Zachary Siciliani (260845463)

Eric Pelletier (260895863)

Michael Hao (260795549)

COMP 512

Programming Assignment 1: Distributing an Application

Introduction

This assignment involved modifying a pre-existing application of a travel reservation system to create a multi-client, multi-server implementation. The system already has built-in functionalities to run a single-server, single-client system using RMI for communication. It allowed the user to perform tasks such as adding, deleting, or querying reservations for flights, rooms, and cars. For the first portion, the single-client, single-server RMI implementation was to be modified by adding a Middleware without modifying the client. The second task involved recreating the system using TCP sockets rather than RMI.

RMI

The main task involved in the RMI section was the designing of a Middleware. The Middleware implements the same interface as the existing ResourceManagers. In order to get the client’s request to pass by the Middleware, we created a new RMIMiddleware class that is very similar to the RMIResourceManager class but utilize the Middleware object to be initialized as the server. The Middleware is then bound to the registry and will be able to acquire all of the client’s requests. To implement the Middleware, we created a Middleware class in the Common folder that implemented the IResourceManager Interface. Inside of the Middleware constructor, we initiate 4 RessourceManagers for the Flights, Cars, Rooms and Customers respectively. The Middleware utilizes these ResourceManagers by calling them when there respective ‘resource’ is needed. For example, when the client wants to add a flight, after handling the client’s request, the Middleware will invoke the Flight ResourceManager to process the command. After, the completion status is returned to the Middleware and then returned to the Client. With this implementation, it makes it harder to handle requests that need to utilize multiple resources such as the reserve a car command since the process of reserving a car requires to check if the customer is valid and if the item exist as well. To handle this, the Middleware initially makes a request to the Customer ResourceManager to check if the customer is valid. Then, it makes a request to the Car ResourceManager to check if the car item is valid. Finally, it calls both the Customer and Car ResourceManagers to update the customer’s reserved items and the car item respectively.

Furthermore, a new bundle functionality was implemented. This function allows a client to reserve a bundle of flights, rooms, and/or cars all at the same time. The bundle function makes use of the existing functionalities for booking and combines them to simplify the process for the client. Before booking an item in the bundle, the Middleware checks if all the items exist by calling the Customer, Flight, Car and Room ResourceManagers. These checks ensure that the bundle will not be partially booked.

TCP

For the TCP implemention, we created a TCPClient class similar to the RMIClient class but it uses TCP Sockets to connect to the server. We initially establish connection to the server socket at the provided port and get the corresponding input and output streams. To ensure that the client stays connected to the Server, we set the KeepAlive value of the socket to true. We then start our TCP implementation of the Client class that executes a while loop reading the command line prompt from the user. The command is then identified using the same switch cases as the RMI Client implementation. However, instead of passing a RMI, the client write a message to the server socket using the output stream. Since all messages are passed using strings, to build the message, we start with the java function name followed by the required parameters separating each word by a ‘space’. For example, if the user enters the command: ‘AddCars, 1, 10, 20, 100’ the corresponding message sent by the client will be ‘addCars 1 10 20 100’. Using this, we will be able to read the messages easily on the server side. After send the message, the client will block and read the message sent back by the server using a Buffered Reader of the input stream. Following the read, the client will wait until another command prompt is entered by the user.

For the Server, we started by created a TCPMiddleware class that extended a MiddlewareTCP class (a modification our Middleware class but without RemoteExceptions) and used it to initialize the server using TCP Sockets. We did this by creating a new Server Socket at the provided port and then waited for a client using a TCP socket to the connect to the same port. Once a client is connected, the server accepts the connection and creates a new TCPMiddlewareThread to handle all requests incoming from this client. This allows the Middleware to handle several clients at the same time without breaking any connections. In the TCPMiddlewareThread, the server waits until the client sends a message through the Input Stream. We read the message using a Buffered Reader and reading one line at a time. We are allowed to do this since we know before hand that the client’s message will be at most a single line. If the message read is equal to “Quit” the corresponding socket will be closed and the connection to the client will be terminated. If not, the message is parsed into commands. Since we know the format of the client’s messages, we are able to match them against a set of switch cases in order for the Middleware to send the request to the correct ResourceManager. The ResourceManagers are then also able to handle multiple requests concurrently because of the Middleware threading. It can listen to a request from one thread while handling the request of another.

The general structure of the TCP implementation can be seen in the diagram in Figure 1. The Middleware connects the clients with a designated thread, which can then communicate with the ResourceManagers.

Diagram

Description automatically generated

Figure 1: The structure of the TCP implementation

Custom Functionality

For custom functionality, we chose to implement the analytics and summaries capabilities. While it was already possible to easily query information regarding a single customer, it was not easy to view information about the system as a whole, which would be very useful for someone trying to use it. We therefore implemented capabilities to track overall summaries of reservations and availabilities, to make it more useful and similar to how a comparable real-world system would work.

Division of Work

Eric was responsible for the RMI middleware implementation. All group members worked on the TCP functionality. Zachary and Michael worked on the custom functionality and analytics. All group members worked on the report.