CS6650 Assignment1

Yijia Ma

CS6650 Assignment1 **1**
1. Repo URL: https://github.com/CocoCSforever/cs6650/tree/main 1
2. Servlet Design(/skiers) 1
3. Client Design 2
4. Client (Part 1) - This should be a screenshot of your output window with your wall time and throughput. Also make sure you include the client configuration in terms of the number of threads used. 3
4.1. Client configuration 3
4.2. Output window 3
4.3. Observation 5
4.4. Little's Law 5
5. Client (Part 2) - run the client as per Part 1, showing the output window for each run with the specified performance statistics listed at the end. 5
5.1. Client configuration 5
5.2. Output window 5
5.3. Little's Law 8
6. Plot of Throughput 8

1. Repo URL: https://github.com/CocoCSforever/cs6650/tree/main
    1.1. Please see SkierServlet API in Assignment1_Servlet.
    1.2. Please see Client API in Assignment1_Clients(I changed the name of the project lately which may show some warning of the dependencies but it runs well.):
    Client Part1 in Assignment1_Clients/src/main/java/clients/client1.
    Client Part2 in Assignment1_Clients/src/main/java/clients/client2.
    Client Test for single thread in Assignment1_Clients/src/test/SkiersApiTest
2. Servlet Design(/skiers)
    2.1. doPost
        a. Sample request url:
        /{resortID}/seasons/{seasonID}/days/{dayID}/skiers/{skierID}
        b. Validation Steps
            i. The function isUrlValid serves as a basic validation. It verifies whether the urlPath contains the seven parts as outlined in our sample request URL and ensures that the second, fourth, and sixth parts are exactly equal to "seasons," "days," and "skiers," respectively.
            ii. A URL path that passes the validation indicates that it has the same structure as outlined in the sample request but is not necessarily guaranteed to have a valid value for a liftRide object.
        c. Status Code
            i. 404 NOT_FOUND

1. If the urlPath is null or empty, it should return a response with a 404 status code and write a message into req.body stating "missing parameters" along with the urlPath.

ii. 400 BAD_REQUEST

1. if isUrlValid(urlPath) returns false, it should return a response with a 400 status code and write a message into req.body stating "Invalid URL Path parameters: " along with the urlPath.
2. If the parameters interpreted from the URL are not valid as attributes in a LiftRide object, it should return a response with a 400 status code and write a message into req.body stating "Bad URL path parameters: " along with the urlPath and the error message obtained from e.getMessage().

iii. 201 CREATED

1. If a LiftRide object is successfully created without any errors, it should return a response with a 201 status code and write the string representation of the object into req.body.

3. Client Design

3.1. Important instruction to reproduce my result

To reproduce my result, please change Client1_Multi_Threads, line 8 with the desired threads and requestsPerThreads, and Client1ProducerConsumer, line 8 to align the size of thread pool with the number of desired threads(it should not be less than 32).

Please change Client2_Multi_Threads, line 17 with the desired threads and requestsPerThreads, and Client1ProducerConsumer, line 12 to align the size of thread pool with the number of desired threads(it should not be less than 32).

3.2. Class structure

My class design for Part1 and Part2 follows the same pattern. Take Client Part1 for example, it includes:

a. Client1_Multi_Threads
   - i. This is where our main method resides. It calls a helper function(pass the number of threads and requests per threads) and records wall time and other necessary information, and prints them out.

b. Client1ThreadHelper
   - i. Fixed thread pool of size 1 is used for generating ready-to-send liftRide events since we are required to use a single dedicated thread.
   - ii. client1StartNThreads is the helper function which returns the wall time for sending client requests. It includes:
     1. Create a ProducerConsumer object.
     2. Set required CountDownLatch.
     3. Generate LiftRide events.
     4. send client requests using the generated events.
     5. wait until all client requests are successfully sent and get good responses.
     6. record wall time and return.
   - iii. generateLiftRideData submit the produce() task to the thread pool and wait until all events are ready using latch.await().
   - iv. sendClientRequest submit the consume() task to thread pool.

c. Client1ProducerConsumer
   i. Constructor takes nOfThread as the number of threads or clients, and requestPerThread as the number of requests to send per thread.
   ii. Fixed thread pool of size $nOfThread is only used for running client1Runnable($ represents the value of the variable).
   iii. produce() will create $nOfThread client1Runnable and generate $requestPerThread liftRide events for each client1Runnable. After the data for a client is ready to go, it adds the client1Runnable to the taskQueue.
   iv. consume() will take $nOfThread client1Runnable from the taskQueue one by one and submit the runnable object to the thread pool.

d. Client1Runnable
   i. This is a class that implements Runnable Interface and run() defines specific actions for each thread to execute.
   ii. Depending on requirements, I set some static variables for class Client1Runnable, and these variables can be shared by all threads created based on Client1Runnable instance, eg. successCounter, failCounter, CountDownLatch, BufferedWriter.
   iii. Each Runnable instance has their own SkierAPI created by different ApiClient so that each thread represents a respective client and sends its requests for the client.
   iv. ArrayList<LiftRideInfo> is a list of size $requestPerThread and contains liftRide events generated by produce().

Some adds on for Client2:

a. Client2Runnable
   i. To write the required info for each request, I created a writeToFile function which is called when we get a successful response. Since writeToFile is a logically synchronized function, I created a fixed thread pool of size 1 in class Client2Runnable to submit a write task to it.

