LULEÅ
UNIVERSITY
OF TECHNOLOGY

M7024E Laboratory 5: Option 1 - Cloud Benchmarking

submitted by

Md Asif Mahmod Tusher Siddique, Muiz Olalekan Raheem

January 8, 2023

submitted to

Dr. Karan Mitra

1 Objectives

The objective of this lab is to:

• Understand concepts regarding Cloud benchmarking.

• Using widely available tools for application performance benchmarking such as Httpperf1 and Apache JMeter2 benchmark an application.

2 Exercises

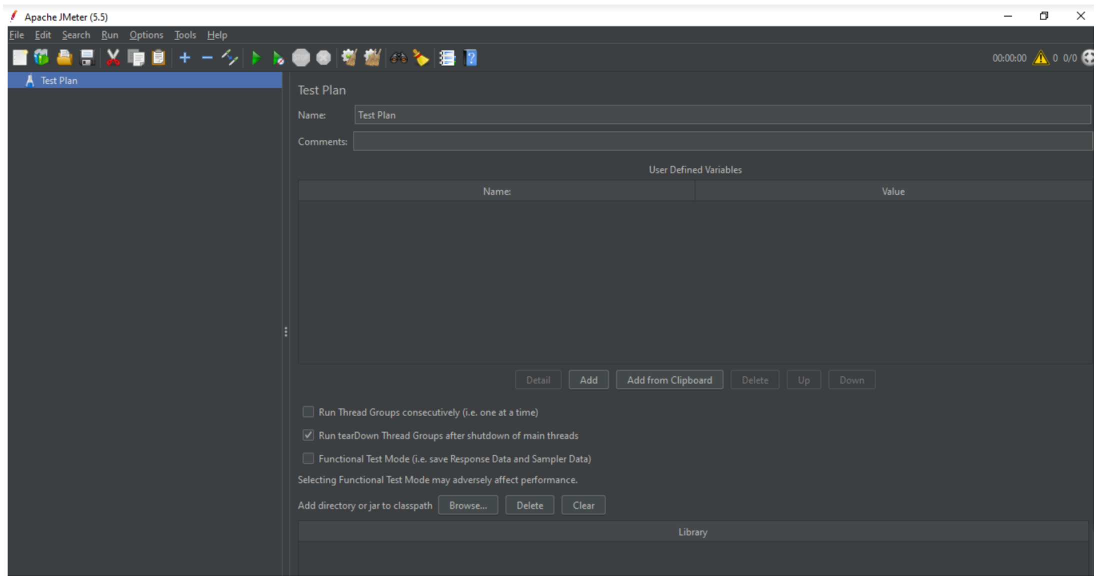
Exercise a: In this exercise, you will learn how to benchmark a simple Web application (it can be your RESTFul service). You will generate the syn- thetic workload (emulating a number of users making several requests per second, see lecture slides) and benchmark the performance of your application.

1. Download and install Apache JMeter on your lab machine/laptop/Cloud instance. Follow the instructions given on the "Getting Started" page on the Apache JMeter website or anywhere on the WWW. Stick to the GUI mode for running JMeter.
2. As a previous lab exercise, you created a Webpage and served it via an application server such as Apache Webserver. Test the performance of your Web service using JMeter. It is also OK to run your application and JMeter on the same machine, if you want.
   (a) Build a "Test Plan" to emulate a number of users, say 10, 50, 100,..., or more. Follow chapter 4 of the "User's Manual" which explains how to test a Web service.
   (b) Execute your "Test Plan" by varying the number of users and the "ramp-up" period.
   (c) Use the Table/Graph to analyze the results and document them.
      i. Explain in detail your "Test plan". For example, explain (via table) the parameters that you used, their relevance and settings.
      ii. Analyze and document the performance of your application w.r.t the load you generate via JMeter. For instance, what is the average number of requests/sec. that saturates the application server, the

latency, the throughput and other relevant metrics that you find useful.

