CS6650 Assignment 1: Load Testing Client-Server (Java/GoLang)

Rahul Arun Kumar

The client consists of 3 Classes: Client1, Client2 and RequestHandler.

➔ **Client1**

Client1 is a Java client application designed for making HTTP requests to a server. It creates multiple threads to send a large number (100/1000) of HTTP requests concurrently to test the server's performance.

Two packages to take note of that it uses are the Apache HTTP-Client and Java's concurrent package. I chose to use apache's client package as it simplifies the idea of concurrency with requests with objects such as PoolingHttpClientConnectionManager. It also allows me to handle sending requests with much more detail as opposed to net.HTTP, as there are specific objects for GET and POST requests and to also introduces the object of ClosableClients, which help me manage how to handle responses, given they don't really matter much in the context of this assignment.

Concurrency wise we have the use of Executors and AtomicIntegers. AtomicIntegers helps me keep track of how many requests are being sent, which is useful for throughput calculations and will not be modified by any thread that is not meant to interact with it. Executors provide a higher-level, simplified way to manage and execute tasks concurrently using a thread pool, which works well with PoolingHttpClientConnectionManager.They are designed to abstract away the low-level details of thread management, making it easier to work with multi-threading and concurrency in Java.

The design of Client1 is meant to encapsulate away the logic as class methods, and call them when necessary. It is able to compartmentalize the logic into different segments and utilize them as needed. The working of Client1 is as follows:

- createHttpClient:
  A method to create and configure an HTTP client using Apache HttpClient library. It uses a connection manager to control connection pooling and limits the maximum number of connections.
- startInitialThreads:
  Starts a group of initial threads that make both POST and GET requests. These threads run concurrently and make a specified number of requests each. The ExecutorService is used to manage these threads.

- startAdditionalThreads:
  Starts the load testing of the threads by making POST and GET requests. These threads run concurrently and make a specified number of requests each. The method also takes a delay parameter to control the gap between requests.
- performRequests:
  This method is responsible for making HTTP requests (GET or POST). It includes logic for retrying failed requests up to a maximum number of retry attempts. It returns the number of retry attempts made for the request.
- main():
  It takes command-line arguments which represent the groupSize, #ofThreads, delay and the IP Address. It calls all the aforementioned methods and get to implementing the solution to the problem. The client also measures the time it takes to complete all requests and calculates throughput.

➔ **Client2**

Client2 is an extended version of the Client1 class, with additional functionality for measuring and recording latency statistics of the HTTP requests. It utilizes the same packages in Client1 for the same purposes as described but makes use of the File package, not just to read the image data that we send but to create a CSV file for our data. Its design logic also matches that of Client1 as it is an extension of it. The only additional logic to Client2 is the calculation of Statistics with methods such as calculateAndDisplayStatistics which calls calculateMedian and calculateP99, which are self explanatory.

➔ **RequestHandler**

RequestHandler is responsible for handling HTTP GET and POST requests to a specified base URL using the Apache HttpClient library. The primary reason for this package is to make use of its MultipartEntityBuilder, which provides a much simpler interface to data fields to our request body, given that we are working with a file and JSON data. It also is encapsulated by design into several methods representing the logic of it, so that the functionality is compartmentalized. It’s primary purpose is to provide a method to perform a GET and/or POST requests:

- sendGetRequest:
  Sends an HTTP GET request to retrieve album information based on an album ID. It constructs the endpoint URL by appending the album ID to the base URL. Executes the GET request using the HTTP client. Calls handleResponse to process the HTTP response and extract the status code.
- sendPostRequest:
  Sends an HTTP POST request to create a new album record. It constructs the endpoint URL for creating albums and builds a JSON profile text for the album's artist, title, and year. It then constructs a MultipartEntityBuilder to handle file uploads (image) and the JSON profile. Finally, sets and executes the HTTP POST request's entity with the

multipart data. Likewise to sendGetRequest it returns the HTTP status code or -1 if an error occurs.

