1540797562988 milliseconds
========================================
Total requests sent: 481000
Total response received: 481000
Total wall time: 592.626 seconds
Total number of successful requests: 481000
Overall throughput across all phases: 811.6417757959558 seconds
latency of 95th Percentile: 0.068 seconds
latency of 99th Percentile: 0.054 seconds
```




```
Client: Thread Count: 128, Iteration Count: 100
Client starting time: 1540797894408 milliseconds
Warmup phase:All threads running...
Warmup phase complete time: 45.000 seconds
Load phase:All threads running...
Load phase complete time: 174.144 seconds
Peak phase:All threads running...
Peak phase complete time: 775.956 seconds
Cooldown phase:All threads running...
Cooldown phase complete time: 89.828 seconds
Client end time: 1540798979346 milliseconds
========================================
Total requests sent: 962000
Total response received: 962000
Total wall time: 1084.938 seconds
Total number of successful requests: 962000
Overall throughput across all phases: 886.6866220842655 seconds
latency of 95th Percentile: 0.156 seconds
latency of 99th Percentile: 0.154 seconds
```

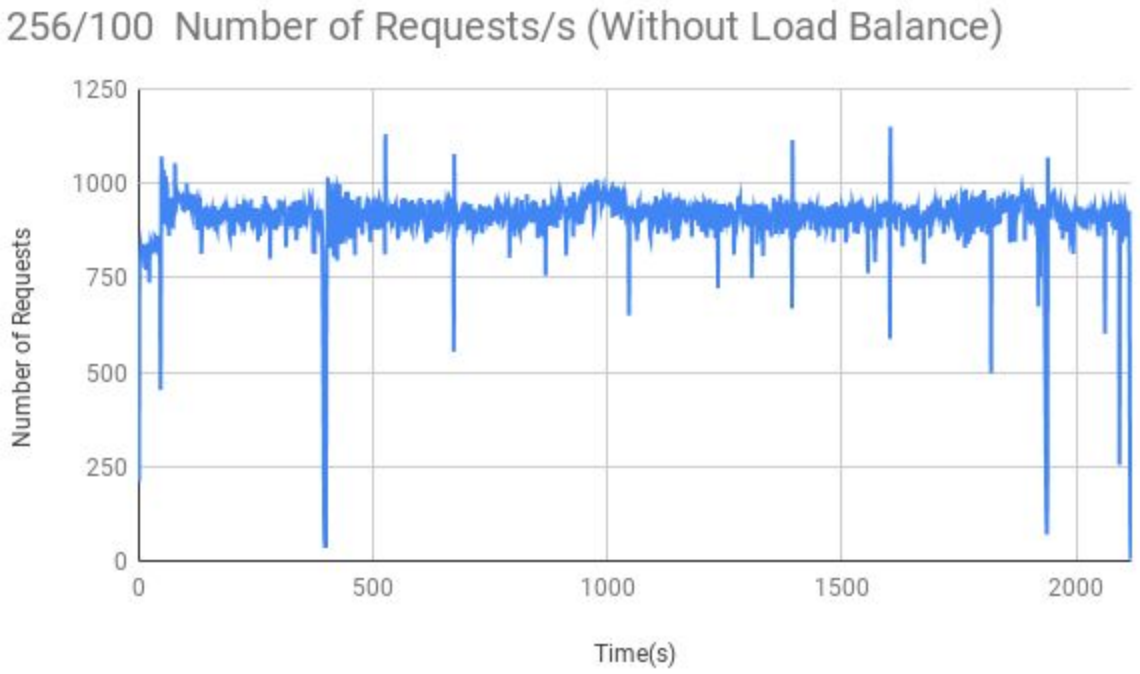
For 256/100 I had a glitch in my network thus loosing some of the responses all the time (javax.ws.rs.ProcessingException: java.net.NoRouteToHostException) . So I uploaded my client application to EC2 and tested it's performance. Below are the results.

