Building Scalable Distributed Systems

Assignment 3

Git Repository: https://github.com/akksshah/building-scalable-distributed-systems

Server Design Description

When the server is deployed, it registers an exchange on the RabbitMQ broker called “purchase_exchange”, it then pre-creates a channel pool for publishing message to the exchange.
The server receives the request from the client in the form of Json. It converts the Json string to the PurchaseOrder object that, the client requests the server to persist in the MySQL database. The conversion of json string to PurchaseOrder object is done using Jackson library.
The server on receiving a purchaseOrder from the client publishes the sames to the RabbitMQ exchange ie the “purchase_exchange”. Upon successfully publishing the message the server returns an HTTP.SC_OK status, otherwise it returns an HTTP_SC_NOT_OK status.

Broker:
RabbitMq broker is hosted on another free tier ec2-instance.
It is attached to an elastic ip, which enables us to avoid frequent ip changes in the server for the configuration

Consumer:
StorePurchase consumer:
The consumer makes a single connection to the broker. It then instantiates 60 consumer threads that read message published to the purchases queue which is bind to the purchase_exchange. Each of the thread has a connection to the Mysql database running on AWS RDS free-tier instance. As a consumer thread reads messages from the exchange, it makes an insert to the database for persistence.

Store MicroService:
Similar to the consumer above, it makes a single connection to the broker, It then instantiates 1024 consumer threads that would read messages from the queue “storeInventory” which is bound to the exchange purchase_exchange. It has 2 global hashMaps for quick lookups.
The first hashmap stores each entry in the following manner {StoreID: {itemID: numberOfItems sold}, … }
The second hashMap stores data in the following manner {itemID: {storeID: numberOfItemsSoldInThat Store}
These maps then make the look ups extremely fast and we can then provide them to the client asap

