Distributed Application

RMI Architecture

The RMI Architecture consists of 3 parts: the Client host, Middleware Server host, and the Resource Manager Server hosts.

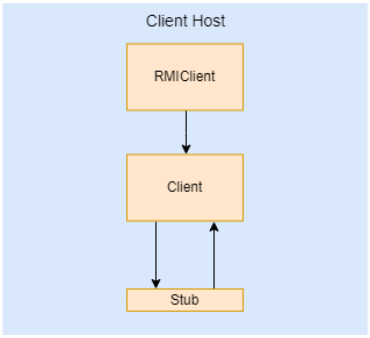

Figure 1

The Client host (Figure 1) is responsible for connecting to the middleware server and executing client commands. Initially, *RMIClient* locates the registry of the middleware server and performs a lookup of the *IResourceManager* remote object (if there are registry or lookup failures, the client waits until these actions can be performed). Afterwards, the *Client* is started, which provides a command-line interface. Upon the user entering in a valid command, the *Client* invokes the appropriate message on the stub object. The stub object makes a message and sends it to the skeleton object residing on the middleware server.

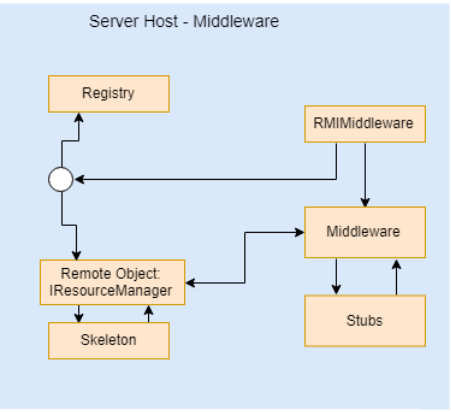

Figure 2

The Middleware Server host (Figure 2) has 2 responsibilities. First, the *RMIMiddleware* creates the *IResourceManager* remote object (which is implemented by the *Middlware* class) and registers it at the registry. Thus, the Client host can now lookup the *IResourceManager.* When the stub object on the Client host sends a message to the skeleton object on the server, the skeleton unmarshals the message and passes the object to the remote object (the *IResourceManager* which is implemented by *Middleware*). In our implementation, *Middleware* is a resource manager itself, and it handles and stores all information regarding the **Customer** data type. Second, the *RMIMiddleware* locates the registry for each of the Resource Manager Server hosts and obtains references to the *IResourceManager* objects for the *Middleware*. Thus, the *Middleware* has access to 3 stubs, as the *Middleware* needs a remote reference to the **Flight**, **Car**, and **Room** Resource Managers. When a message is received, the *Middleware* performs any **Customer** actions and forwards any **Flight**, **Car**, and **Room** actions to the appropriate Resource Manager server (using the stubs).

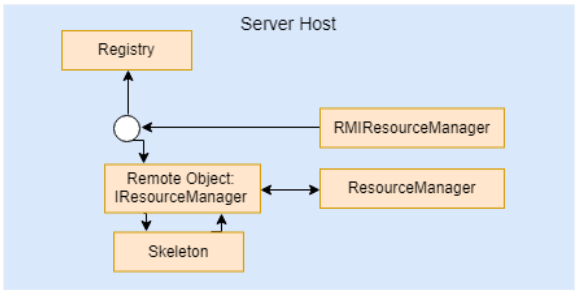

Figure 3

Lastly, the Resource Manager Server hosts (Figure 3) are responsible for performing actions related to their data types. There are 3 Resource Manager Servers, one for the **Flight**, **Car**, and **Room** data types. For each server, the *RMIResourceManager* creates the *IResourceManager* remote object (implemented by the *ResourceManager* class) and registers it at the registry. Now, the Middleware server can lookup the remote objects for the **Flight**, **Car**, and **Room** data types and use the remote objects to perform the desired client action. For example, when the **Flight** stub sends a message to the **Flight** Server host, the skeleton unmarshals the message, the *ResourceManager* performs the action and the skeleton returns the result to the stub.

TCP Architecture

The TCP Architecture consists of 3 parts: the Client host, Middleware Server host, and the Resource Manager Server hosts.

The Client host is responsible for sending messages to the Middleware Server with specific client commands to execute. The *Client* provides a Command-line interface for the user to use, and it ensures that the command is of the correct format and that the appropriate arguments have been provided. The *Client* uses the *TCPClient* to send the message. The *TCPClient* opens a client socket, using the host and port of the Middleware Server socket. With this connection, the message is sent to the Middleware to be parsed. The sending of a message is blocking, and thus, the *Client* execution waits for a message to return (or an exception to occur) before resuming execution.