Client executed from EC2:




```
[ec2-user@ip-172-31-21-51 ~]$ java -jar Client.jar 256 ec2-34-212-13-150.us-west-2.compute.amazonaws.com 1 1000000 100
Client: Thread Count: 256, Iteration Count: 100
Client starting time: 1540718050887 milliseconds
Warmup phase:All threads running...
Warmup phase complete time: 35.504 seconds
Load phase:All threads running...
Load phase complete time: 264.106 seconds
Peak phase:All threads running...
Peak phase complete time: 1098.668 seconds
Cooldown phase:All threads running...
Cooldown phase complete time: 134.430 seconds
Client end time: 1540719583599 milliseconds
========================================
Total requests sent: 1925500
Total response received: 1925500
Total wall time: 1532.712 seconds
Total number of successful requests: 1925500
Overall throughput across all phases: 1256.269902373985 seconds
latency of 95th Percentile: 0.206 seconds
latency of 99th Percentile: 0.168 seconds
```

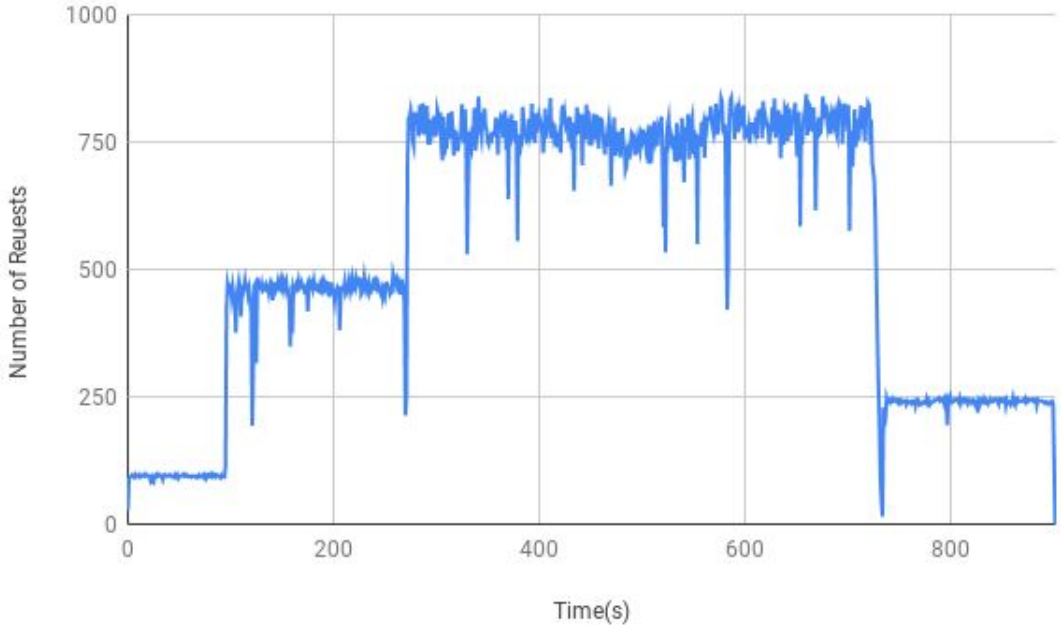
Client executed from local machine:




```
Client: Thread Count: 256, Iteration Count: 100
Client starting time: 1540799926582 milliseconds
Warmup phase:All threads running...
Warmup phase complete time: 46.735 seconds
Load phase:All threads running...
Load phase complete time: 354.175 seconds
Peak phase:All threads running...Peak phase complete time: 1543.479 seconds
Cooldown phase:All threads running...
