School of Computer Science, McGill University

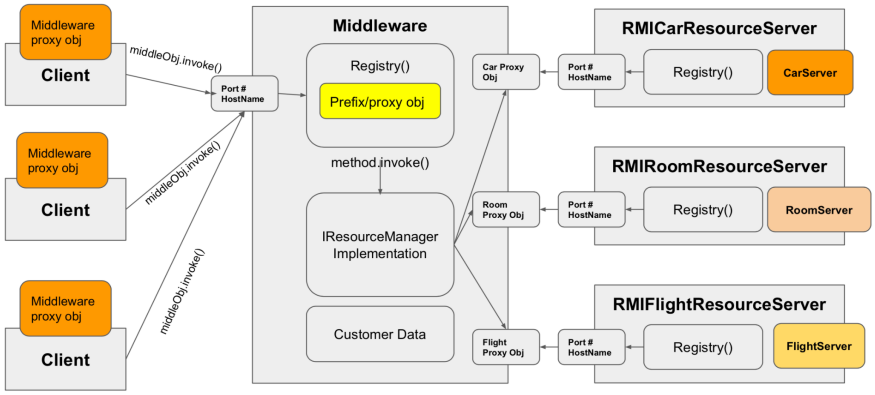
COMP-512 Distributed Systems

Project 1: **Distributing a Travel Registration System**

Group04: Mian Hamza - Jack (Jingxu) Hu - Elvin Zhang

RMI

The RMI architecture of the travel registration system is broken down into 3 parts, namely RMI Client, RMI Middleware, and RMI Resource Servers. RMI stands for Remote Method Invocation which is a multi-client/server distribution technique for organizing a large software system that requires a plethora of transactions between many different active clients and servers. The architecture requires both middleware and resource servers to implement the same interface functionalities as this allows the remote invocation of server side methods through the use of server registries that contain server proxy objects. When a resource server starts up it first exports a rmi registry on a specific server host name and port number. Then it exports it's own proxy object into the registry by binding it with a specific name/prefix. The server host name, port number, and proxy object prefix will be used by any client who wishes to connect with it and execute RMIs on the server side for processing. In reality, clients do not communicate directly with the resource servers thus a middleware server would store the proxy objects of each of the resource servers (after connecting with their specific registries using host name and port number) and delegates each RMI call from the client to the appropriate resource server proxy object by performing RMI on them.



TCP

Architecture

The TCP Architecture involves creating a socket connection from the client to the Middleware layer, this connection will remain until the termination by the client. The Middleware then creates connections to the Resource Servers on demand. That is, whenever the client requests something from a particular Server, the Middleware will create a connection with the target Server, and will terminate the connection upon completion of the particular request.

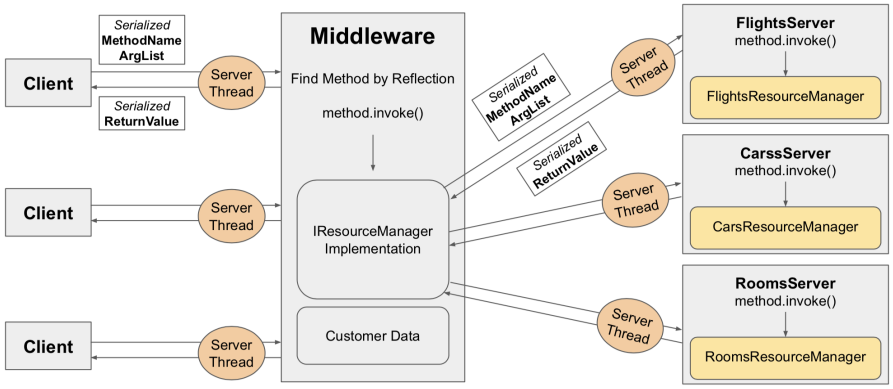
Reflection and Message Passing

In the TCP part, when executing a command, a client sends the function name of the corresponding function as a serialized string object through *ObjectOutputStream* to the middleware, as well as the parameter list as a serialized objects array. The Middleware receives the name and parameter list through *ObjectInputStream*, then finds the method by the name and parameter list through reflection mechanism. It passes the parameter list and invokes it, and then sends the serialized return value object back to the client. Therefore, the client completes an invocation of a function on the Middleware. When the Middleware needs to call functions on ResourceManagers, it sends the serialized function name and parameter list to the corresponding resource server. The resource server finds the function on its ResourceManager, executes the function and sends back the result to the Middleware. By making classes such as Customer, RMItem implement Serializable Interface, we can also pass these objects as parameters between the Middleware and Resource Servers.

