Name: Nachiket Dani

Assignment-1 Report

CS6650 : Distributed Systems

The following describes the final (Client2) design for the assignment that simulates purchase post requests through multiple threads and how the command line arguments must be set, results are printed to the console/ terminal, the overall design (UML diagram) and few results for both Clients1 and 2 as developed after deployment of the server and clients via Amazon AWS EC2 instances. Git links provided in Canvas submission.

Client Functionality

The main control functionality of the client is achieved via the "PurchaseManager" class. This class through its main method achieves the following (initiated in order):

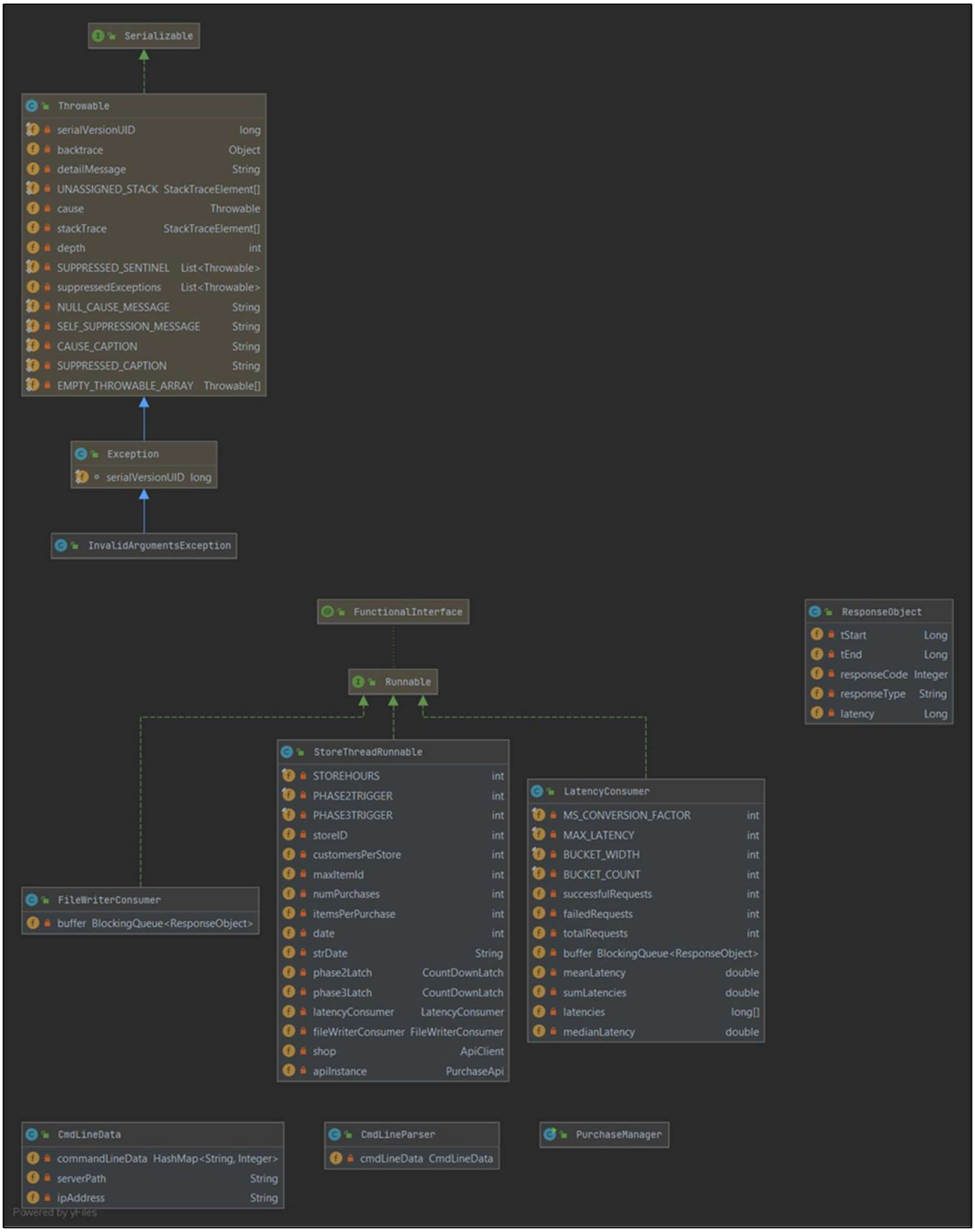
- **Action**: **Set and Parse Command Line Arguments:** calls and passes the command line arguments to the command line parser (CmdLineParser class). The command line parser validates and sets the command line data (CmdLineData class).
  Result: This operation sets the command line argument settings for the client call with regards to the following parameters and tags required by the command line:
  1. Number of stores (-maxStores)
  2. Customers per store (-customersPerStore)
  3. Max item ids (-maxItemId)
  4. Number of purchases per hour (-numPurchases)
  5. Items purchased per purchase (-itemsPerPurchase)
  6. Date of purchase (-date)
  7. Server IP and Port number to the queried (-ip ; -portNumber)

