FIT5183 Programming for Distributed, Parallel and Mobile Systems

Assignment One, Semester 2A, 2014

Weiwei SUN ID: 25467247

PART I: Design Document

1.1 Introduction

This document will be helpful to clarify the detail of my work in assignment one. Assignment 1 involves the design and implementation of a Flight Booking Center (acts as a broker). This Flight Booking Center is an online system that allows user to search for, compare rates, check availability, and buy e-tickets of International flights that are carried by different airlines.

This document will demonstrate the process of this distributed Flight Booking Center based on Java Socket, and the system will follow the design principal of MVC (Model-View-Controller). To support the data access, the JDBC acts as a middleware tool and MySQL acts as a database will be used in this system.

To facilitate the version control of this project, Git is selected as a version control tool and Github is taken as a remote repository. Therefore, you also can get the sources codes using the following commands:

```
$git clone https://github.com/wwsun/flight-tickets-service.git
```

1.2 System Analyses

1.2.1 System Architecture

This system will be developed using a typical 3-tier system, including representation tier, service tier, and persistent tier. These three tiers are helping to lead a lower coupling which is helpful for program modifying and expanding. Figure 1 shows the three tiers deployment diagram.

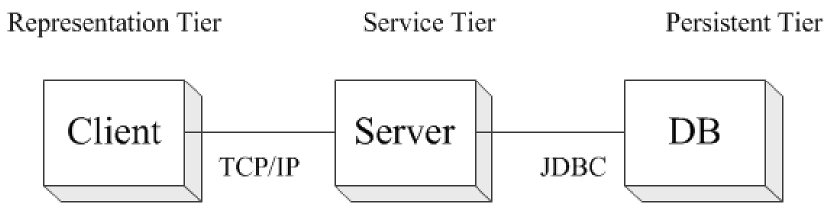


Figure 1: Deployment Diagram

The representation tier used to accept the data sending from the server and afford a user interface to show the data, the service layer should including two parts, client-side and server-side.

The service tier used to process the data from database and distribute the data to the appropriate applications, the service tier will be encapsulated in the serve-side. In this system, the service tier will be implemented by the DAO mechanism which allows programmers to divide the database access logic and service logic.

The persistent tier used to store and access data, the JDBC, which is a powerful middleware technology that support MySQL-based applications, will be used to connect the persistent tier and service tier.

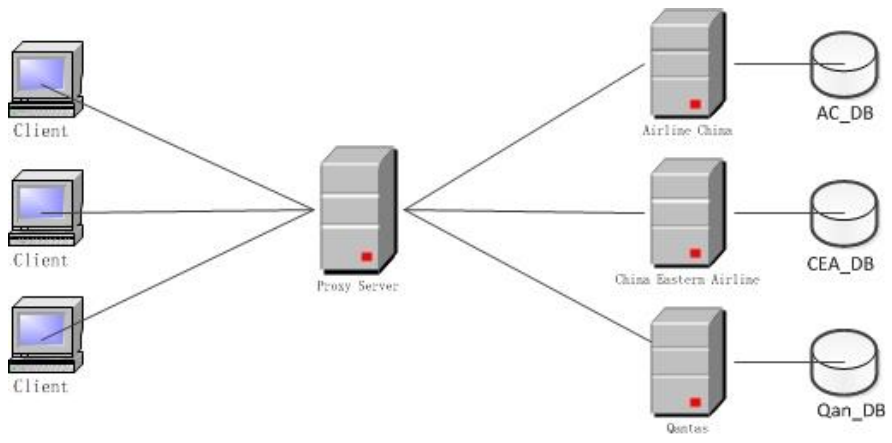


Figure 2: System architecture

Figure 2 shows the system architecture, the proxy server will be the center of this system, which in charges of accepting the requests and distributing them to different servers of different airlines. In this system, users can query, compare rates, check availability and buy tickets of International flights via the proxy server.

Each airline has its own servers and databases, they maintain their own databases. Three airlines are implemented in this system, including Airline China (AC), China Eastern Airlines (CEA), and Qantas (Qan). AC and CEA are two famous airlines in China, also Qantas is a famous airline in Australia, which means that this flight booking system serves for international flights.

