Client Description

This program is for CS6650 assignment 1. It utilizes HttpClient to build a muti-thread client to make POST requests to a specified URL, and test the performance of the server.

The repository url: https://github.com/tjuqxb/6650-assignment-1

Server url: might change due to restart instance

How it works

The program entry point Class is ClientService. The typical way of running the program (both part 1 and part 2) is specifying the parameters in parameters.xml file (another way is using command line). The user needs to sepcify:

- arg0: maximum number of threads to run (numThreads - max 1024)
- arg1: number of skier to generate lift rides for (numSkiers - max 100000), This is effectively the skier's ID (skierID)
- arg2: number of ski lifts (numLifts - range 5-60, default 40)
- arg3: mean numbers of ski lifts each skier rides each day (numRuns - default 10, max 20)
- arg4: IP address and port number of the server

Part 1 client would only produce output window results. Part 2 client would produce output window results, a CSV file and a time graph. Users can use class ImageCreator to create other graphs based on output window results.

Main classes

ClientService: This is the entry point Class for the program. It would accept user specified paramters, generate multiple SingleClientThread threads, and use a thread pool and multiple CountDownLatch to control the whole process of the 3 phases. After execution of all the generated threads, it would gather information(the thread results are returned using Future) and generate the final results.
SingleClientThread: This is the class of a single thread client using HttpClient to make multiple requests to specified URL (or generated URLs). It contains one instance of URLBuilder, which is responsible for generating dynamic POST URLs, and one instance of RandInfoCreator, which is responsible for generate random information for a POST. It would record the number of successful and failed requests, the latency of each record, the maximum latency and the minimum latencies and the start and end time (implemented in part 2).
URLBuilder: This class can build URL string based on input parameters.
RandInfoCreator: This class can generate random information and create a SkierPost object based on the information.

ImageCreator: This is a utility class using JFreeChart to produce images.
Utils: This class contains static common utility functions.
Parameters: The wrapper class for mapping XML parameters.
ThreadInfo: The wrapper class for storing thread returned information, including the number of successful and failed requests, the latency of each record, the maximum latency and the minimum latencies and the start and end time (implemented in part 2).
SkierPost: This class can be mapped to a standard POST JSON object specified by the skier API.

Packages

domain: This package is used to store classes which can be mapped to standard POST JSON objects specified by the API. It currently contains one class: SkierPost.

Calculation of Throughput

Using single thread to test, the average latency is about 28 ms and the throughput is about (1000/28 = 35.7/s). Also, according to Jmeter test, the maximum throughput of the naive server is about 2500/s to 3000/s.
Assuming the number of max threads is N. The total number of requests is M.
There are total 3 phases, phase 1 would use 0.25N threads to send 2/9 M requests, phase 2 would use 0.6N threads to send 2/3 M requests and phase 3 would use 0.1N threads to send 1/9 M requests.
According to test, when 20% of total number of threads of one phase have finished, more than 95% of the total number of the requests of that phase also have been sent. (It is reasonable as all the threads of one phase would have the same priorities and at any point they would almost have sent the same number of requests). So, it is reasonable to calculate the wall time of the 3 phases and add them up for prediction.
There are total 180000 requests. Use average latency = 28 ms to estimate.
Using Little's law, we can get a function: N = Throughput * W (NumberOfThreads = Throughput * Latency).
Throughput = NumberOfThreads / Latency

For 32 threads:
phase1-throughput = 32 * 0.25 * 1000 / 28 = 285
phase2-throughput = 32 * 1000 / 28 = 1143
phase3-throughput = 32 * 0.1 * 1000 / 28 = 114
walltime = 40000 / 285 + 120000 / 1143 + 20000 / 114 = 420.8s
throughput = 180000/420.8 = 428
For 64 threads:
phase1-throughput = 64 * 0.25 * 1000 / 28 = 570
phase2-throughput = 64 * 1000 / 28 = 2286
phase3-throughput = 64 * 0.1 * 1000 / 28 = 228
walltime = 40000 / 570 + 120000 / 2286 + 20000 / 228 = 210.4s
throughput = 180000/210.4 = 856

For 128 threads:

phase1-throughput = 128 * 0.25 * 1000 / 28 = 1140

phase2-throughput = min{128 * 1000 / 28, 2500} = max {4572, 2500} = 2500

phase3-throughput = 128 * 0.1 * 1000 / 28 = 456

walltime = 40000 / 1140 + 120000 / 2500 + 20000 / 456 = 126.9s

throughput = 180000/126.9 = 1418

For 256 threads:

phase1-throughput = 256 * 0.25 * 1000 / 28 = 2280

phase2-throughput = min{256 * 1000 / 28, 2500} = max {9144, 2500} = 2500

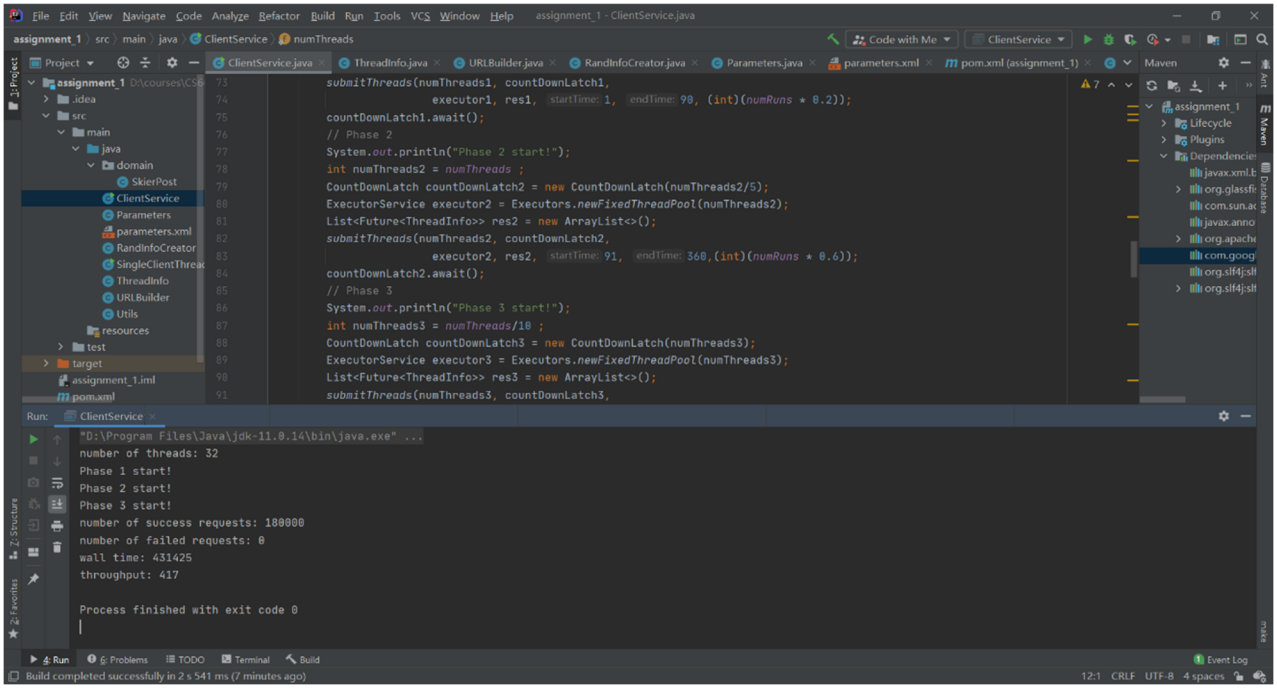
phase3-throughput = 256 * 0.1 * 1000 / 28 = 912