  If a tag is not specified explicity as part of the command line arguments, the setting is reverted to a default value as specified in the assignment requirement.

- **Action: Running Individual Store Threads:** The number of stores is representative of the thread count to be run. As per the requirement I break the number of stores into 3 phases: East, Central and the West phase stores. Once the east phase is triggered, the subsequent phases are chained using 2 countdown latches that are chained sequentially as they countdown based on the trigger requirement. Each phase refers to the Runnable "StoreThreadRunnable class". The threads have separate join conditions in order to await completion of the specific phase threads before we know the program has completed querying the server.
  Result: This operation triggers the purchase post requests to the server via the Runnable. The run method refers to the makePurchasePost helper that uses a try and catch block to query the server. Once the server responds (either successful or failure message), the response message is cast into a "ResponseObject" (ResponseObject class) that uses the request start, request end timestamps, request status code and a hard-coded response type (POST) within its constructor. This has been done in order to implement 2 Producer- Consumer queue co-ordination mechanisms:

- For capturing Latency aggregated statistics (LatencyConsumer class)
- For writing individual thread information into a csv file (results.csv) as per the specification requirement (FireWriterConsumer class).

Fig: (Below) UML Diagram showing the class relations and fields



Both the Producer-Consumer queues implement a LinkedBlockingQueue of our pre-defined ResponseObjects. This allows us to queue up responses from the server into this queue and avoid deadlock and race conditions as the responses are evaluated 1 by 1 for aggregating statistics and writing results into the file respectively.

a) LatencyConsumer: Implements a runnable that is started along with the store threads. Run method removes a queued ResponseObject, checks its response code and increments a successful vs failed request counter accordingly. In addition, we initiate and fill out a bucketed latency array (long array). Buckets have been pre-set based on a few assumptions:
    a. Max latency for the tomcat server on AWS assumed as 10,000ms
    b. Bucket Width: 5ms ; which creates roughly 10,000/5 buckets to for categorizing the resultant latency of a specific thread.

   We increment the count within the appropriate bucket (index) that the thread latency belongs to for every response. This is further utilized to determine the median latency.
   To calculate the mean latency, we keep a running sum of latencies as well in this class. This is divided by the total number of requests in the end to get the mean latency.

b) FileWriterConsumer: For writing results for each thread into a csv file we use a similar Producer Consumer mechanism facilitated with a LinkedBlockingQueue. We use a BufferedWriter to write individual lines using the thread information within the ResponseObject.

Back within the PurchaseManager, we closeout the LatencyConsumer and FileWriterConsumer threads through a join method.

- **Action: Calculate Wall Time and Printing Results to the Console:** We calculate the wall time using a time capture before running the threads and after they have run join. We also print out results as required to the console/terminal. Results for the Client1 and Client2 are different as Client2 captures granular raw data for each thread and additionally provides a mean, median and 99th percentile statistic.

Server Functionality

The Server is implemented by examining the requirement from the POST Purchase servlet described in the existing Swagger client authored by Prof. Gorton.