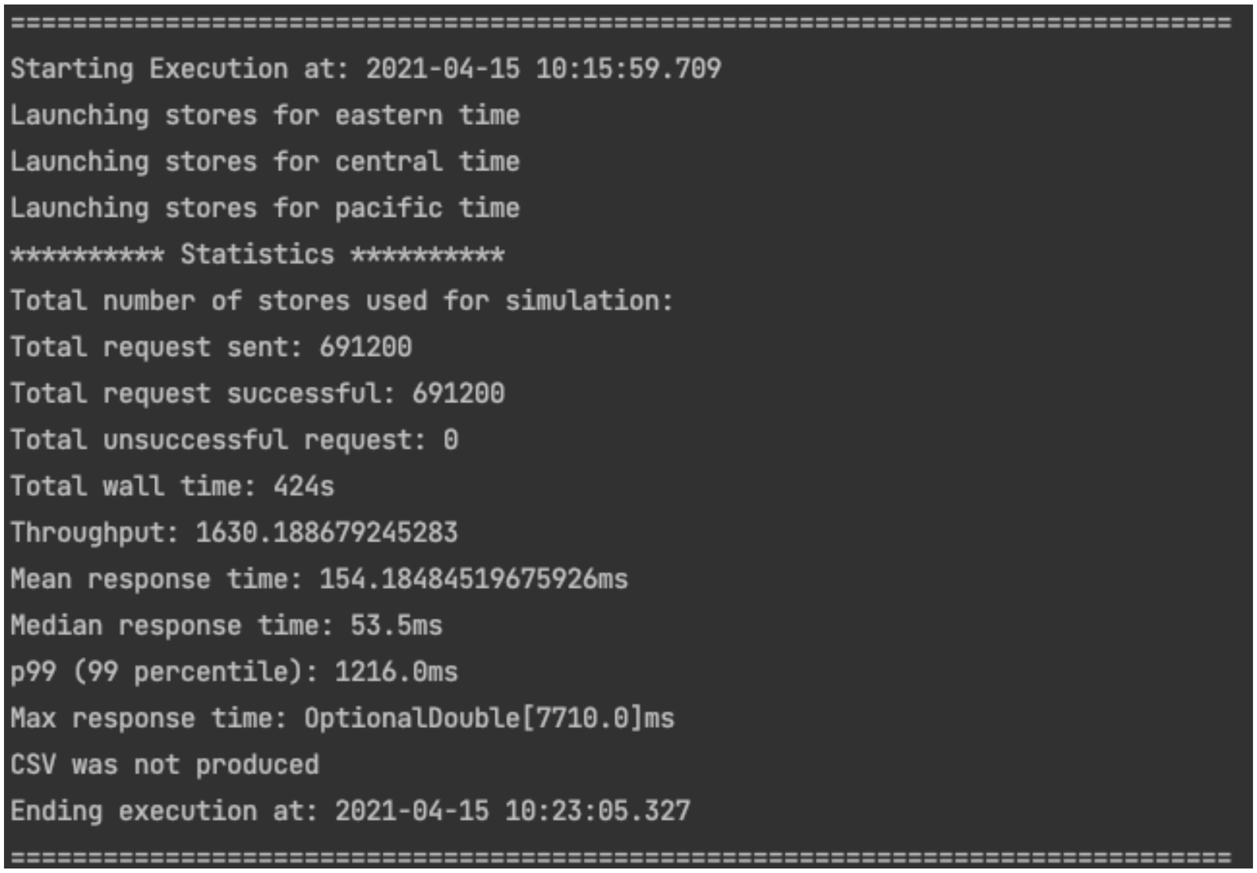
Database design: Unchanged

Single Server Test:

Persistent V/S Non-Persistent Queues

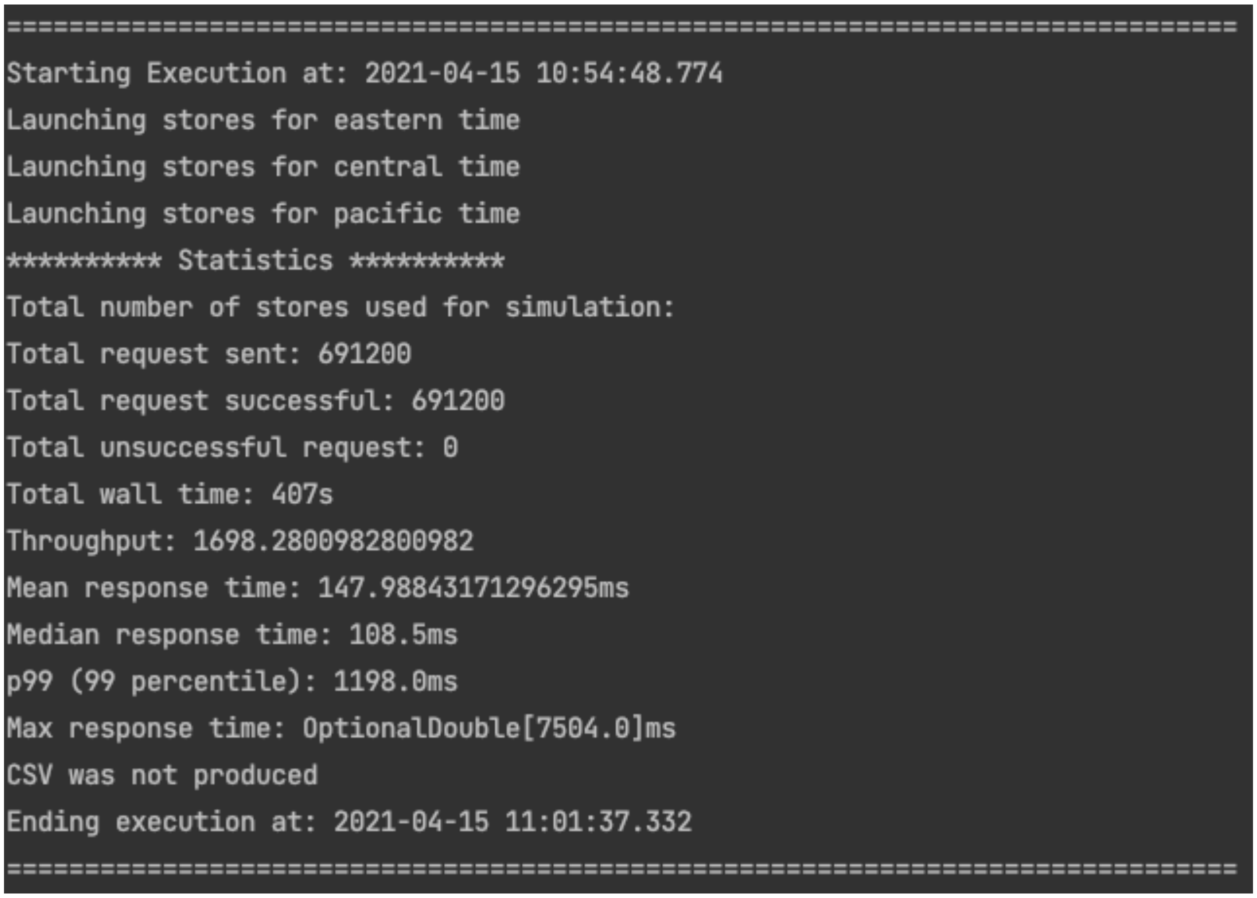
256 Threads

Persistent Queue



```
================================================================================
Starting Execution at: 2021-04-15 10:15:59.709
Launching stores for eastern time
Launching stores for central time
Launching stores for pacific time
********** Statistics **********
Total number of stores used for simulation:
Total request sent: 691200
Total request successful: 691200
Total unsuccessful request: 0
Total wall time: 424s
Throughput: 1630.188679245283
Mean response time: 154.18484519675926ms
Median response time: 53.5ms
p99 (99 percentile): 1216.0ms
Max response time: OptionalDouble[7710.0]ms
CSV was not produced
Ending execution at: 2021-04-15 10:23:05.327
================================================================================
```

