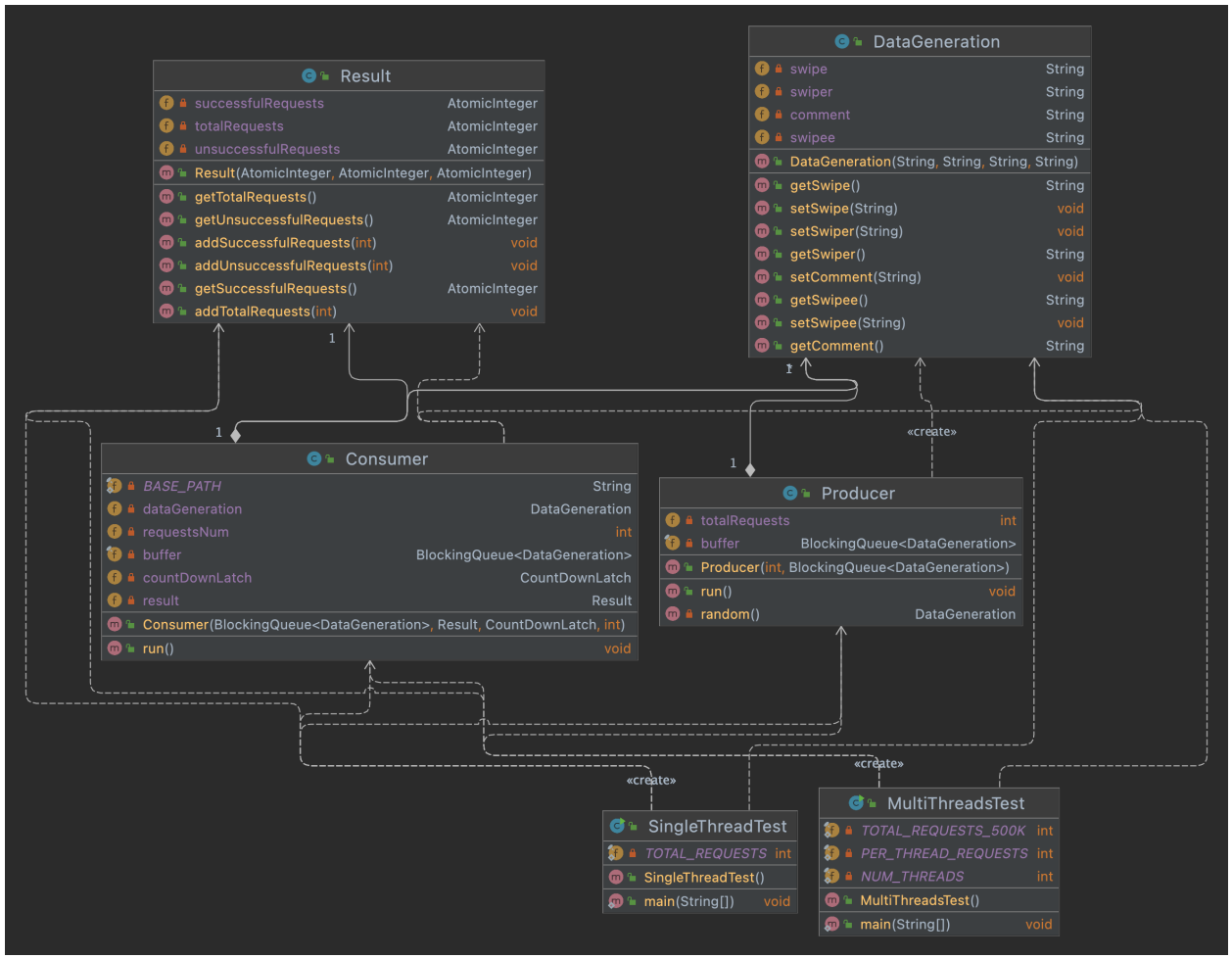
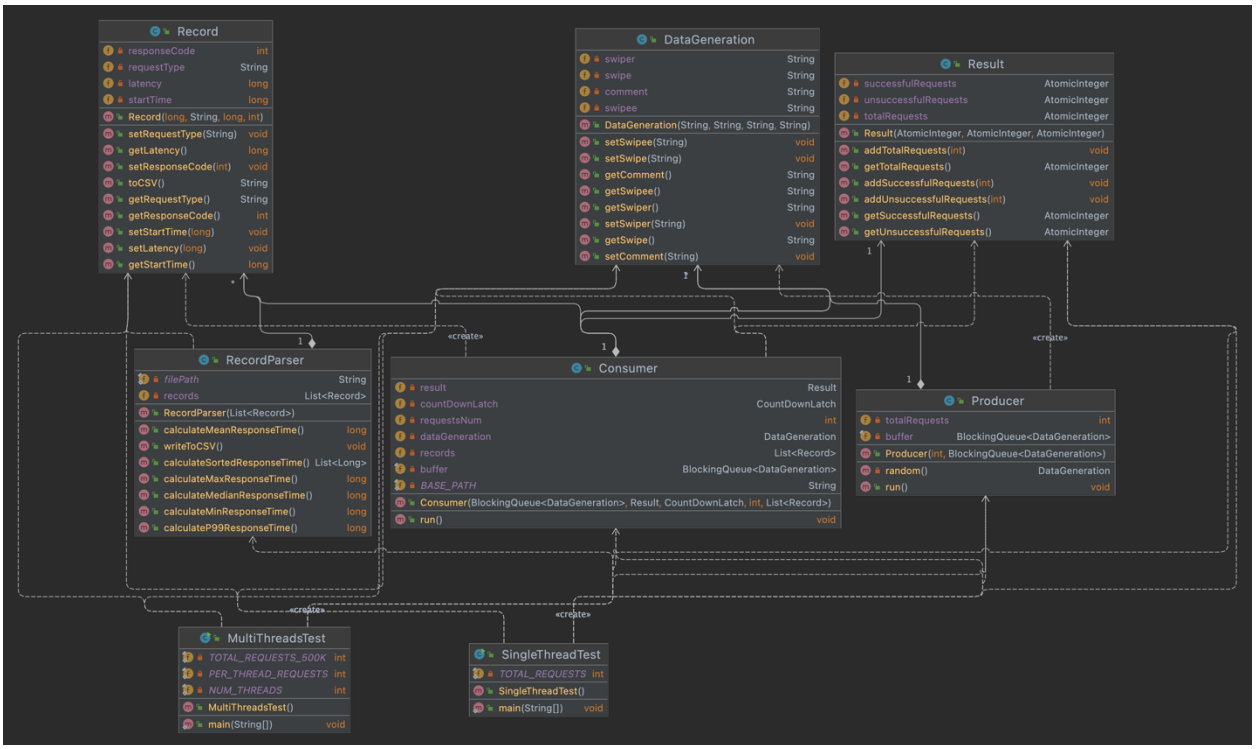
1. Siyuan Chen's GitHub Repo: https://github.com/Janice1457/Building-Scalable-Distributed-Systems/tree/main/HW1