Ref: https://app.swaggerhub.com/apis/gortonator/GianTigle/1.0.0#/purchase/newPurchase

The Swagger codegen was used to create the Swagger client API interface which the client is built on.

Server implements a simple URL validation methodology. To verify the request body, I created a POJO PurchaseItem and Purchase. Then used GSON to decode the incoming JSON string and test casting into the POJO. Failure to do so was considered a POST failure.

Results

The following data and graph showcase a comparison between the latencies observed for Client1 and Client2.

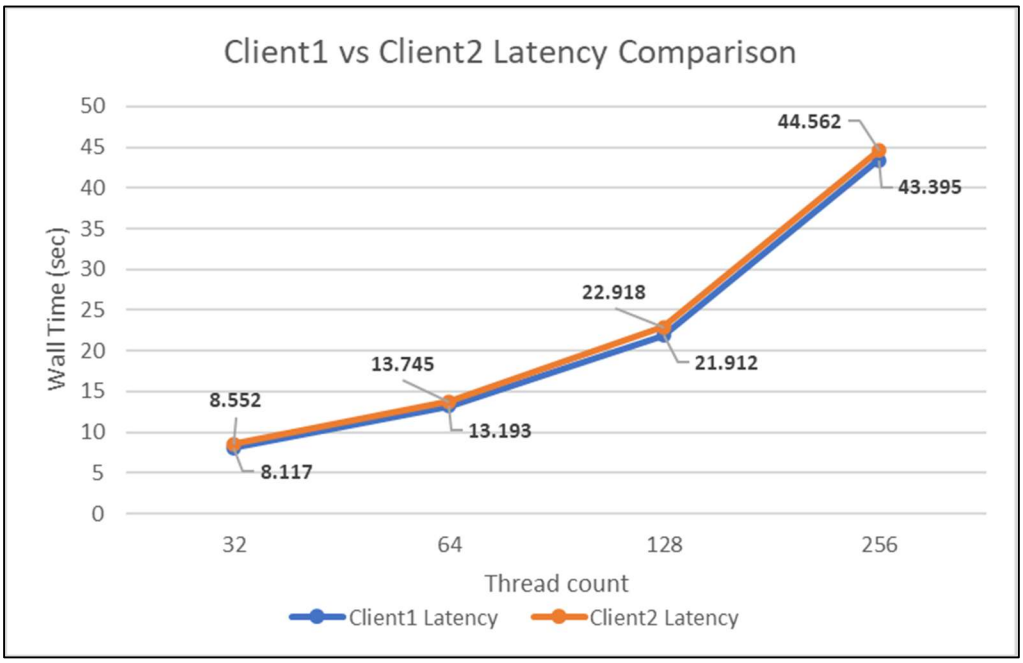


fig: (Above) Comparison of Client1 and Client2 Latencies using sample thread counts. I have added wall time for Client 2 as a additional comparison. I see that the 2 differ very slightly despite having large differences within how they operate. I continue to use this raw data I achieved using 32, 64, 128, 256 threads to further exhibit the Client2 results.

