Stress-Testing Containerized Microservices

Meleena Radcliffe (ID: 1319196)
07/06/2019

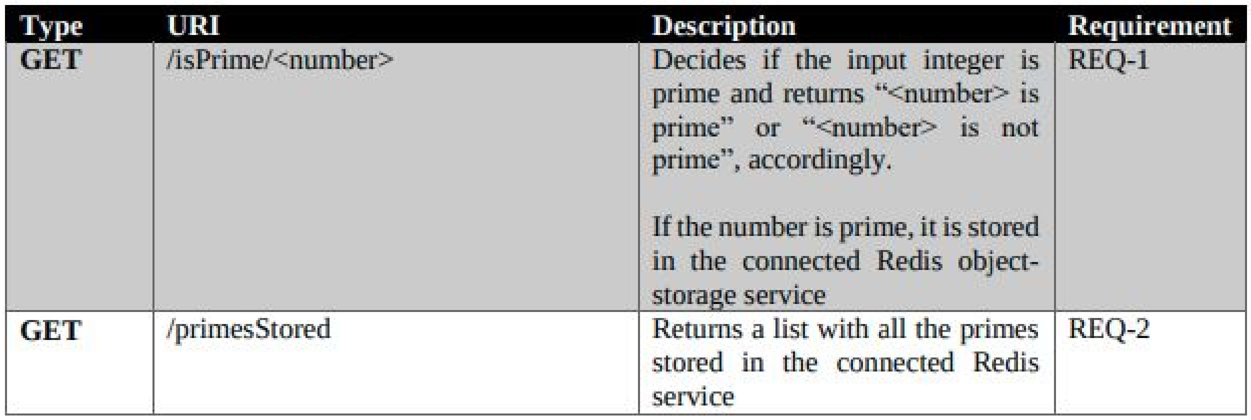
Link to GitHub repository: **https://github.com/lyre-the-human/COMPX341-A4-Docker-**

Table of Contents

Introduction

This document details both the implementation of a simple containerized application-server that implements an HTTP restful API as well as the stress testing of this API using JMeter.

The API consists of two functions written in Python.



Type	URI	Description	Requirement
GET	/isPrime/<number>	Decides if the input integer is prime and returns "<number> is prime" or "<number> is not prime", accordingly. If the number is prime, it is stored in the connected Redis object-storage service	REQ-1
GET	/primesStored	Returns a list with all the primes stored in the connected Redis service	REQ-2

Software used in the process of creating this includes Docker, Flask, the Python API of redis, and JMeter.

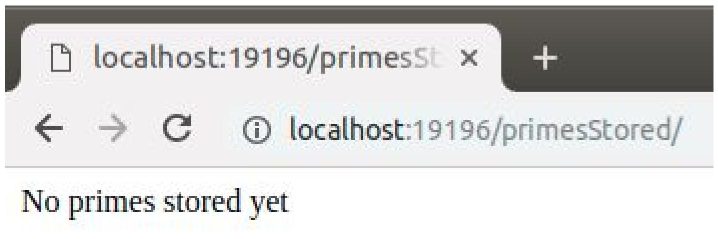
Discussion of Test Cases

Black Box Testing for isPrime:

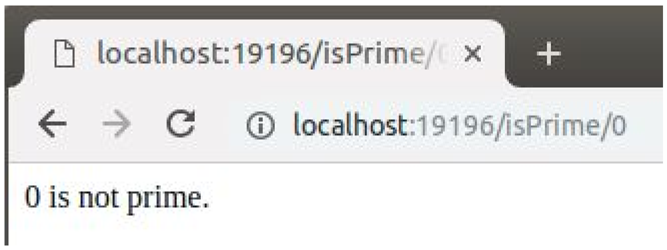
Test Case	Expected output	Unexpected output	Explanation
/isPrime/0	“0 is not prime”	Anything else	Invalid case
/isPrime/1	“1 is not prime”	Anything else	Boundary case
/isPrime/-1	“-1 is not prime”	Anything else	Negative numbers cannot be prime (invalid case)
/isPrime/2	“2 is prime”	Anything else	Valid case
/isPrime/hello	“Please provide an integer”	Error messages, etc.	Invalid input should not cause a server error
/isPrime/2.5	“Please provide an integer”	Anything else	Prime numbers must be integers. The program should not try to process a negative number

Black Box Testing for isPrime AND primesStored:

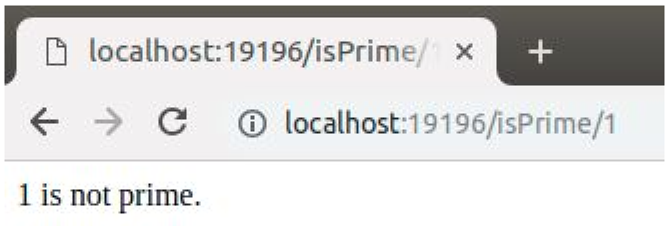
Test Case	Expected output	Unexpected output	Explanation
/primesStored/	“No primes stored yet”	Errors, etc. Anything else	If no primes have been stored then none should be displayed
/isPrime/0 /primesStored/	“No primes stored yet”	Anything else	0 is not prime, so it should not appear in the cache
/isPrime/2 /primesStored/	2	Anything else	2 is prime (valid case) so it should appear in the list
/isPrime/2 /isPrime/2 /isPrime/5 /primesStored/	2, 5	Anything else e.g. ‘2 , 2, 5’ ‘ 2’	Multiple primes have been stored; multiple primes should be displayed. Duplicates should not be stored



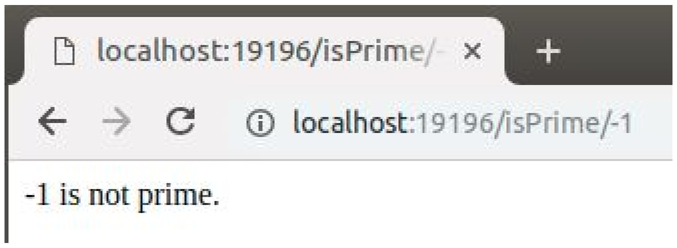
```
No primes stored yet
```



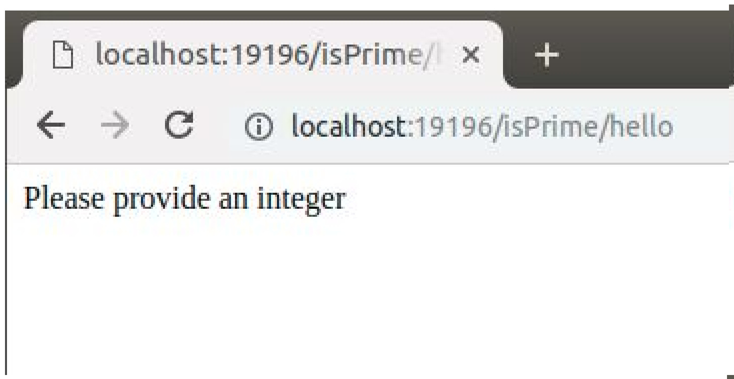
```
0 is not prime.
```



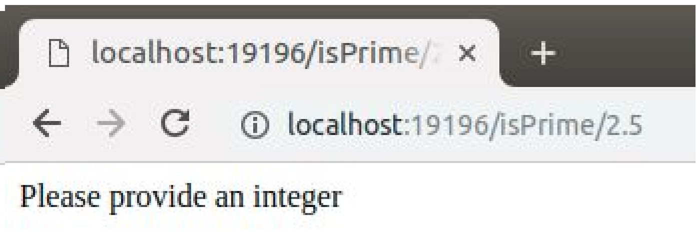
```
1 is not prime.
```



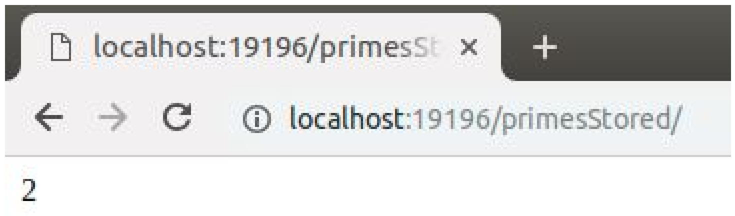
```
-1 is not prime.
```



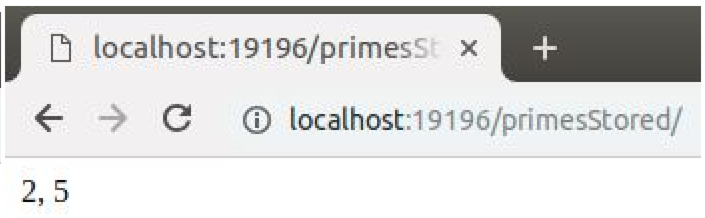
```
Please provide an integer
```



```
Please provide an integer
```



```
2
```



```
2, 5
```

White Box Testing

I also made use of white box testing techniques by testing each block of code as I implemented the restful API.

