Assignment 1: Concurrency and Client-server

Zhiyuan Luo

1. Server implementation

My client is in local.

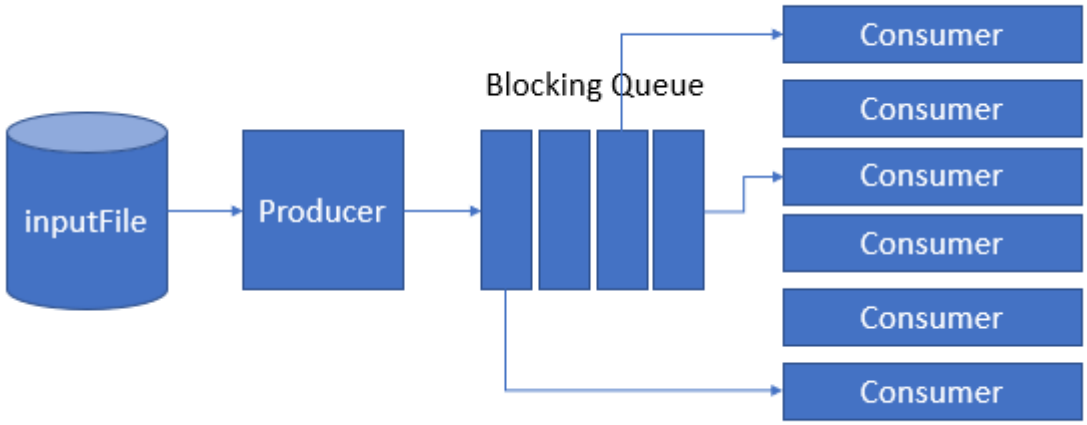
My server is in EC2: http://35.169.82.198:8080/servlet_war/servlet/

Repo: https://github.com/julianluo/CS6650-assignments/tree/master/assignment1

inputFile = https://raw.githubusercontent.com/gortonator/bsds-6650/master/assignments-2021/bsds-summer-2021-testdata-assignment2.txt

2. Client design description

I am using producer consumer solution to build this multithreading client. The producer consumer solution of multithreading client is shown as below:




I have the following classes in my client: Main, Producer, Consumer, ParamValidater, PostStatistics, ResultAnalyzer.

The Main class is the entry point into the client. The Main class creates a ParamValidater instance and validate the parameters. And then it creates a Producer instance to read the file line by line and add each line as a String to a BlockingQueue<String>. It creates a CountDownLatch instance to make sure after all threads are started, the client wait until all the threads successfully complete. It creates a PostStatistics instance to accumulate and store the number of successful posts, the number of failed posts, and to record statistics of each request and response. And the client starts to send requests and receive responses using producer consumer pattern of multithreading. The statistics are calculated and displayed in the output window. A record containing {start time, request type, latency, response code} is written out in CSV format file.

The ParamValidater class is used to validate the parameters. The validate method is used to check if there is argument, if the number of thread exceed maximum of threads. The getNumThreads method return the number of threads.