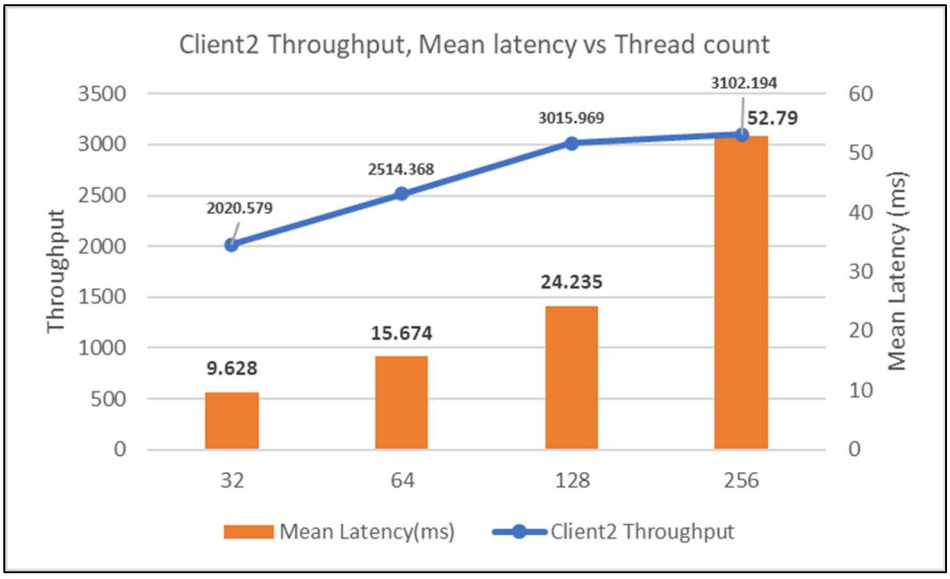
(Below) Raw data from the comparisons showing <=5% difference between latencies for the 2 client implementations

MaxStores	Client1 Latency	Client2 Latency	% Change
32	8.117	8.552	5%
64	13.193	13.745	4%
128	21.912	22.918	5%
256	43.395	44.562	3%

Following is a Client2 summary: Throughput, Average latency against the thread count:

MaxStores	Client2 Throughput	Mean Latency(ms)
32	2020.579	9.628
64	2514.368	15.674
128	3015.969	24.235
256	3102.194	52.79

Graphical Representation: We observe that the latency and throughput grow as a function of the thread count. I assume the throughput will flatten out as we increase the number of threads further beyond 256. I was able to capture total failure of the server to handle 1024 requests from client 2.



Following are the screengrabs from the Raw data capture for Client 1 (32,64,128,256 threads in order):

```
ec2-user@ip-172-31-18-243:~/assignment1
[ec2-user@ip-172-31-18-243 assignment1]$ java -jar CS6650_a1Client1.jar -ip 54.163.156.157 -portNumber 8080 -maxStores 32
Configuration settings:
{date=20210101, numPurchases=60, customersPerStore=1000, maxItemId=100000, itemsPerPurchase=5, maxStores=32, portNumber=8080}
Server address queried:
http://54.163.156.157:8080/CS6650_a1Servlet_war
----------------------------
RESULT REPORT:
----------------------------
There were 17280 successful purchases posted!
0 requests failed to post.
A total of 17280 requests were made.
Processing purchases for the stores took 8.117 seconds (Wall Time)
----------------------------
```

```
ec2-user@ip-172-31-18-243:~/assignment1
[ec2-user@ip-172-31-18-243 assignment1]$ java -jar CS6650_a1Client1.jar -ip 54.163.156.157 -portNumber 8080 -maxStores 64
Configuration settings:
{date=20210101, numPurchases=60, customersPerStore=1000, maxItemId=100000, itemsPerPurchase=5, maxStores=64, portNumber=8080}
Server address queried:
http://54.163.156.157:8080/CS6650_a1Servlet_war
----------------------------
RESULT REPORT:
----------------------------
There were 34560 successful purchases posted!
0 requests failed to post.
A total of 34560 requests were made.
Processing purchases for the stores took 13.193 seconds (Wall Time)
----------------------------
```

```
ec2-user@ip-172-31-18-243:~/assignment1
[ec2-user@ip-172-31-18-243 assignment1]$ java -jar CS6650_a1Client1.jar -ip 54.163.156.157 -portNumber 8080 -maxStores 128
Configuration settings:
{date=20210101, numPurchases=60, customersPerStore=1000, maxItemId=100000, itemsPerPurchase=5, maxStores=128, portNumber=8080}
Server address queried:
http://54.163.156.157:8080/CS6650_a1Servlet_war
----------------------------
RESULT REPORT:
----------------------------
There were 69120 successful purchases posted!
0 requests failed to post.
A total of 69120 requests were made.
Processing purchases for the stores took 21.912 seconds (Wall Time)
----------------------------
```

```
ec2-user@ip-172-31-18-243:~/assignment1
[ec2-user@ip-172-31-18-243 assignment1]$ java -jar CS6650_a1Client1.jar -ip 54.163.156.157 -portNumber 8080 -maxStores 256
Configuration settings:
{date=20210101, numPurchases=60, customersPerStore=1000, maxItemId=100000, itemsPerPurchase=5, maxStores=256, portNumber=8080}
Server address queried:
http://54.163.156.157:8080/CS6650_a1Servlet_war
----------------------------
RESULT REPORT:
----------------------------
There were 138240 successful purchases posted!
0 requests failed to post.
A total of 138240 requests were made.
Processing purchases for the stores took 43.395 seconds (Wall Time)
----------------------------
```

