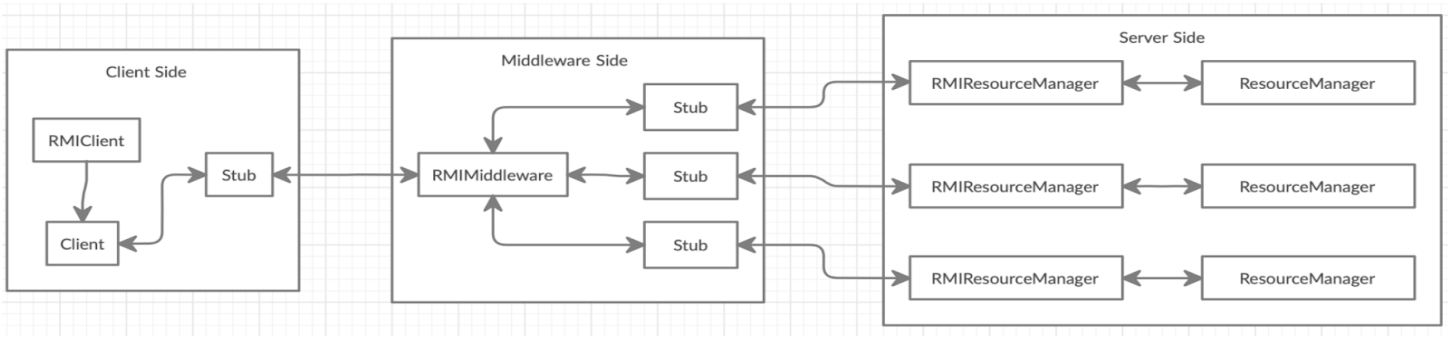
Project Part 1: Distributing an application
COMP512 Distributed System

Group8: Zhang Yong, Eric Shen, Winnie Tan

RMI Architecture:

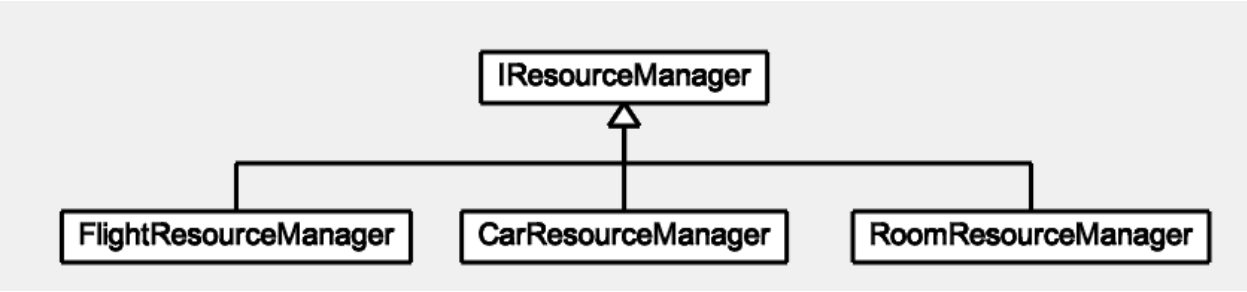


As stated in the requirements, the RMI Architecture mainly consists of three parts: the Client part, the RMIMiddleware part, and three different servers (basically resource managers) parts.

First, at start-up, the client would try to search the middleware given its host name and its port number by looking up the remote object of the middleware. The implementation class *RMIClient* would not be going to process the input until it finds the middleware. Then, the *Client* class will parse the command provided by the user. Once the command matches the correct format, the Client would call the corresponding method on the remote object, which is middleware in this case.

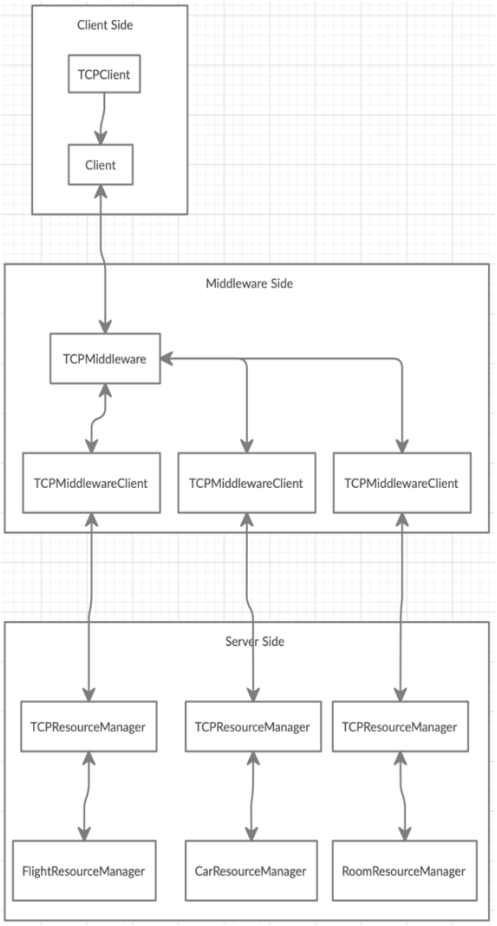
The Middleware is like a router and also a server. At the very beginning, the *RMIMiddleware* will export itself as a stub for clients to find and then search the three specific resource manager servers. Once found, it is ready to accept the requests from clients and navigate them to corresponding resource manager servers. For instance, the addFlight operation would be forwarded to *FlightResourceManager*. In our design, the *RMIMiddleware* is also a type of customer resource manager since it handles the requests related Customer.