For example, within my isPrime function I first tested that the function could get the number from the app route ‘/isPrime/<number>’. I tested this by having the program return the number as-is. Once I verified that the function could get the number from the URL, I next wrote some code to check if the number was below 2, as all prime numbers must be 2 or greater. Next I implemented an algorithm to check if the number was prime. Only after testing each of these blocks of code did I move on to implement redis functionality for this function.

Similarly, I first tested within my primesStored function that I could return a list of integers where all the elements were hardcoded when receiving a request at the app route ‘/primesStored/’. Only after this did I add the use of redis to this function in order to get at a list where the list of integers is not necessarily known.

Stress Testing with JMeter

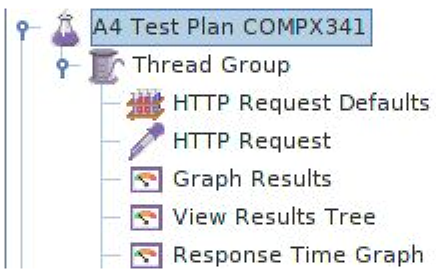
Scenario 1

Scenario 1 repeatedly decides if the number 2147483647 is prime by invoking the app's isPrime URL.

For this scenario I did the following:

- Created a thread group of 50 users with a ramp-up period of 1 second, checked the "Loop Count" box to 'forever', and set the duration to 60 seconds.
- Added a "HTTP Request Defaults" with a port number of 80 and a server name of "localhost:19196" to the thread group.
- Added an HTTP request with a path of "/isPrime/2147483647" to the thread group
- Added a "graph results" listener to the thread group
- Added a 'response time graph' listener to the thread group

  (Right: Screenshot of test plan for Scenario 1)



A4 Test Plan COMPX341
Thread Group
HTTP Request Defaults
HTTP Request
Graph Results
View Results Tree
Response Time Graph

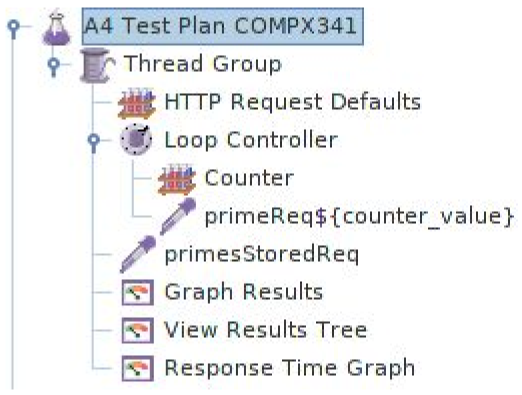
Scenario 2

Scenario 2 first invokes the isPrime API for all numbers between 1 and 100.
It then repeatedly invokes the primesStored URL of the app.

For this scenario I did the same as scenario 1, except for the following:

- Added a loop controller to the thread group with a loop count of 100.

- Added a counter to the loop controller that starts at 1 and increments by 1 with a maximum of 100. I also added “counter_value” as a reference name.



```
A4 Test Plan COMPX341
  Thread Group
    HTTP Request Defaults
    Loop Controller
      Counter
      primeReq${counter_value}
    primesStoredReq
    Graph Results
    View Results Tree
    Response Time Graph
```

- Put the first HTTP request i created inside the loop controller. I also changed the path of my HTTP request to “/isPrime/${counter_value}”

- Created another HTTP request outside the scope of my loop controller and below this group. I set the path of this to /primesStored/

(Left: Screenshot of test plan for Scenario 2)

Results of Stress Tests

Key for Throughput results:

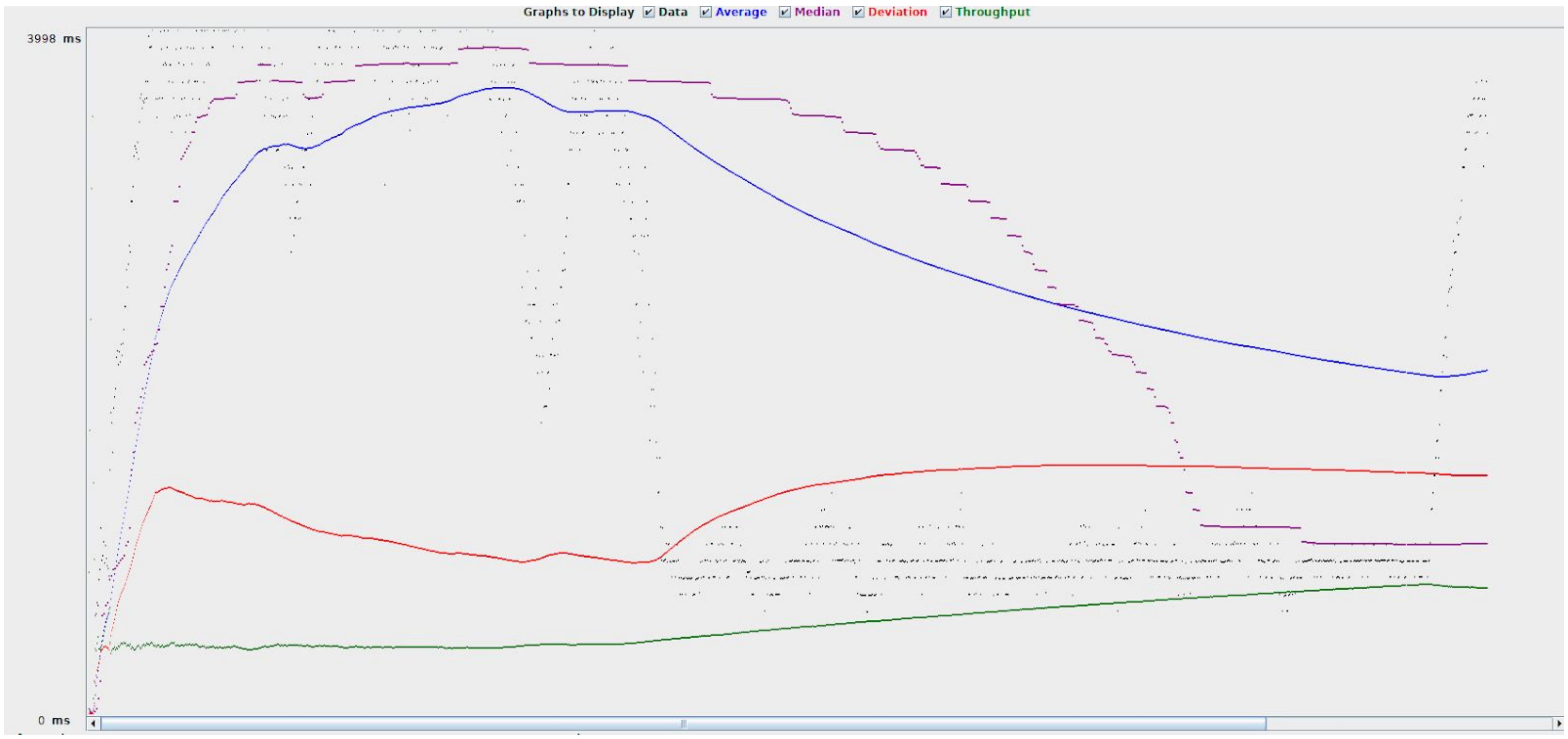
Green = Throughput (requests per minute); Red = Deviation; Purple = Median; Blue = Average.
The x-axis represents elapsed time, and the y-axis represents latency.