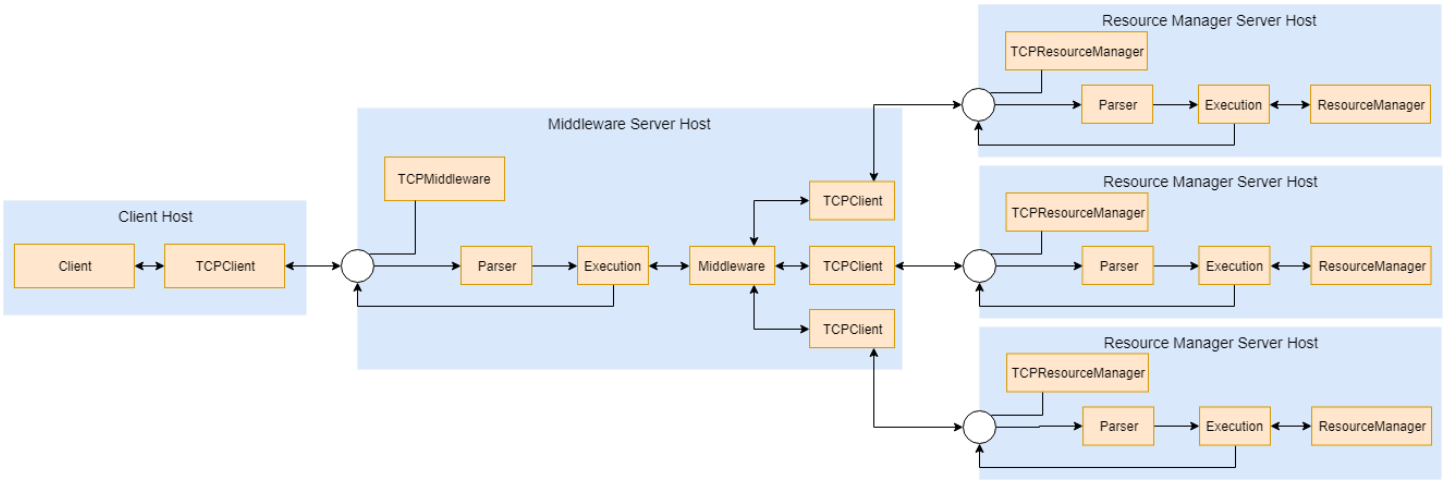

Figure 4

The Middleware Server host provides a server socket endpoint for the Client host, manages the **Customer** data type using the *Middleware* class extension of the *ResourceManager*, and, if need be, forwards **Flight**, **Car**, and **Room** data type commands to the respective server. The *TCPMiddleware* sets up a server socket to listen for incoming connections at a specific port. Upon an incoming client socket connection, the *TCPMiddleware* starts a new thread, which handles the message. The thread parses the message using the *Parser* class, and the parsed message is then used to execute a *Middleware* command through the *Execution* class. Depending on the command, *Middleware* will handle the command, will forward the message, or a combination of the two will occur. The *Execution* class returns the result, and the result is returned to the Client's *TCPClient*.

The Resource Manager Server hosts provide a server socket endpoint for the Middleware host and manages either the **Flight**, **Car**, or **Room** data type. The *TCPResourceManager* sets up a server socket to listen for incoming connections at a specific port. Upon an incoming middleware *TCPClient* connection, the *TCPResourceManager* starts a thread to handle the requested command. The command message is parsed (*Parser* class), executed (*Execution* class to *ResourceManager*), and returned to the middleware *TCPClient* connection.

Message Passing

In the TCP Architecture, *TCPClients* send messages to the server socket endpoint, and these endpoints parse and execute the commands. In Figure 5, one can see how messages are passed and parsed. The *TCPClient* sends the message as a string of comma-separated inputs (the inclusion of the square brackets is optional). The first input is the command the user wants to execute, and the rest of the inputs are the arguments. The *Parser* parses the command by splitting on the ",". This produces a comma-separated list of strings. The *Execution* class uses the command to know what argument types to expect, and it converts from the string-typed argument to the expected argument type. With the command and the arguments, the *Execution* class calls the appropriate *ResourceManager* or *Middleware* method.

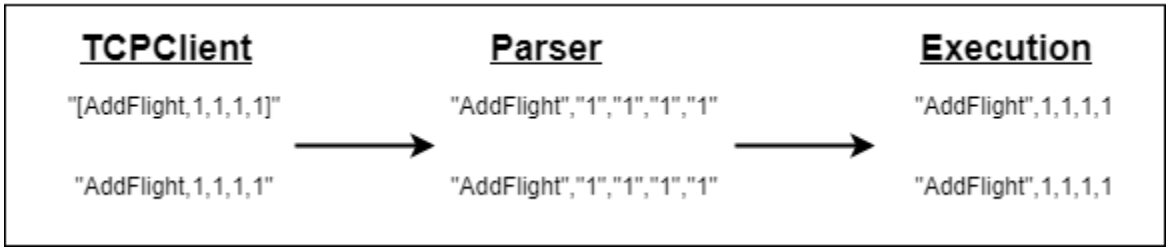

Figure 5

Concurrency

Beyond the synchronization of threads when accessing the HashMap of the data, additional concurrency concerns have been dealt with at the Middleware Server level. For any command that doesn't modify the HashMap of the data

(ex. QueryFlight), the Middleware simply forwards the message to the corresponding Resource Server. Otherwise, for the adding commands, the threads are synchronized on the Resource Server communication object (RMI: remote object, TCP: TCPClient). For deletion and reservations, the threads are synchronized on the customer object and the communication object. For bundles, the threads are synchronized on the customer object and any communication object that will be used to reserve an item.

This concurrency ensures the HashMap data is never corrupted. This setup prevents: the price of a flight changing while a customer is reserving an item, a customer being deleted while reserving an item, the availability of an item changing while a reservation or a bundle is being filled, and a bundle being only partially filled.

Functionality

The **Customer** data type functionality is implemented on the Middleware Server host tier of each architecture. Moreover, the Bundle command is synchronized on the customer and all the communication objects needed. This ensures that neither the customer's details nor the availability of an item changes during the execution of the command. Thus, Bundle is atomic: executed in totality (reserving all items for a specific customer) or not at all.

Moreover, the Custom functionality implemented was the inclusion (for RMI and TCP) of the "Summary" and "Analytics" commands. "Summary" takes in the xid (which is not used), and it outputs a summary of the client-resource allocation. For each client, the resources they have reserved will be printed. "Analytics" takes in the xid (not used) and an upper bound. Any resource with remaining quantities less than or equal to the upper bound will be printed. Thus, "Analytics,1,0" will print all those resources with no remaining items, "Analytics,1,1" will print those with 1 or 0 remaining items, etc.