Following are the screengrabs from the Raw data capture for Client 2 (32,64,128,256 threads in order):

```
ec2-user@ip-172-31-18-243:~/assignment1
[ec2-user@ip-172-31-18-243 assignment1]$ java -jar CS6650_a1Client2.jar -ip 54.163.156.157 -portNumber 8080 -maxStores 32
Configuration settings:
{date=20210101, numPurchases=60, customersPerStore=1000, maxItemId=100000, itemsPerPurchase=5, maxStores=32, portNumber=8080}
Server address queried:
http://54.163.156.157:8080/CS6650_a1Servlet_war
----------------------------
RESULT REPORT:
----------------------------
There were 17280 successful purchases posted!
0 requests failed to post.
A total of 17280 requests were made.
The requests were processed in 8.552seconds. (Wall Time)
Throughput: 2020.5799812909263 requests/second
The average latency was 9.628240740740742 seconds.
The median latency was 12.0 seconds.
----------------------------
```

```
ec2-user@ip-172-31-18-243:~/assignment1
[ec2-user@ip-172-31-18-243 assignment1]$ java -jar CS6650_a1Client2.jar -ip 54.163.156.157 -portNumber 8080 -maxStores 64
Configuration settings:
{date=20210101, numPurchases=60, customersPerStore=1000, maxItemId=100000, itemsPerPurchase=5, maxStores=64, portNumber=8080}
Server address queried:
http://54.163.156.157:8080/CS6650_a1Servlet_war
----------------------------
RESULT REPORT:
----------------------------
There were 34560 successful purchases posted!
0 requests failed to post.
A total of 34560 requests were made.
The requests were processed in 13.745seconds. (Wall Time)
Throughput: 2514.368861404147 requests/second
The average latency was 15.674421296296297 seconds.
The median latency was 17.0 seconds.
----------------------------
```

