COMP 512 - Project Phase 1 Report

Jérémie Poisson (260627104), Théo Szymkowiak (260619310)

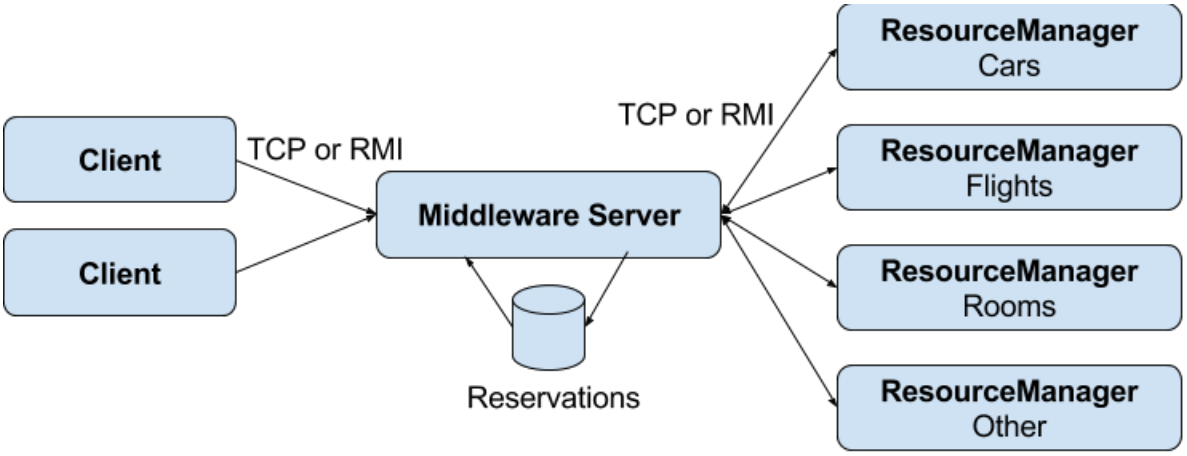
October 1st 2017

1 Introduction

In this phase of the project, our main task was to distribute resources and create a component-based distributed information system. The example used is a travel reservation system.

We were first tasked to implement a middleware server that would span and distribute the different resources of the system across the network over a different physical server. The servers allocated to manage said resources are called "resource-managers" and provide a Remote Method Invocation (RMI) interface for the middleware server layer to interact with. The remote clients would then access the exposed middleware server layer in order to interact with its subsystems adequately.

The figure below displays an overview of the architecture described in this report. The subsequent sections explain our approach in distributing such system.



2 Resource Distribution Strategy

We wanted to distribute the system such that cars, flights and rooms entities would be managed in different resource managers distinctively. We used three (3) distinct resource manager servers, each to manage the flights, cars and rooms entities independently i.e. the *Car Resource Manager* server contains every car available for reservation and every subsequent car-related operations (addCar, deleteCar, etc.) are redirected to this server through the middleware layer, and so on.

On a side note, every resource manager server implements the whole set of operation exposed in the *ResourceManager* interface (accessible through an RMI channel or TCP requests). However, the specific resource managers only use the operations related to the kind of data they hold (car only execute car-related opera-

tions, flights only execute flight-related operations, etc). We plan to only export the required operations by the different resource managers with using several exposed interfaces as a future work.

The *Other* resource manager server is used to store every other kind of resources needed in the system. For now, this only includes the collection of customers using the system.

Because customers may reserve a different kind of reservable items such as cars, flights, and rooms, all of which are located on different physical servers, we need a centralized server to keep track of such reservations. Thus the middleware server layer has an internal customer reservation database that keeps track of every reservation for customers.

In addition, because we need to update the quantity of certain items that only exist on different physical servers, we had to add a new *updateReservedQuantities()* method to the exposed resource managers interface. This method simply updates the quantity of a given item on a server without being hard-wired to a customer directly (since it resides on another server) –the bookkeeping for customers' reservations is done on the middleware server.

When executing operations that require interactions with multiple resource managers (e.g. *reserveItinerary*, which might book flights, rooms, and cars) the middleware provide an abstraction layer that calls every concerned backend resource manager servers.

Below are some key examples of the kind of data stored and where it is stored:

- **List of cars at a given location (location, quantity, price)**: Car Resource Manager Server
- **List of rooms at a given location (location, quantity, price)**: Rooms Resource Manager Server
- **List of flights available (no, seats, price)**: Flight Resource Manager Server
- **List of customers registered in the system (customerId)**: Other Resource Manager Server
- **Reservations for each customers**: Middleware Server

