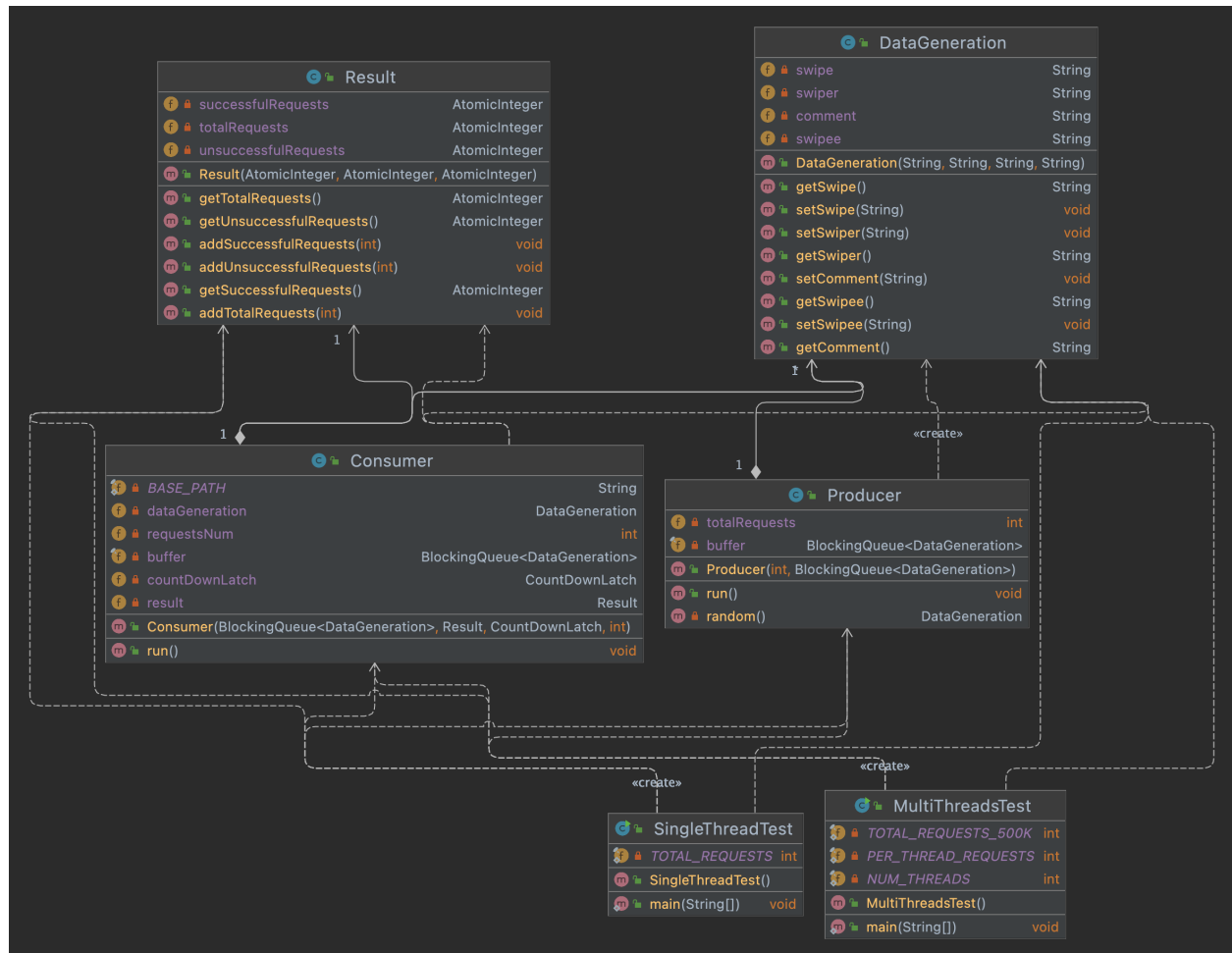
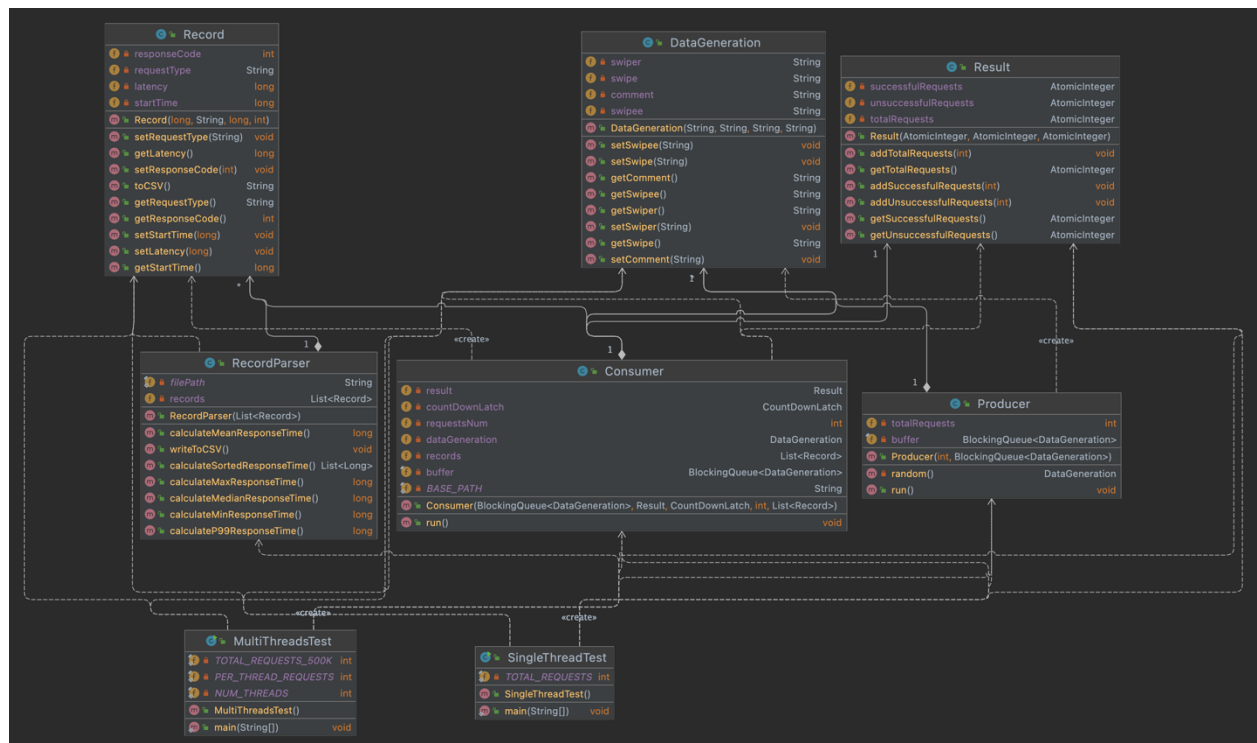


