Name: Nachiket Dani

Assignment-3 Report

CS6650 : Distributed Systems

The following describes the assignment that advances further on top of the client and database service developed within assignment2. In this assignment I transform the server-side architecture into a Microservices for Purchase Store viz. in-memory store for Purchase data. We retain the Database storage service however we now introduce a Client pool of threads that feed Purchase data from the HTTP request body into the rabbitmq exchange. We deploy a RabbitMQ RPC styled architecture for the PurchaseStore Microservice and a simple fan-out that delivers purchase data to both the Database and PurchaseStore. The following depicts the container system architecture for the system as prepared. Client still simulates purchase post requests through multiple threads comparing a single server to a scaled (4) server setup with a load balancer within AWS environment. Git links to code base: (Contains separate folders for Client, Servlets and services.

A3: *https://github.com/NachiketDani/CS6650_DistributedSystems/tree/main/CS6650_a3*

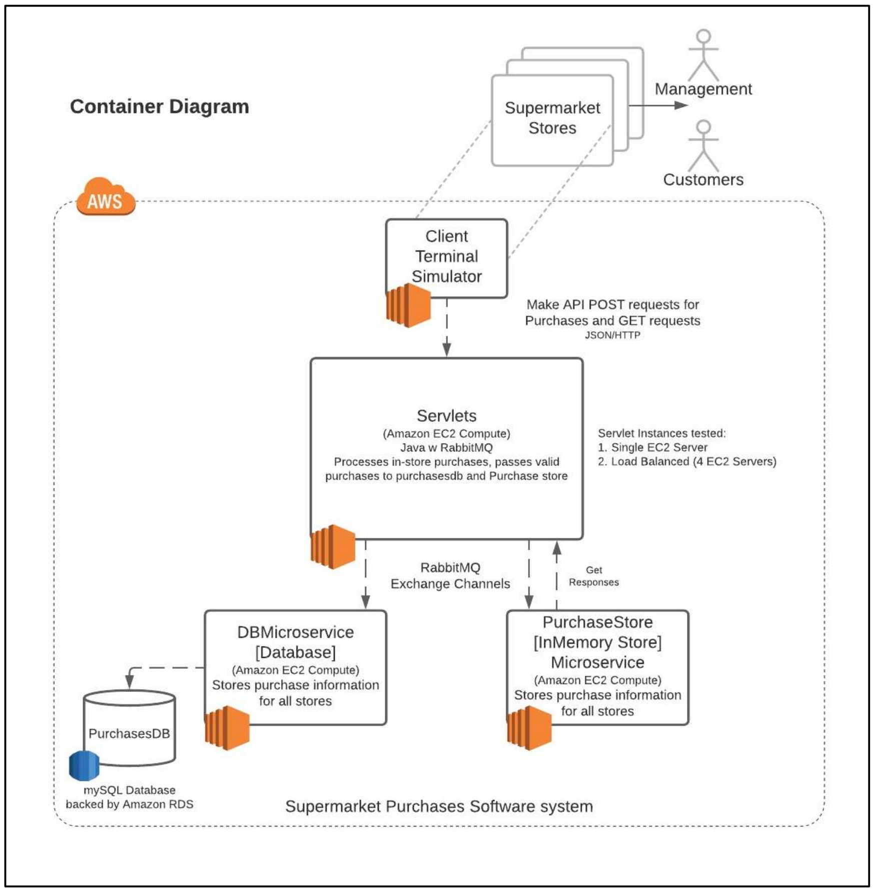

Container Diagram
Supermarket Stores
Management
Customers
AWS
Client Terminal Simulator
Make API POST requests for Purchases and GET requests
JSON/HTTP
Servlets
(Amazon EC2 Compute)
Java w RabbitMQ
Processes in-store purchases, passes valid purchases to purchasesdb and Purchase store
Servlet Instances tested:
1. Single EC2 Server
2. Load Balanced (4 EC2 Servers)
RabbitMQ Exchange Channels
Get Responses
DBMicroservice [Database]
(Amazon EC2 Compute)
Stores purchase information for all stores
PurchaseStore [InMemory Store] Microservice
(Amazon EC2 Compute)
Stores purchase information for all stores
PurchasesDB
mySQL Database backed by Amazon RDS
Supermarket Purchases Software system

fig: System Architecture: Container diagram

Database Service:

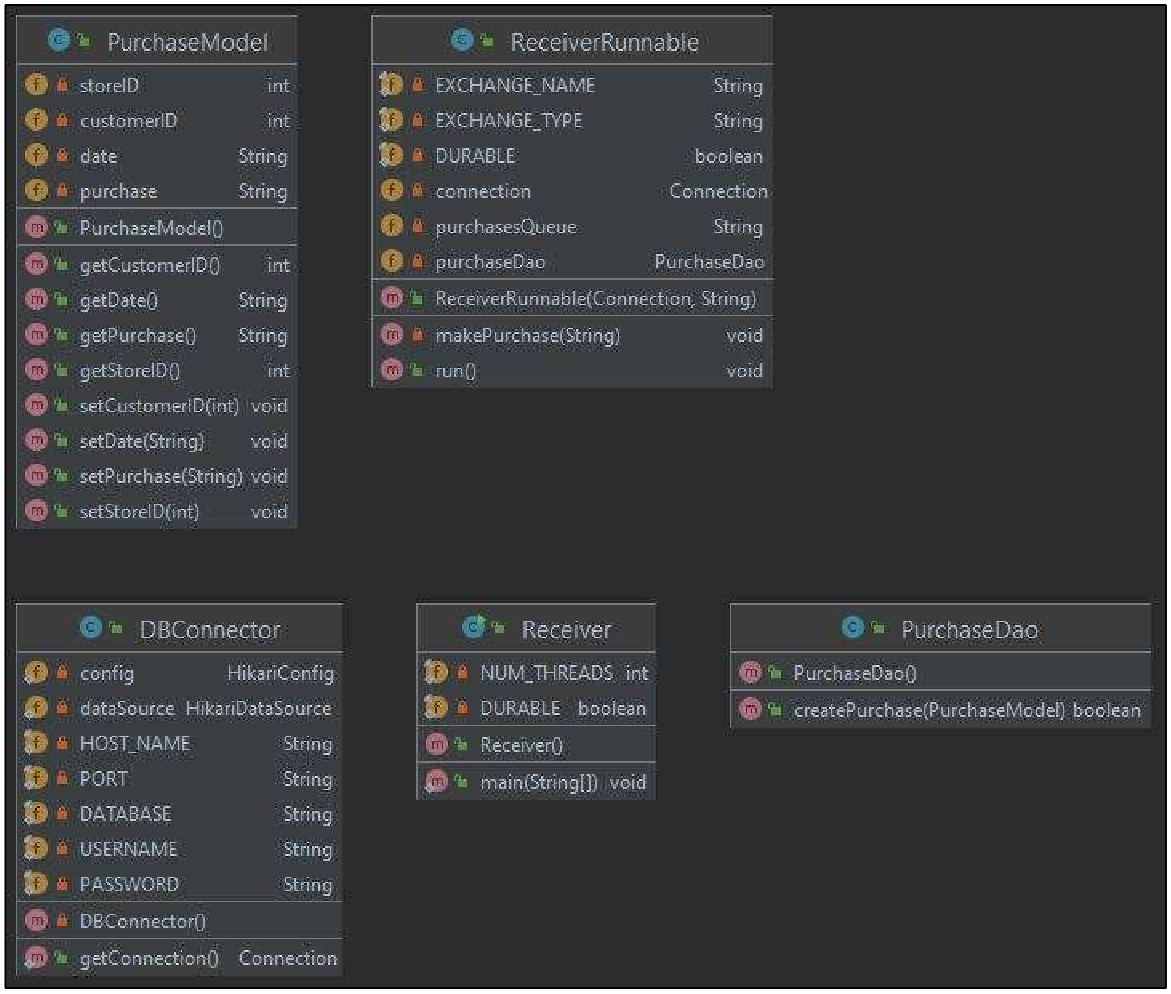


fig: Database service UML diagram

A database stores purchase data from the client made POST requests. We use an Amazon RDS storage via a local mySQL Workbench instance. We create a new POJO: PurchaseModel that encases the Purchase class which in turn holds PurchaseItems. This allows us to use a DAO layer (PurchaseDao) and a database connector class (DBConnector) to pass the parsed request body information to the database. The PurchaseModel created reflects the database schema. It consists of the fields: storeID, customerID, purchaseDate and items. We do not use a primary key for the table as we want to prioritize data insertion. Additionally, we use a JSON data type for items. Refer "PurchasesDB_Schema.sql" for the database schema. This file is embedded in the CS6650_a2Servlet project files.The DBConnector uses HikariCP which is a lightweight JDBC connection pool to handle the volume of write requests effectively. This decision was made after consulting the TA since the Apache Commons DBCP was not able to handle the data flow.

We use the fanout RabbitMQ feature to asynchronously couple the 2 components. On the database component side, we deploy 75 threads to feed purchase data into the database.

Purchase Store Microservice:

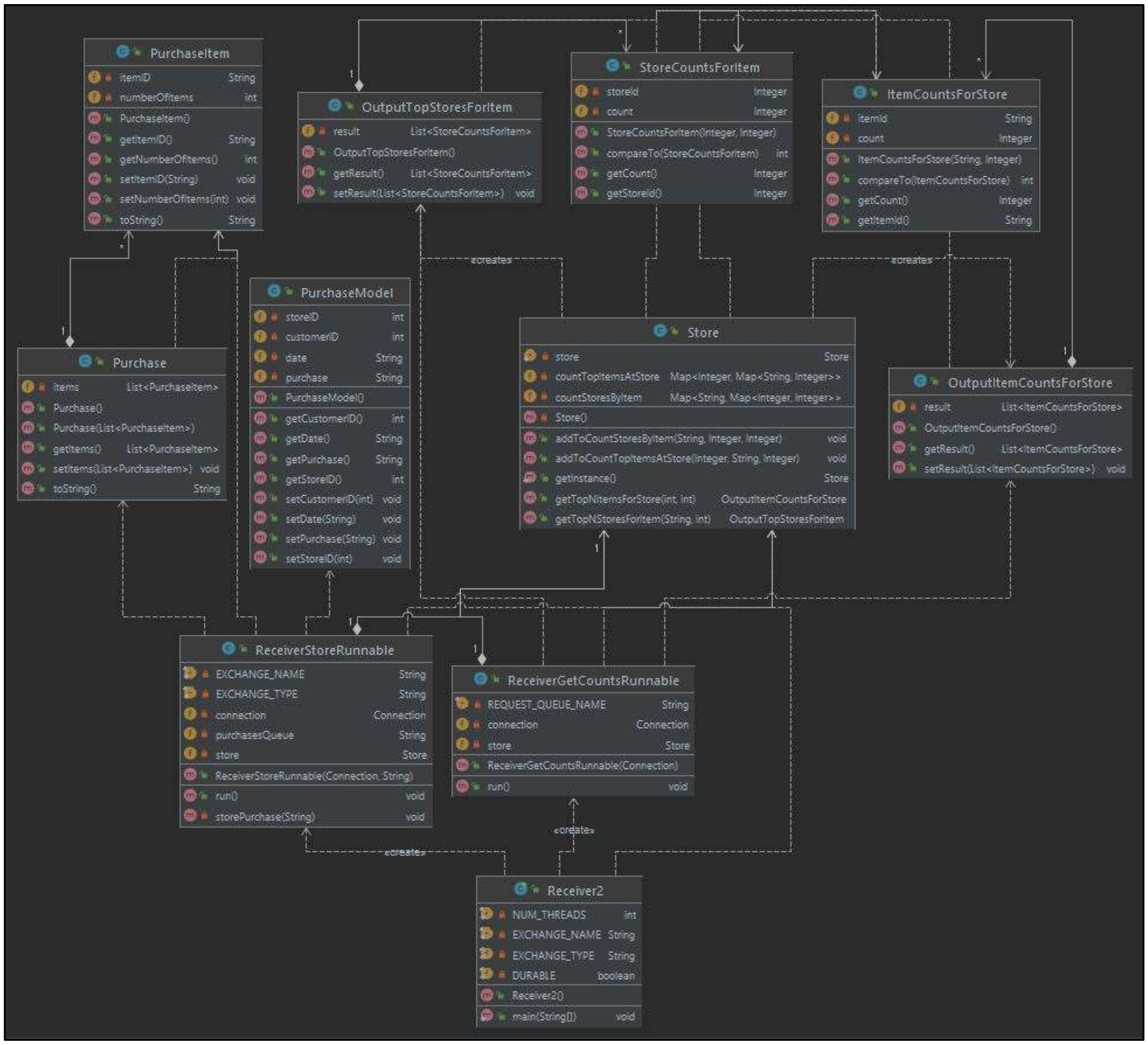


fig: PurchaseStore Microservice UML Diagram

The PurchaseStore Microservice implements the receiver main method for RabbitMQ (Receiver2) through 2 separate Runnable thread mechanisms:

- ReceiverStoreRunnable: Runnable that uses the fanout channel for receiving purchases from the server and storing them in 2 key-value stores/ Maps implemented within the Store class. We need 2 maps to address 2 separate data needs mandated by the GET requests.
- ReceiverGetCountsRunnable: This thread mechanism addresses both types of GET requests based on the HTTPRequest details parsed and forwarded by the Servlet. This method uses the 2 maps to create PriorityQueues (heaps) to obtain ascending counts of both Items sold at each store and stores each item has sales in. The assignment specification mentions of the top 10 in each GET request. However I have used a more generic way that allows the user to change it to whichever

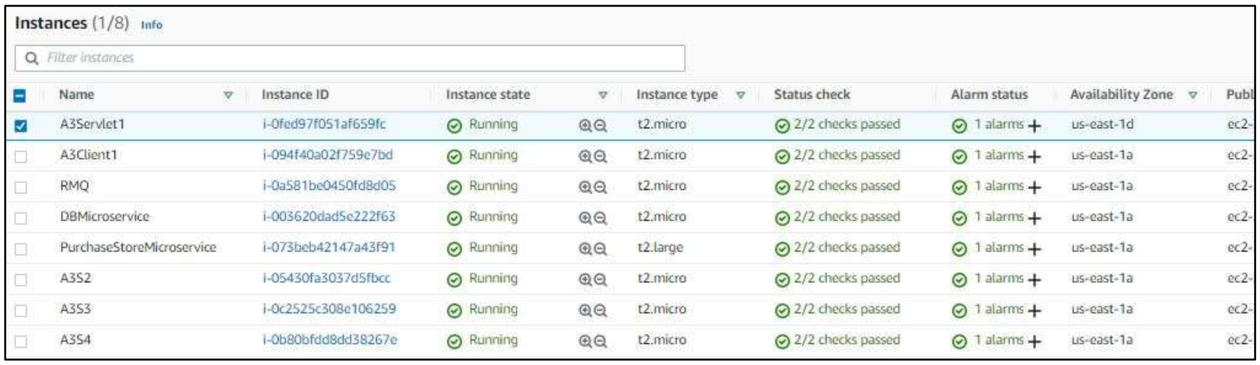
range they need based on the GET request. The heap then is popped and feeds into the result as needed. Refer to Store.java.

The PurchaseModel, Purchase, PurchaseItem classes serve as data models for the database model, purchase data and purchased item formats. Classes ItemCountsForStore and StoreCountsForItem serve to create Pairs for counts of items and sales at stores and item, store ids. This is essential to implement the compareTo method that helps us build the PriorityQueues (heaps) in the Store class. The OutputItemCountsForStore and OutputTopStoresForItem classes serve to create a structure for the output data formed from the GET requests. This data is further converted to strings and sent back as response through the rpc queue of RabbitMQ.

Experimentation:

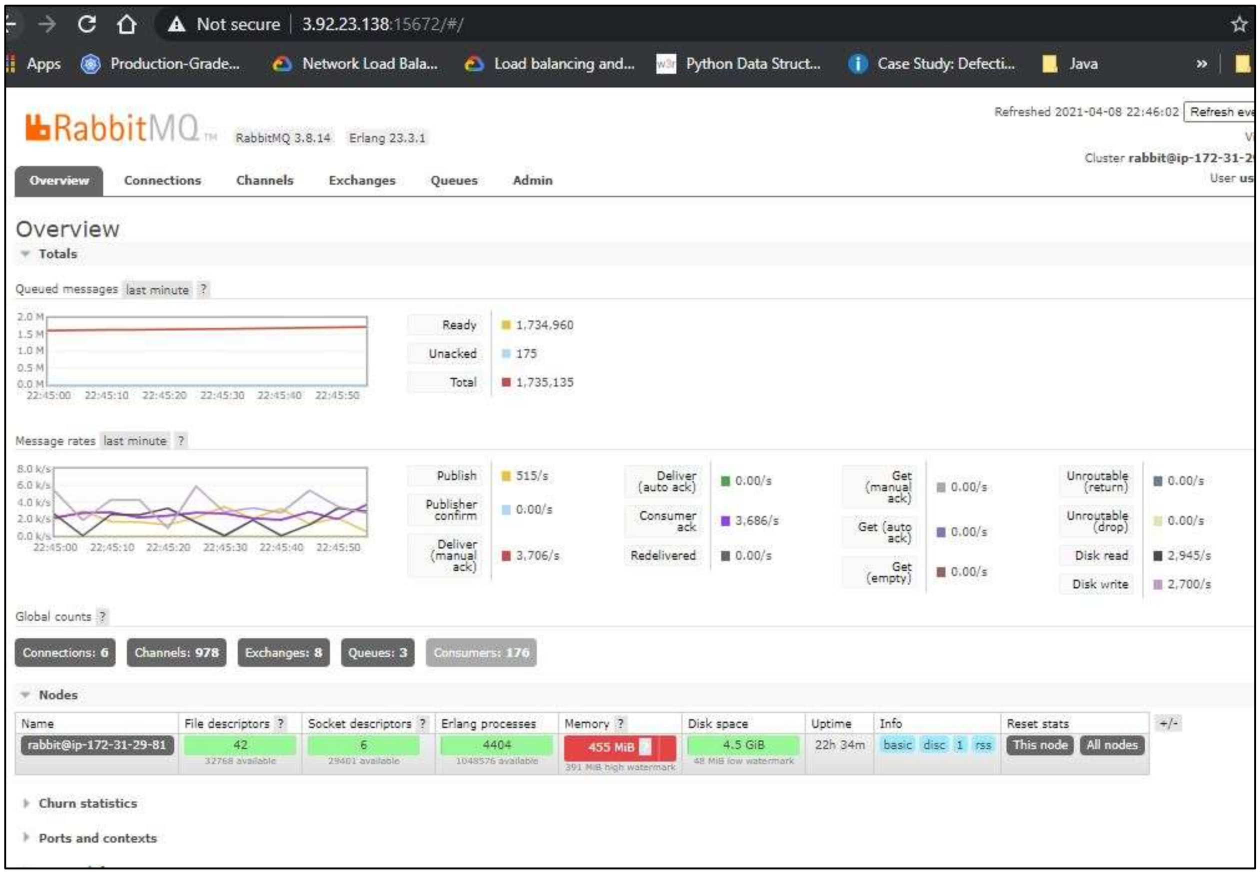
I was able to run the Single Server as well as the Scaled system and write records into the database for 32, 64, 128, 256 and 512 threads (maxStores). I ran into some issues executing the multi-server load balancer instance. Service was inconsistent and ran well for some tests while threw exceptions and timeouts for another. I believe these issues must have been due to latency issues faced with the load balancer and AWS EC2 instances while working with RabbitMQ. I ran both the Single server and Multi-Server for 1024 threads (stores), however both configurations failed to perform at this level. Additionally, the RabbitMQ instance was consistently running out of memory as captured in a screengrab below.

Both Server and Client were deployed on separate instances of Amazon EC2 (free tier, Virginia). I used the image created of the single server to create additional instances (3) and used a application load balancer to coordinate usage between the replicas. *(Images now disabled)*



Name	Instance ID	Instance state	Instance type	Status check	Alarm status	Availability Zone	Publ
A3Servlet1	i-0fed97f051af659fc	Running	t2.micro	2/2 checks passed	1 alarms +	us-east-1d	ec2-
A3Client1	i-094f40a02f759e7bd	Running	t2.micro	2/2 checks passed	1 alarms +	us-east-1a	ec2-
RMQ	i-0a581be0450fd8d05	Running	t2.micro	2/2 checks passed	1 alarms +	us-east-1a	ec2-
DBMicroservice	i-003620dad5e222f63	Running	t2.micro	2/2 checks passed	1 alarms +	us-east-1a	ec2-
PurchaseStoreMicroservice	i-073beb42147a43f91	Running	t2.large	2/2 checks passed	1 alarms +	us-east-1a	ec2-
A3S2	i-05430fa3037d5fbcc	Running	t2.micro	2/2 checks passed	1 alarms +	us-east-1a	ec2-
A3S3	i-0c2525c308e106259	Running	t2.micro	2/2 checks passed	1 alarms +	us-east-1a	ec2-
A3S4	i-0b80bfdd8dd38267e	Running	t2.micro	2/2 checks passed	1 alarms +	us-east-1a	ec2-

fig: AWS EC2 Instances deployed



Refreshed 2021-04-08 22:46:02

Cluster rabbit@ip-172-31-2

Overview

Queued messages last minute

Ready	1,734,960
Unacked	175
Total	1,735,135

Message rates last minute

Publish	515/s	Deliver (auto ack)	0.00/s	Get (manual ack)	0.00/s	Unroutable (return)	0.00/s		
Publisher confirm	0.00/s	Consumer ack	3,686/s	Get (auto ack)	0.00/s	Unroutable (drop)	0.00/s		
Deliver (manual ack)	3,706/s	Redelivered	0.00/s	Get (empty)	0.00/s	Disk read	2,945/s		
						Disk write	2,700/s		

Global counts

Connections: 6 Channels: 978 Exchanges: 8 Queues: 3 Consumers: 176

Nodes

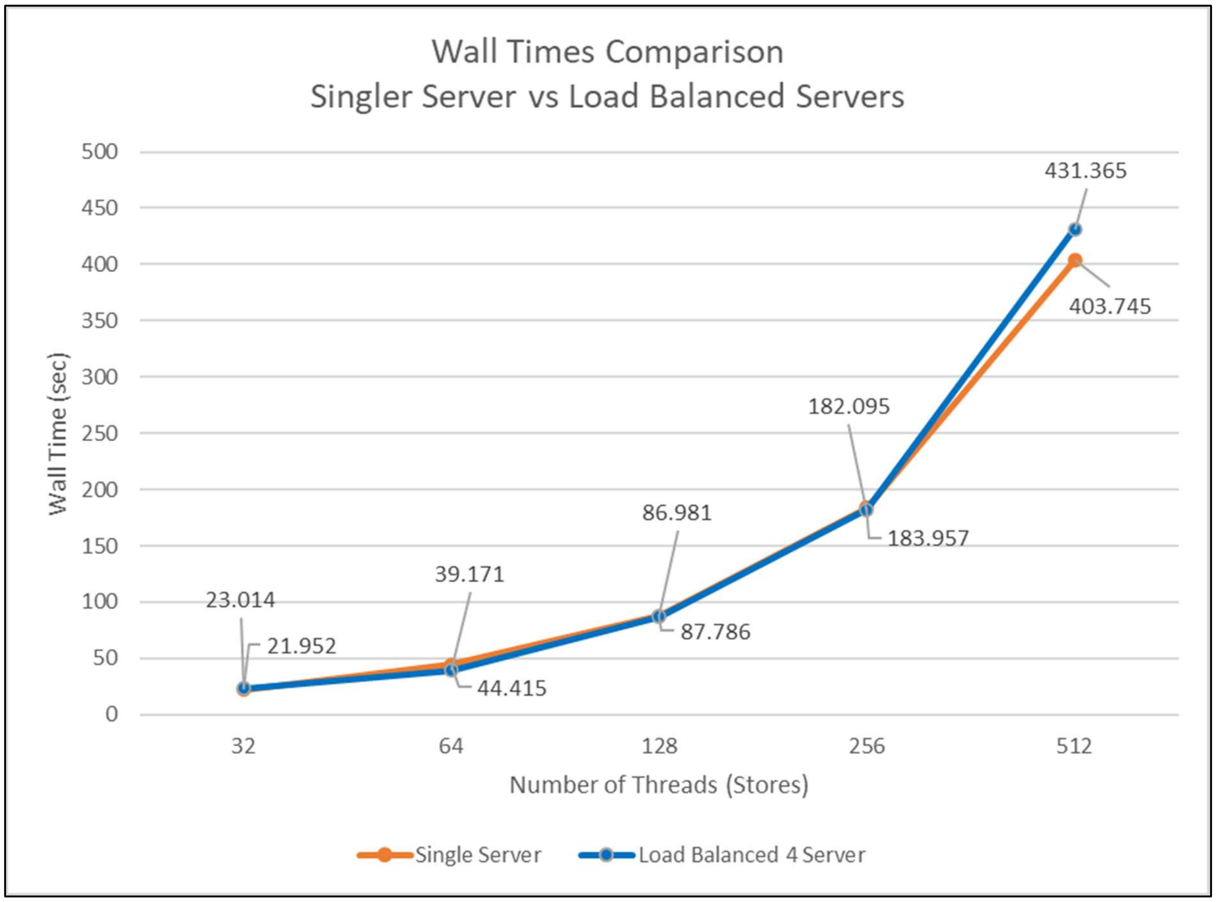
Name	File descriptors	Socket descriptors	Erlang processes	Memory	Disk space	Uptime	Info	Reset stats
rabbit@ip-172-31-29-81	42 32768 available	6 29401 available	4404 1048576 available	455 MiB 391 MiB high watermark	4.5 GiB 48 MiB low watermark	22h 34m	basic disc 1 rss	This node All nodes

fig: 1024 thread instance where RabbitMQ runs out of memory mid process

Comparison Results

Wall time Comparison

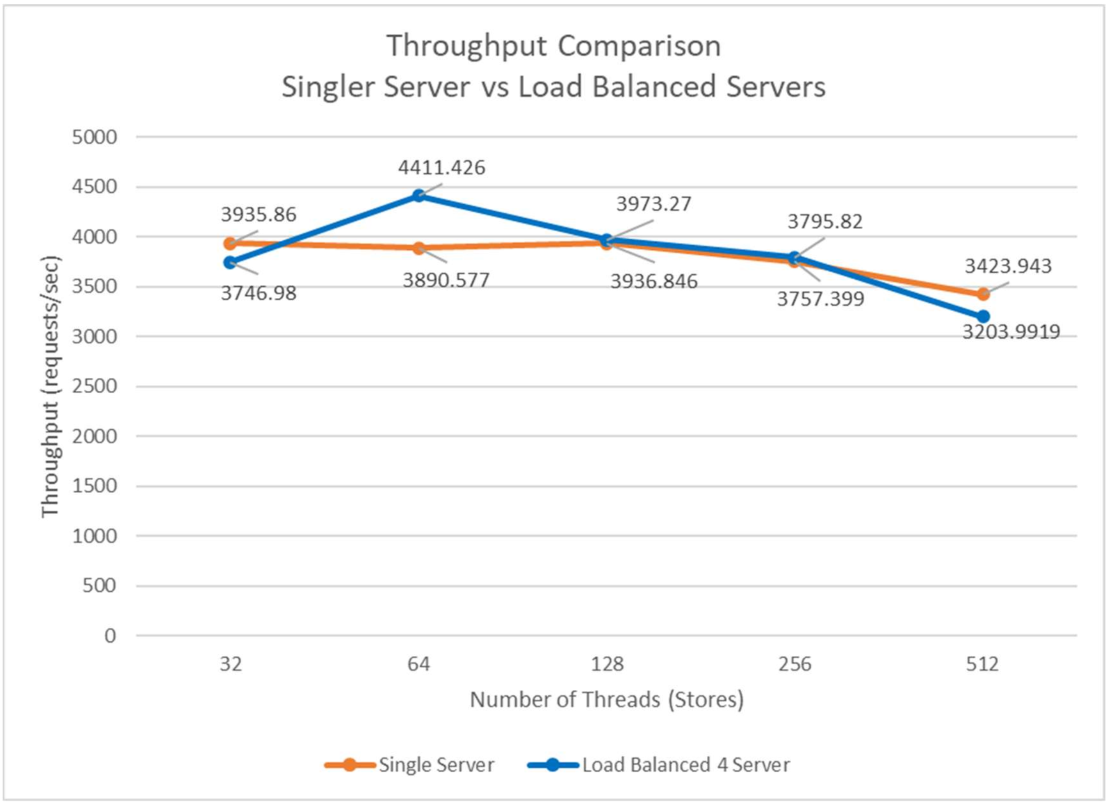
Number of Threads	Single Server	Load Balanced 4 Server
32	21.952	23.014
64	44.415	39.171
128	87.786	86.981
256	183.957	182.095
512	403.745	431.365



We observe that the wall times this time around did not improve that significantly for our purchase system. This may be in part due to the size of the channel pool we use for our client. Additional latency may have been introduced due to bad internet service at my end. Further experimentation and tuning will be needed to take advantage of the multi-server load balanced system further since it theoretically should have lower wall-times. These results can be considered inconclusive based on the current parameters.

Throughput comparison

Number of Threads	Single Server	Load Balanced 4 Server
32	3935.86	3746.98
64	3890.577	4411.426
128	3936.846	3973.27
256	3757.399	3795.82
512	3423.943	3203.9919



Throughput for the scaled system is also not conclusive in terms of showcasing the better strategy. We do see that however for both systems as thread counts increase, throughput stalls and falls off in both cases.

System Design Considerations

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

Ref: https://app.swaggerhub.com/apis/gortonator/GianTigle/1.11

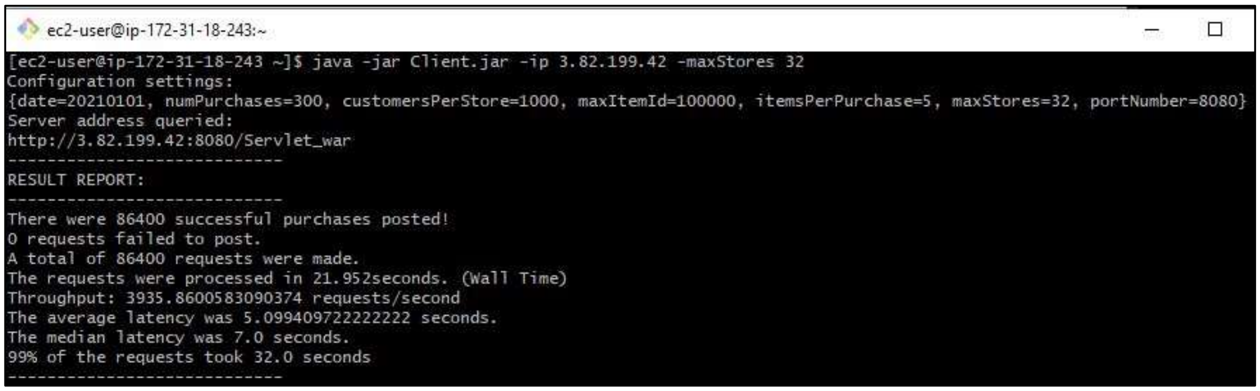
The Swagger codegen was used to create the Swagger client API interface which the client is built on.

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.

Raw Data Screengrabs

Single Server Data:

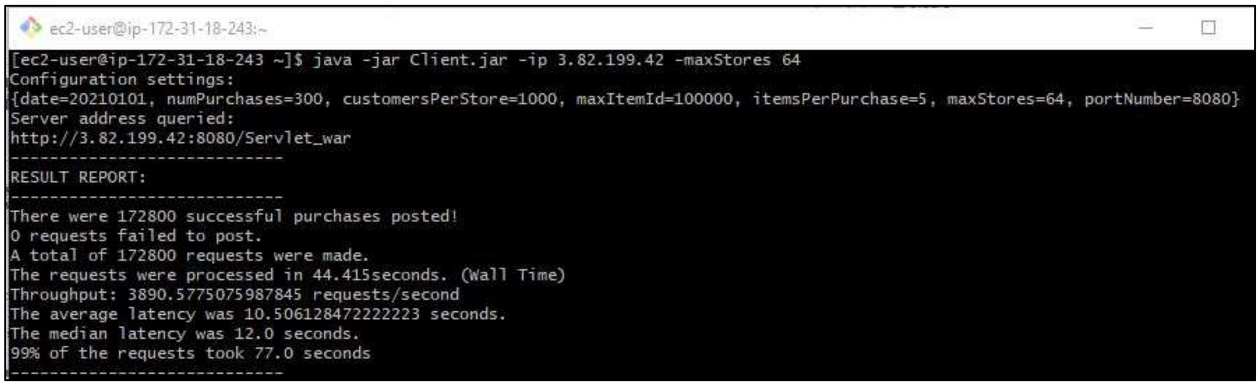
32 Threads:



ec2-user@ip-172-31-18-243:~

```
[ec2-user@ip-172-31-18-243 ~]$ java -jar Client.jar -ip 3.82.199.42 -maxStores 32
Configuration settings:
{date=20210101, numPurchases=300, customersPerStore=1000, maxItemId=100000, itemsPerPurchase=5, maxStores=32, portNumber=8080}
Server address queried:
http://3.82.199.42:8080/Servlet_war
----------------------------
RESULT REPORT:
----------------------------
There were 86400 successful purchases posted!
0 requests failed to post.
A total of 86400 requests were made.
The requests were processed in 21.952seconds. (Wall Time)
Throughput: 3935.8600583090374 requests/second
The average latency was 5.099409722222222 seconds.
The median latency was 7.0 seconds.
99% of the requests took 32.0 seconds
----------------------------
```

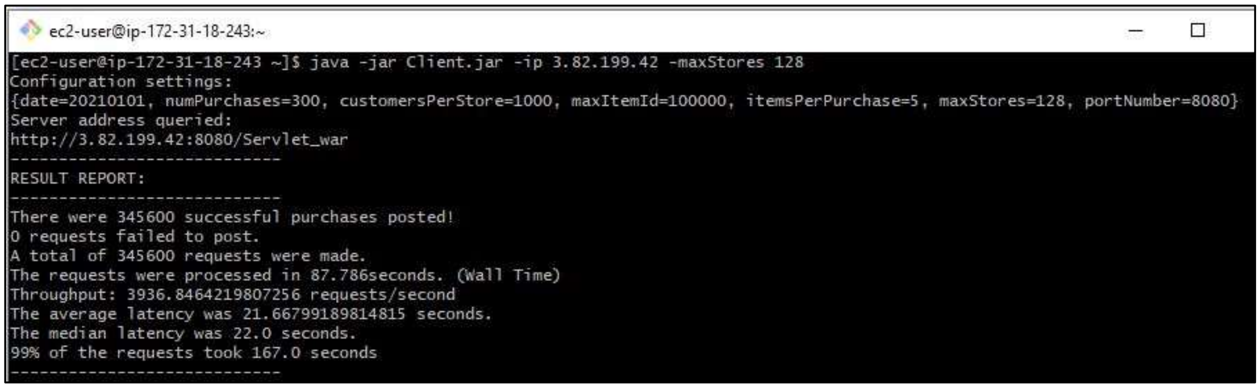
64 Threads:



ec2-user@ip-172-31-18-243:~

```
[ec2-user@ip-172-31-18-243 ~]$ java -jar Client.jar -ip 3.82.199.42 -maxStores 64
Configuration settings:
{date=20210101, numPurchases=300, customersPerStore=1000, maxItemId=100000, itemsPerPurchase=5, maxStores=64, portNumber=8080}
Server address queried:
http://3.82.199.42:8080/Servlet_war
----------------------------
RESULT REPORT:
----------------------------
There were 172800 successful purchases posted!
0 requests failed to post.
A total of 172800 requests were made.
The requests were processed in 44.415seconds. (Wall Time)
Throughput: 3890.5775075987845 requests/second
The average latency was 10.506128472222223 seconds.
The median latency was 12.0 seconds.
99% of the requests took 77.0 seconds
----------------------------
```

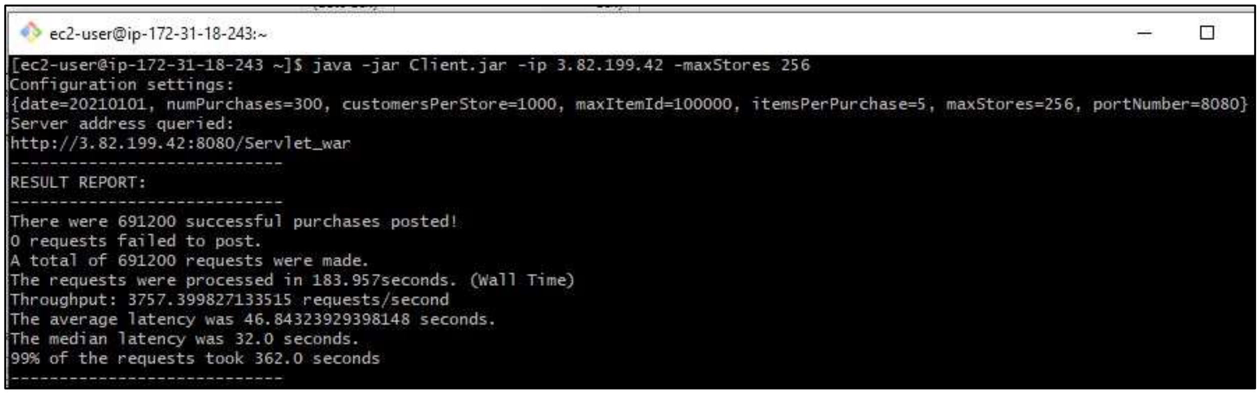
128 Threads:



ec2-user@ip-172-31-18-243:~

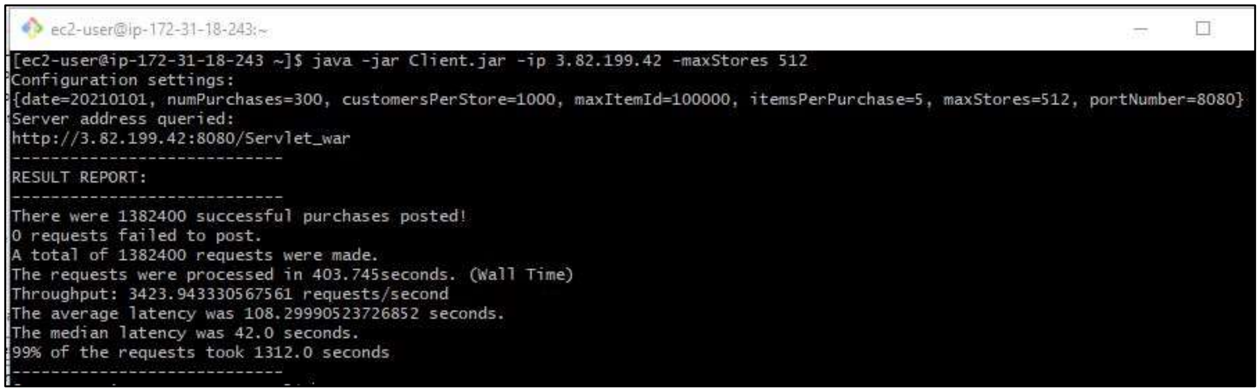
```
[ec2-user@ip-172-31-18-243 ~]$ java -jar Client.jar -ip 3.82.199.42 -maxStores 128
Configuration settings:
{date=20210101, numPurchases=300, customersPerStore=1000, maxItemId=100000, itemsPerPurchase=5, maxStores=128, portNumber=8080}
Server address queried:
http://3.82.199.42:8080/Servlet_war
----------------------------
RESULT REPORT:
----------------------------
There were 345600 successful purchases posted!
0 requests failed to post.
A total of 345600 requests were made.
The requests were processed in 87.786seconds. (Wall Time)
Throughput: 3936.8464219807256 requests/second
The average latency was 21.66799189814815 seconds.
The median latency was 22.0 seconds.
99% of the requests took 167.0 seconds
----------------------------
```

256 Threads:



```
ec2-user@ip-172-31-18-243:~
[ec2-user@ip-172-31-18-243 ~]$ java -jar Client.jar -ip 3.82.199.42 -maxStores 256
Configuration settings:
{date=20210101, numPurchases=300, customersPerStore=1000, maxItemId=100000, itemsPerPurchase=5, maxStores=256, portNumber=8080}
Server address queried:
http://3.82.199.42:8080/Servlet_war
----------------------------
RESULT REPORT:
----------------------------
There were 691200 successful purchases posted!
0 requests failed to post.
A total of 691200 requests were made.
The requests were processed in 183.957seconds. (Wall Time)
Throughput: 3757.399827133515 requests/second
The average latency was 46.84323929398148 seconds.
The median latency was 32.0 seconds.
99% of the requests took 362.0 seconds
----------------------------
```

