Building Scalable Distributed Systems – Assignment 1

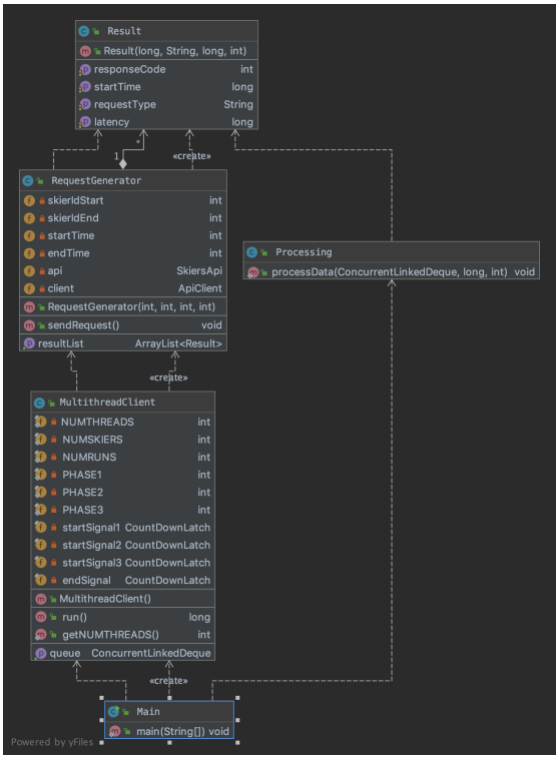
Yiwen Zhang

Client Design

1. Classes for this assignment

Four classes were created for this assignments. They are Main(main function), MultithreadClient(to concurrently run threads with three phases), RequestGenerator(to generate and send requests by swagger API), Result(to record parameters for each request). The diagram and UML are attached below:

Programming Diagram

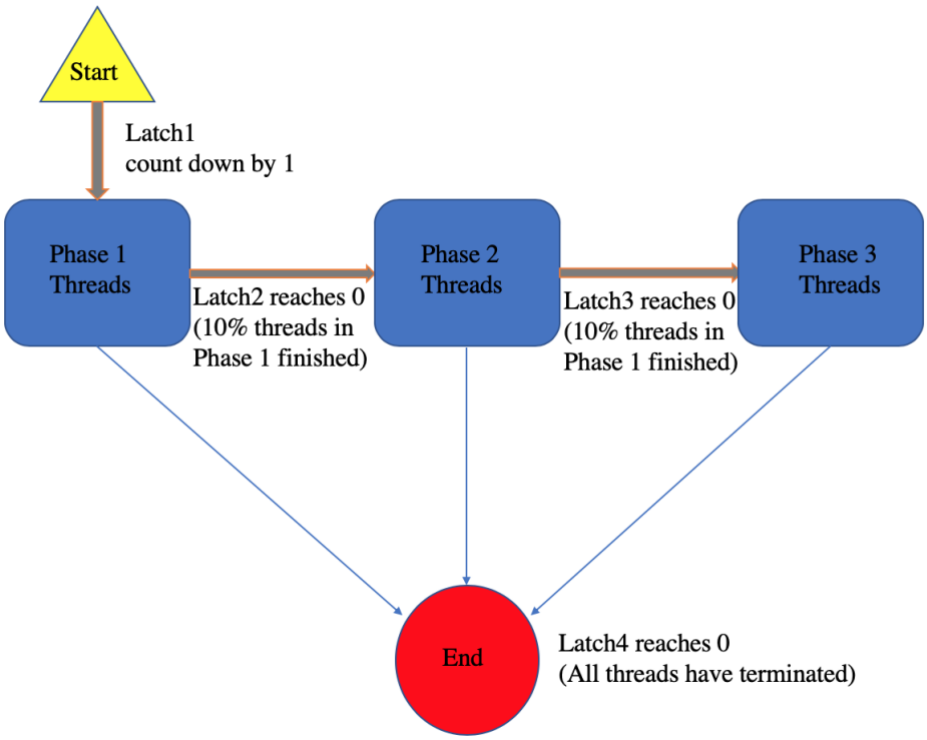


UML

2. Concurrency Control

To control multi-threads, four countdown latches are applied. The first one is called startSignal1 with initial value of 1. The second one is startSignal2 with initial value of 10% of threads in phase

2 to control threads in phase 2. The third one is startSignal3 with initial value of 10% of threads in phase 3 to control threads in phase 3. The fourth one is endSignal which is to collect all threads before terminating the main thread. The diagram is shown below:



3. Data processing

I used concurrent queue to collect results after each thread finished. After the multithread.run() terminal, the data in the concurrent queue will be processed.

Client Part1

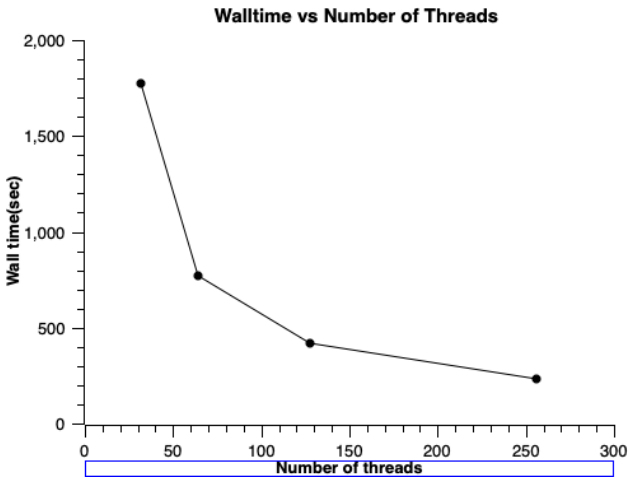
One of the possible reasons that the wall time is so large is that my server is in Virginia.

```
Total Number of Threads:32
Number of Successful Posts: 400000
Number of Unsccessful Posts: 0
Run Time:1177515

Process finished with exit code 0
```

```
Total Number of Threads:64
Number of Successful Posts: 400000
Number of Unsccessful Posts: 0
Run Time:793078

Process finished with exit code 0
```

```
Total Number of Threads:128
Number of Successful Posts: 400000
Number of Unsccessful Posts: 0
Run Time:420198

Process finished with exit code 0
```

```
Total Number of Threads:256
Number of Successful Posts: 400000
Number of Unsccessful Posts: 0
Run Time:226586

Process finished with exit code 0
```