1. Siyuan Chen's GitHub Repo:  
<https://github.com/Janice1457/Building-Scalable-Distributed-Systems/tree/main/HW1>

## 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 size 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  $\text{myArray}[\text{count}(\text{myArray}) * 0.99]$

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

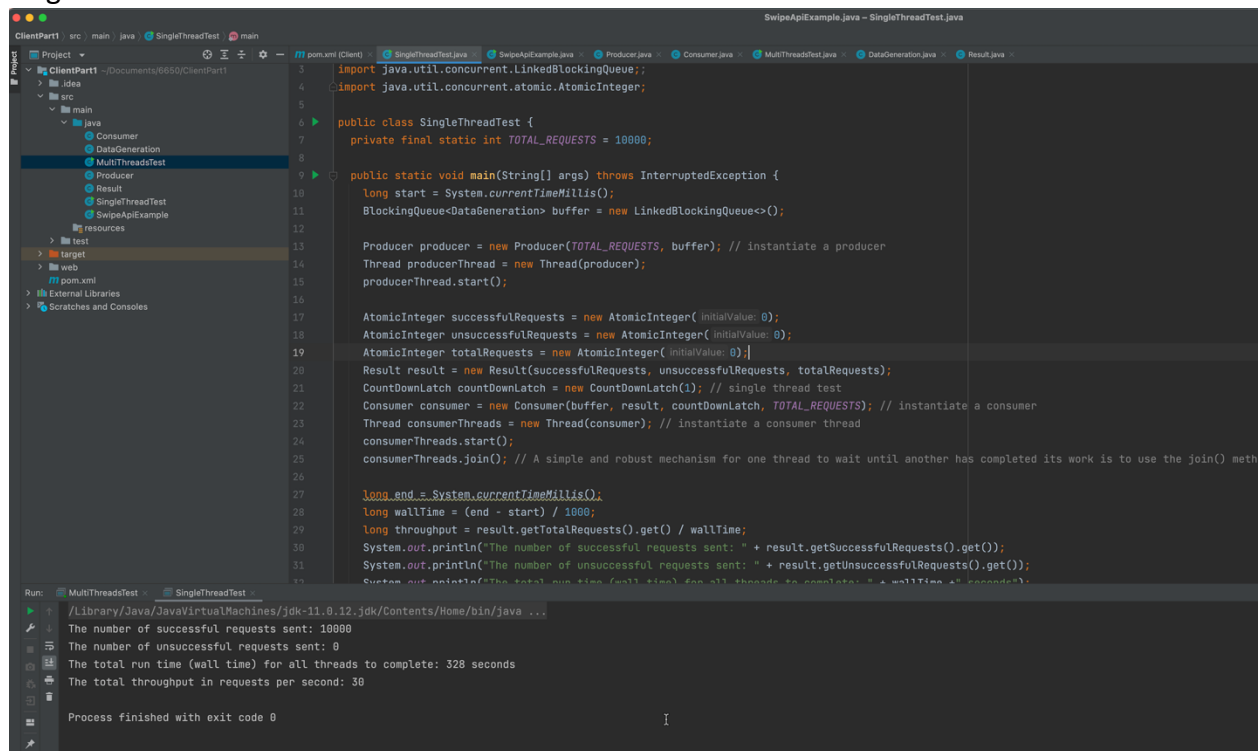
7) calculateMaxResponseTime: the value in last index of sorted list.