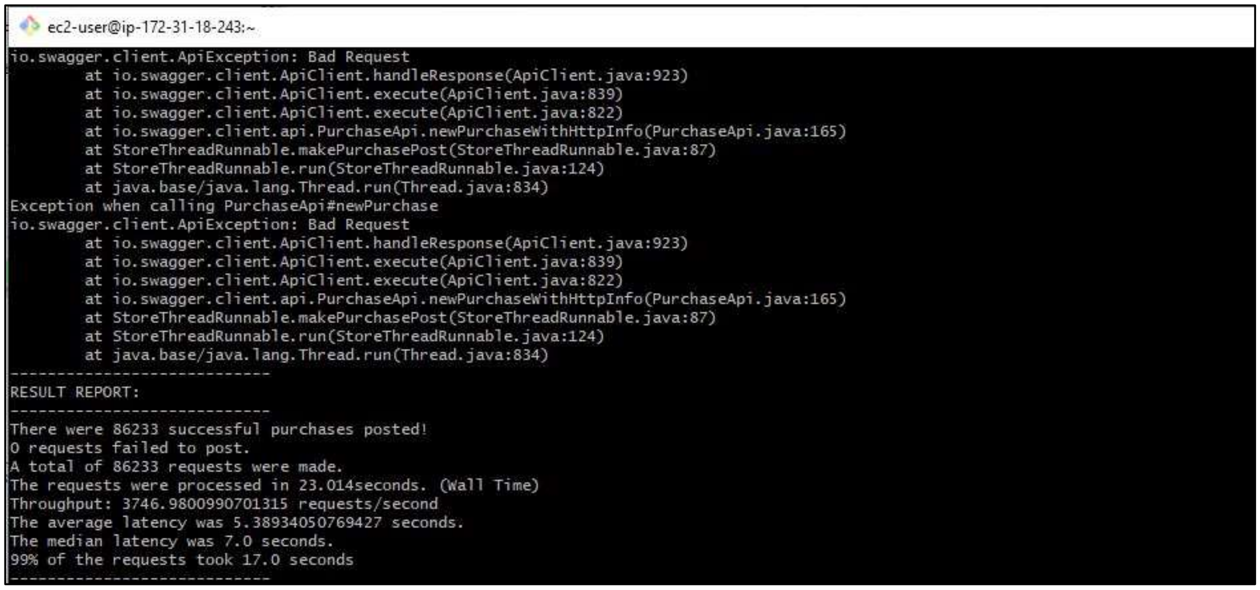
512 Threads:



```
ec2-user@ip-172-31-18-243:~
[ec2-user@ip-172-31-18-243 ~]$ java -jar Client.jar -ip 3.82.199.42 -maxStores 512
Configuration settings:
{date=20210101, numPurchases=300, customersPerStore=1000, maxItemId=100000, itemsPerPurchase=5, maxStores=512, portNumber=8080}
Server address queried:
http://3.82.199.42:8080/Servlet_war
----------------------------
RESULT REPORT:
----------------------------
There were 1382400 successful purchases posted!
0 requests failed to post.
A total of 1382400 requests were made.
The requests were processed in 403.745seconds. (Wall Time)
Throughput: 3423.943330567561 requests/second
The average latency was 108.29990523726852 seconds.
The median latency was 42.0 seconds.
99% of the requests took 1312.0 seconds
----------------------------
```

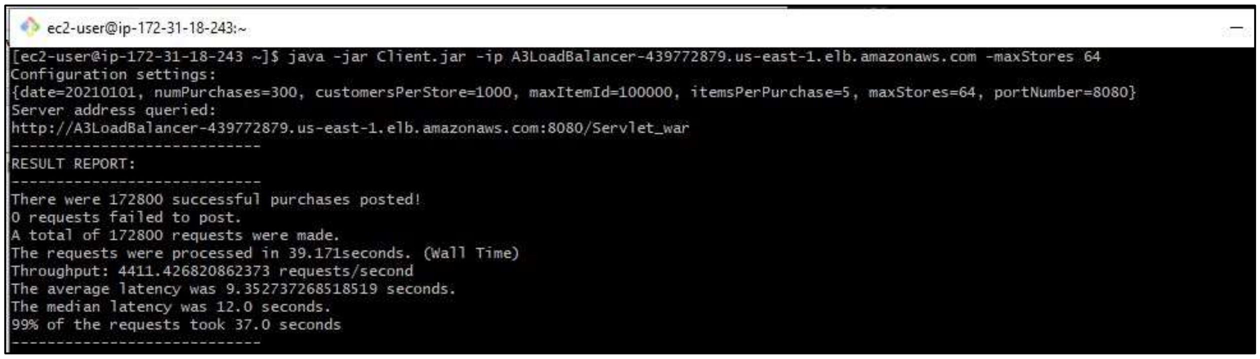
Scaled System Data:

32 Threads



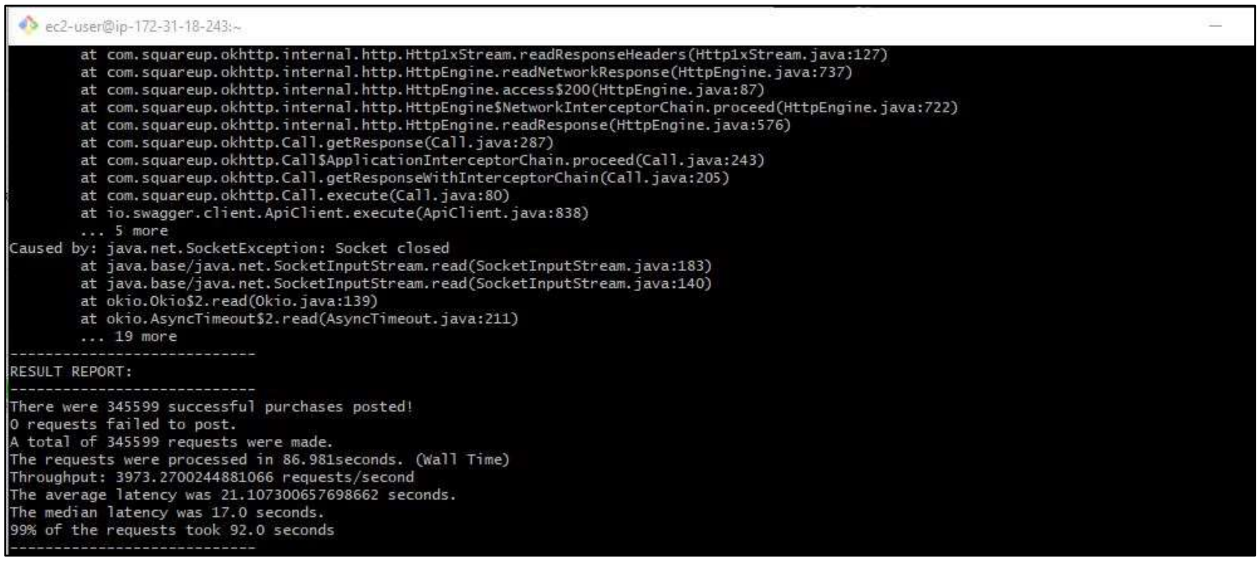
```
ec2-user@ip-172-31-18-243:~
io.swagger.client.ApiException: Bad Request
        at io.swagger.client.ApiClient.handleResponse(ApiClient.java:923)
        at io.swagger.client.ApiClient.execute(ApiClient.java:839)
        at io.swagger.client.ApiClient.execute(ApiClient.java:822)
        at io.swagger.client.api.PurchaseApi.newPurchaseWithHttpInfo(PurchaseApi.java:165)
        at StoreThreadRunnable.makePurchasePost(StoreThreadRunnable.java:87)
        at StoreThreadRunnable.run(StoreThreadRunnable.java:124)
        at java.base/java.lang.Thread.run(Thread.java:834)
Exception when calling PurchaseApi#newPurchase
io.swagger.client.ApiException: Bad Request
        at io.swagger.client.ApiClient.handleResponse(ApiClient.java:923)
        at io.swagger.client.ApiClient.execute(ApiClient.java:839)
        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)
----------------------------
RESULT REPORT:
----------------------------
There were 86233 successful purchases posted!
0 requests failed to post.
A total of 86233 requests were made.
The requests were processed in 23.014seconds. (Wall Time)
Throughput: 3746.9800990701315 requests/second
The average latency was 5.38934050769427 seconds.
The median latency was 7.0 seconds.
99% of the requests took 17.0 seconds
----------------------------
```

