COMP41720-Distributed Systems-Group - 19

Mihir Vir

23207095

Navjot Singh

23200408

Ruchen Lai

23201083

Yilin Zhang

2221299

Xing Zheng

23200962

## Synopsis

We have created a distributed airline agent system. It offers a comprehensive interface for clients to seamlessly browse and purchase tickets from various airlines. The platform encompasses user-friendly functionalities, including management of user accounts, sending emails and efficient order tracking. It also extends support to airlines by offering a dedicated facility for seamlessly adding or updating flight information. Additionally, we provide a dynamic feature for user activity tracking, allowing the system to intelligently adjust ticket prices based on user interactions like user searches and ratings.

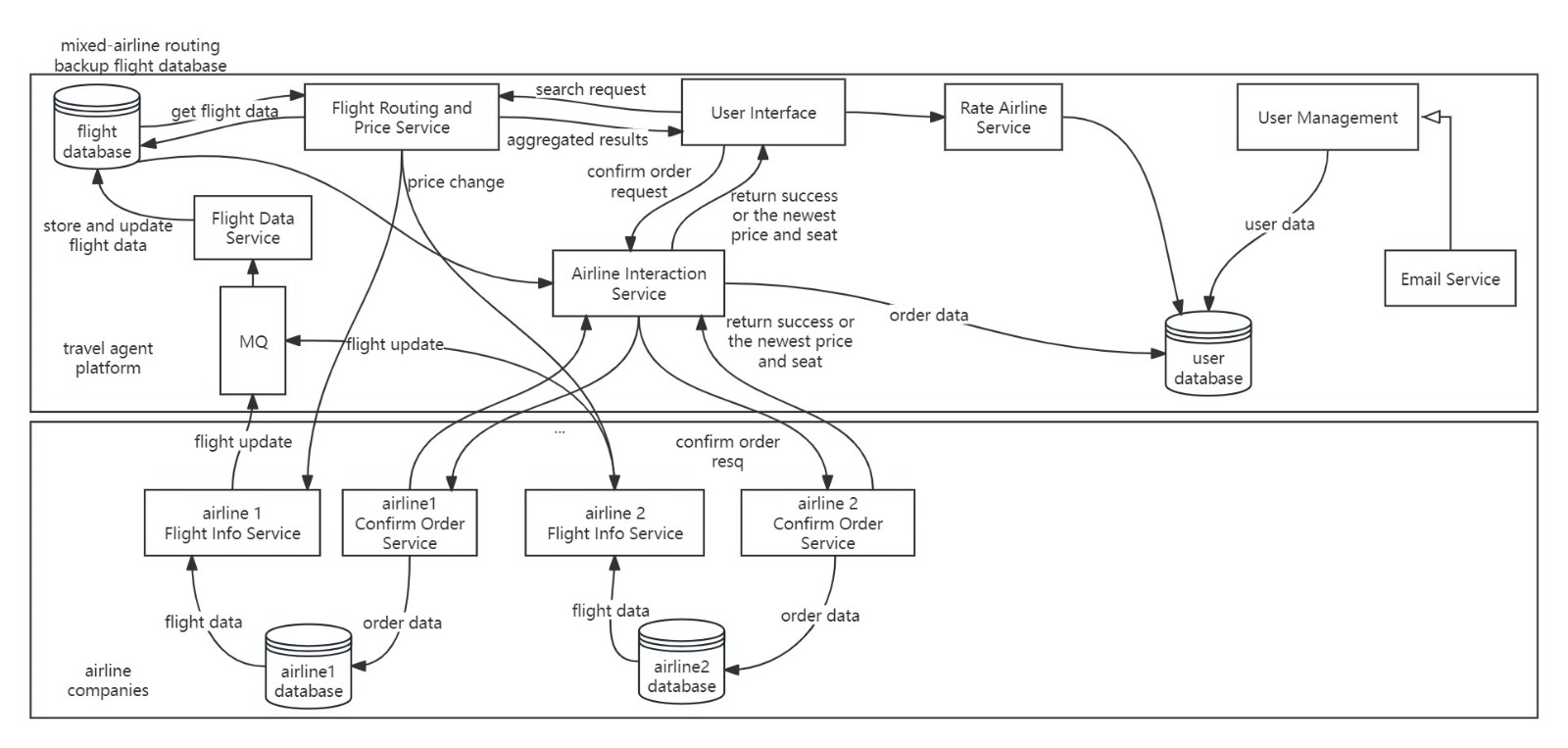
## Technology Stack

* Kubernetes: We have used Kubernetes to deploy and manage our services. Its main advantage is that it provides fault tolerance and scalability.
* SprintBoot: it provides REST communication with java services
* Flask: it provides REST communication with python services
* Mongo: A NoSQL database that is being used to store and manage data
* RabbitMQ: A message-queueing service which we use to update flights data asynchronously
* Spark Graphframe: A graph processing library which we used to calculate flight routes
* ReactJS: frontend library which is used to build reusable UI components