Solution:

In the first step we downloaded and installed Apache JMeter in one of our laptop.



Test Plan

Name: Test Plan

Comments:

☐ Run Thread Groups consecutively (i.e. one at a time)

☑ Run tearDown Thread Groups after shutdown of main threads

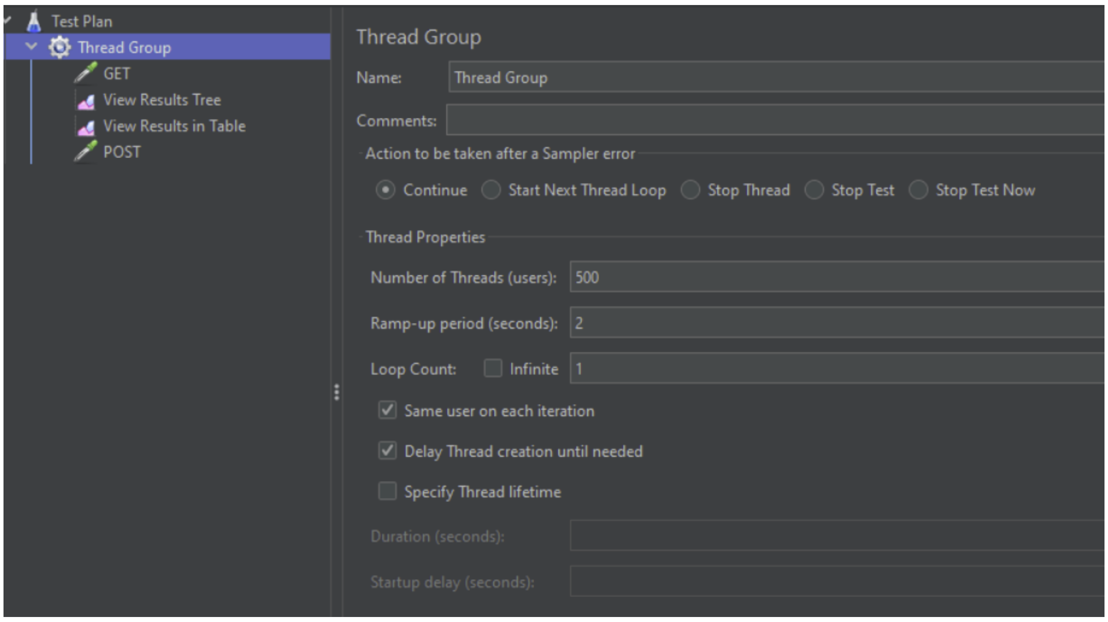
☐ Functional Test Mode (i.e. save Response Data and Sampler Data)

Selecting Functional Test Mode may adversely affect performance.

Add directory or jar to classpath

Figure 1: Apache JMeter

In the next step, We started by creating a thread group with 500 users and a ramp-up period of 2 seconds.


Test Plan
Thread Group
GET
View Results Tree
View Results in Table
POST

Thread Group

Name: Thread Group

Comments:

Action to be taken after a Sampler error

[✓ Continue] Start Next Thread Loop Stop Thread Stop Test Stop Test Now

Thread Properties

Number of Threads (users): 500

Ramp-up period (seconds): 2

Loop Count: ☐ Infinite 1

☑ Same user on each iteration

☑ Delay Thread creation until needed

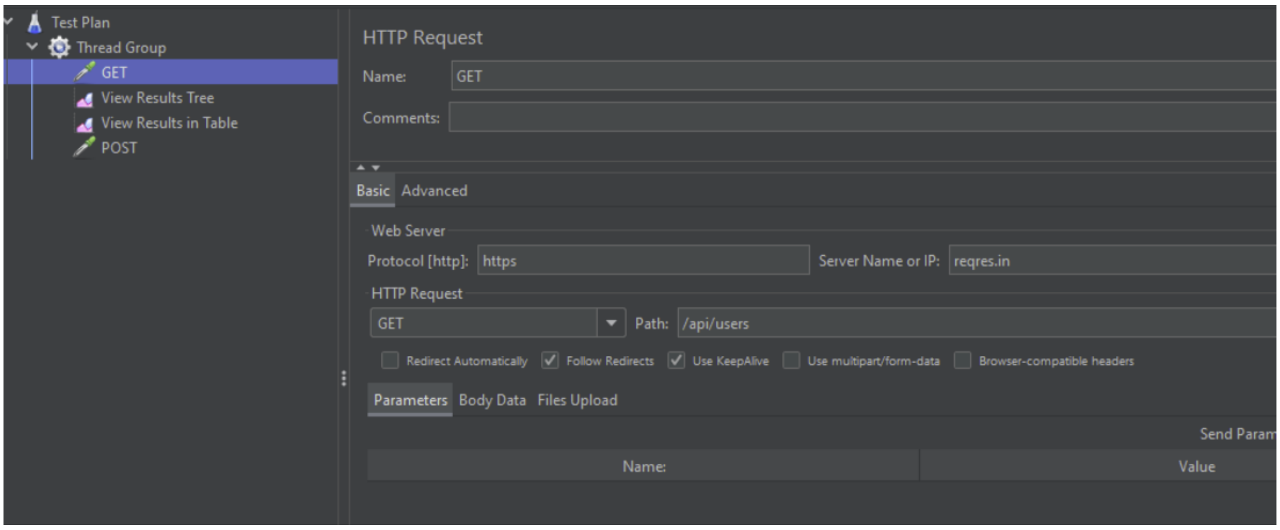
☐ Specify Thread lifetime

Duration (seconds):

Startup delay (seconds):

Figure 2: Creating Thread group

Then, to process GET request we used the following configuration:


Test Plan
Thread Group
GET
View Results Tree
View Results in Table
POST

HTTP Request

Name: GET

Comments:

Basic Advanced

Web Server

Protocol [http]: https Server Name or IP: reqres.in

HTTP Request

GET Path: /api/users

☐ Redirect Automatically ☑ Follow Redirects ☑ Use KeepAlive ☐ Use multipart/form-data ☐ Browser-compatible headers

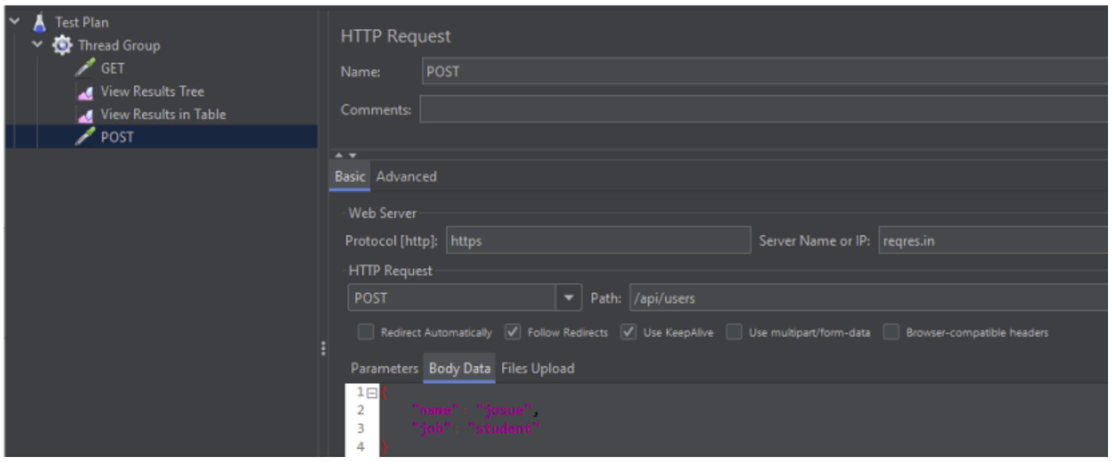
Parameters Body Data Files Upload

Send Param

Name:	Value

Figure 3: Configuration of GET request

To configure for the POST request we did the following:


Test Plan
Thread Group
GET
View Results Tree
View Results in Table
POST

HTTP Request

Name: POST

Comments:

Basic | Advanced

Web Server

Protocol [http]: https   Server Name or IP: reqres.in

HTTP Request

POST   Path: /api/users

[ ] Redirect Automatically [✓] Follow Redirects [✓] Use KeepAlive [ ] Use multipart/form-data [ ] Browser-compatible headers

Parameters | Body Data | Files Upload

```
{
    "name": "josue",
    "job": "student"
}
```

Figure 4: Configuration of POST request

After making 500 calls to the API using both GET and POST methods with a 2-second ramp-up period for each call, all the HTTP responses were correct. This indicates that the API was able to handle the stress test. The elapsed time for the GET method was longer than the POST method, and there was also more variance in the response time for the GET method.

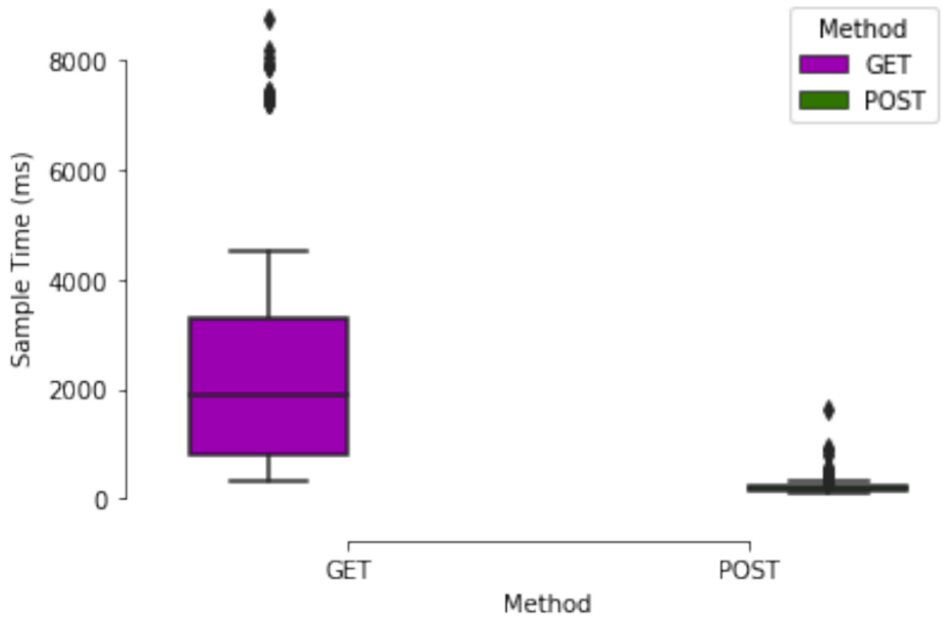


Figure 5: HTTP request time

In case of latency we observed the same behavior. POST method had far less vriance as well as dispersion.