Non-Persistent Queue

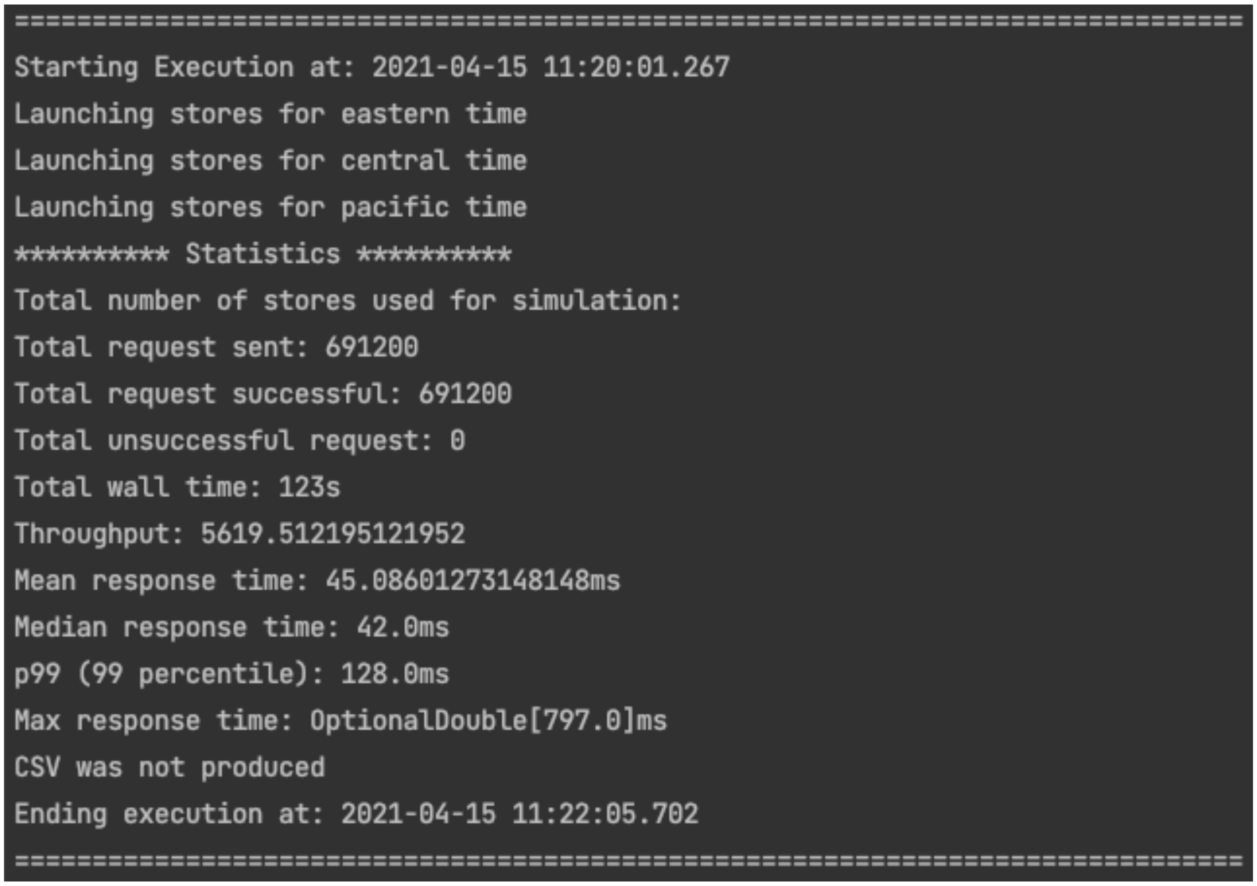


```
=================================================================================
Starting Execution at: 2021-04-15 10:54:48.774
Launching stores for eastern time
Launching stores for central time
Launching stores for pacific time
********** Statistics **********
Total number of stores used for simulation:
Total request sent: 691200
Total request successful: 691200
Total unsuccessful request: 0
Total wall time: 407s
Throughput: 1698.2800982800982
Mean response time: 147.98843171296295ms
Median response time: 108.5ms
p99 (99 percentile): 1198.0ms
Max response time: OptionalDouble[7504.0]ms
CSV was not produced
Ending execution at: 2021-04-15 11:01:37.332
=================================================================================
```

We can see that the throughput of persistent queues is slightly less than that of the non-persistent queues. This behavior is something that we definitely would expect. Given that the system has very minimal