Part1-UML



Part2-UML



2. Design:

Client Part1:

I have read the textbook, and decided to use Producer and Consumer with a LinkedBlockingQueue pattern since This solution absolves the programmer from being concerned with the implementation of coordinating access to the shared buffer, and greatly simplifies the code.

There are six classes, including Producer, Consumer, DataGeneration, Result, SingleThreadTest and MultiThreadTest. And DataGeneration and Result are in Models package.

DataGeneration: has 4 fields, including swipe, swiper, swipee, and comment. For each POST my client needs to randomly generate values for the request. And the randomly generated values will be put into a buffer by producer.

Result: has 3 fields, including successfulRequests, unsuccessfulRequests and totalRequests. This will help print out the number of successful requests sent, the number of unsuccessful requests, and the total number of requests.

Producer: Producers generate and send messages via a shared FIFO buffer to consumers. My producer will put randomly generated dataGeneration message to the buffer.

Consumer: Consumers retrieve these messages, process them, and then ask for more work from the buffer. For each request, the consumer will take the dataGeneration message from the buffer. And using the message we get from the buffer to get the response. If the statusCode is 200 or 201, then add up one successful request, if the statusCode is not 200 or 201, it will retry 5 times, and add to the unsuccessful request when it still fails after 5 retries.

SingleThreadTest: Run a simple test and send 10000 requests from a single thread to test how long a single request takes to estimate my latency. Instantiate a start time, an end time, a producer, a buffer as LinkedBlockingQueue, a Result with values be 0, and a Consumer. And then start producer and consumer thread. The consumer thread needs to wait until another thread terminates.

MultiThreadTest: Run a multithreading test and send 500k requests to test the results we need. The difference between single thread test is that the consumer will use different number of threads to execute. And I test when the number of threads to be 50,100 and 200. And the total number of requests here is 500k.

Client Part2:

There are 8 classes. And apart from the classes in part 1, I add another two classes, including Record and RecordParser.

DataGeneration, **Result**, **Producer** and **SingleThreadTest** classes are the same as part 1.

Consumer: I add start and end timestamp before and after each request, and write out a record object containing {start time, request type (POST), latency, response code}. And instantiate a new record for each quest and use a list called records to store each record.

Record: has 4 fields including start time, request type (POST), latency, response code.

RecordParser: has 7 methods.
1) writeToCSV: will parse each value in records and write them to a csv file. And the head of the CSV file is start time, request type (POST), latency, response code.

2) calculateMeanResponseTime: calculate total latency and then divide by number of records.

3) calculateSortedResponseTime: sort the latency of all records.

4) calculateMedianResponseTime: first call calculateSortedResponseTime to get total sorted response time, and then calculate if the szie of total response time is odd or even. If it is odd, then the median is located in the middle, if it is even, then calculate the two in the middle.

5) calculateP99ResponseTime: According to the website explanation, if I want to calculate the 99th percentile, I need to sort all my values from least to greatest, then find the value at myArray[count(myArray) * 0.99]

6) calculateMinResponseTime: the value in first index of sorted list.

