SOEN 423

Assignment 1

By: Saffia Niro

October 9, 2020

1.0 Overview

This application uses a client-server model.

There are three stores--QC, ON, and BC--each of which have their own server class. Each store also has their own inventory of type HashMap<String, Item>. An `Item` object has the following attributes:

- `String itemID`
- `String itemName`
- `int quantity`
- `Double price`
- `Date dateOfPurchase`
- `ArrayList<String> IDOfBuyer`

  This ArrayList contains the IDs of customers who have bought it
- `PriorityBlockingQueue<String> waitingList`

  This PriorityBlockingQueue contains the IDs of the customers who are waiting to buy an item that is currently out of stock

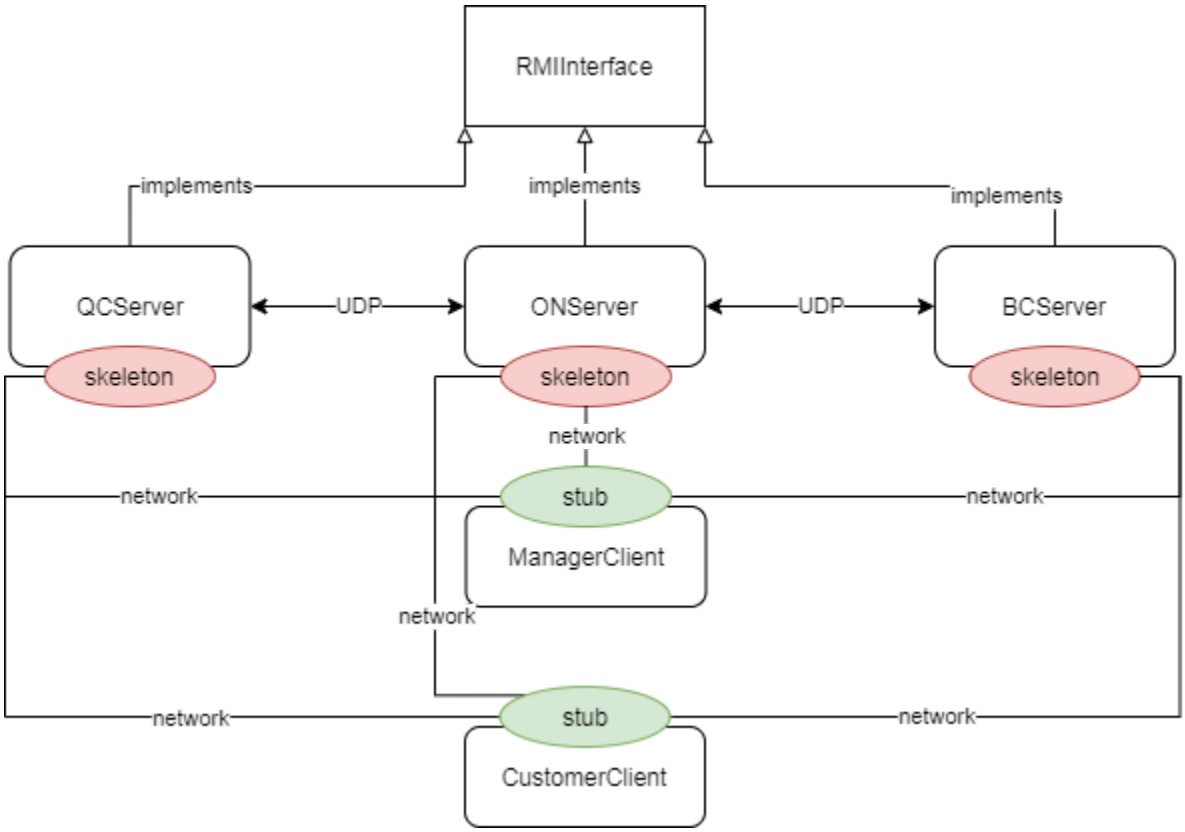
Store inventories are initialized with 5 items each.

There are two types of clients, customers and managers, both of which have their own client class, `CustomerClient` and `ManagerClient`, respectively.

