Prime App

Functional and Performance Testing Report

version 1.0

Table of Contents

1 Introduction

1.1 Purpose

This report's purpose is to summarise and discuss the method and results of the functional and performance tests undertaken for Prime App. Prime App is a containerised application-server made using the Flask microframework and Docker.

1.2 Environment

Testing was carried out on a system running 64-bit openSUSE Tumbleweed 20200618. The containerised application-server was hosted using Docker version 19.03.11, and functional testing was done using Firefox 77.01. Performance testing was done using Apache JMeter 5.3, alongside OpenJDK version 11.0.7.

A repository consisting of source code, JMeter test plans and results pertaining to Prime App is publicly available at https://github.com/rflambert/primeApp

2 Functional Testing

2.1 Code Coverage

The user has access to three routes, that is, ‘/’, ‘/isPrime/#’, and ‘/primesStored’. ‘/isPrime/#’ and ‘/primesStored’ are the only complicated routes and so will be the only ones covered here.

‘/primesStored’ only requires two test cases to achieve 100% code coverage. That is, when the list of primes is empty, and when the list of primes is not empty. The code for the primeStored function is shown in figure 1.

‘/isPrime/#’ has more branches than the previous route. Of note here is the external function used, “storePrime()”. This function has no branching pathways, so will always have 100% code coverage as long as it is called. That said, a test set with 100% code coverage for the isPrime function is given in table 1. The code for the isPrimes function is shown in figure 2.

Table 1. Test case for 100% code coverage of isPrime()

Route	Lines Executed
/isPrime/1	33 34 39
/isPrime/2	33 34 36 37
/isPrime/4	33 41 42
/isPrime/9	33 41 42
/isPrime/25	33 41 43 44 45 46
/isPrime/28	33 41 43 44 45 46
/isPrime/29	33 41 43 44 45 47 44 49 50

```
61  # Output all prime numbers stored in redis
62  @app.route('/primesStored')
63  def primesStored():
64      primes = json.loads(cache.get('primes'))
65      # Check if list is empty
66      if len(primes) == 0:
67          return 'No primes are stored'
68      # Return a string of all elements in the list
69      output = ''
70      for prime in primes:
71          output += str(prime) + ', '
72      return output
73
```

Figure 1. Snippet of app.py showing primesStored function.

```
30  # Output if the given number is a prime or not
31  @app.route('/isPrime/<int:number>')
32  def isPrime(number):
33      if number <= 3:
34          if number > 1:
35              # 2 or 3, is a prime
36              storePrime(number)
37              return '%d is prime' % number
38          else:
39              return '%d is not prime' % number
40      # Check for every possible factor using the 6k+i optimisation
41      elif number % 2 == 0 or number % 3 == 0:
42          return '%d is not prime' % number
43      i = 5
44      while i*i <= number:
45          if number % i == 0 or number % (i + 2) == 0:
46              return '%d is not prime' % number
47          i += 6
48      # No factors found, is a prime
49      storePrime(number)
50      return '%d is prime' % number
```

Figure 2. Snippet of app.py showing isPrime function.

3 Performance Testing

3.1 Test Method

Two scenarios were outlined for testing. Both scenarios were stress tests designed to poll a specific route on the application-server at a set interval for a minute duration. Each scenario's test was run utilising 50 separate threads, simulating 50 separate users.

These threads all begin simultaneously and have a lifetime of 60 seconds. In retrospect, the threads beginning concurrently results in data that does not accurately represent real world values. The implications of this design decision on the data is discussed in section 3.2. Ideally, the number of threads running should have increased over time, from 1 thread to the maximum of 50, and the duration of the test doubled to account.

18 separate test were undertaken in total, 9 for each scenario. These 9 tests were undertaken with a combination of 1 of 3 predetermined CPU utilisation values (5%, 10%, 30%), and 1 of 3 request intervals per thread (300 ms, 1000 ms, 5000 ms). Tests were run in increasing request intervals of scenario 1, followed by scenario 2, then repeated with increased CPU utilisation.

3.1.1 Scenario 1

Repeatedly make an HTTP GET request to the server at "/isPrime/2147483647", using 50 threads, each with a lifetime of 60 seconds.

3.1.2 Scenario 2

Make an HTTP GET request to "/isPrime/1" through to "/isPrime/100" using 1 thread. Following this, repeatedly make an HTTP GET request to "/primesStored", using 50 threads, each with a lifetime of 60 seconds.

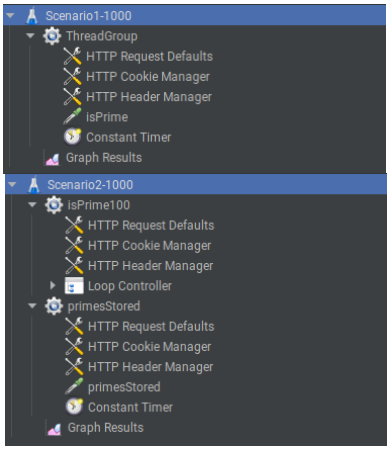

Figure 3. Test plan structure of scenario 1 with a 1000 ms interval (top), and scenario 2 with a 1000 ms interval (bottom)

3.2 Results

Figure 4 shows the median and standard deviation of the response times in each test, as well as the median throughput of all requests. It is clear that both a higher available CPU utilisation, and a longer interval between requests reduces the response time of the server, as well as the standard deviation. Doubling the CPU utilisation approximately halves the response time, as expected. Due to the large number of threads, changes to the interval between requests made has a much smaller affect on the response time.

Test	Scenario	CPUs	Interval (ms)	Median (ms)	StDev (ms)	Med. Throughput (B/s)
1	1	0.05	300	13296.00	3037.44	24.52
2	1	0.05	1000	12697.00	2692.23	25.68
3	1	0.05	5000	8994.00	2016.61	36.25
4	1	0.1	300	5994.00	954.07	54.39
5	1	0.1	1000	5492.00	839.50	59.36
6	1	0.1	5000	1600.00	856.64	203.75
7	1	0.3	300	1496.00	153.19	217.91
8	1	0.3	1000	902.00	120.01	361.42
9	1	0.3	5000	82.50	327.66	3951.66
10	2	0.05	300	4603.00	1666.22	69.52
11	2	0.05	1000	4002.00	1459.80	79.96
12	2	0.05	5000	94.00	735.88	3382.98
13	2	0.1	300	2303.00	665.08	138.95
14	2	0.1	1000	1690.00	478.93	189.35
15	2	0.1	5000	6.00	395.46	53333.33
16	2	0.3	300	704.00	141.90	454.55
17	2	0.3	1000	7.00	61.65	45714.29
18	2	0.3	5000	7.00	129.63	45714.29

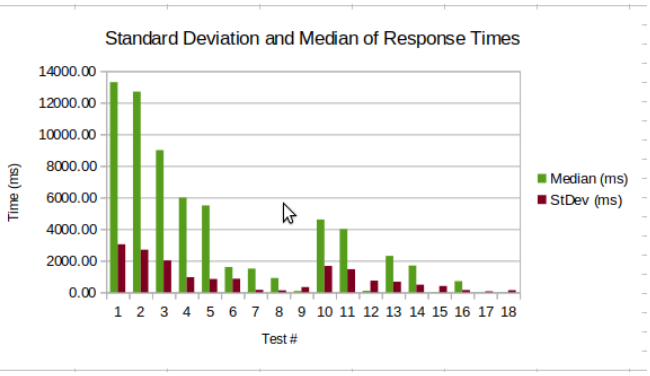

Figure 4. Summary of performance test data

The second test scenario overall returns a response faster than the first. This is partly due to the second scenario test cases including the low response time values of the ‘/isPrime/1’ through ‘/isPrime/100’ requests, however it is primarily due to the much lower processing time needed on the server side for scenario 2.

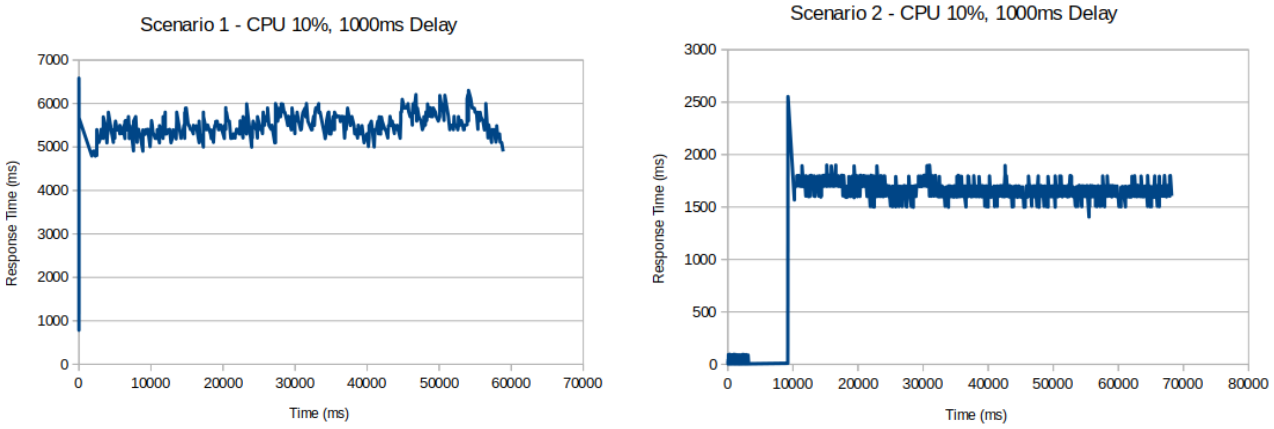

Figure 5. Response time graphs of test 5 and test 14

Show in figure 5 are the response time graphs for both scenario 1 and scenario 2, with 10% CPU utilisation and a request interval of 1000 milliseconds. The first 9 seconds of scenario 2 represent the ‘/isPrime/#’ requests being made to setup preconditions for the stress test. The proceeding 60 seconds clearly shows the lower response time of scenario 2 compared to scenario 1.

The effect of starting each thread simultaneously as previously discussed can be identified here as the large spike, more visible in scenario 2’s graph. In actuality this spike is a section comprised of many different requests with extremely high variance, as the server is unable to keep up with the large amount of requests being sent at once. The variance settles as the server sends staggered responses back, thus meaning the test starts sending requests at a more manageable rate.

Test data did not contain a large variance between request sizes. The minimum amount of bytes sent in a single request was 317, while the maximum amount of bytes send was 326. This means that the values for throughput per test are primarily a result of the differing response times. Therefore, they follow a similar pattern, with throughput being much higher with increased CPU utilisation and increased interval times.