1.2.2 System Logic

Figure 3 shows the system logic and message passing path, all the message should be handled by the proxy server, then distribute to the specific server(s). In this system, HOPP (Half-object plus Protocol) is designed for separating different logics, that is to say, some business logics should be extracted out as an independent class. Especially in the proxy server, two different HOPPs are developed to separate two different logics, CM (communication module) could invoke the methods of PSHOPP to execute different services, and PSHOPP could invoke the methods providing by PCHOPP to execute the real logics that communicate with the CM of airline server.

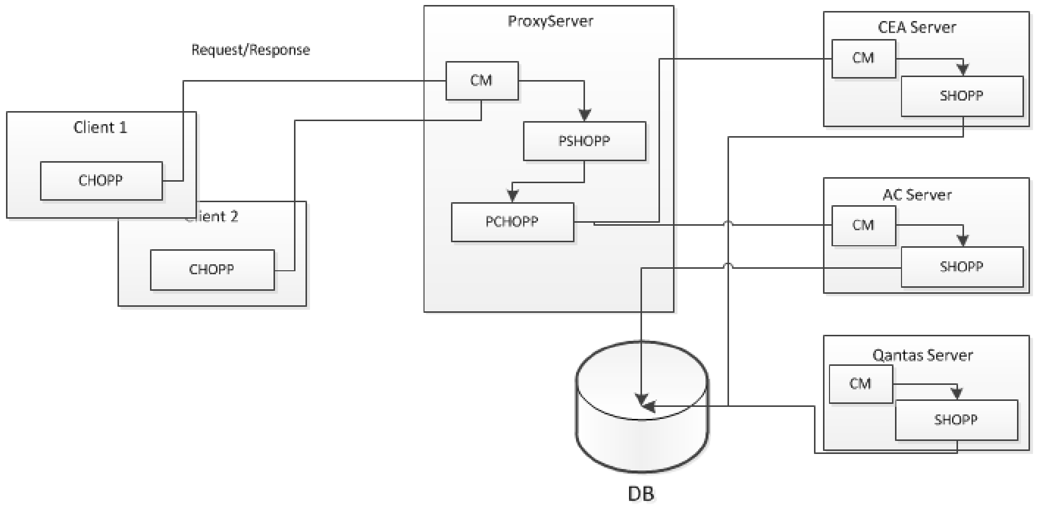


Figure 3: System logic structure

1.2.3 Text Protocol

The text protocol and message passing mechanism, and some other naming notations will be talked in this part. For a purpose of good understanding this system, it is helpful for us to design the text protocol and message format.

(1) Text protocol

Client request	Server response
query <from, to, leaveTime>	"from" means the city the client you want to leave "to" means the city the client want to go "leaveTime" means the date you want to leave send "ERROR" if failed send the list of available airlines
query <from, to, leaveTime, returnTime>	"from" means the city the client you want to leave "to" means the city the client want to go "leaveTime" means the date you want to leave "returnTime" means the date you want to go back send "ERROR" if failed send the list of available airlines
reg <air, username, phone,email,creditcard>	
order <fid1,fid2,username>	"fid1" and "fid2" means two specific flights that the client want to order (return trip) "username" means the name of the user send "ERROR" if failed send "SUCCEEDED" if succeed
order <fid,username>	"fid" means the specific flight that the client want to order (one-stop trip)
quit	close the connection

Table 1: Text Protocol

(2) City code

In order to simplify the complexity of this system, only four cities are support in this system. Therefore, we assume that all the international flights only support Beijing, Shanghai, Melbourne, Sydney. Each city should have a code to stand for its name in the ticket number. The city code shows in Table 2.

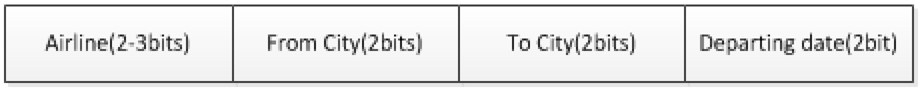
City code	City name
01	Melbourne
02	Sydney
03	Shanghai
04	Beijing

Table 2: City code

(3) Flight number

In this system, the flight number should follow a specific format, that is to say, we using different bit to stand for different function. Figure 4 shows the format of flight number that is defined in this system.

For instance, the flight number QAN010310 means the Qantas airline have a flight departing from Melbourne in 10th, March, and will landing in Shanghai some later.