## 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:



```
ClientPart1 | src | main | java | SingleThreadTest | main | pom.xml (Client) | SingleThreadTest.java | SwipeApiExample.java | Producer.java | Consumer.java | MultiThreadTest.java | DataGeneration.java | Result.java

3 import java.util.concurrent.LinkedBlockingQueue;
4 import java.util.concurrent.atomic.AtomicInteger;
5
6 public class SingleThreadTest {
7     private final static int TOTAL_REQUESTS = 10000;
8
9     public static void main(String[] args) throws InterruptedException {
10         long start = System.currentTimeMillis();
11         BlockingQueue<DataGeneration> buffer = new LinkedBlockingQueue<>();
12
13         Producer producer = new Producer(TOTAL_REQUESTS, buffer); // instantiate a producer
14         Thread producerThread = new Thread(producer);
15         producerThread.start();
16
17         AtomicInteger successfulRequests = new AtomicInteger(initialValue: 0);
18         AtomicInteger unsuccessfulRequests = new AtomicInteger(initialValue: 0);
19         AtomicInteger totalRequests = new AtomicInteger(initialValue: 0);
20         Result result = new Result(successfulRequests, unsuccessfulRequests, totalRequests);
21         CountDownLatch countDownLatch = new CountDownLatch(1); // single thread test
22         Consumer consumer = new Consumer(buffer, result, countDownLatch, TOTAL_REQUESTS); // instantiate a consumer
23         Thread consumerThread = new Thread(consumer); // instantiate a consumer thread
24         consumerThread.start();
25         consumerThread.join(); // A simple and robust mechanism for one thread to wait until another has completed its work is to use the join() meth
26
27         long end = System.currentTimeMillis();
28         long wallTime = (end - start) / 1000;
29         long throughput = result.getTotalRequests().get() / wallTime;
30         System.out.println("The number of successful requests sent: " + result.getSuccessfulRequests().get());
31         System.out.println("The number of unsuccessful requests sent: " + result.getUnsuccessfulRequests().get());
32         System.out.println("The total run time (wall time) for all threads to complete: " + wallTime + " seconds");
33     }
34 }
```

Run: MultiThreadTest | SingleThreadTest

```
/Library/Java/JavaVirtualMachines/jdk-11.0.12.jdk/Contents/Home/bin/java ...
The number of successful requests sent: 10000
The number of unsuccessful requests sent: 0
The total run time (wall time) for all threads to complete: 328 seconds
The total throughput in requests per second: 30
Process finished with exit code 0
```

- 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 screenshot shows an IDE with the following components:

- Project Explorer:** Shows a project named 'ClientPart' with sub-packages 'Models', 'DataGeneration', 'Result', 'Consumer', 'MultiThreadTest', 'Producer', and 'SingleThreadTest'.
- Code Editor:** Displays the 'MultiThreadTest.java' file. The code is as follows:
 

```

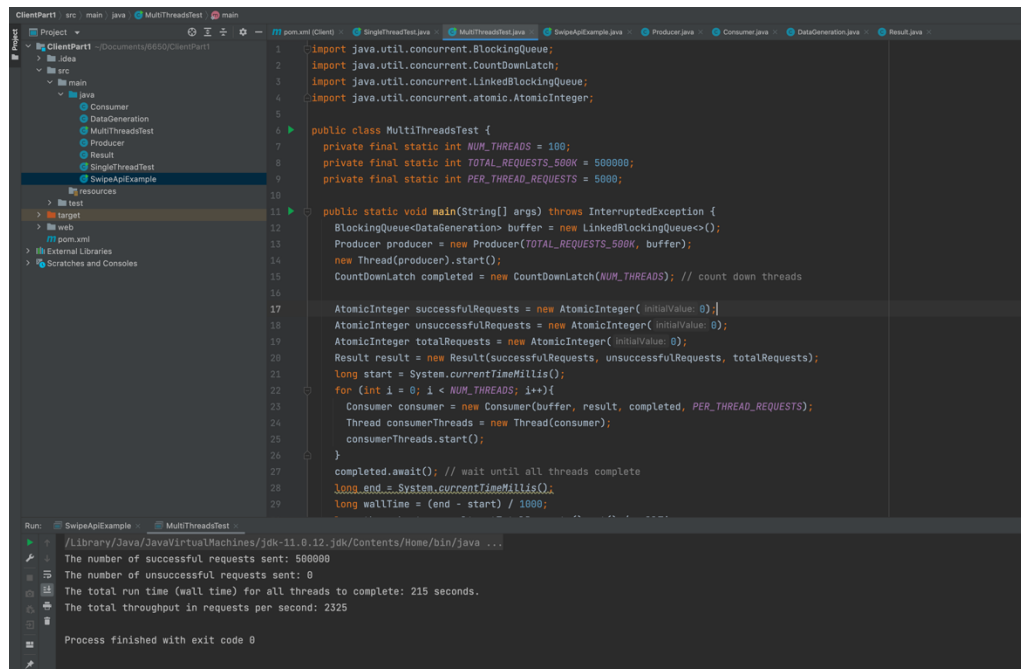
1 import Models.DataGeneration;
2 import Models.Result;
3 import java.util.concurrent.BlockingQueue;
4 import java.util.concurrent.CountDownLatch;
5 import java.util.concurrent.LinkedBlockingQueue;
6 import java.util.concurrent.atomic.AtomicInteger;
7
8 public class MultiThreadTest {
9     private final static int NUM_THREADS = 50;
10    private final static int TOTAL_REQUESTS_500K = 500000;
11    private final static int PER_THREAD_REQUESTS = 10000;
12
13    public static void main(String[] args) throws InterruptedException {
14        BlockingQueue<DataGeneration> buffer = new LinkedBlockingQueue<>();
15        Producer producer = new Producer(TOTAL_REQUESTS_500K, buffer);
16        long start = System.currentTimeMillis();
17        new Thread(producer).start();
18        CountDownLatch completed = new CountDownLatch(NUM_THREADS); // count down threads
19
20        AtomicInteger successfulRequests = new AtomicInteger(0);
21        AtomicInteger unsuccessfulRequests = new AtomicInteger(0);
22        AtomicInteger totalRequests = new AtomicInteger(0);
23        Result result = new Result(successfulRequests, unsuccessfulRequests, totalRequests);
24        for (int i = 0; i < NUM_THREADS; i++){
25            Consumer consumer = new Consumer(buffer, result, completed, PER_THREAD_REQUESTS);
26            Thread consumerThread = new Thread(consumer);
27            consumerThread.start();
28        }
29        completed.await(); // wait until all threads complete
30        long end = System.currentTimeMillis();
31        long wallTime = (end - start) / 1000;
32        long throughput = result.getTotalRequests().get() / wallTime;
33        System.out.println("The number of successful requests sent: " + result.getSuccessfulRequests().get());
34        System.out.println("The number of unsuccessful requests sent: " + result.getUnsuccessfulRequests().get());
35        System.out.println("The total run time (wall time) for all threads to complete: " + wallTime + " seconds.");
36        System.out.println("The total throughput in requests per second: " + throughput);
37    }
38 }
      
```
- Run Output:** Shows the execution results:
 

```

Run: MultiThreadTest
[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: 368 seconds.
The total throughput in requests per second: 1358
Process finished with exit code 0
      
```