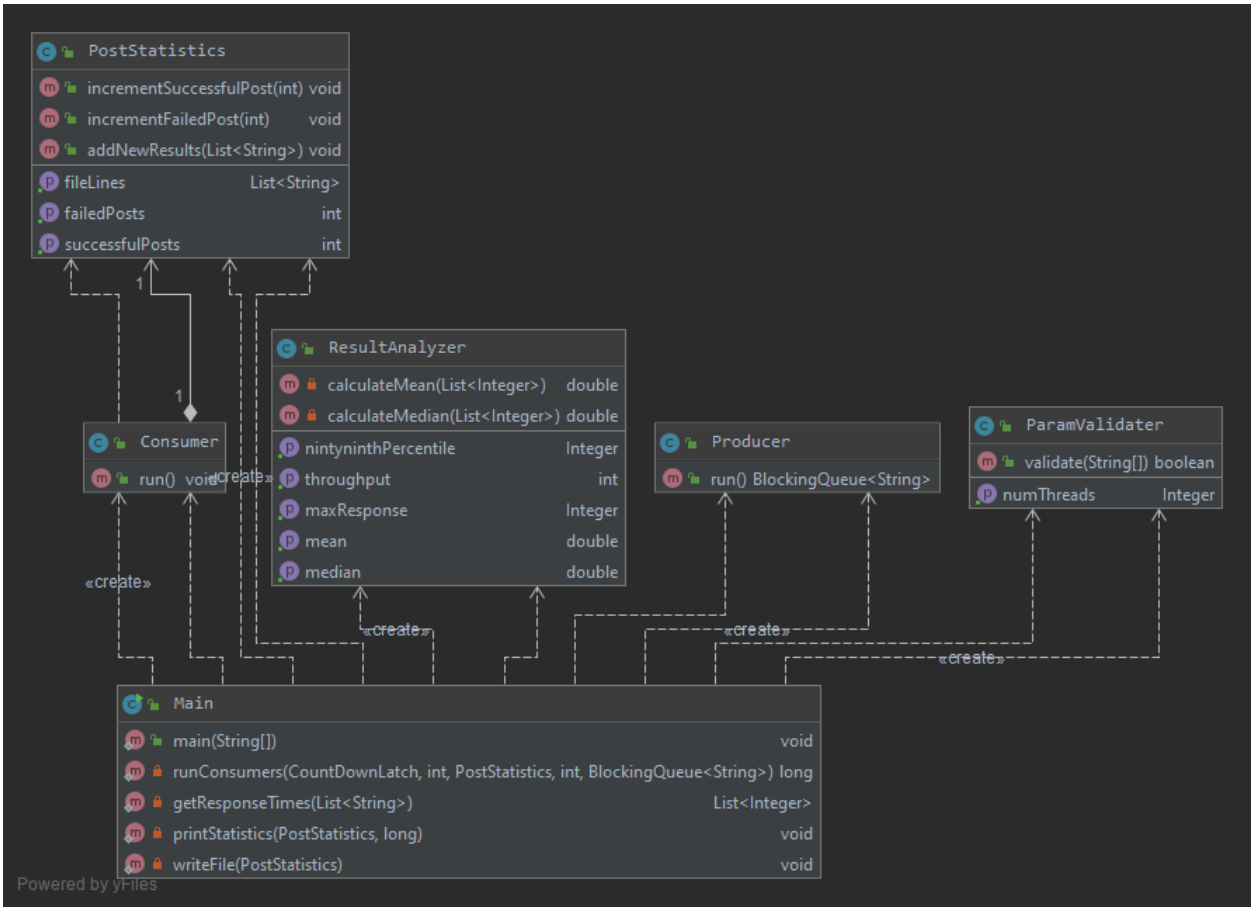
The Producer class is the producer in the producer consumer pattern. The Producer class has a run() method which return a BlockingQueue<String>. When producer.run() is invoked, the producer use BufferedReader to read the text file line by line and add each line into the BlockingQueue<String>. And it add a few "/End of File/" strings at the end. The count of "/End of File/" strings is the number of threads.

The consumer class is the consumer in the producer consumer pattern. The Consumer class has a run() method. When consumer.run() is invoked, the consumer uses the swagger TextbodyApi and sets the base path of ApiClient to be my EC2 servlet or my local servlet. The apiInstance sends request to the server and return an ApiResponse<ResultVal>. Every time apiInstance sends a request to the server, it creates a TextLine instance, pops a string from the BlockingQueue<String>, and set the string to be the message of the TextLine instance. At the same time, the number of successful posts and the number of failed posts are accumulated and recorded by a PostStatistics instance. And start time, request type, latency, response code of each request and response is recorded. If the the BlockingQueue<String> peeks a /End of File/ String, the thread is terminated. A CountDownLatch is counted down in this process. Consumer will keep sending requests and receiving responses until all the lines of the input file are exhausted by for loop.

The PostStatistics class is used to accumulate and store the number of successful posts, the number of failed posts, and to record the start time, request type, latency, response code of each request and response. An instance of PostStatistics class is initialized in the Main class and it is passed to Consumer. Its three methods: incrementSuccessfulPost, incrementFailedPost, addNewResults are invoked in Consumer class.