Cooldown phase complete time: 177.969 seconds
Client end time: 1540802048951 milliseconds
========================================
Total requests sent: 1925500
Total response received: 1925497
Total wall time: 2122.369 seconds
Total number of successful requests: 1925497
Overall throughput across all phases: 907.2409622990231 seconds
latency of 95th Percentile: 0.297 seconds
latency of 99th Percentile: 0.263 seconds
```

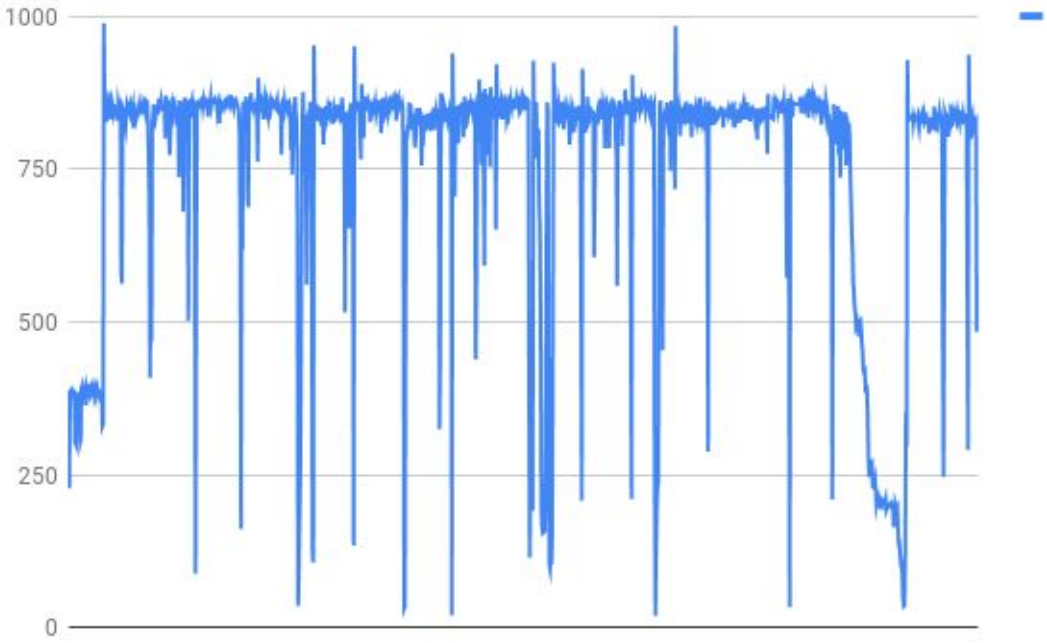
From the above graphs, we can see for higher number of threads the distinction between peak and load phase is not so clear. At this point I started exploring for options where in I can make the web server more efficient. So, I created a flask web server deployed using Gunicorn. It uses the SQLAlchemy library which has capabilities similar to any DataSource library available in Java. The advantage of using SQLAlchemy in such a scenario is that we can create the web server multithreaded. Gunicorn provides configuration to make the web server run with multiple worker threads and we can also specify the worker types as sync or gthread(threaded worker). Here the number of threaded worker can be fixed by the user -Ideally this can be set as

2*number of CPU cores*1. So digging deeper, I found out my EC2 instance has 2 cores and so I tested the web servers with configs of 4 worker-4 threads, 8 worker-8 threads and so on. To run the web server I used the following Gunicorn command - gunicorn app:app -w 4 -k gthread -b 0.0.0.0. Below as some of the outputs from the tests:



The above output is for 4-gthread worker threads for 64/100.

For 128/100 4-4 config : gunicorn app:app -w 4 -k gthread --threads 4 -b 0.0.0.0



```
Client: Thread Count: 128, Iteration Count: 100
Client starting time: 1540607034347 milliseconds
Warmup phase:All threads running...
Warmup phase complete time: 47.820 seconds
Load phase:All threads running...
Load phase complete time: 193.512 seconds
Peak phase:All threads running...
Peak phase complete time: 1049.598 seconds
Cooldown phase:All threads running...
Cooldown phase complete time: 99.624 seconds
Client end time: 1540608424914 milliseconds
========================================
Total requests sent: 962000
Total response received: 962000
Total wall time: 1390.567 seconds
Total number of successful requests: 962000
Overall throughput across all phases: 691.8041263132021 seconds
latency of 95th Percentile: 0.076 seconds
latency of 99th Percentile: 0.060 seconds
```

We can see the wall time is more than that of the Java web server but the latencies for 95th and 99th request is much less.

