

Repositories

Git repo for the server: <https://github.com/bo-niu/SkiServer>

Git repo for the client: <https://github.com/bo-niu/SkiClient>

Client design

This part illustrates how I design my client:

Packages

I separate the client side codes into several packages:

- **model**: This package has all the models needed in this assignment. Currently since we are only sending one kind of POST request. So, there is only one related model in this package, which is LiftUsage. As we go further in the future, more models will be added into this package.
- **part1**: This package has all the part 1 specific classes, basically all related to the three unique phases sending POST requests.
- **part2**: This package has all the part 2 specific classes, also related to the three unique phases sending POST requests.
- **util**: This package has some util classes that will be used by other packages. Currently just one RandomNumberGenerator class that can generate a random number for each thread to use.

Threads

The trickiest part of each phase is that phase 2 started when 10% of phase 1's thread finished. And phase 3 started when 10% of phase 2's thread finished. I use **CountDownLatch** to achieve such feature. So I passed the instance of a CountDownLatch object to all the threads of phase 1, who has been set to a value of 10% of the total phase 1's thread number. And when each thread of phase 1 finished, that object will be deducted by 1. So when that CountDownLatch object has been 0, phase 2 will start.

Below is a brief description of the most import classes for part1 and part2:

- **MainRunner**: The entry point of the client where main method resides in.
- **SkiHttpClient**: The class that will actually send the POST request to the server.
- **PhaseCommon**: The base class for all the three phases' threads which implements Runnable. Contain the common variables that all threads should have.
- **Phase1Thread, Phase2Thread, Phase3Thread**: The classes for the specific phase, which all extents PhaseCommon.
- **Summary**: Used to record all the metrics and calculate all the results required by the assignment.

How to run the client

To start part 1, you need to give the command line arguments below and start the main method.

`--numThreads 64 --numSkiers 20000 --numLifts 40 --numRuns 10 --ip 54.164.89.118 --port 8080`

The same applies to part 2. Please note that the IP of my server is 54.164.89.118 and port is 8080.

Little's Law prediction

From my experiment (run a lot of requests in a single thread and calculate the average response time), the average response time of the POST request is **122** ms.

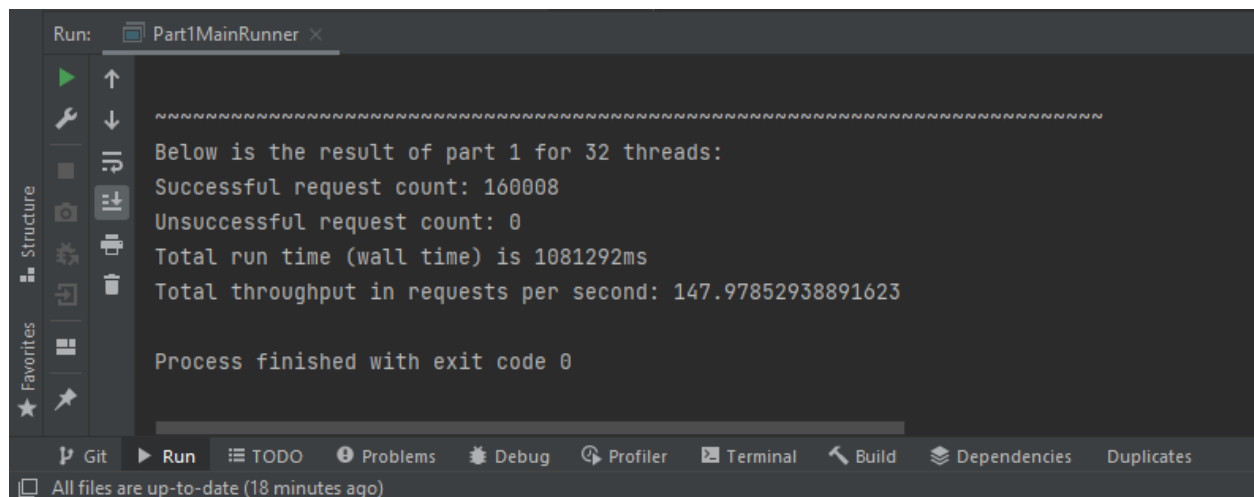
Since we have different phases and each with different number of threads running. It's impossible to calculate the exact expected throughput for each case. Here I just have a upper bound and lower bound for each case. The method I use to calculate the lower and upper bound is that:

- For each case, we have 3 phases, where phase 2 is the busiest phase with the greatest number of threads. So phase 2 have the highest throughput and phase 1 has the lowest throughput. So the actual throughput for the whole case should be within the lowest and highest bound. The table below is the max and min throughput I calculated by the Little's law and it turns out that the actual results I get from part 1 and part 2 meet the requirement (You can find the actual throughput in Part2 section).

