Assignment 1 Report Shengguo Zhou

URL: https://github.com/Shengguo-Zhou/CS6650_Distributed_System/tree/main/assignment-1

Description of my client design:

Client Part 1:
Main function is in "MultiThread" class. Inside this class:

First step, I created a queue and put 500k request inside it. And the queue is a global variable.

Nest step, I use a for loop to generate 200 threads to achieve multiply-thread. For each of thread, whenever this thread is started, it will pull a request from the queue. If the queue is empty, it will pull nothing, which means this thread would do nothing. So which thread is started first, which one may be tasked with more sending request tasks. Therefore, within this step, there would be 500k requests in total, and the 200 threads need to send it to the local or EC2 server. When all the requests are sent, which means the queue is empty, the 200 threads would then have a rest.

Last step, there would be some other global variable that would count the successful request number and the failed one, the lasting time and so on.

RunInOneThread.java: This class is designed for each one of the thread, I need to send the data through *api.swipeWithHttpInfo*. Inside the run function, the detail of each thread is implemted, queue is the global passed from the main function, when a new thread is created, it will get a request body from the queue. And it won't stop until the queue is empty. There are another two parameters to count the number of successful and failed sending tasks. In the sentEvent function, it will try at most 5 times to sent one request, if it is successful within 5 times of try, success will plus one. When it is failed, this request will be abandoned and failure will plus one.

SwipeEvent.java inside the event file and BodyList.java and GenerateSingleBody.java is the helper function for main and singe-thread functions. For each of the request, I have packaged the info within the request and named it to a event class. When all the 500k request body is generated, it would be put into the list, to be specific, it is the queue that I have mentioned.

Client Part 2:
In general, Client Part 2 = Client Part 1 + counting class + output class.
For this part, I added nothing but result data processing class and a csv output file class. Inside the RunInOneThread.java class, when a new request is successfully sent, I will put the success info into another global list to store it. When all the requests are sent, I will get a list of 500k success info. And then I can sort this list by certain criteria. Like I can sort it by the delay time, then I can get the result of minimum, maximum, medium, 99%percentage response time. And like this, if I sort it by its start time, I can plot the chart by its beginning time to know for each of a certain second, how many requests would be sent.

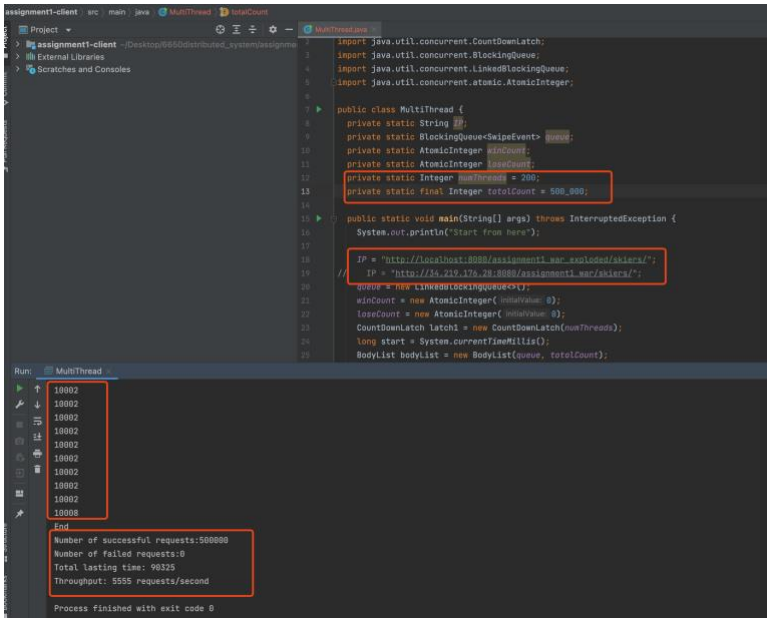
Result:

Part1:

Local host as a server (MacBook Air)