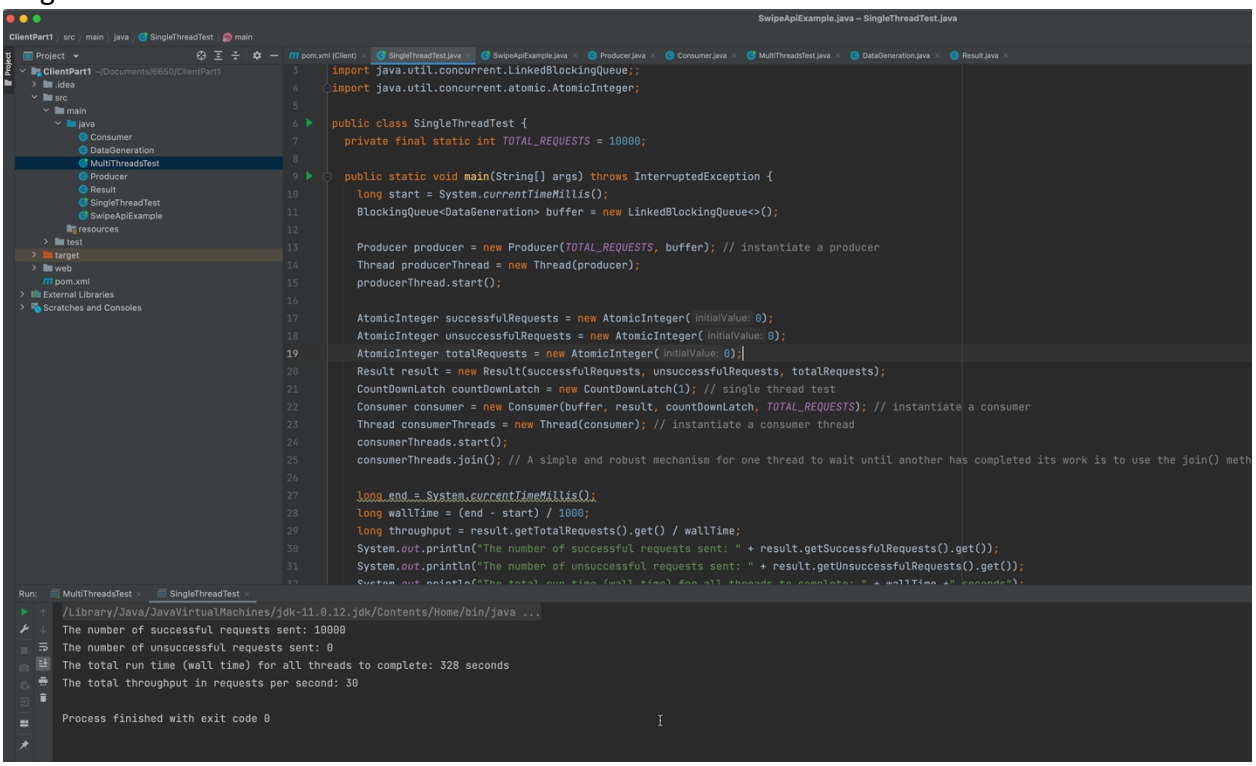
7) calculateMaxResponseTime: the value in last index of sorted list.

MultiThreadTest:

The difference from part 1 is that I need to instantiate a recordParser and print out the mean response time, median response time, throughput, p99 response time, min response time and max response time.

3. Client Part 1:

SingleThreadTest:



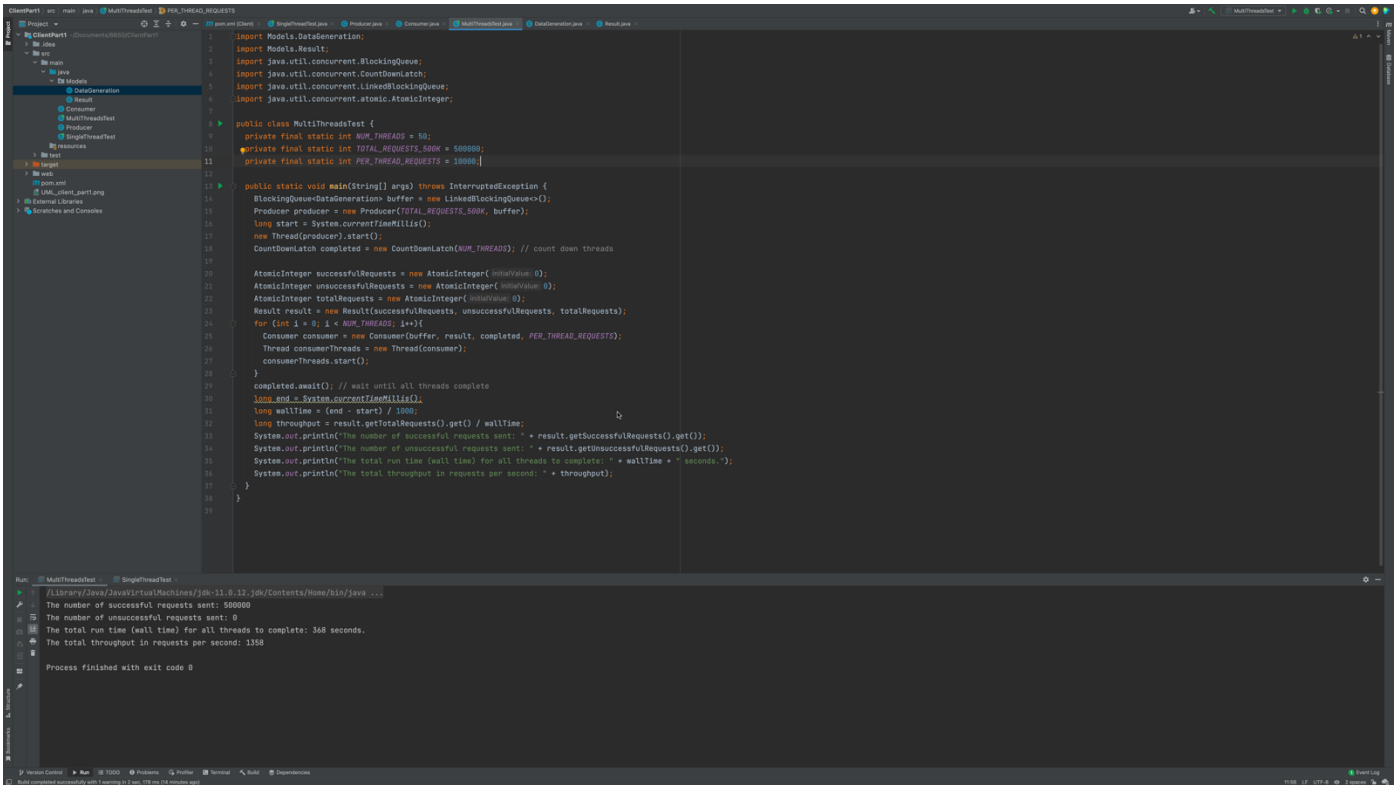
- each request takes 328/10000= 0.0328 s (W)
- If I have 50 threads, and hence the maximum is 50 concurrent requests (N), I can use Little Laws to estimate throughput as 50/0.0328 =1524 requests per second.