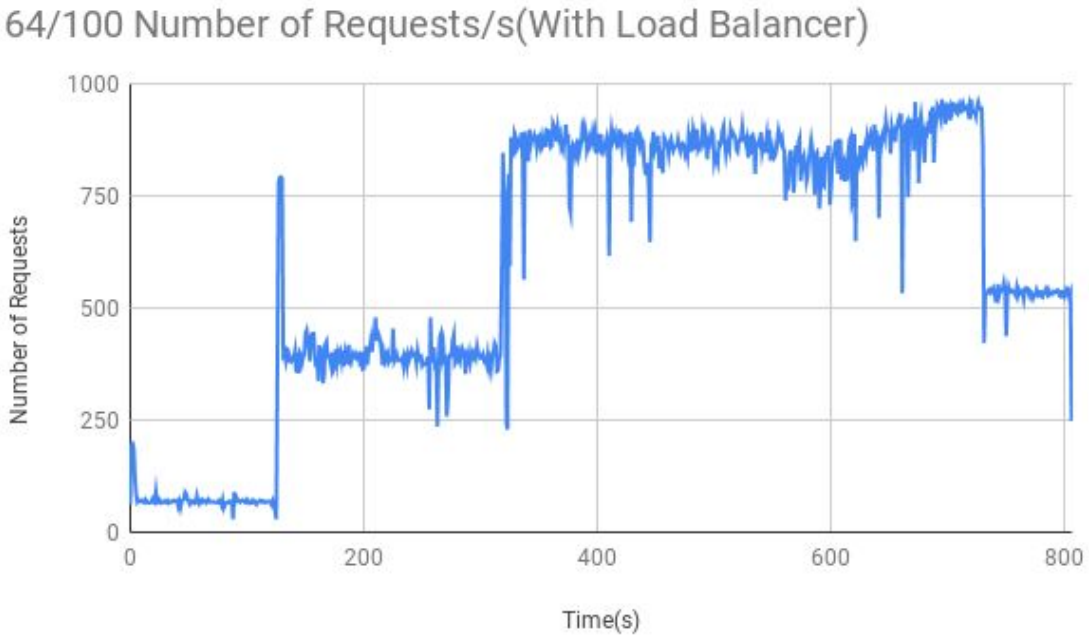
Step 4:

Load Balancer used - Classic Load Balancer with scaling policy of increasing group size by 1 when CPU utilization >=40% and decrease group size by 1 when CPU utilization is <=20%
Total Group is 1<=size<=3 instances

When I tested the performance of the 32/100 and 64/100 configuration of the Load Balancer, I used my home internet network which could be to blame for the slowness shown in the performance. So decided to test the load balancer step for the rest of the configurations by executing a runnable jar of the Client application on a t2.micro EC2 instance.

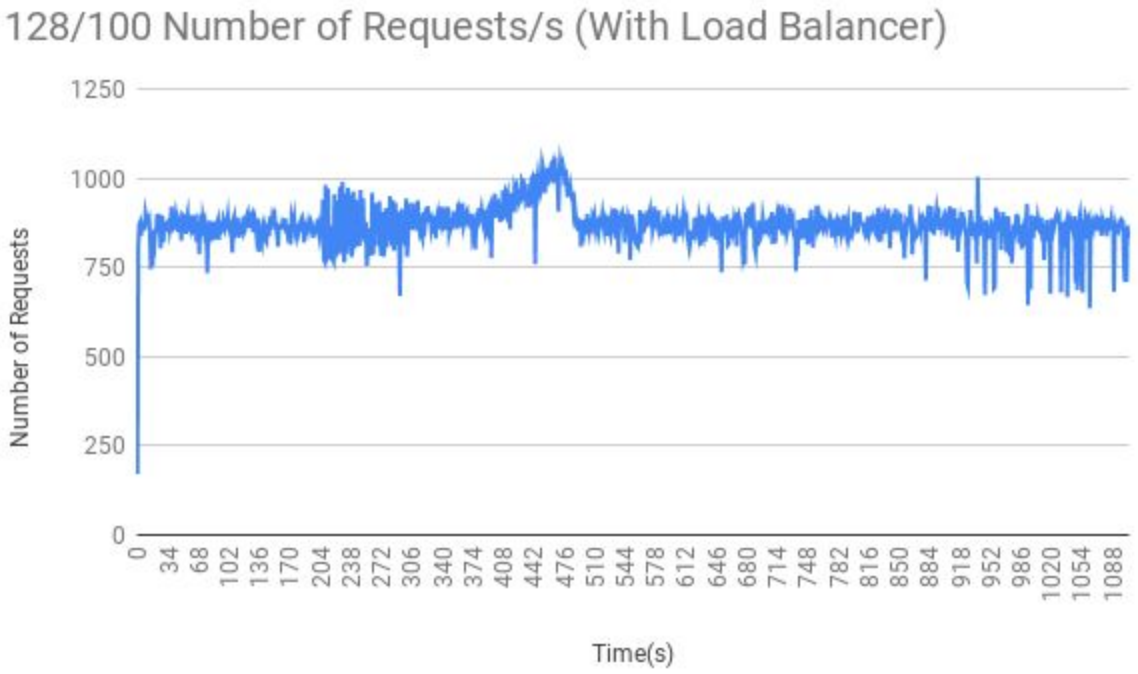
When testing for 32/100 and 64/100 the CPU utilization had a maximum value of 30.2% and 33% respectively so I did not see any added instances to the autoscale group.


```
Client: Thread Count: 32, Iteration Count: 100
Client starting time: 1540862748324 milliseconds
Warmup phase:All threads running...
Warmup phase complete time: 46.455 seconds
Load phase:All threads running...
Load phase complete time: 74.457 seconds
Peak phase:All threads running...
Peak phase complete time: 207.959 seconds
Cooldown phase:All threads running...
Cooldown phase complete time: 73.568 seconds
Client end time: 1540863150773 milliseconds
========================================
Total requests sent: 240500
Total response received: 240500
Total wall time: 402.449 seconds
Total number of successful requests: 240500
Overall throughput across all phases: 597.5912399737053 seconds
latency of 95th Percentile: 0.039 seconds
latency of 99th Percentile: 0.042 seconds
```



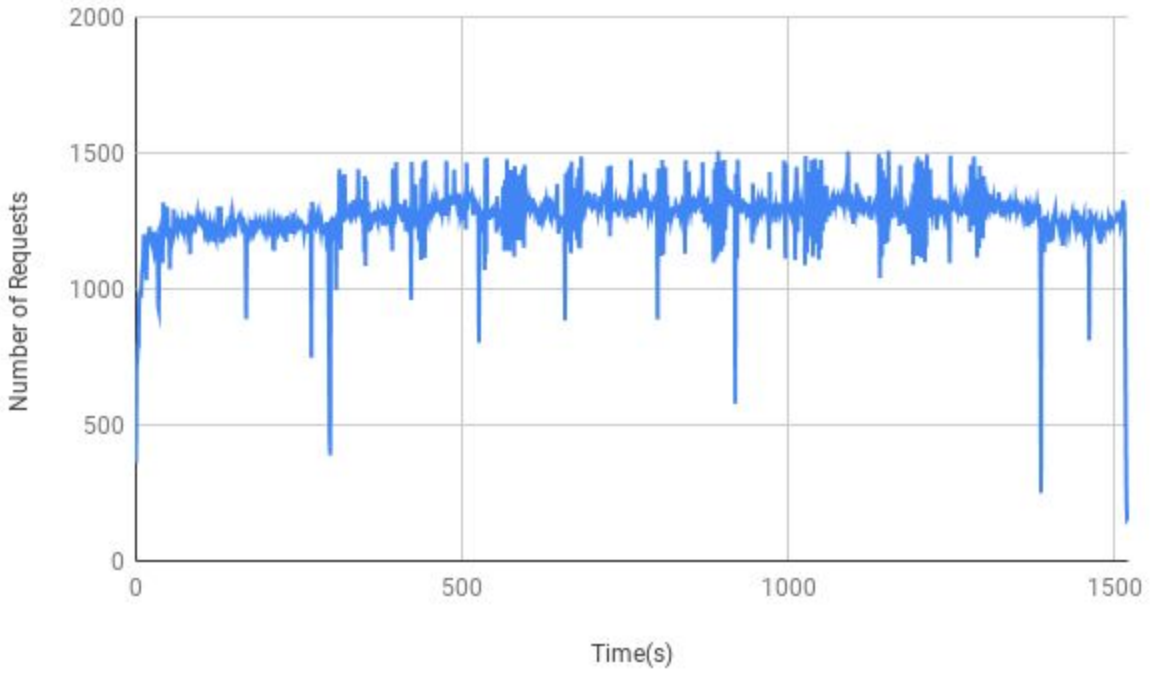

```
Client: Thread Count: 64, Iteration Count: 100
Client starting time: 1540864235294 milliseconds
Warmup phase:All threads running...
Warmup phase complete time: 128.253 seconds
Load phase:All threads running...
Load phase complete time: 198.069 seconds
Peak phase:All threads running...
Peak phase complete time: 408.968 seconds
Cooldown phase:All threads running...
Cooldown phase complete time: 75.769 seconds
Client end time: 1540865046360 milliseconds
=========================================
Total requests sent: 481000
Total response received: 481000
Total wall time: 811.066 seconds
Total number of successful requests: 481000
Overall throughput across all phases: 593.0466971265766 seconds
latency of 95th Percentile: 0.058 seconds
latency of 99th Percentile: 0.098 seconds
```

For 128/100 configuration the CPU utilization of the instance under auto scale group reached upto 55.83% thus scaling out the instance group from 1 to 2.




```
Client: Thread Count: 128, Iteration Count: 100
Client starting time: 1540865442940 milliseconds
Warmup phase:All threads running...
Warmup phase complete time: 22.101 seconds
Load phase:All threads running...
Load phase complete time: 185.642 seconds
Peak phase:All threads running...
Peak phase complete time: 807.604 seconds
Cooldown phase:All threads running...
Cooldown phase complete time: 94.988 seconds
Client end time: 1540866553287 milliseconds
========================================
Total requests sent: 962000
Total response received: 962000
Total wall time: 1110.347 seconds
Total number of successful requests: 962000
Overall throughput across all phases: 866.3957845914721 seconds
latency of 95th Percentile: 0.073 seconds
latency of 99th Percentile: 0.074 seconds
```

For 256/100 configuration the CPU utilization of the instance under auto scale group reached upto 55.55% thus auto scaling the group to 2 instances




```
[ec2-user@ip-172-31-21-51 ~]$ java -jar Client.jar 256 ec2-34-212-13-150.us-west-2.compute.amazonaws.com 1 1000000 100
Client: Thread Count: 256, Iteration Count: 100
Client starting time: 1540718050887 milliseconds
Warmup phase:All threads running...
Warmup phase complete time: 35.504 seconds
Load phase:All threads running...
Load phase complete time: 264.106 seconds
Peak phase:All threads running...
Peak phase complete time: 1098.668 seconds
Cooldown phase:All threads running...
Cooldown phase complete time: 134.430 seconds
Client end time: 1540719583599 milliseconds
========================================
Total requests sent: 1925500
Total response received: 1925500
Total wall time: 1532.712 seconds
Total number of successful requests: 1925500
Overall throughput across all phases: 1256.269902373985 seconds
latency of 95th Percentile: 0.206 seconds
latency of 99th Percentile: 0.168 seconds
```

From the above tests, I assume that choosing CPU utilization as the scaling policy though is appropriate for auto scaling it will vary on a number of factors: how efficient is the client application, how are the other resources being utilized like memory and network bandwidth. So finding the exact configuration by loads of testing to know when the server gets overloaded is the key to create the scaling policy.

Another point of interest is that the EC2 instance which sends the concurrent requests for the Client applications clocks at 100% utilization at when there are more than 256 threads so this means the instance actually starts underperforming because of over usage. Thus lesser HTTP requests are received by the server making the server side auto scale group only show a CPU utilization of 33-35%.