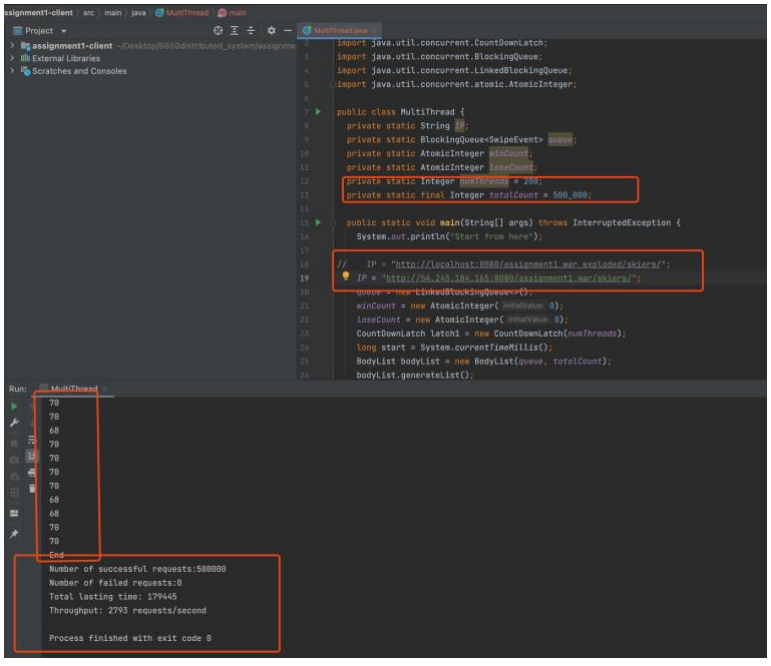
200 threads, 500k request

Result output:



EC2 as a server

200 threads, 500k request:

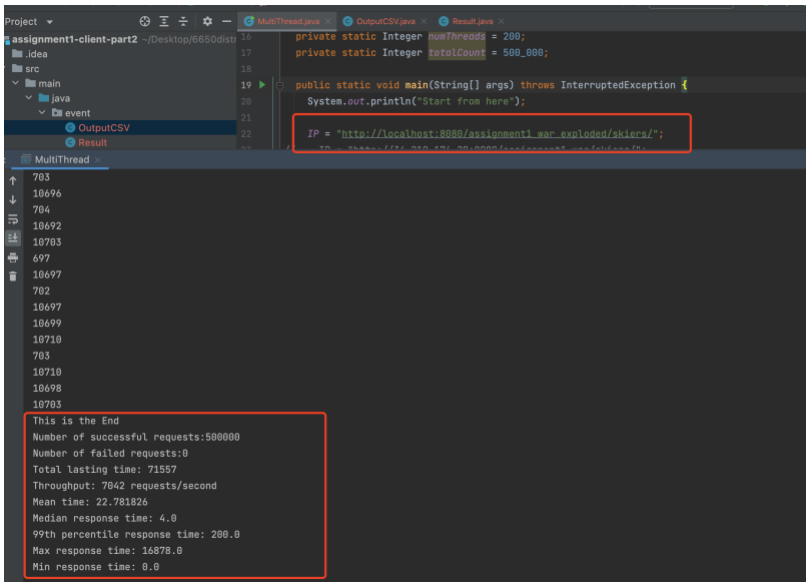


Part2

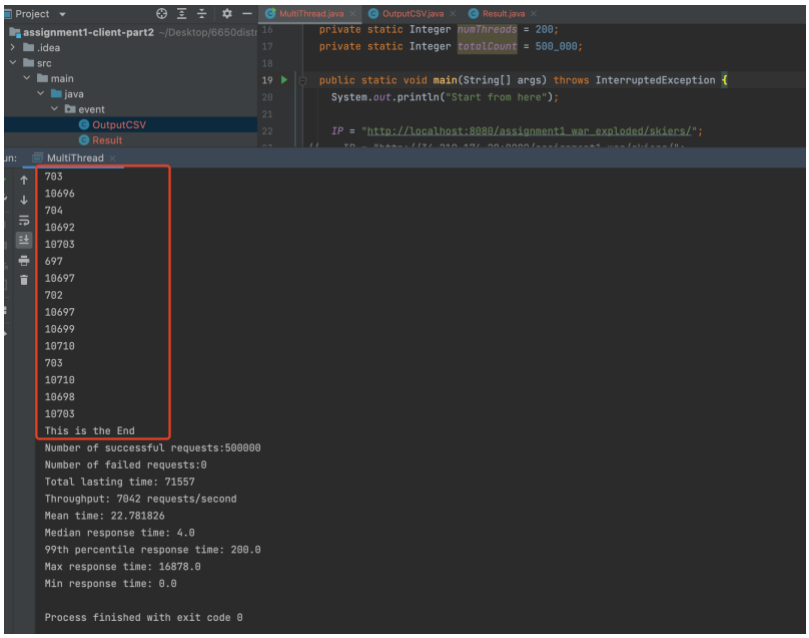
Local host as a server (MacBook Air)