Number of Threads	High Number of Threads (phase2)	Low Number of Threads (phase1)	Average Response Time (ms)	Expected max Throughput	Expected min Throughput
32	32	8	122	262.295082	65.57377049
64	64	16	122	524.590164	131.147541
128	128	32	122	1049.18033	262.295082
256	256	64	122	2098.36066	524.5901639

Part 1

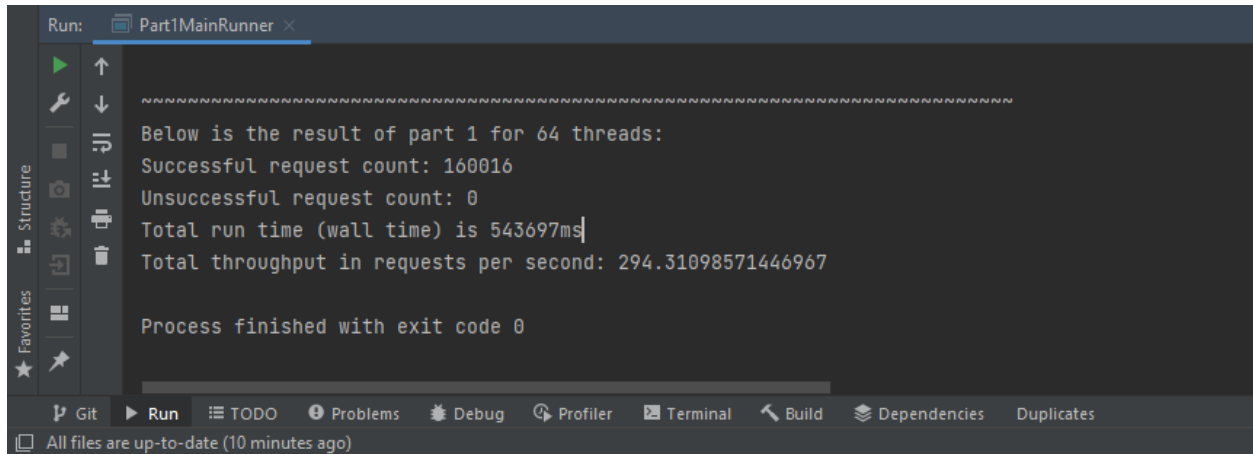
32 Threads



```
Run: Part1MainRunner x
Below is the result of part 1 for 32 threads:
Successful request count: 160008
Unsuccessful request count: 0
Total run time (wall time) is 1081292ms
Total throughput in requests per second: 147.97852938891623

Process finished with exit code 0
```