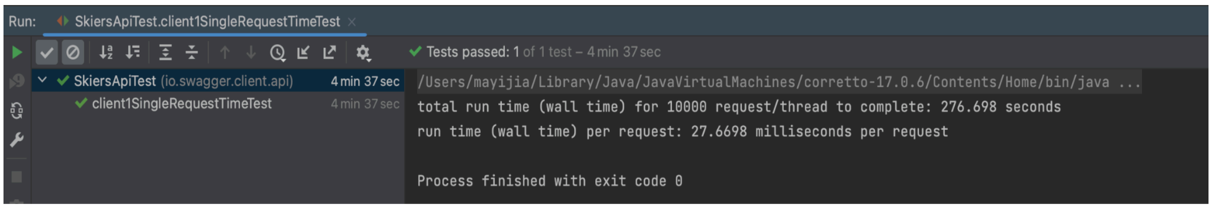
b. Client2_calculate
   i. It’s created for analyzing the data we wrote to output.csv such as to calculate the mean response time, median response time and so on.

3.3. Packages

a. My packages(src/main/java/clients)
   i. I write Client Part1 and Part2 in two folders in package clients.
   ii. I modified SkiersApiTest: added one test for each client part.

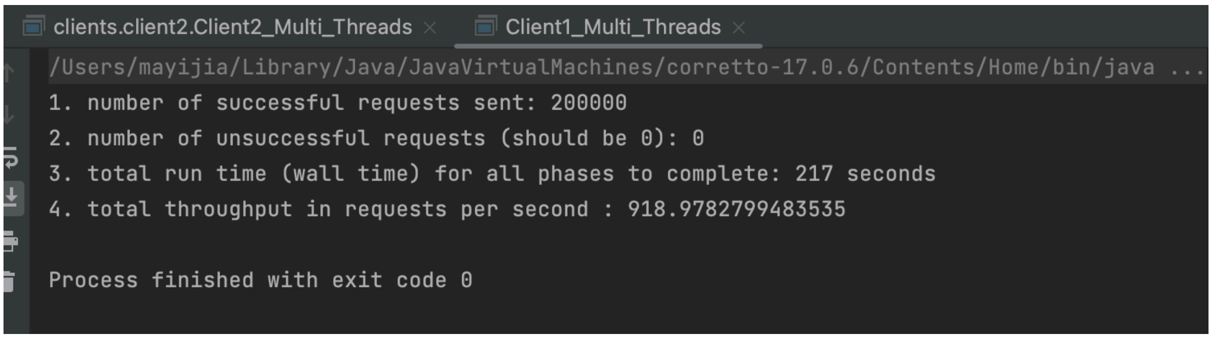
b. Swagger generated API(src/main/java/io.swagger.client)
   i. I mainly relied on ApiClient, ApiCallback, SkiersApi to send client requests to my server.
   ii. I added a constructor for SkiersApi that takes an ApiClient instance so that every thread can have a different client instance instead of all set to the configuration’s default client.
   iii. I implement ApiCallback to have different tasks done onFailure and onSuccess of client requests.

4. Client (Part 1) - This should be a screenshot of your output window with your wall time and throughput. Also make sure you include the client configuration in terms of the number of threads used.
   4.1. Client configuration
      a. It first created 32 threads each sending 1k POST requests. After completing all these requests, it created x threads(x = 32, 80, 120 in different test cases) until 200k POST requests have been sent.
      b. Please see 4.2 Output window a and f as the best throughput.
      c. Please see 4.2 Output window g as testing for single thread 10000 requests for single request latency.
   4.2. Output window
   a. Single Thread sending 10000 requests to West EC2


Run: SkiersApiTest.client1SingleRequestTimeTest
Tests passed: 1 of 1 test – 4 min 37 sec
SkiersApiTest (io.swagger.client.api) 4 min 37 sec
client1SingleRequestTimeTest 4 min 37 sec

```
/Users/mayijia/Library/Java/JavaVirtualMachines/corretto-17.0.6/Contents/Home/bin/java ...
total run time (wall time) for 10000 request/thread to complete: 276.698 seconds
run time (wall time) per request: 27.6698 milliseconds per request

Process finished with exit code 0
```

   b. Running on West EC2(32 threads * 1k + 32 threads * 5.25k)

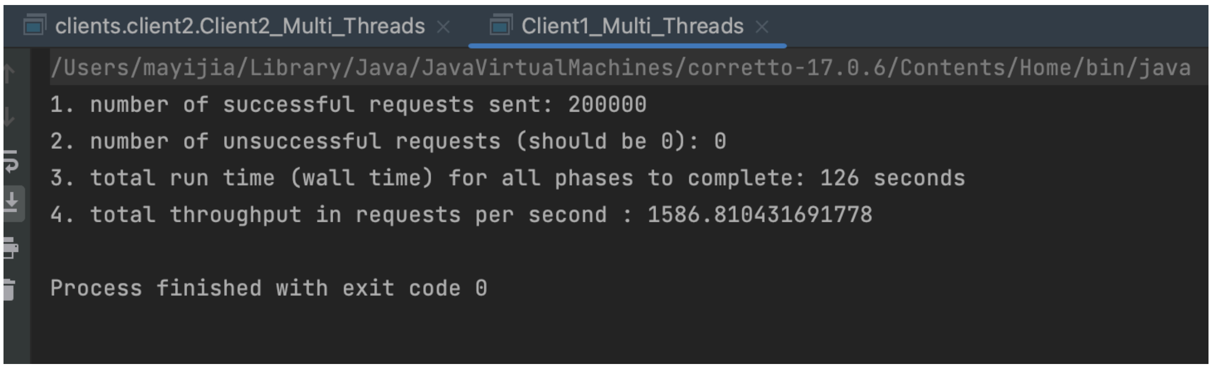


```
/Users/mayijia/Library/Java/JavaVirtualMachines/corretto-17.0.6/Contents/Home/bin/java ...
1. number of successful requests sent: 200000
2. number of unsuccessful requests (should be 0): 0
3. total run time (wall time) for all phases to complete: 217 seconds
4. total throughput in requests per second : 918.9782799483535

Process finished with exit code 0
```