For the server side, they are the resource managers that handle the clients' request and change the database as well. To split the functionalities of the server, there are Flight, Car, and Room resource managers. Each time the server starts, it registers itself at the registry. The *RMIMiddleware* can forward the request to a specific server by designating the command to a remote object of the server, and the server would handle the request, sending the response back to the middleware.



TCP Architecture:

The architecture of TCP is similar to RMI from a high-level abstract perspective. It mainly consists of three parts: Client, Middleware, and three different resource managers.



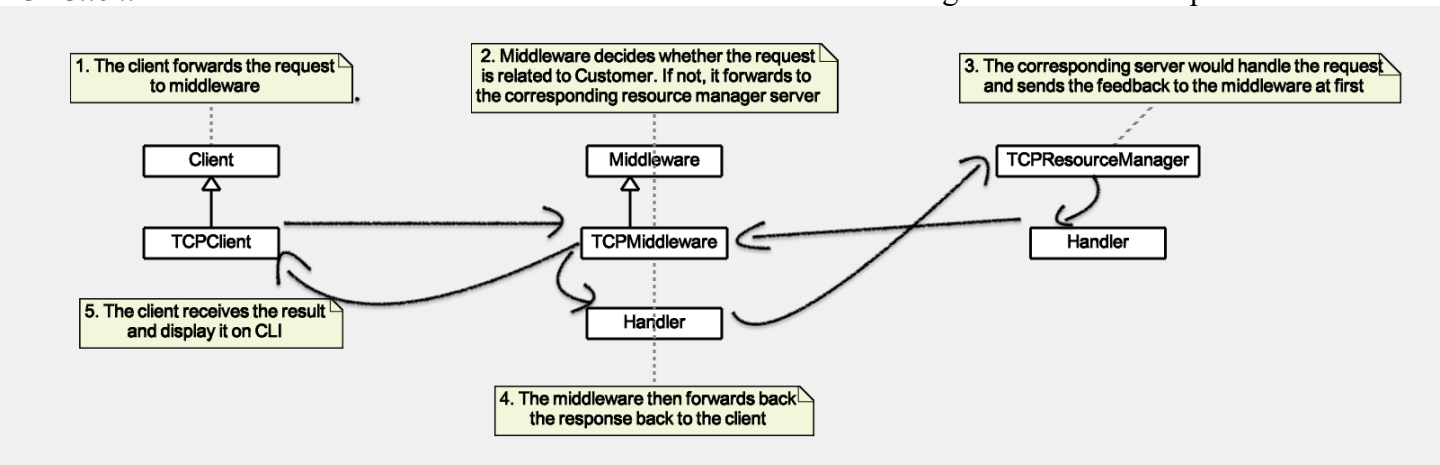
For the Client side, it takes the input from the user via a command-line interface. Before sending it to the *RMIMiddleware*, the client host checks whether the input matches one of pre-defined operations by parsing the command into the correct format. From then on, the client is going to send the command with arguments via socket by delivering the command to *TCPClient* class. The client host would remain blocking until it receives the feedback from the middleware. Once finished, the client side is ready for processing the next operation.

For the Middleware side, it could be regarded as a client and server at the same time. On one side, it handles the requests that are related to Customer from the clients and sends the feedback. On the other side, it also forwards the other requests to the specific servers. Every time a client request comes, a new thread, *TCPMiddlewareHandler*, which is also a handler for the incoming request, would be created. In doing so, the middleware is able to receive the inputs without needing to let other clients wait.

On the Server side, once it starts, it is going to create a ServerSocket and continuously listen to a port. Then, the middleware will try to connect the server by using this specific port. Same as the middleware, every time it is going to create a thread to handle the request.

Message Passing:

In TCP, unlike RMI, we cannot directly invoke the corresponding method on the registered remote objects if we want to call remote service. All of our messages are serialized and sent back and forth via network. As stated in the template code, an incoming string needs to be parsed into the correct command format before it gets executed on the server or middleware side. This has been done by the *parse* method implemented in some classes. The overall message or call flow is represented below.



Concurrency:

Aside from the synchronization of data stored in the resource manager, the synchronization of the resource manager itself is also being enforced when read and write operations are happening at the same time. That is to say, at any time, a specific resource manager could only serve one client. This would ensure that for instance if

one client is trying to reserve a flight while the price of it is changing, the client would only wait until the price is stable or before the change happens.

Functionality:

Instead of creating a new tier for handling the customers, we decided to handle the customer specific requests directly at middleware. In doing so, it will reduce the amount of work for retrieving the customer info and a rewinding path for synchronization of data for different resource managers, which may result in data replication.

For bundle command, there is a possibility that certain reservations would fail. In our design, we try to make the customer reserve as much as he can until there is a reserve failure. Thus, every specific reserve is atomic, and the feedback is provided for each atomic reserve.

Custom Functionality:

Research on serialization and reflection:

By serialization as well as deserialization of the arguments, data can be converted into a byte stream, which can be stored on the local disk and sent over to the client side in TCP distribution. The results of RMI are even carried out automatically by the middleware through deserialization. The programmers may not need to write their own read and write object methods to achieve the goal.

In java, the serialization process includes the class information as well. Reflection supports the ability to enquire about the properties of a class (the names, types and instance etc.) which can be used by serialization. Then there is no need to create specific marshalling functions for the remote objects on the server side.

Extra client functionality:

By adding new functionality to the resource managers, analyticsFlight, analyticsRoom and analyticsCar are available to clients to check the items with low remaining quantity. The customer summaries function can report the reserved item for each resource type (flights, rooms and cars)