- If I have 100 threads, and hence the maximum is 100 concurrent requests (N), I can use Little Laws to estimate throughput as 100/0.0328 =3048 requests per second.

- If I have 200 threads, and hence the maximum is 200 concurrent requests (N), I can use Little Laws to estimate throughput as 200/0.0328 =6097 requests per second.

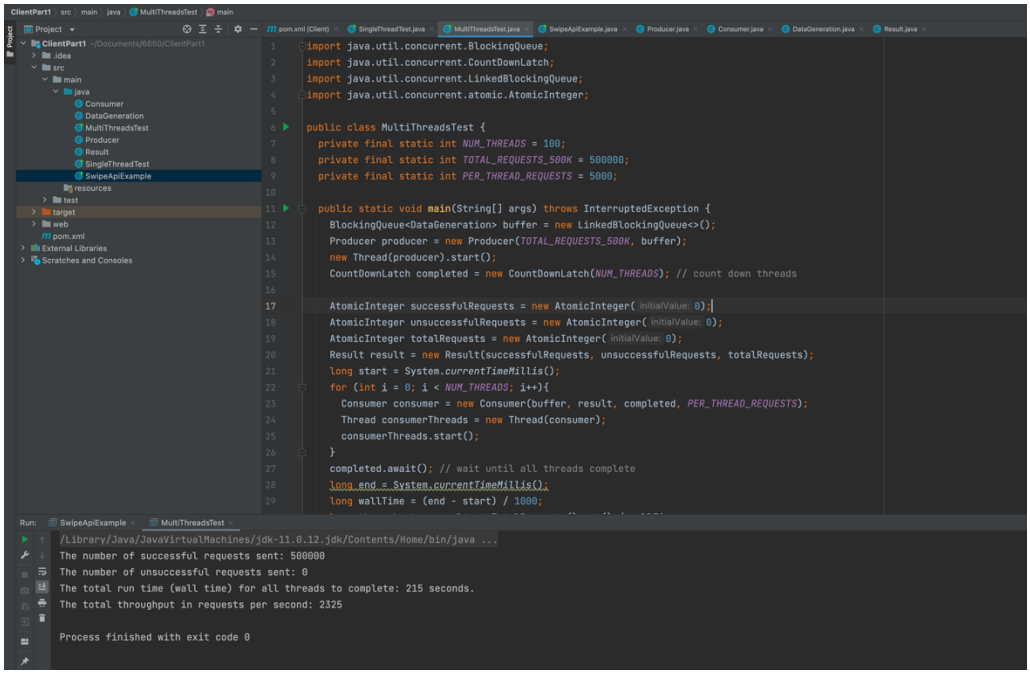
MultiThreadTest:

1. Threads = 50



The throughput is about 1358, which is close to Little's law's prediction 1524.

2. Threads = 100

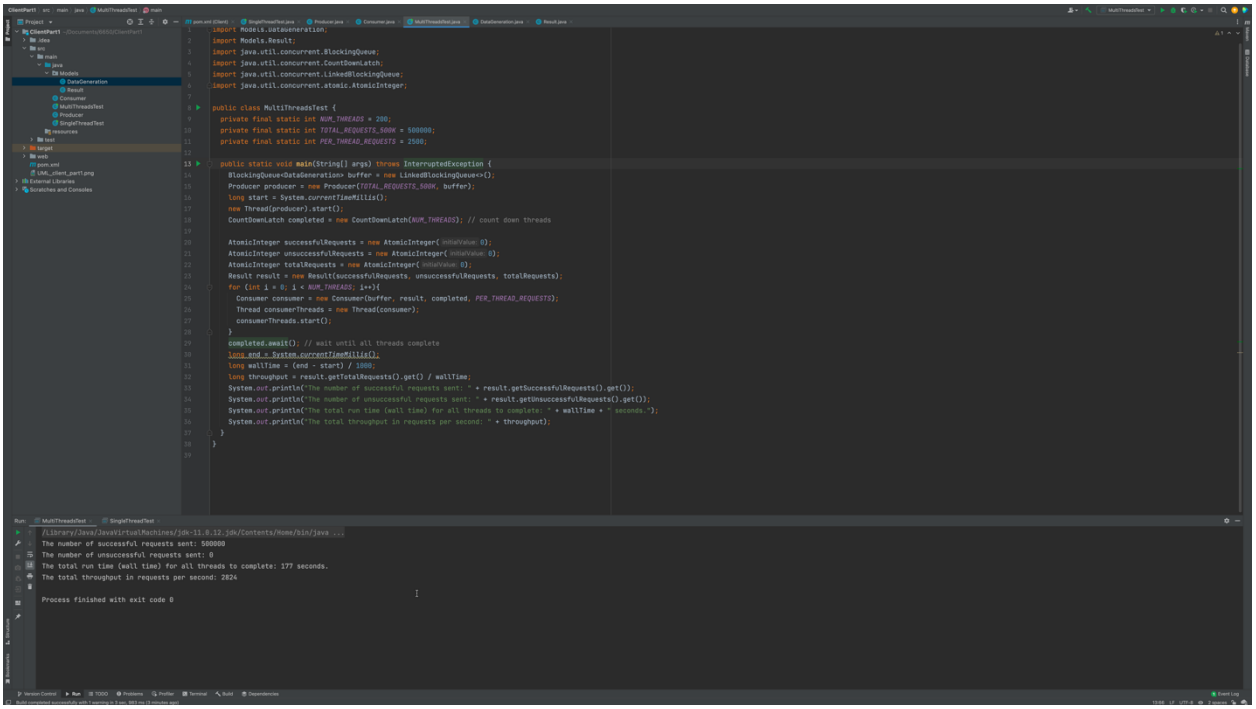


```
/Library/Java/JavaVirtualMachines/jdk-11.0.12.jdk/Contents/Home/bin/java ...
The number of successful requests sent: 500000
The number of unsuccessful requests sent: 0
The total run time (wall time) for all threads to complete: 215 seconds.
The total throughput in requests per second: 2325

Process finished with exit code 0
```

