

A1: Concurrency and Client-Server

1. Git repo

URL: <https://github.com/cindy-hsin/CS6650-A1>

2. Client Design:

I tested with 3 different models trying to find out which finishes sending 500K faster.

Average Model:

Related package: average

The load is distributed evenly across all threads. Each thread is in charge of generating and sending the same number of requests: $\text{NUM_TOTAL_REQUEST} / \text{NUM_THREADS}$.

Producer Consumer Model:

Related package: producerconsumer

There are two kinds of threads: Producer threads and Consumer threads.

A Producer thread keeps generating requests and pushing them into a buffer queue.

A Consumer thread keeps taking a request from the buffer queue and sending it to the server.

Each Producer thread will generate and enqueue the same amount of requests:

$\text{NUM_TOTAL_REQUEST} / \text{NUM_PRODUCER_THREAD}$.

Each Consumer thread will not necessarily take and send the same amount of requests. Once a Consumer finishes sending the previous request and is switched back to "Running" state, it will start taking the next request and send it to the server, until the total number of successful requests reaches 500K. Since the load distribution is decided by the sending speed of each thread and the JVM scheduler, "faster" or "more frequently scheduled" threads can be leveraged to send more requests.

Less Blocking Model:

Related package: `lessblocking`

This model is implemented with the intention of having less “blocking” than the first two models, and is expected to have a shorter wall time.

Each thread will increment the global counter: `numTakenReqs` by a fixed number of requests: `NUM_REQUEST_BATCH`, and then send this number of requests. Repeat this process until `numTakenReqs` reaches 500K. Notice that the global counter is incremented once every `NUM_REQUEST_BATCH` requests are sent, while in the first two models, the global counters (i.e. `numSuccessfulReqs`, `numFailedReqs`) are incremented once a request is send. So the “blocking” part is less than the first two models.

Packages and Classes/Interfaces:

`config`, `thread`, `request` packages are used by all three models.

1. `request` package:
 - a. `Request`: A POJO class encapsulating the swipe direction and the request body(`SwipeDetails`)
 - b. `RequestGenerator`: A utility class with static methods to generate a single `Request` with random fields(swipe direction, swiperid, swipeeld, comment).
2. `thread` package:
 - a. `ISendRequestThread`: An interface that declares a `sendSingleRequest` method signature.
 - b. `AbsSendrequestThread`: An abstract class that implements `ISendRequestThread` interface, and is inherited by all thread classes that are in charge of sending requests in the above 3 models.
3. `config` package:
 - a. `LoadTestConfig`: A class that encapsulates all constants related to the load test, including the number of total requests to be sent, the number of threads, the max times of retries, the server url.
4. `average` package: Implementation of the Average Model
 - a. `SendRequestAverageThread`:
 - i. A `Runnable` thread that generates and sends requests (amount: `NUM_TOTAL_REQUEST / NUM_THREADS`). For every request, if it is successfully sent within 5 retries, the global counter: `numSuccesfulReqs` is incremented; otherwise, another global counter: `numFailedReqs` is incremented.

- b. `MainAverageVersion`: The main class that initializes the global counters and the count down latch, starts the threads and waits all threads to finish, and prints out the wall time.
- 5. `producerconsumer` package: Implementation of the Producer Consumer Model
 - a. `Producer`: A Runnable thread that generates requests and puts requests into a buffer queue (amount: `NUM_TOTAL_REQUEST / NUM_PRODUCER_THREAD`).
 - b. `Consumer`: A Runnable thread that keeps taking requests from the buffer queue and sending them to the server, until the total number of successful requests (across all threads) reaches 500K.
 - c. `MainQueueVersion`: The main class that initializes the global counters, the count down latch, and the buffer queue, starts the threads and waits all threads to finish, and prints out the wall time.
- 6. `lessblocking` package: Implementation of the Less Blocking Model
 - a. `SendRequestLessBlockingVersion`: A Runnable thread that generates and sends requests in multiple rounds (amount of reqs per round: `NUM_REQUEST_BATCH`). Each thread will start a new round to repeat, until the number of requests taken/sent by all threads reaches 500K.
 - b. `MainLessBlockingVersion`: The main class that initializes the global counters and the count down latch, starts the threads and waits all threads to finish, and prints out the wall time.
- 7. `part2latency` package:
 - a. `RequestType`: A Enum that defines that request type ("POST", "GET", "PUT", "DELETE"), which is a field of `Record` class. In this assignment, the only type of request we have is "POST".
 - b. `Record`: A class that encapsulates the field of a record of a single request, including start time, request type, latency/response time, response code.
 - c. `RunningMetrics`: A class with methods that calculate the statistics of request records. It also keeps track of the intermediate calculation results in the fields.
 - d. `SendRequestAverageThread`: It's almost the same with `average.SendRequestAverageThread`, because I only tested with the Average Model in Part2. The only difference is that this class has a new method called `sendSingleRequestWithRecord` which adds a start and end timestamp before and after a request is sent, and the method returns a `Record`.
 - e. `MainPart2`: The main class that initializes the global counters, the count down latch, and the buffer queue for writing records to CSV. It starts the threads, pulls the generated records out of the buffer queue and writes to CSV, calculates some of the metrics (i.e. min, max, mean latency) on the fly. When all threads terminate, it continues to calculate other metrics (i.e. median, 99 percentile latency), group the records by start time and write to CSV for plotting, and finally prints out the wall time and statistics.

