# **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

Text

Description automatically generated

## 64 Threads

A screenshot of a computer

Description automatically generated

## 128 Threads

A screenshot of a computer

Description automatically generated

## 256 Threads

Text

Description automatically generated

## Charts

|  |  |
| --- | --- |
| Num of Threads | wallTime |
| 32 | 1081.292 |
| 64 | 543.697 |
| 128 | 300.027 |
| 256 | 216.528 |

# **Part 2**

## 32 Threads

Text

Description automatically generated

## 64 Threads

A screenshot of a computer

Description automatically generated

## 128 Threads

Text

Description automatically generated

## 256 Threads

A screenshot of a computer

Description automatically generated

## 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 |