The ResultAnalyzer class is used to calculate mean, median, throughput, max response, and 99th percentile. It includes methods: calculateMean, calculateMedian, getMean, getMedian, getThroughput, getMaxResponse, getNintyninthPercentile. These methods are wrapped into printStatistics method in Main class. When printStatistics is invoked in Main, all these statistics are calculated and displayed in the output window.

The UML graph is shown below:




The packages I used include following:

java.io.BufferedWriter;

java.io.FileWriter;

java.io.IOException;

java.util.ArrayList;

java.util.List;

java.sql.Timestamp;

java.util.concurrent.BlockingQueue;

java.util.concurrent.CountDownLatch;

java.util.concurrent.LinkedBlockingQueue;

java.util.Collections;

io.swagger.client.ApiClient;

io.swagger.client.ApiException;

io.swagger.client.ApiResponse;

io.swagger.client.model.*;

io.swagger.client.api.TextbodyApi;

3. Client Part 1

32 threads run:

```
Number of threads: 32
Number of successful posts: 99300
Number of failed posts: 0
Wall time: 271secs, 271275ms
Throughput: 366 requests/sec
```

64 threads run:

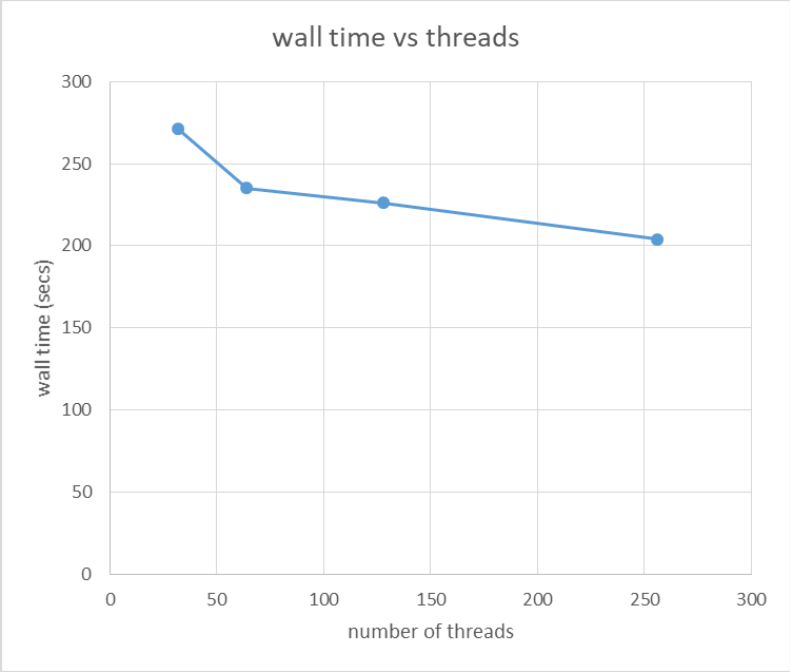
```
Number of threads: 64
Number of successful posts: 99300
Number of failed posts: 0
Wall time: 235secs, 235147ms
Throughput: 422 requests/sec
```

128 threads run:

```
Number of threads: 128
Number of successful posts: 99300
Number of failed posts: 0
Wall time: 226secs, 226201ms
Throughput: 439 requests/sec
```

256 threads run:

```
Number of threads: 256
Number of successful posts: 99300
Number of failed posts: 0
Wall time: 204secs, 204756ms
Throughput: 486 requests/sec
```




4. Client Part 2

32 threads run:

```
Number of threads: 32
Number of successful posts: 99300
Number of failed posts: 0
Wall time: 271secs, 271275ms
Throughput: 366 requests/sec
Average response time: 87.33179254783484 ms
Median response time: 78.0 ms
99th percentile: 216 ms
Max response time: 1557 ms
```

64 threads run:

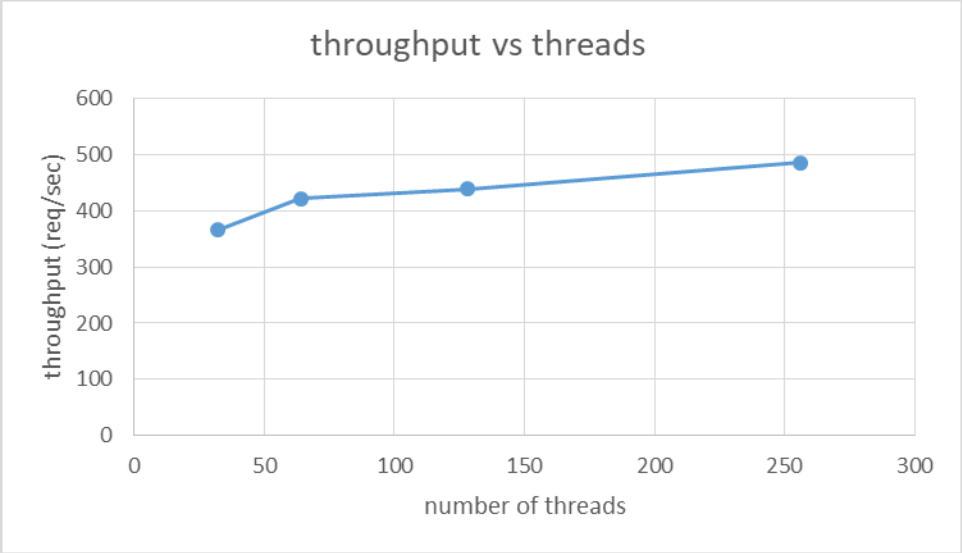
```
Number of threads: 64
Number of successful posts: 99300
Number of failed posts: 0
Wall time: 235secs, 235147ms
Throughput: 422 requests/sec
Average response time: 151.41090634441088 ms
Median response time: 140.0 ms
99th percentile: 375 ms
Max response time: 3343 ms
```

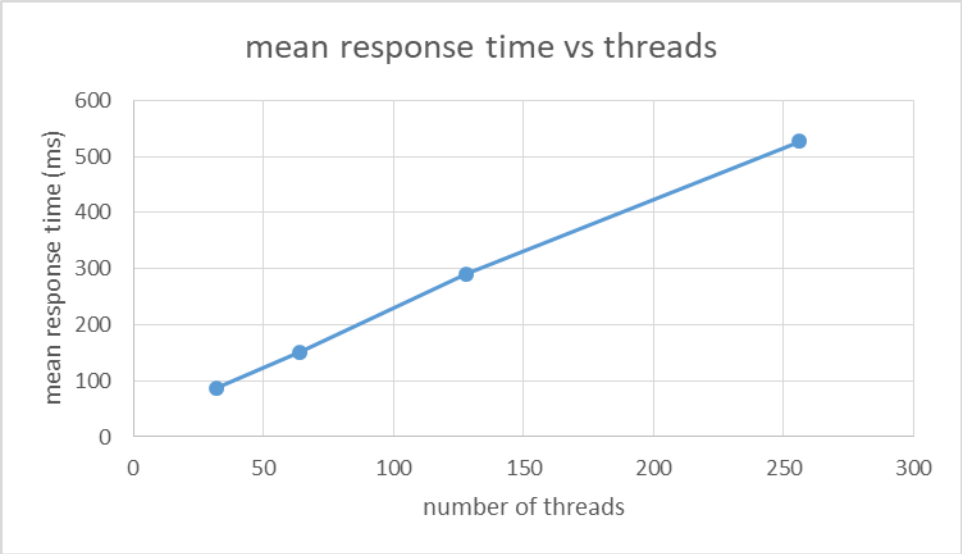
128 threads run:

```
Number of threads: 128
Number of successful posts: 99300
Number of failed posts: 0
Wall time: 226secs, 226201ms
Throughput: 439 requests/sec
Average response time: 291.29177240684794 ms
Median response time: 265.0 ms
99th percentile: 547 ms
Max response time: 2176 ms
```

256 threads run:

```
Number of threads: 256
Number of successful posts: 99300
Number of failed posts: 0
Wall time: 204secs, 204756ms
Throughput: 486 requests/sec
Average response time: 526.9134541792548 ms
Median response time: 516.0 ms
99th percentile: 797 ms
Max response time: 3125 ms
```