difference in throughput when going with a persistent queue, we should prefer persistent queues so that they can survive failures both when the broker(RabbitMQ) or the consumer is down

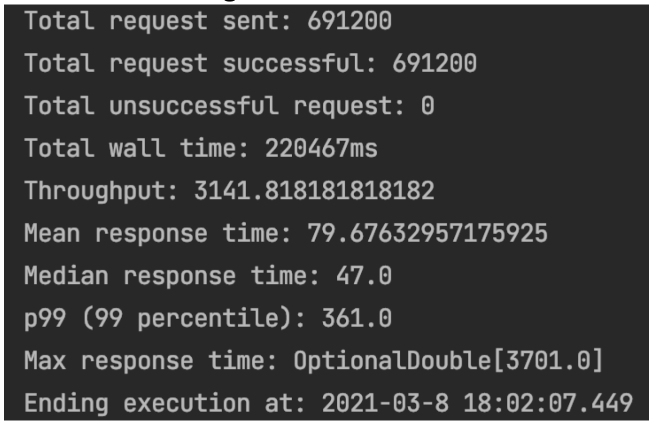
Load Balanced Server Test

256 Threads with RabbitMQ broker



```
================================================================================
Starting Execution at: 2021-04-15 11:20:01.267
Launching stores for eastern time
Launching stores for central time
Launching stores for pacific time
********** Statistics **********
Total number of stores used for simulation:
Total request sent: 691200
Total request successful: 691200
Total unsuccessful request: 0
Total wall time: 123s
Throughput: 5619.512195121952
Mean response time: 45.08601273148148ms
Median response time: 42.0ms
p99 (99 percentile): 128.0ms
Max response time: OptionalDouble[797.0]ms
CSV was not produced
Ending execution at: 2021-04-15 11:22:05.702
================================================================================
```