Airline(2-3bits)	From City(2bits)	To City(2bits)	Departing date(2bit)

Figure 4: Flight number format

1.3 System Design

1.3.1 Database Schema

Three airlines are designed in this distributed Flight Booking system, including Airline China (AC, China), China Eastern Airlines (CEA, China), and Qantas (QAN, Australia). In order to simplify the database design, it is assumed that all airlines (Airline China, China Eastern Airlines, and Qantas) will maintain the same schema of these three tables, including table user, table flight, and table order.

User Schema (including cea_user, ac_user, and Qantas_user) shows in Table 3, the key of this table is username.

Name	**Type**	**Comment**
username*	varchar	20
phone	varchar	11
email	varchar	20
creditcard	varchar	16

Table 3: User table

Flight Schema (including cea_flight, ac_flight, and Qantas_flight) shows in Table 4, the key of this table is fid.

Name	Type	Comment
fid*	varchar	20
airline_comp	varchar	30
departure_city	varchar	20
destination_city	varchar	20
departing_date	date	
tickets	int	2

Table 4: Flight table

Order Schema (including cea_order, ac_order, and Qantas_order) shows in Table 5, the key of this table is oid. Attribute Fid is a foreign key of table flight, and attribute Username is a foreign key of tables user.

Name	Type	Comment
oid*	int	5
fid	varchar	9
username	varchar	20

Table 5: Oder table

1.3.2 Functions Design

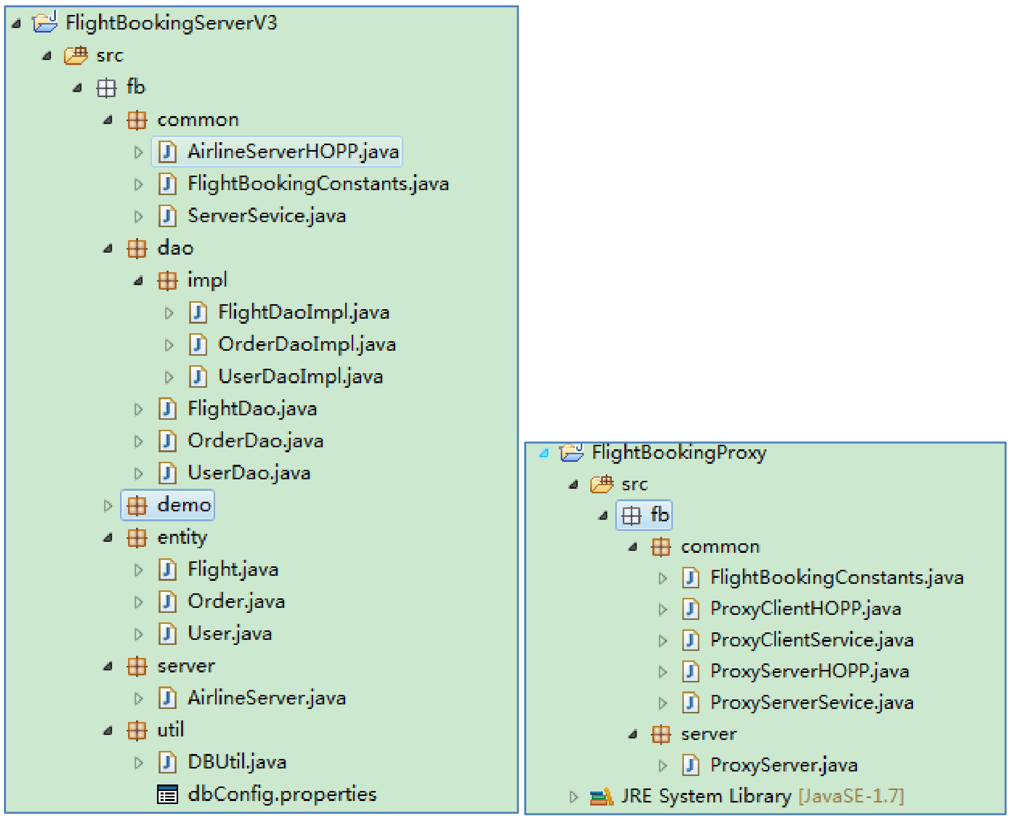
This system will implement the core function of flight listing, flight querying (based on the departure city and destination city, or the leaving time and returning time), and flight checking which are needed in a real Flight Booking system.

In this system, only two-way (return trip), direct (one stop) International flights operated by various airlines between cities in Australia and China are considered. That is to say, domestic flights (e.g. Shanghai to Beijing) are not supported in this system. There are four functions in this system, including flights querying, user register, flight ordering, and orders checking.

User should follow some specific commands to use this system because only command line is developed in this system; also someone other should develop their own Graphic User Interface (GUI) by some other programming languages or tools. In this system, the core function is how to design a distributed system based on Java Socket, rather than the GUI. In this section, only function logic will be talked, the detail of user operation will be talked in Part II User Guide.

1.3.3 Packet organization

For the purpose of a good project structure, the project will be organized with different packets, which will be helpful to modified and expanded. In this project, the packet organization in server-side is developed into five main parts; in consist of common, dao (dao.impl), entity, server, and util. Each packet used to organize different Java classes. Figure 5 shows the packets in server-side. It is similar in client-side.



```
FlightBookingServerV3
  src
    fb
      common
        AirlineServerHOPP.java
        FlightBookingConstants.java
        ServerSevice.java
      dao
        impl
          FlightDaoImpl.java
          OrderDaoImpl.java
          UserDaoImpl.java
        FlightDao.java
        OrderDao.java
        UserDao.java
      demo
      entity
        Flight.java
        Order.java
        User.java
      server
        AirlineServer.java
      util
        DBUtil.java
        dbConfig.properties
```

```
FlightBookingProxy
  src
    fb
      common
        FlightBookingConstants.java
        ProxyClientHOPP.java
        ProxyClientService.java
        ProxyServerHOPP.java
        ProxyServerSevice.java
      server
        ProxyServer.java
  JRE System Library [JavaSE-1.7]
```

Figure 5: Packet organization in Server-side

Packet Common used to store the public interface, constants, and HOPP of server-side.

Packet Dao used to store the database access classes including database read and write methods. It’s child packet Impl used to store their interface implemented classes.

Packet Entity used to store the entity classes in response to the tables in database.

Packet Server used to store the server class including the main function.

Packet Util used to store the utility classes and property file.

1.4 UML Class diagram

In this part, we show the class diagrams of all three parts of this flight booking system. Figure 6 shows the class diagram of airline server, Figure 7 shows the class diagram of proxy server and Figure 8 shows the class diagram of client.

The airline server is the most complex part of this project because of its direct connection with database; there are six packages in the project of airline server. Figure 6 shows the linkages of different modules. To clarify this figure, the ServerService Interface used to specify the HOPP interface that the HOPP class must implement. And three DAO interfaces used to specify the database access interface that the data access class must implement. For a purpose of lose coupling, every different business logics are extracted out as an independent Java class.

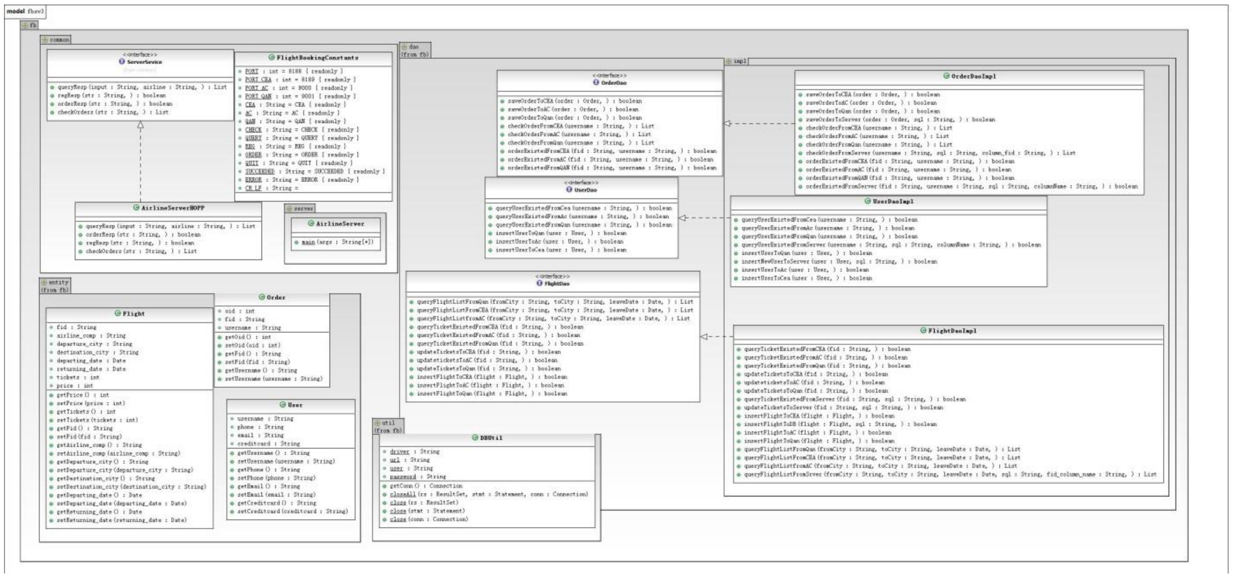

model fbsv3
fb
common
<<interface>>
ServerService
queryResp (input : String, airline : String,) : List
regResp (str : String,) : boolean
orderResp (str : String,) : boolean
checkOrders (str : String,) : List
FlightBookingConstants
PORT : int = 8188 { readonly }
PORT CEA : int = 8189 { readonly }
PORT AC : int = 9000 { readonly }
PORT QAN : int = 9001 { readonly }
CEA : String = CEA { readonly }
AC : String = AC { readonly }
QAN : String = QAN { readonly }
CHECK : String = CHECK { readonly }
QUERY : String = QUERY { readonly }
REG : String = REG { readonly }
ORDER : String = ORDER { readonly }
QUIT : String = QUIT { readonly }
SUCCEEDED : String = SUCCEEDED { readonly }
ERROR : String = ERROR { readonly }
CR LF : String =
AirlineServerHOPP
queryResp (input : String, airline : String,) : List
orderResp (str : String,) : boolean
regResp (str : String,) : boolean
checkOrders (str : String,) : List
server
AirlineServer
main (args : String[*])
entity (from fb)
Flight
fid : String
airline_comp : String
departure_city : String
destination_city : String
departing_date : Date
returning_date : Date
tickets : int
price : int
getPrice () : int
setPrice (price : int)
getTickets () : int
setTickets (tickets : int)
getFid () : String
setFid (fid : String)
getAirline_comp () : String
setAirline_comp (airline_comp : String)
getDeparture_city () : String
setDeparture_city (departure_city : String)
getDestination_city () : String
setDestination_city (destination_city : String)
getDeparting_date () : Date
setDeparting_date (departing_date : Date)
getReturning_date () : Date
setReturning_date (returning_date : Date)
Order
oid : int
fid : String
username : String
getOid () : int
setOid (oid : int)
getFid () : String
setFid (fid : String)
getUsername () : String
setUsername (username : String)
User
username : String
phone : String
email : String
creditcard : String
getUsername () : String
setUsername (username : String)
getPhone () : String
setPhone (phone : String)
getEmail () : String
setEmail (email : String)
getCreditcard () : String
setCreditcard (creditcard : String)
dao (from fb)
<<interface>>
OrderDao
saveOrderToCEA (order : Order,) : boolean
saveOrderToAC (order : Order,) : boolean
saveOrderToQan (order : Order,) : boolean
checkOrderFromCEA (username : String,) : List
checkOrderFromAC (username : String,) : List
checkOrderFromQan (username : String,) : List
orderExistedFromCEA (fid : String, username : String,) : boolean
orderExistedFromAC (fid : String, username : String,) : boolean
orderExistedFromQAN (fid : String, username : String,) : boolean
<<interface>>
UserDao
queryUserExistedFromCea (username : String,) : boolean
queryUserExistedFromAc (username : String,) : boolean
queryUserExistedFromQan (username : String,) : boolean
insertUserToQan (user : User,) : boolean
insertUserToAc (user : User,) : boolean
insertUserToCea (user : User,) : boolean
<<interface>>
FlightDao
queryFlightListFromQan (fromCity : String, toCity : String, leaveDate : Date,) : List