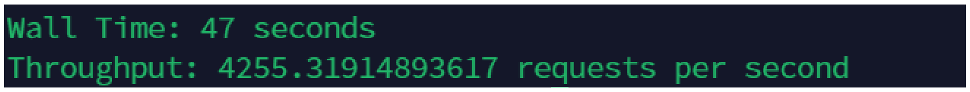
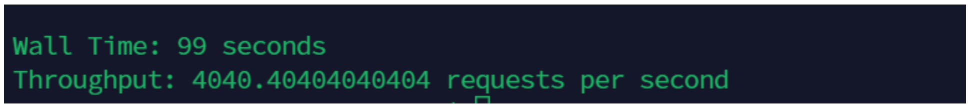
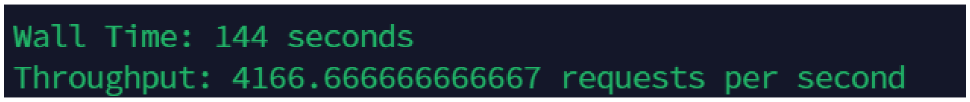
Additional Notes:

- The client and the server both run on different AWS instance, this was done due to:
    - My internet speed is so erratic that sometimes it works perfectly, while other times, the requests take too long to send
    - My internet base speed is too slow to get good throughput
    - Running them on the instances of the same definition, allows for a more concrete set of test results as the environment remains the same.
- To plot the final graph, I used Python and the other two were done thru Google Docs

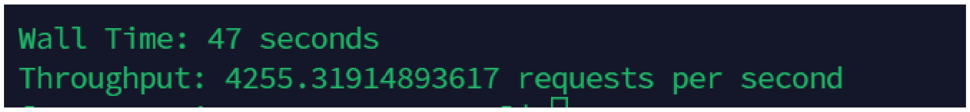
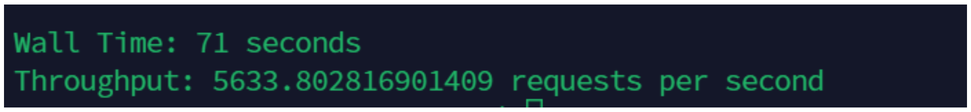
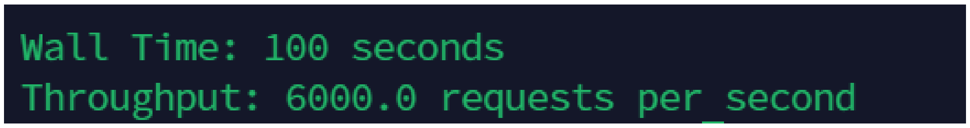
Link to Repo: https://github.com/rahulleoak/CS6650

Screenshots:

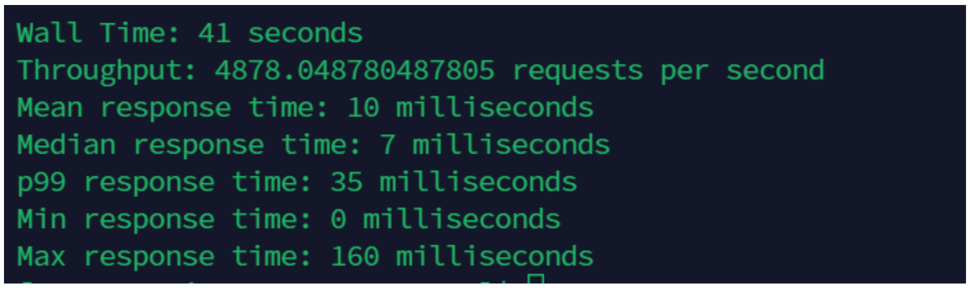
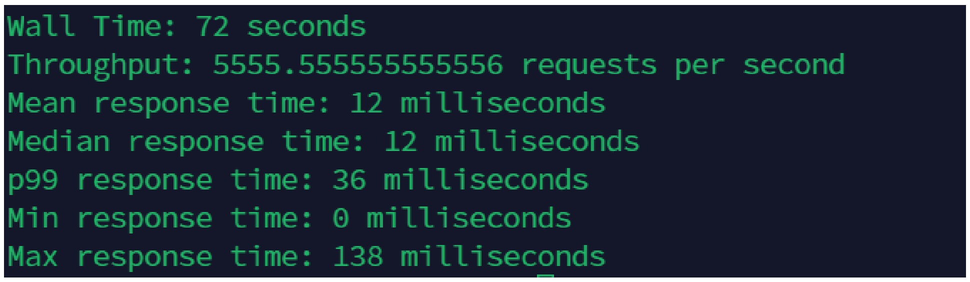
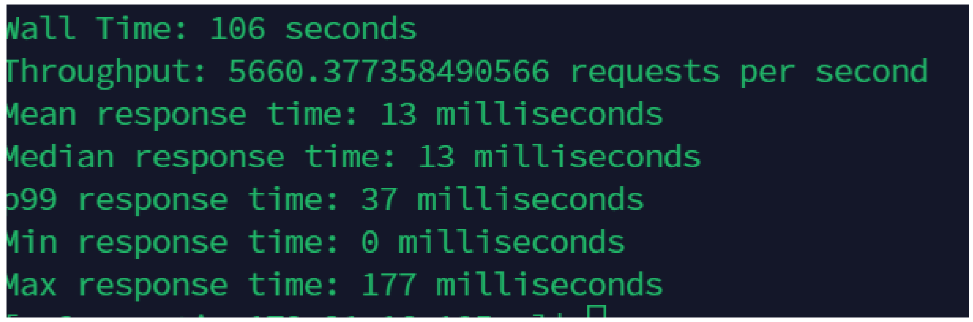
- Part 1
    - Java

10 Thread Groups	 `Wall Time: 47 seconds` `Throughput: 4255.31914893617 requests per second`
20 Thread Groups	 `Wall Time: 99 seconds` `Throughput: 4040.40404040404 requests per second`
30 Thread Groups	 `Wall Time: 144 seconds` `Throughput: 4166.666666666667 requests per second`

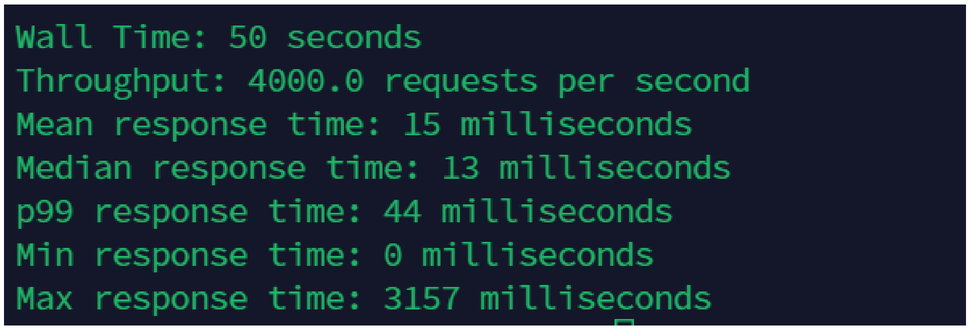
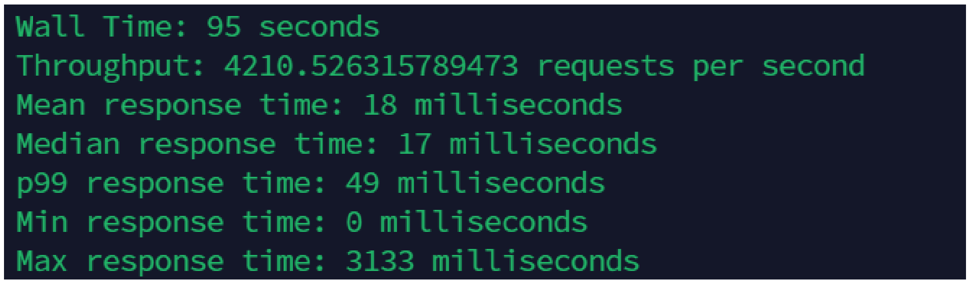
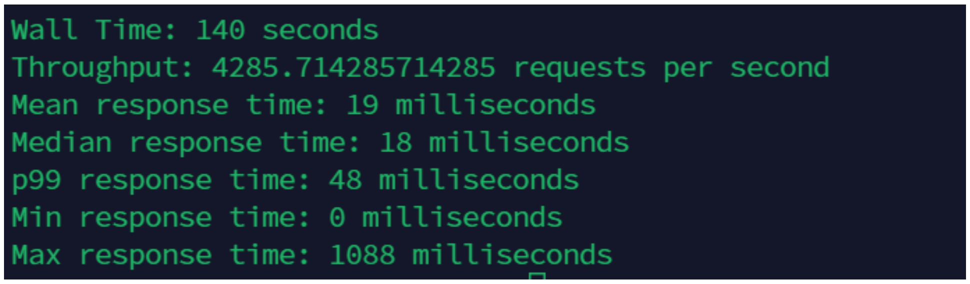
-
    - Go

10 Thread Groups	 `Wall Time: 47 seconds` `Throughput: 4255.31914893617 requests per second`
20 Thread Groups	 `Wall Time: 71 seconds` `Throughput: 5633.802816901409 requests per second`
30 Thread Groups	 `Wall Time: 100 seconds` `Throughput: 6000.0 requests per second`

- Part2
    - Java

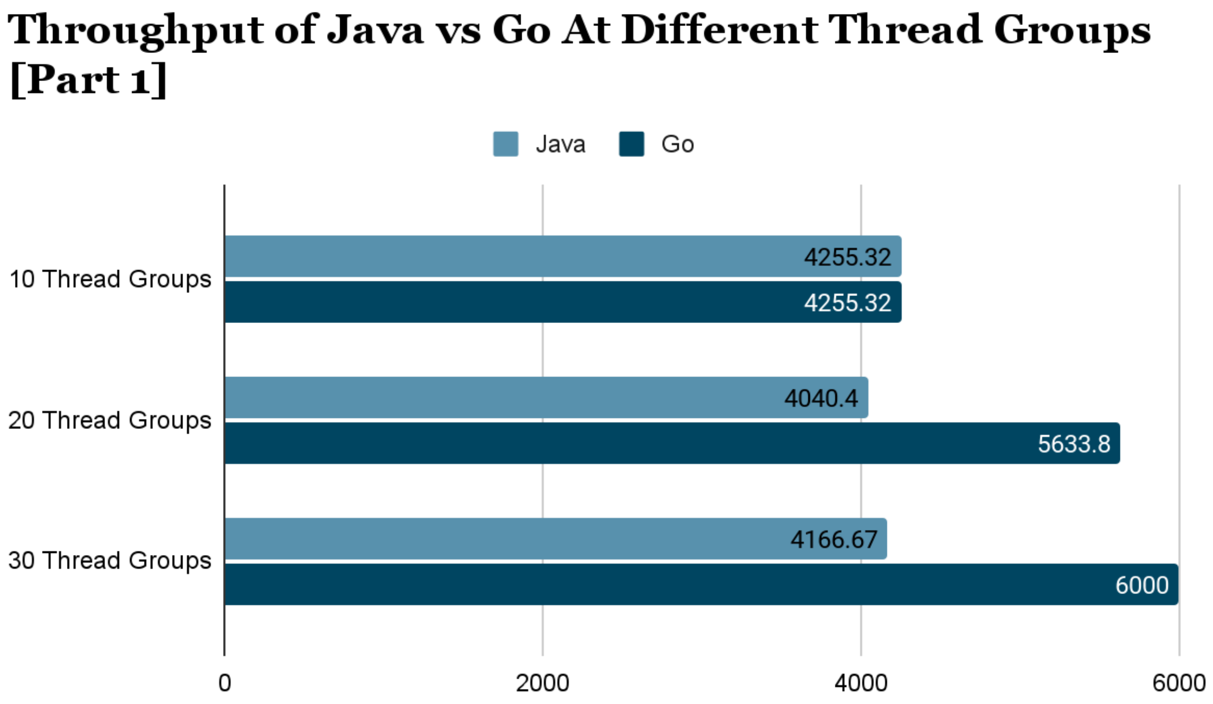
10 Thread Groups	 Wall Time: 41 seconds Throughput: 4878.048780487805 requests per second Mean response time: 10 milliseconds Median response time: 7 milliseconds p99 response time: 35 milliseconds Min response time: 0 milliseconds Max response time: 160 milliseconds
20 Thread Groups	 Wall Time: 72 seconds Throughput: 5555.555555555556 requests per second Mean response time: 12 milliseconds Median response time: 12 milliseconds p99 response time: 36 milliseconds Min response time: 0 milliseconds Max response time: 138 milliseconds
30 Thread Groups	 Wall Time: 106 seconds Throughput: 5660.377358490566 requests per second Mean response time: 13 milliseconds Median response time: 13 milliseconds p99 response time: 37 milliseconds Min response time: 0 milliseconds Max response time: 177 milliseconds

- Go

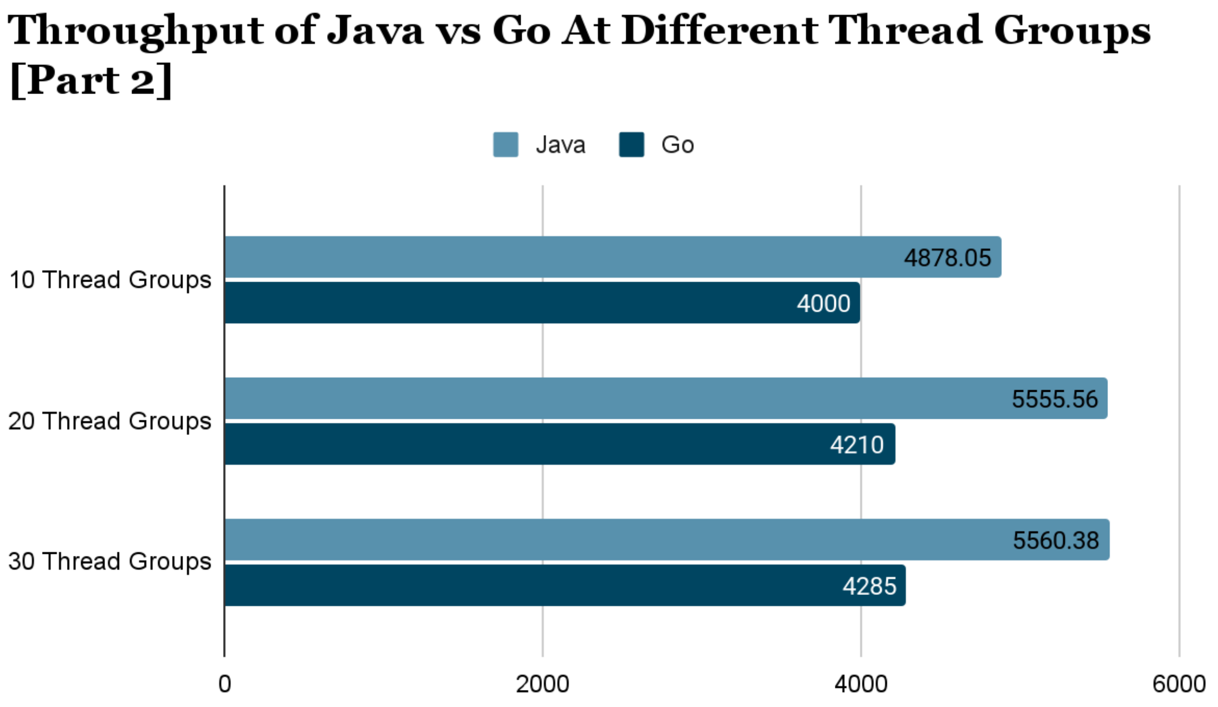
10 Thread Groups	 Wall Time: 50 seconds Throughput: 4000.0 requests per second Mean response time: 15 milliseconds Median response time: 13 milliseconds p99 response time: 44 milliseconds Min response time: 0 milliseconds Max response time: 3157 milliseconds
20 Thread Groups	 Wall Time: 95 seconds Throughput: 4210.526315789473 requests per second Mean response time: 18 milliseconds Median response time: 17 milliseconds p99 response time: 49 milliseconds Min response time: 0 milliseconds Max response time: 3133 milliseconds
30 Thread Groups	 Wall Time: 140 seconds Throughput: 4285.714285714285 requests per second Mean response time: 19 milliseconds Median response time: 18 milliseconds p99 response time: 48 milliseconds Min response time: 0 milliseconds Max response time: 1088 milliseconds

Graphs:

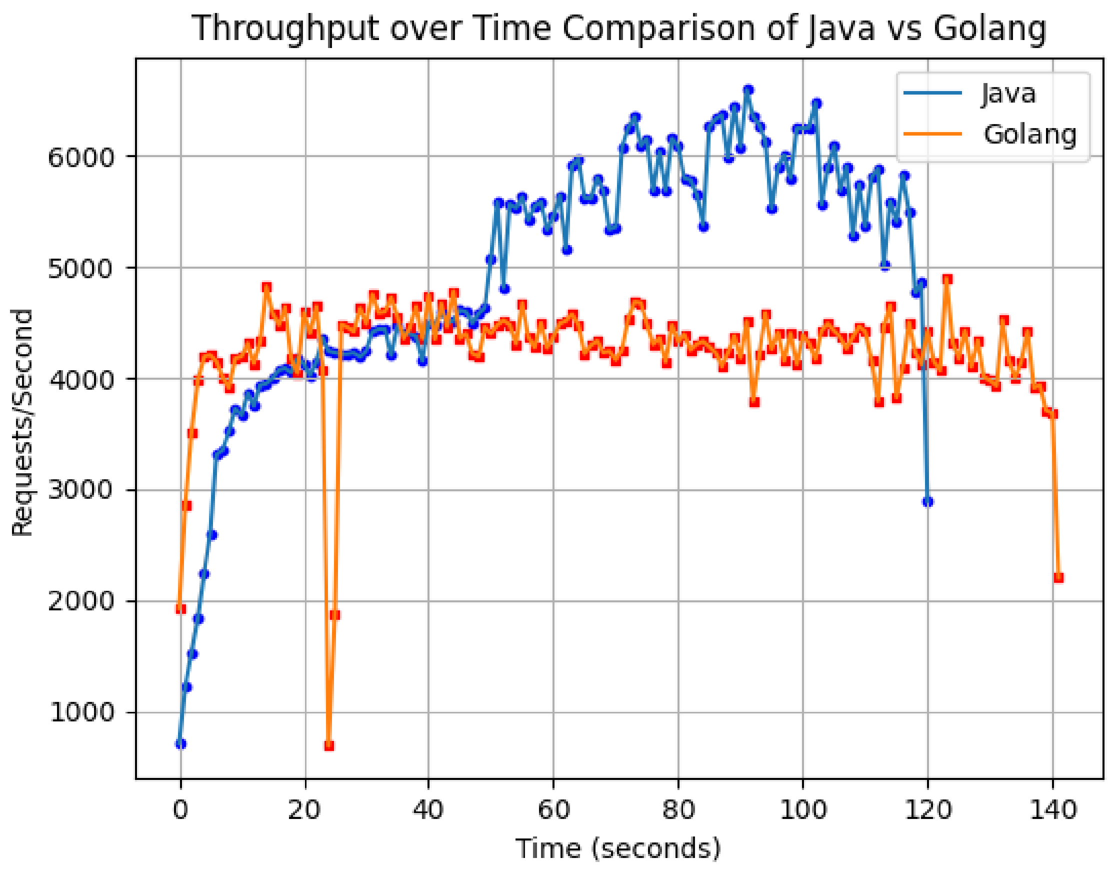
- Throughput Comparison in Part 1



- Throughput Comparison in Part 2



- Throughput Over Time on Part 2 for the test case:
  - threadGroupSize = 10,
  - numThreadGroups = 30,
  - delay = 2


Throughput over Time Comparison of Java vs Golang
Java
Golang
Requests/Second
Time (seconds)