Note: How I calculated the statistics:

In Part2, I tracked the response time of each request and calculated the statistics with two methods.

Method1: Manual calculation without small memory usage. I used a buffer queue to store the list of records produced by each thread. Once the queue is populated with a list, the main thread will start to take this list of records and write the records into a CSV file, thus avoiding using a large collection in the memory to track all 500K records. Since there is no such collection in the memory that contains all records, I can only manually calculate the statistics. For the max, min, and sum of response time, I update the running variables in the RunningMetrics class after sending each request. To calculate the median and 99 percentile response time, I read the CSV back into the memory line by line(record by record). When each record is read, it is placed into a “response time range” bucket, which means the count of that bucket is incremented by 1. By doing such grouping, a histogram of “count over different response time range” can be formed, and can be utilized to find the percentile.

Method2: Automatic calculation using DescriptiveStatistics but needs to consume large memory. This method is only for testing purposes (thus the related codes are commented out). I used it to check if the manually calculated results from Method1 are correct. To use DescriptiveStatistics, I stored every record into an in-memory collection, and then applied corresponding methods of DescriptiveStatistics to calculate the statistics.

3. Result-Part1:

A. Discover the best number of threads

I tested with the Average Model to check the performance of using different numbers of threads. The result is as follows:

Model	Thread Count	Queue capacity	Wall Time(s)	Success Reqs	Failed Reqs	Throughput(req/s)
Average	100	/	178.077	500000	0	2807.774165
	160	/	159.531	500000	0	3134.187086
	200	/	158.472	500000	0	3155.131506
	500	/	172.218	500000	0	2903.296984

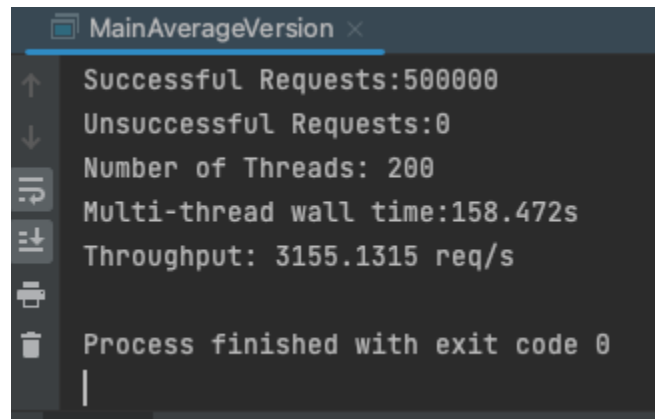
The best number of threads is around 200, so I'll use 200 threads for the following load tests.

B. Compare throughput of different client design

Model	Thread Count	Queue capacity	Wall Time(s)	Success Reqs	Failed Reqs	Throughput(req/s)
Average	200	/	158.472	500000	0	3155.131506
Producer Consumer	200 (1 Producer +	10000	159.962	500000	0	3125.742364