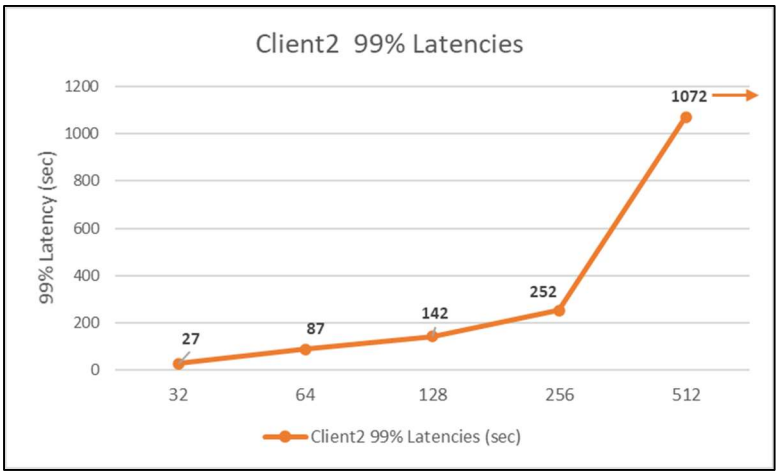
```
ec2-user@ip-172-31-18-243:~/assignment1
[ec2-user@ip-172-31-18-243 assignment1]$ java -jar CS6650_a1Client2.jar -ip 54.163.156.157 -portNumber 8080 -maxStores 256
Configuration settings:
{date=20210101, numPurchases=60, customersPerStore=1000, maxItemId=100000, itemsPerPurchase=5, maxStores=256, portNumber=8080}
Server address queried:
http://54.163.156.157:8080/CS6650_a1Servlet_war
----------------------------
RESULT REPORT:
----------------------------
There were 138240 successful purchases posted!
0 requests failed to post.
A total of 138240 requests were made.
The requests were processed in 44.562seconds. (Wall Time)
Throughput: 3102.1946950316415 requests/second
The average latency was 52.790183738425924 seconds.
The median latency was 37.0 seconds.
----------------------------
```

```
ec2-user@ip-172-31-18-243:~/assignment1
[ec2-user@ip-172-31-18-243 assignment1]$ java -jar CS6650_a1Client2.jar -ip 54.163.156.157 -portNumber 8080 -maxStores 128
Configuration settings:
{date=20210101, numPurchases=60, customersPerStore=1000, maxItemId=100000, itemsPerPurchase=5, maxStores=128, portNumber=8080}
Server address queried:
http://54.163.156.157:8080/CS6650_a1Servlet_war
---------------------------
RESULT REPORT:
---------------------------
There were 69120 successful purchases posted!
0 requests failed to post.
A total of 69120 requests were made.
The requests were processed in 22.918seconds. (Wall Time)
Throughput: 3015.969979928441 requests/second
The average latency was 24.23553240740741 seconds.
The median latency was 22.0 seconds.
---------------------------
```

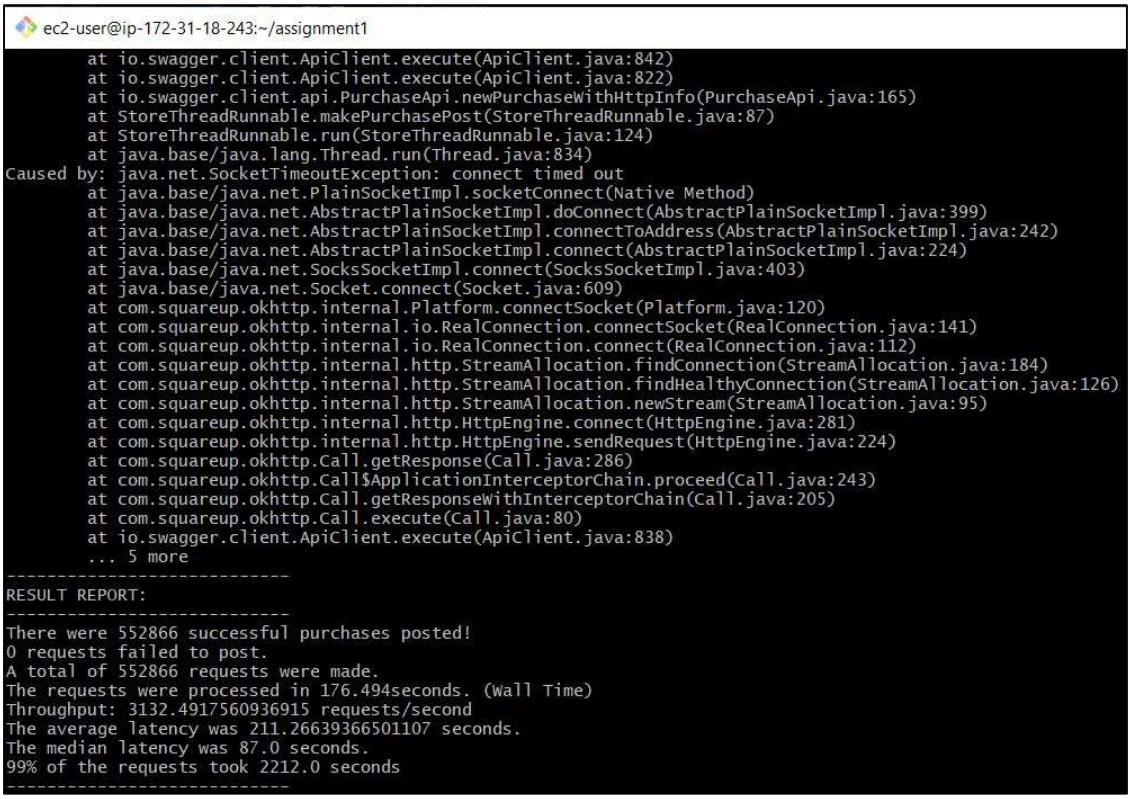
Additional Observations, Breaks

Analysis and Observation of Client2 99% percentile data for 32, 64, 128, 256, 512 threads:

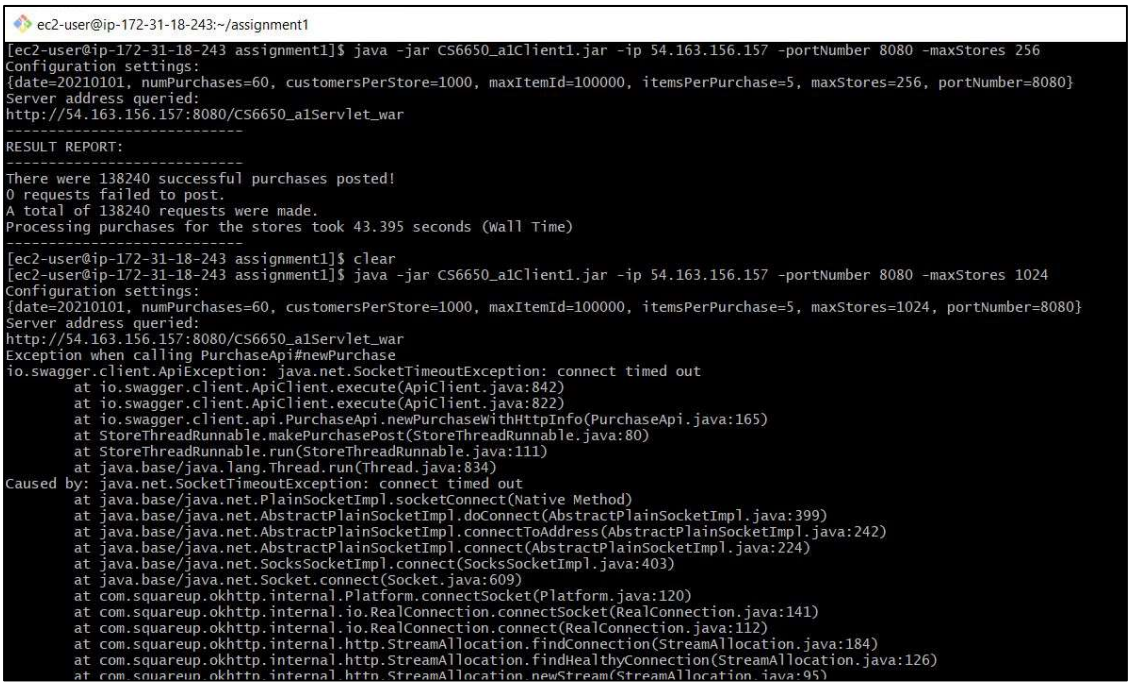
MaxStores	Client2 99% Latencies (sec)
32	27
64	87
128	142
256	252
512	1072
1024	break



I observe that the client2 throws Socket Timeout exceptions for a thread count of 1024 and 'breaks'. However, from the result report that is obtained at the end of the run, it is observed that the client believes requests were posted successfully. Which is not true since I did not account for requests that would get dropped entirely by the server due to bottlenecks. Additionally, I did not account of retry of post requests that fail to reach the server.