Key for Response Time results:

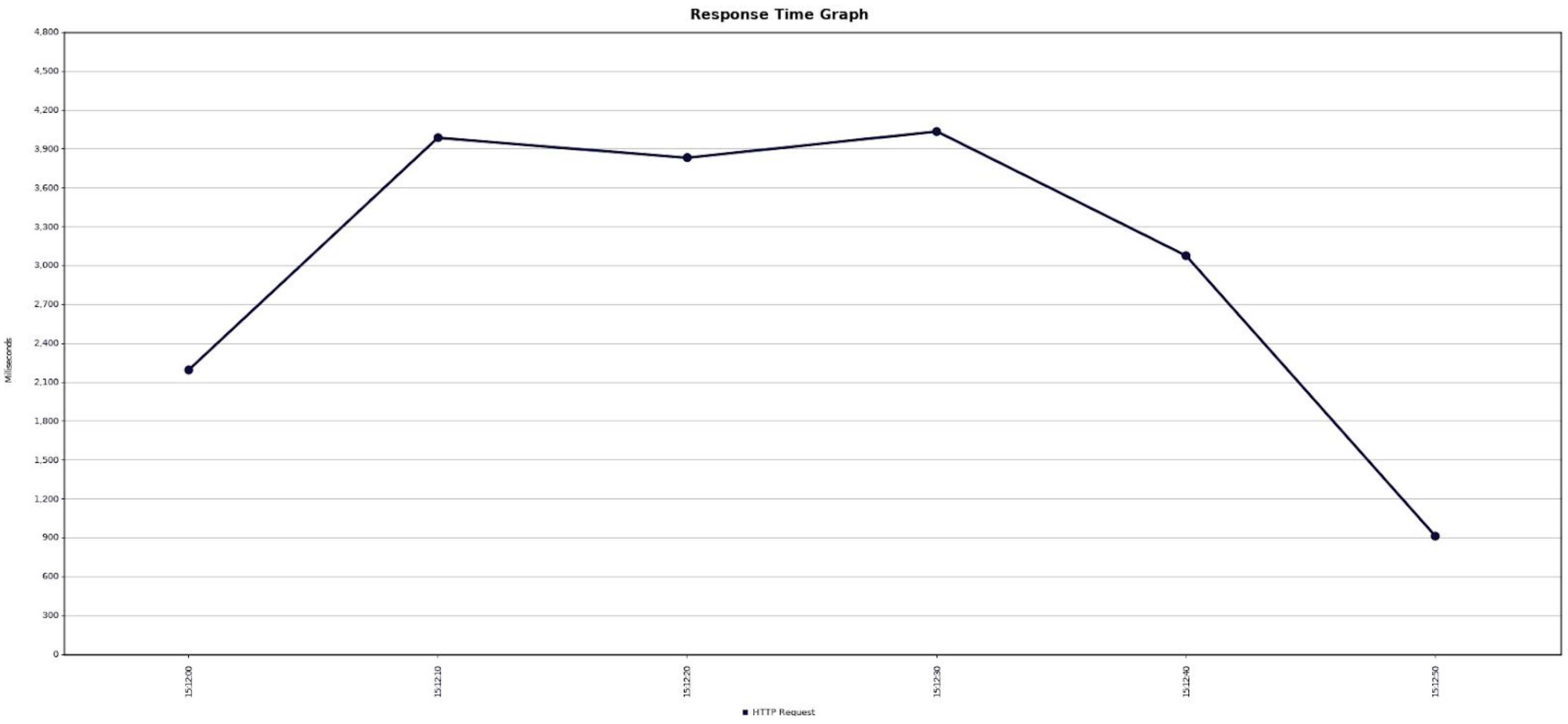
X-axis represents time of execution, while y-axis represents the response time of each request. Each type of request is tracked using a different colour to graph.