256 Threads Assignment 2

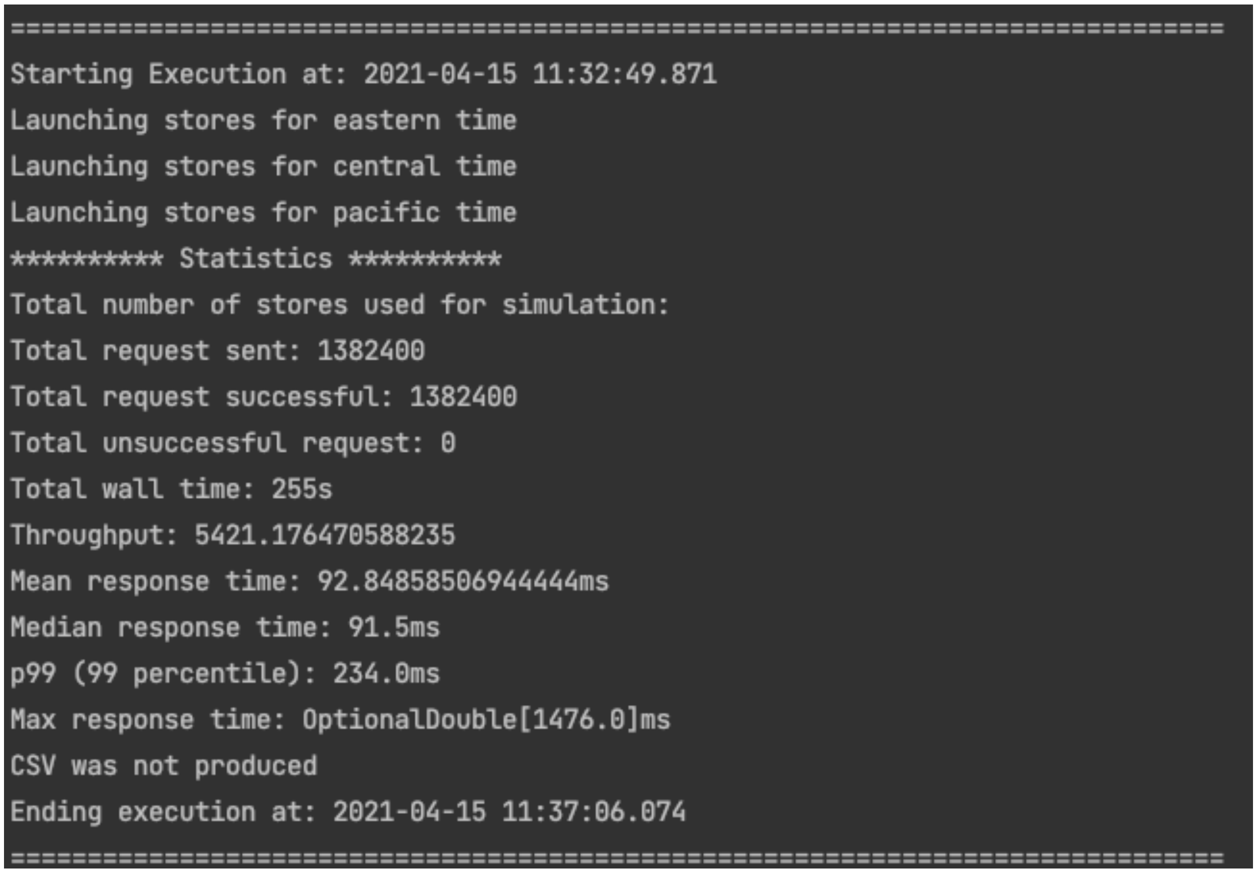


```
Total request sent: 691200
Total request successful: 691200
Total unsuccessful request: 0
Total wall time: 220467ms
Throughput: 3141.818181818182
Mean response time: 79.67632957175925
Median response time: 47.0
p99 (99 percentile): 361.0
Max response time: OptionalDouble[3701.0]
Ending execution at: 2021-03-8 18:02:07.449
```

