Name: Nachiket Dani

Assignment-4 Report

CS6650 : Distributed Systems

The following describes the assignment that advances further on top of the client, database service and store microservice developed within assignment3. In this assignment I did a couple of changes:

- **Implement Apache Kafka:**
  Transform the message streaming that delivers purchase data from the Servlet from RabbitMQ to Apache Kafka. We retain the RabbitMQ RPC styled architecture for the Purchase Store Microservice and keep the simple fan-out that delivers purchase data to the Purchase Store. The following depicts the container system architecture for the current system. Client still simulates purchase post requests through multiple threads to a scaled (4) server setup with a load balancer within AWS environment.
- **Implement a Distributed database (sharded database):**
  From the single instance of the MySQL RDS database, I implemented in a3, I now create a total of 3 separate instances and distribute the purchase data that is received from the servlet based on a hash function. The database schema remains the same since we only scale out the database to handle lot more data. This coupled with the use of Apache Kafka enable us to implement a stronger streaming queue that appears better than the use of RabbitMQ from a3.

Git links to code base: (Contains separate folders for Client, Servlets, and services).

A4: *https://github.com/NachiketDani/CS6650_DistributedSystems/tree/main/CS6650_a4*

< Refer next page for the system container and overall architecture>

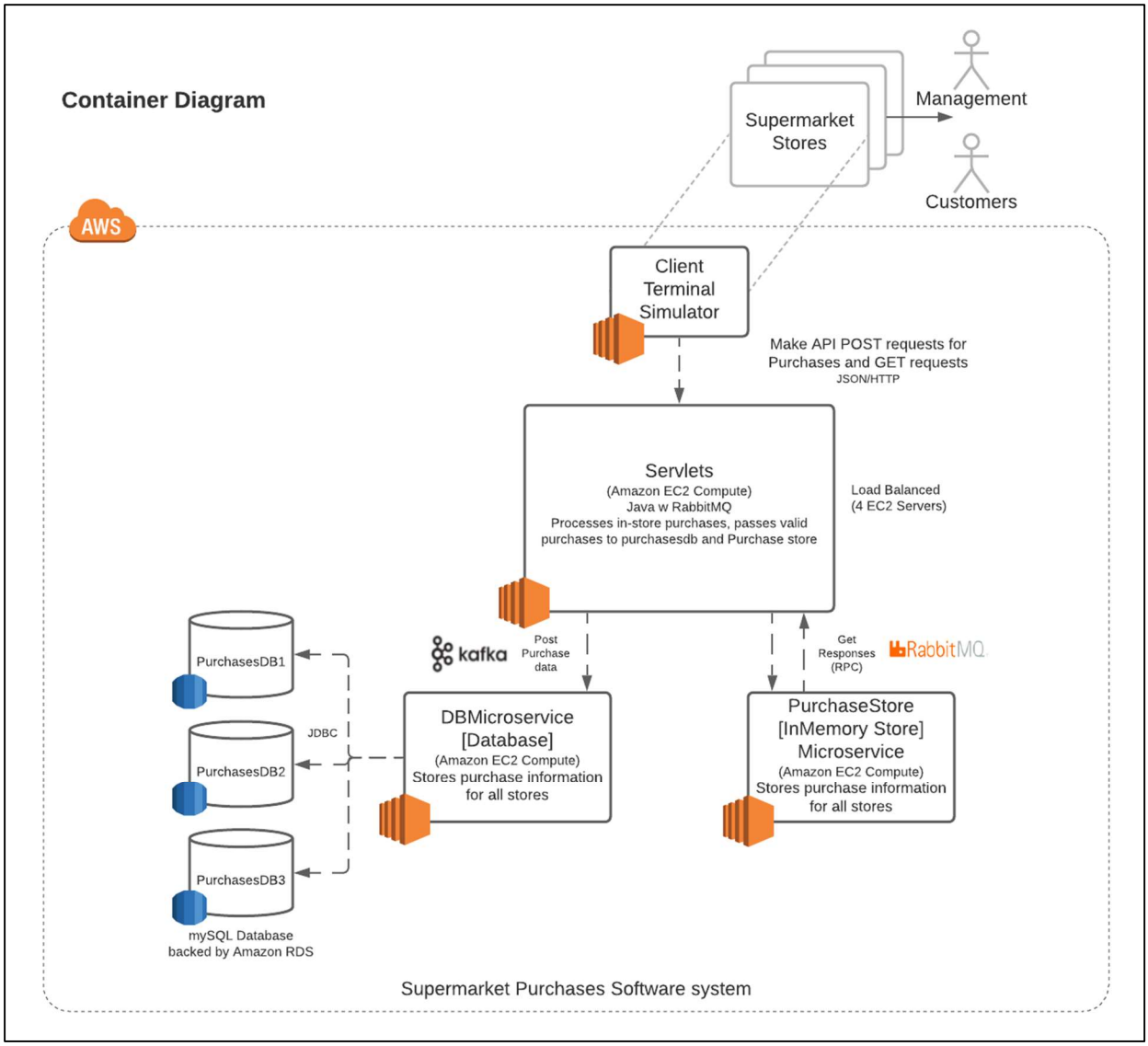


fig: Final System Architecture: Container diagram

Database Service:

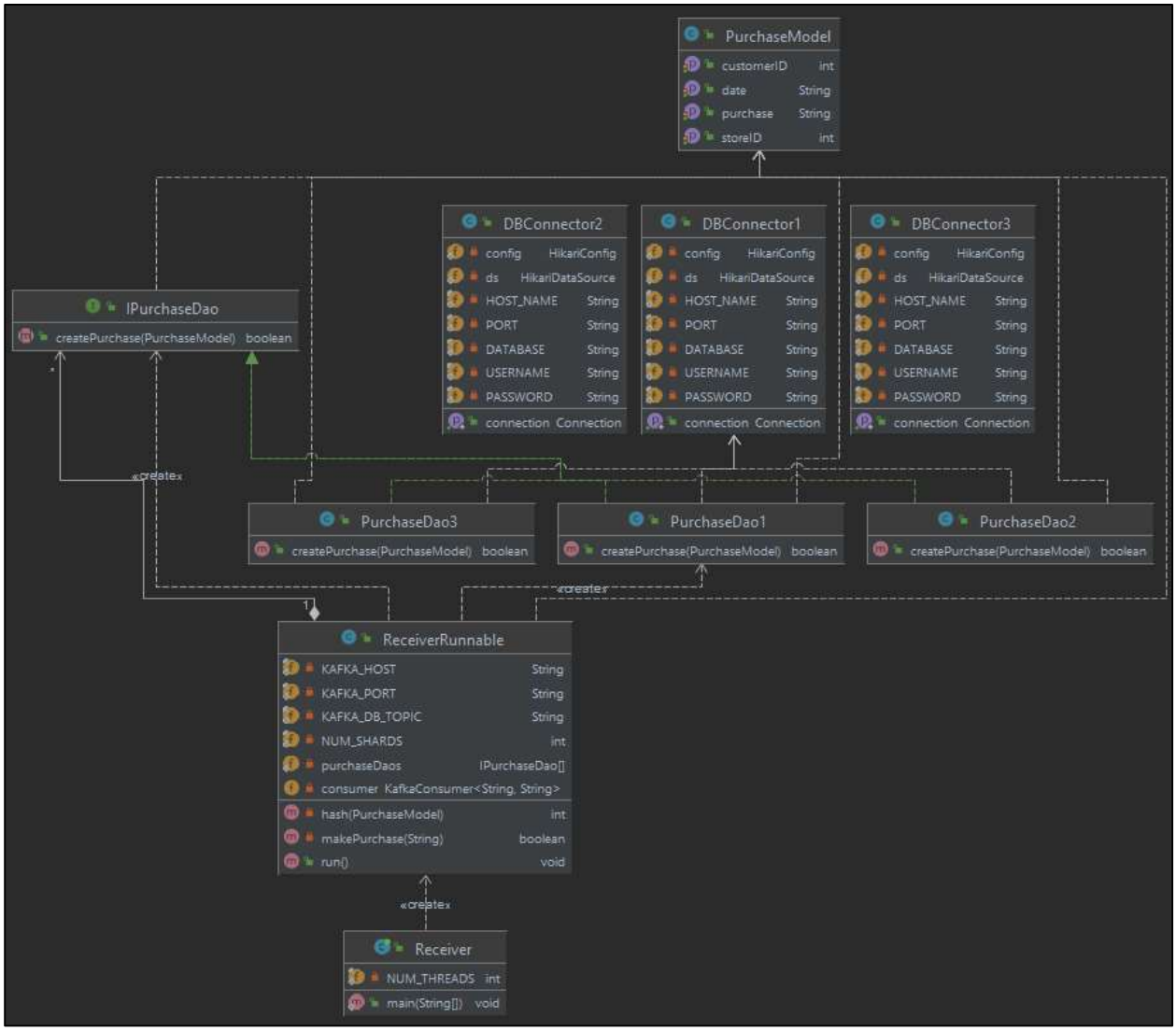


fig: Database service UML diagram

A database stores purchase data from the client made POST requests. I use the Amazon RDS storage via a local mySQL Workbench instance. I 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. Since I have sharded the database into 3 separate instances, I use 3 instances of the PurchaseDao (1, 2 and 3) along with 3 different DBConnectors. The PurchaseModel created reflects the database schema across the instances and remains the same. 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 used for our distributed database. 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.