Wall time are all within 5% of wall time for Client Step 1:

| 32 threads: | |
|---|---|
| total number of requests | 99300 |
| wall time | 04:31.0 |
| throughput | 366.4206642 |

| 64 threads: | |
|---|---|
| total number of requests | 99300 |
| wall time | 03:48.3 |
| throughput | 434.9540079 |

| 128 threads: | |
|---|---|
| total number of requests | 99300 |
| wall time | 03:45.9 |
| throughput | 439.5750332 |

| 256 threads: | |
|---|---|
| total number of requests | 99300 |
| wall time | 03:24.1 |
| throughput | 486.5262126 |

5. Bonus Points

Break Things：

When number of threads is 1000: requests will take so long they will timeout and fail. Below is the error

Exception when calling TextbodyApi#analyzeNewLine

io.swagger.client.ApiException: java.net.SocketTimeoutException: Read timed out

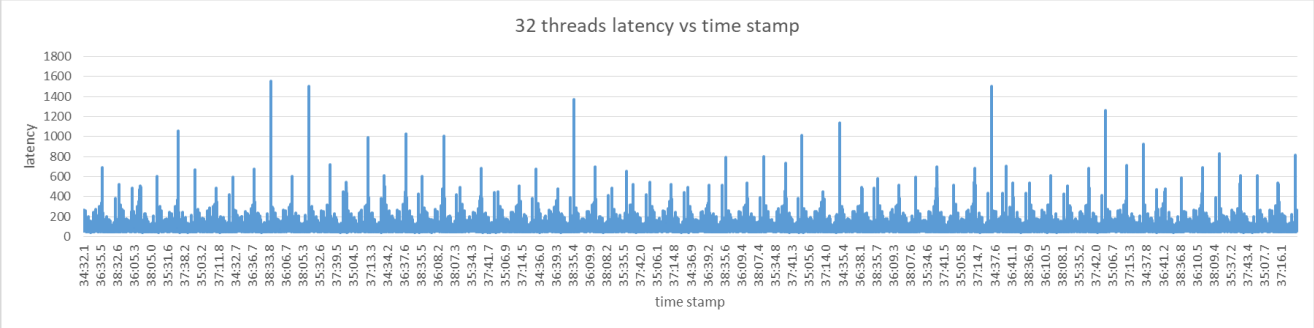
    at io.swagger.client.ApiClient.execute(ApiClient.java:844)

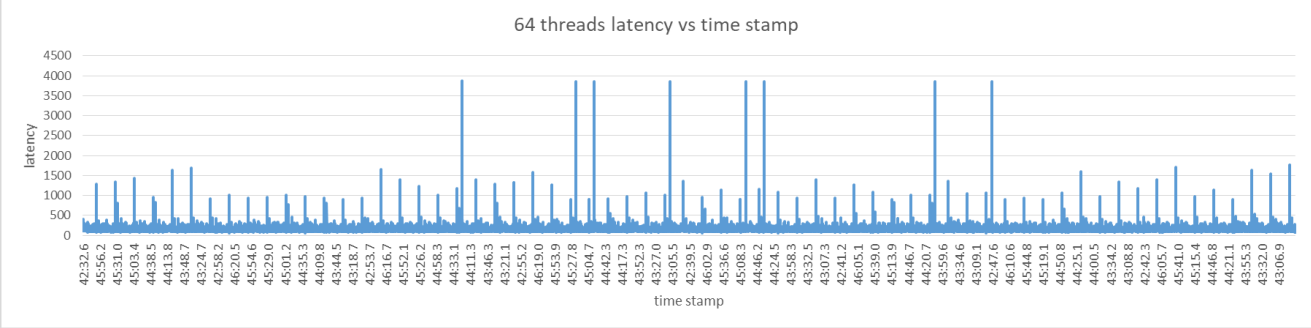
    at io.swagger.client.api.TextbodyApi.analyzeNewLineWithHttpInfo(TextbodyApi.java:153)

