Assignment 1

5/31/2021 - Report

Github repo URL

https://github.com/coffinated/bsds-a1

Client Design

To begin with, I used the Swagger generated Java client. I tweaked the base URL in the ApiClient class to point to my server, but otherwise I didn't do much tweaking to what was provided. The generation of a request is handled in these provided classes.

The files I added for Part 1 include TextbodyApiClient, TextProducer, and TextConsumer. The main() class is in TextbodyApiClient, which accepts a number of threads and file path as arguments. From this main thread, a BlockingQueue is initialized, along with variables for the counts of successful and failed requests, and finally, the producer thread and maxThreads consumer threads (started in a for loop). I initialized a CountDownLatch with the number of consumer threads and passed that in to each consumer.

The producer thread, in the TextProducer class, takes the file path, BlockingQueue, and maxThreads as parameters to its constructor. When it starts up, it reads the input file line by line into the BlockingQueue, skipping over any empty lines. When it reaches the end of the file, it puts the string "EOF" on the BlockingQueue in a loop that runs maxThreads times. This ensures that there is one 'signal' for each consumer thread to pick up from the queue. Finally, the producer closes the file reader and its execution is completed.

The consumer threads, in the TextConsumer class, take a BlockingQueue, a CountDownLatch, and a TextbodyApiClient instance as parameters to the constructor. Each thread creates an instance of TextbodyApi class with which to send a request. In a while loop, the consumer takes the next message off the BlockingQueue, checks if it reads "EOF" and breaks out of the loop if so. If not, the consumer initializes a TextLine instance and sets its message to the line of text from the queue. Next, it sends a request to the server using the TextbodyApi analyzeNewLine() method and checks the response message for good measure. (Since the instructions indicated our servers don't really need to do any text processing for this assignment, I set mine up to return a simple word count of each request's message. The consumer reads this returned message mostly for testing purposes at this stage.) If the request received a 200 response code, it uses the TextbodyApiClient's successCount() method to tally up a success; if not, execution goes to the catch block, where the failCount() method is called.

Once the consumer thread picks up "EOF" from the BlockingQueue, it breaks out of this while loop and calls countDown() on the CountDownLatch on which the main thread is waiting. After all of those consumer threads have finished running and the CountDownLatch reached 0, the main thread back in TextbodyApiClient calculates the stats and prints to the terminal. The wall time is collected by taking the time with System.currentTimeMillis() before the threads are initialized and comparing it to the time after all threads have completed their tasks.

For Part 2, I added two more classes to help with logging: RequestStat and RequestStatCollector. A RequestStat is an object with all the stats needed for output to the CSV: start time, request method, latency, and response code. One of these will be produced

for each request sent. There's also a toString() method here to make the file writing simpler, and a type of RequestStat that lets the writing thread know it has reached the end.

The RequestStatCollector is a Runnable class that is initiated within the main thread in the Part 2 version of TextbodyApiClient. It takes a BlockingQueue as its only parameter, and this is separate from the queue used between the producer and consumer threads. This BlockingQueue is just to log the stats while the requests are being sent, so it also gets sent to each Consumer thread as it is initialized. In a loop, the stat collector thread takes the next RequestStat off the BlockingQueue, and writes it to output.csv (unless it's a special EndOfStats RequestStat, in which case it breaks out of the loop). While it collects each stat for writing to the output, it also sums the wall time and keeps each individual response latency in an ArrayList. When it reaches an EndOfStats object, it breaks out of the loop, sorts the ArrayList of latencies, and closes the file writer. There are other methods in this class that are called from the main thread to provide the stats on the test run - getMedian(), getPercentile(), getMean(), and getMax(). These make use of the sorted latency ArrayList to calculate and return the stats required.

The main thread initializes and starts the RequestStatCollector thread right before starting the producer and consumer threads. TextbodyApiClient is also responsible for creating and sending the EndOfStats object to the stats queue (only after awaiting the completion of the CountDownLatch) so that the collector knows it can finish up.

This sums up my client design, parts 1 and 2, for assignment 1. Next, I'll provide screenshots of my output along with some charts.