Each server and each client has their own .txt log file, where all operations are recorded. Since multiple operations may be occurring at any one time (for example, an item can be purchased for a customer that was on a waiting list at the same time as another customer is searching for an item), access to these files is synchronized.

To allow for the use of remote objects, Java Remote Method Invocation (RMI) is used. To allow for communication between servers, UDP is used. To allow for concurrency, Java

threads are used. A simplified diagram of the application is shown below:



2.0 Server Design

Each server implements the RMI interface, which includes the following methods:

- ➔ **boolean addItem**(String managerID, String itemID, String itemName, int quantity, double price) throws java.rmi.RemoteException;
- ➔ **boolean removeItem**(String managerID, String itemID, int quantity) throws java.rmi.RemoteException;
- ➔ **String listItemAvailability**(String managerID) throws java.rmi.RemoteException;
- ➔ **double purchaseItem**(String customerID, String itemID, Date dateOfPurchase) throws java.rmi.RemoteException;
- ➔ **String findItem**(String customerID, String itemName) throws java.rmi.RemoteException;
- ➔ **boolean returnItem**(String customerID, String itemID, Date dateOfReturn) throws java.rmi.RemoteException;

Servers also contain the following method to facilitate synchronization:

- ➔ **void checkAvailability**(String itemID);

  This method, which runs on a separate thread, checks the availability of an item that went out of stock, and when it becomes available invokes the `purchaseItem` method as well as removing the purchaser from the queue. To prevent race conditions with the `purchaseItem` method, both methods acquire a lock when they are modifying an item's quantity.

To export the RMI object and bind it to the registry, the following method is used:

- ➔ **static void** RMI();

To allow the server to act as a UDP client that can send requests to other stores, the following methods are used:

- ➔ `String sendPurchaseRequest(String store, String customerID, String itemID);`
- ➔ `String sendFindItemRequest(int serverPort, String customerID, String itemName);`
- ➔ `String sendReturnRequest(String customerID, String itemID);`

Finally, to allow servers to act as a UDP server that will receive requests from other stores, the following method is used:

- ➔ `static void receive();`

To start the server, there is a `main` method containing two threads, one for the UDP connection and one for the RMI.

3.0 Client Design

There are two client classes, `ManagerClient` and `CustomerClient`. They are in charge of keeping track of individual users' information, as well as requesting information from stores using RMI. They contain the following attributes:

- `String store`
- `RMIInterface stub`
- `File log`
- ManagerClient only:
    - `String ManagerID`
- CustomerClient only:
    - `String customerID`

- `double budget`

Clients can perform operations by using the RMI Naming class to lookup the remote object that is associated with their own store, using its registry URL. From there, the client can use a stub to call the correct method (`addItem`, `removeItem`, and `listItemAvailability` in the case of managers; `purchaseItem`, `findItem`, and `returnItem` in the case of customers).

4.0 Application

Finally, to run the application, a Driver class is provided. This class allows a user to input information from the console.

To increase performance, each operation attempted by a customer or manager runs on a thread. Synchronization is used to ensure that the right output is printed to the screen at the right time. Thus, if theoretically this program was run on multiple hosts, users would be able to concurrently use the application.

5.0 Example test scenario

Procedure:

1. Log in as a QC manager and list items
2. Log in as a user from BC
3. Purchase an item that is out of stock in QC and go on the waiting list
4. Log in as a user from ON
5. Purchase the same item that was out of stock in QC and go on the waiting list

6. Log in as a QC manager and add 1 instance of the out of stock item
7. Log in as the BC user who automatically bought the item and return it
8. Log in as an ON user and find said item
9. Log in as a manager and remove said item
10. Log in as a QC user and try to purchase an item that is too expensive from ON
11. Exit program

6.0 Final remarks

The most difficult, yet certainly the most important, part of this assignment was enabling communication between the stores and the users. In particular, it was sometimes a challenge to figure out how to pass the required information to the right place without incurring too much overhead. One way I solved this was by using strings in the request on the sender side, with arguments separated by a comma. These requests are then parsed into their components on the receiver side.