Average max threads	Response time (milliseconds)	Estimated arrival/exit rate	Actual Throughput
32	27.67	1156 requests/second	918 requests/second

32/27.67(ms) = 1156 requests/second

   c. Running on West EC2(32 threads * 1k + 80 threads * 2.1k)



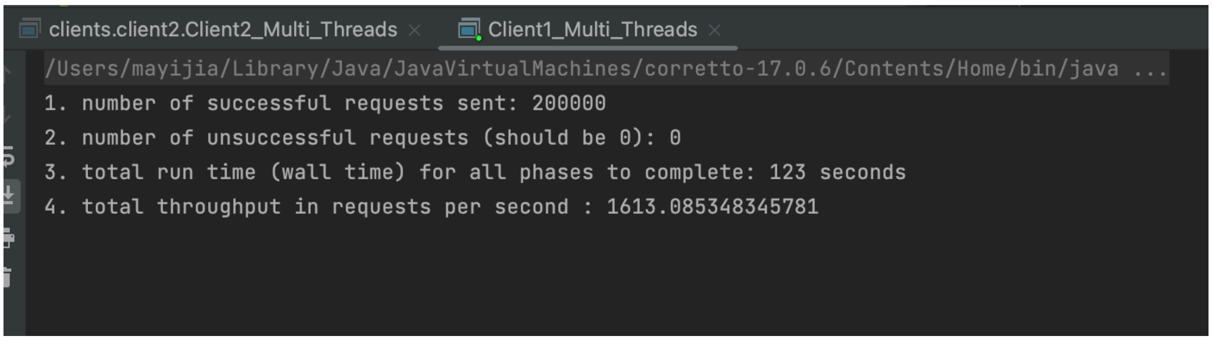
```
/Users/mayijia/Library/Java/JavaVirtualMachines/corretto-17.0.6/Contents/Home/bin/java
1. number of successful requests sent: 200000
2. number of unsuccessful requests (should be 0): 0
3. total run time (wall time) for all phases to complete: 126 seconds
4. total throughput in requests per second : 1586.810431691778

Process finished with exit code 0
```

Average max threads	Response time (milliseconds)	Estimated arrival/exit rate	Throughput

32~80	27.67	1156~2891 requests/second	1586 requests/second

32/27.67(ms) = 1156 requests/second
80/27.67(ms) = 2891 requests/second

d. Running on West EC2(32 threads * 1k + 120 threads * 1.4k)



```
/Users/mayijia/Library/Java/JavaVirtualMachines/corretto-17.0.6/Contents/Home/bin/java ...
1. number of successful requests sent: 200000
2. number of unsuccessful requests (should be 0): 0
3. total run time (wall time) for all phases to complete: 123 seconds
4. total throughput in requests per second : 1613.085348345781
```

Average max threads	Response time (milliseconds)	Estimated arrival/exit rate	Throughput
32~120	27.67	1156~4336 requests/second	1613 requests/second

32/27.67(ms) = 1156 requests/second
120/27.67(ms) = 4336 requests/second

4.3. Observation

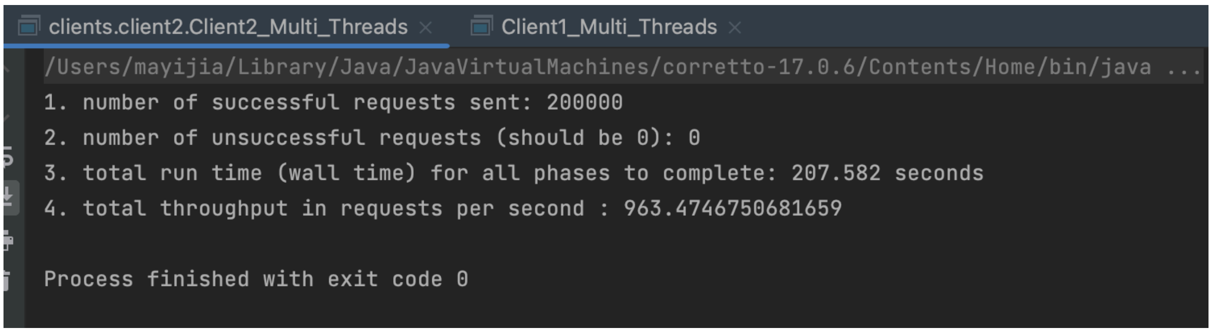
a. Optimal throughput is achieved with 32 client threads locally but with 120 client threads on EC2. This difference is likely because local runs share resources between server and client threads, limiting the total available, while EC2 provides more resources, allowing for more client threads to operate without depleting the system's thread capacity, thereby supporting increased throughput.