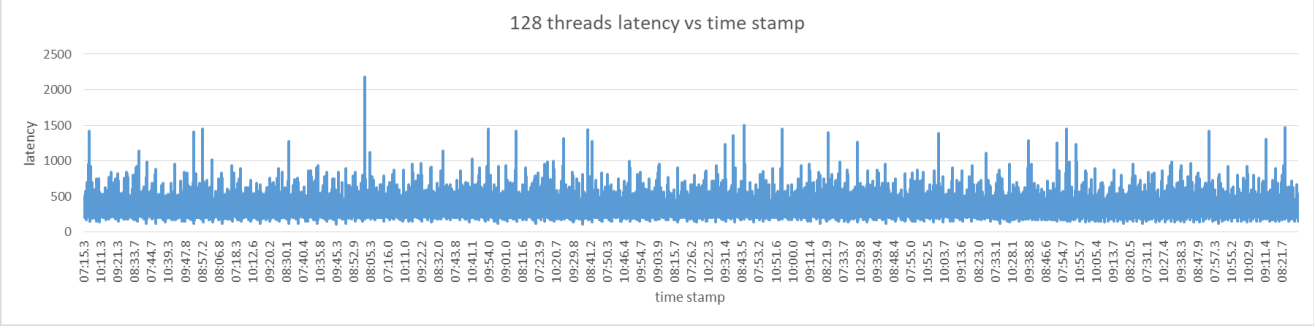
    at Consumer.run(Consumer.java:56)

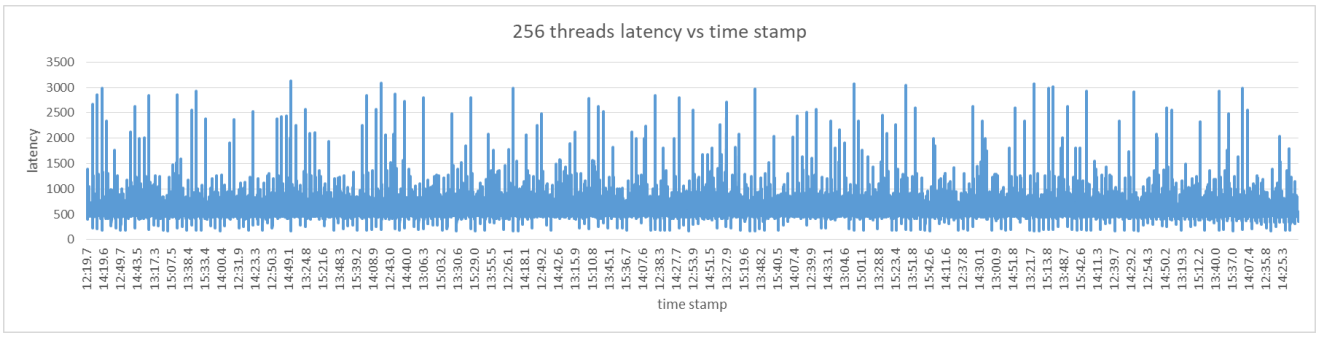
Caused by: java.net.SocketTimeoutException: Read timed out

Charting:




256 threads latency vs time stamp
latency
3500
3000
2500
2000
1500
1000
500
0
12:19.7
14:19.6
12:49.7
14:43.5
13:17.3
15:07.5
13:38.4
15:33.4
14:00.4
12:31.9
14:23.3
12:50.3
14:49.1
13:24.8
15:21.6
13:48.3
15:39.2
14:08.9
12:43.0
14:40.0
13:06.3
15:03.2
13:30.6
15:29.0
13:55.5
12:26.1
14:18.1
12:49.2
14:42.6
13:15.9
15:10.8
13:45.1
15:36.7
14:07.6
12:38.3
14:27.7
12:53.9
14:51.5
13:27.9
15:19.6
13:48.2
15:40.5
14:07.4
12:39.9
14:33.1
13:04.6
15:01.1
13:28.8
15:23.4
13:51.8
15:42.6
14:11.6
12:37.8
14:30.1
13:00.9
14:51.8
13:21.7
15:13.8
13:48.7
15:42.6
14:11.3
12:39.7
14:29.2
12:54.3
14:50.2
13:19.3
15:12.2
13:40.0
15:37.0
14:07.4
12:35.8
14:25.3
time stamp