Client Part2

1. Sensible results

As you can see in the pictures, the wall time doubles if the number of thread decreased by half. This totally makes sense as the step dominating the wall time is the network. I have checked CPU occupancy on AWS, it is only 30 %. Therefore, if there are less number of threads, more requests will be processed sequentially. This is the reason why the wall time doubles. It can also be noticed that mean response time remains the same with different number of threads, which also supports my assumption. What is more, p99 increases with the increasing number of threads. This is because multiple requests arrived at the server around the same time. They will be put in a queue. The more requests arrives around the same time the long the queue is. In addition, the difference of wall time is within 5% range.

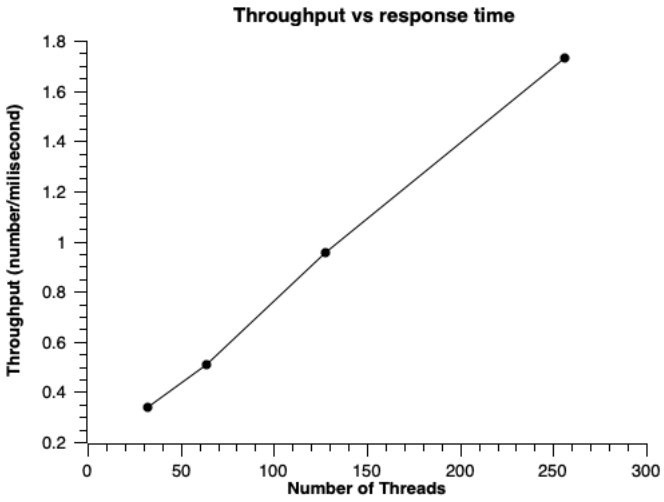
The response time and throughput all increased with number of threads. This is sensible too. As the number of threads increased, the length of the queue increased so the response time increased. If the number of threads is higher, the request can be sent more concurrently. The throughput will increase accordingly.

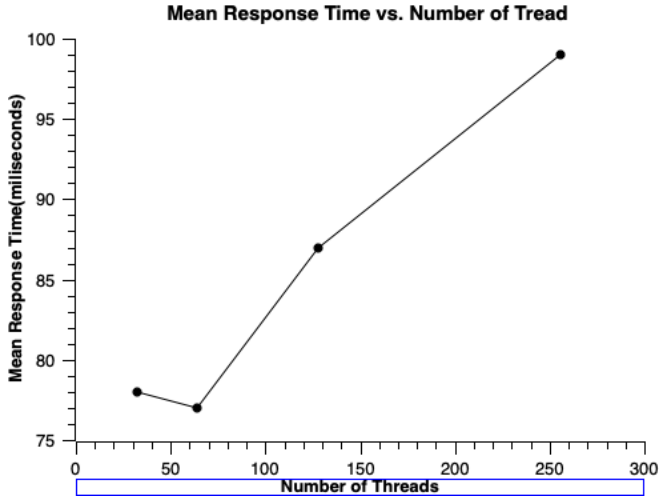
```
total time spent: 1175594
number of threads: 32
number of successful requests: 400000
number of unsuccessful requests: 0
mean response time: 78
median response time: 74
throughput: 0.34025352289991273
p99: 150
max response time: 1436

Process finished with exit code 0
```

```
total time spent: 771319
number of threads: 64
number of successful requests: 400000
number of unsuccessful requests: 0
mean response time: 77
median response time: 73
throughput: 0.518592177814886
p99: 141
max response time: 4362

Process finished with exit code 0
```

```
total time spent: 418149
number of threads: 128
number of successful requests: 400000
number of unsuccessful requests: 0
mean response time: 87
median response time: 83
throughput: 0.9565968111845299
p99: 176
max response time: 834

Process finished with exit code 0
```

```
total time spent: 230633
number of threads: 256
number of successful requests: 400000
number of unsuccessful requests: 0
mean response time: 99
median response time: 94
throughput: 1.734357182189886
p99: 200
max response time: 7535

Process finished with exit code 0
```





Bonus

1. Break things

```
total time spent: 170985
number of threads: 2000
number of successful requests: 397579
number of unsuccessful requests: 2421
mean response time: 752
median response time: 234
throughput: 2.3393864958914525
p99: 7308
max response time: 18218
```

When the number of threads increased to 2000, I started seeing unsuccessful requests as there are too many request arrived around the same time and the queue is filled up. Some requests were dumped by EC2. This is why I didn't get any response for some requests.

```
1570844988477,POST,10004,0
```

2. The way I did it is to first capture the earliest start time and the latest start time. Then I initialized the array list with size of latest start time - earliest start time + 1. Then, for each request, with a specific start time, I threw it into the according bucket and solved the average of response time. I generated the csv file but didn't have time to plot it. I think my plot will be fantastic.