3 Middleware Server using an RMI Implementation

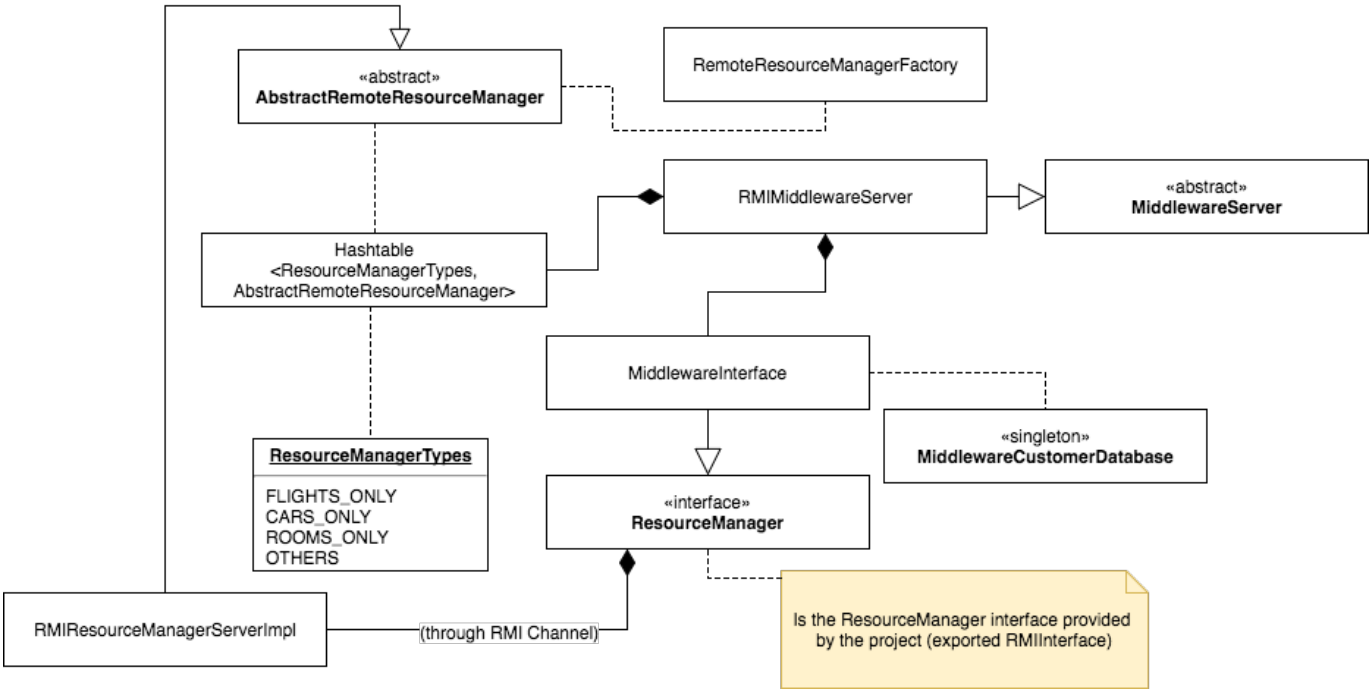
The *MiddlewareServer* is first initialized with a list of RMI server URIs. Upon initialization, a hashtable of *AbstractRemoteResourceManager* is populated for each resource managers (cars, flights, rooms, and others) which then connects to all back-end resource manager servers.

In this RMI implementation, the *MiddlewareInterface* is exported as an RMI object. This object is remotely called by an operating RMI-enabled client, which can use the implemented methods directly.

The *MiddlewareInterface*, which has a handle to the middleware server and every other resource manager servers, executes the operations on the appropriate set of resource managers depending on the requested operation.

From a conceptual point of view, the middleware server exports a layer of abstraction that allows the different kind of resources to be hosted on several physical machines. Because the middleware layer has to keep track of the customers' reservation in a centralized manner and that multiple concurrent connections are possible, the *CustomerReservationsDatabase* is a synchronized object that provides an atomic way of managing every reservation.

Below is a simplified class diagram for an overview of the whole middleware RMI implementation.



4 Socket-based Middleware Server

The implementation of the socket-based middleware uses a simple Java *ServerSocket* along with *ObjectOutputStream* and *ObjectInputStream* in order to transmit object over the network. The way to communicate with the server is by passing two kinds of objects: requests and responses. Basically, one creates a specific request object and passes it to the server over the *ObjectOutputStream*. Upon receiving a valid request the server executes the underlying command of said request and returns a response (wraps it into a valid response object) over the network again using the *ObjectOutputStream*. The client then receives the response and interprets it. This is our basic request/reply protocol.

Once the server socket receives an incoming connection, a new *MiddlewareTCPSession* session is created and listen for incoming client commands in a thread (part of a thread pool). Each *MiddlewareTCPSession* receive incoming requests as a subclass of *TCPRequest*, an abstract class inherited by every requests. Every subclass of requests contains the code specific to the command by overloading the abstract method *executeCommand()* of *TCPRequest*.

The execute command method returns an instance of a *TCPResponse* after being executed. Overall, there are several types of possible responses returned by requests (exception, integer, strings, success/failure, void). The value is simply wrapped within one of those response classes.

Since we had to implement the communication with back-end resource manager servers also with a socket-based approach, we simply extended the approach we used in part 1). Basically, we created a second kind of *AbstractRemoteResourceManager* that uses a *ResourceManagerTCPClient* in order to communicate with the TCP-based back-end resource manager servers. This way the middleware is able to communicate with the socket-based back-end servers easily.