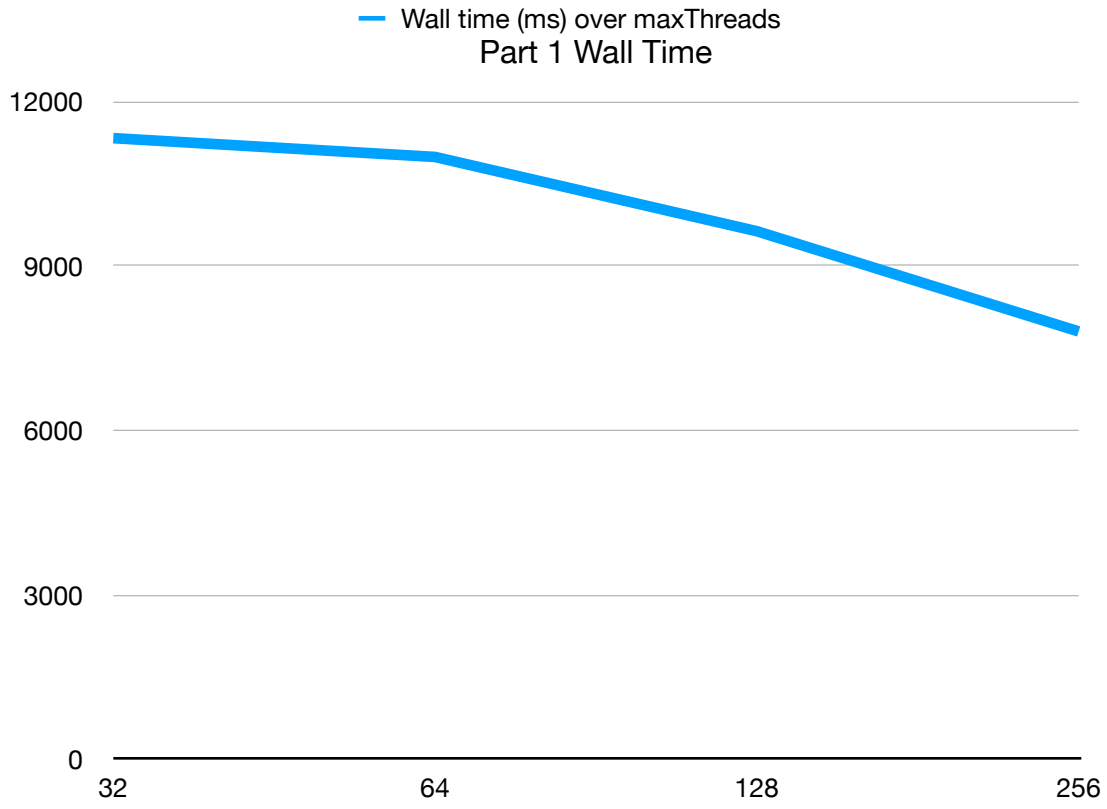
Part 1 screenshots & charts

```
Starting 32 threads...
++++++++++++++++Stats++++++++++++++++
Wall time: 11328ms
Successful requests: 9930
Failed requests: 0
Throughput: 876.5887830508476 requests per second
++++++++++++++++++++++++++++++++++++

Process finished with exit code 0
```

```
Starting 64 threads...
++++++++++++++++Stats++++++++++++++++
Wall time: 10985ms
Successful requests: 9930
Failed requests: 0
Throughput: 903.9599453808638 requests per second
++++++++++++++++++++++++++++++++++++

Process finished with exit code 0
```

```
Starting 128 threads...
+++++++++++++++++Stats+++++++++++++++++
Wall time: 9631ms
Successful requests: 9930
Failed requests: 0
Throughput: 1031.0455819748727 requests per second
++++++++++++++++++++++++++++++++++++
```

```
Starting 256 threads...
+++++++++++++++++Stats+++++++++++++++++
Wall time: 7804ms
Successful requests: 9930
Failed requests: 0
Throughput: 1272.4243977447463 requests per second
++++++++++++++++++++++++++++++++++++
```