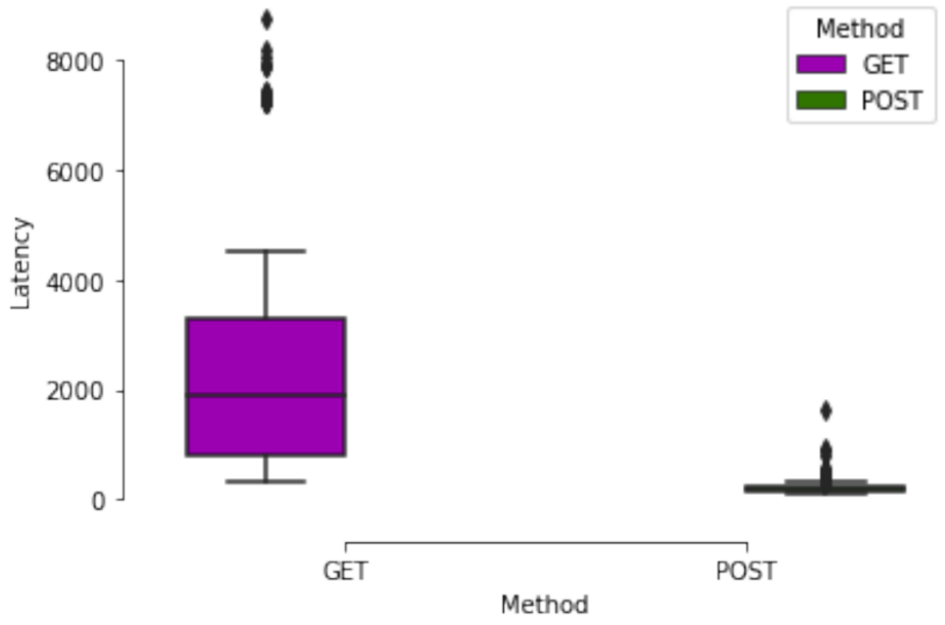


Figure 6: HTTP request latency

The mean response time was higher for the GET method, but the payload for the GET method was also larger than the payload for the POST method.

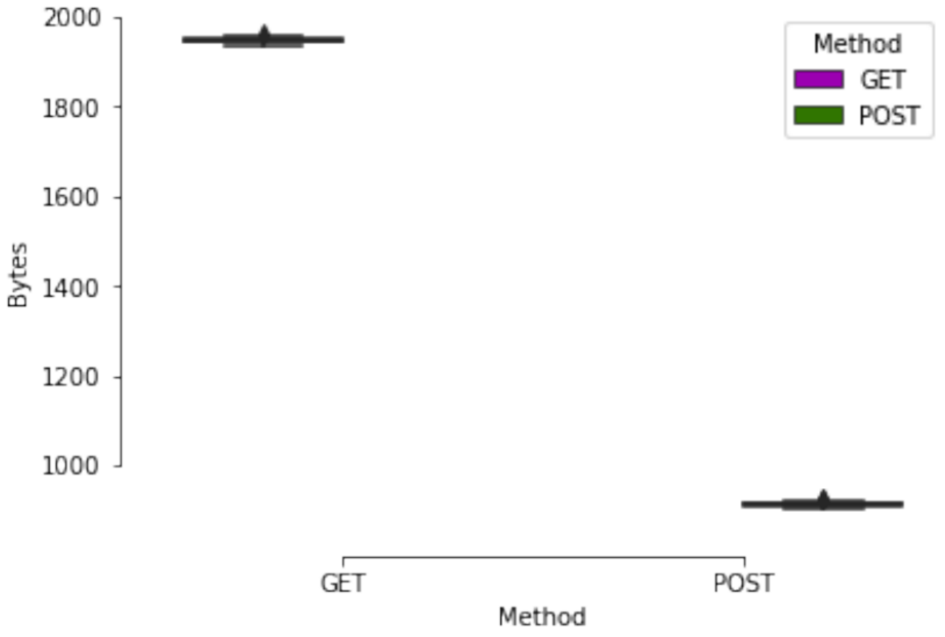


Figure 7: Total Byte Traffic

Method	Bytes (Mean)	Mean Sample time (ms)	Mean Throughput (bps)
Get	1944.23	1999.453	798.456
POST	878.45	178.673	43546.463

Table 1: Calculation of mean throughput

After discovering that the payload size had a significant impact on the response time, we calculated the mean throughput per request.

The benchmarking test using JMeter was successful in generating a realistic workload and producing traces. However, when we attempted to send more than 1000 requests, JMeter crashed instead of the API. This prevented us from being able to determine the API's failure point. As mentioned in section 5, the throughput for POST was almost 5 times higher than for GET. This could be due to the time it takes to retrieve the requested data from the database using the GET method.