4.4. Little's Law

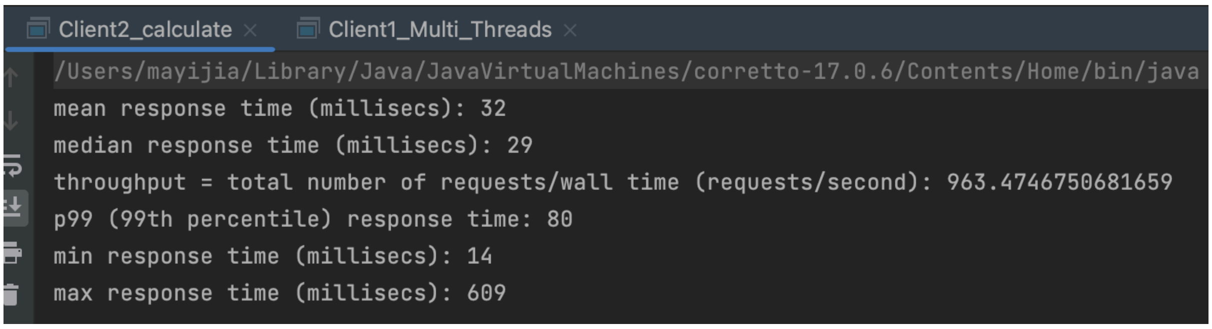
a. Observation
We'll use 32 client threads to analyze it since this case maintains a constant long-term average number of clients. The long term average throughput is 32 threads(my server max thread is set to default 200). Response time for each request: 27.67 millisecond, thus the arrival/exit rate should be 32/27.67= 1156 requests per second. The Actual throughput turns out to be 918 requests per second.
For the best throughput test run(32 threads * 1k + 120 threads * 1.4k), the estimated throughput range is from 1156 to 4336 requests/second while the actual throughput is 1613 requests/second.

b. Justification
The gap may be caused by the synchronized functions that I used to increment and get successCounter and failCounter which makes the threads to wait for each other until the lock on the instance is released. In addition, when we use one single thread to send 10000 requests, we are using one client and possibly using the same connection to the server while in a multithread case, we spend more time establishing a connection.

5. Client (Part 2) - run the client as per Part 1, showing the output window for each run with the specified performance statistics listed at the end.
    5.1. Client configuration
        a. It first created 32 threads each sending 1k POST requests. After completing all these requests, it created x threads(x = 32, 80, 120 in different test cases) until 200k POST requests have been sent.
        b. Please see 5.2 Output window e and i as the best throughput.
        c. Please see 5.2 Output window j as testing single thread 10000 requests for single request latency.
    5.2. Output window
        a. Running on EC2
           32 threads * 1k requests
           32 threads * 5.25k requests
    (with extra 1 threads writing data to file and 1 thread generating liftRide Event)

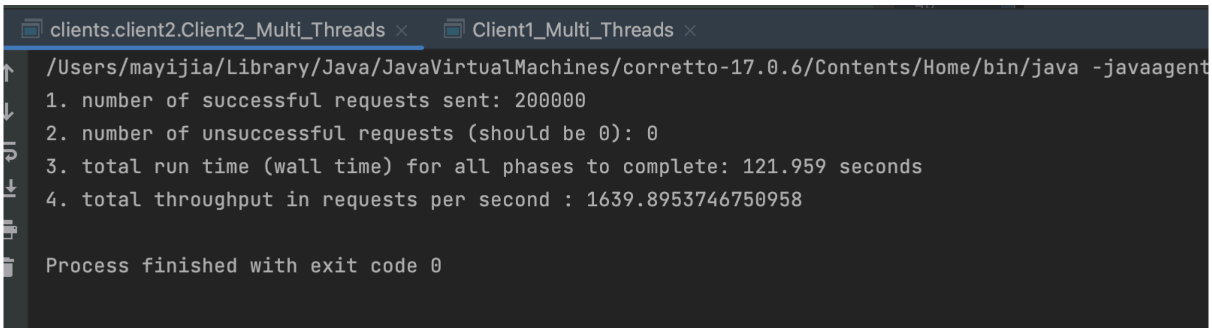


```
/Users/mayijia/Library/Java/JavaVirtualMachines/corretto-17.0.6/Contents/Home/bin/java ...
1. number of successful requests sent: 200000
2. number of unsuccessful requests (should be 0): 0
3. total run time (wall time) for all phases to complete: 207.582 seconds
4. total throughput in requests per second : 963.4746750681659

Process finished with exit code 0
```