queryFlightListFromCEA (fromCity : String, toCity : String, leaveDate : Date,) : List
queryFlightListfromAC (fromCity : String, toCity : String, leaveDate : Date,) : List
queryTicketExistedFromCEA (fid : String,) : boolean
queryTicketExistedFromAC (fid : String,) : boolean
queryTicketExistedFromQan (fid : String,) : boolean
updateTicketsToCEA (fid : String,) : boolean
updateticketsToAC (fid : String,) : boolean
updateTicketsToQan (fid : String,) : boolean
insertFlightToCEA (flight : Flight,) : boolean
insertFlightToAC (flight : Flight,) : boolean
insertFlightToQan (flight : Flight,) : boolean
impl
OrderDaoImpl
saveOrderToCEA (order : Order,) : boolean
saveOrderToAC (order : Order,) : boolean
saveOrderToQan (order : Order,) : boolean
saveOrderToServer (order : Order, sql : String,) : boolean
checkOrderFromCEA (username : String,) : List
checkOrderFromAC (username : String,) : List
checkOrderFromQan (username : String,) : List
checkOrderFromServer (username : String, sql : String, column_fid : String,) : List
orderExistedFromCEA (fid : String, username : String,) : boolean
orderExistedFromAC (fid : String, username : String,) : boolean
orderExistedFromQAN (fid : String, username : String,) : boolean
orderExistedFromServer (fid : String, username : String, sql : String, columnName : String,) : boolean
UserDaoImpl
queryUserExistedFromCea (username : String,) : boolean
queryUserExistedFromAc (username : String,) : boolean
queryUserExistedFromQan (username : String,) : boolean
queryUserExistedFromServer (username : String, sql : String, columnName : String,) : boolean
insertUserToQan (user : User,) : boolean
insertNewUserToServer (user : User, sql : String,) : boolean
insertUserToAc (user : User,) : boolean
insertUserToCea (user : User,) : boolean
FlightDaoImpl
queryTicketExistedFromCEA (fid : String,) : boolean
queryTicketExistedFromAC (fid : String,) : boolean
queryTicketExistedFromQan (fid : String,) : boolean
updateTicketsToCEA (fid : String,) : boolean
updateticketsToAC (fid : String,) : boolean
updateTicketsToQan (fid : String,) : boolean
queryTicketExistedFromServer (fid : String, sql : String,) : boolean
updateTicketsToServer (fid : String, sql : String,) : boolean
insertFlightToCEA (flight : Flight,) : boolean
insertFlightToDB (flight : Flight, sql : String,) : boolean
insertFlightToAC (flight : Flight,) : boolean
insertFlightToQan (flight : Flight,) : boolean
queryFlightListFromQan (fromCity : String, toCity : String, leaveDate : Date,) : List
queryFlightListFromCEA (fromCity : String, toCity : String, leaveDate : Date,) : List
queryFlightListfromAC (fromCity : String, toCity : String, leaveDate : Date,) : List
queryFlightListFromSrver (fromCity : String, toCity : String, leaveDate : Date, sql : String, fid_column_name : String,) : List
util (from fb)
DBUtil
driver : String
url : String
user : String
password : String
getConn () : Connection
closeAll (rs : ResultSet, stmt : Statement, conn : Connection)
close (rs : ResultSet)
close (stmt : Statement)
close (conn : Connection)

