

## **Performance Analysis for Jax Web Services and Node.js(REST)**

The performance analysis is performed for the server 1 which is simple math solver having two functions find\_prime and check\_prime. Find\_prime takes a number as a input and output all the prime number between 1 and that number. Check\_prime takes a number as input and returns whether the number is prime or not.

### **Case 1: 1000 calls by a single user**

Average Time Taken (**JAX WEB Services**) :

1000 calls -  $17400/1000 = 17.4$  ms

Average Time Taken (**node.js(REST)**):-

1000 calls - 8ms

Increase in latency is more than 10%

### **Case 2: 5000 calls by a single user**

Average Time Taken (**JAX WEB Services**) :

1000 calls -  $19399/1000 = 19.4$  ms

Average Time Taken (**node.js(REST)**):-

1000 calls - 7 ms

Increase in latency is more than 10%

### **Case 3: 100 users with 1000 calls**

Average Time Taken (**JAX WEB Services**) :

Aprox. 21 ms

**Analysis:**

latency is the time delay between the response received for the request sent. So According to above analysis node.js(REST) is able to resolve the request received much faster than the JAX web services. Following are the reasons:

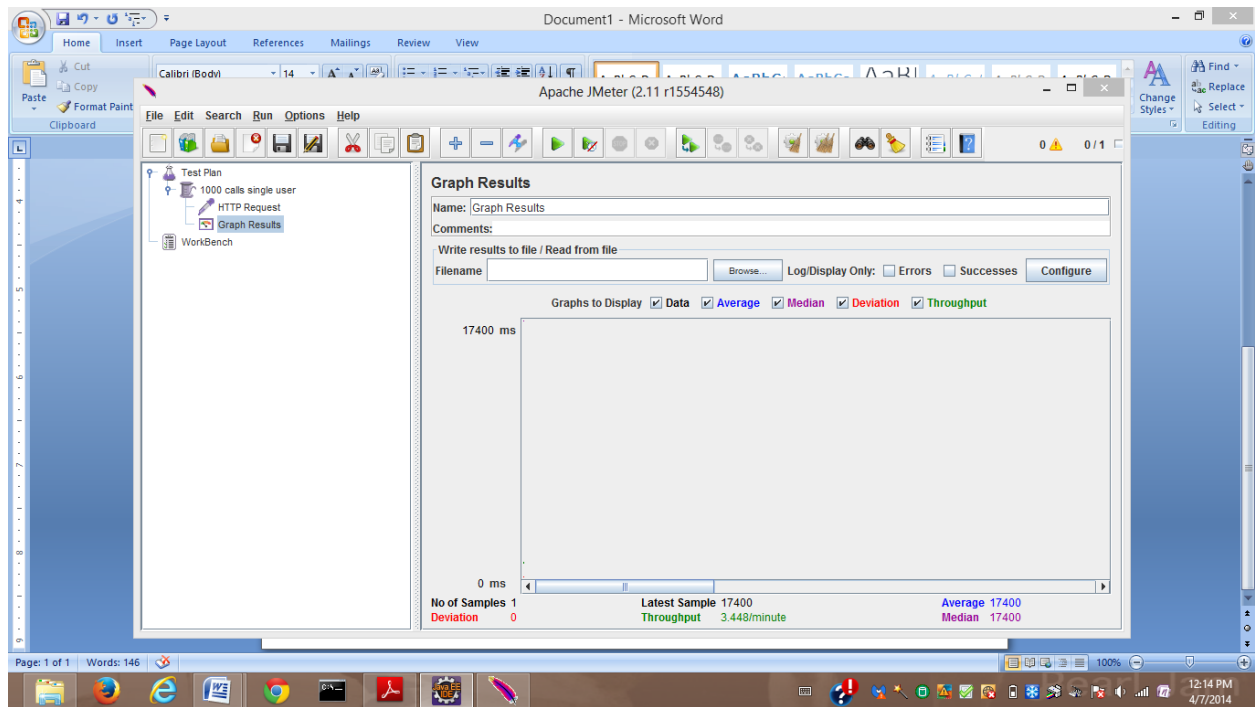
>node.js(REST) does not uses synchronous calls and thus does not wait for the response of a request if it is taking long. It goes on to serve another request. Since it executes asynchronously it is able to serve more requests which increases its throughput and the average time taken to resolve a request. It uses callback functions to support it's model. In case of the JAX-WS it resolves a request synchronously. So if a request takes more time to resolve , it waits for that request to finish and then only it goes on to serve other request. hence the average time taken increases and thus the latency increases.

> Other reason is the architecture of node.js(REST) and JAX-WS. Node.js(REST) does not require Tomcat server for its execution and everything is sent along the request and response which helps it to serve the requests faster. In case of JAX-WS the request and response has to pass through different layers which increases the latency time.