Concurrency

The Middleware creates a thread and a threaded socket when a client connects to it. When receiving the method name and parameter list from a client, the threaded Middleware calls the function by Java reflection. When the function is to request something from a Resource Server, e.g. *addFlight*, *reserveCars*, the Middleware creates a socket to connect to the target server, and calls the target function in the same way as a client calls function on Middleware.

The Resource Server code also consists of two elements, the main server file that creates the threads, and the Threaded ServerSocket. The main server file creates an object ResourceManager identical to the one in the RMI part. The RM handles all of the data and the functions of the server. The threaded Server receives the method name and parameter list, and invokes the function on RM, and then returns the results to the Middleware. The Middleware, upon receiving the result from the Server, will close the socket with the Server, and return the value to the Client. Functions that write data to the hashmap are synchronized, to avoid multiple threads entering the critical region and causing errors.



Implementation Intricacies

Handling of Customers

We store customers in a hashmap and handle customer functions in the Middleware, because 3 ResourceManagers should share 1 copy of customers, and the crash of resource servers will not affect the operations on customers, e.g. creating new customers.

Implementation of Bundling

We added a function in the interface and implemented it in FlightsResourceManager, to reserve flights by a list of flight numbers. In function 'bundle', the Middleware checks the customer ID, calls that function to reserve flights, calls 'reserveCars' if the parameter cars is true, and then calls 'reserveRooms' if the parameter rooms is true.

Synchronization

Since the Middleware allows multiple clients to connect it, methods that were related with writing data to the Hashmap of the Middleware or a ResourceManager has to be synchronized to become thread-safe. For example, when a client reserves a flight, the RM reads the flight from the hashmap, decreases the number of seats and then writes it back. When 2 clients call reserveFlight at the same time, 2 threads will enter the critical region, both decrease the number of seats by 1 and write the same number back. The number of seats will be incorrect. (Tested it by putting *Thread.sleep(5000)* between readData and writeData.) Thus, we made functions e.g. reserveItem, deleteItem, deleteCustomer synchronized to avoid data races.

Custom Functionality

JAVA Reflection and Serialization

Java reflection allows us to obtain and invoke a function by the function name and the parameter list. Serialization allows us to send the function name and parameter list as serialized objects conveniently through sockets between clients, the Middleware and servers. By these two techniques, our clients can execute methods in the Middleware and

servers, then get the result. Thus it helped us to directly reuse the code of ResourceManagers and the Middleware in the RMI part since the implementation worked perfectly. (See: **TCP - Reflection and Message Passing**)

Extra Functionality

1. QueryReservableItems

We added functions in ResourceManagers to query reservable flights/cars/rooms. The Middleware calls the function in 3 ResourceManagers, combines the results and returns to clients.

2. QueryFlightReservers / QueryCarsReservers / QueryRoomsReservers

Traverse the customer hashmap in the Middleware. For each customer, traverse the reserved item list, add it to the result list if it's a flight/car/room item.

Contribution

Elvin Zhang: 1. Built the RMI architecture that connects clients, Middleware and 3 ResourceManagers.
2. Implemented customer functions, Bundle function, and the extra QueryReservableItems function.
3. Tested the concurrency and handled the race condition by synchronization.
4. Used Java Reflection to make method invocation in the Middleware and Resource Servers, to implement all functionalities in the TCP part without extra modification on ResourceManagers.

Jack Hu: Created/refactored the RMI template code and distributed server side functionalities to include middleware and resource servers. Created client and server bash scripts for easy invocation of project code. Created a rough version of TCP distributed architecture that reused code from RMI (resource servers, middleware, and client) with server sockets and data input/output streams. Implemented server code with multi-threaded design in order to account for many clients trying to connect to its hostname and socket port number at the same time. You may examine my branch's history here: https://github.com/jhu960213/COMP_512_Projects/tree/dev_jack

Hamza Mian: Worked on RMI Portion using the template code, and then consolidated all of the working RMI portions from team members into a single RMI implementation that the team built upon. Created the customer creation functions for Middleware, and worked on the individual Server RMI implementations. Created the base TCP implementation, and created the working architecture that uses ResourceManager. Also created the Middleware layer, as well as the socket communication functionality between Middleware, Server and Client. Created and tested a working TCP implementation, and made the accompanying Makefile. The team then added the finalized functionality to my TCP implementation.