## 

## System Overview

Our system consists of a front-end client interfacing with four distinct services and multiple airlines registered with it. The ‘user management service’, coupled with an ‘email service’, handles user-related functionalities. Simultaneously, the ‘flight routing and price service’ manage flight details, routes and its prices, while a third ‘airline interaction service’ facilitates communication with two partner airlines. These airlines provide comprehensive flight information and confirm bookings. The system's distributed architecture enhances scalability and flexibility, ensuring efficient management of user interactions, flight operations, and seamless collaboration with airline partners.



**System Flow**

Our service starts from the airlines. They provide their flight details to our system through a rabbitMQ queue. Our system reads this Queue and inserts data into our Mongo DB.

We provide an interface for users where they can create accounts. When they first sign up we send them an email for verification, and then they can subsequently log in.

To get flights, the user contacts the flight routing service and sends their source and destination. Flight routing service loads data from Mongo which was provided by the airlines earlier.

The flight routing service uses GraphFrames to find and return direct and layover flights. It also keeps track of user activity like a number of searches to dynamically modify the price for each user.

All the flights are then shown to the user, the user can select a preferred flight and place an order through our airline interaction service to get the latest flight prices. To ensure the timeliness of orders, we provide order timeout cancellation by using rabbitMQ's dead letter queue.

The airline interaction service then passes order information to the airlines, which confirms the booking. We provide efficient interaction between airline interaction services and airlines by using multi-threading.

After this user can provide a rating to the airline which is further sent to airlines for the sake of analytics so they can modify their future prices.

**Fault tolerance and Scalability**

First and foremost, we deployed our application using Kubernetes. It provides multiple replicas of the same services. So if for any reason 1 goes down, its replica can take over and Kubernetes can start another replica in the background. We also use ingress nginx within Kubernetes, which provides scalability through load balancing.

Also for our database, we are using Mongo, a distributed NoSQL database. It scales by dividing data across many servers and ensures fault tolerance through data replication. In case of server failures, MongoDB automatically switches to a backup.

To calculate flight routes we are using Apache GraphFrames. It divides the job of finding routes into a Directed Acyclic Graph (DAG). This DAG is then divided into many smaller tasks which are computed in parallel. it also provides fault tolerance by storing intermediate data in resilient distributed datasets and providing a data lineage which can be traced back in case of failures to recover lost data.

## Contributions

**Navjot Singh (23200408)**

Worked on calculating flight routes using GraphFrames, keeping track of user activity and rating in order to modify price. deployed services on kubernetes and set up mongo, rabbitmq and ingress nginx.

**Mihir Vir (23207095)**

Worked on user management service, designed client and deployed client, user management service and email service on kubernetes

**Ruchen Lai (23201083)**

Worked on flight data service, airline interaction service to interact with airlines, storing and updating flights in database from airlines, creating order and purchasing the order based on interacting with airlines.

**Yilin Zhang (22212992)**

Worked on email service. Verification links are sent through the email service, which can be clicked on by the user to automatically complete account verification.

**Xing Zheng (23200962)**

Worked on airline services, initializing airline databases, sending flight info to flight data service through rabbitmq, implementing REST APIs about updating prices, getting flight info and order confirmation. Besides, deployed airline services and initialization jobs on kubernetes and recorded the video.

## Reflections

One major challenge we encountered was working in a group. As everyone had different approaches to the problem and all of our services depended on each other, a lot of communication was required. We solved this by discussing all the ideas in group discussions, so we could have a poll on whether to proceed with that idea or not. Also to manage the details of how services would interact with each other we created a document that would contain the contract of all APIs, schemas of databases etc. So even if a service is not ready, other services dependent on it can still be developed and worked on.

The next major challenge was to use Kubernetes, as it was an entirely new technology for us, there was a major learning curve. We had to learn deployment and routing. Also, Kubernetes was a very heavy tool. We were not sure if our laptop would be able to handle it. so we had to configure the settings of docker to increase memory, CPU and storage and had to make our service efficient, so they could run smoothly.