The throughput is about 2325, which is close to Little's law's prediction 3048

3. Threads = 200



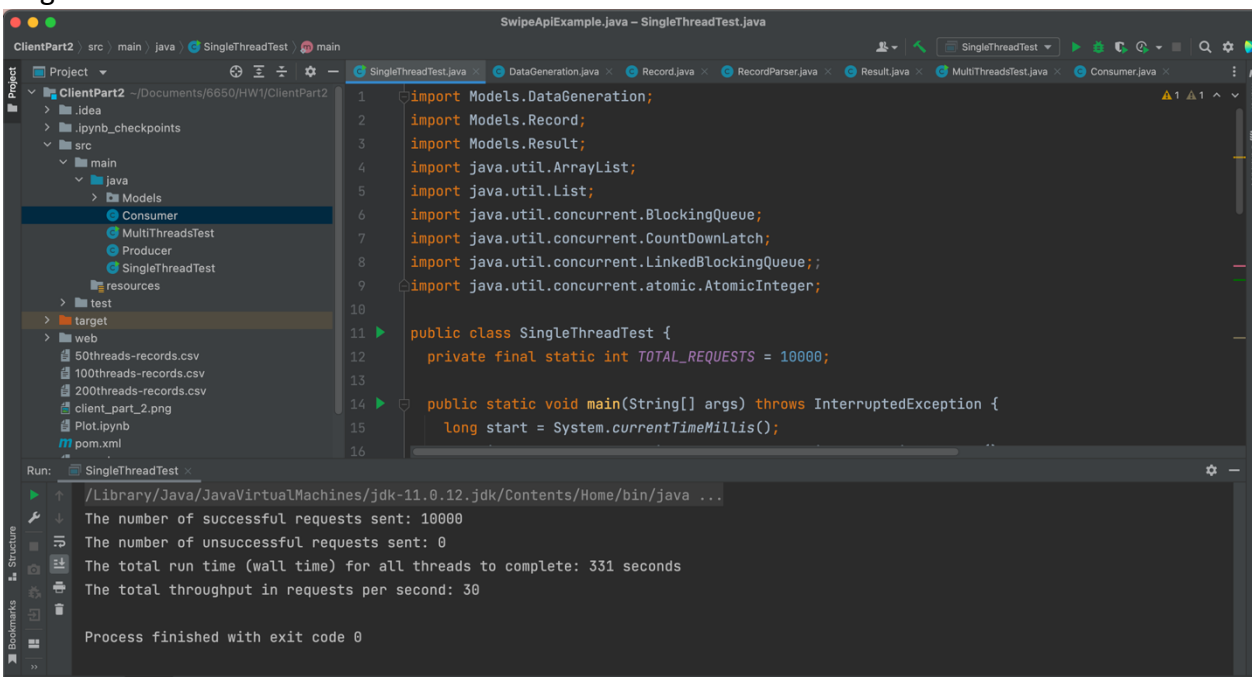
```
/Library/Java/JavaVirtualMachines/jdk-11.0.12.jdk/Contents/Home/bin/java ...
The number of successful requests sent: 500000
The number of unsuccessful requests sent: 0
The total run time (wall time) for all threads to complete: 177 seconds.
The total throughput in requests per second: 2824

Process finished with exit code 0
```

The throughput is about 2824, which is close to Little's law's prediction 6097

4. Client Part 2

SingleThreadTest:



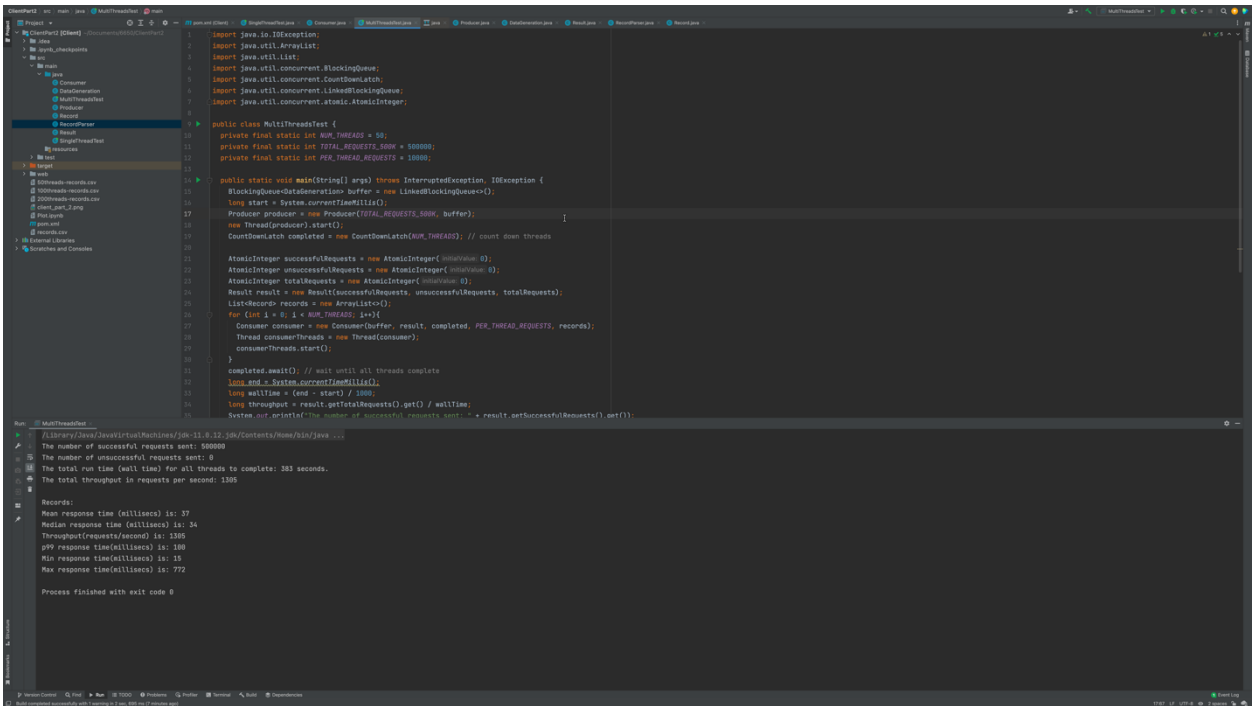
- each request takes 331/10000= 0.0331 s (W)
- If I have 50 threads, and hence the maximum is 50 concurrent requests (N), I can use Little Laws to estimate throughput as 50/0.0331 =1510 requests per second.

- If I have 100 threads, and hence the maximum is 100 concurrent requests (N), I can use Little Laws to estimate throughput as 100/0.0331 =3021 requests per second.

- If I have 200 threads, and hence the maximum is 200 concurrent requests (N), I can use Little Laws to estimate throughput as 200/0.0331 =6042 requests per second.