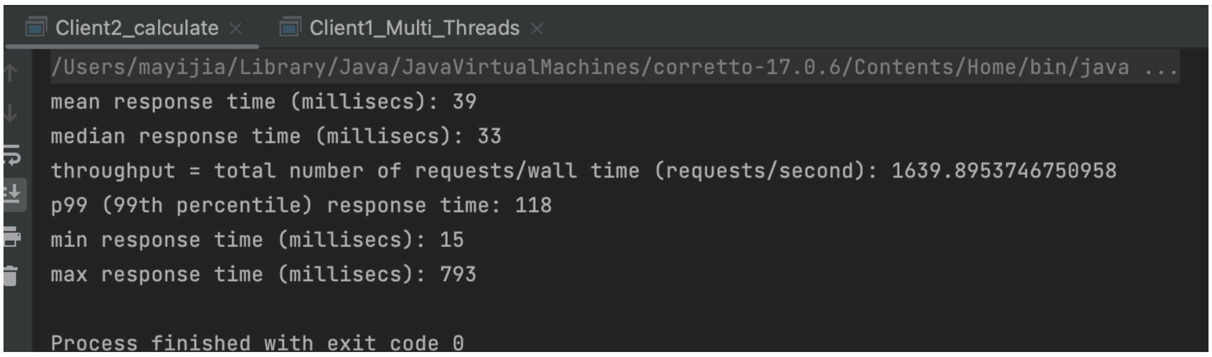


```
/Users/mayijia/Library/Java/JavaVirtualMachines/corretto-17.0.6/Contents/Home/bin/java
mean response time (millisecs): 32
median response time (millisecs): 29
throughput = total number of requests/wall time (requests/second): 963.4746750681659
p99 (99th percentile) response time: 80
min response time (millisecs): 14
max response time (millisecs): 609
```

        b. Running on EC2
           32 threads * 1k requests
           80 threads * 2.8k requests
    (with extra 1 thread writing data to file and 1 thread generating liftRide Event)

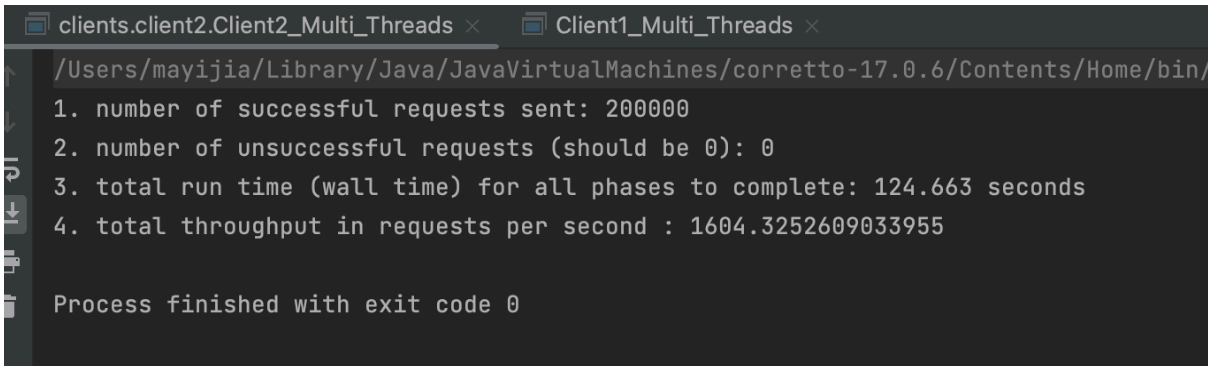


```
/Users/mayijia/Library/Java/JavaVirtualMachines/corretto-17.0.6/Contents/Home/bin/java -javaagent
1. number of successful requests sent: 200000
2. number of unsuccessful requests (should be 0): 0
3. total run time (wall time) for all phases to complete: 121.959 seconds
4. total throughput in requests per second : 1639.8953746750958

Process finished with exit code 0
```

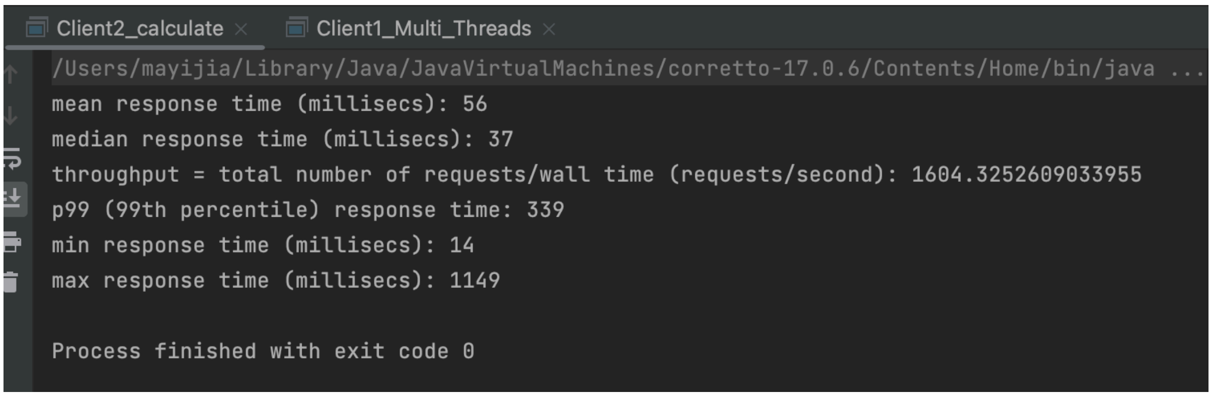


```
/Users/mayijia/Library/Java/JavaVirtualMachines/corretto-17.0.6/Contents/Home/bin/java ...
mean response time (millisecs): 39
median response time (millisecs): 33
throughput = total number of requests/wall time (requests/second): 1639.8953746750958
p99 (99th percentile) response time: 118
min response time (millisecs): 15
max response time (millisecs): 793

Process finished with exit code 0
```

c. Running on EC2**(Best Throughput)**
   32 threads * 1k requests
   120 threads * 1.4k requests
(with extra 1 thread writing data to file and 1 thread generating liftRide Event)