64 Threads



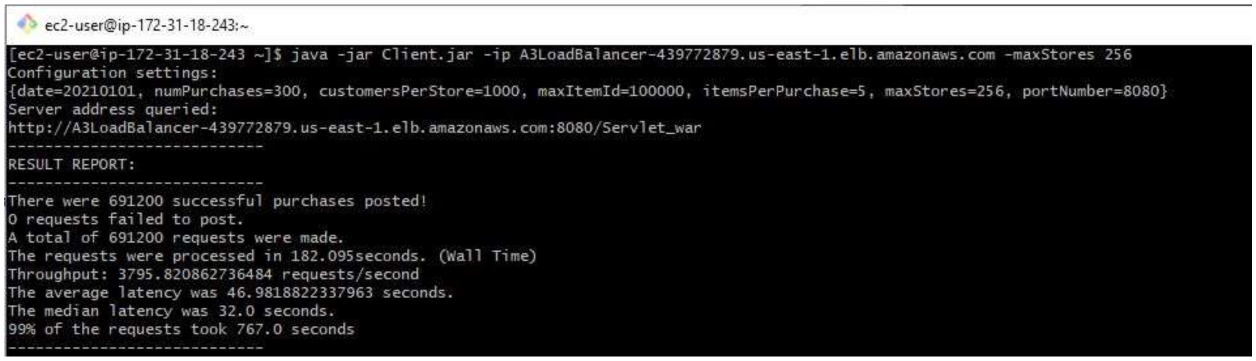
```
ec2-user@ip-172-31-18-243:~
[ec2-user@ip-172-31-18-243 ~]$ java -jar Client.jar -ip A3LoadBalancer-439772879.us-east-1.elb.amazonaws.com -maxStores 64
Configuration settings:
{date=20210101, numPurchases=300, customersPerStore=1000, maxItemId=100000, itemsPerPurchase=5, maxStores=64, portNumber=8080}
Server address queried:
http://A3LoadBalancer-439772879.us-east-1.elb.amazonaws.com:8080/Servlet_war
----------------------------
RESULT REPORT:
----------------------------
There were 172800 successful purchases posted!
0 requests failed to post.
A total of 172800 requests were made.
The requests were processed in 39.171seconds. (Wall Time)
Throughput: 4411.426820862373 requests/second
The average latency was 9.352737268518519 seconds.
The median latency was 12.0 seconds.
99% of the requests took 37.0 seconds
----------------------------
```

128 Threads



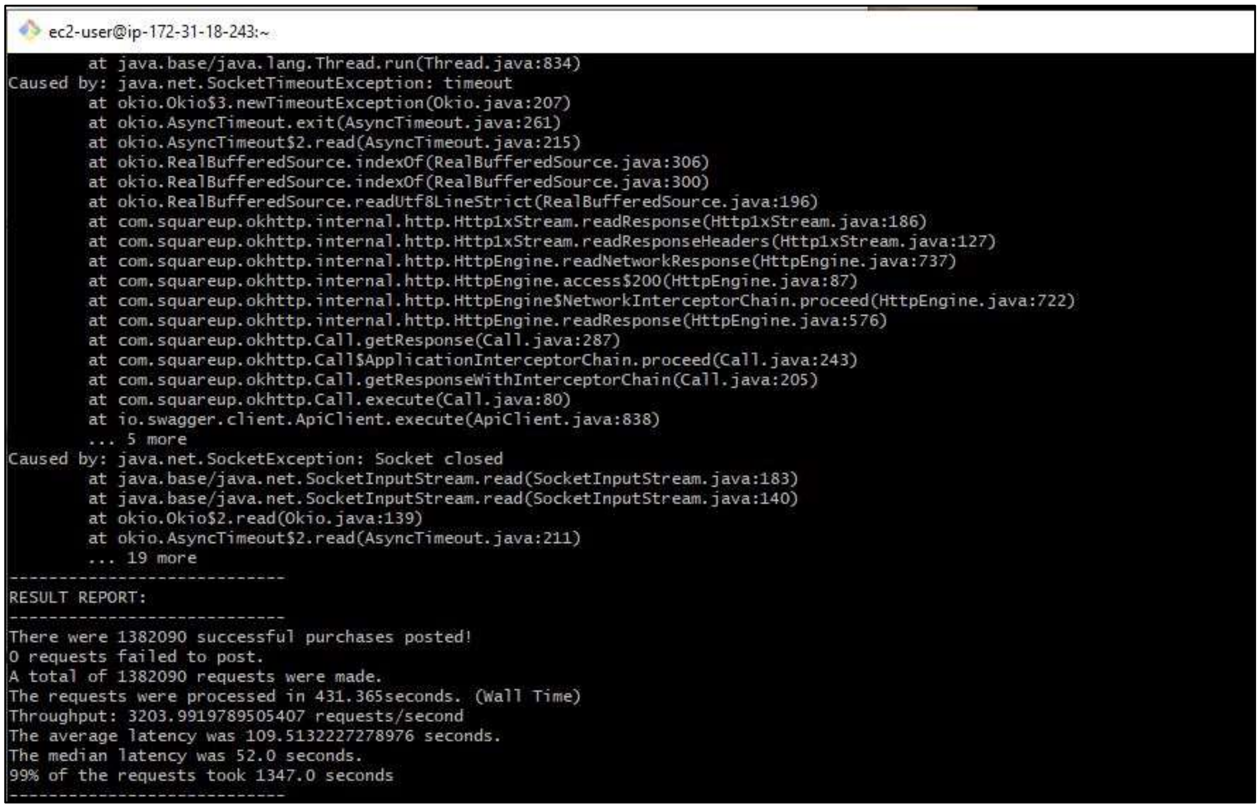
```
ec2-user@ip-172-31-18-243:~
        at com.squareup.okhttp.internal.http.Http1xStream.readResponseHeaders(Http1xStream.java:127)
        at com.squareup.okhttp.internal.http.HttpEngine.readNetworkResponse(HttpEngine.java:737)
        at com.squareup.okhttp.internal.http.HttpEngine.access$200(HttpEngine.java:87)
        at com.squareup.okhttp.internal.http.HttpEngine$NetworkInterceptorChain.proceed(HttpEngine.java:722)
        at com.squareup.okhttp.internal.http.HttpEngine.readResponse(HttpEngine.java:576)
        at com.squareup.okhttp.Call.getResponse(Call.java:287)
        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
Caused by: java.net.SocketException: Socket closed
        at java.base/java.net.SocketInputStream.read(SocketInputStream.java:183)
        at java.base/java.net.SocketInputStream.read(SocketInputStream.java:140)
        at okio.Okio$2.read(Okio.java:139)
        at okio.AsyncTimeout$2.read(AsyncTimeout.java:211)
        ... 19 more
----------------------------
RESULT REPORT:
----------------------------
There were 345599 successful purchases posted!
0 requests failed to post.
A total of 345599 requests were made.
The requests were processed in 86.981seconds. (Wall Time)
Throughput: 3973.2700244881066 requests/second
The average latency was 21.107300657698662 seconds.
The median latency was 17.0 seconds.
99% of the requests took 92.0 seconds
----------------------------
```

256 Threads



```
ec2-user@ip-172-31-18-243:~
[ec2-user@ip-172-31-18-243 ~]$ java -jar Client.jar -ip A3LoadBalancer-439772879.us-east-1.elb.amazonaws.com -maxStores 256
Configuration settings:
{date=20210101, numPurchases=300, customersPerStore=1000, maxItemId=100000, itemsPerPurchase=5, maxStores=256, portNumber=8080}
Server address queried:
http://A3LoadBalancer-439772879.us-east-1.elb.amazonaws.com:8080/Servlet_war
----------------------------
RESULT REPORT:
----------------------------
There were 691200 successful purchases posted!
0 requests failed to post.
A total of 691200 requests were made.
The requests were processed in 182.095seconds. (Wall Time)
Throughput: 3795.820862736484 requests/second
The average latency was 46.9818822337963 seconds.
The median latency was 32.0 seconds.
99% of the requests took 767.0 seconds
----------------------------
```

