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

Text

Description automatically generated

Non-Persistent QueueText

Description automatically generated

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 Text

Description automatically generated

256 Threads Assignment 2

Text

Description automatically generated

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

Text

Description automatically generated

### Store Microservice

/items/store/:storeID

Graphical user interface, application

Description automatically generated

/items/top10/:itemID

Text

Description automatically generated

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.

1. 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.

1. 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.