CUSTOS TESTING

Endpoints references :
https://cwiki.apache.org/confluence/display/CUSTOS/Use+Custos+REST+Endpoints

Load testing and stress testing

To analyze the performance of the CUSTOS client application. We have performed load and stress testing. We have used JMeter for our experiments.

Load Testing: Load Testing is a non-functional software testing process in which the performance of software applications is tested under a specific expected load. It determines how the software application behaves while being accessed by multiple users simultaneously. The goal of Load Testing is to improve performance bottlenecks and to ensure stability and smooth functioning of software applications before deployment.

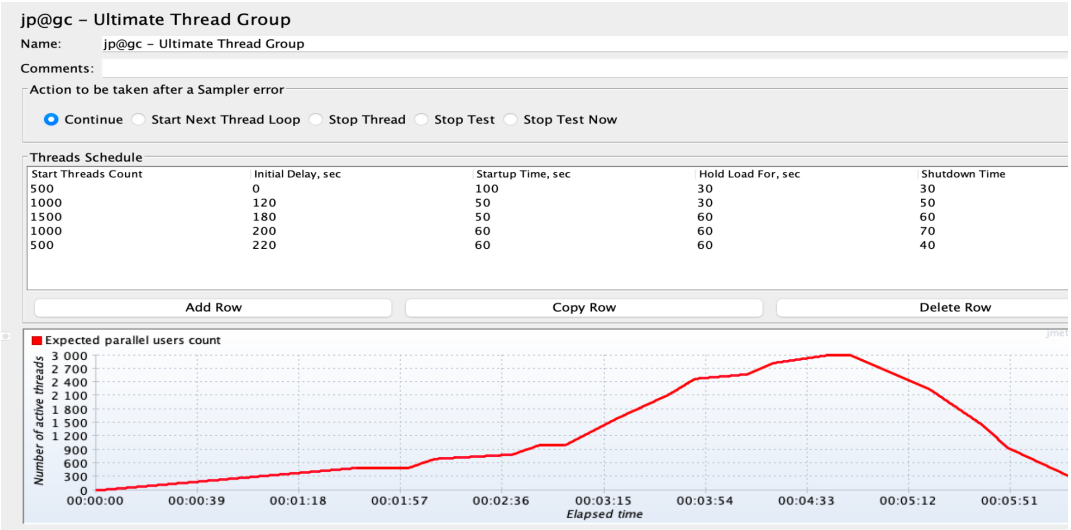
Stress Testing: Stress Testing is a type of software testing that verifies the stability & reliability of software applications. The goal of Stress testing is to measure software on its robustness and error handling capabilities under extremely heavy load conditions and ensure that software doesn't crash under crunch situations. It even tests beyond normal operating points and evaluates how the software works under extreme conditions.

Test Plan:

For executing these tests we have used "Ultimate Thread Group" extension in JMeter. For each of our endpoints we used the following test plan:

1. Add 500 users every second.
2. Add 1000 users, 120 seconds after the test has started.
3. Add 1500 users, 180 seconds after the test has started.
4. Add 1000 users, 200 seconds after the test has started.
5. Add 500 users, 220 seconds after the test has started.

The JMeter's Ultimate thread group extension's settings can be viewed below:



We have used this test on our endpoints:

- URL: custos.scigap.org/31499

- Endpoints tested:
    - ★ `/register-user`
    - ★ `/user-exist`
    - ★ `/user-access`
    - ★ `/get-user`

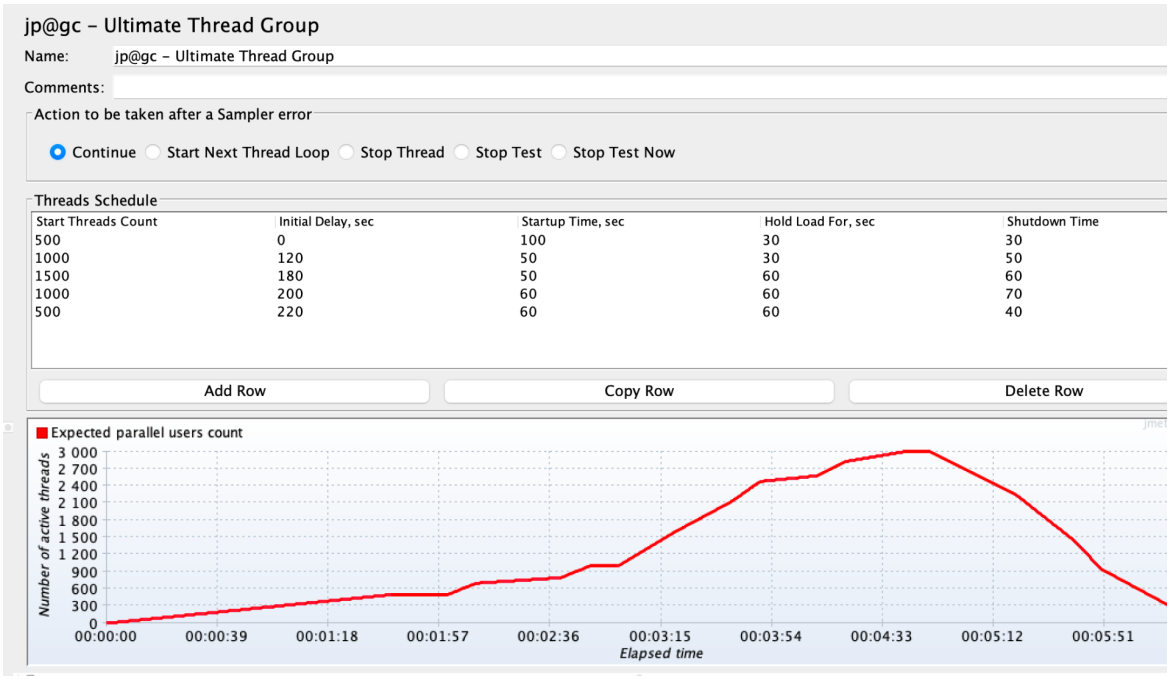
● Test results :

We have run our load testing on the following endpoints. The plots for all iterations of different numbers of users are shown below.

1) User-exist

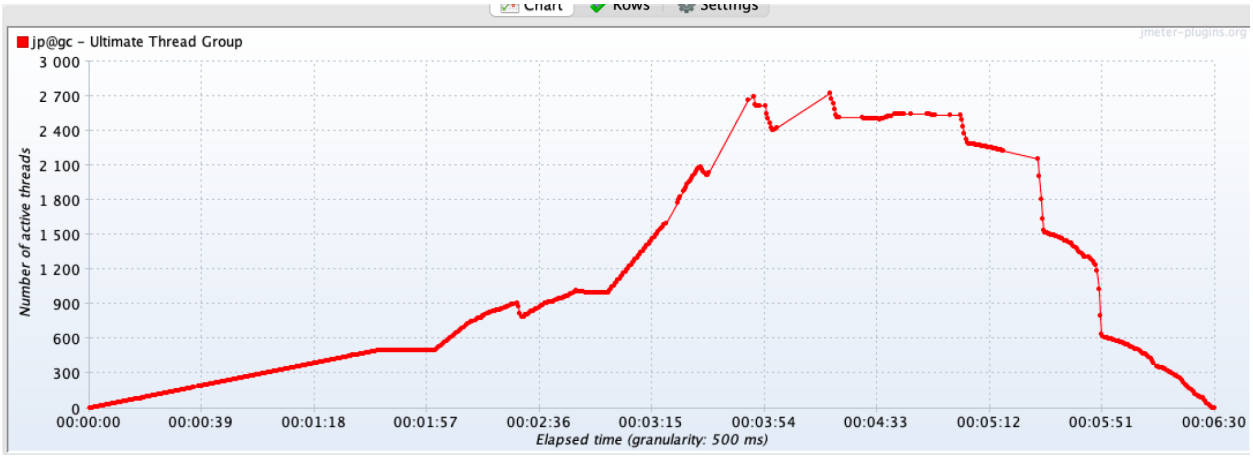
Stress testing with load shown in the graph below:

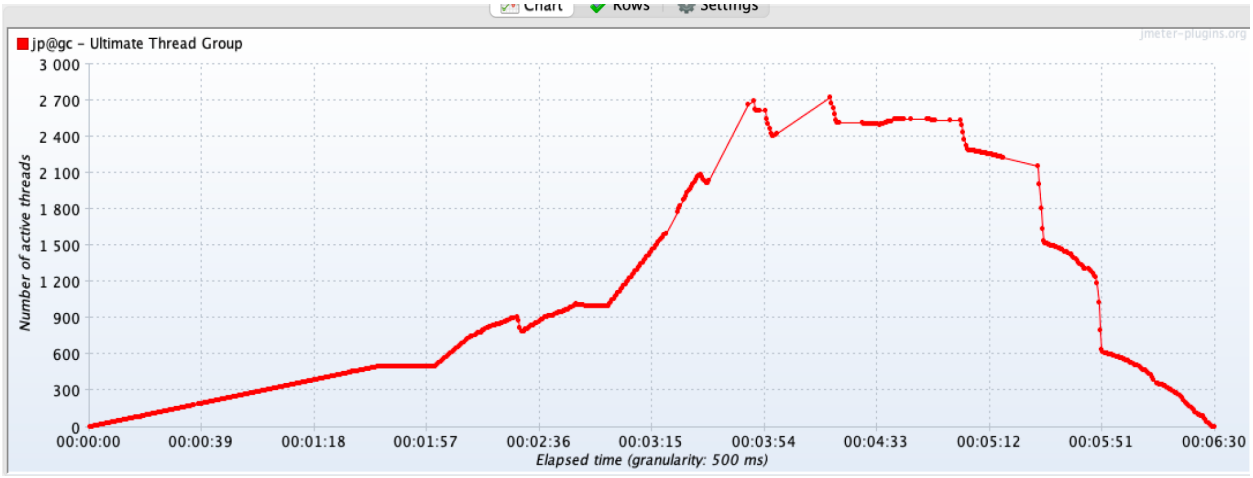
We started our testing experiment with 500 threads and increased it gradually upto 4500 threads for user-existing endpoint.



1. <u>Active thread over time :</u> shows how many active threads are there in each thread group during test run.