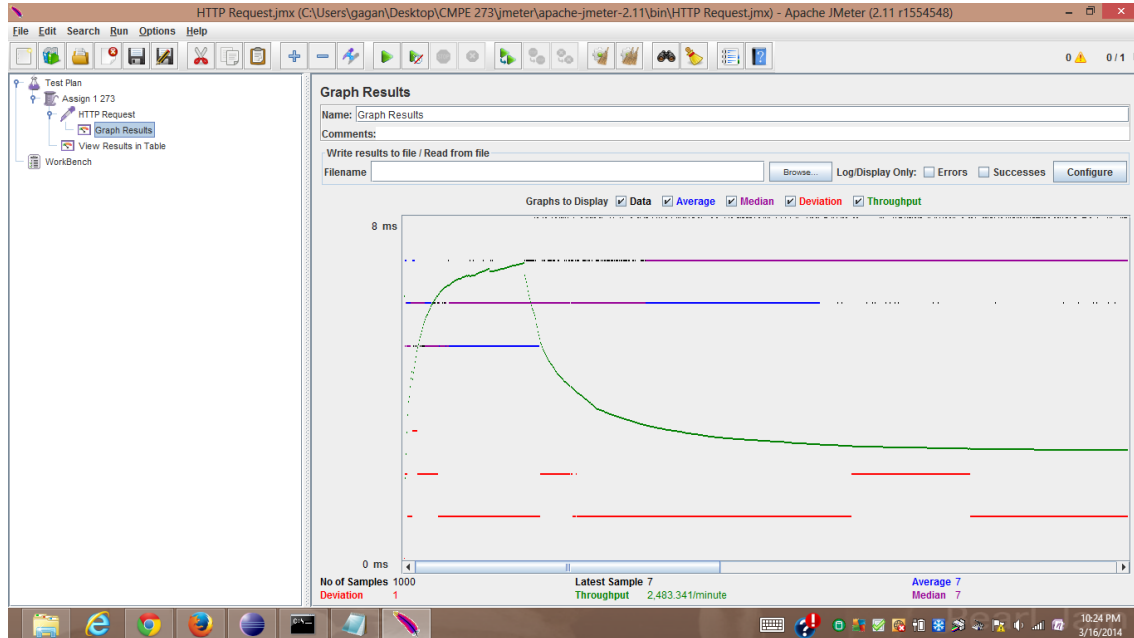
>One more observation is that node.js(REST) scales better than the JAX-WS. AS the average time for 5000 requests decreased in case of node.js that is due to it's asynchronous nature using event driven model as it does not queue requests and response like JAX-WS.

**Following are the result screenshots:**

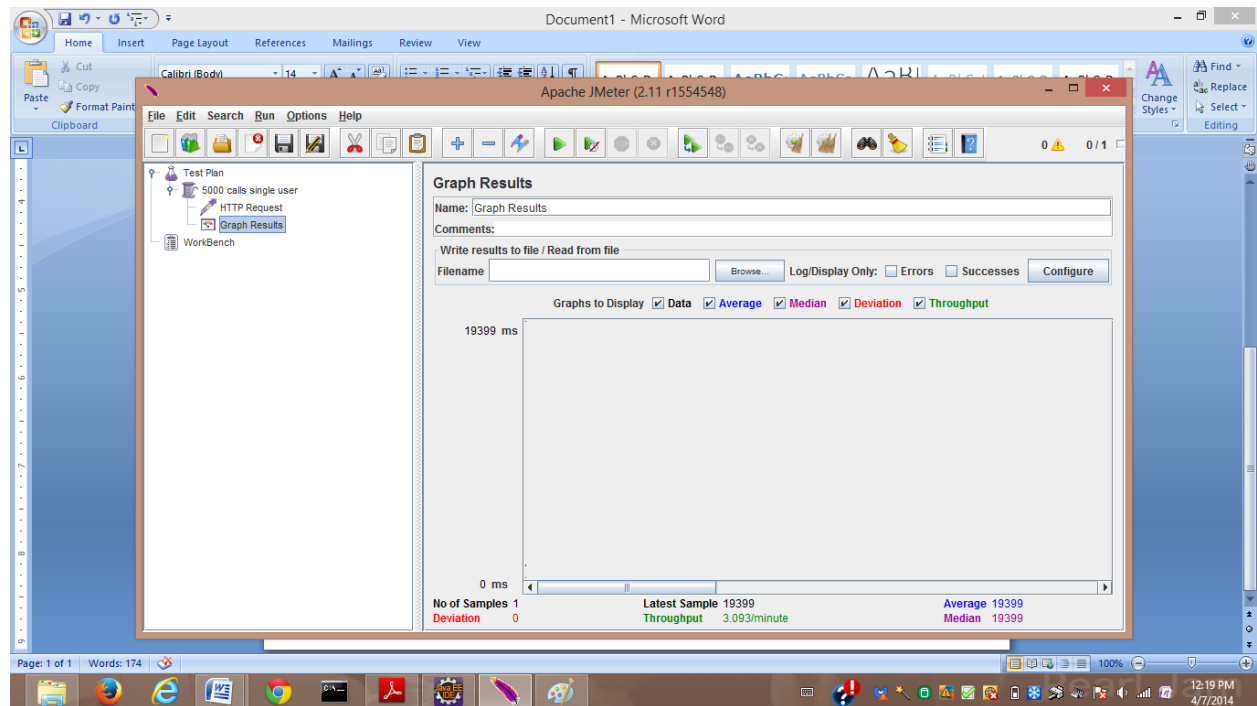
**1000 calls by a single user in JAX web service:**



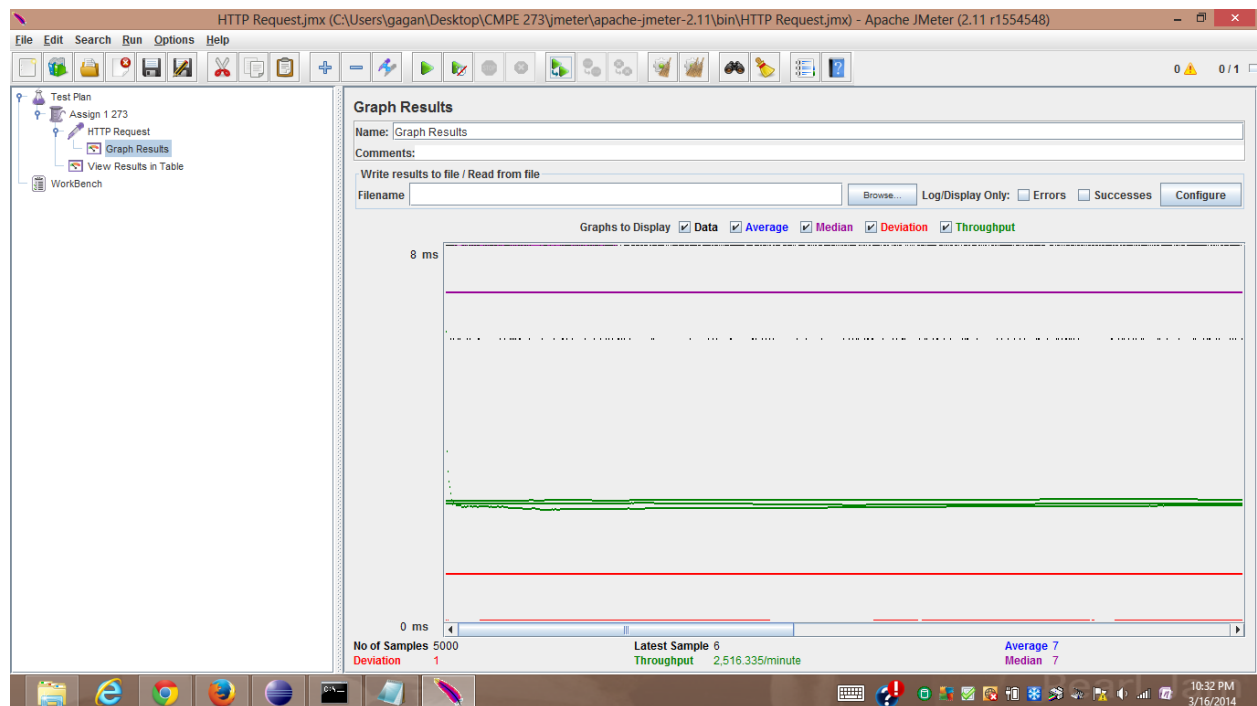
graph with 1000 calls to the server(single user)(node.js)