	199Consumer)					
Less Blocking	200	/	153.617	500000	0	3254.848096

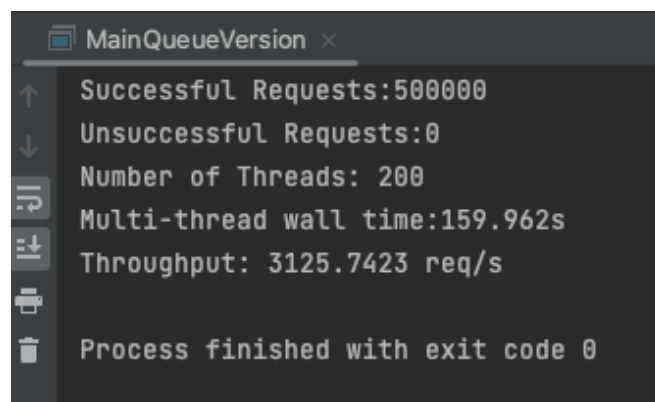
The output window screenshots for the tests are presented below:



```

MainAverageVersion x
Successful Requests:500000
Unsuccessful Requests:0
Number of Threads: 200
Multi-thread wall time:158.472s
Throughput: 3155.1315 req/s
Process finished with exit code 0

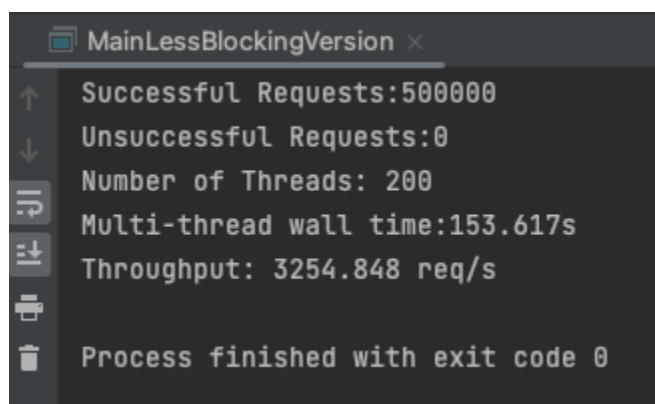
```



```

MainQueueVersion x
Successful Requests:500000
Unsuccessful Requests:0
Number of Threads: 200
Multi-thread wall time:159.962s
Throughput: 3125.7423 req/s
Process finished with exit code 0

```



```

MainLessBlockingVersion x
Successful Requests:500000
Unsuccessful Requests:0
Number of Threads: 200
Multi-thread wall time:153.617s
Throughput: 3254.848 req/s
Process finished with exit code 0

```

Although I expected the LessBlocking Model would have a higher throughput, the actual results didn't show much difference across different models.

I think the reason might be that the real bottleneck lies on the server side. Since we are sending a large load of requests using 200 threads (which is equal to the default number of threads in Tomcat server), the server is nearly overloaded. So no matter which client models we used, the requests will always “queue up” on the server side, and the server needs to take time switching between threads. Thus, the effect of a more efficient client design becomes really subtle.

Based on this observation, for **Result-Part2**, I will only present the result of one model: the **Average Model**, which has a most straightforward implementation.

C. Little’s Law throughput predictions

WallTime(s)	Success Reqs	Failed Reqs	Response Time(s/req)	Estimated Throughput for 200 threads (req/s)
293.061	10000	0	0.0293061	6824.517763

The actual throughputs (in section B, 3155.131 req/s) are only around 50% of the ideal throughput estimated by Little’s Law. It’s because the test here is conducted with only one client thread, which causes less congestion on the server side compared to the tests in section B with 200 client threads. So the estimated throughput derived from this test only suggests an ideal value for 200 threads, which ignores the overhead time on context switching and requests queueing on the server side.

The tests in Result-Part2 will validate the actual throughput (in section B) by observing the response time of each request.