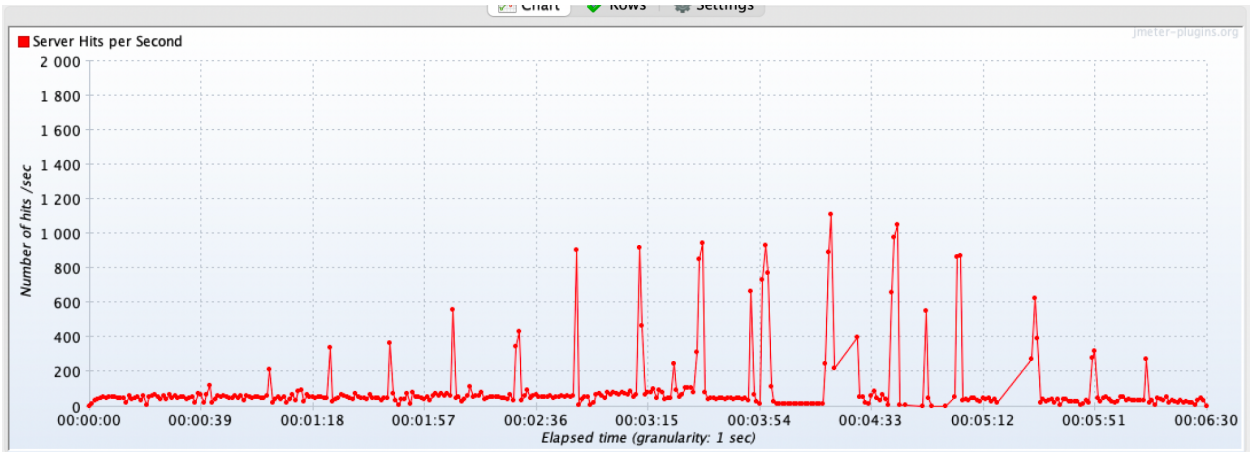
   For testing the number of active threads vs the time elapsed, we started the test and slowly increased the threads upto 3000.
   We noticed that the number of active threads increases slowly upto 500 threads after which it increases by a greater margin upto 2700 threads. That is the peak point. AFter the two peaks, the number of active threads line curve seems to decline with elapsing time.





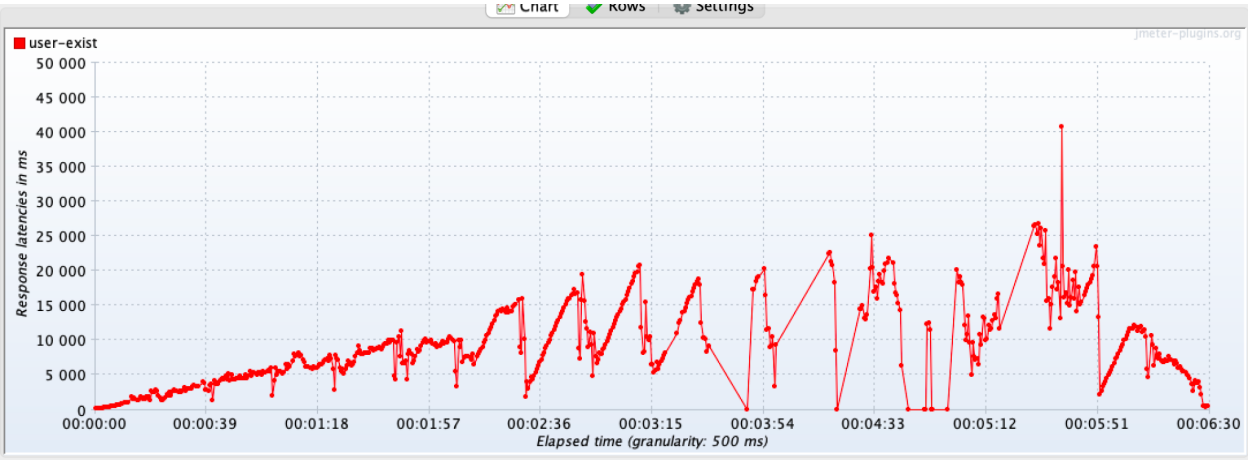
2. Hits per second: refers to the number of HTTP requests sent by the user(s) to the Web server in a second.

   In the graph below we can observe that just like the previous graph, the elapsed time for the number of hits increases gradually and spikes up at around 900 hits/sec. There are several spikes upto 1100 hits/sec after which the spikes decrease gradually.



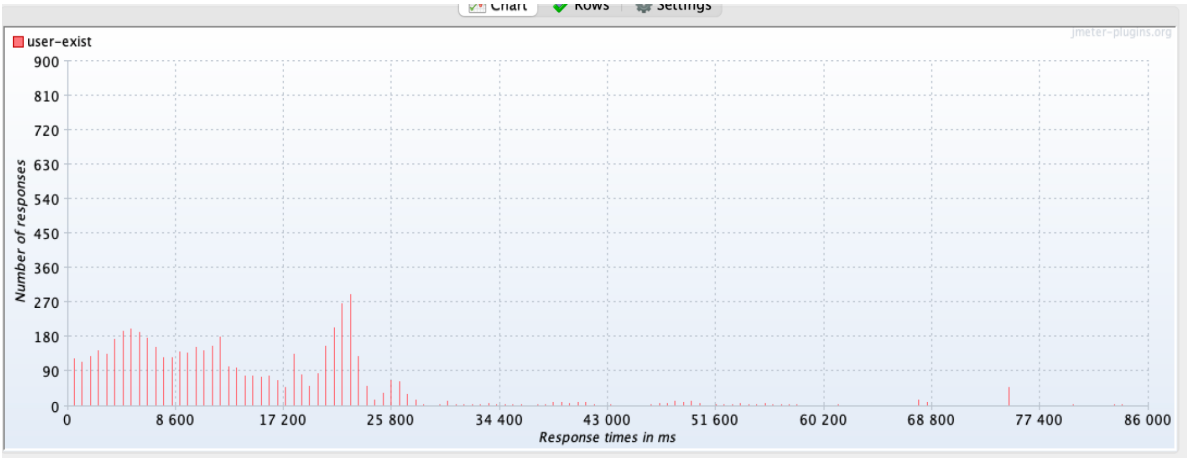
3. Response latencies over time (interesting): provides the information about the time the tested server spends for the processing of the requests (samples).

   A latency is the duration between the end of the request and the beginning of the server response. The graph below shows that the latencies increase over time gradually but also has several drops with increasing elapsed time. The highest latency spike of 40000 ms occurs at 5:12 seconds.



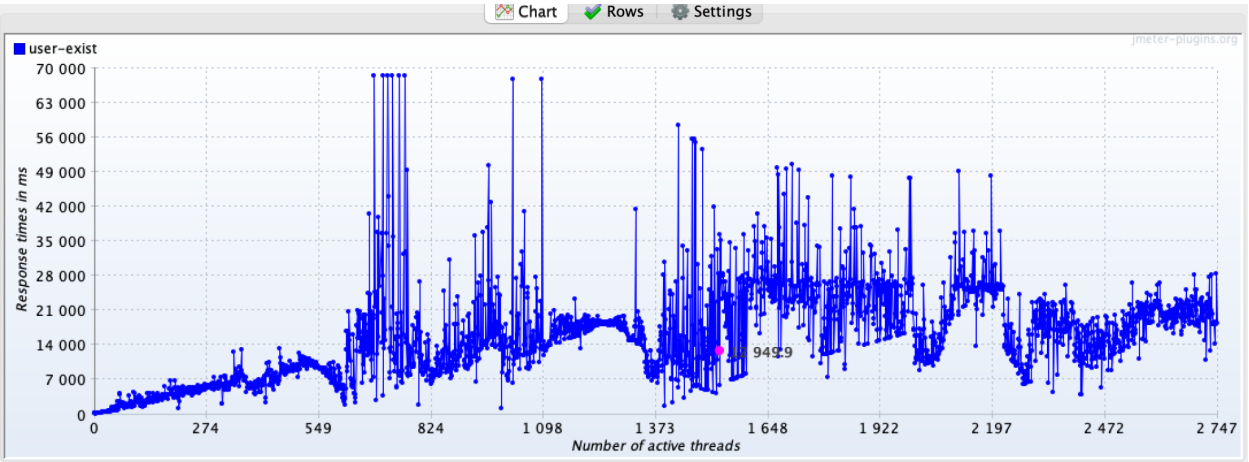
4. Response time distribution : The response times are grouped by their durations. Each group is an interval, for example from 0 to t ms, from t ms to 2t ms, where t is the configured step under the settings tab of the listener.

   In the graph below we can see that the response times occur around 25000 ms and keep on increasing and decreasing. The mode response time is 17800 ms (270 responses took this length of time). Most of the users experience page load times between 17200 and 25800 ms. The average response time is the sum of all response times divided by the number of responses.



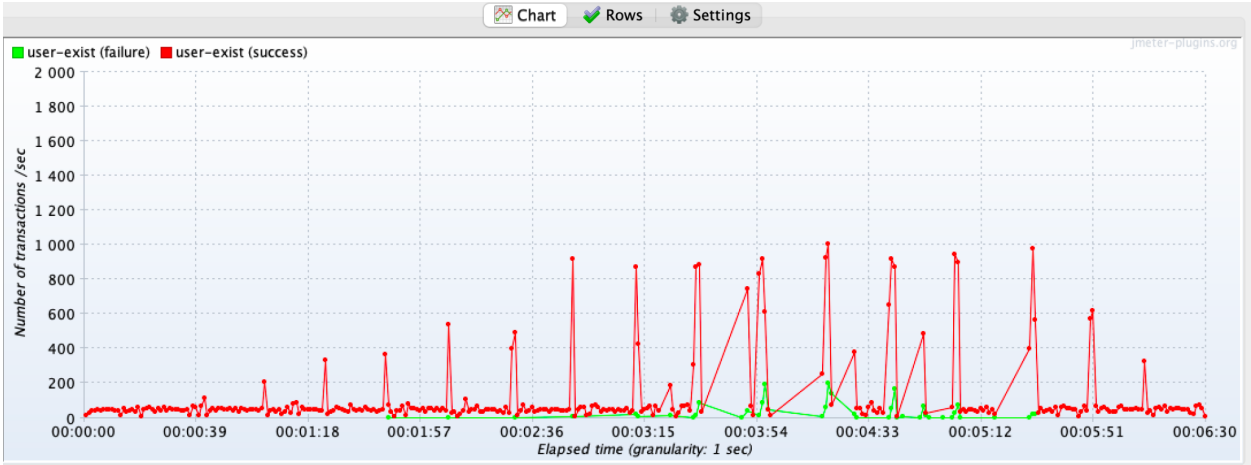
5. Response time vs Threads: This graph shows how Response Time changes with the amount of parallel threads. The server takes longer to respond when a lot of users requests it simultaneously.

The response time increases with the number of active thread requests generally but there are several thread counts which have high response times up to roughly 70000 ms. After about 2200 simultaneous active threads, the response time does not increase much.



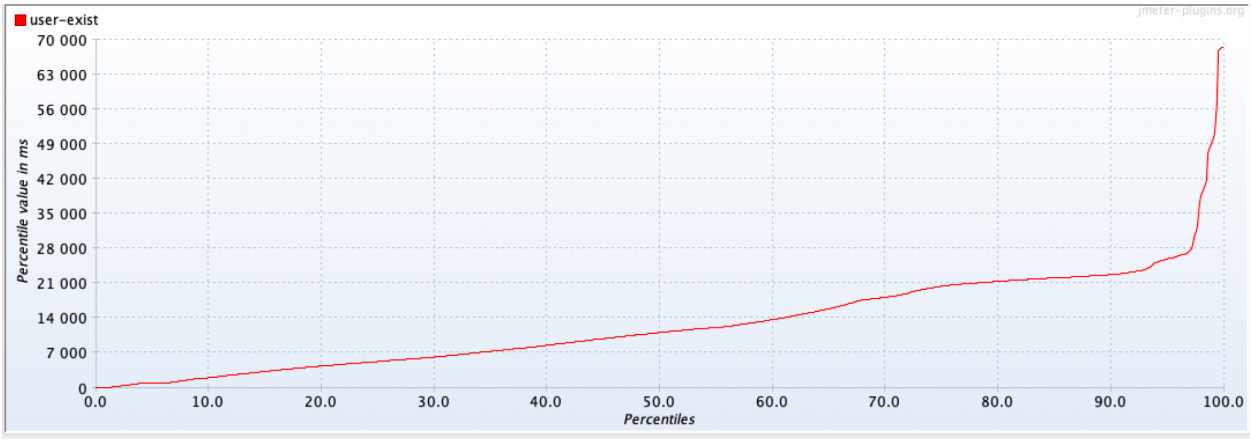
6. Transaction per second: It counts for each second the number of finished transactions.

The graph shows the number of successful and unsuccessful (failure) transactions for user-exist route. Failure rates are shown in green and success rates are shown in red. The success rates for user-exist are greater as compared to failure rates. With time elapsing, the number of transactions per second increases upto a point of 1000 transactions at roughly 5:30 seconds after which it reduces.

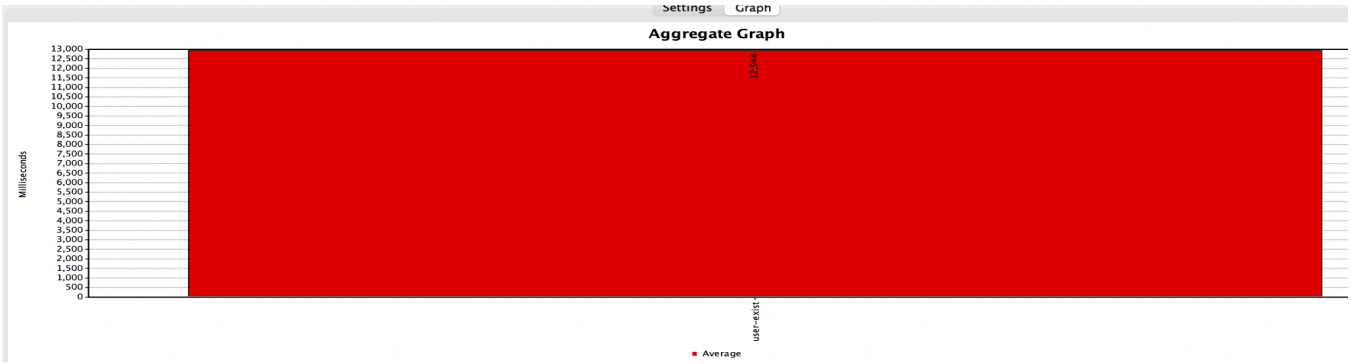


7. Response time percentile: displays the percentiles with respect to response time in milliseconds.

The percentile increases gradually in a slope but at a percentile value of about 97%, the response time increases in an exponential fashion from 28000 ms to about 69000 ms at the 100th percentile.



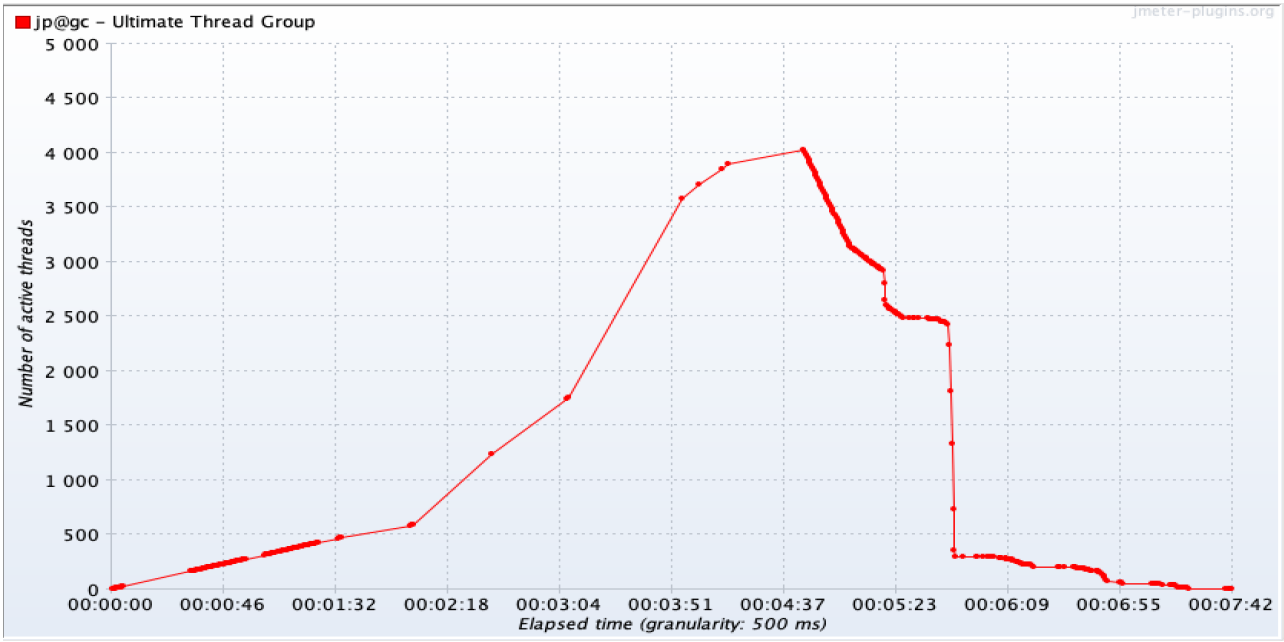
8. Aggregate Graph: plots a bar chart for the response time metrics for each sample. The response time increases to 13000 milliseconds for user-exist.



2) Register

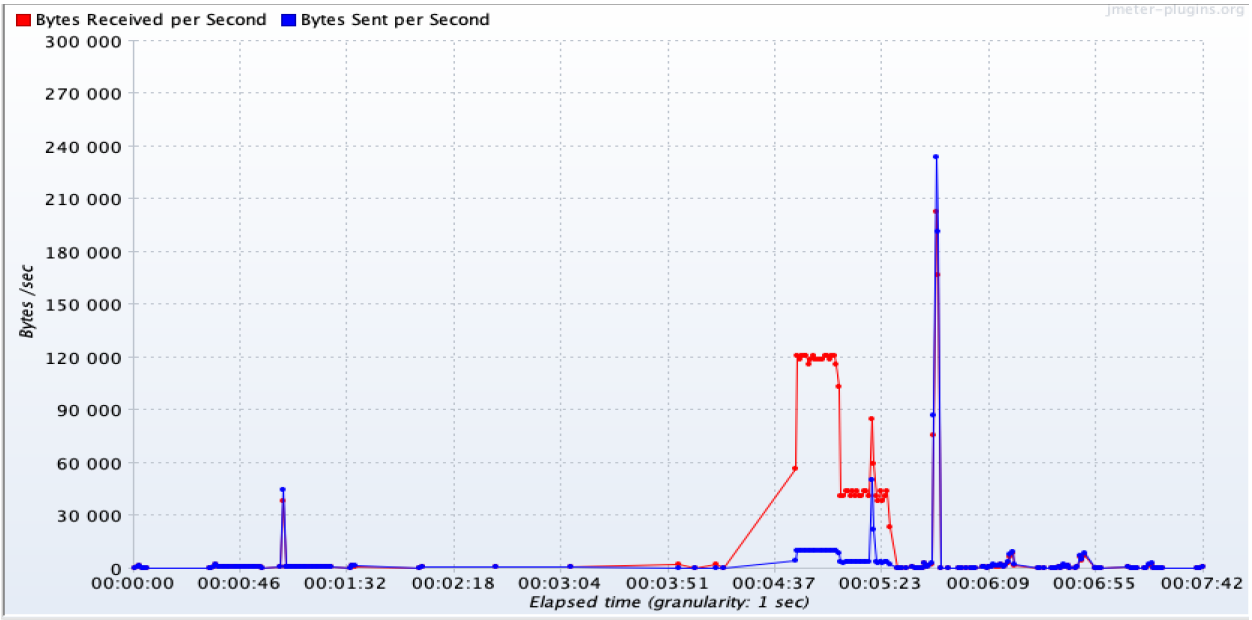
1. Active Threads over Time : shows how many active threads are there in each thread group during test run. Jmeter counts the elapsed time from just before sending the request to just after the last response has been received.

   For testing the number of active threads vs the time elapsed, we started the test with 500 threads or users and slowly increased the threads upto 5000.
   We noticed that with time elapsing, the number of active threads increases gradually upto a peak point of 4000 active threads after which it decreases steeply.



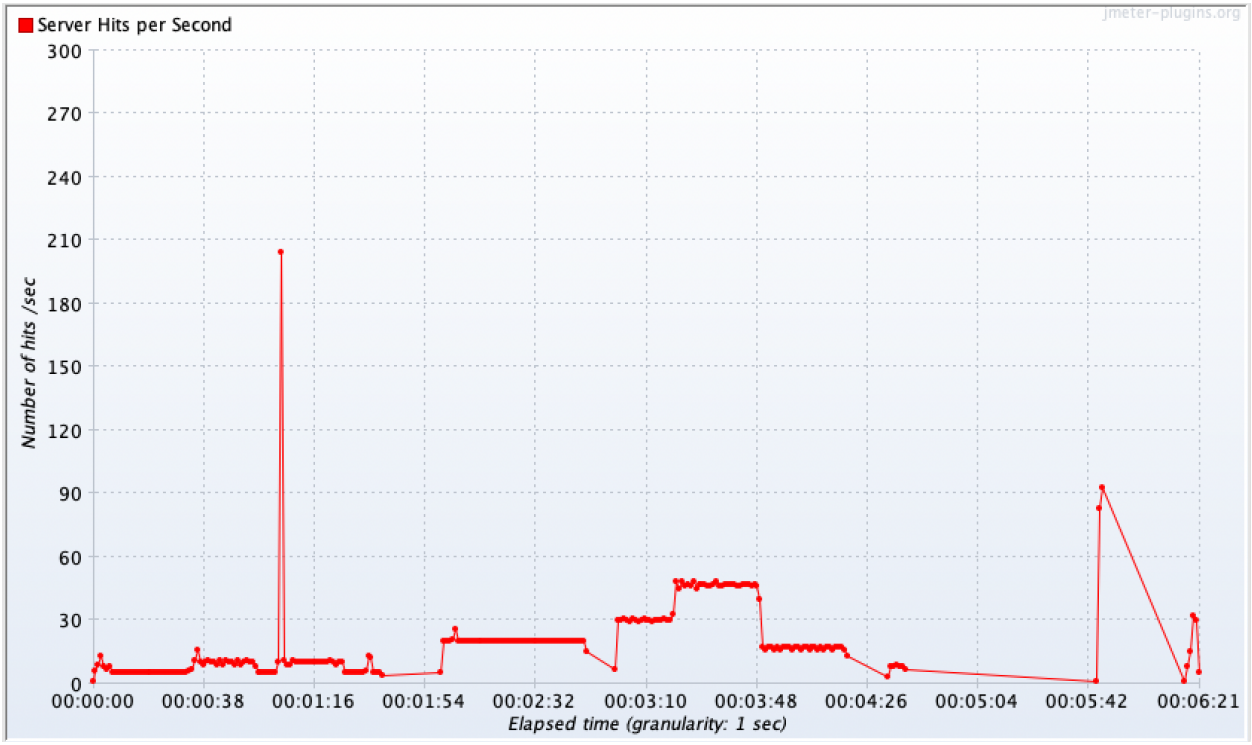
2. Bytes throughput over time: shows the number of bytes that were sent and received, per time unit, throughout the test.

   In the graph below we can see that the bytes are sent and received continuously but there is a spike at time 4-5 seconds where bytes received per second are more than bytes sent. Another spike occurs and it has a large number of bytes sent and received per second.



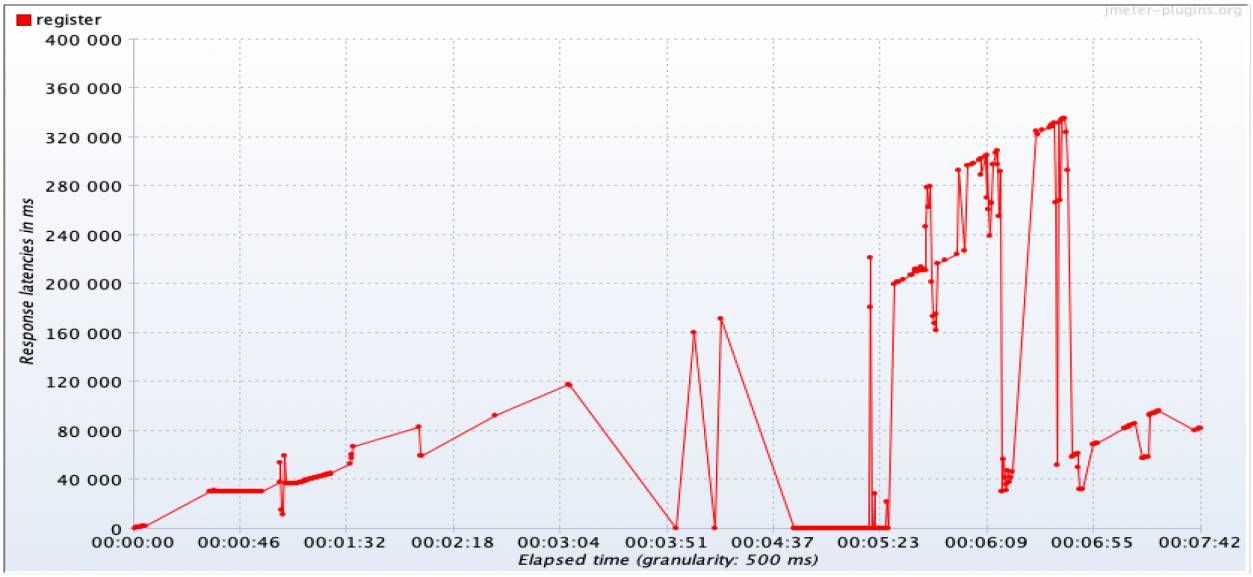
3. Hits per second: refers to the number of HTTP requests sent by the user(s) to the Web server in a second.

In the graph below we can observe the number of hits spikes up at for 205 hits/sec. The server hits per second do not have a pattern in general and form plateaus at certain elapsed time periods. There are no major spikes which are seen.



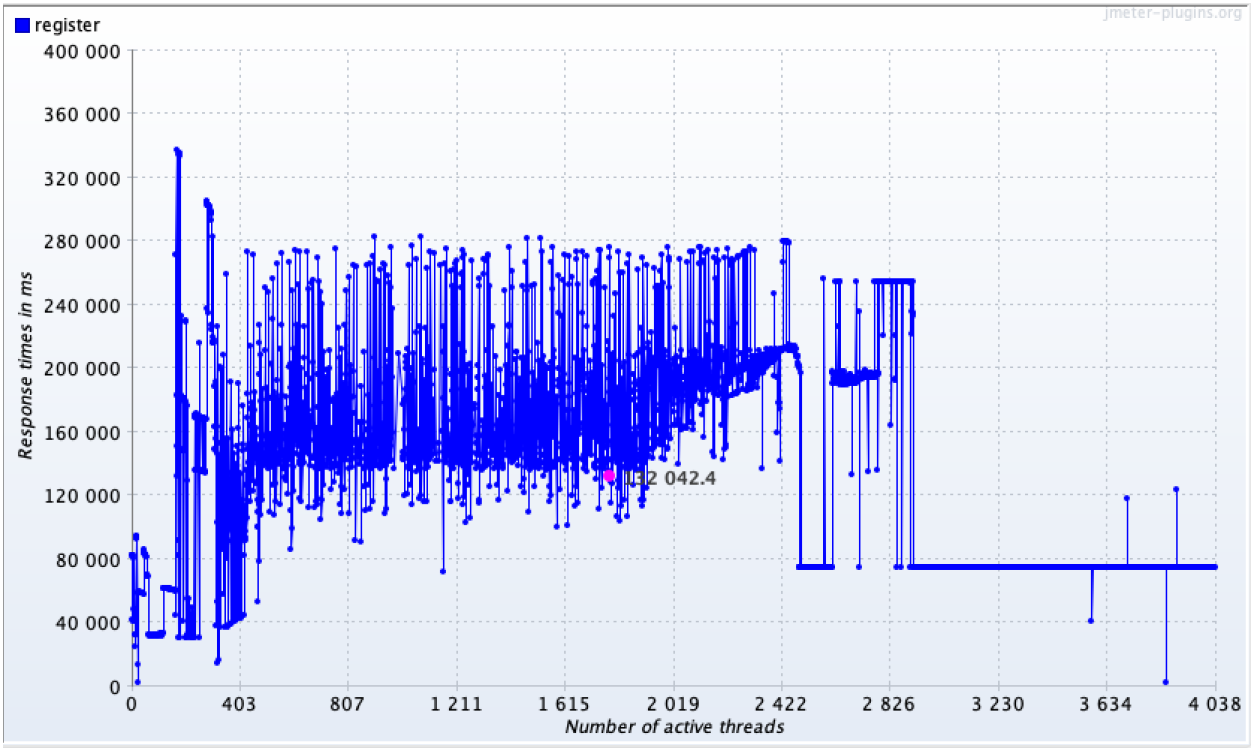
4. Response latencies over time: provides the information about the time the tested server spends for the processing of the requests (samples).

A latency is the duration between the end of the request and the beginning of the server response. The graph below shows that the latencies increase over time gradually but also has several drops with increasing elapsed time. The highest latency spike of 333000 ms occurs at 6:45 roughly seconds. From the spikes we can infer that the tested server requires more time for processing those requests.



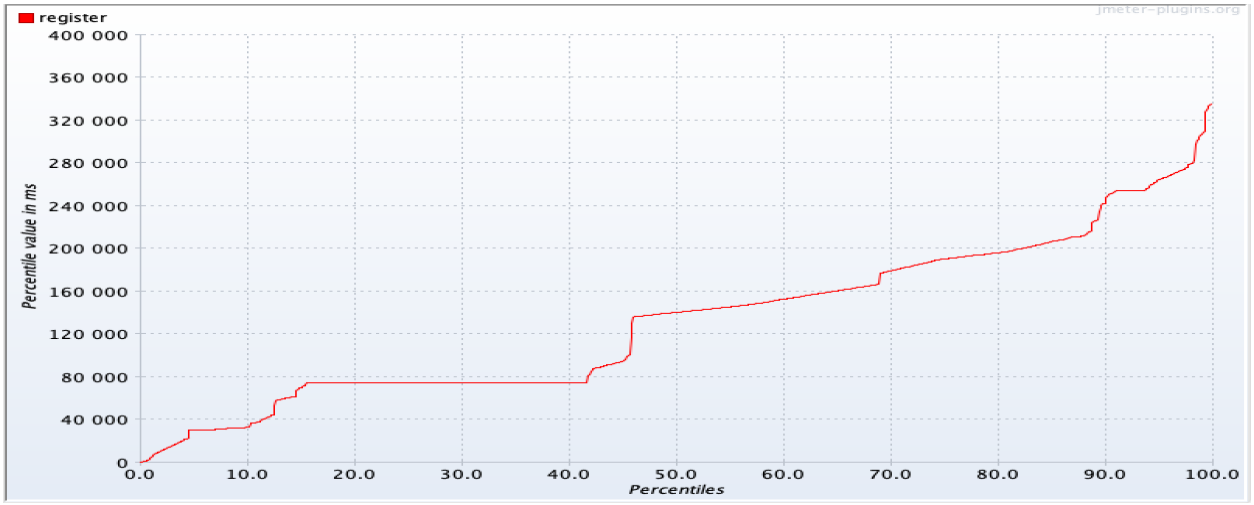
5. Response time vs Threads: This graph shows how Response Time changes with the amount of parallel threads. The server takes longer to respond when a lot of users requests it simultaneously.

   The response times remain in a uniform range in between values 120000 - 280000 ms for increasing number of active threads after which the response times decrease and do not seem to vary after 3000 active threads. The pink dot shows the optimum values.

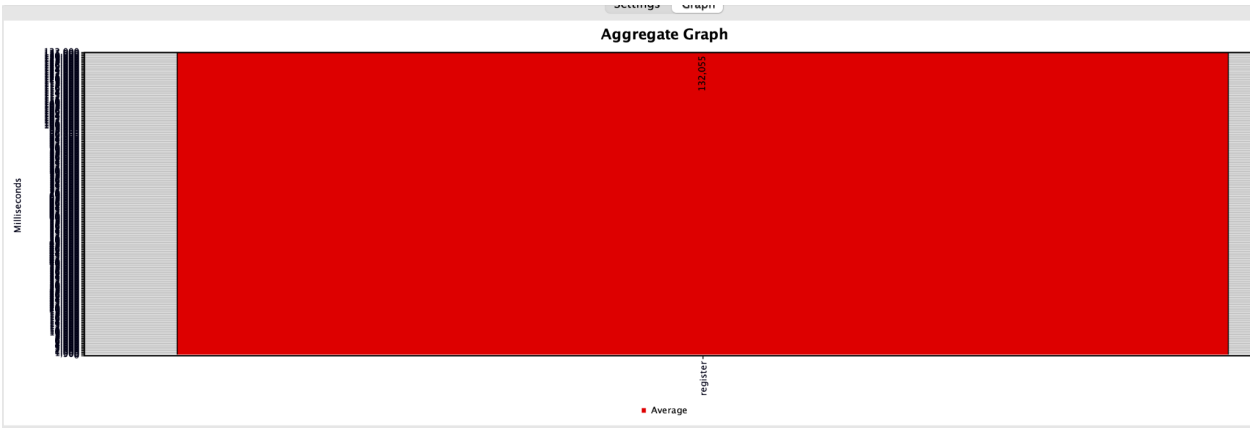


6. Response time percentile : displays the percentiles with respect to response time in milliseconds.

The percentile increases gradually with values forming a plateau between percentile 15.0 - 40.0. The 100th percentile has a percentile value of about 320000 seconds.



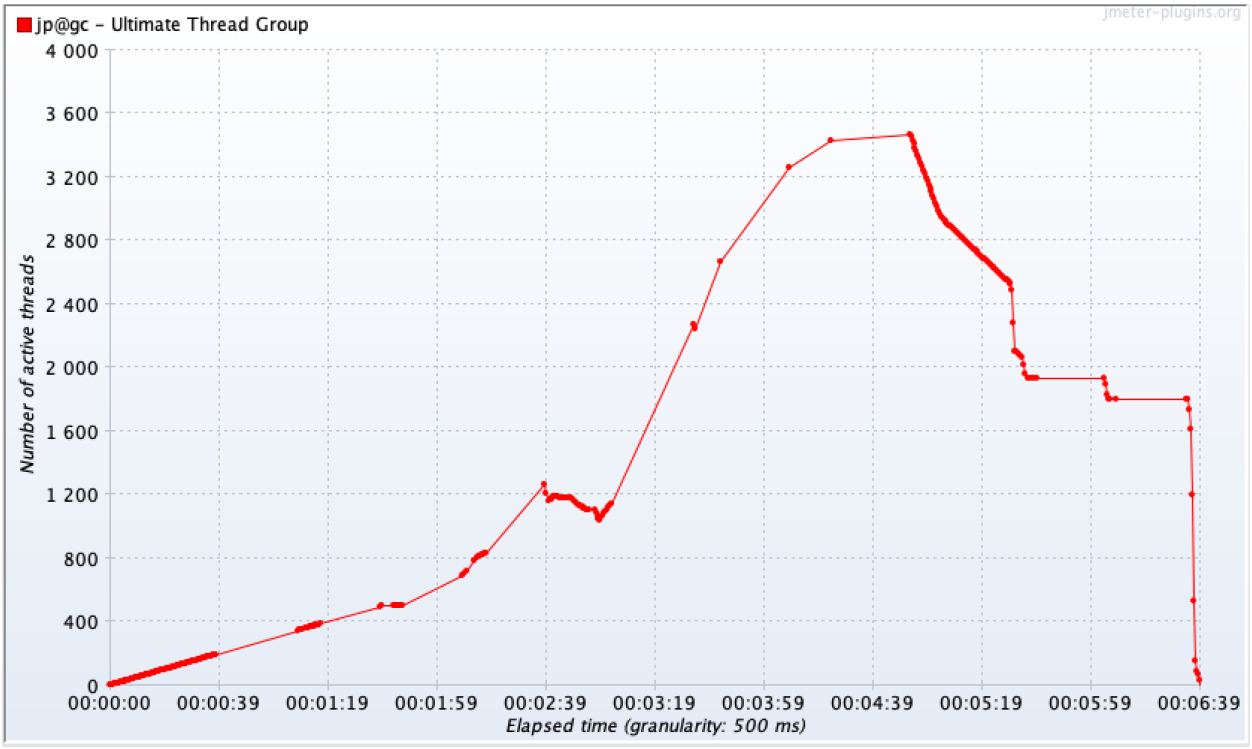
7. Aggregate Graph: plots a bar chart for the response time metrics for each sample. The response time increases a lot for /register which is why it is not visible in the graph.



3) User Access

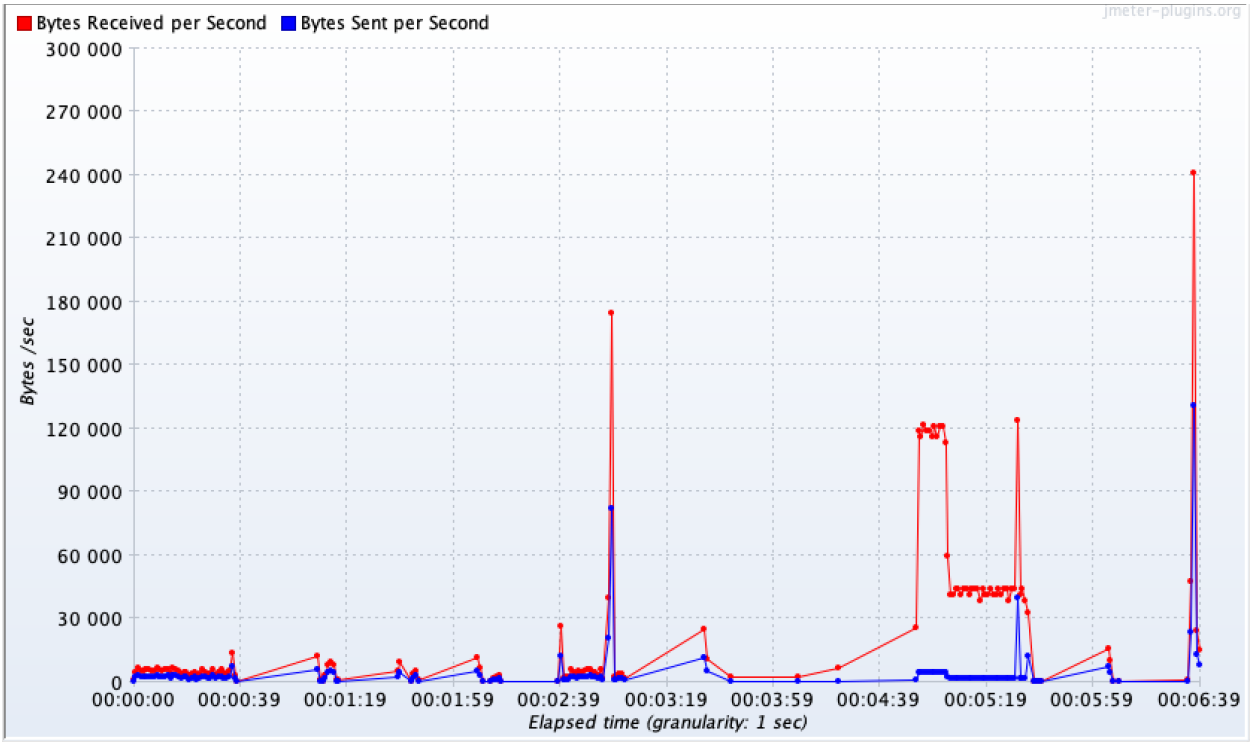
1. Active Threads over Time: shows how many active threads are there in each thread group during test run. Jmeter counts the elapsed time from just before sending the request to just after the last response has been received.

   We noticed that with time elapsing, the number of active threads increases gradually upto a peak point of 3400 active threads after which it decreases steeply. The peak point forms a small flat region where the time increases but the number of threads remains a constant.



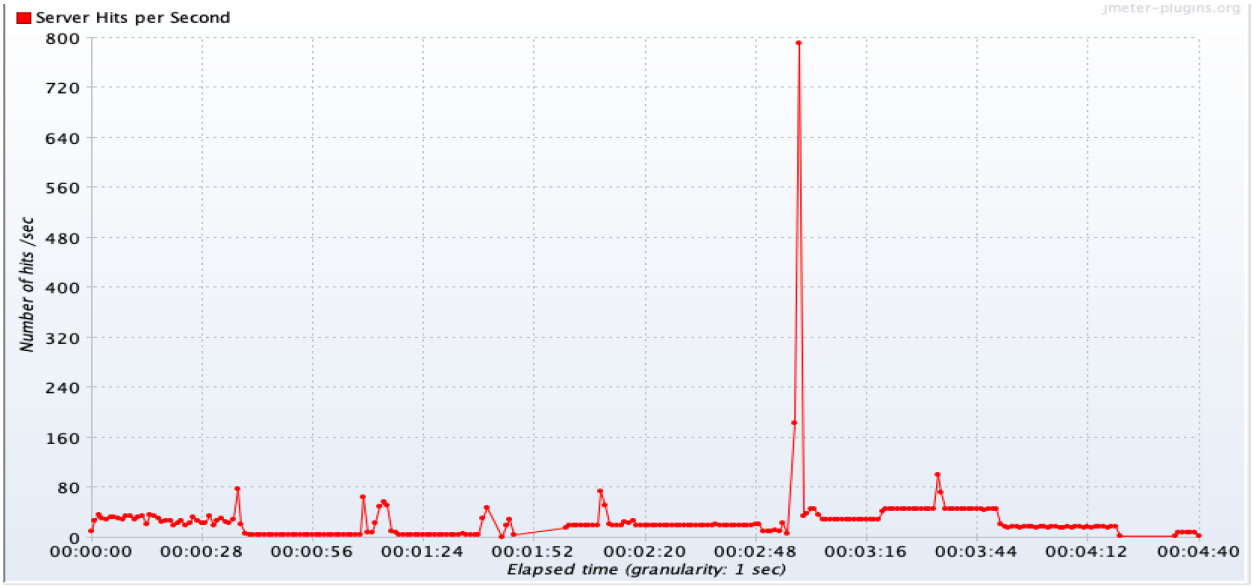
2. Bytes throughput over time: shows the number of bytes that were sent and received, per time unit, throughout the test.

   In the graph below we can see that the bytes are sent and received continuously with the similar spike as shown in the Bytes throughput over time graph of the previous route. Overall, the bytes received are more than the bytes that are being sent per second. There are 2 other major spikes in the graph at elapsed time of 3 seconds and 6.39 seconds.



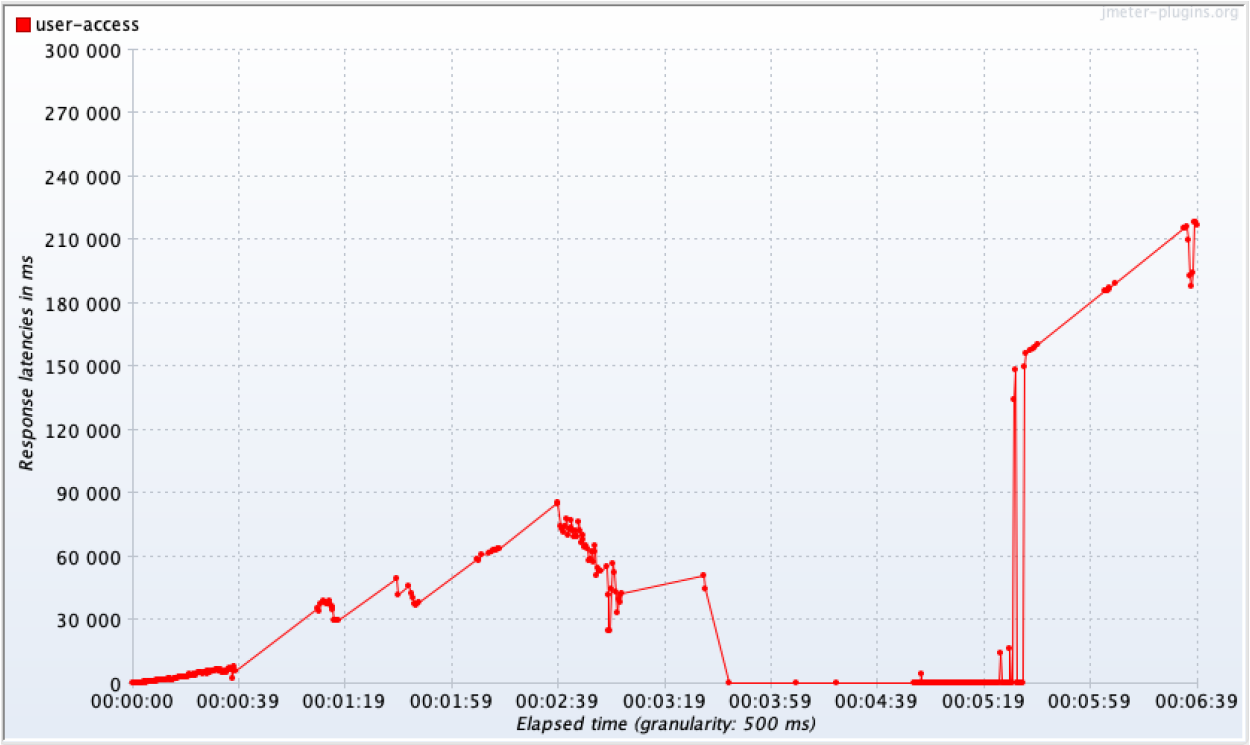
3. Hits per second: refers to the number of HTTP requests sent by the user(s) to the Web server in a second.

In the graph below we can observe that the number of hits spikes up at 800 hits/sec at about 3 seconds of the elapsed time. The server hits per second do not have a pattern in general and form plateaus at certain elapsed time periods. There are no major spikes which are seen apart from the one mentioned.



4. Response latencies over time: provides the information about the time the tested server spends for the processing of the requests (samples).

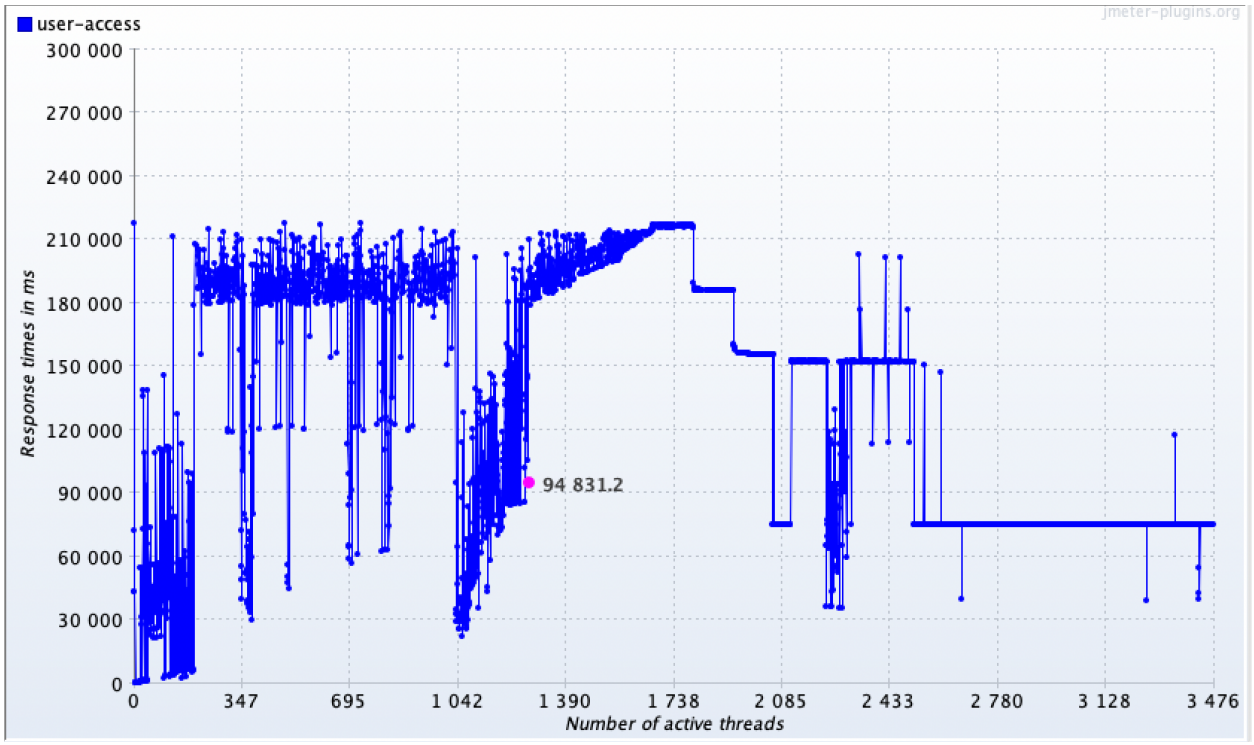
A latency is the duration between the end of the request and the beginning of the server response. The graph below shows that the latencies increase over time gradually but also has a few drops with increasing elapsed time. From 3.30s to 5.35s roughly, the response latencies are 0 The highest latency spike of 210000 ms occurs at 6:39 roughly seconds. From the spikes we can infer that the tested server requires more time for processing those requests.



5. Response time vs threads: This graph shows how Response Time changes with the amount of parallel threads. The server takes longer to respond when a lot of users request it simultaneously.

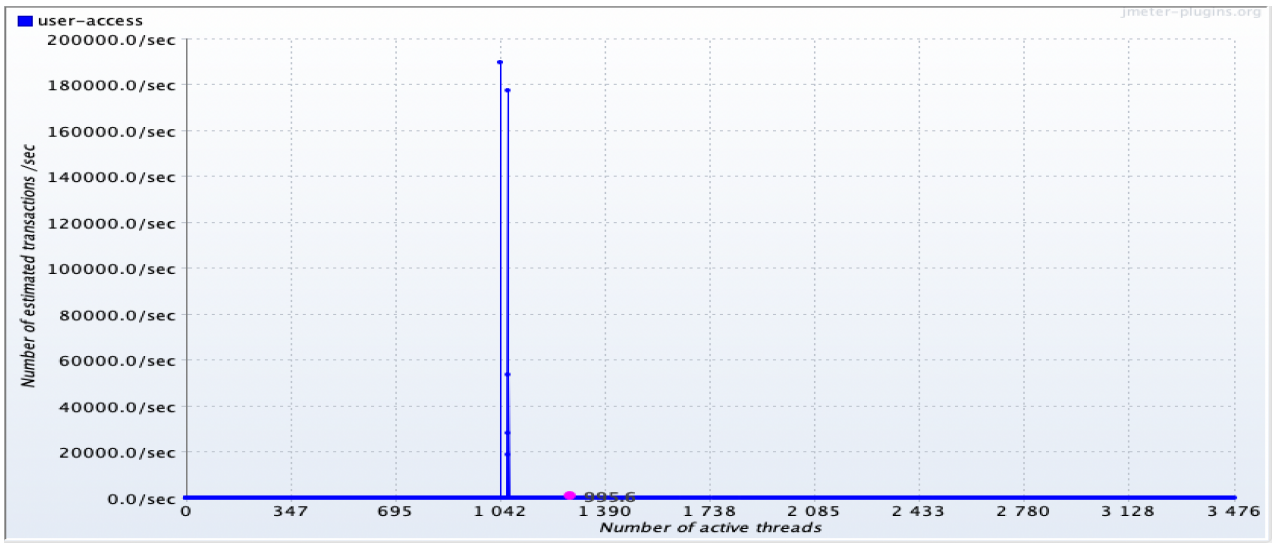
   With an increasing number of active threads, the response time is generally high in the start with values of 120000 to 150000 ms after which it increases but again becomes uniform. Thereafter, there are highs and lows in the response times with increasing number of active threads but at 2.5 seconds to 3.5 seconds, the response times become uniform.

.



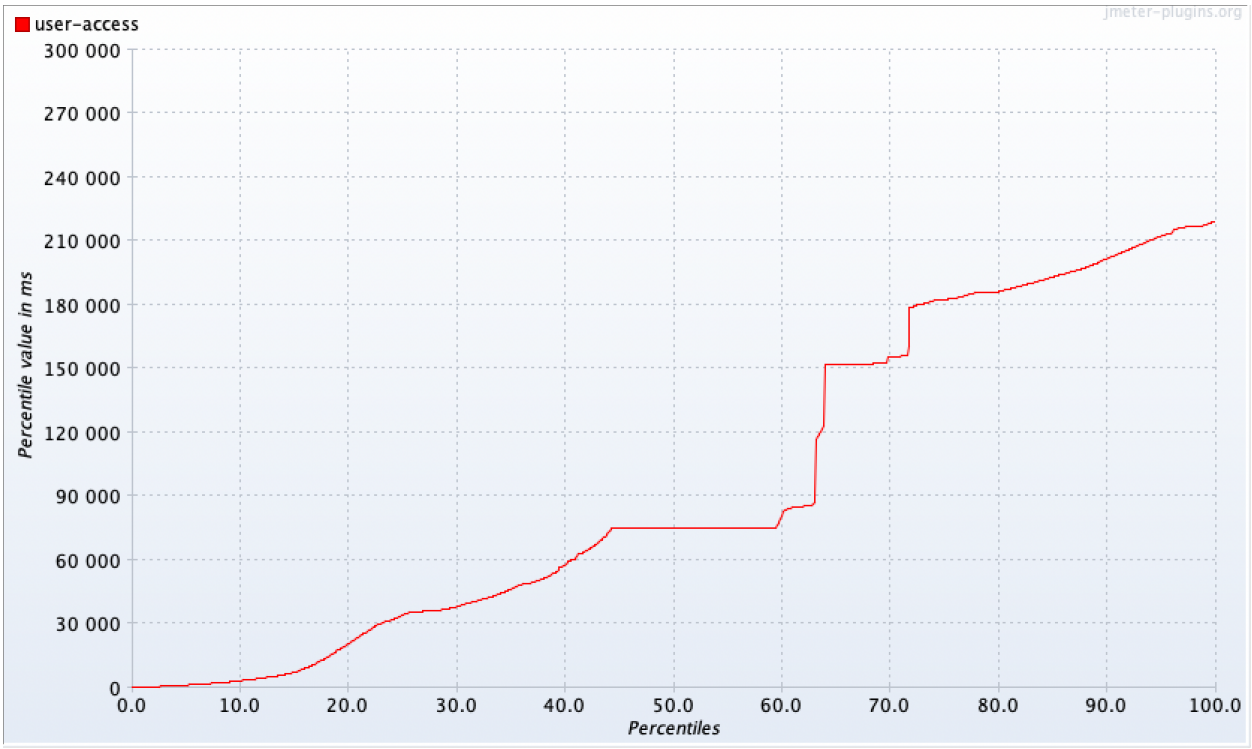
6. Transaction Throughput vs Threads: The formula for total server transaction throughput is <active threads> * 1 second / <1 thread response time>. It shows the total server's transaction throughput for active test threads.

   From the graph below we can observe that with an increasing number of active threads, the number of estimated transactions per second stays at 0.0/sec but there is a spike observed at 1042 active threads with the number of transactions / sec to be between 180000/sec to 190000 sec.

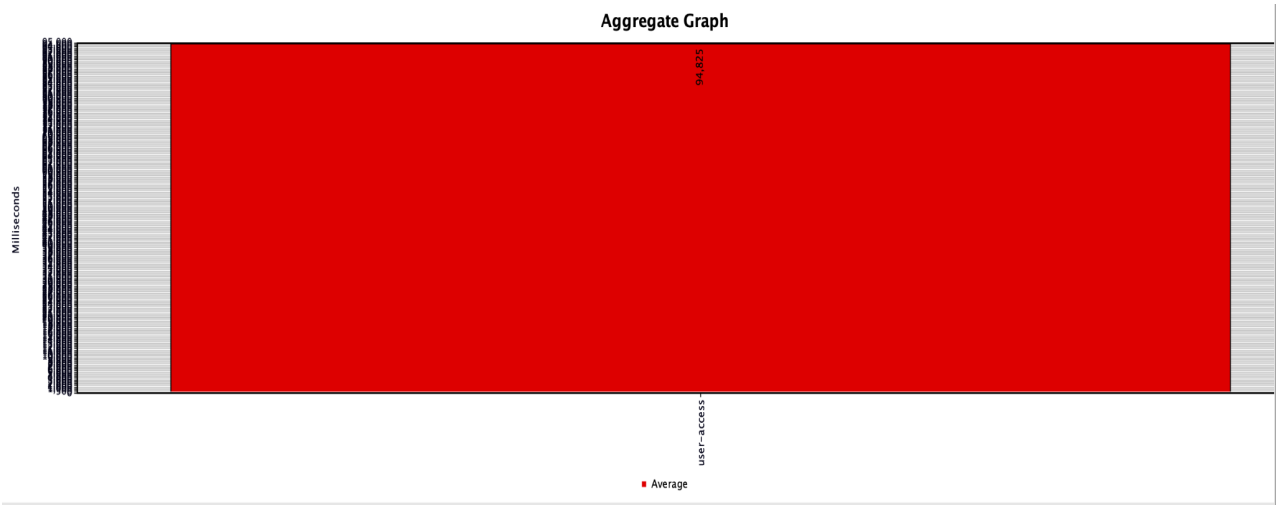


7. Response time percentile: displays the percentiles with respect to response time in milliseconds.

The percentile increases gradually with values forming a plateau between percentile 45.0 - 60.0. After which it steeply rises to a percentile value of 180000 ms and then it slowly rises. The 100th percentile has a percentile value of about 210000 seconds.



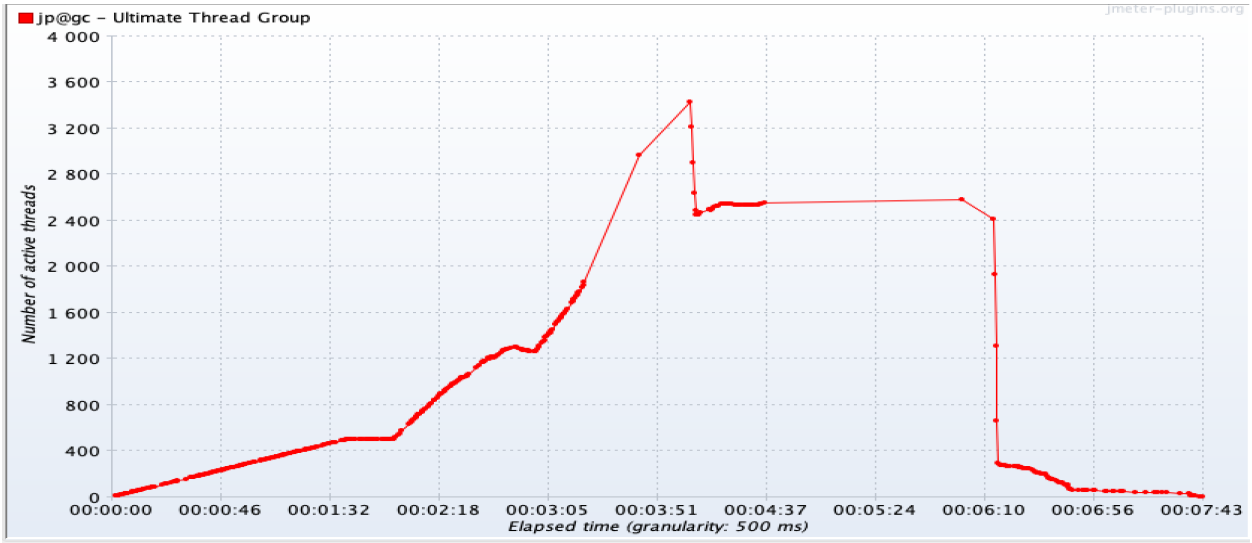
8. Aggregate graph : plots a bar chart for the response time metrics for each sample. The response time is a lot for /user-access which is why it is not visible in the graph.



4.) Get-user

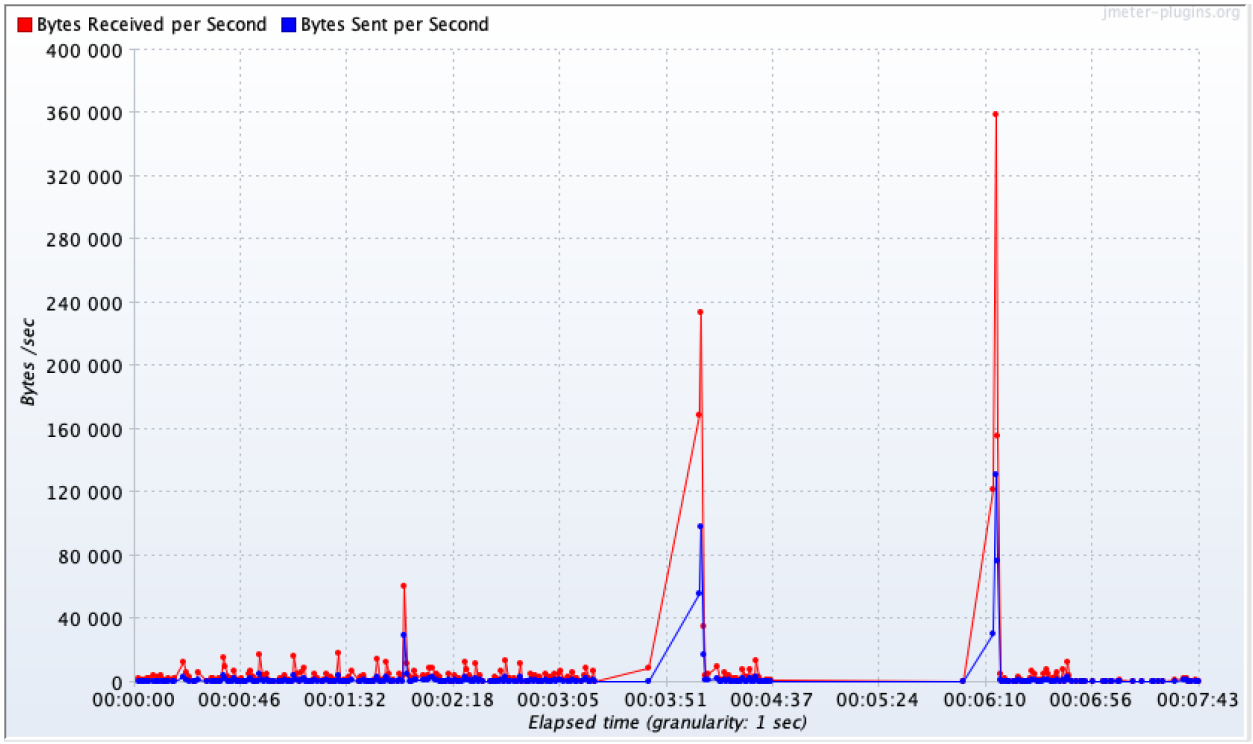
1. Active Threads over Time

From the below graph , we get from the number of active threads v/s Elapsed time stats. We notice that the peak comes at 3200 concurrent requests for get users which suggests the system is incapable of handling more than 3200 concurrent requests for get-users. Also, as the number of active threads increase, the elapsed time or the response for those increases as well.This is clearly shown in the graph until the peak is achieved.



2. Bytes throughput over time: shows the number of bytes that were sent and received, per time unit, throughout the test.

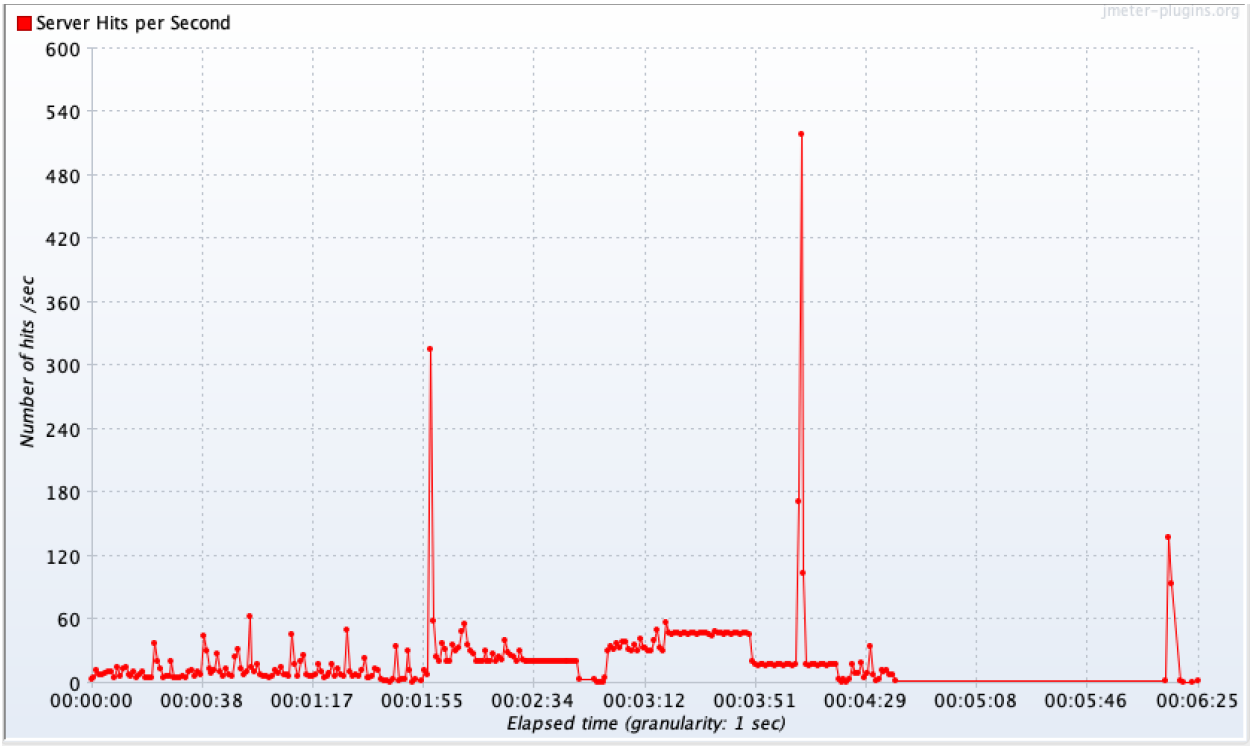
   In the graph below we can see that the bytes are sent and received continuously at a uniform rate with some spikes wherein the bytes sent and received, both increase but the bytes received per second is greater than than the bytes sent. There are 2 major spikes in the graph at elapsed time of 3.40 seconds and 6 seconds.



3. Hits per second

refers to the number of HTTP requests sent by the user(s) to the Web server in a second.