200 threads, 500k request

Result output:

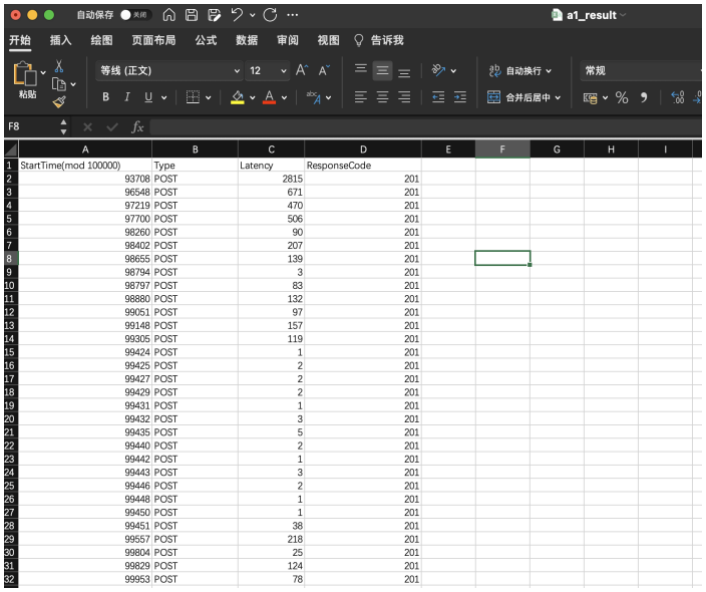


Wall time for each resquest:

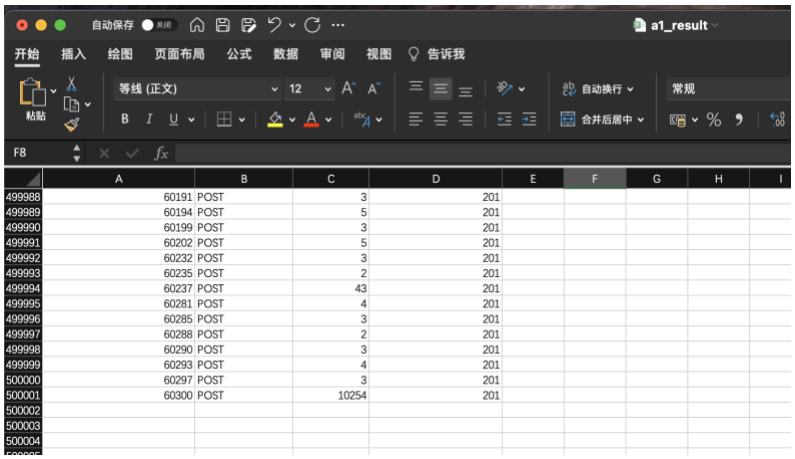


Generated csv file

Inside this csv file



StartTime(mod 100000)	Type	Latency	ResponseCode
93708	POST	2815	201
96548	POST	671	201
97219	POST	470	201
97700	POST	506	201
98260	POST	90	201
98402	POST	207	201
98655	POST	139	201
98794	POST	3	201
98797	POST	83	201
98880	POST	132	201
99051	POST	97	201
99148	POST	157	201
99305	POST	119	201
99424	POST	1	201
99425	POST	2	201
99427	POST	2	201
99429	POST	2	201
99431	POST	1	201
99432	POST	3	201
99435	POST	5	201
99440	POST	2	201
99442	POST	1	201
99443	POST	3	201
99446	POST	2	201
99448	POST	1	201
99450	POST	1	201
99451	POST	38	201
99557	POST	218	201
99804	POST	25	201
99829	POST	124	201
99953	POST	78	201