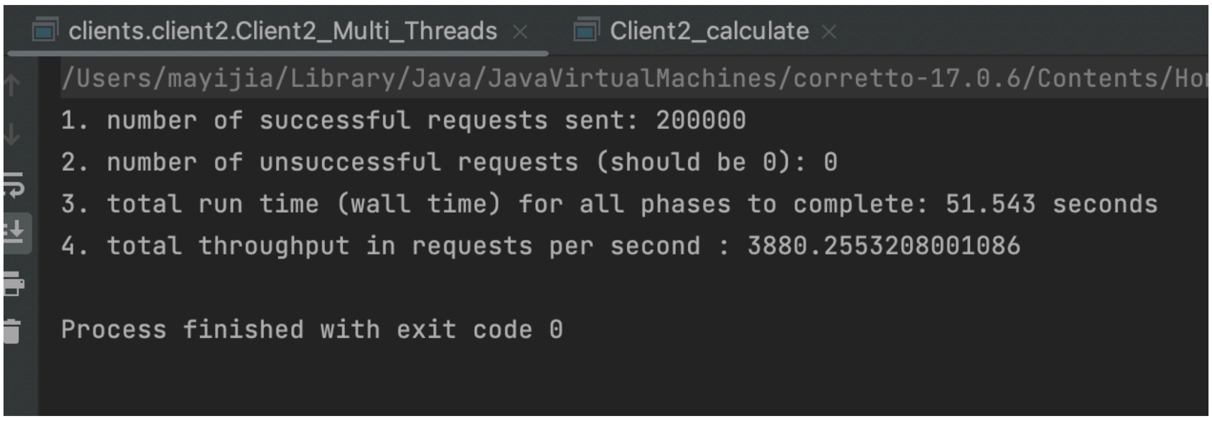
```
/Users/mayijia/Library/Java/JavaVirtualMachines/corretto-17.0.6/Contents/Home/bin/
1. number of successful requests sent: 200000
2. number of unsuccessful requests (should be 0): 0
3. total run time (wall time) for all phases to complete: 124.663 seconds
4. total throughput in requests per second : 1604.3252609033955

Process finished with exit code 0
```



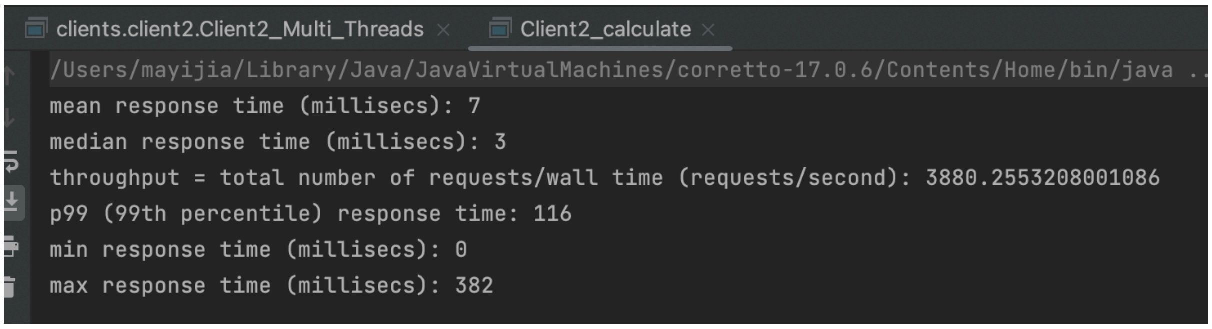
```
/Users/mayijia/Library/Java/JavaVirtualMachines/corretto-17.0.6/Contents/Home/bin/java ...
mean response time (millisecs): 56
median response time (millisecs): 37
throughput = total number of requests/wall time (requests/second): 1604.3252609033955
p99 (99th percentile) response time: 339
min response time (millisecs): 14
max response time (millisecs): 1149

Process finished with exit code 0
```

d. Running on EC2
   Failed with Connection Error when trying to create 150 client threads.
   32 threads * 1k requests
   150 threads * 2.8k requests
(with extra 1 thread writing data to file and 1 thread generating liftRide Event)

e. Running locally
   32 threads * 1k requests
   32 threads * 5.25k requests

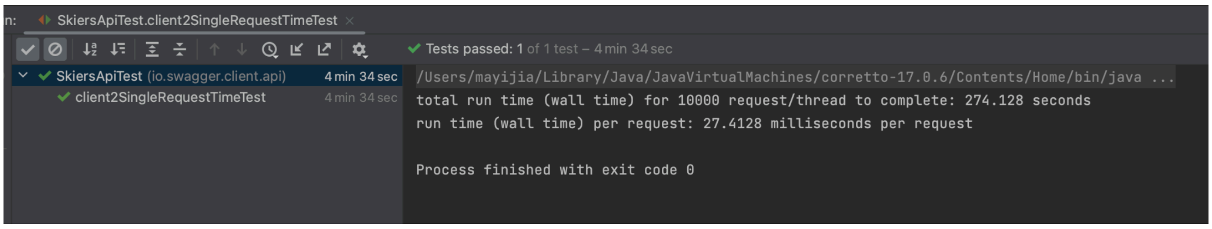


```
/Users/mayijia/Library/Java/JavaVirtualMachines/corretto-17.0.6/Contents/Ho
1. number of successful requests sent: 200000
2. number of unsuccessful requests (should be 0): 0
3. total run time (wall time) for all phases to complete: 51.543 seconds
4. total throughput in requests per second : 3880.2553208001086

Process finished with exit code 0
```



```
/Users/mayijia/Library/Java/JavaVirtualMachines/corretto-17.0.6/Contents/Home/bin/java ..
mean response time (millisecs): 7
median response time (millisecs): 3
throughput = total number of requests/wall time (requests/second): 3880.2553208001086
p99 (99th percentile) response time: 116
min response time (millisecs): 0
max response time (millisecs): 382
```