Figure 6: Class diagram of airline sever

The proxy server is the core part of this flight booking system, because all clients should communicate with the airline servers via the proxy server, that is to say, the proxy server handle the messages passing from clients and distribute them to different airline servers. To separate this two different business logics, two HOPP are implemented in this part, the PSHOPP used to handle the requests and response, PCHOPP used to handle the requests processed by PSHOPP and distributed them to the airline servers. Figure 7 shows the class diagram of proxy server.

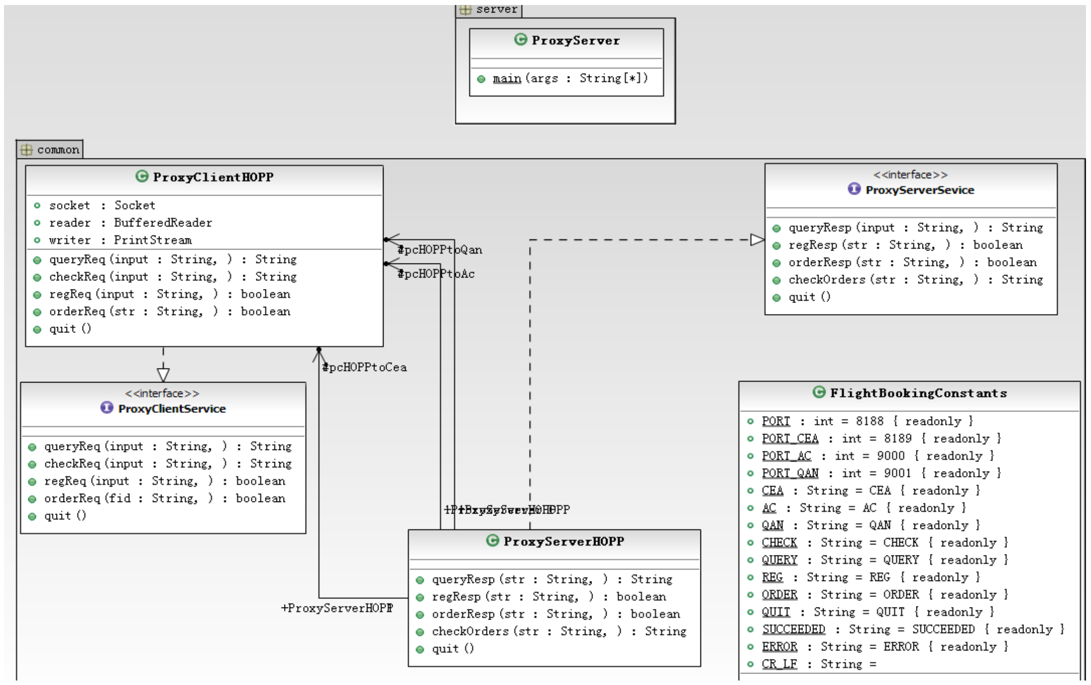

server
ProxyServer
main (args : String[*])
common
ProxyClientHOPP
socket : Socket
reader : BufferedReader
writer : PrintStream
queryReq (input : String,) : String
checkReq (input : String,) : String
regReq (input : String,) : boolean
orderReq (str : String,) : boolean
quit ()
#pcHOPPtoQan
#pcHOPPtoAc
#pcHOPPtoCea
<<interface>>
ProxyServerSevice
queryResp (input : String,) : String
regResp (str : String,) : boolean
orderResp (str : String,) : boolean
checkOrders (str : String,) : String
quit ()
<<interface>>
ProxyClientService
queryReq (input : String,) : String
checkReq (input : String,) : String
regReq (input : String,) : boolean
orderReq (fid : String,) : boolean
quit ()
+ProxyServerHOPP
ProxyServerHOPP
queryResp (str : String,) : String
regResp (str : String,) : boolean
orderResp (str : String,) : boolean
checkOrders (str : String,) : String
quit ()
FlightBookingConstants
PORT : int = 8188 { readonly }
PORT_CEA : int = 8189 { readonly }
PORT_AC : int = 9000 { readonly }
PORT_QAN : int = 9001 { readonly }
CEA : String = CEA { readonly }
AC : String = AC { readonly }
QAN : String = QAN { readonly }
CHECK : String = CHECK { readonly }
QUERY : String = QUERY { readonly }
REG : String = REG { readonly }
ORDER : String = ORDER { readonly }
QUIT : String = QUIT { readonly }
SUCCEEDED : String = SUCCEEDED { readonly }
ERROR : String = ERROR { readonly }
CR_LF : String =

Figure 7: Class diagram of Proxy

The client part in this project is mainly responsible for providing a user interfaces and interacting with users, that is to say, users send request to the servers using this part. In fact, it is easy to develop the client, because there are no complex logics in this part. The FlightBookingService Interface used to specify the communication logics using HOPP and input checking logics using regular expression. Figure 8 shows the class diagram of the airline servers.

