COMPX341-20A

Assignment 5 - Dominic Maas – 1344405

Contents

[Testing Environment 1](#_Toc43820696)

[Server 1](#_Toc43820697)

[Client 1](#_Toc43820698)

[Testing Methodology 1](#_Toc43820699)

[Scenario 1 1](#_Toc43820700)

[Scenario 2 1](#_Toc43820701)

[Testing Scenario 1 2](#_Toc43820702)

[No Limits (Baseline) 2](#_Toc43820703)

[50% Single CPU Limit 2](#_Toc43820704)

[1% Single CPU Limit 2](#_Toc43820705)

[Testing Scenario 2a 2](#_Toc43820706)

[No Limits (Baseline) 2](#_Toc43820707)

[50% Single CPU Limit 2](#_Toc43820708)

[1% Single CPU Limit 2](#_Toc43820709)

[Testing Scenario 2b 3](#_Toc43820710)

[No Limits (Baseline) 3](#_Toc43820711)

[50% Single CPU Limit 3](#_Toc43820712)

[1% Single CPU Limit 3](#_Toc43820713)

[Analysis 3](#_Toc43820714)

# Tests

I originally planned to have tests integrated into the project, but I could not figure out how to do relative python imports within the docker container.

## Black Box Tests

These tests are run using the Flask testing framework (to test individual HTTP requests and their results).

1. Navigate to **/clear** (check that the result is “Cleared!”)
2. Navigate to a known prime number **/isPrime/2** ensure response is “2 is prime”
3. Navigate to **/primesStored** and ensure there is only one response “2” (match against the expected HTML)
4. Navigate to a known non-prime number **/isPrime/1** and ensure the response is “1 is not prime”
5. Finally navigate to **/primesStored** and ensure there is still only one response

## White Box Tests

These tests are running using pytest that interact with the methods that Flask uses to generate the routes. The following methods would be tested on docker image build (to ensure no images can be built with failing tests):

* is\_number\_prime(number)
* push\_onto\_cache(number)
* get\_from\_cache()

# Testing Environment

The following hardware was used to run the tests. Originally the tests were to be run on a different PC (using docker with the WSL2 backend), but there were socket connection issues while running JMeter. Instead tests were run from a Windows 10 PC across a local network (via gigabyte wired connection) to a separate server (running Ubuntu Server).

## Server

* **CPU**: AMD Ryzen 5 3400G (8) @ 3.7GHz
* **Memory**: 32GB DDR4
* **Operating System:** Ubuntu 20.04 LTS x86\_64
* **Kernel:** 5.4.0-37-generic
* **Docker:** 19.03.8, build afacb8b7f0

## Client

* **JMeter:** 5.3
* **Java:** 1.8.0

# Testing Methodology

Before running any tests, ensure “docker-compose down” is run first to generate a blank testing slate (otherwise results from previous tests may impact the currently running test).

## Scenario 1

1. Ensure all limits are removed from the “docker-compose.yml” file
2. Run the following commands: “docker-compose down”, “docker-compose build”, “docker-compose up”. This will ensure a clean slate with the latest changes
3. Open “scenario\_1.jmx” within JMeter (5.3 was used for the following results)
4. Click on “Scenario 1” then click on the green start button to begin performance testing
5. Repeat steps 2-4 adding a CPU limit of 0.5 to the “docker-compose.yml” file
6. Repeat steps 2-4 adding a CPU limit of 0.01 to the “docker-compose.yml” file

Once complete, you should have 3 sets of data, one for no CPU limit, one for 0.5 CPU limit and a final dataset for 0.01 CPU limit. CPU limits are defined as follows: 0.5 -> 50% of a single core.

## Scenario 2

1. Ensure all limits are removed from the “docker-compose.yml” file
2. Run the following commands: “docker-compose down”, “docker-compose build”, “docker-compose up”. This will ensure a clean slate with the latest changes
3. Open “scenario\_2.jmx” within JMeter (5.3 was used for the following results)
4. Click on “Scenario 2” then click on the green start button to begin performance testing (the thread groups will be run in sequential order)
5. Repeat steps 2-4 adding a CPU limit of 0.5 to the “docker-compose.yml” file
6. Repeat steps 2-4 adding a CPU limit of 0.01 to the “docker-compose.yml” file

Once complete, you should have 6 sets of data, two for no CPU limit, two for 0.5 CPU limit and a final two datasets for 0.01 CPU limit. CPU limits are defined as follows: 0.5 -> 50% of a single core.

# Testing Scenario 1

## No Limits (Baseline)

* **Throughput:** 149.25 / second
* **Response Time:**
  + **Average:** 333ms
  + **Min:** 28ms
  + **Max:** 380ms

## 50% Single CPU Limit

* **Throughput:** 68.08 / second
* **Response Time:**
  + **Average:** 730ms
  + **Min:** 37ms
  + **Max:** 873ms

## 1% Single CPU Limit

* **Throughput:** 1.05 / second
* **Response Time:**
  + **Average:** 37,943ms
  + **Min:** 1,973ms
  + **Max:** 54,696ms

# Testing Scenario 2a

## No Limits (Baseline)

* **Throughput:** 668.41 / second
* **Response Time:**
  + **Average:** 74ms
  + **Min:** 7ms
  + **Max:** 107ms

## 50% Single CPU Limit

* **Throughput:** 243.82 / second
* **Response Time:**
  + **Average:** 204ms
  + **Min:** 71ms
  + **Max:** 2,332ms

## 1% Single CPU Limit

* **Throughput:** 3.16 / second
* **Response Time:**
  + **Average:** 14,091ms
  + **Min:** 23ms
  + **Max:** 18,099ms

# Testing Scenario 2b

## No Limits (Baseline)

* **Throughput:** 8.3 / second
* **Response Time:**
  + **Average:** 5,803ms
  + **Min:** 966ms
  + **Max:** 6,611ms

## 50% Single CPU Limit

* **Throughput:** 11.37 / second
* **Response Time:**
  + **Average:** 4,263ms
  + **Min:** 505ms
  + **Max:** 4,996ms

## 1% Single CPU Limit

* **Throughput:** 1.76 / second
* **Response Time:**
  + **Average:** 24,036ms
  + **Min:** 2,200ms
  + **Max:** 30,297ms

# Analysis

While running scenario 2 (b) I noticed that the baseline test performed worse that the 50% limited CPU test (in terms of average response time and throughput). Initially I thought I had some data mixed up, so I reran the no limit test and 50% limit test again and received the same results. I am unsure how to explain why I got these results.

In general, there are massive performance drops (throughput and latency) once CPU limits are put in place (implying that the web application is CPU limited vs memory limited\*). It can then be assumed that by increasing the hardware within the testing server (more specifically CPU power), we should get higher throughput (by lowering the amount of time spent calculating if a number is prime or not) as stated by Little’s Law.

Comparing the results between the 50% and 1% single threaded CPU tests allows us to perform more detailed analysis as the no limit test may be using multiple cores (the testing server had 4 cores / 8 threads). In Scenario 1 the latency increased from an average of 730ms to 37,943ms, while the throughput dropped from 149.25 per second to only 1 per second. This is heavily implying that the application is CPU limited in calculating the prime number. And since the web application is only running on a single thread, other clients must wait to be served (in a queue), which further increases the response time).

If this application were to be deployed, I’d recommend prioritising CPU speed over the amount of memory (possible by choosing CPU optimised virtual machines for example).

*\*Although the memory results are not written down, the web application did not use more than 30MB or RAM throughout the tests (using “docker stats”).*