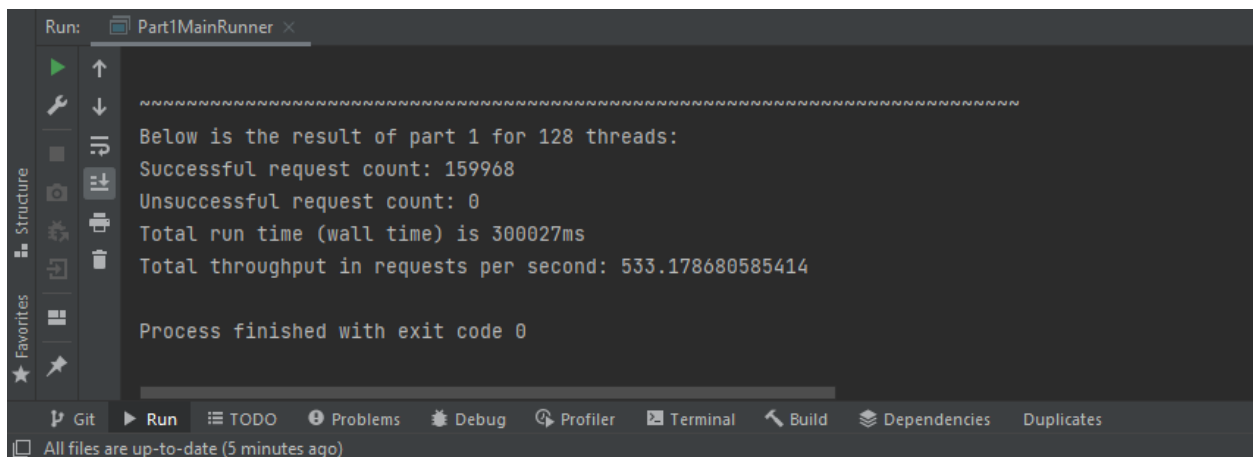
64 Threads



```
Run: Part1MainRunner x
Below is the result of part 1 for 64 threads:
Successful request count: 160016
Unsuccessful request count: 0
Total run time (wall time) is 543697ms
Total throughput in requests per second: 294.31098571446967

Process finished with exit code 0
```

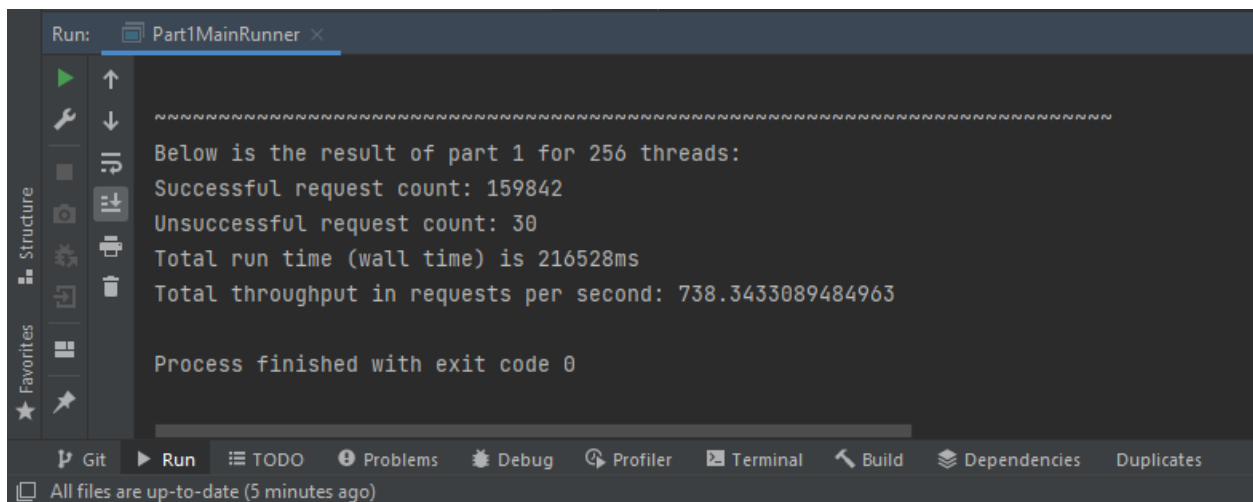
128 Threads



```
Run: Part1MainRunner x
Below is the result of part 1 for 128 threads:
Successful request count: 159968
Unsuccessful request count: 0
Total run time (wall time) is 300027ms
Total throughput in requests per second: 533.178680585414

Process finished with exit code 0
```