Additionally, as seen in the screengrabs above for both Client1 and 2, the clients fail to complete post requests and timeout for 1024 threads.



```
ec2-user@ip-172-31-18-243:~/assignment1
[ec2-user@ip-172-31-18-243 assignment1]$ java -jar CS6650_a1Client2.jar -ip 54.163.156.157 -portNumber 8080 -maxStores 1024
Configuration settings:
{date=20210101, numPurchases=60, customersPerStore=1000, maxItemId=100000, itemsPerPurchase=5, maxStores=1024, portNumber=8080}
Server address queried:
http://54.163.156.157:8080/CS6650_a1Servlet_war
Exception when calling PurchaseApi#newPurchase
io.swagger.client.ApiException: java.net.SocketTimeoutException: connect timed out
        at io.swagger.client.ApiClient.execute(ApiClient.java:842)
        at io.swagger.client.ApiClient.execute(ApiClient.java:822)
        at io.swagger.client.api.PurchaseApi.newPurchaseWithHttpInfo(PurchaseApi.java:165)
        at StoreThreadRunnable.makePurchasePost(StoreThreadRunnable.java:87)
        at StoreThreadRunnable.run(StoreThreadRunnable.java:124)
        at java.base/java.lang.Thread.run(Thread.java:834)
Caused by: java.net.SocketTimeoutException: connect timed out
        at java.base/java.net.PlainSocketImpl.socketConnect(Native Method)
        at java.base/java.net.AbstractPlainSocketImpl.doConnect(AbstractPlainSocketImpl.java:399)
        at java.base/java.net.AbstractPlainSocketImpl.connectToAddress(AbstractPlainSocketImpl.java:242)
        at java.base/java.net.AbstractPlainSocketImpl.connect(AbstractPlainSocketImpl.java:224)
        at java.base/java.net.SocksSocketImpl.connect(SocksSocketImpl.java:403)
        at java.base/java.net.Socket.connect(Socket.java:609)
        at com.squareup.okhttp.internal.Platform.connectSocket(Platform.java:120)
        at com.squareup.okhttp.internal.io.RealConnection.connectSocket(RealConnection.java:141)
        at com.squareup.okhttp.internal.io.RealConnection.connect(RealConnection.java:112)
        at com.squareup.okhttp.internal.http.StreamAllocation.findConnection(StreamAllocation.java:184)
        at com.squareup.okhttp.internal.http.StreamAllocation.findHealthyConnection(StreamAllocation.java:126)
        at com.squareup.okhttp.internal.http.StreamAllocation.newStream(StreamAllocation.java:95)
        at com.squareup.okhttp.internal.http.HttpEngine.connect(HttpEngine.java:281)
        at com.squareup.okhttp.internal.http.HttpEngine.sendRequest(HttpEngine.java:224)
        at com.squareup.okhttp.Call.getResponse(Call.java:286)
        at com.squareup.okhttp.Call$ApplicationInterceptorChain.proceed(Call.java:243)
        at com.squareup.okhttp.Call.getResponseWithInterceptorChain(Call.java:205)
        at com.squareup.okhttp.Call.execute(Call.java:80)
        at io.swagger.client.ApiClient.execute(ApiClient.java:838)
        ... 5 more
Exception when calling PurchaseApi#newPurchase
io.swagger.client.ApiException: java.net.SocketTimeoutException: connect timed out
        at io.swagger.client.ApiClient.execute(ApiClient.java:842)
        at io.swagger.client.ApiClient.execute(ApiClient.java:822)
        at io.swagger.client.api.PurchaseApi.newPurchaseWithHttpInfo(PurchaseApi.java:165)
        at StoreThreadRunnable.makePurchasePost(StoreThreadRunnable.java:87)
```

Both Server and Client were deployed on separate instances of Amazon EC2 (free tier, Virginia).

Server deployed at:

http://54.163.156.157:8080/CS6650_a1Servlet_war

Sample Postman Query with the appropriate JSON body to:

http://54.163.156.157:8080/CS6650_a1Servlet_war/purchase/8/customer/8/date/20210102

Sample Command line execution arguments in AWS (via ssh):

Client1:

>> java -jar CS6650_a1Client1.jar -ip 54.163.156.157 -port 8080 -maxStores 256

Client2:

>> java -jar CS6650_a1Client2.jar -ip 54.163.156.157 -port 8080 -maxStores 256