f. Single Thread sending 10000 requests


SkiersApiTest.client2SingleRequestTimeTest
Tests passed: 1 of 1 test – 4 min 34 sec
SkiersApiTest (io.swagger.client.api) 4 min 34 sec
client2SingleRequestTimeTest 4 min 34 sec

```
/Users/mayijia/Library/Java/JavaVirtualMachines/corretto-17.0.6/Contents/Home/bin/java ...
total run time (wall time) for 10000 request/thread to complete: 274.128 seconds
run time (wall time) per request: 27.4128 milliseconds per request

Process finished with exit code 0
```

5.3. Little's Law

a. Observation

We'll use 32 client threads to analyze it since this case maintains a constant long-term average number of clients. The min number of threads = 32, this is the long term average requests present in the system. Following the little's law, the arrival rate = 32/27(ms) = 1185 requests per second. However, the actual arrival rate(total throughput in request per second) we observed is 963 per second. For best throughput, the estimated arrival rate range is from 32/27(ms) = 1185 requests per second to 120/27(ms) = 4444 requests per second while the actual throughput is 1604 requests per second.

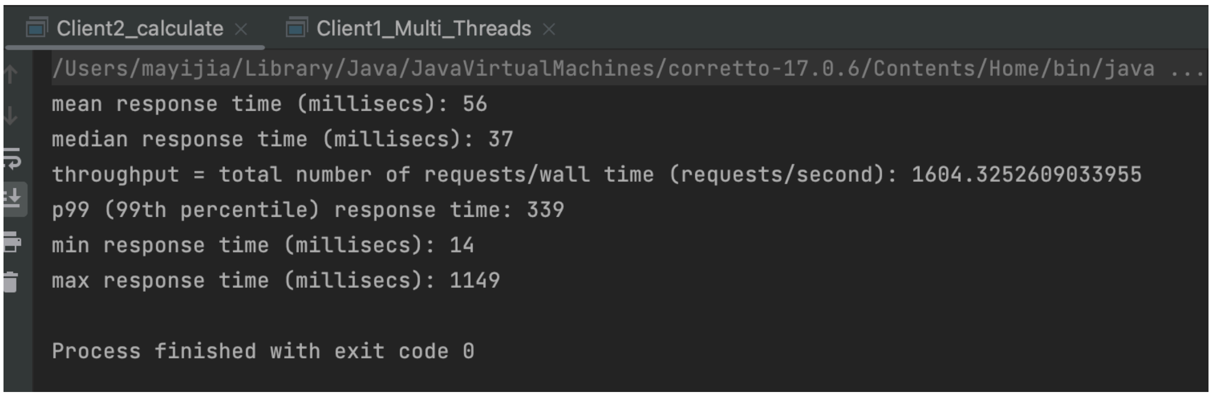
b. Justification

The significant delay may result from the synchronized functions I implemented to increment and retrieve successCounter and failCounter. These synchronized functions introduce a lock on the instance, forcing threads to wait for each other until the lock is released.

In a single-threaded scenario, the client requests are sent sequentially, and there is no contention for shared data. However, in a multi-threaded environment, when multiple threads need access to shared data, the synchronized functions impose a lock that allows only one thread to access the data at a time. Consequently, every other thread must wait until the current thread exits the method, leading to considerable time wastage.

In addition, when we use a single thread to send 10000 requests, we are using one client and possibly using the same connection to the server while in a multithread case, we spend more time establishing a connection.