In the comparision of the throughput and the response time with the previous assignments, we can see that there has been a significant increase in throughput when we use a messaging broker like RabbitMQ to make the database writes asynchronous and there is a huge improvement in the overall wall time, mean response time and the p99 time of the requests.

Espescially we can see that 99% of our requests are served in one third of the time that they used to take in assignment 2. Which I feel is a lot of improvement.

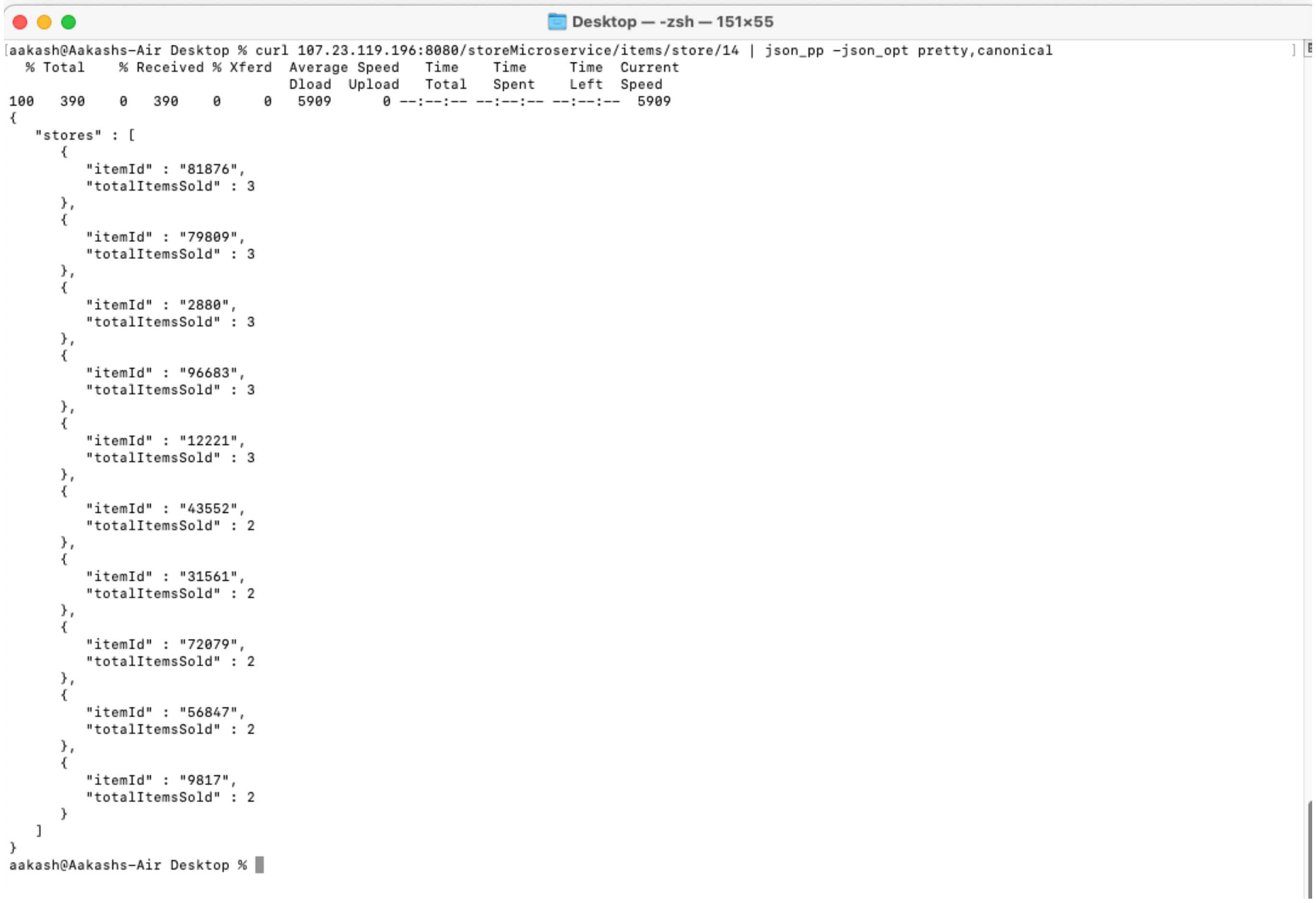
Run With 512 clients



```
================================================================================
Starting Execution at: 2021-04-15 11:32:49.871
Launching stores for eastern time
Launching stores for central time
Launching stores for pacific time
********** Statistics **********
Total number of stores used for simulation:
Total request sent: 1382400
Total request successful: 1382400
Total unsuccessful request: 0
Total wall time: 255s
Throughput: 5421.176470588235
Mean response time: 92.84858506944444ms
Median response time: 91.5ms
p99 (99 percentile): 234.0ms
Max response time: OptionalDouble[1476.0]ms
CSV was not produced
Ending execution at: 2021-04-15 11:37:06.074
================================================================================
```