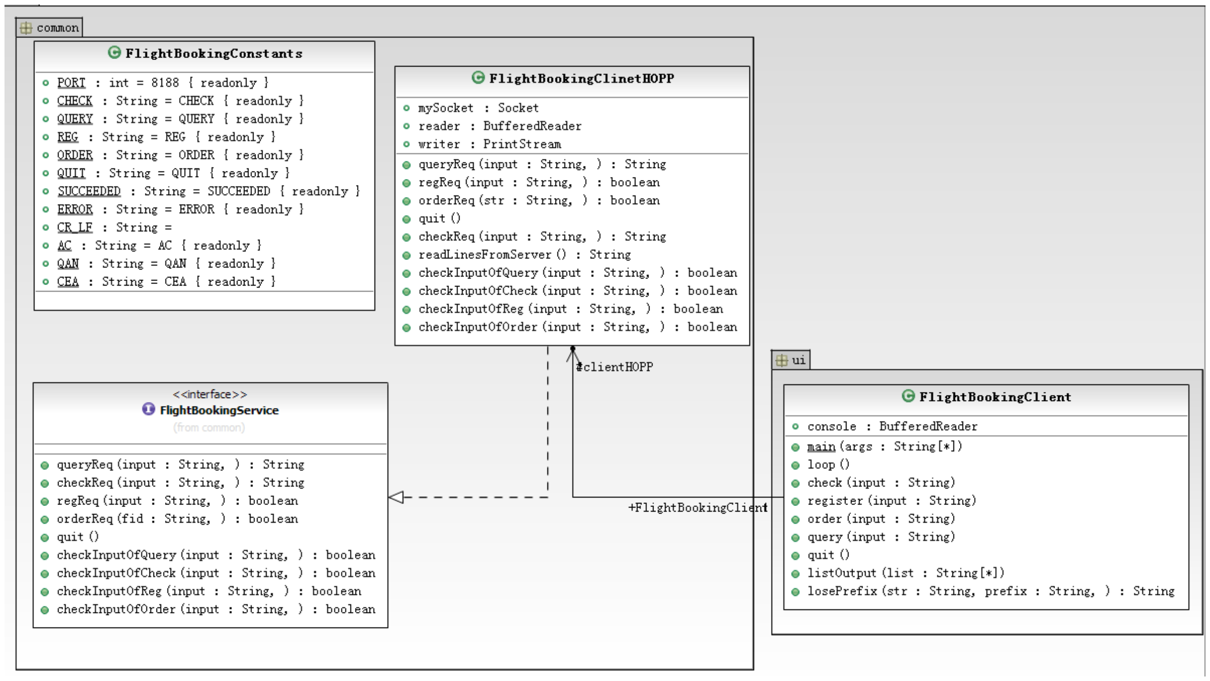

common
FlightBookingConstants
PORT : int = 8188 { readonly }
CHECK : String = CHECK { readonly }
QUERY : String = QUERY { readonly }
REG : String = REG { readonly }
ORDER : String = ORDER { readonly }
QUIT : String = QUIT { readonly }
SUCCEEDED : String = SUCCEEDED { readonly }
ERROR : String = ERROR { readonly }
CR_LF : String =
AC : String = AC { readonly }
QAN : String = QAN { readonly }
CEA : String = CEA { readonly }
FlightBookingClinetHOPP
mySocket : Socket
reader : BufferedReader
writer : PrintStream
queryReq (input : String,) : String
regReq (input : String,) : boolean
orderReq (str : String,) : boolean
quit ()
checkReq (input : String,) : String
readLinesFromServer () : String
checkInputOfQuery (input : String,) : boolean
checkInputOfCheck (input : String,) : boolean
checkInputOfReg (input : String,) : boolean
checkInputOfOrder (input : String,) : boolean
#clientHOPP
<<interface>>
FlightBookingService
(from common)
queryReq (input : String,) : String
checkReq (input : String,) : String
regReq (input : String,) : boolean
orderReq (fid : String,) : boolean
quit ()
checkInputOfQuery (input : String,) : boolean
checkInputOfCheck (input : String,) : boolean
checkInputOfReg (input : String,) : boolean
checkInputOfOrder (input : String,) : boolean
+FlightBookingClient
ui
FlightBookingClient
console : BufferedReader
main (args : String[*])
loop ()
check (input : String)
register (input : String)
order (input : String)
query (input : String)
quit ()
listOutput (list : String[*])
losePrefix (str : String, prefix : String,) : String

Figure 8: Class diagram of client

1.5 Conclusions

In conclusion, a distributed flight booking system is developed in this assignment, users could use some commands that specified by the system to use this system. Four main functions are developed in this system, including flight querying, user registering, flight booking, and orders checking.

For the purpose of implementing the techniques that taught in class to improve the system, design patterns such as MVC, DAO, and HOPP are used in this system, and text protocols used to afford interfaces for someone else to expanding this system. Further, this flight booking system is capable of handling three airline servers concurrently. Therefore, this flight booking system could meet the requirements of the assignment.