Scenario 1

<u>Scenario 1 testing with with a CPU limit of 0.1</u>

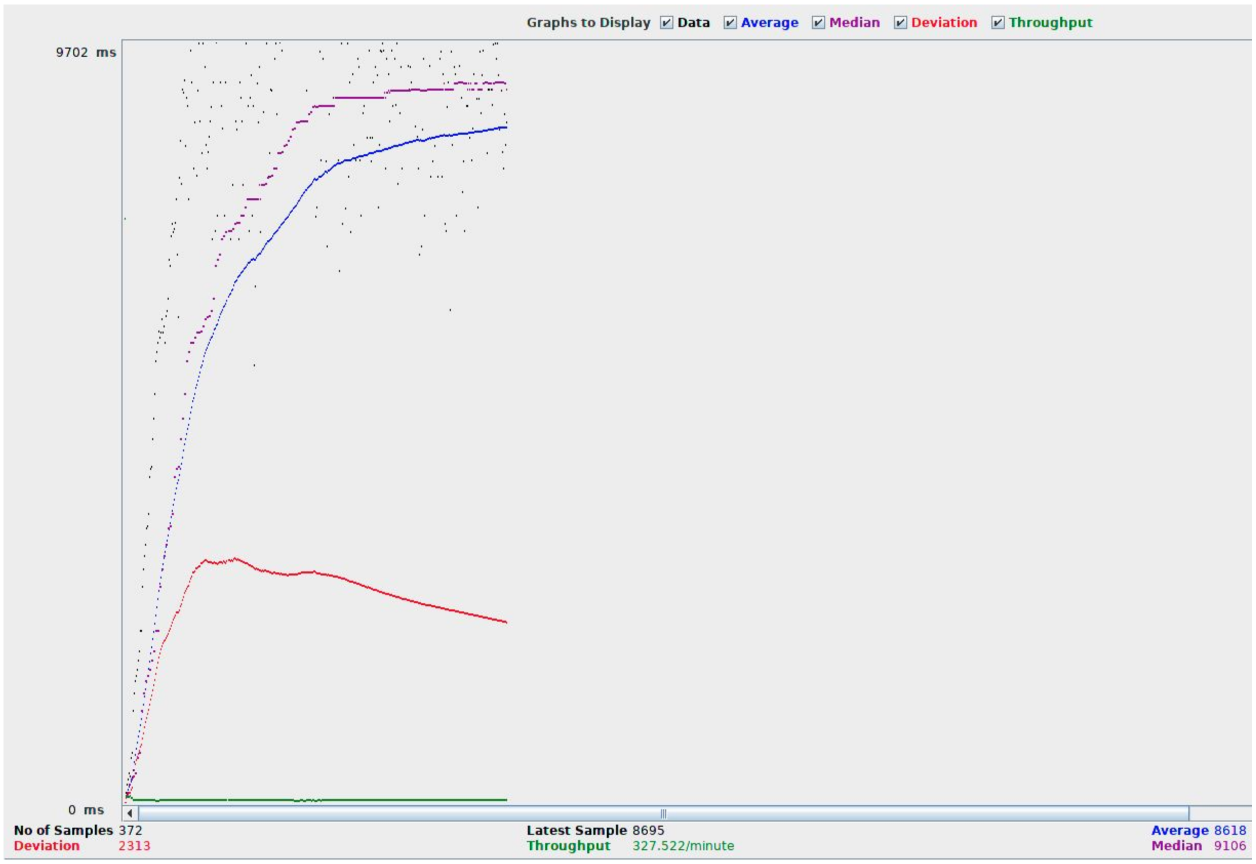


In scenario 1, throughput has a slight upward trend over 60s

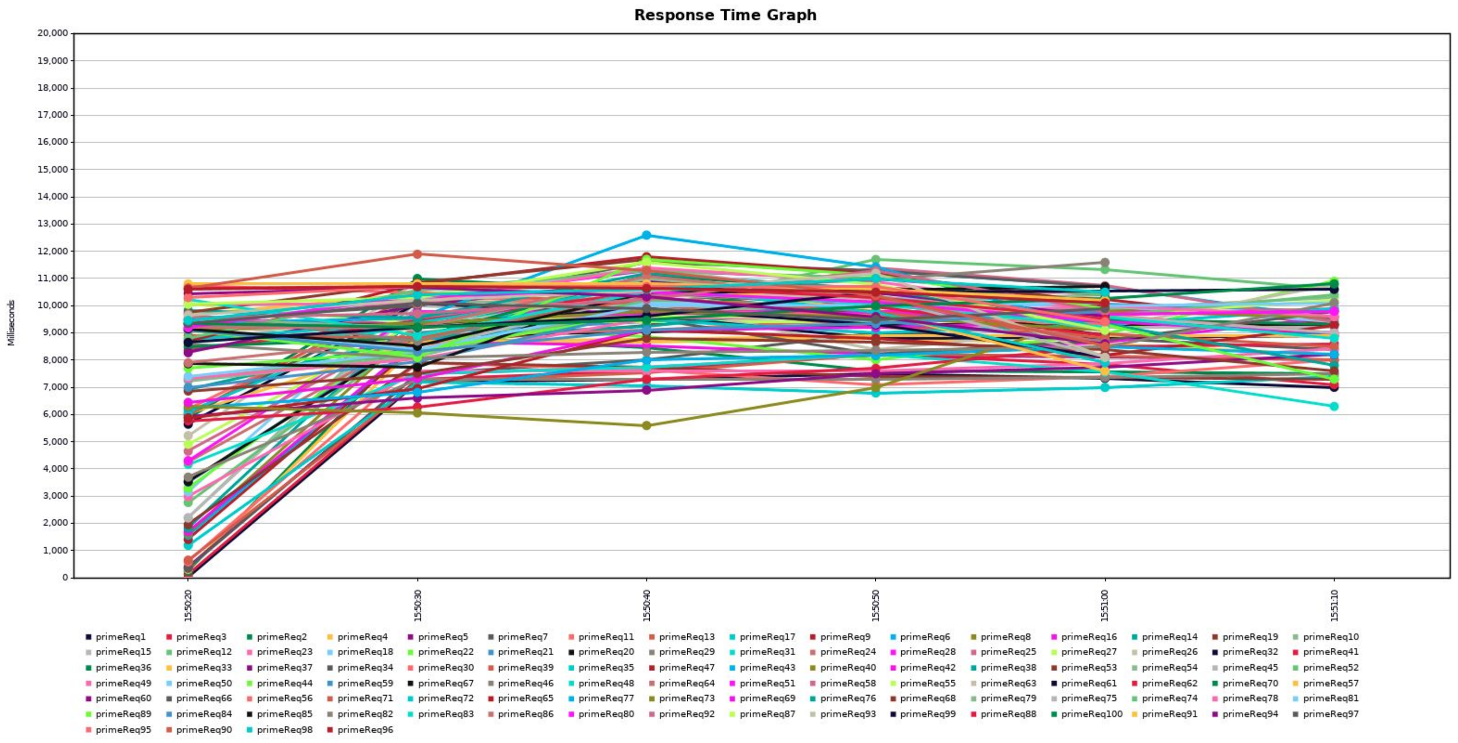


Scenario 2

__Scenario 2 testing with with a CPU limit of 0.1__



No of Samples 372
Deviation 2313
Latest Sample 8695
Throughput 327.522/minute
Average 8618
Median 9106



Scenario 1 With Varied CPU limits and Timer Delays

I decided to run my experiments exclusively on the test plan for scenario 1, as the data I collected from scenario 2 was difficult to interpret due to how many different types of requests there were.

I included my ‘default’ CPU limit (i.e. 0.1) as one of my 3 different CPU limits for the web service. In addition to this limit, I also tested CPU limits 0.2 and 0.5.

I tested three different timer delays as well. I tested a constant timer of 300ms, a gaussian random timer with constant delay offset of 300ms and a deviation of 100ms, and a uniform random timer with a random delay maximum of 300ms and a constant delay offset of 0ms.

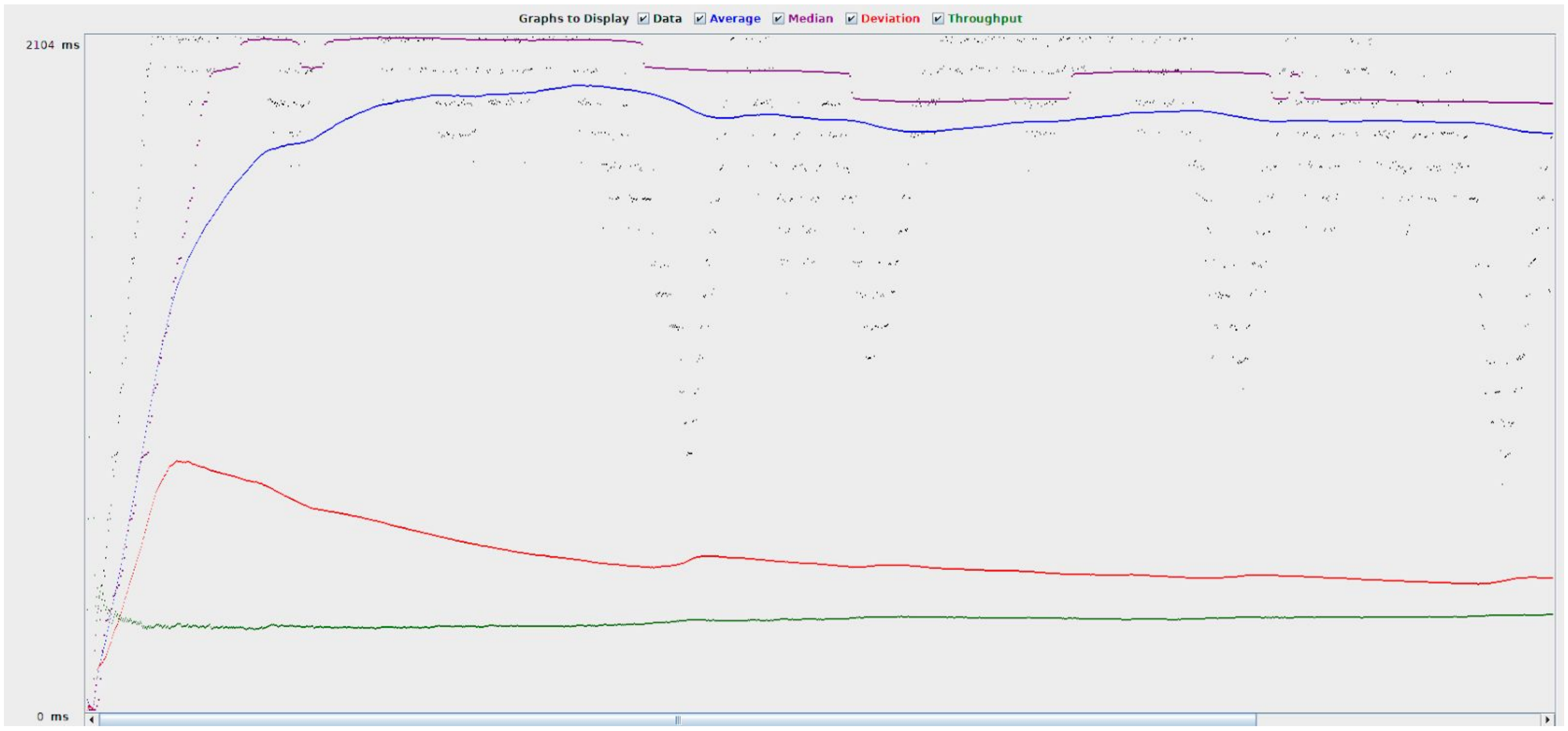
Scenario 1 tested by itself is discussed in relation to experiments done on it. However the results of Scenario 2 are discussed here.

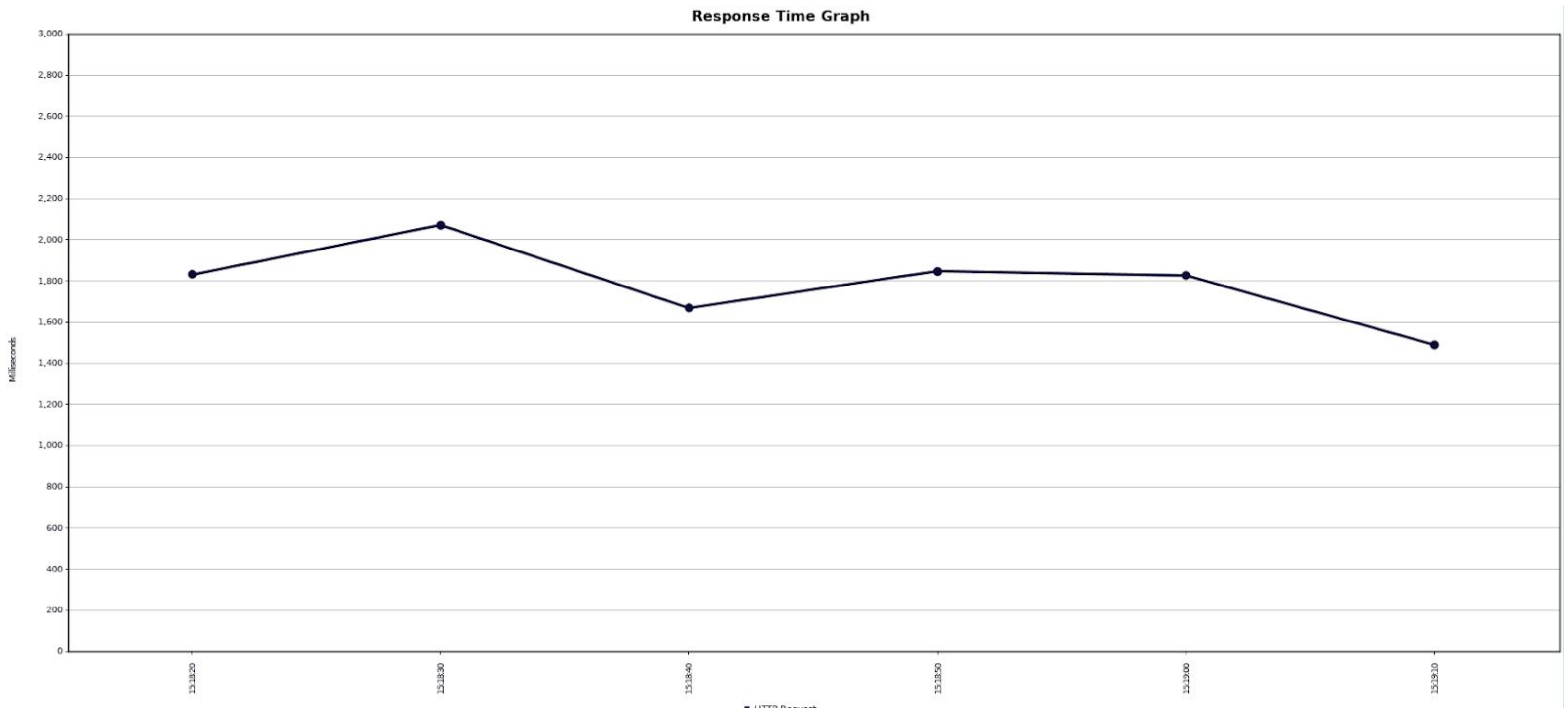
Results of Scenario 2 with “default” settings (CPU limit at 0.1)

Throughput stayed low at ~300 req- sending lots of requests at once is not necessarily accurate to real life, and /minute. This was likely because of how many requests the server was receiving at once, meaning that the queue would have become full very quickly and probably would not have emptied by the time the 60s were over.

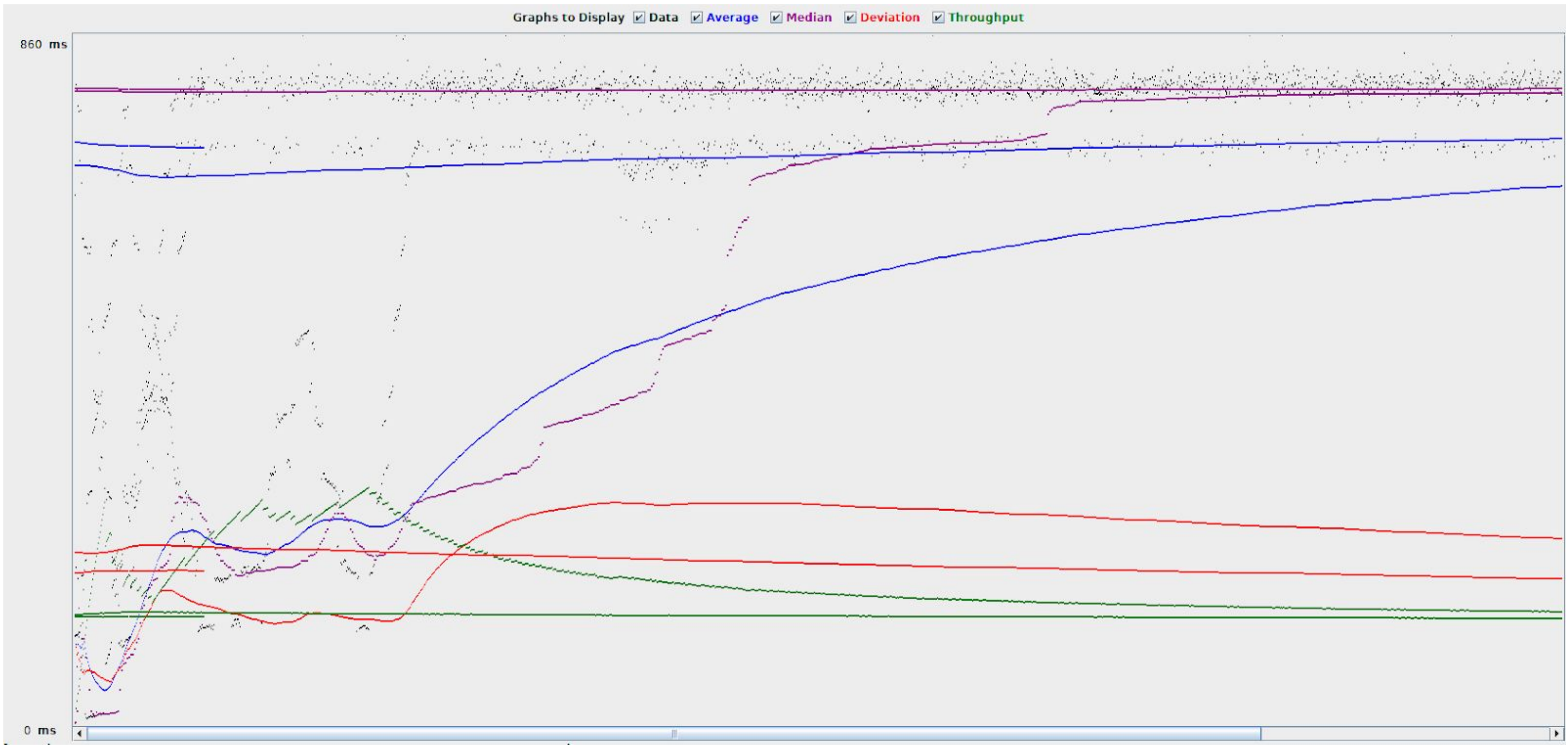
In general, request response time started out low for requests that were made for lower numbers, which makes sense as isPrime for lower numbers was requested first, meaning that the queue has not yet filled up. However, as time goes on this distinction disappears, with all requests made taking roughly equally long to process once the queue is filled with requests to process.

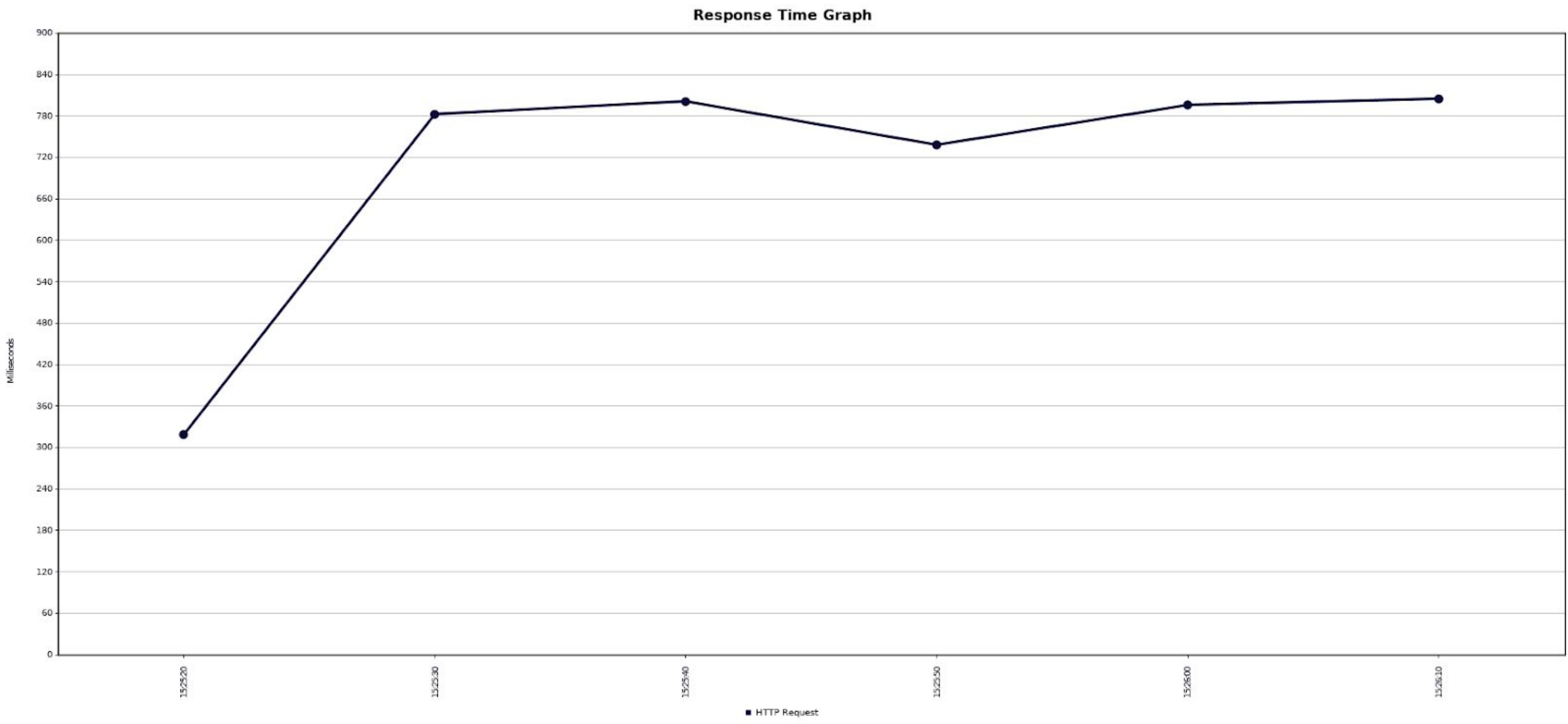
Scenario 1 testing with with a CPU limit of 0.2



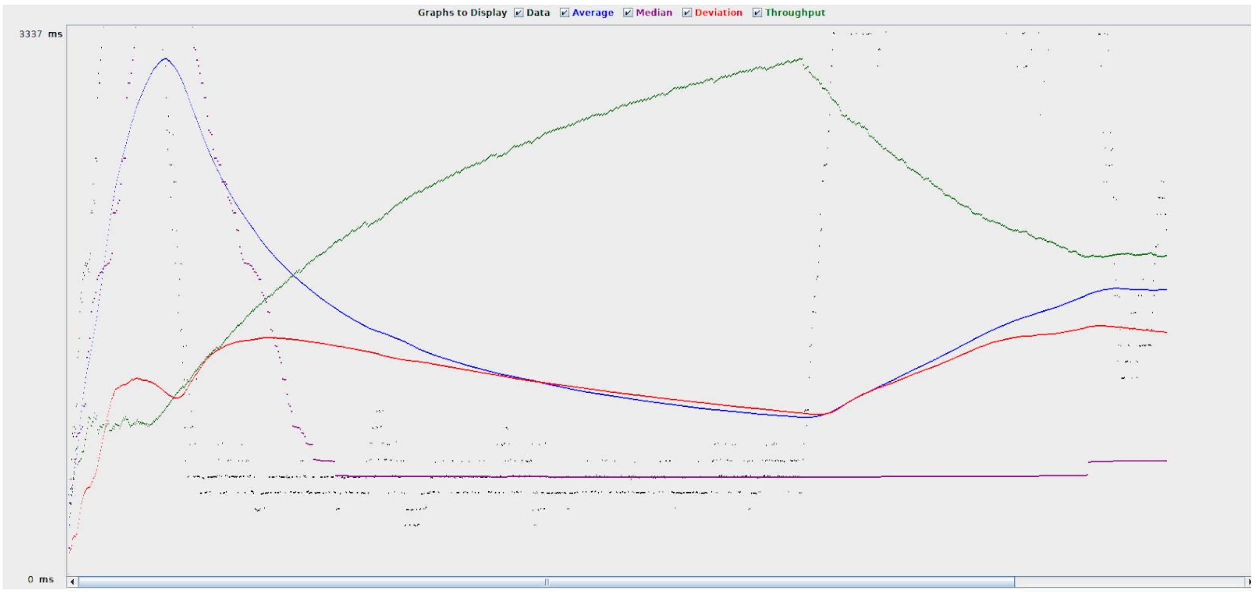


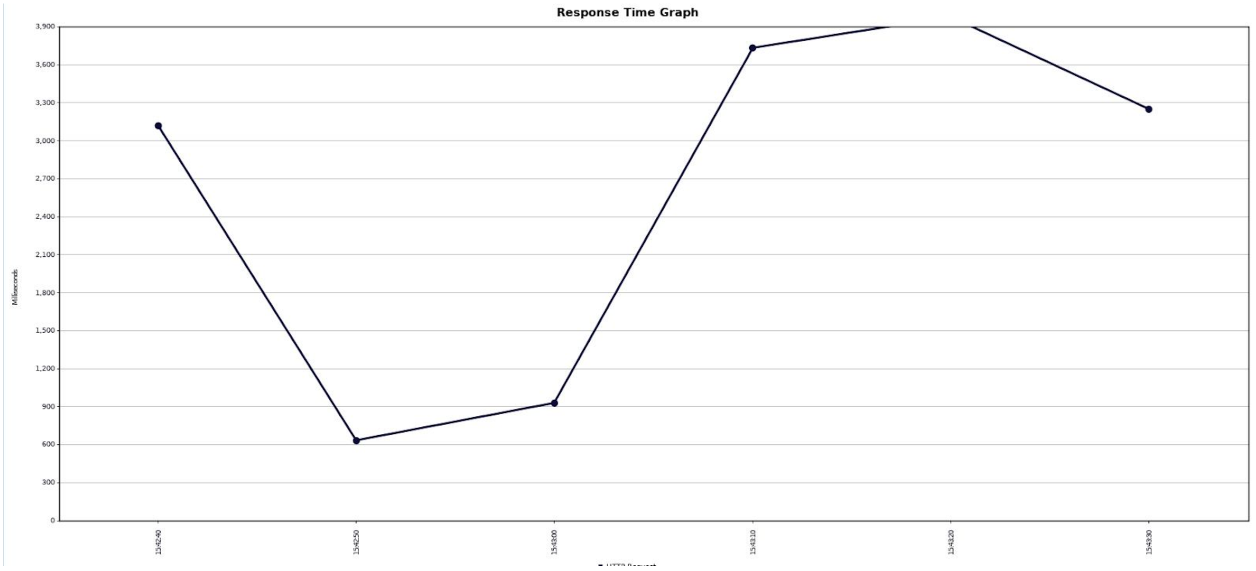
<u>Scenario 1 testing with with a CPU limit of 0.5</u>



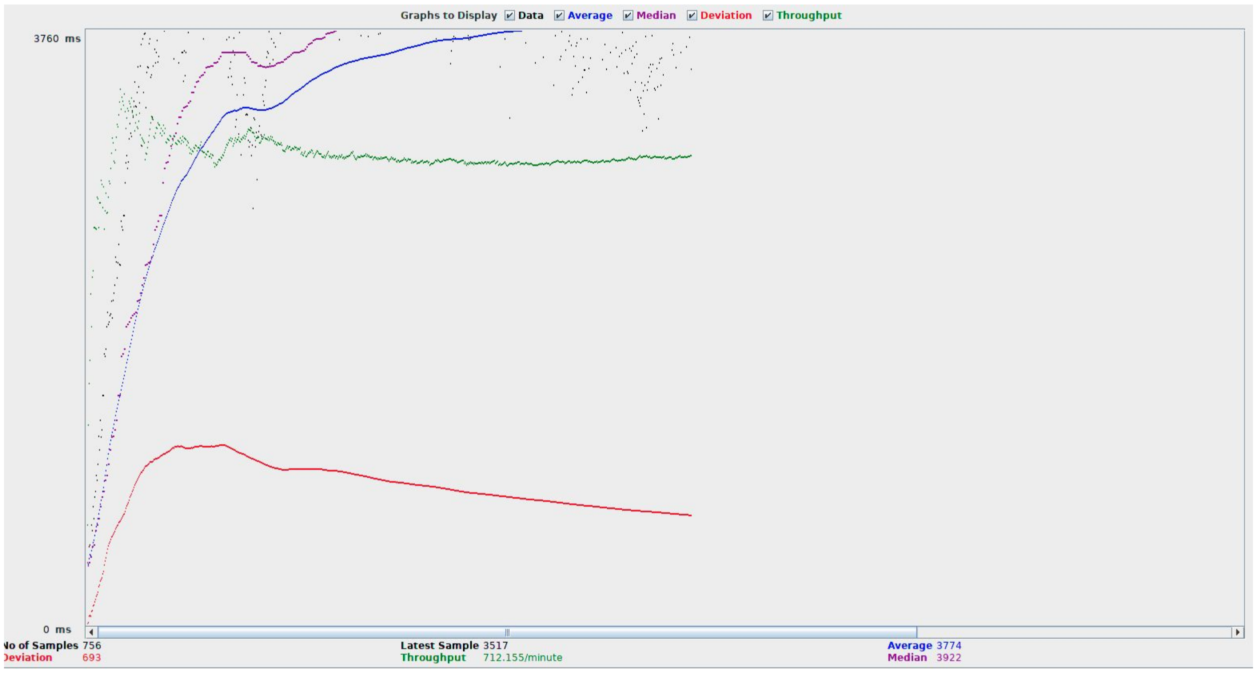


Scenario 1 testing with with a Constant timer

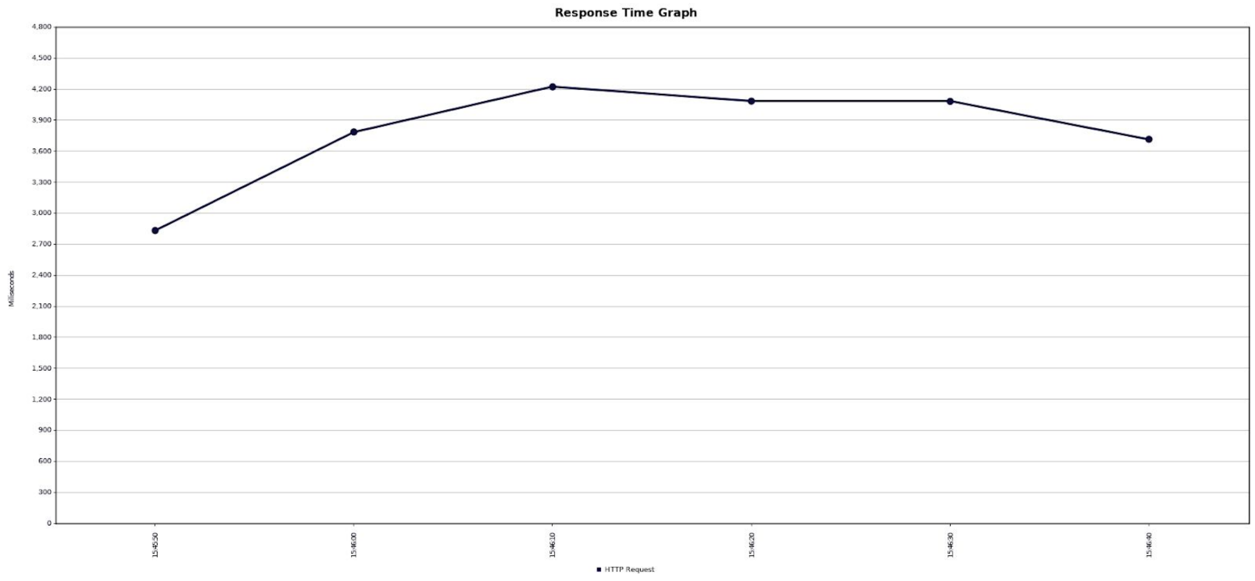




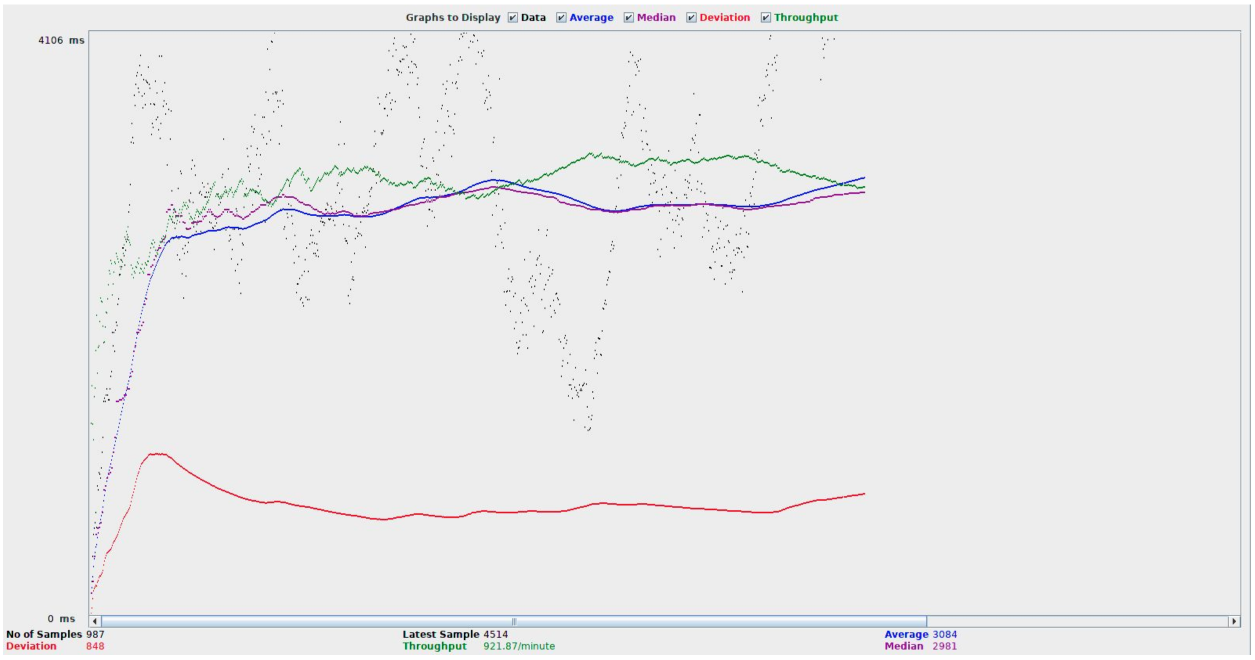
<u>Scenario 1 testing with Gaussian timer</u>



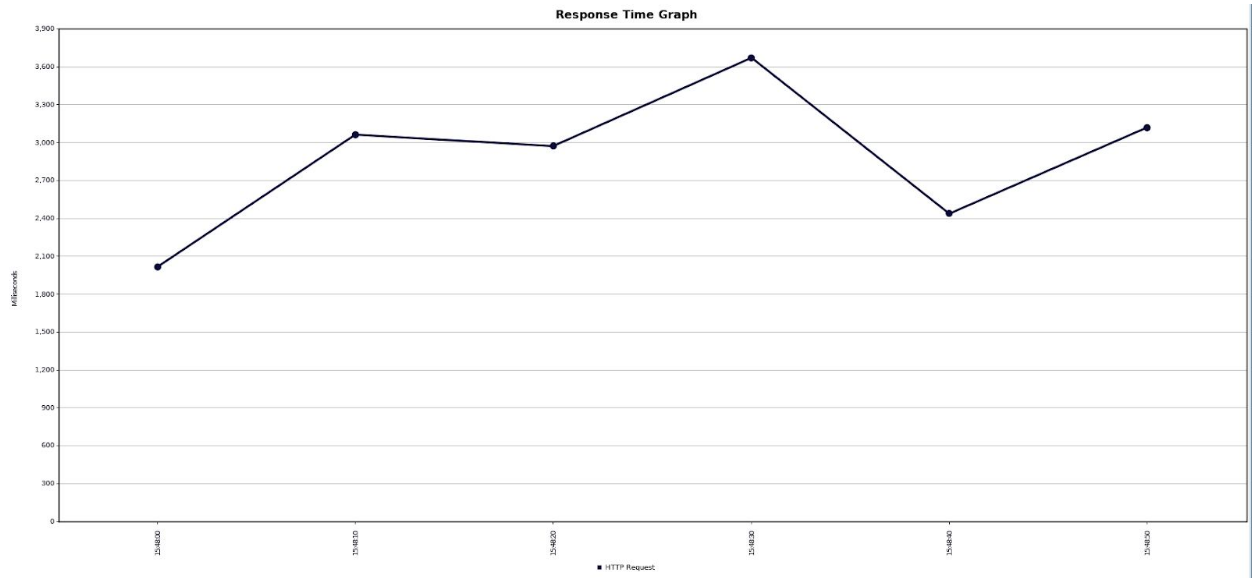
No of Samples 756
Deviation 693
Latest Sample 3517
Throughput 712.155/minute
Average 3774
Median 3922



Scenario 1 testing with Uniform Random timer



No of Samples 987
Deviation 848
Latest Sample 4514
Throughput 921.87/minute
Average 3084
Median 2981



Comparing experiments

Experiment	Throughput	Response Time
CPU 0.1 (“Default”)	Throughput starts out at ~700 requests per minute but rises slightly to ~1100 req/minute	Rises to ~3600ms, plateaus for 20s and then drops again
CPU 0.2	Throughput is slightly higher at the beginning at ~1300, and plateaus at 1500 req/minute	Starts out at ~1800ms; peaks at ~2100ms after 10s and gradually decreases
CPU 0.5	Throughput rises sharply at the start to 4500 req/minute and then continues to rise gently until it peaks at 4900 req/minute.	Rises sharply in the first 10s from 300 to 780ms and stays between 700 and 800 ms for the rest of the test.
Constant timer	Throughput rises sharply in the beginning and continues to rise until it plateaus at ~850 req/minute.	Starts at ~3200ms and sharply decreases in the next 10s to 600ms, then rises again to 3500ms in the next 20 seconds. Stays at above 3000ms for the rest of the test.
Gaussian Timer	Throughput rises rapidly to ~1000 requests/minute but then dips and slowly decreases to ~860	Starts at ~2800ms and rises to 4200ms in the next 20 seconds. Stays near this value for the rest of the test.
Uniform Timer	Throughput rises rapidly to ~900 requests/ms and continues to rise slightly as time goes on.	Rises from ~2000ms to ~3700ms over 40s, then stays between 3200ms and 2400ms for the rest of the test.

Throughput values tended to increase as available CPU increased, with 0.1 producing the lowest peak throughput of 1100 req/minute and 0.5 producing the highest peak throughput of 4900 req/minute. Predictably, the response time decreased as CPU limits increased, with 0.1 having the greatest response time peaking at 3600 ms and 0.5 having the least peak response time of ~800ms. This is expected because the greater proportion of CPU there is available, the more requests can be handled at once (i.e. in parallel).

Both throughput and response time changed depending on the timer used.

In the “default” i.e. no-timer case, the throughput rose steadily to 1100 req/min. On the other hand, throughput with a constant timer rose sharply in the beginning and plateaued at 1300 req/min, while throughput with a Gaussian timer rose rapidly to ~1000 req/min and slowly decreased after that. Throughput with the uniform timer rose rapidly to ~900 req/min and continued to rise slightly as time went on.

A timer with a constant break in between sending requests means that there is more chance of a request exiting the queue before a new request arrives. This likely why the maximum throughput with a constant timer was greater than without.

A Gaussian timer uses a Gaussian (Normal) distribution, which means that the time in between requests varies in the shape of a bell curve i.e. it is more likely to wait for values close to the ‘mean’ (300ms) than values far away from it (e.g. 10ms). This means that occasionally there will be very short breaks between requests being sent and occasionally comparatively long intervals. This is why it is not significantly different from the results observed for the previous two scenarios.

A uniform timer uses a Uniform distribution as the basis for how long JMeter waits before sending another request. In a uniform distribution, there is an equal likelihood of getting any waiting time between the specified minimum and maximum value (0 and 300). This means that on average the waiting time between requests would be shorter than for the other two cases, which is why the peak throughput was less for testing with this timer.

Conclusion

In conclusion, it would be reasonable to state that the allocation of resources is important when considering how well services perform during times where many requests occur at once. As CPU increases both the number of requests that can be addressed per minute as well as the response time per request become more favourable i.e. more req/minute and lesser time to respond. Moreover the type of interval between requests has a significant impact on how the service will perform- there is a significant difference between requests arriving at a constant interval vs something which is closer to real life e.g. in the form of a probability distribution.