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

## 2. Threads = 100



```
1 import java.util.concurrent.BlockingQueue;
2 import java.util.concurrent.CountDownLatch;
3 import java.util.concurrent.LinkedBlockingQueue;
4 import java.util.concurrent.atomic.AtomicInteger;
5
6 public class MultiThreadTest {
7     private final static int NUM_THREADS = 100;
8     private final static int TOTAL_REQUESTS_500K = 500000;
9     private final static int PER_THREAD_REQUESTS = 5000;
10
11     public static void main(String[] args) throws InterruptedException {
12         BlockingQueue<DataGeneration> buffer = new LinkedBlockingQueue<>();
13         Producer producer = new Producer(TOTAL_REQUESTS_500K, buffer);
14         new Thread(producer).start();
15         CountDownLatch completed = new CountDownLatch(NUM_THREADS); // count down threads
16
17         AtomicInteger successfulRequests = new AtomicInteger(initialValue: 0);
18         AtomicInteger unsuccessfulRequests = new AtomicInteger(initialValue: 0);
19         AtomicInteger totalRequests = new AtomicInteger(initialValue: 0);
20         Result result = new Result(successfulRequests, unsuccessfulRequests, totalRequests);
21         long start = System.currentTimeMillis();
22         for (int i = 0; i < NUM_THREADS; i++){
23             Consumer consumer = new Consumer(buffer, result, completed, PER_THREAD_REQUESTS);
24             Thread consumerThreads = new Thread(consumer);
25             consumerThreads.start();
26         }
27         completed.await(); // wait until all threads complete
28         long end = System.currentTimeMillis();
29         long wallTime = (end - start) / 1000;
```

Run: MultiThreadTest

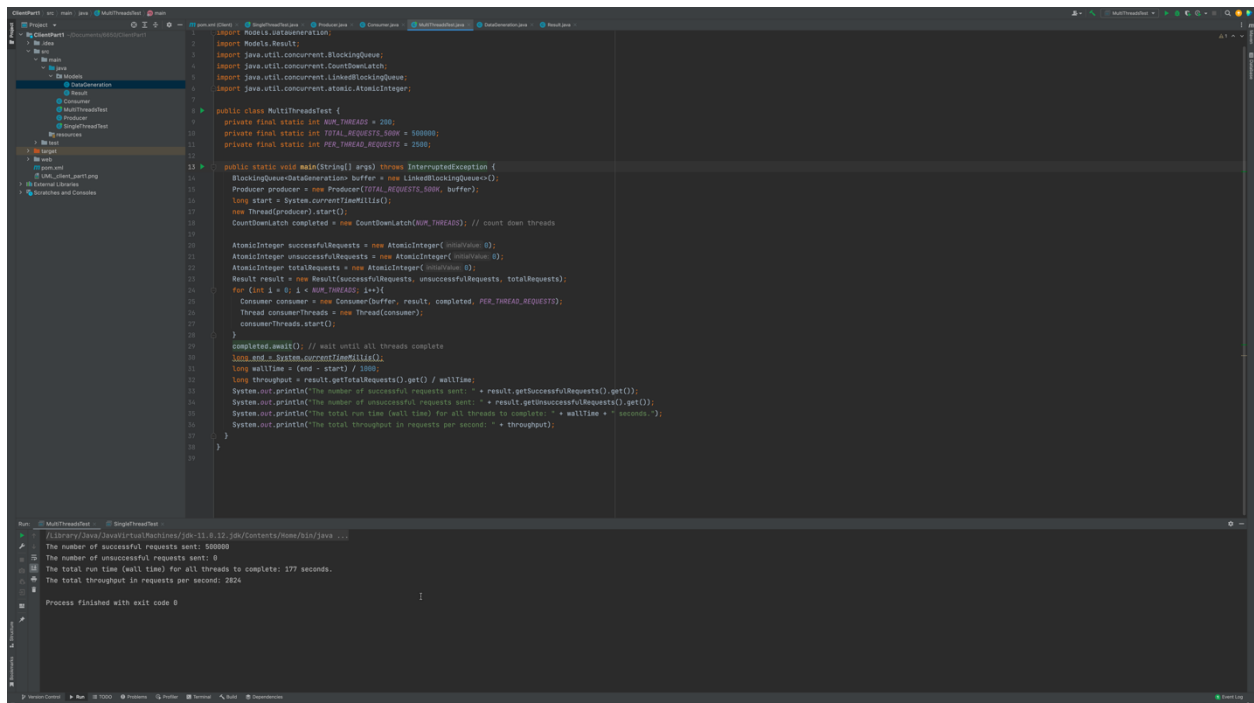
/Library/Java/JavaVirtualMachines/jdk-11.0.12\_jdk/Contents/Home/bin/java ...