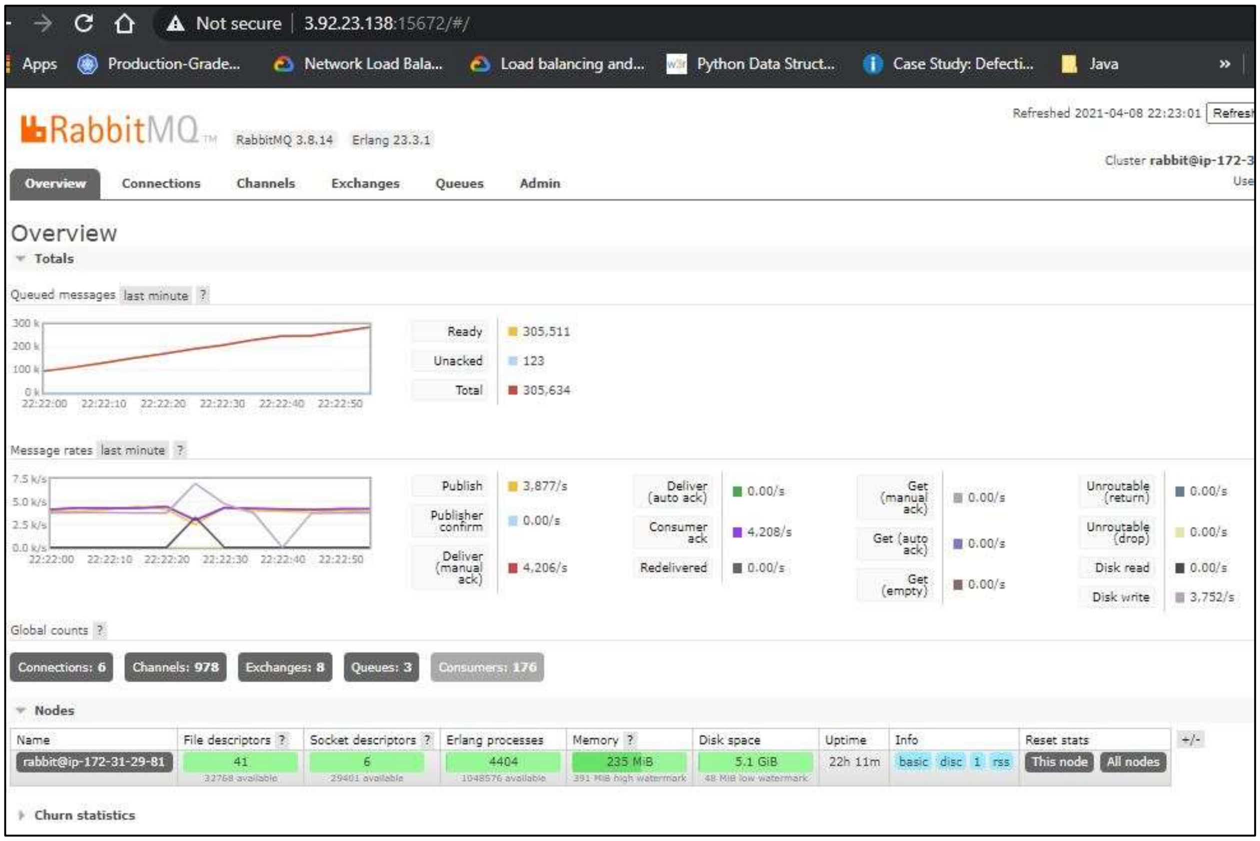
512 Threads



ec2-user@ip-172-31-18-243:~

```
        at java.base/java.lang.Thread.run(Thread.java:834)
Caused by: java.net.SocketTimeoutException: timeout
        at okio.Okio$3.newTimeoutException(Okio.java:207)
        at okio.AsyncTimeout.exit(AsyncTimeout.java:261)
        at okio.AsyncTimeout$2.read(AsyncTimeout.java:215)
        at okio.RealBufferedSource.indexOf(RealBufferedSource.java:306)
        at okio.RealBufferedSource.indexOf(RealBufferedSource.java:300)
        at okio.RealBufferedSource.readUtf8LineStrict(RealBufferedSource.java:196)
        at com.squareup.okhttp.internal.http.Http1xStream.readResponse(Http1xStream.java:186)
        at com.squareup.okhttp.internal.http.Http1xStream.readResponseHeaders(Http1xStream.java:127)
        at com.squareup.okhttp.internal.http.HttpEngine.readNetworkResponse(HttpEngine.java:737)
        at com.squareup.okhttp.internal.http.HttpEngine.access$200(HttpEngine.java:87)
        at com.squareup.okhttp.internal.http.HttpEngine$NetworkInterceptorChain.proceed(HttpEngine.java:722)
        at com.squareup.okhttp.internal.http.HttpEngine.readResponse(HttpEngine.java:576)
        at com.squareup.okhttp.Call.getResponse(Call.java:287)
        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
Caused by: java.net.SocketException: Socket closed
        at java.base/java.net.SocketInputStream.read(SocketInputStream.java:183)
        at java.base/java.net.SocketInputStream.read(SocketInputStream.java:140)
        at okio.Okio$2.read(Okio.java:139)
        at okio.AsyncTimeout$2.read(AsyncTimeout.java:211)
        ... 19 more
----------------------------
RESULT REPORT:
----------------------------
There were 1382090 successful purchases posted!
0 requests failed to post.
A total of 1382090 requests were made.
The requests were processed in 431.365seconds. (Wall Time)
Throughput: 3203.9919789505407 requests/second
The average latency was 109.5132227278976 seconds.
The median latency was 52.0 seconds.
99% of the requests took 1347.0 seconds
----------------------------
```

RabbitMQ in Operation:



Overview

Queued messages last minute

Ready	305,511
Unacked	123
Total	305,634

Message rates last minute

Publish	3,877/s	Deliver (auto ack)	0.00/s	Get (manual ack)	0.00/s	Unroutable (return)	0.00/s
Publisher confirm	0.00/s	Consumer ack	4,208/s	Get (auto ack)	0.00/s	Unroutable (drop)	0.00/s
Deliver (manual ack)	4,206/s	Redelivered	0.00/s	Get (empty)	0.00/s	Disk read	0.00/s
						Disk write	3,752/s

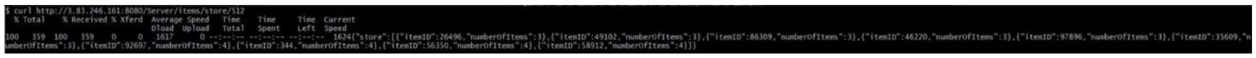
Global counts

Connections: 6 Channels: 978 Exchanges: 8 Queues: 3 Consumers: 176

Name	File descriptors	Socket descriptors	Erlang processes	Memory	Disk space	Uptime	Info	Reset stats
rabbit@ip-172-31-29-81	41 32768 available	6 29401 available	4404 1048576 available	235 MiB 391 MiB high watermark	5.1 GiB 48 MiB low watermark	22h 11m	basic disc 1 rss	This node All nodes

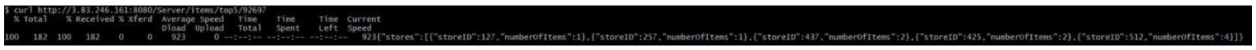
cURL Requests:

1.



```
$ curl http://3.83.246.161:8080/Server/items/store/512
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100   359  100   359    0     0   1617      0 --:--:-- --:--:-- --:--:--  1624{"store":[{"itemID":26496,"numberOfItems":3},{"itemID":49102,"numberOfItems":3},{"itemID":86309,"numberOfItems":3},{"itemID":46220,"numberOfItems":3},{"itemID":97896,"numberOfItems":3},{"itemID":35609,"n
umberOfItems":3},{"itemID":92697,"numberOfItems":4},{"itemID":344,"numberOfItems":4},{"itemID":56350,"numberOfItems":4},{"itemID":58912,"numberOfItems":4}]}
```

2.



```
$ curl http://3.83.246.161:8080/Server/items/top5/92697
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100   182  100   182    0     0    923      0 --:--:-- --:--:-- --:--:--   923{"stores":[{"storeID":127,"numberOfItems":1},{"storeID":257,"numberOfItems":1},{"storeID":437,"numberOfItems":2},{"storeID":425,"numberOfItems":2},{"storeID":512,"numberOfItems":4}]}
```