Purchase Store Microservice:

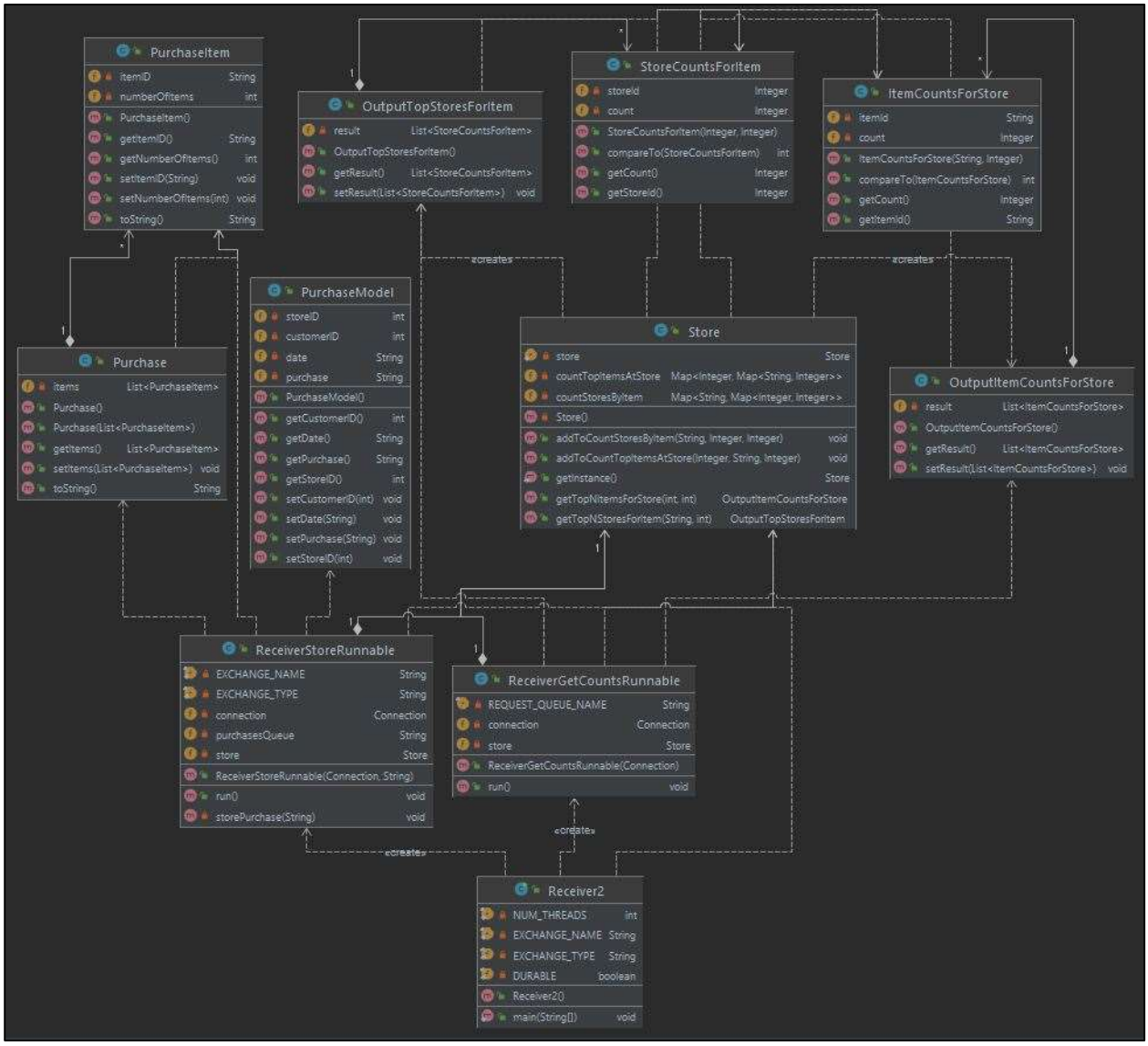


fig: PurchaseStore Microservice UML Diagram

The PurchaseStore Microservice implements the receiver main method for RabbitMQ (Receiver2) through 2 separate Runnable thread mechanisms and remains the same as I did not make any changes since a3:

- 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

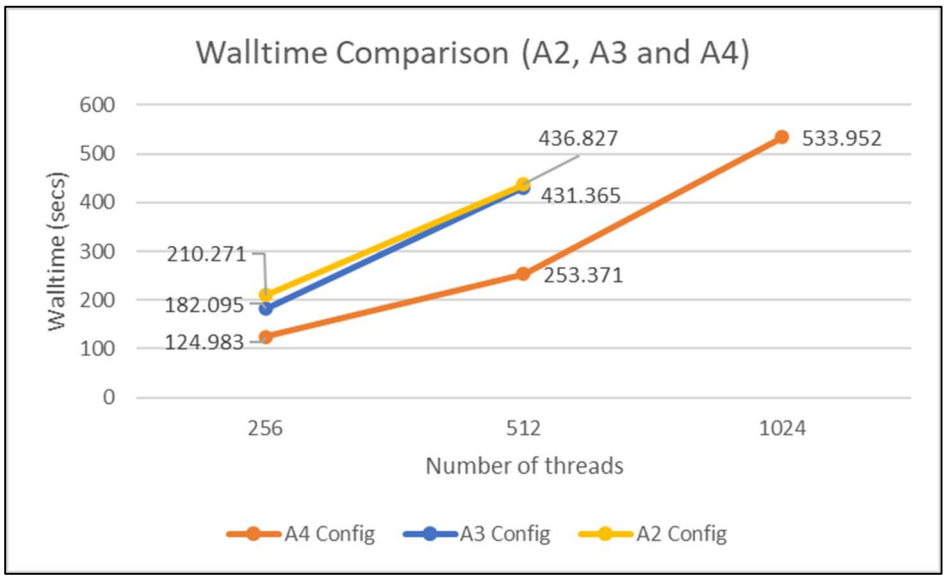
range they need based on the GET request. The heap then is popped and feeds into the result as needed. Refer to Store.java.

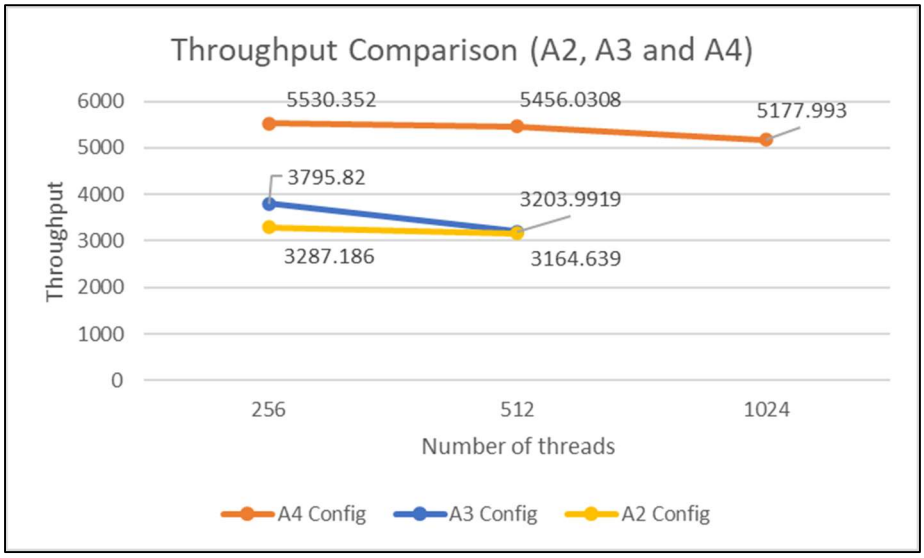
The PurchaseModel, Purchase, PurchaseItem classes serve as data models for the database model, purchase data and purchased item formats. Classes ItemCountsForStore and StoreCountsForItem serve to create Pairs for counts of items and sales at stores and item, store ids. This is essential to implement 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 load balanced server (set of 4 EC2 instances grouped with a load balancer) as well as the Scaled system and write records into the database for 256, 512 and 1024 threads (maxStores). I see that the Kafka version I have worker on in a4 is the best version so far if I compare the wall time and throughputs for older versions. Below section depicts the wall time and throughput comparisons as achieved across the 3 assignments.

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 an application load balancer to coordinate usage between the replicas. I also provisioned additional Ubuntu instances for my Kafka and RabbitMQ services *(Images now disabled).*





Servlet 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

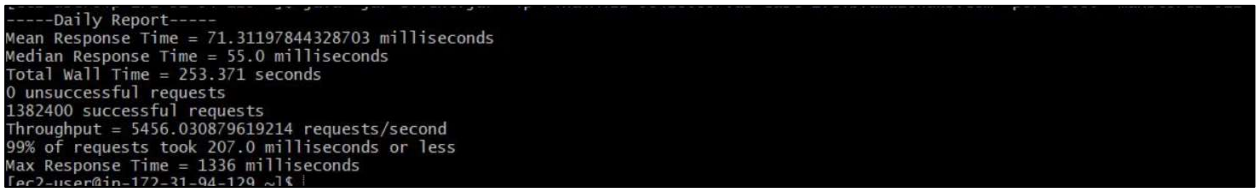
The Swagger codegen was used to create the Swagger client API interface which the client is built on.

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

256 Threads:

512 Threads:



1024 Threads: