Online Restaurant Service

Author: Anshul Rao

I. Introduction

As part of this project I implemented an online restaurant service where there'll be three kinds of clients, a user who will place orders, delivery agents who will be responsible for delivering the orders and a chef who will add items to the menu as they are made and prepare the orders that are placed. While there can be multiple users and delivery agents, only one chef is present. The service will have two main components: the OrderService (for handling users and their orders, etc.), and KitchenService (for managing items in/out of the menu, processing orders placed, assigning delivery to agents, etc.). OrderService will have replicated secondary server(s) whose state will be maintained and when the primary server is down or crashes, then the user(s) automatically start communicating with the secondary healthy server. KitchenService is not replicated and only a single instance exists for it. There is also a third-party service FinanceService for payments but while the OrderService will communicate with it, it will remain out of scope of the restaurant service as a whole. The algorithms used will be distributed mutual exclusion, peer-to-peer communication, replicated data management, etc.

For simplicity I will not maintain user accounts in the database. This means that registration is not required to use this service. Any individual can place an order using this service. To track the user who placed a certain order, we will refer to their contact details.

Another key detail about the service is that it opens at 12 AM and ends at 10 PM. So the order log of every day is maintained and freshed before starting every morning. This means that there is no need to historically maintain the order records in one table. Also, relational databases like MySQL have not been used. Instead, the data is maintained in memory and stored as files on the server. Like mentioned before, since the data is accumulated over a course of one day, the files are dated. For example, order data for 23rd December, 2022 will be present in the file 20221223.

Lastly, the items that the restaurant provides are constant. They are Burger, Pizza, Pasta and Fries. Only the count of these items available at a specific point of time varies. It increases when the chef adds them and decreases when an order is placed.

II. Component Functionalities

Clients communicating with the service:-

1. *User*:
    - Communicates with the OrderService.
    - Can view items that are currently available,
    - Can place a new order.
    - Can track status of a placed order.

2. *DeliveryAgent*:
    - Constantly polls the KitchenService for any ready unassigned orders.
    - Marks the orders as done once the order is delivered.

3. *Chef*
    - Adding items using KitchenService.
    - Preparing orders and marking them as ready using KitchenService.

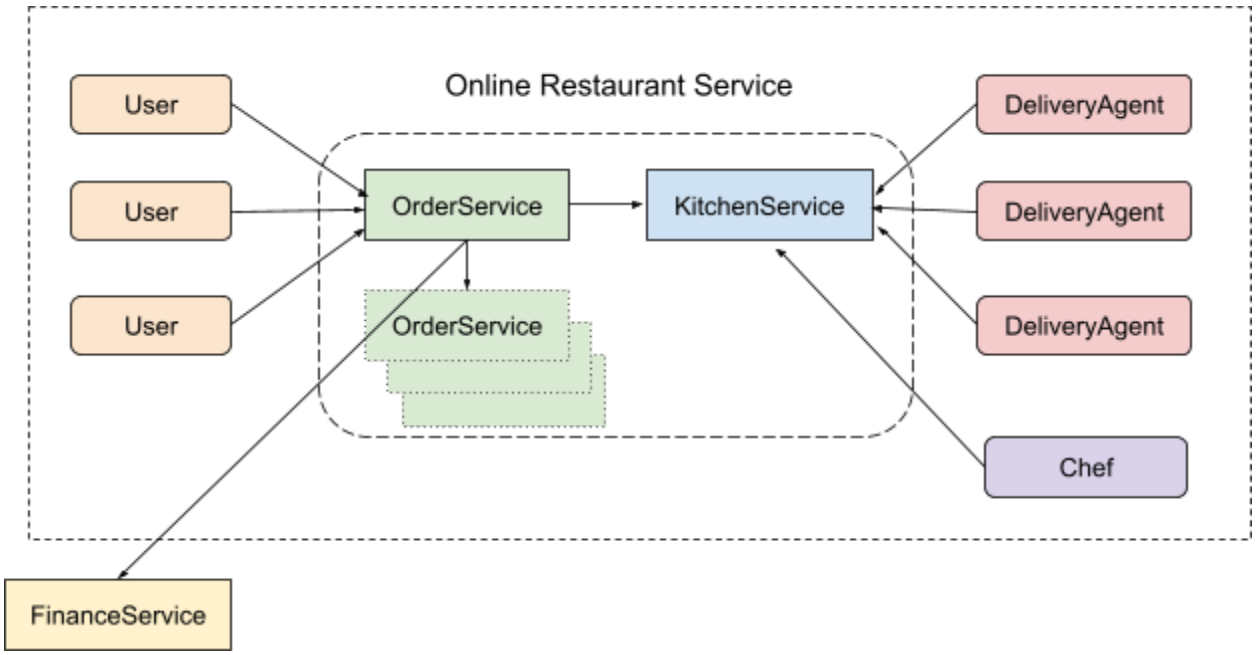
Servers communicating with the clients (and each other):-

1. *OrderService:*
    a. Handles User clients and places their order, returns the latest menu and sends the current status of the order when asked.
    b. Interacts with KitchenService for placing and tracking orders.

2. *KitchenService*:
    a. Processes the order placed by OrderService.
    b. As soon as an order is placed it is placed in a queue that Chef is polling continuously.
    c. Enables Chef client to add items and mark an order as ready.
    d. Enables DeliveryAgent clients to poll ready orders and mark them as complete/delivered.

For a comprehensive view of methods, refer to the class diagram on the next page.

III. Architecture Overview Diagram



IV. Database Schema

NOTE: The data is not maintained using a relational database management system like MySQL but stored in memory and written to files.

OrderInstance: User-Defined Object	
orderID	String
orderStatus	'PLACED', 'READY', 'COMPLETE', 'INVALID'
name	String
amount	double
items	HashMap<String, Integer>
userContact	long

OrderData: ConcurrentHashMap<String, OrderInstance>

```
OrdersReady: Set<String>
```
Eg. {O-11670788336624, O-21670788346574}

```
OrdersComplete: Set<String>
```
Eg. {O-11670475936624, O-21670768790574}

```
OrdersAssigned: Set<String>
```
Eg. {O-10990788336624, O-25648788346574}

```
ItemCounts: ConcurrentHashMap<String, Integer>
```
Eg. {'Burger': 14, 'Fries': 12, 'Pasta': 4, 'Pizza': 20}

```
OrdersPlaced: ConcurrentLinkedDeque<OrderInstance>
```

The project will be implemented in Java and the key libraries used will be the ones provided by it and no external libraries/frameworks will be used.

V. Implementation

Key Algorithms

1. Distributed Mutual Exclusion - The method *placeOrder*() in *OrderService* is synchronized meaning only one user can place an order at one point of time. This is important because when an order is placed items are decreased so parallel orders cannot be successfully placed. Similarly, *findReadyOrder*() is also synchronized, meaning that while multiple delivery agents will check for ready unassigned orders, only one gets assigned a specific order (and not two or more of them). Apart from them the queue and hash maps used are also concurrent which ensures safe insertion, updation and retrieval of elements.
2. Group Communication/Polling - The idea proposed earlier was that once the order has been placed, the *KitchenService* multicasts datagram packets to all the delivery agents who are active and listening to its port. The first one to reply back, gets assigned the delivery. But during implementation I realized that multicasting datagrams might add complexity and desired behavior can be achieved efficiently by simply making the delivery agents poll for read unsigned orders from the

KitchenService. This way only one delivery agent is assigned a specific order and unless that delivery is completed, that agent is not assigned a new delivery. It can be said that this approach (of continuously checking with the server) is resource intensive but given the current scope, it was not impacting the performance negatively.

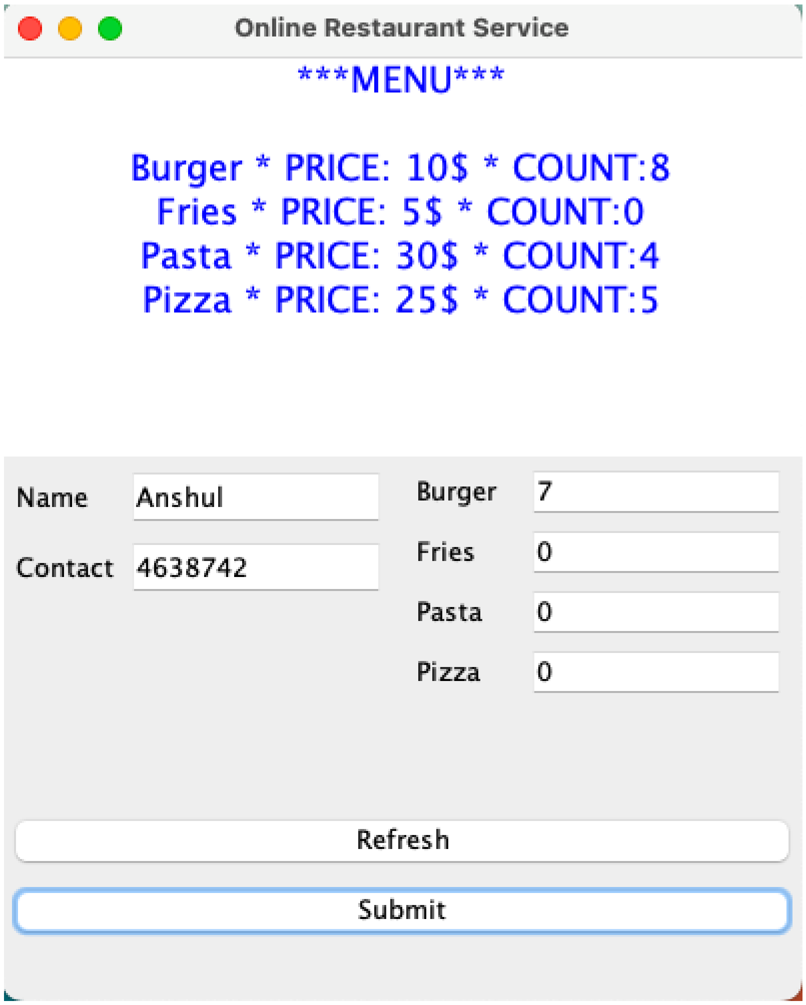
3. Replicated Data Management/Fault Tolerance - I have implemented a customized version of passive (primary-backup) replication where the clients communicate with the primary server and secondary server takes primary's place if primary fails. The *OrderService* server which directly deals with the customer (*User*) is replicated. The *User* communicates with the primary server and if it goes down or crashes then the User falls back to communicating with the secondary (replicated) server. All updates and changes are made to both the primary and secondary server(s). Also, all clients will be communicating with one server only, primary or one of the healthy secondary servers at one point of time. To enable Users to detect the secondary server(s), their endpoints should be added to *secondary-order-service.properties*. This feature also enables continual operation of the *User* (client) despite the failure of the primary server, and hence it makes the infrastructure fault tolerant too to an extent.
4. P2P Network - There are component(s) in the system which do not fit strictly into a server or client role but both. For example, the *KitchenService* and *OrderService* are peers and there is communication between them. Similarly, the third-party service called *FinanceService* is also a peer of *OrderService* but it is communicating with it too.

Other Features

1. User client times out if the OrderService takes more than 5 seconds to respond to it and it considers that request as having failed.
2. If an order is active then the same user cannot place another order, they are notified that they already have an active order if they try to place another one.
3. All item counts needed by the User (including 0) need to be entered by them otherwise they are notified that they need to enter valid counts.
4. The UI needs to be refreshed by the User to see the latest item counts and the latest status of their order (if they have placed one).

Client Interfaces

User



Chef

```
1. ADD <item-name> <item-count>
2. READY <order-id>
3. EXIT
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
A new order has been placed!
OrderInstance{orderID='O-21670793688572', items={Burger=0, Pizza=1, Fries=0, Pasta=0}}
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
ADD Pasta 4
Dec 11, 2022 4:22:00 PM client.chef.Chef processResponse
INFO: 1670793720691: Response received from server, the operation was executed successfully :)
1. ADD <item-name> <item-count>
2. READY <order-id>
3. EXIT
READY O-21670793688572
Dec 11, 2022 4:22:25 PM client.chef.Chef processResponse
INFO: 1670793745324: Response received from server, the operation was executed successfully :)
```

Delivery Agent

```
********************************************************************************
A new delivery has been assigned!
ORDER ID: O-21670793688572
********************************************************************************
done
Dec 11, 2022 4:23:23 PM client.delivery_agent.DeliveryAgent execute
INFO: 1670793803412: Order with ID = O-21670793688572 has been delivered!
```

VI. Conclusion

I was able to build the system that I had originally planned while proposing this project. There are some key design changes that I made as I worked on this project as they were more suitable and efficient. For example, earlier I planned to multicast datagrams for assigning delivery to delivery agents but later found polling much less complex and efficient. In fact this was one of the key takeaways where I found how polling a queue made communication where a client needs to be notified of a quick change/update much more efficient and robust as long as the resources are available. The idea/algorithm was influenced by the concept of polling message queues in distributed systems.

The current system is stable but I would like to extend and enhance some features in the long run. For example, currently the secondary service is not looked for dynamically when the primary server fails. I tried getting this to work but there were exceptions when I did that and due to limited time and resources I had to drop this idea. Apart from that, the GUI needs the user to enter values for item counts (even if they are zero), I would like to get rid of that and let empty text fields imply zero counts. This is a far easier change to add but again, due to limited time I decided not to focus on UI constraints and worked on more important distributed systems aspects. But this will still be a good improvement that should be done.

VII. Appendix

- GITHUB: https://github.com/anshulrao/online-restaurant-service
- CODE: https://github.com/anshulrao/online-restaurant-service/tree/master/src
- DEMO VIDEO: https://github.com/anshulrao/online-restaurant-service/blob/master/Demo.mp4