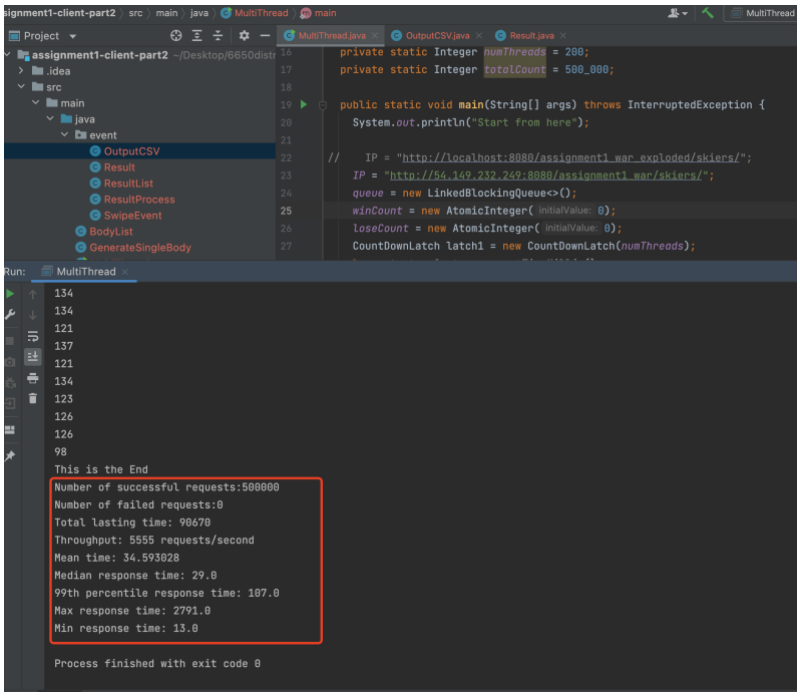
499988	60191	POST	3	201
499989	60194	POST	5	201
499990	60199	POST	3	201
499991	60202	POST	5	201
499992	60232	POST	3	201
499993	60235	POST	2	201
499994	60237	POST	43	201
499995	60281	POST	4	201
499996	60285	POST	3	201
499997	60288	POST	2	201
499998	60290	POST	3	201
499999	60293	POST	4	201
500000	60297	POST	3	201
500001	60300	POST	10254	201
500002				
500003				
500004				

EC2 as a server

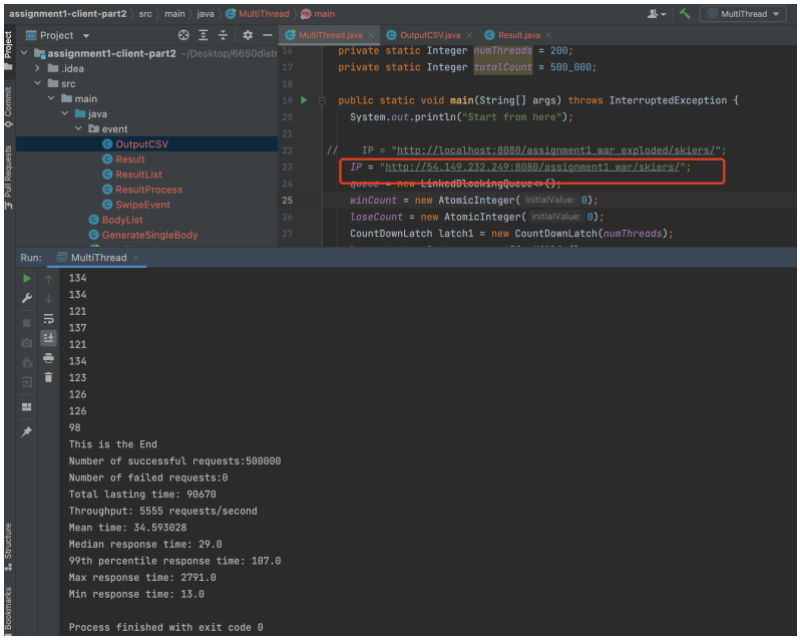
200 threads, 500k request

Current ip: 54.149.232.249

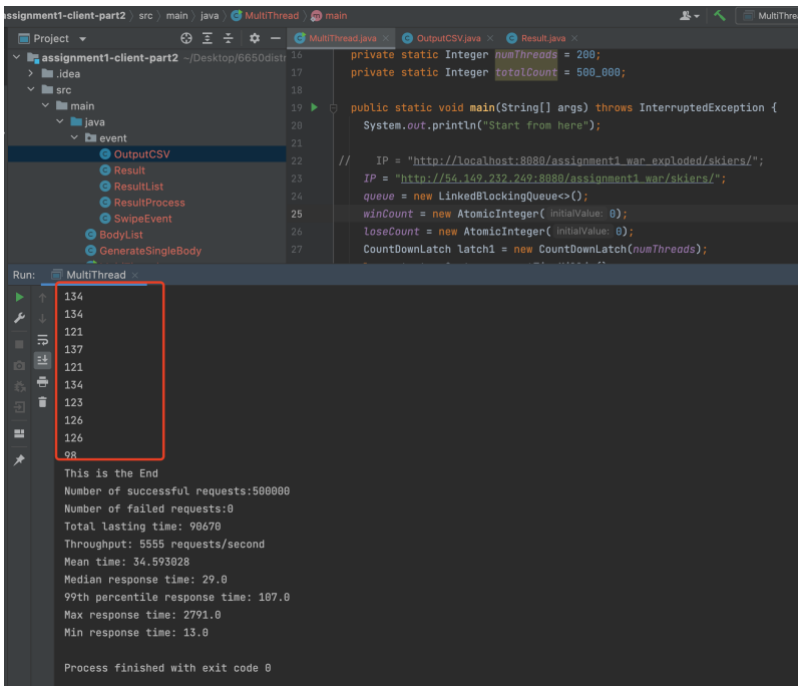
Fastest time: 90.6s



Current ip:

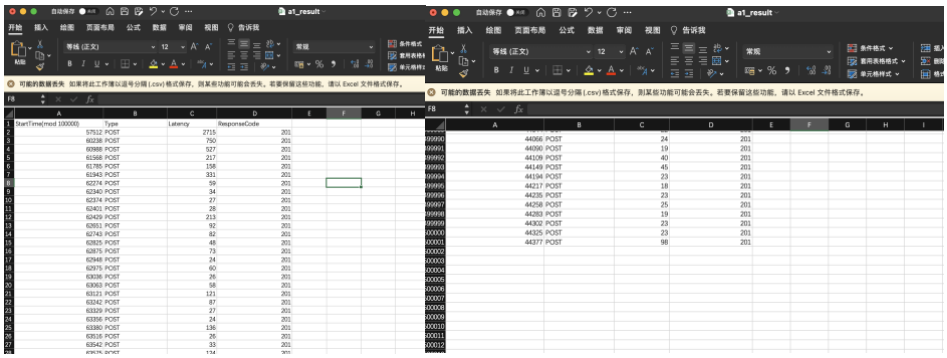


Each wall time for each request



Result csv:

Inside a csv



Mean Throughput (λ): λ = L / W

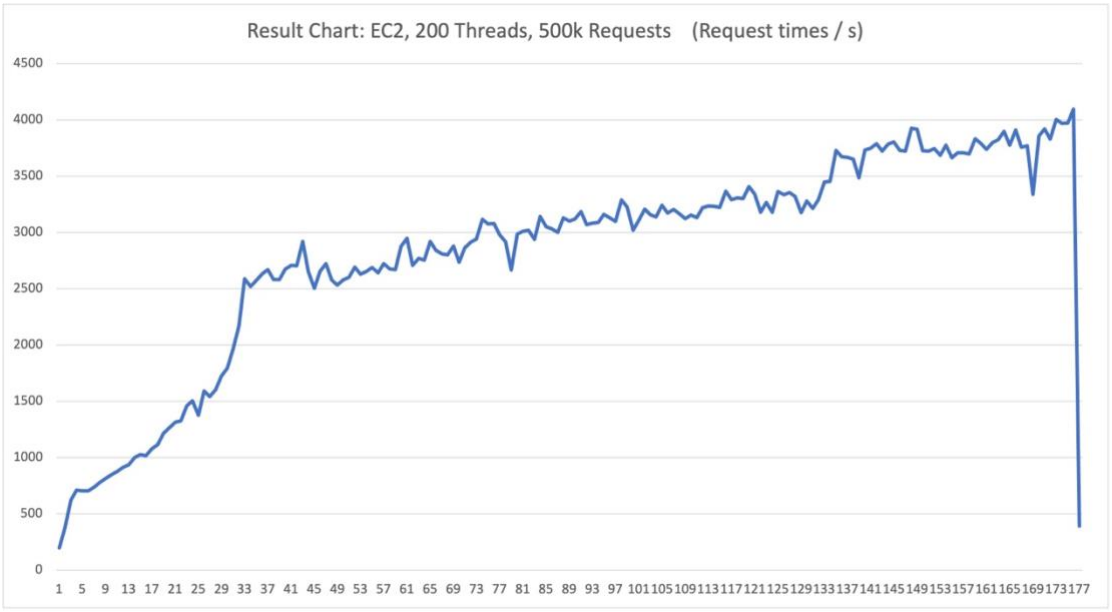
Median Throughput: To calculate the median throughput, I first need to sort the data in ascending or descending order and then find the value that separates the data into two halves.

p99 Throughput: The p99 throughput is the throughput value that corresponds to the 99th percentile, meaning that 99% of the data points fall below this value. You'll need to sort the data and find the

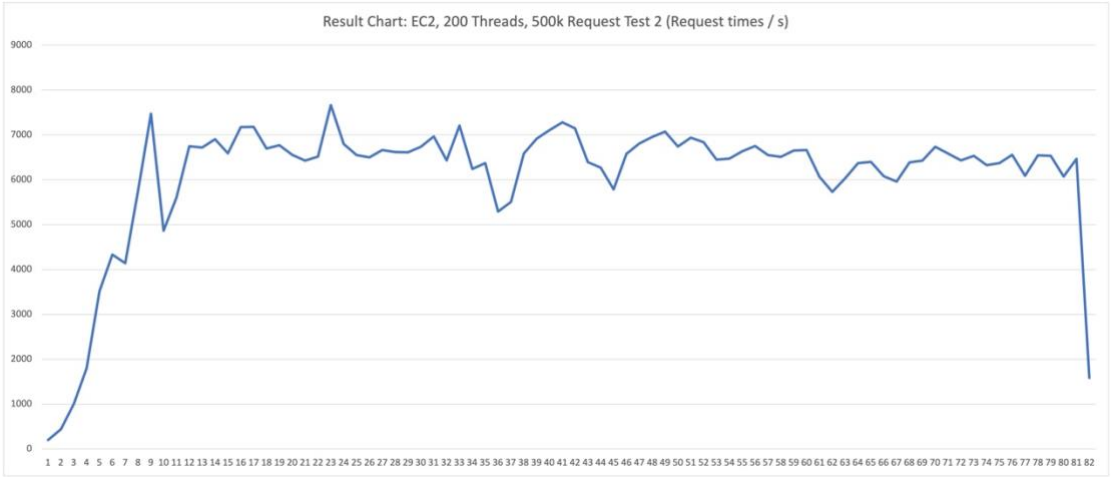
value that corresponds to the 99th percentile.
Max Throughput: The max throughput is simply the highest value in the data set.
The estimation of the sending speed on AWS EC2 should be around 5500 – 6000 requests/s, max time should within 5000ms, and min time could be 0 after calculation. 99% and mean and medium time should within 2750ms to 3000ms, so my client and server is qualified in this case.

Graph:
After getting the data within a csv file, I can get the chart below:
First two pictures are tested on EC2, which is stable, sometimes the total sending time is 177s, the picture two is the fastest try, which is around 90s.

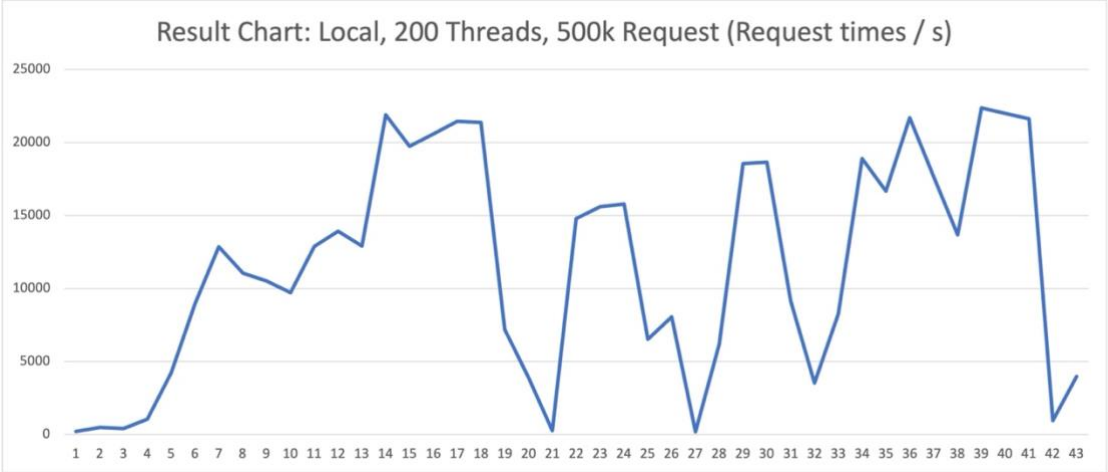
Test 1 (Slow)



Test 2 (Fastest)



If the local is a server, it is faster, the total time cost only around 45s, but it is not stable.



Comparasion:

Swagger is an API framework that can be used to design, build, and document RESTful APIs, while Spring is a Java-based framework that provides a comprehensive programming and configuration model for modern Java-based enterprise applications. Spring Boot may be faster in this case, because it has some specific time complexity improvement for sending and receiving data. And with its comprehensive package for maven of Java, it is more convinent if we use Java SpringBoot as server.