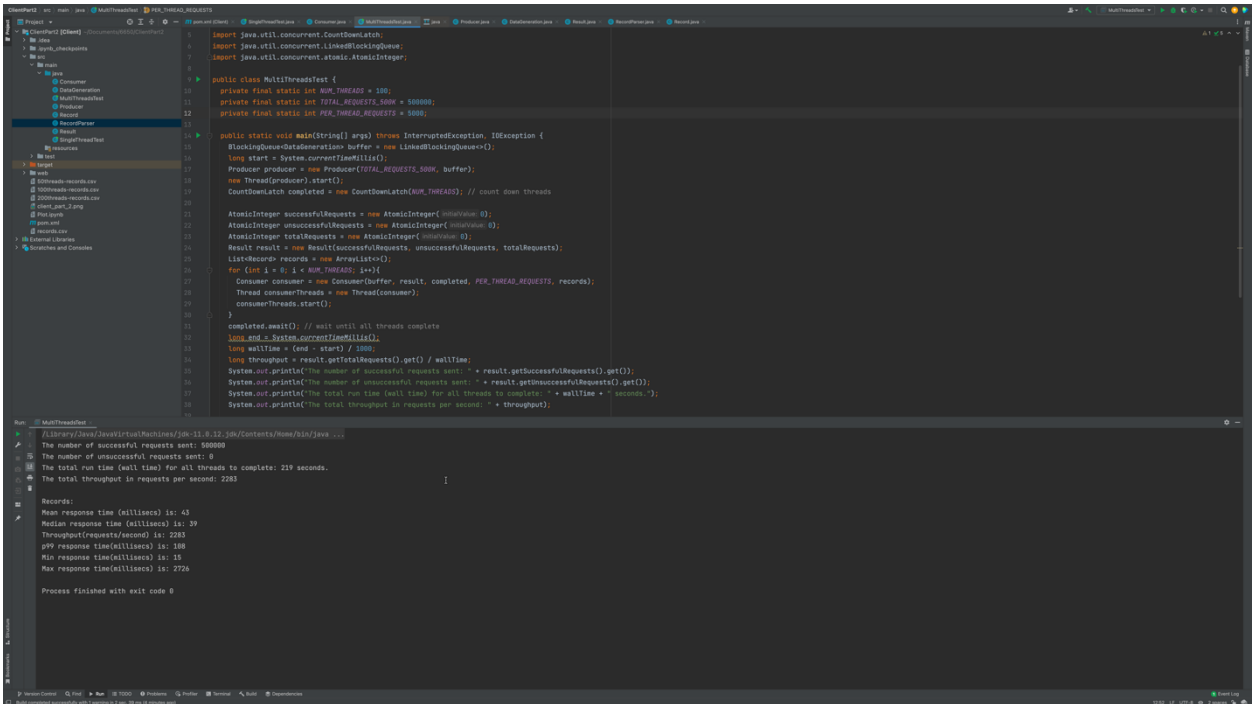
Multithread Test

1. Thread = 50



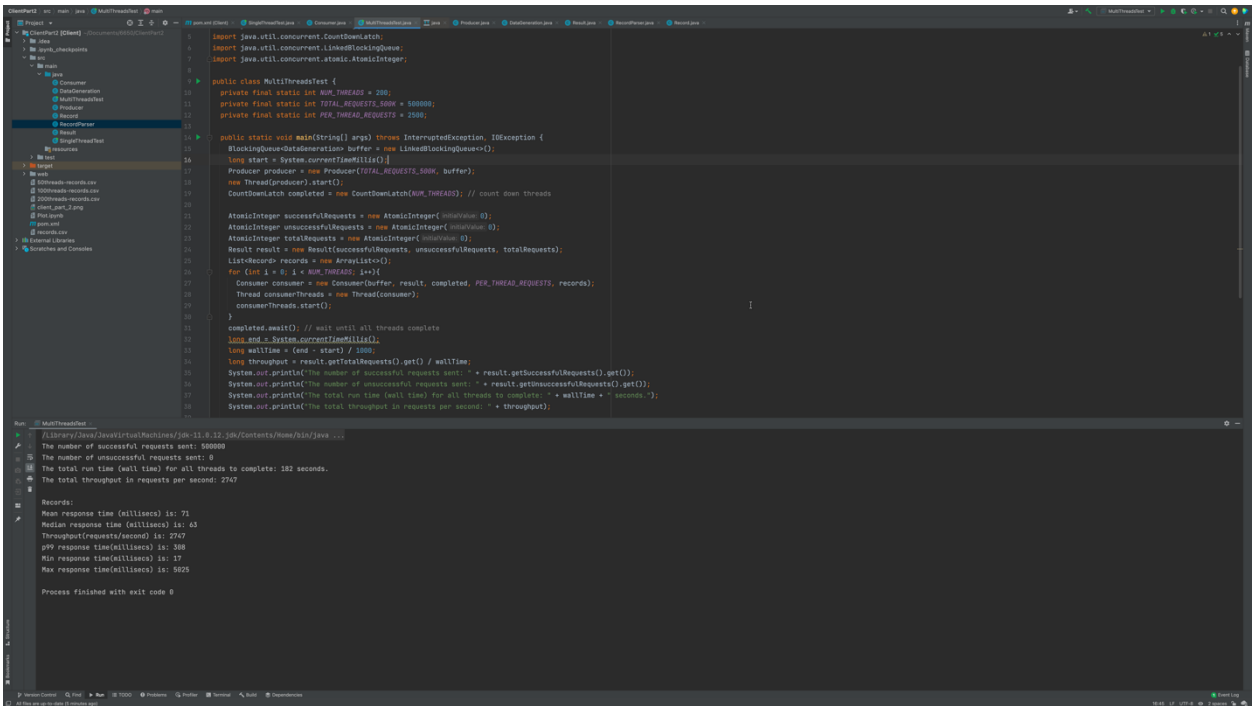
The throughput in part1 is about 1358, the throughput in part2 is about 1305, which is within 5% of client part 1.

2. Thread = 100



The throughput in part1 is about 2325, the throughput in part2 is about 2203, which is within 5% of client part 1.