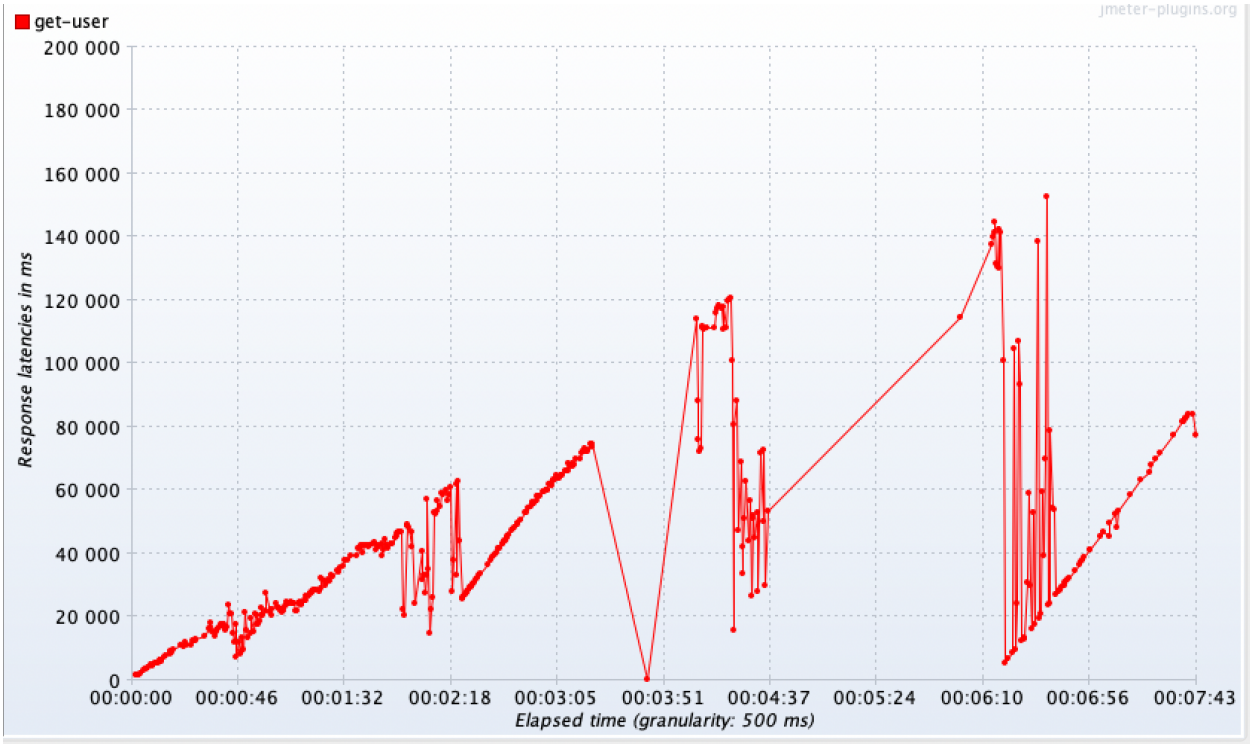
In the graph below we can observe that the number of hits spikes up at 520 hits/sec at about 4.17 seconds of the elapsed time. The server hits per second do not have a pattern in general and form plateaus at certain elapsed time periods. There are no major spikes which are seen apart from the one mentioned.



4. Response latencies over time

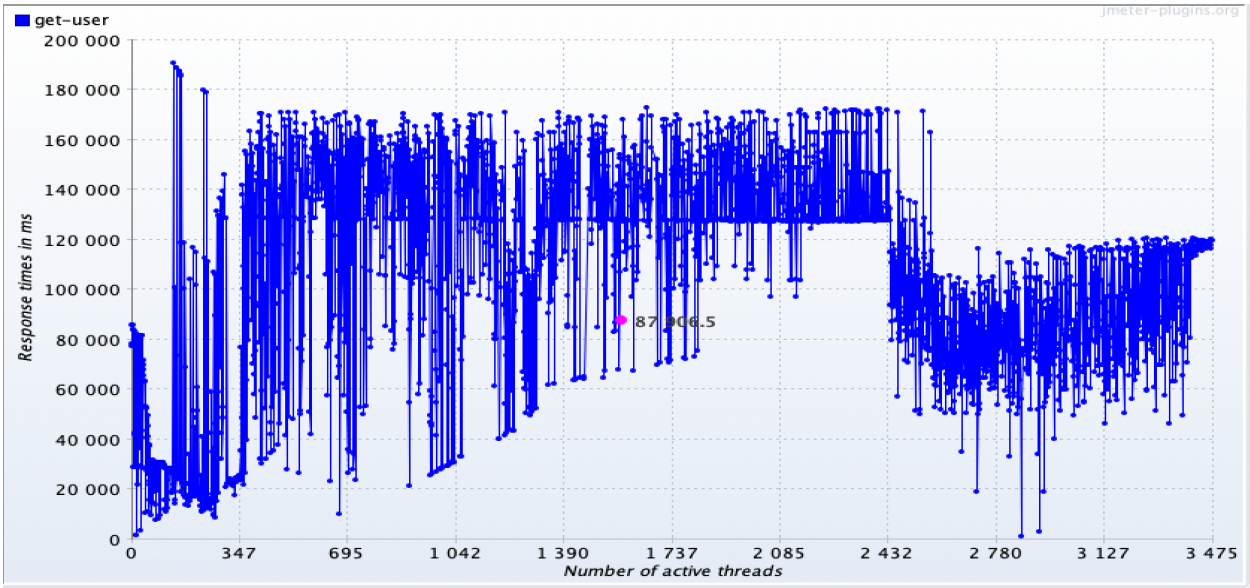
- provides the information about the time the tested server spends for the processing of the requests (samples).

  A latency is the duration between the end of the request and the beginning of the server response. The graph below shows that the latencies increase over time gradually but also has several drops with increasing elapsed time. The highest latency spike of 150000 ms occurs at 6:32 seconds.



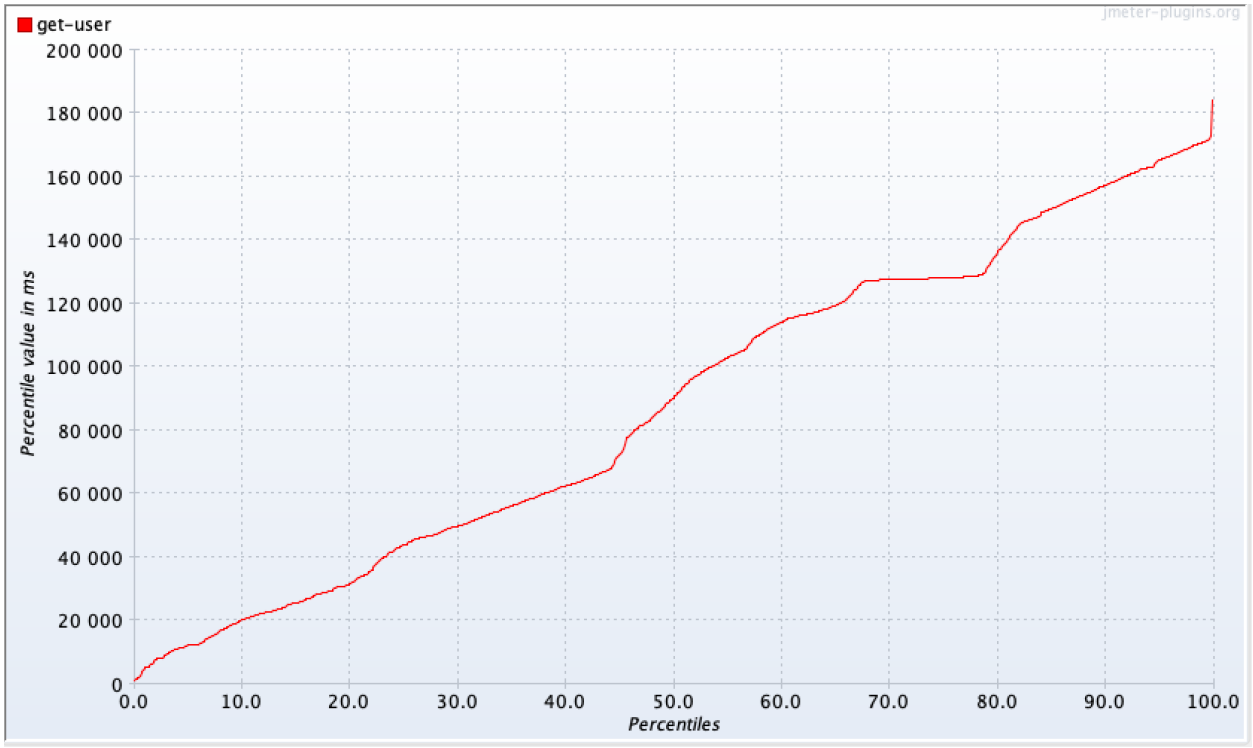
5. Response time vs threads: This graph shows how Response Time changes with the amount of parallel threads. The server takes longer to respond when a lot of users requests it simultaneously.

   The response time roughly remains in the same large range of 20000 - 170000 ms for the number of active threads as 1042 approximately. After which the response times vary in a lesser range and ultimately drop down slightly in the range of 40000 - 130000 ms roughly. The pink dot shows the average or optimal response time.



6. Response time percentile

displays the percentiles with respect to response time in milliseconds.
It seems to be a monotonous graph with continuously increasing percentile values in ms with percentiles. The 100th percentile has a percentile value of about 180000 seconds.



● Interpretation of the results and how the architecture might be improved:

We have displayed the interpretation of the graphs we obtained from each of the use cases we have and have mentioned what might be the possible solutions to make a better microservice architecture . Please refer to all the above graphs and their interpretations. Following points are observations and probable improvements from custos deployment as well as custos sdk testing:

- Errors while deploying custos and how that might be improved :It could be deployed with a lot of hacks , but we could stabilize the system by deploying all the services in the dev or staging. Services **iam-admin-core-service** and **identity-core-service** need to be redeployed again in staging after being in dev, so that might be improved from code perspective so that deployment becomes easier . We understand the pods run with the dev branch but we

faced issues with certificates while we were trying to create the tenant, that was the reason those two pods needed to be deployed in staging, so we could fix those certificate issues in the dev branch itself.

- Architecture understanding and possible enhancements : We noticed the architecture involves 33 microservices, which in itself becomes a huge task for communication between those microservices, and it's prone to errors if not handled properly. There were some microservices for even small tasks, if the system needs to be scaled , we think some smaller microservices should be merged so as to reduce the total number of microservices.

- Another interesting thing we noticed during our experiments is that while we were performing load testing on the CUSTOS client application. The microservice seemed to have gone "down". Every time we sent a request for any of the endpoints, we received the following error message from the CUSTOS microservice:

  <_InactiveRpcError of RPC that terminated with:
  status = StatusCode.UNAVAILABLE
  details = "Received http2 header with status: 504"
  debug_error_string = "{"created":"@1651782904.732950000","description":"Received http2 :status header with non-200 OK status","file":"src/core/ext/filters/http/client/http_client_filter.cc","file_line":124,"grpc_message":"Received http2 header with status: 504","grpc_status":14,"value":"504"}"
  >

  Looking at the error message we can make out that, when the microservice is under a lot of load, some part of the microservice is not functioning as it is supposed to and because of that it is sending out a wrong status header. We tried looking at the CUSTOS source code but, we weren't able to figure out the exact reasoning behind this error. This is a serious issue that can cause big problems in the future when there are a lot of users using this application.