256 Threads



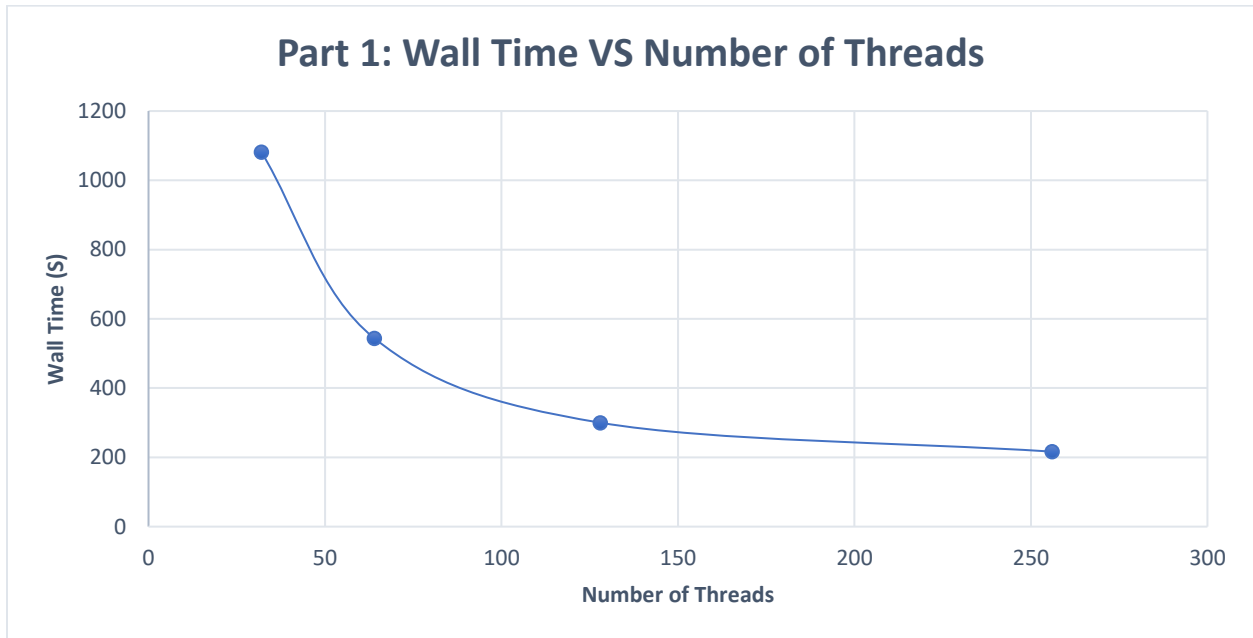
```
Run: Part1MainRunner x
Below is the result of part 1 for 256 threads:
Successful request count: 159842
Unsuccessful request count: 30
Total run time (wall time) is 216528ms
Total throughput in requests per second: 738.3433089484963

Process finished with exit code 0
```

Charts

Num of Threads	wallTime
----------------	----------

32	1081.292
64	543.697
128	300.027
256	216.528



Part 2

32 Threads

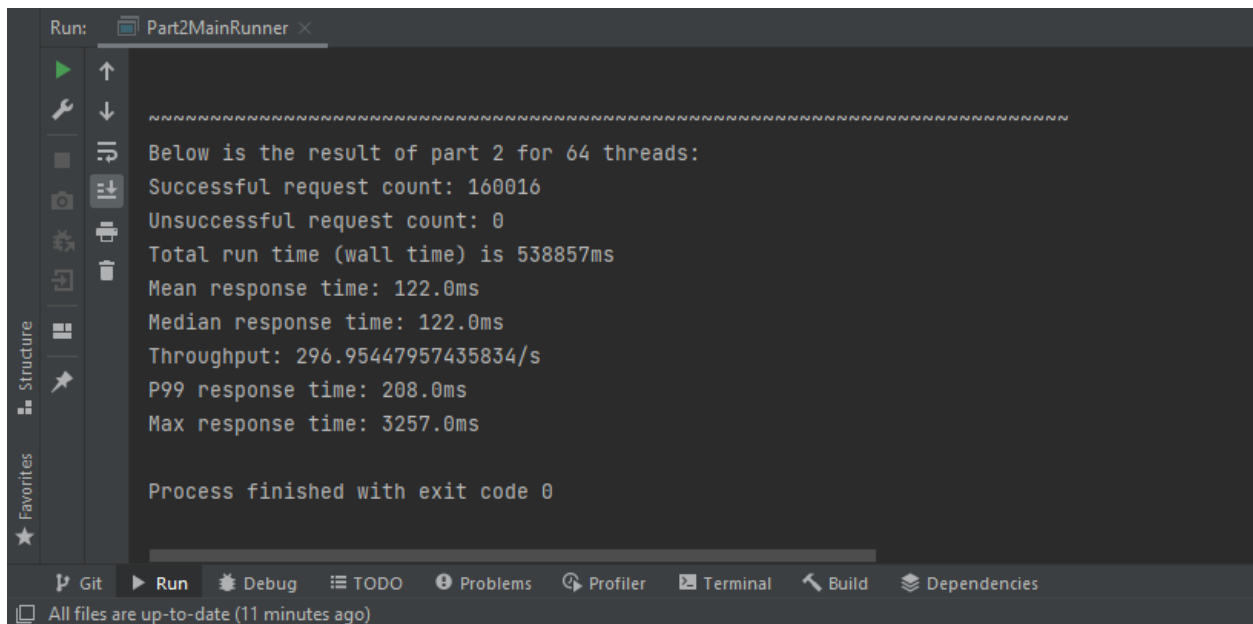
```

Run: Part2MainRunner x
Below is the result of part 2 for 32 threads:
Successful request count: 160008
Unsuccessful request count: 0
Total run time (wall time) is 1080075ms
Mean response time: 122.0ms
Median response time: 122.0ms
Throughput: 148.14526768974375/s
P99 response time: 227.0ms
Max response time: 2559.0ms

Process finished with exit code 0

```

64 Threads

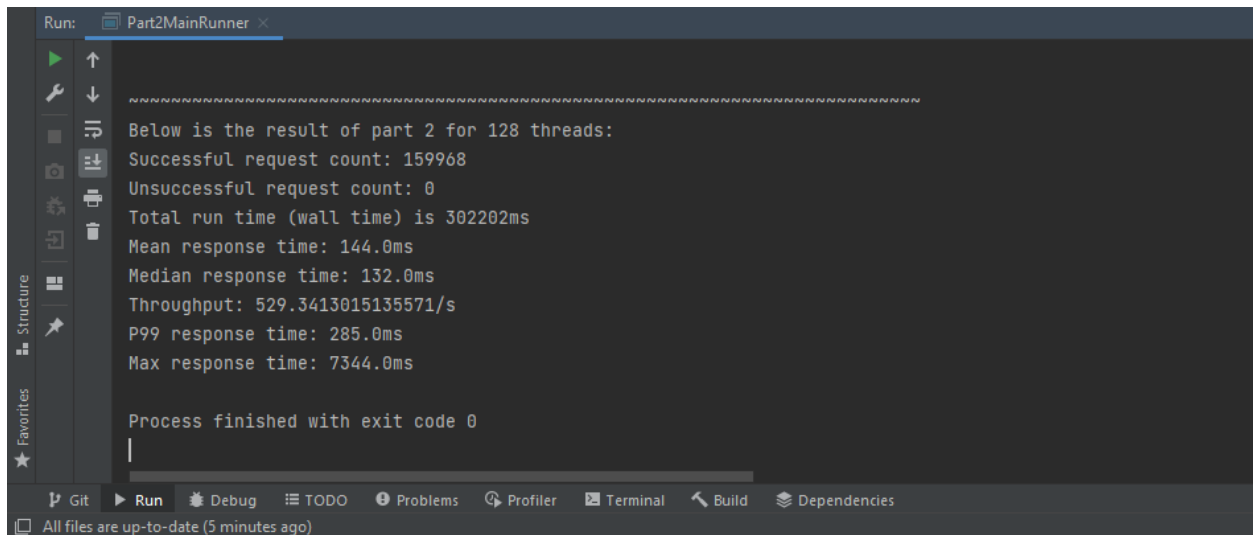


The screenshot shows an IDE terminal window titled "Run: Part2MainRunner". The terminal output displays the results of a test for 64 threads. The output is as follows:

```
~~~~~  
Below is the result of part 2 for 64 threads:  
Successful request count: 160016  
Unsuccessful request count: 0  
Total run time (wall time) is 538857ms  
Mean response time: 122.0ms  
Median response time: 122.0ms  
Throughput: 296.95447957435834/s  
P99 response time: 208.0ms  
Max response time: 3257.0ms  
  
Process finished with exit code 0
```

The IDE interface includes a sidebar with "Structure" and "Favorites" views, and a bottom toolbar with icons for Git, Run, Debug, TODO, Problems, Profiler, Terminal, Build, and Dependencies. A status bar at the bottom indicates "All files are up-to-date (11 minutes ago)".

128 Threads



The screenshot shows an IDE terminal window titled "Run: Part2MainRunner". The terminal output displays the results of a test for 128 threads. The output is as follows:

```
~~~~~  
Below is the result of part 2 for 128 threads:  
Successful request count: 159968  
Unsuccessful request count: 0  
Total run time (wall time) is 302202ms  
Mean response time: 144.0ms  
Median response time: 132.0ms  
Throughput: 529.3413015135571/s  
P99 response time: 285.0ms  
Max response time: 7344.0ms  
  
Process finished with exit code 0  
|
```

The IDE interface is consistent with the previous screenshot, showing the same sidebar and toolbar. The status bar at the bottom indicates "All files are up-to-date (5 minutes ago)".

256 Threads

```
Run: Part2MainRunner x
~~~~~
Below is the result of part 2 for 256 threads:
Successful request count: 159869
Unsuccessful request count: 3
Total run time (wall time) is 221982ms
Mean response time: 251.0ms
Median response time: 255.0ms
Throughput: 720.1890243353065/s
P99 response time: 710.0ms
Max response time: 7528.0ms

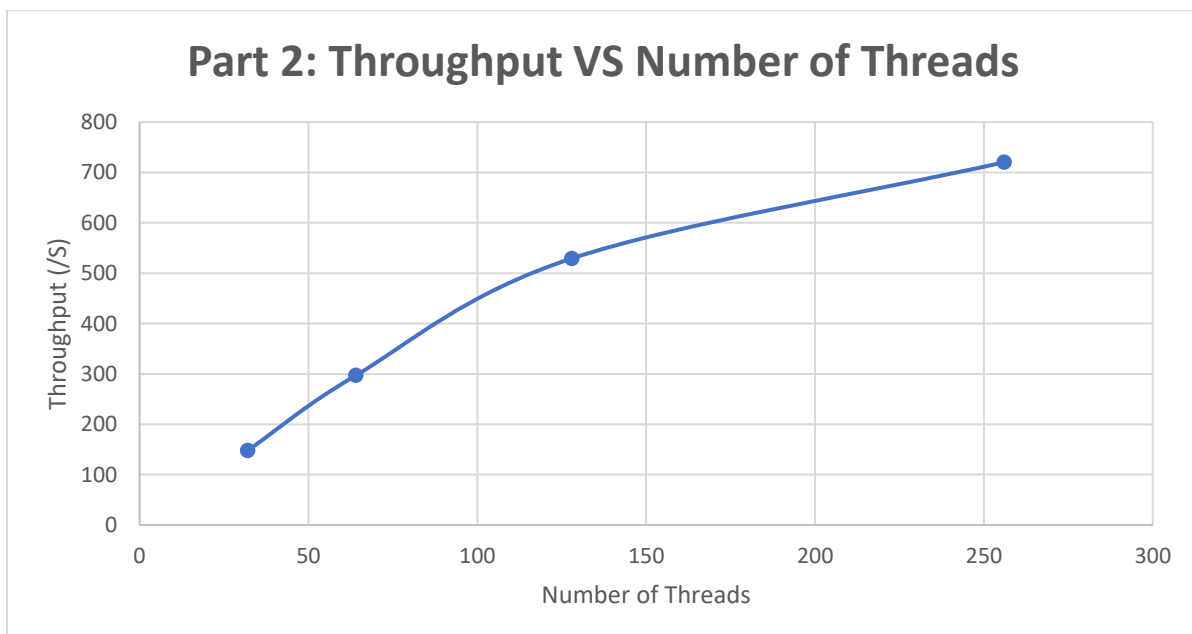
Process finished with exit code 0
~~~~~
```

Git Run Debug TODO Problems Profiler Terminal Build Dependencies

All files are up-to-date (11 minutes ago)

Charts

Number of Threads	Throughput (/s) lambda	Mean Response Time (ms) W
32	148.14	122
64	296.95	122
128	529.34	132
256	720.18	255



Part 2: Mean Response Time VS Number of Threads