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



```
1 import Models.DataGeneration;
2 import Models.Result;
3 import java.util.concurrent.BlockingQueue;
4 import java.util.concurrent.CountDownLatch;
5 import java.util.concurrent.LinkedBlockingQueue;
6 import java.util.concurrent.atomic.AtomicInteger;
7
8 public class MultiThreadTest {
9     private final static int NUM_THREADS = 200;
10     private final static int TOTAL_REQUESTS_500K = 500000;
11     private final static int PER_THREAD_REQUESTS = 2500;
12
13     public static void main(String[] args) throws InterruptedException {
14         BlockingQueue<DataGeneration> buffer = new LinkedBlockingQueue<>();
15         Producer producer = new Producer(TOTAL_REQUESTS_500K, buffer);
16         new Thread(producer).start();
17         CountDownLatch completed = new CountDownLatch(NUM_THREADS); // count down threads
18
19         AtomicInteger successfulRequests = new AtomicInteger(initialValue: 0);
20         AtomicInteger unsuccessfulRequests = new AtomicInteger(initialValue: 0);
21         AtomicInteger totalRequests = new AtomicInteger(initialValue: 0);
22         Result result = new Result(successfulRequests, unsuccessfulRequests, totalRequests);
23         for (int i = 0; i < NUM_THREADS; i++){
24             Consumer consumer = new Consumer(buffer, result, completed, PER_THREAD_REQUESTS);
25             Thread consumerThreads = new Thread(consumer);
26             consumerThreads.start();
27         }
28         completed.await(); // wait until all threads complete
29         long end = System.currentTimeMillis();
30         long wallTime = (end - start) / 1000;
31         long throughput = result.getTotalRequests().get() / wallTime;
32         System.out.println("The number of successful requests sent: " + result.getSuccessfulRequests().get());
33         System.out.println("The number of unsuccessful requests sent: " + result.getUnsuccessfulRequests().get());
34         System.out.println("The total run time (wall time) for all threads to complete: " + wallTime + " seconds.");
35         System.out.println("The total throughput in requests per second: " + throughput);
36     }
37 }
```

Run: MultiThreadTest

/Library/Java/JavaVirtualMachines/jdk-11.0.12\_jdk/Contents/Home/bin/java ...

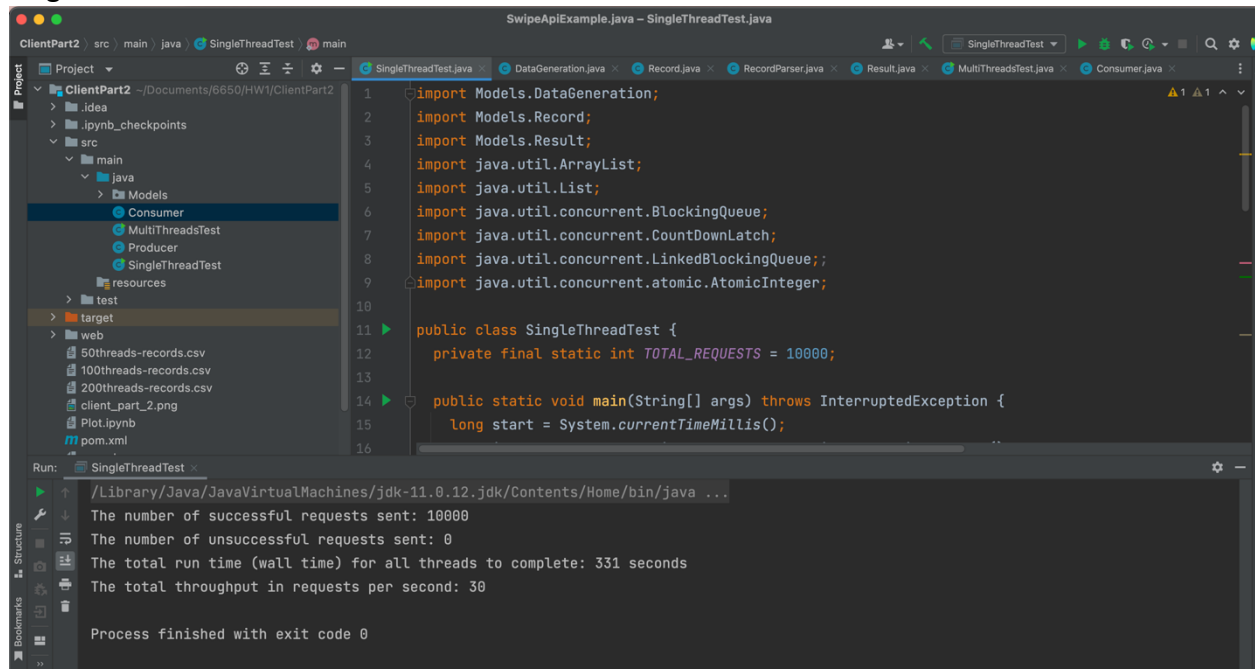
The number of successful requests sent: 500000  
The number of unsuccessful requests sent: 0  
The total run time (wall time) for all threads to complete: 177 seconds.  
The total throughput in requests per second: 2824

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:



The screenshot shows an IDE window titled "SwipeApiExample.java - SingleThreadTest.java". The left sidebar displays a project structure for "ClientPart2" with a "main" directory containing "java" and "resources". The "java" directory contains "Models" (with "Consumer", "MultiThreadTest", "Producer", and "SingleThreadTest") and "resources". The "main" directory also contains "target" and "web". The "web" directory contains "50threads-records.csv", "100threads-records.csv", "200threads-records.csv", "client\_part\_2.png", "Plot.ipynb", and "pom.xml". The "SingleThreadTest.java" file is open in the editor, showing the following code:

```
1 import Models.DataGeneration;
2 import Models.Record;
3 import Models.Result;
4 import java.util.ArrayList;
5 import java.util.List;
6 import java.util.concurrent.BlockingQueue;
7 import java.util.concurrent.CountDownLatch;
8 import java.util.concurrent.LinkedBlockingQueue;
9 import java.util.concurrent.atomic.AtomicInteger;
10
11 public class SingleThreadTest {
12     private final static int TOTAL_REQUESTS = 10000;
13
14     public static void main(String[] args) throws InterruptedException {
15         long start = System.currentTimeMillis();
16     }
```

The "Run" tab at the bottom shows the execution output for "SingleThreadTest":

```
/Library/Java/JavaVirtualMachines/jdk-11.0.12.jdk/Contents/Home/bin/java ...
The number of successful requests sent: 10000
The number of unsuccessful requests sent: 0
The total run time (wall time) for all threads to complete: 331 seconds
The total throughput in requests per second: 30
Process finished with exit code 0
```

- 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

```
1 import java.io.IOException;
2 import java.util.ArrayList;
3 import java.util.List;
4 import java.util.concurrent.BlockingQueue;
5 import java.util.concurrent.CountDownLatch;
6 import java.util.concurrent.LinkedBlockingQueue;
7 import java.util.concurrent.atomic.AtomicInteger;
8
9 public class MultiThreadTest {
10     private final static int NUM_THREADS = 50;
11     private final static int TOTAL_REQUESTS_500K = 50000;
12     private final static int PER_THREAD_REQUESTS = 1000;
13
14     public static void main(String[] args) throws InterruptedException, IOException {
15         BlockingQueue<DataGeneration> buffer = new LinkedBlockingQueue<>();
16         long start = System.currentTimeMillis();
17         Producer producer = new Producer(TOTAL_REQUESTS_500K, buffer);
18         new Thread(producer).start();
19         CountDownLatch completed = new CountDownLatch(NUM_THREADS); // count down threads
20
21         AtomicInteger successfulRequests = new AtomicInteger(0);
22         AtomicInteger unsuccessfulRequests = new AtomicInteger(0);
23         AtomicInteger totalRequests = new AtomicInteger(0);
24         Result result = new Result(successfulRequests, unsuccessfulRequests, totalRequests);
25         List<Record> records = new ArrayList<>();
26         for (int i = 0; i < NUM_THREADS; i++) {
27             Consumer consumer = new Consumer(buffer, result, completed, PER_THREAD_REQUESTS, records);
28             Thread consumerThread = new Thread(consumer);
29             consumerThread.start();
30         }
31         completed.await(); // wait until all threads complete
32         long end = System.currentTimeMillis();
33         long throughput = result.getTotalRequests().get() / wallTime;
34         System.out.println("The number of successful requests sent: " + result.getSuccessfulRequests().get());
35         System.out.println("The number of unsuccessful requests sent: " + result.getUnsuccessfulRequests().get());
36         System.out.println("The total run time (wall time) for all threads to complete: " + wallTime + " seconds.");
37         System.out.println("The total throughput in requests per second: " + throughput);
38     }
39
40     // MultiThreadTest
41     // J:\000000\java\src\main\resources\pom-11.0.12_38\Contents\home\bin\java ...
42     // The number of successful requests sent: 50000
43     // The number of unsuccessful requests sent: 0
44     // The total run time (wall time) for all threads to complete: 363 seconds.
45     // The total throughput in requests per second: 135
46
47     Records:
48     Mean response time (milliseconds) is: 37
49     Median response time (milliseconds) is: 34
50     Throughput(requests/second) is: 1385
51     p99 response time(milliseconds) is: 180
52     Min response time(milliseconds) is: 15
53     Max response time(milliseconds) is: 772
54
55     Process finished with exit code 0
```

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

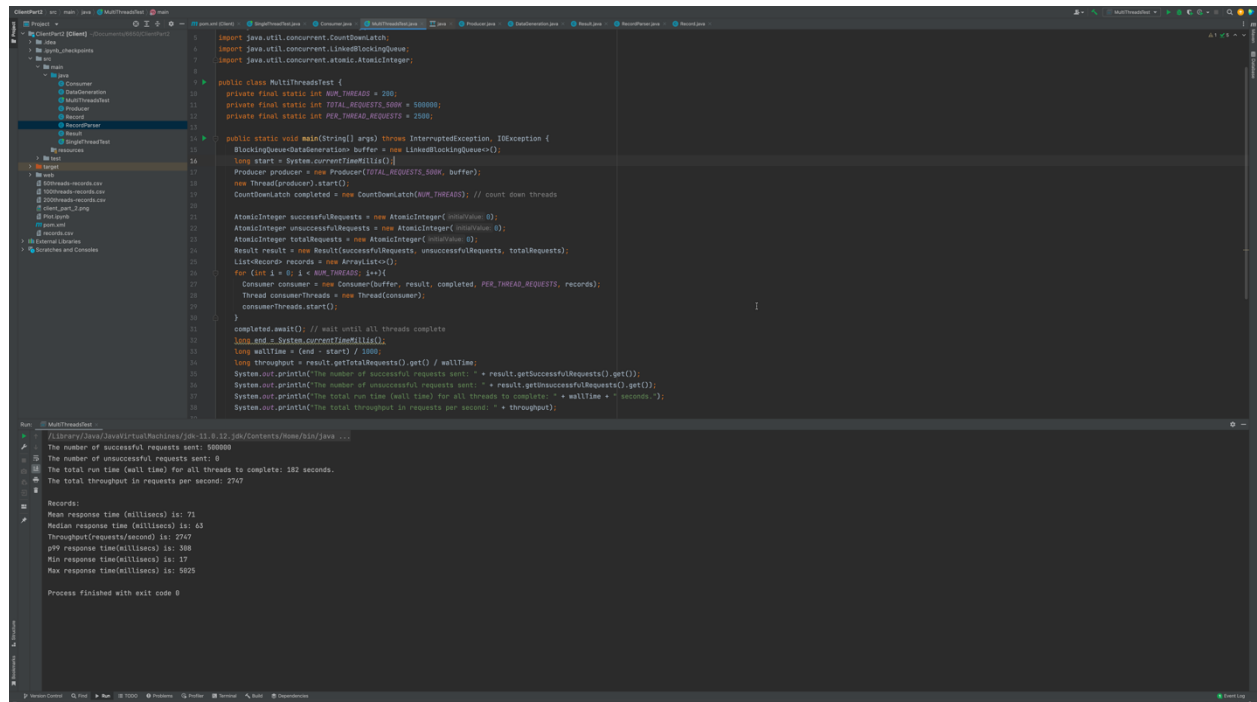
## 2. Thread = 100

```
1 import java.io.IOException;
2 import java.util.ArrayList;
3 import java.util.List;
4 import java.util.concurrent.BlockingQueue;
5 import java.util.concurrent.CountDownLatch;
6 import java.util.concurrent.LinkedBlockingQueue;
7 import java.util.concurrent.atomic.AtomicInteger;
8
9 public class MultiThreadTest {
10     private final static int NUM_THREADS = 100;
11     private final static int TOTAL_REQUESTS_500K = 50000;
12     private final static int PER_THREAD_REQUESTS = 500;
13
14     public static void main(String[] args) throws InterruptedException, IOException {
15         BlockingQueue<DataGeneration> buffer = new LinkedBlockingQueue<>();
16         long start = System.currentTimeMillis();
17         Producer producer = new Producer(TOTAL_REQUESTS_500K, buffer);
18         new Thread(producer).start();
19         CountDownLatch completed = new CountDownLatch(NUM_THREADS); // count down threads
20
21         AtomicInteger successfulRequests = new AtomicInteger(0);
22         AtomicInteger unsuccessfulRequests = new AtomicInteger(0);
23         AtomicInteger totalRequests = new AtomicInteger(0);
24         Result result = new Result(successfulRequests, unsuccessfulRequests, totalRequests);
25         List<Record> records = new ArrayList<>();
26         for (int i = 0; i < NUM_THREADS; i++) {
27             Consumer consumer = new Consumer(buffer, result, completed, PER_THREAD_REQUESTS, records);
28             Thread consumerThread = new Thread(consumer);
29             consumerThread.start();
30         }
31         completed.await(); // wait until all threads complete
32         long end = System.currentTimeMillis();
33         long wallTime = (end - start) / 1000;
34         long throughput = result.getTotalRequests().get() / wallTime;
35         System.out.println("The number of successful requests sent: " + result.getSuccessfulRequests().get());
36         System.out.println("The number of unsuccessful requests sent: " + result.getUnsuccessfulRequests().get());
37         System.out.println("The total run time (wall time) for all threads to complete: " + wallTime + " seconds.");
38         System.out.println("The total throughput in requests per second: " + throughput);
39     }
40
41     // MultiThreadTest
42     // J:\000000\java\src\main\resources\pom-11.0.12_38\Contents\home\bin\java ...
43     // The number of successful requests sent: 50000
44     // The number of unsuccessful requests sent: 0
45     // The total run time (wall time) for all threads to complete: 229 seconds.
46     // The total throughput in requests per second: 223
47
48     Records:
49     Mean response time (milliseconds) is: 43
50     Median response time (milliseconds) is: 39
51     Throughput(requests/second) is: 2283
52     p99 response time(milliseconds) is: 180
53     Min response time(milliseconds) is: 15
54     Max response time(milliseconds) is: 2726
55
56     Process finished with exit code 0
```

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



```
import java.util.concurrent.CountDownLatch;
import java.util.concurrent.locks.LockingQueue;
import java.util.concurrent.atomic.AtomicInteger;

public class MultiThreadTest {
    private final static int NUM_THREADS = 200;
    private final static int TOTAL_REQUESTS_500K = 500000;
    private final static int PER_THREAD_REQUESTS = 2500;

    public static void main(String[] args) throws InterruptedException, IOException {
        BlockingQueue<Data> queue = new LinkedBlockingQueue<>();
        long start = System.currentTimeMillis();
        Producer producer = new Producer(TOTAL_REQUESTS_500K, queue);
        new Thread(producer).start();
        CountDownLatch latch = new CountDownLatch(NUM_THREADS); // count down threads
        AtomicInteger successfulRequests = new AtomicInteger(0);
        AtomicInteger unsuccessfulRequests = new AtomicInteger(0);
        AtomicInteger totalRequests = new AtomicInteger(0);
        Result result = new Result(successfulRequests, unsuccessfulRequests, totalRequests);
        List<Record> records = new ArrayList<>();
        for (int i = 0; i < NUM_THREADS; i++) {
            Consumer consumer = new Consumer(queue, result, completed, PER_THREAD_REQUESTS, records);
            Thread consumerThread = new Thread(consumer);
            consumerThread.start();
        }
        completed.wait(); // wait until all threads complete
        long end = System.currentTimeMillis();
        long totalTime = (end - start) / 1000;
        long throughput = result.getTotalRequests().get() / totalTime;
        System.out.println("The number of successful requests sent: " + result.getSuccessfulRequests().get());
        System.out.println("The number of unsuccessful requests sent: " + result.getUnsuccessfulRequests().get());
        System.out.println("The total run time (wall time) for all threads to complete: " + totalTime + " seconds.");
        System.out.println("The total throughput in requests per second: " + throughput);
    }
}
```

Run MultiThreadTest

```
7: Library: /usr/lib/jvm/java-11.0.12-jdk-9/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: 182 seconds.
The total throughput in requests per second: 2747

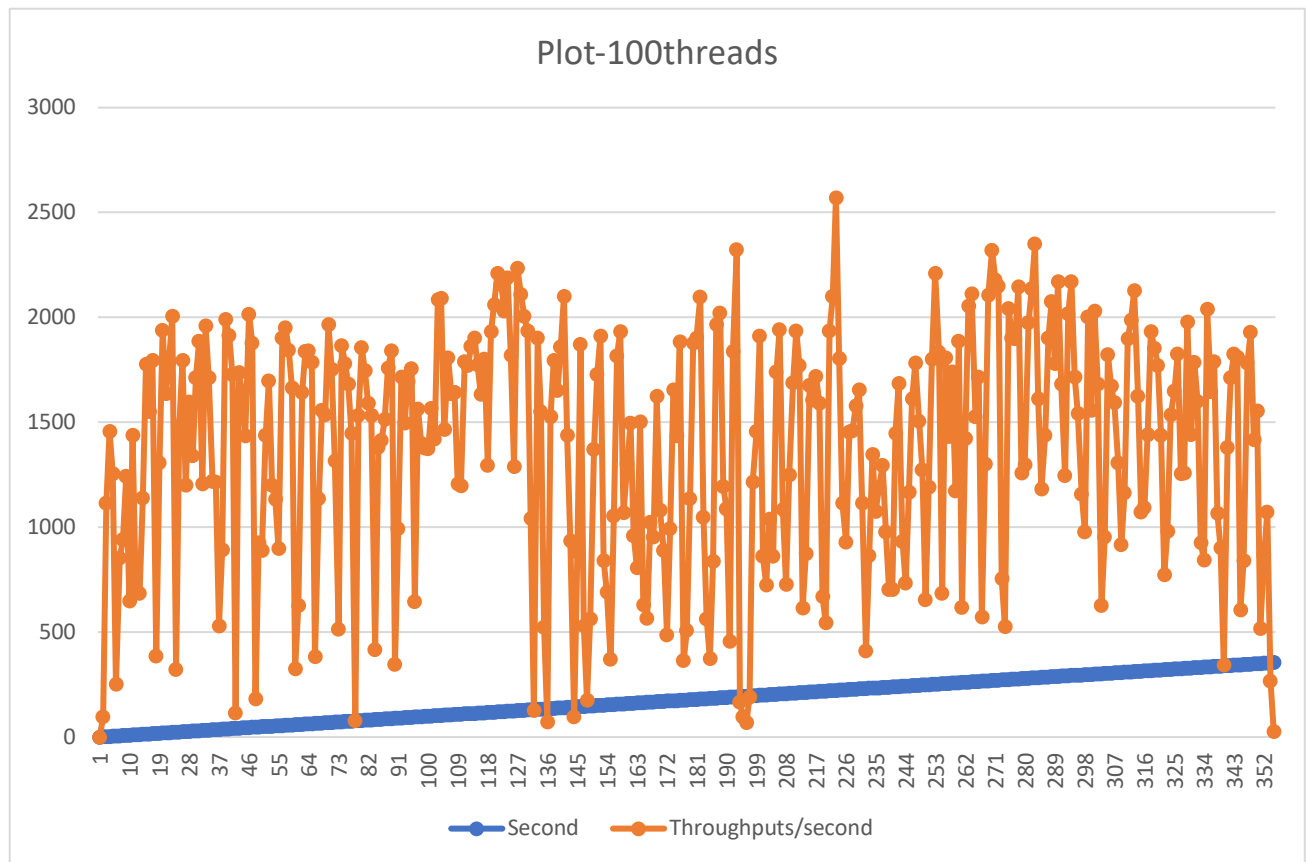
Records:
Mean response time (milliseconds) is: 71
Median response time (milliseconds) is: 63
Throughput (requests/second) is: 2747
p99 response time (milliseconds) is: 100
Min response time (milliseconds) is: 17
Max response time (milliseconds) is: 9825

Process finished with exit code 0
```

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

