Comp 512 P1 Travel Reservation System Backend

1 RMI Architecture Description

1.1 Client

The RMI Architecture consists of a Client Server, a Middleware Server, and Resource Managers.
The client here is responsible for establishing a connection to the middleware and passing use commands to the middleware server.
When the client is launched, it will start looking for the remote object (stub), this action will be waiting if the remote object cannot be found. The user interface to input commands will be provided when connecting successfully. User commands are parsed and sent to the middleware server for further execution.

1.2 Middleware

The middleware server creates the remote object for the client to perform operations. The middleware is also responsible for customer operations and storing customer objects. The other objects (Flight, Room, Car) can be seen as stubs as they are located on different remote servers, only references can be seen by the middleware to perform related operations. When the middleware accepts the incoming messages from the client, it will extract them and separate *Customer* operations from Flight, Car, and Room operations. The other operations will be forwarded to the corresponding Remote Servers. For example, a user invoked the ReserveFlight() operation. Then, our Client forwards the message to the middleware Server. The middleware Server updates the actions on the particular customer (add reservation) and then forward the Flight actions (update flight seat) to the corresponding Resource Manager server.

1.3 Resource Server Hosts

These are the resource manager servers. Each of them is responsible for the operations of their specific data types. The RMI Resource Manager will create the remote object and register it in the registry. It will accept the messages from the middleware stub, performs actions inside the resource server, and returns the result to the stub on the middleware server.

2 TCP Architecture Description

2.1 Structure

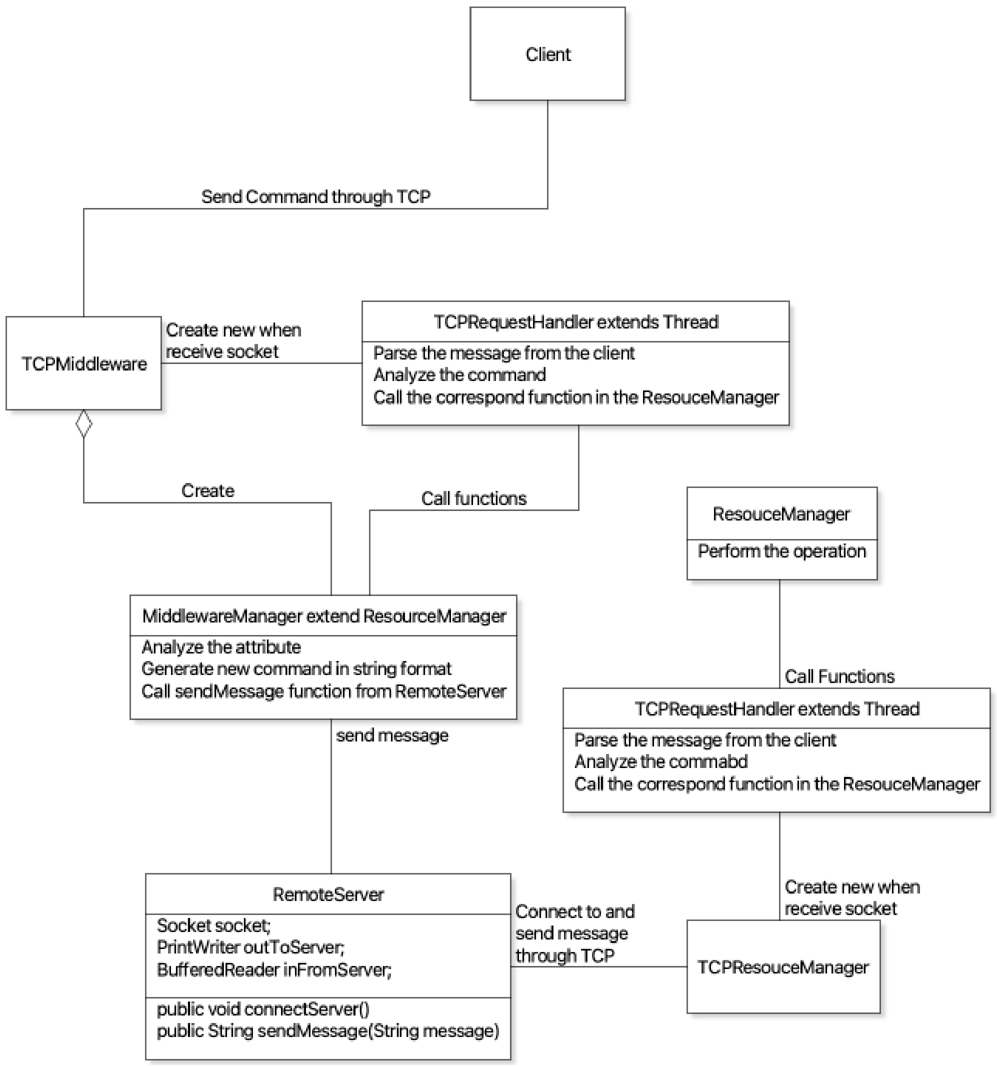
Client
Send Command through TCP
TCPMiddleware
Create new when receive socket
TCPRequestHandler extends Thread
Parse the message from the client
Analyze the command
Call the correspond function in the ResouceManager
Create
Call functions
ResouceManager
Perform the operation
MiddlewareManager extend ResourceManager
Analyze the attribute
Generate new command in string format
Call sendMessage function from RemoteServer
Call Functions
TCPRequestHandler extends Thread
Parse the message from the client
Analyze the commabd
Call the correspond function in the ResouceManager
send message
RemoteServer
Socket socket;
PrintWriter outToServer;
BufferedReader inFromServer;
public void connectServer()
public String sendMessage(String message)
Connect to and send message through TCP
Create new when receive socket
TCPResouceManager

Figure 1: Overview

The figure above illustrates the overall design logic. The main idea is to pass commands through the TCP sockets as string messages from the client to the middleware and at last to the remote servers. First, the client sends messages as strings to the middleware. The middleware parses the string, analyzes the command, and calls the corresponding functions. These functions generate new commands based on the received command and send them to the respective remote servers (Flight server, car server, room server).

2.2 Concurrency

To handle concurrency, we introduce the TCPRequestHandler class which extends java thread class. An instance of it is created and launched whenever TCPResouceManager or TCPMiddleware accepts a new socket from their clients. It is responsible to receive the message, parse it and call the corresponding function in its attribute, the ResouceManager, which performs the operation for the client. In terms of middleware, the ResourceManager in the TCPRequestHandler is a a subclass of ResourceManager named MiddlewareResourceManager, which implements the functions that generate new commands and pass them to the remote servers. In this way, the functionality between middleware and remote servers are separated while using the same code to handle message parsing and concurrency.