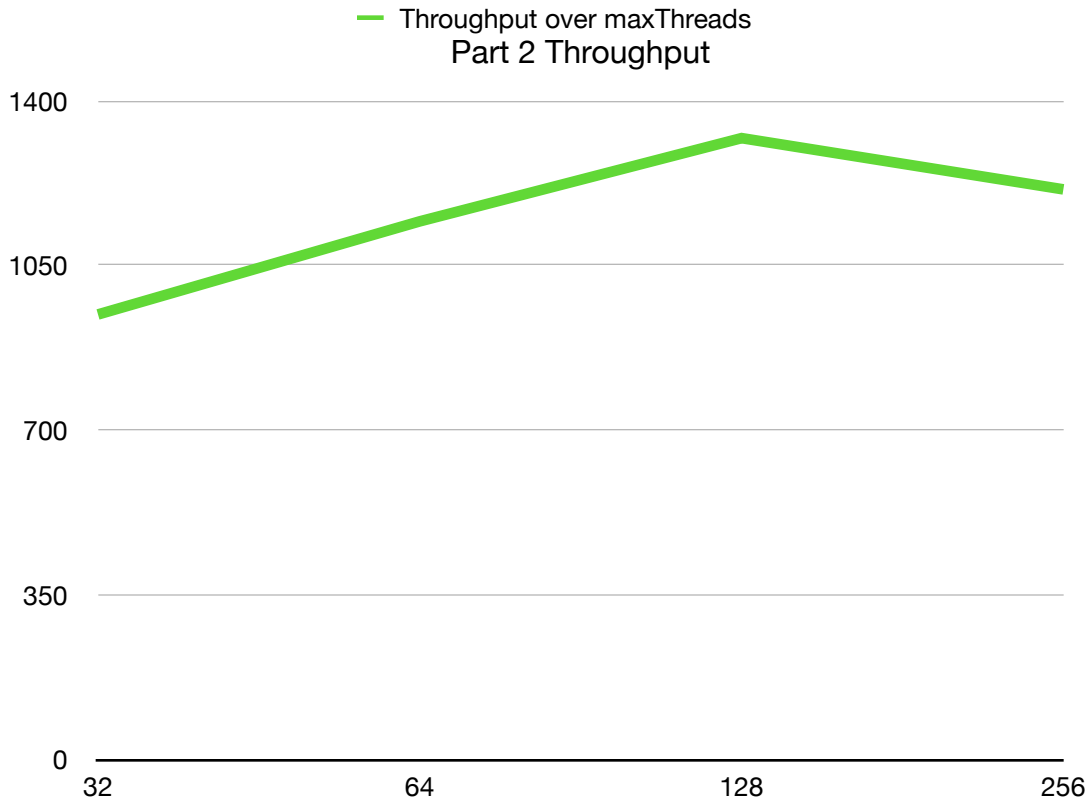
Part 2 screenshots & charts

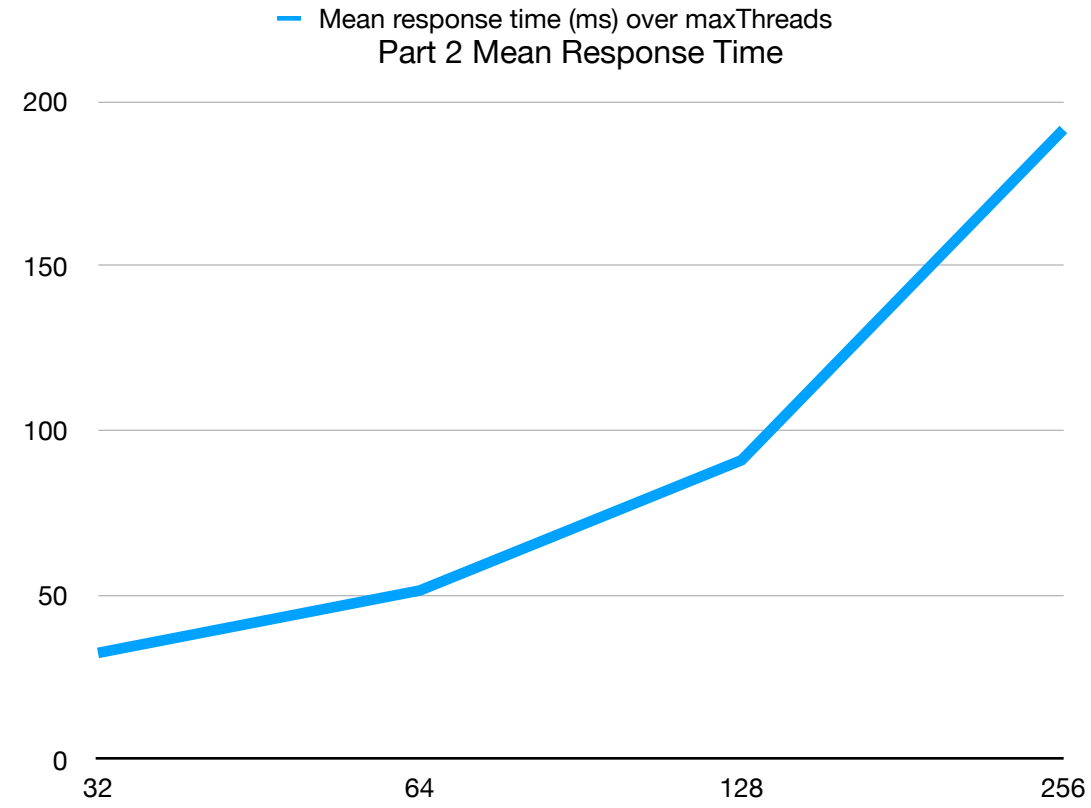
```
Starting 32 threads...
++++++++++++++++Stats++++++++++++++++
Successful requests: 9938
Failed requests: 0
Mean response time: 32.431621349446125 ms
Median response time: 30 ms
Wall time: 10494ms
Throughput: 946.2550028587765 requests per second
p99 response time: 54 ms
Max response time: 638 ms
++++++++++++++++++++++++++++++++++++

Process finished with exit code 0
```

```
Starting 64 threads...
++++++++++++++++Stats++++++++++++++++
Successful requests: 9930
Failed requests: 0
Mean response time: 51.399194368523666 ms
Median response time: 42 ms
Wall time: 8678ms
Throughput: 1144.2728739348862 requests per second
p99 response time: 192 ms
Max response time: 812 ms
++++++++++++++++++++++++++++++++++++

Process finished with exit code 0
```

```
Starting 128 threads...
++++++++++++++++Stats++++++++++++++++
Successful requests: 9930
Failed requests: 0
Mean response time: 90.98318227593153 ms
Median response time: 88 ms
Wall time: 7517ms
Throughput: 1321.0057203671677 requests per second
p99 response time: 637 ms
Max response time: 690 ms
++++++++++++++++++++++++++++++++++++

Process finished with exit code 0
```

```
Starting 256 threads...
++++++++++++++++Stats++++++++++++++++
Successful requests: 9930
Failed requests: 0
Mean response time: 191.447029294431 ms
Median response time: 175 ms
Wall time: 8193ms
Throughput: 1212.0102526547053 requests per second
p99 response time: 1147 ms
Max response time: 2550 ms
++++++++++++++++++++++++++++++++++++

Process finished with exit code 0
```


Throughput over maxThreads
Part 2 Throughput
1400
1050
700
350
0
32
64
128
256


Mean response time (ms) over maxThreads
Part 2 Mean Response Time
200
150
100
50
0
32
64
128
256

Conclusions

The results from Part 1 look pretty much how I expected, with wall time decreasing as the number of threads increased. My Part 2 tests make the Tomcat configuration's 200 maxThread limit much clearer! However, I saw a good amount of variation from test to test, and I got a max response time even higher than the 2500ms shown in the 256 thread output for Part 2 here. I tried playing with the Tomcat configuration a bit, and the differences weren't as pronounced as I expected. There could be something happening in the consumer threads or the server that creates a bottleneck, but I wasn't able to figure it out if so.

Another thing I played around with was increasing the maxThreads - my test with 512 threads was able to run, but 3 of the requests failed. At 1024, the server refused the requests. After spending many hours making small tweaks like moving a line of code down two lines and seeing a big difference in my results, I definitely feel a little more comfortable with threaded programming in Java now. Excited to see what's in store for assignment 2, and I look forward to your feedback on what I've got so far.