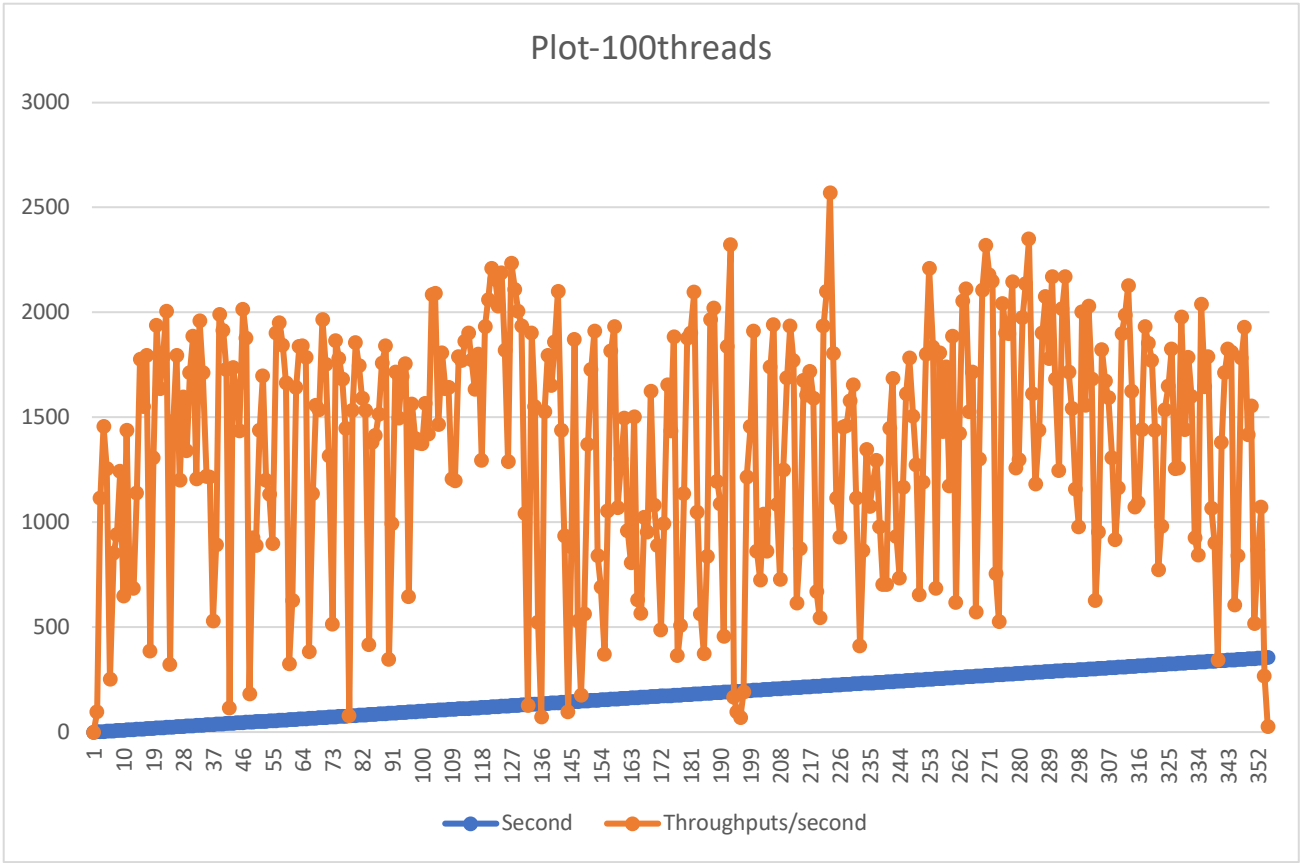
3. Thread = 200



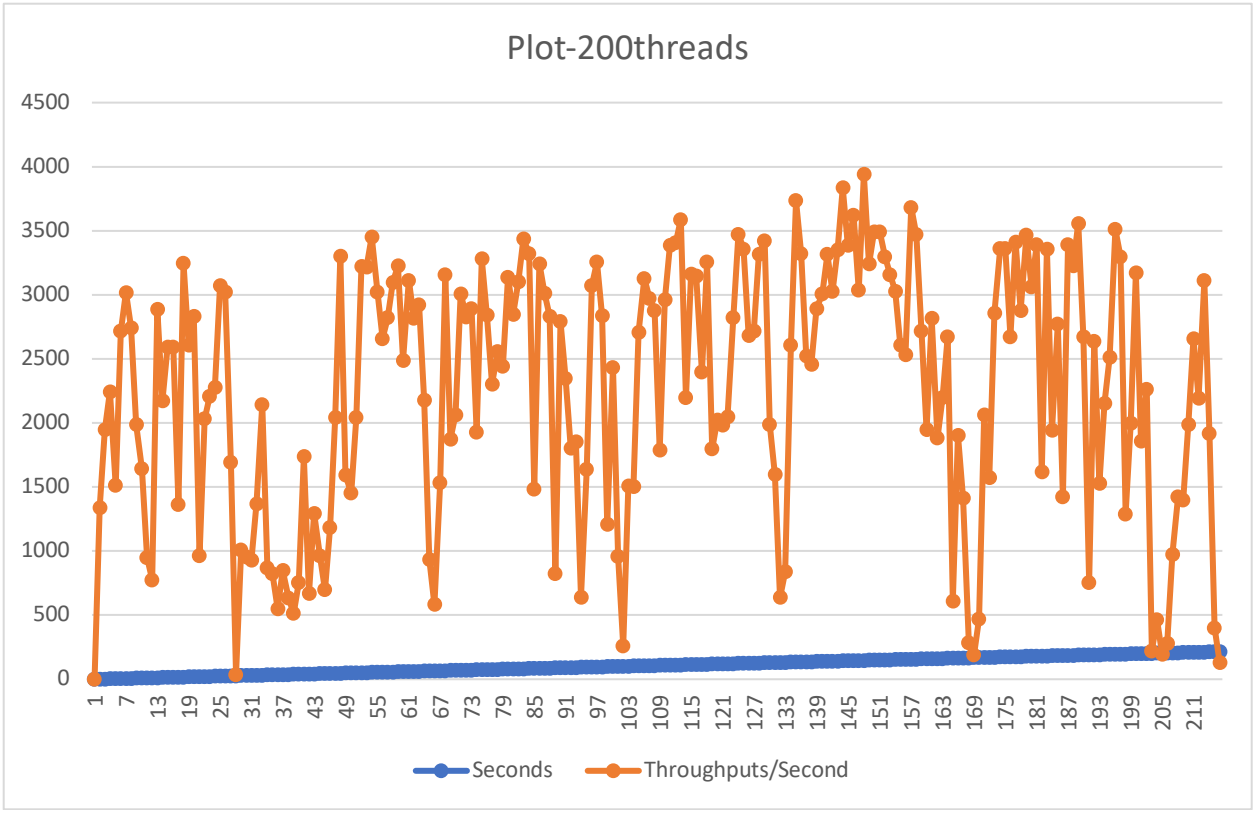
The throughput in part1 is about 2824, the throughput in part2 is about 2747, which is within 5% of client part 1.

5. Plot Performance:

1.The following Plot is one test performance when the number of threads is 100.



2.This plot is one test performance when the number of threads is 200


Plot-200threads
4500
4000
3500
3000
2500
2000
1500
1000
500
0
1
7
13
19
25
31
37
43
49
55
61
67
73
79
85
91
97
103
109
115
121
127
133
139
145
151
157
163
169
175
181
187
193
199
205
211
Seconds
Throughputs/Second