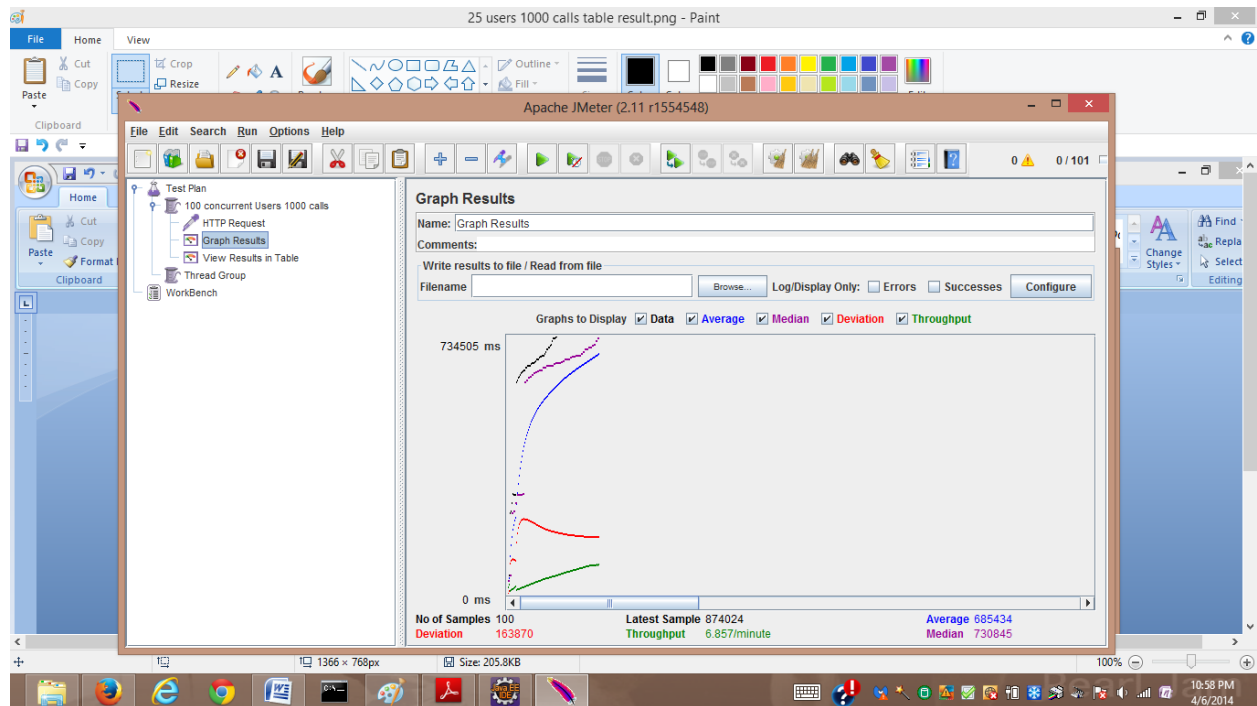
## Graph with 5000 calls for single user using Jax-Ws



## Graph with 5 thousand calls(single user)(node.js)



## Graph for JAX-WS for 25 users with 1000 calls(table and graph result):



ReportLab2(009301019).docx - Microsoft Word

Apache JMeter (2.11 r1554548)

Test Plan

- 100 concurrent Users 1000 calls
  - HTTP Request
  - Graph Results
  - View Results in Table
  - Thread Group
  - WorkBench

Sample #	Start Time	Thread Name	Label	Sample Time(ms)	Status	Bytes
1	22:32:22.199	100 concurrent ...	HTTP Request	36868	Success	233
2	22:32:22.338	100 concurrent ...	HTTP Request	53184	Success	233
3	22:32:23.071	100 concurrent ...	HTTP Request	232253	Success	233
4	22:32:22.123	100 concurrent ...	HTTP Request	236499	Success	233
5	22:32:23.004	100 concurrent ...	HTTP Request	260325	Success	233
6	22:32:22.301	100 concurrent ...	HTTP Request	285177	Success	233
7	22:32:22.726	100 concurrent ...	HTTP Request	284776	Success	233
8	22:32:23.007	100 concurrent ...	HTTP Request	284677	Success	233
9	22:32:22.818	100 concurrent ...	HTTP Request	284890	Success	233
10	22:32:22.545	100 concurrent ...	HTTP Request	605540	Success	233
11	22:32:23.098	100 concurrent ...	HTTP Request	609093	Success	233
12	22:32:22.224	100 concurrent ...	HTTP Request	618766	Success	233
13	22:32:23.182	100 concurrent ...	HTTP Request	620922	Success	233
14	22:32:22.052	100 concurrent ...	HTTP Request	625837	Success	233
15	22:32:22.888	100 concurrent ...	HTTP Request	630803	Success	233
16	22:32:22.837	100 concurrent ...	HTTP Request	635008	Success	233
17	22:32:22.270	100 concurrent ...	HTTP Request	637294	Success	233
18	22:32:22.602	100 concurrent ...	HTTP Request	639599	Success	233
19	22:32:22.966	100 concurrent ...	HTTP Request	640346	Success	233
20	22:32:22.035	100 concurrent ...	HTTP Request	646777	Success	233
21	22:32:22.788	100 concurrent ...	HTTP Request	646224	Success	233
22	22:32:22.093	100 concurrent ...	HTTP Request	650866	Success	233
23	22:32:22.714	100 concurrent ...	HTTP Request	650983	Success	233
24	22:32:22.515	100 concurrent ...	HTTP Request	652034	Success	233
25	22:32:23.041	100 concurrent ...	HTTP Request	653009	Success	233

Scroll automatically? ☐ Child samples? ☐ No of Samples 100  
Latest Sample 874024  
Average 685434

## Graph with 100 users with concurrent 1000 calls(Node.js)