Store Microservice

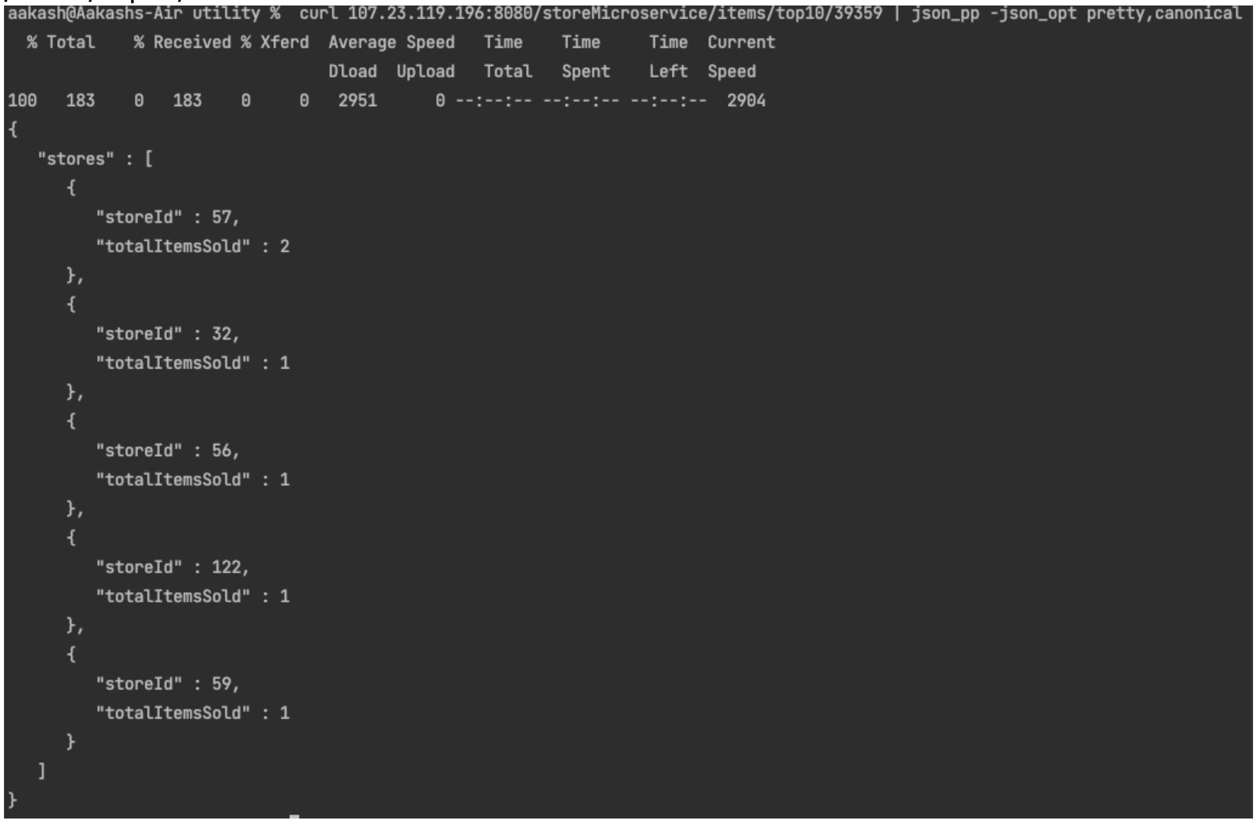
/items/store/:storeID



Desktop — -zsh — 151×55

```
aakash@Aakashs-Air Desktop % curl 107.23.119.196:8080/storeMicroservice/items/store/14 | json_pp -json_opt pretty,canonical
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100   390    0   390    0     0   5909      0 --:--:-- --:--:-- --:--:--  5909
{
   "stores" : [
      {
         "itemId" : "81876",
         "totalItemsSold" : 3
      },
      {
         "itemId" : "79809",
         "totalItemsSold" : 3
      },
      {
         "itemId" : "2880",
         "totalItemsSold" : 3
      },
      {
         "itemId" : "96683",
         "totalItemsSold" : 3
      },
      {
         "itemId" : "12221",
         "totalItemsSold" : 3
      },
      {
         "itemId" : "43552",
         "totalItemsSold" : 2
      },
      {
         "itemId" : "31561",
         "totalItemsSold" : 2
      },
      {
         "itemId" : "72079",
         "totalItemsSold" : 2
      },
      {
         "itemId" : "56847",
         "totalItemsSold" : 2
      },
      {
         "itemId" : "9817",
         "totalItemsSold" : 2
      }
   ]
}
aakash@Aakashs-Air Desktop % 
```

/items/top10/:itemID



```
aakash@Aakashs-Air utility %  curl 107.23.119.196:8080/storeMicroservice/items/top10/39359 | json_pp -json_opt pretty,canonical
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100   183    0   183    0     0   2951      0 --:--:-- --:--:-- --:--:--  2904
{
   "stores" : [
      {
         "storeId" : 57,
         "totalItemsSold" : 2
      },
      {
         "storeId" : 32,
         "totalItemsSold" : 1
      },
      {
         "storeId" : 56,
         "totalItemsSold" : 1
      },
      {
         "storeId" : 122,
         "totalItemsSold" : 1
      },
      {
         "storeId" : 59,
         "totalItemsSold" : 1
      }
   ]
}
```

1. Do I need load balancing? Or can my system work with 1 free-tier (or slightly upgraded) server?
   As we can see from the test run, having a load balancer with 4 ec2 instances definitely improves the throughput for the client load. Infact it provides almost 3.4x of the throughput.

2. How many consumers nodes do I need?
   I essentially needed just 1 ec2 instance to keep the transient queued message on RabbitMQ broker to be zero. Although I had 60 consumer threads pulling out data out of the broker continuously.

3. What messaging system design should I use?
   I used a pub-sub system where the server would publish a message to the exchange. The exchange was subscribed by 2 consumers. This allowed the message from the server to be transmitted to both the consumers for processing.