4. Result-Part2:

```
MainPart2 x
Successful Requests:500000
Unsuccessful Requests:0
Number of Threads: 200
Multi-thread wall time:162.421s
Throughput: 3078.4197 req/s

Mean Response Time (ms): 64.45886
Median Response Time (ms): 60.25651
Throughput (req/s): 3078.4197
99th Percentile Response Time: 148.1432
Min Response Time (ms): 17
Max Response Time (ms): 4141

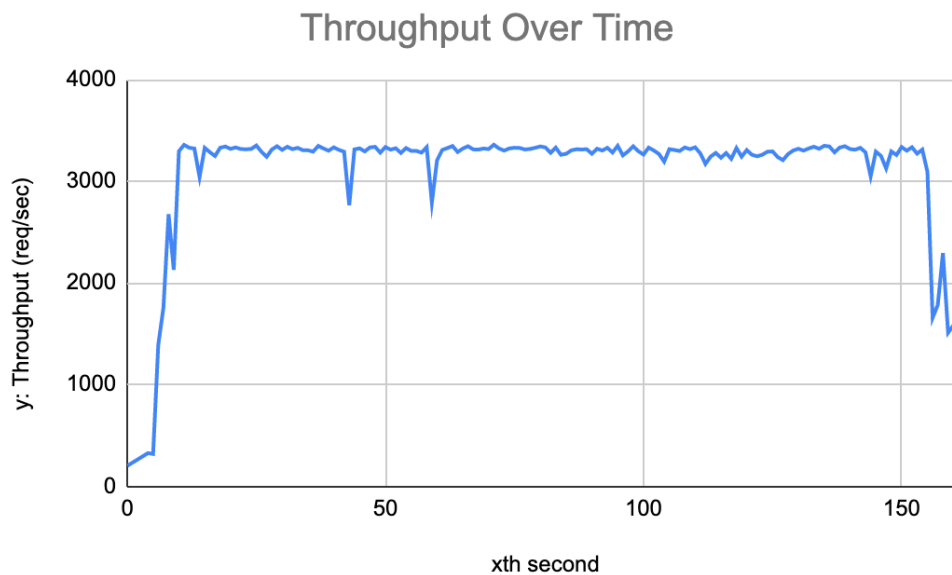
Process finished with exit code 0
```

The throughput here in Part 2(3078.41 req/s) is 2.5% lower than [Part 1\(3155.13 req/s\)](#), because of the statistics calculation overhead.

The mean response time/latency for sending a single request is 64.4ms, or 0.0644s. Compared to the “ideal” response time in [Result-Part1-sectionC](#) (single-threaded test for Little’s Law throughput estimation): 0.0293s, this reflects the “actual” response time when client sends requests with 200 threads(i.e. server is overloaded).

With this mean response time, we can calculate another “actual” throughput for 200 threads with Little’s Law: $200 / 0.0644s = 3105.59 \text{ req/s}$, which is much closer to the multi-threaded throughput in [Part 1\(3155.13 req/s\)](#) compared to the “ideal” throughput estimated in [Result-Part1-sectionC](#). This also validates that the actual throughput we got in [Part 1](#) is rational.

5. Plot: Throughput over time



This plot corresponds to the window output shown in [Result-Part2](#).

In the beginning, the throughput is low because it takes time for client threads to start and build connections with the server through the TCP “3-way handshake”. After around 10s, the throughput rises up and maintains a “plateau” as the response time becomes steady. There are a few drops around the 50th second, which might be caused by AWS cloud congestion (i.e. the client side has to do more retries to successfully send a request). After the 150th second, the throughput starts to fall as some of the client threads finish sending their share of requests and terminate, eventually reaching 0.

6. Bonus Point: Comparison with SpringBoot Server

I implemented a SpringBoot server ([code](#)) and compared its performance with Java Plain Servlet(the server for all the above sections). Both are tested with 500K requests sent from 200 client threads, using the Average Model client design. The results are as follow:

Plain Servlet

1st run

```
MainAverageVersion x
Successful Requests:497955
Unsuccessful Requests:2045
Number of Threads: 200
Multi-thread wall time:145.478s
Throughput: 3422.8887 req/s
```

2nd run

```
MainAverageVersion x
Successful Requests:500000
Unsuccessful Requests:0
Number of Threads: 200
Multi-thread wall time:127.973s
Throughput: 3907.0742 req/s
Process finished with exit code 0
```

Spring(boot)

```
MainAverageVersion x
Successful Requests:500000
Unsuccessful Requests:0
Number of Threads: 200
Multi-thread wall time:168.153s
Throughput: 2973.4824 req/s
Process finished with exit code 0
```

```
MainAverageVersion x
Successful Requests:500000
Unsuccessful Requests:0
Number of Threads: 200
Multi-thread wall time:159.913s
Throughput: 3126.7002 req/s
Process finished with exit code 0
```

I tested with 2 runs, both of which showed that Spring(boot) server has a lower throughput than Java Plain Servlet. According to [this post](#), Java Servlets are based on low-level API while Spring(boot) is a higher level abstraction built on top of Servlets. I think that's why Spring(boot) server has a lower throughput, as it might take longer time to run the server-side code.