c. Data processing



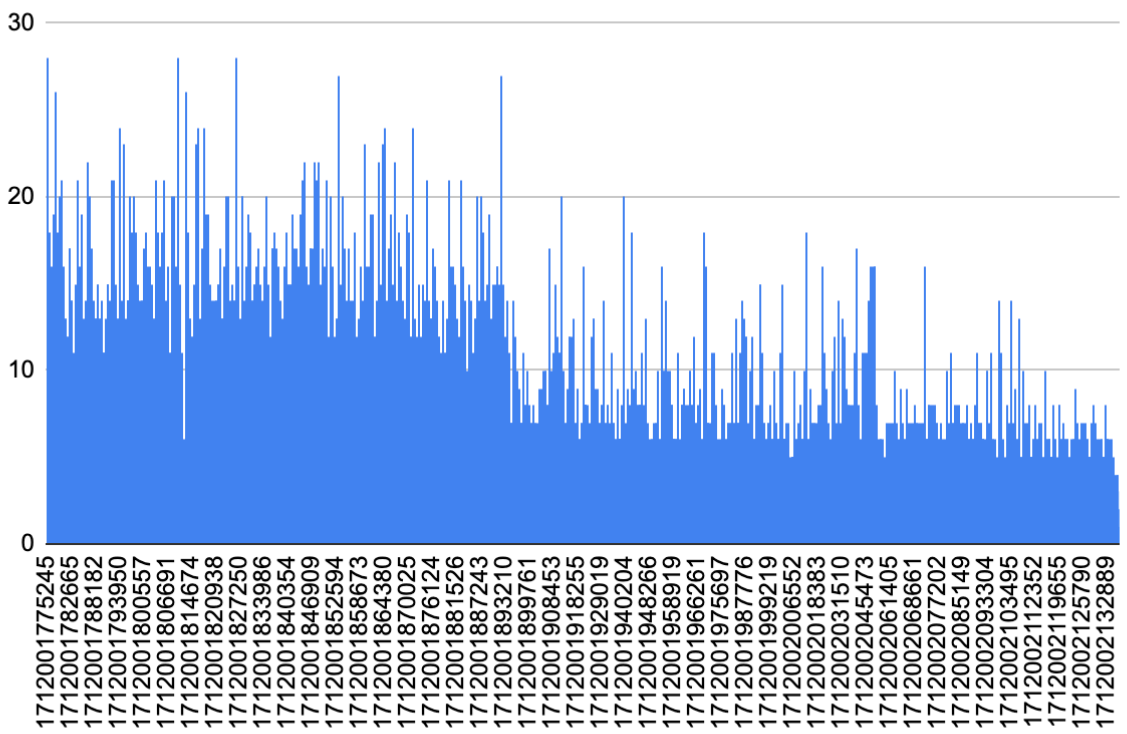
```
/Users/mayijia/Library/Java/JavaVirtualMachines/corretto-17.0.6/Contents/Home/bin/java ...
mean response time (millisecs): 56
median response time (millisecs): 37
throughput = total number of requests/wall time (requests/second): 1604.3252609033955
p99 (99th percentile) response time: 339
min response time (millisecs): 14
max response time (millisecs): 1149

Process finished with exit code 0
```

I created a class Client2_calculate to do the data processing after we got all the information written to the output.csv. Detailed output is included as part of the output window.

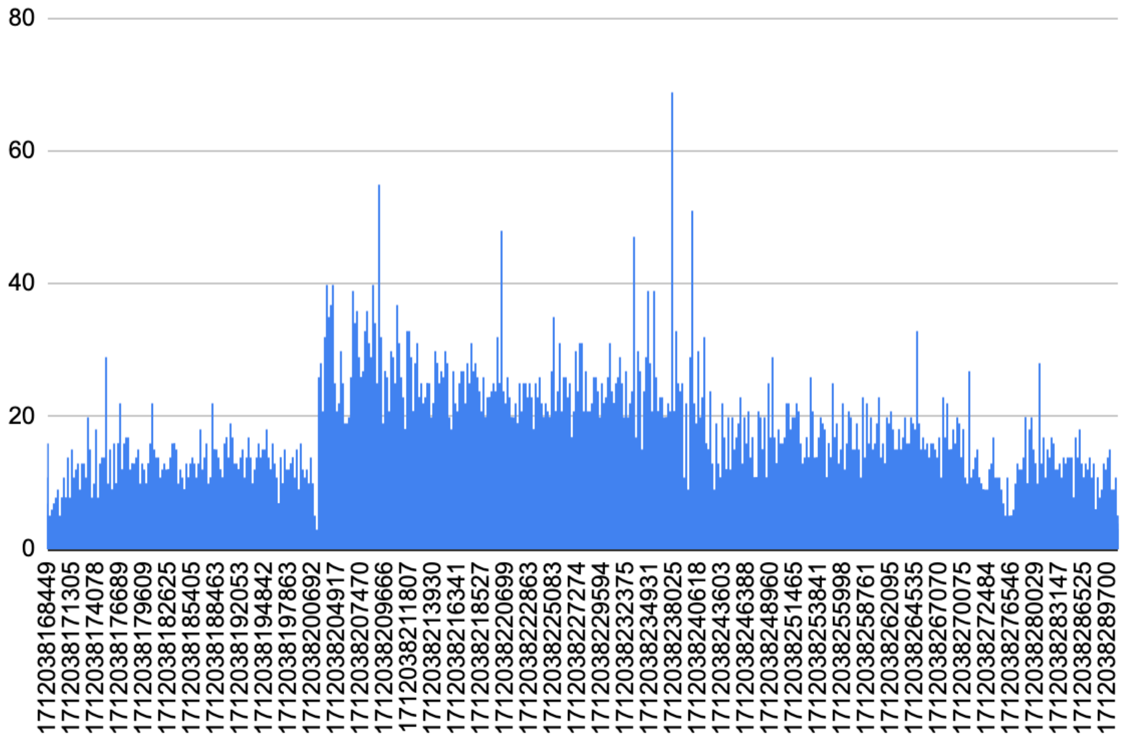
6. Plot of Throughput

April-output

a. 32 threads * 1k requests, 32 threads * 5.25k requests



b. 32 threads * 1k requests, 120 threads * 1.4k requests(best throughput)


80
60
40
20
0
1712038168449
1712038171305
1712038174078
1712038176689
1712038179609
1712038182625
1712038185405
1712038188463
1712038192053
1712038194842
1712038197863
1712038200692
1712038204917
1712038207470
1712038209666
1712038211807
1712038213930
1712038216341
1712038218527
1712038220699
1712038222863
1712038225083
1712038227274
1712038229594
1712038232375
1712038234931
1712038238025
1712038240618
1712038243603
1712038246388
1712038248960
1712038251465
1712038253841
1712038255998
1712038258761
1712038262095
1712038264535
1712038267070
1712038270075
1712038272484
1712038276546
1712038280029
1712038283147
1712038286525
1712038289700