walltime = 40000 / 2280 + 120000 / 2500 + 20000 / 912 = 87.5s

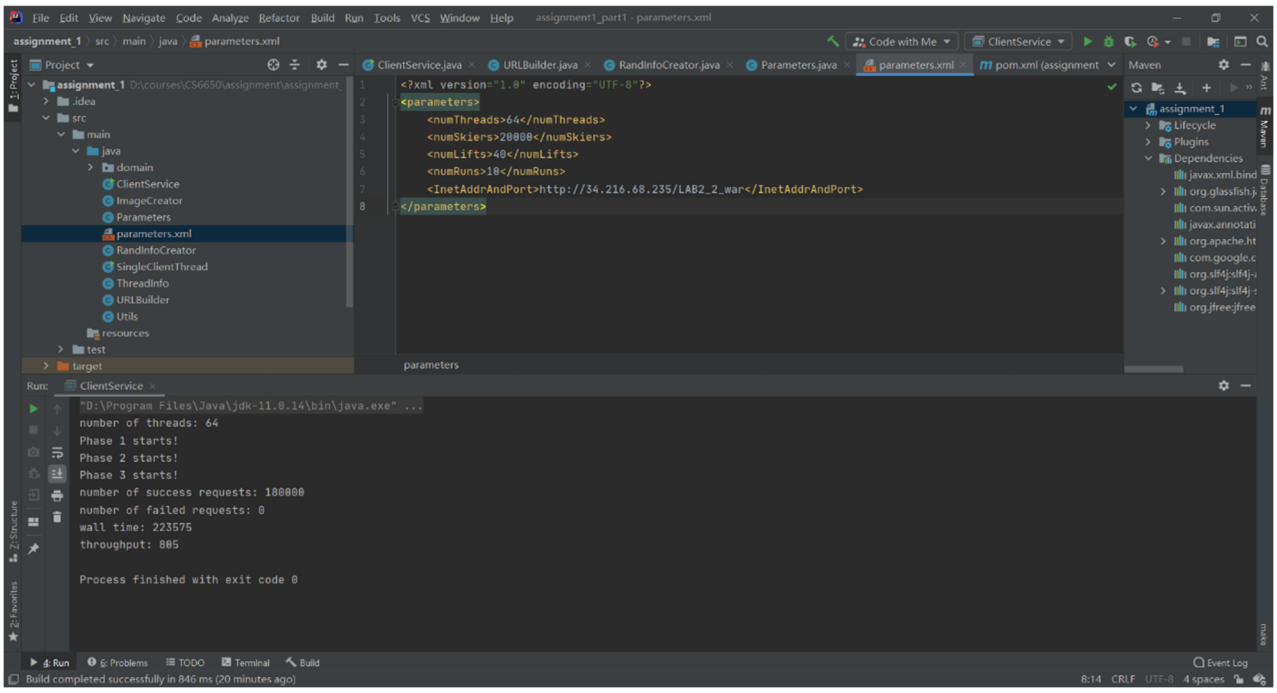
throughput = 180000/87.5 = 2057

Output and Charts

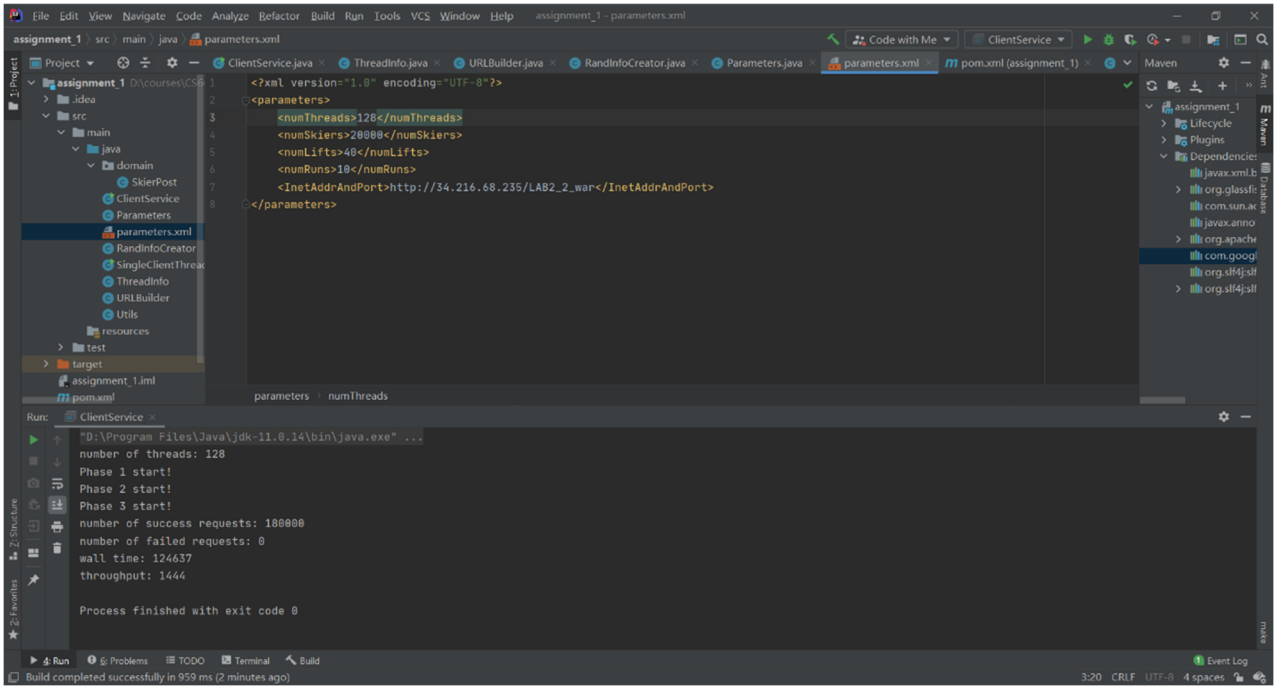
Part1:



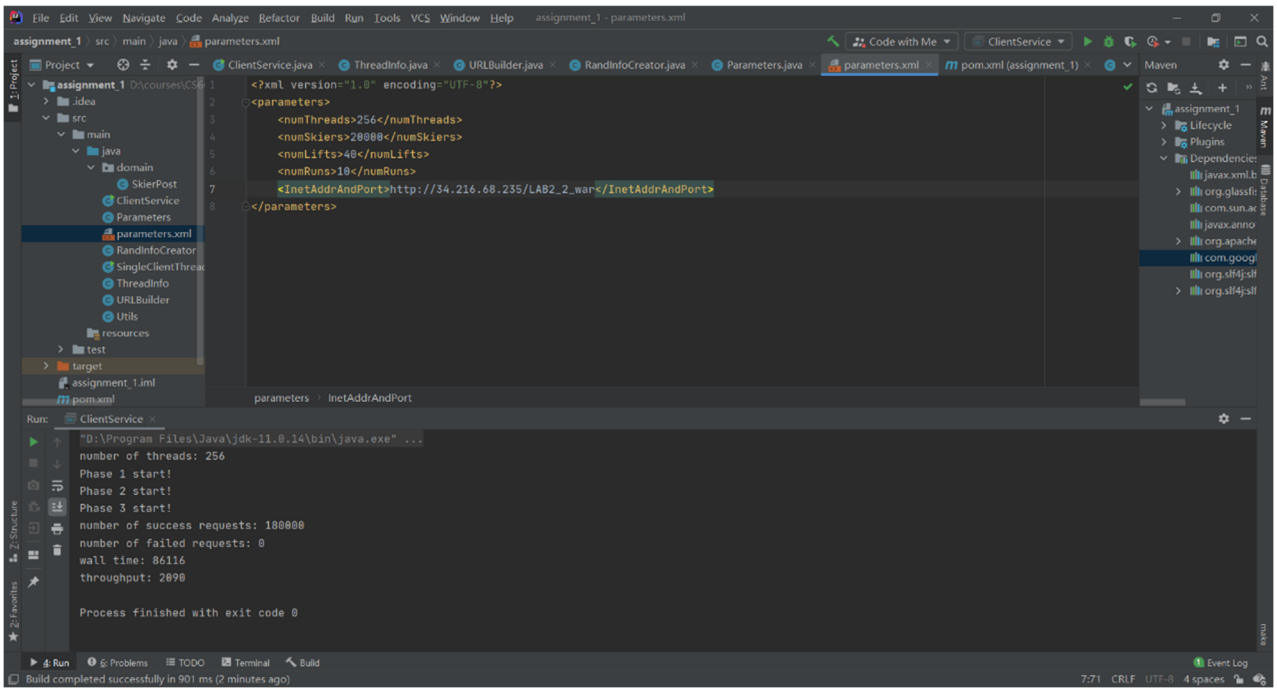
32 Threads



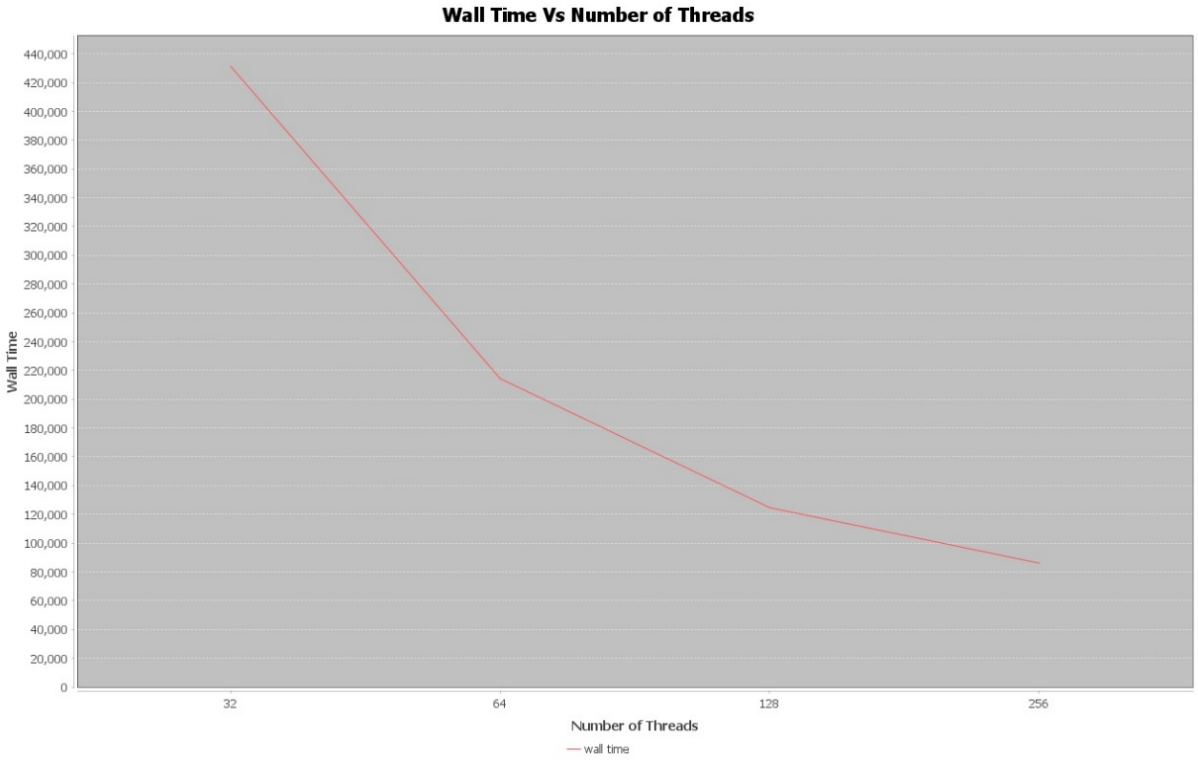
64 Threads



128 Threads

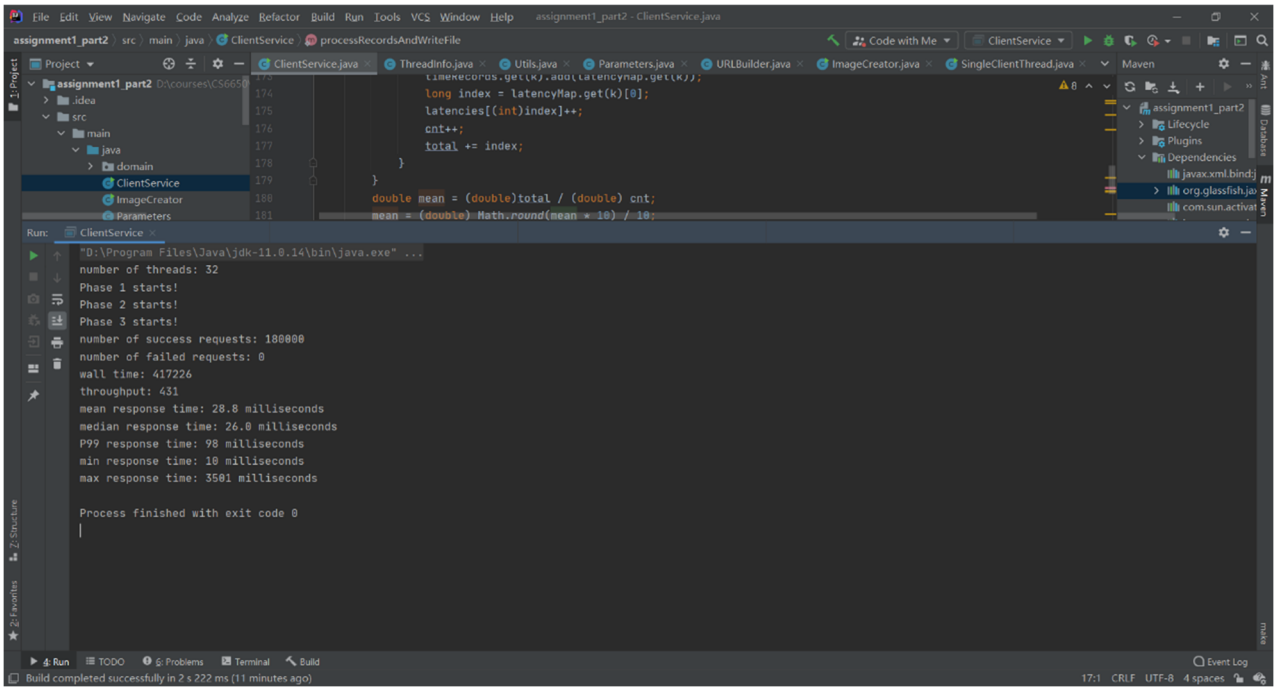


256 Threads

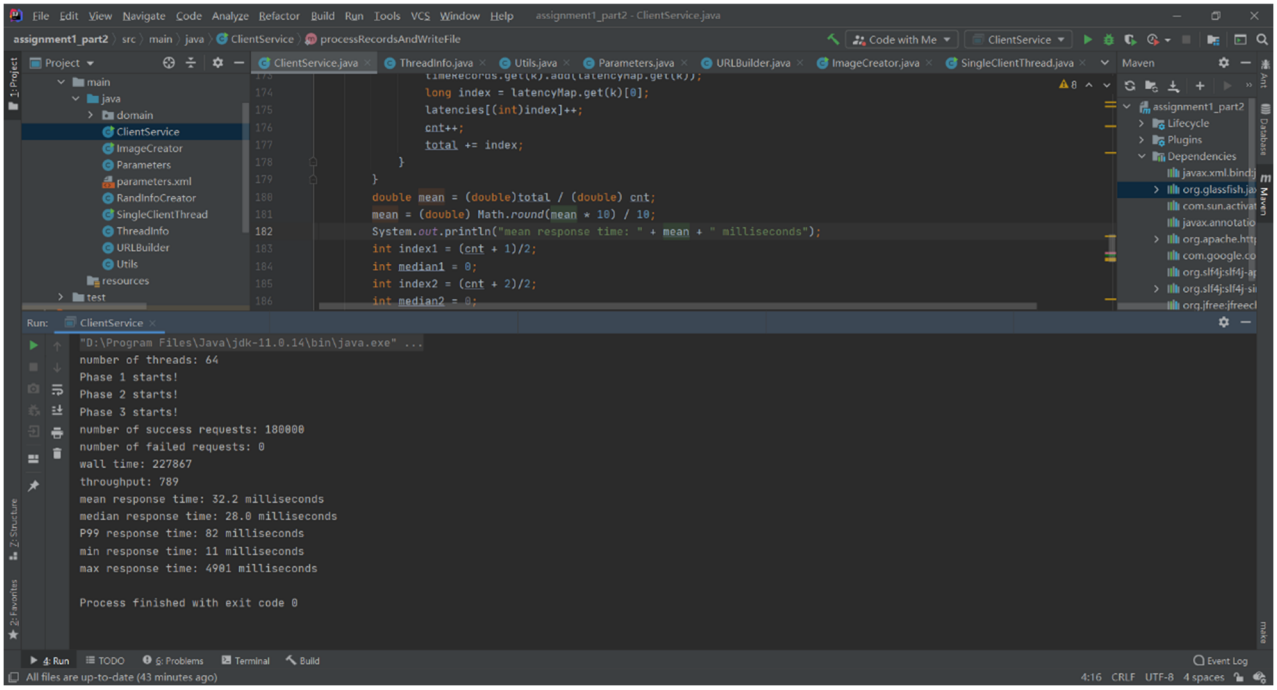


Walltime VS Number of Threads

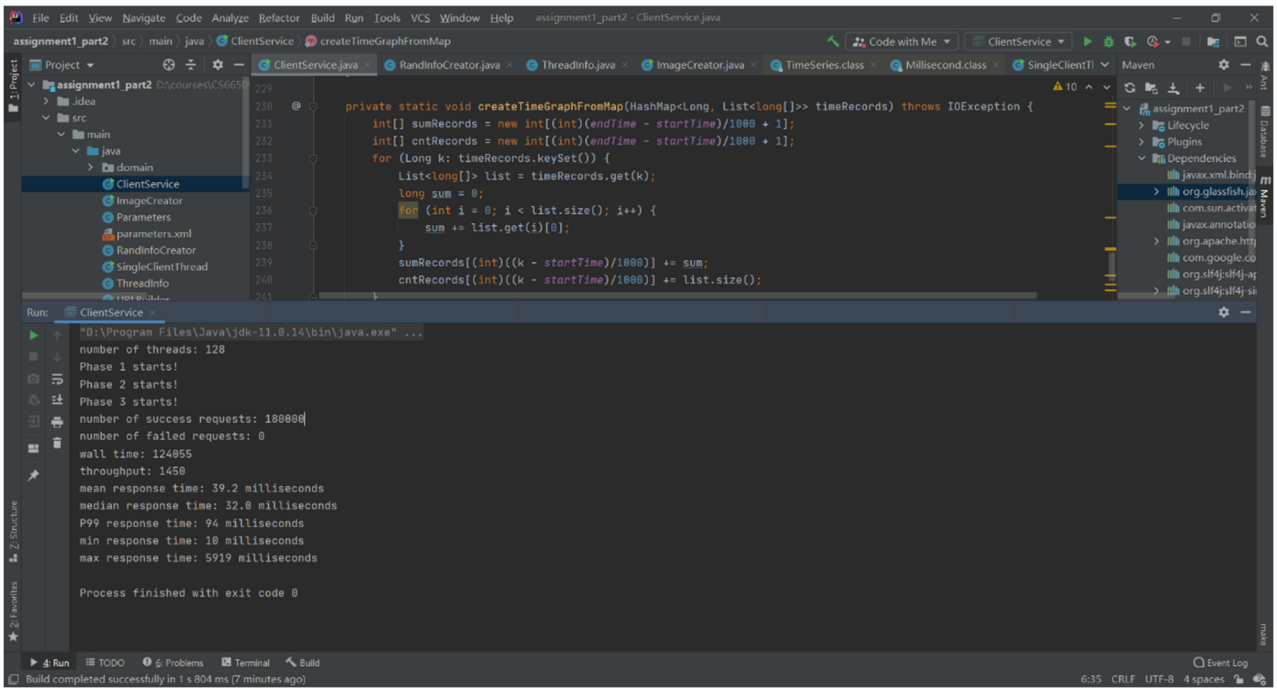
Part2:



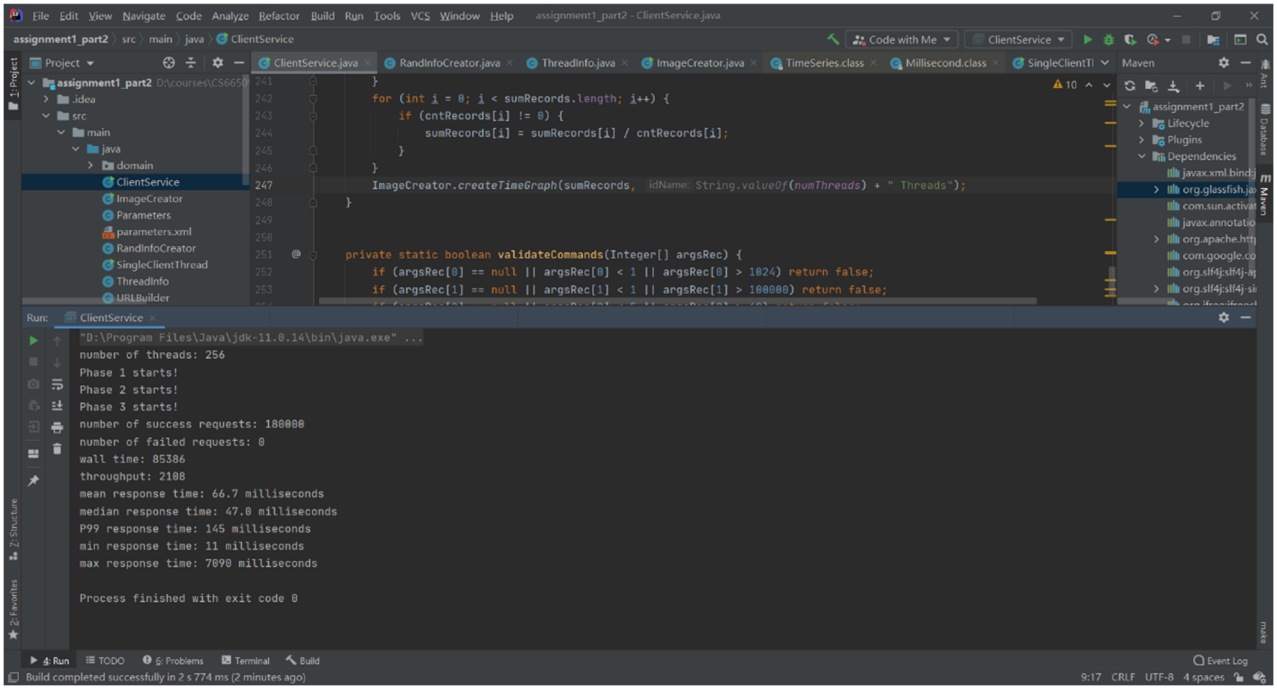
32 Threads



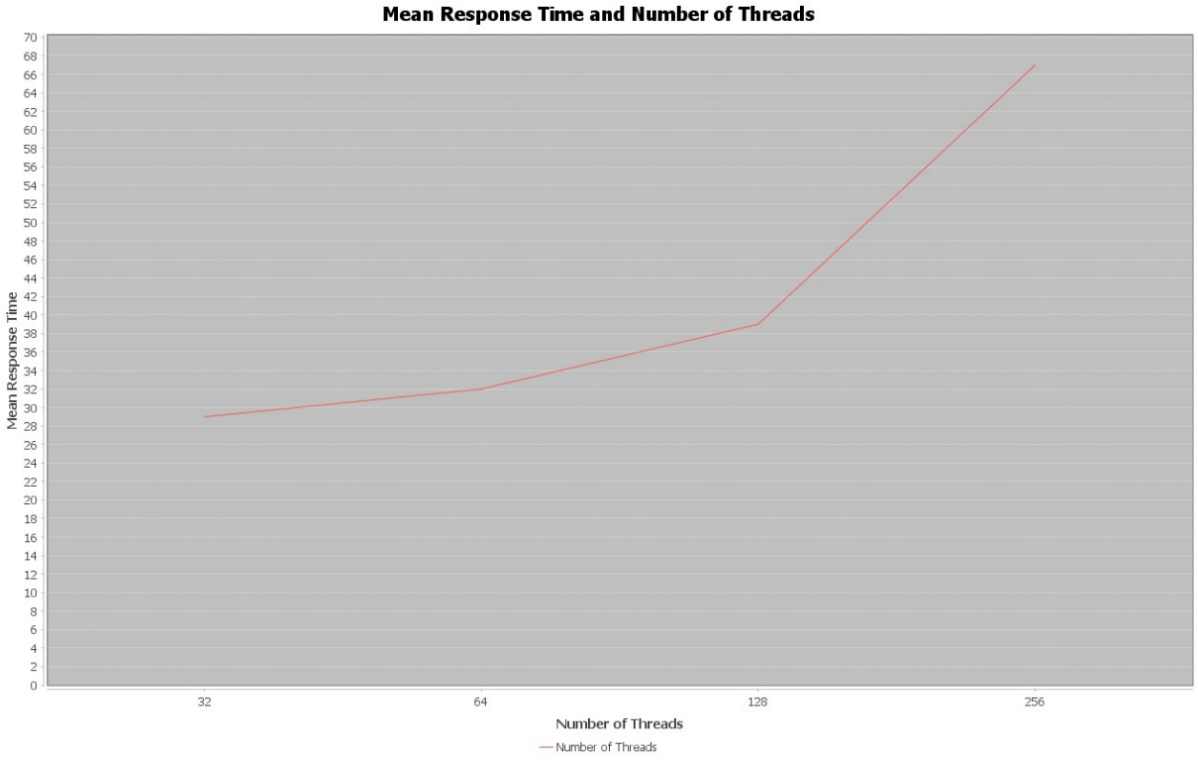
64 Threads



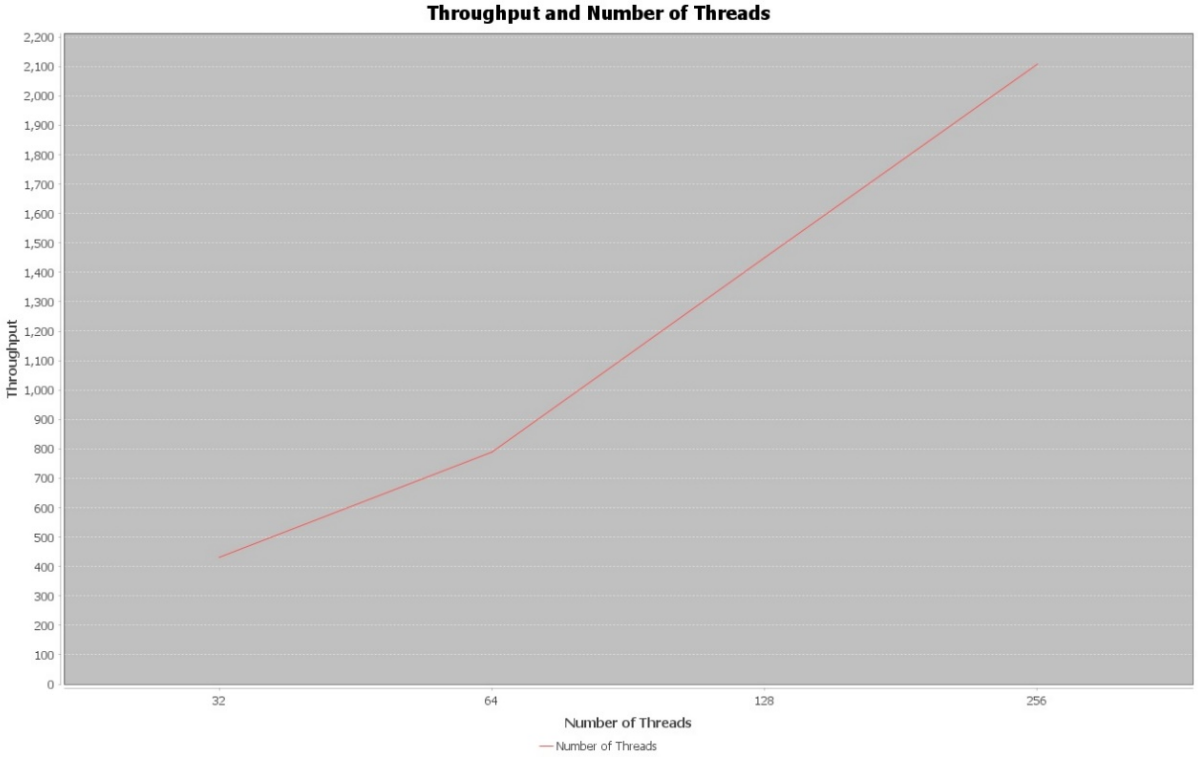
128 Threads



256 Threads

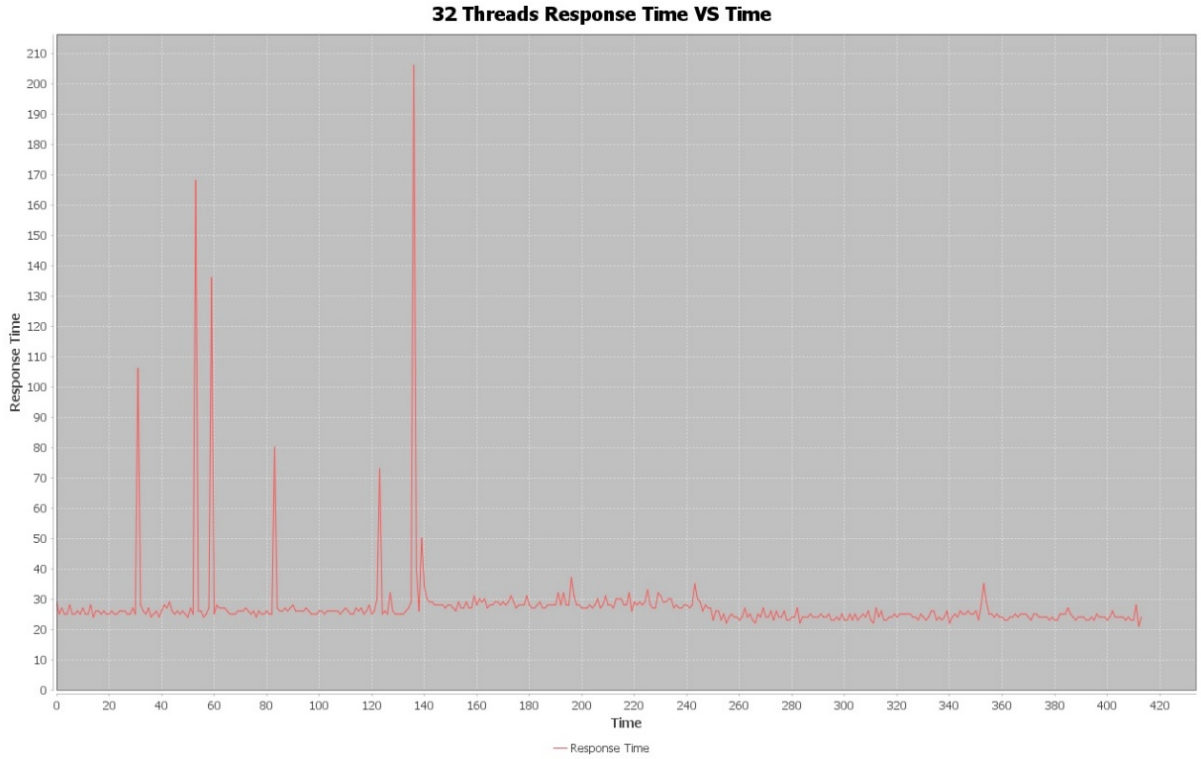


Mean Response Time VS Number of Threads

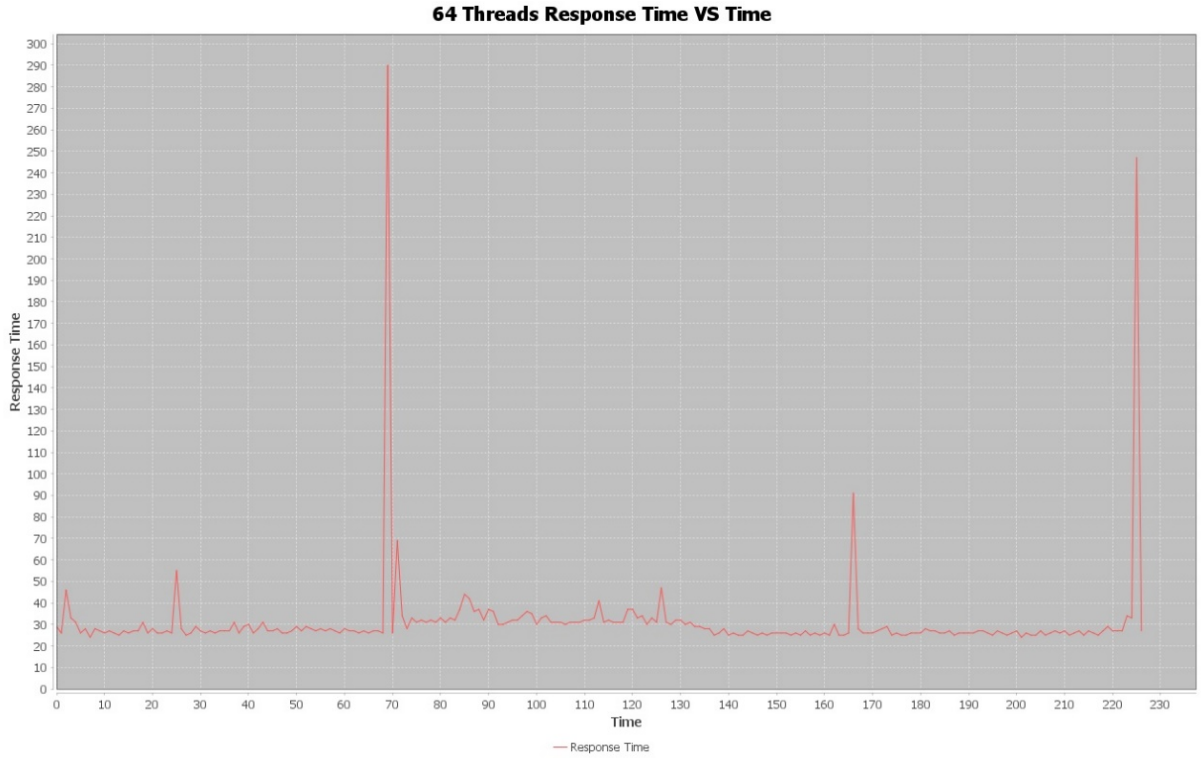


Throughput VS Number of Threads

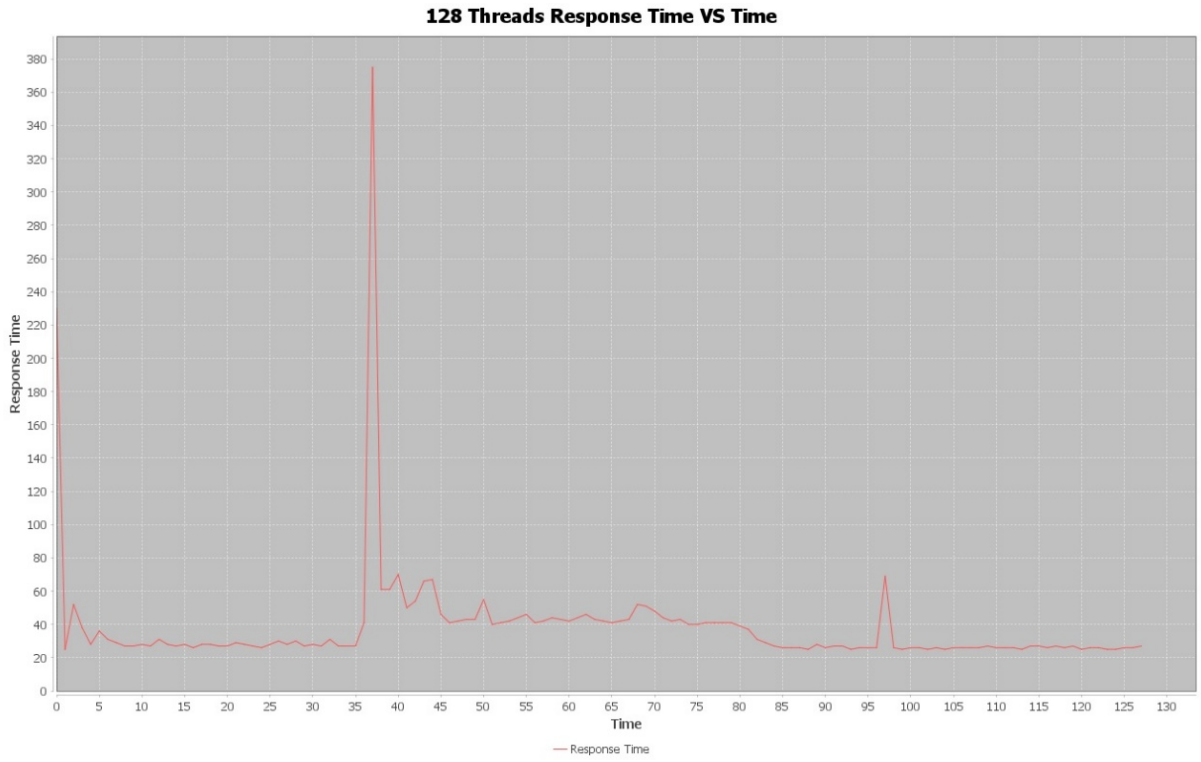
Part 2 Time Charts:



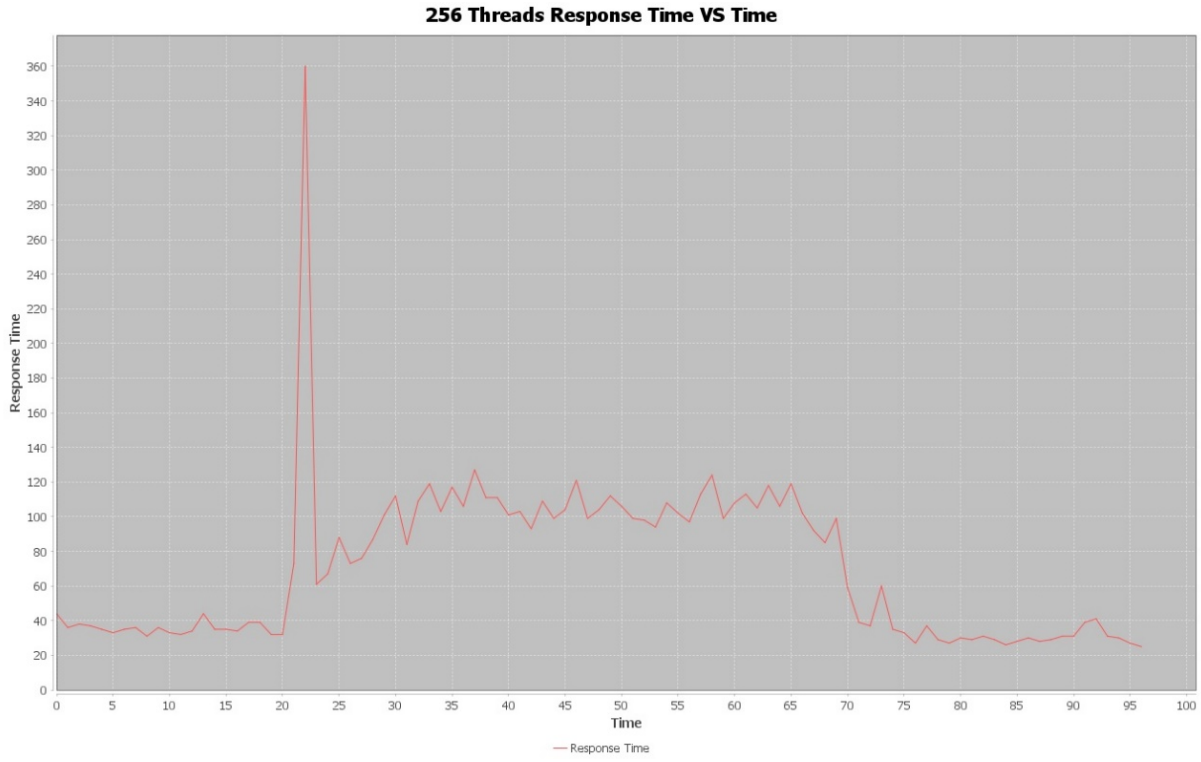
32 Threads



64 Threads



128 Threads



256 Threads