For a client to communicate with the TCP middleware implementation, one simply needs to create a client socket and connect to the server, open up an *ObjectOutputStream* and *ObjectInputStream* to write a *MiddlewareTCPRequest* object over the stream and immediately listen for incoming responses on the *ObjectInputStream*. Below is an example of how a typical client would communicate with the server over TCP.

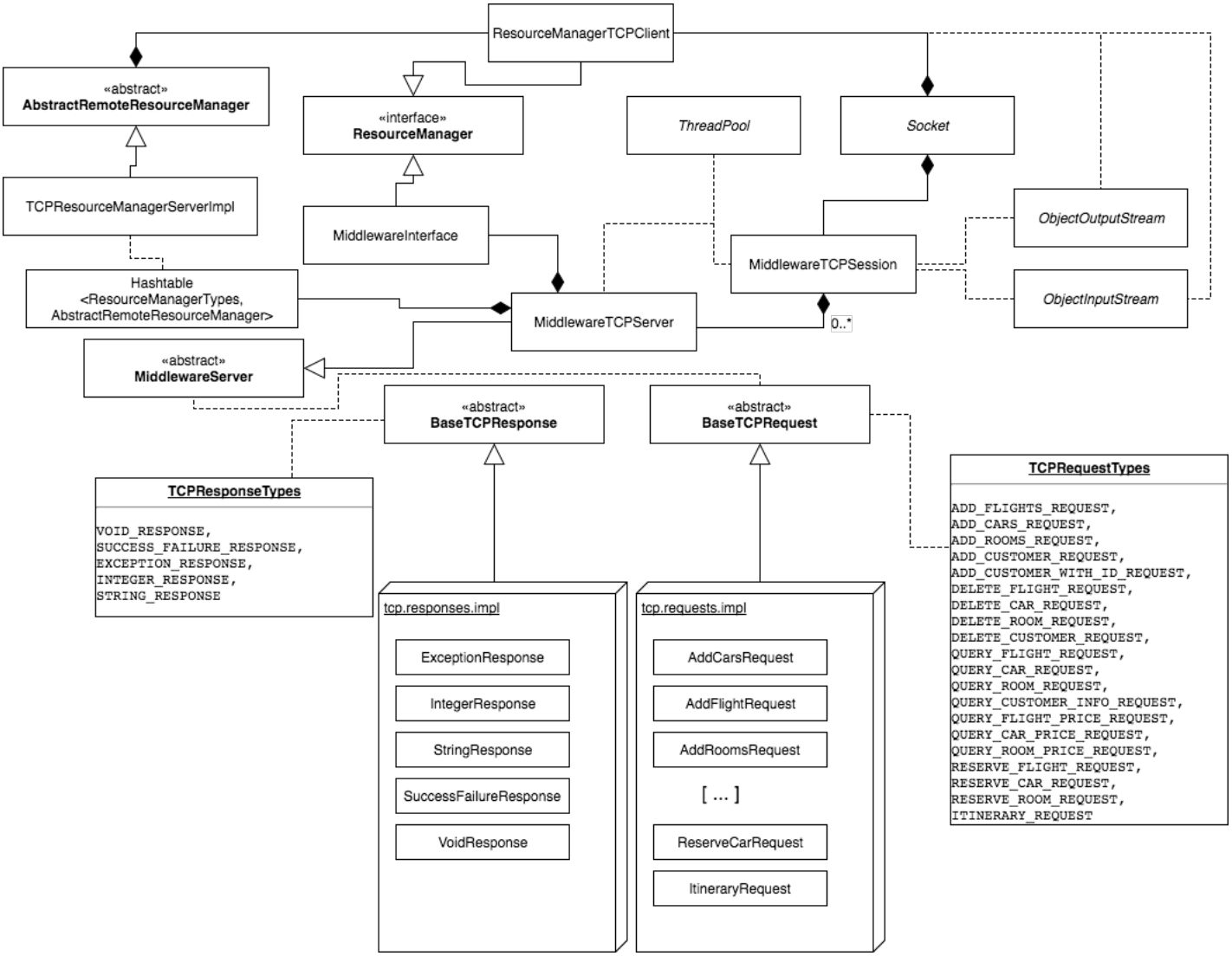
```
static Socket clientSock = new Socket("localhost", 8080);
static ObjectOutputStream objOut = new ObjectOutputStream(clientSock.getOutputStream());
static ObjectInputStream objIn = new ObjectInputStream(clientSock.getInputStream());

static TCPResponse send(TCPRequest req) {
    objOut.writeObject(req);
    Object respObj = (TCPResponse) objIn.readObject();
    if(respObj == null) {
        throw new RuntimeException("Invalid response from server");
    }
    return (TCPResponse) respObj;
}

AddFlightRequest addFlightReq = new AddFlightRequest(10, 10, 100);
TCPResponse resp = send(addFlightReq);
System.out.println("Was successful ? " + ((SuccessFailureResponse) resp).value);

QueryCarPriceRequest queryCarPriceReq = new QueryCarPriceRequest("mtl");
TCPResponse resp = send(queryCarPriceReq);
System.out.println("Car price = " + ((IntegerResponse) resp).value);
```

Below is a simplified class diagram for an overview of the whole middleware TCP implementation.



5 Testing

In order to test our TCP-implementation, we created several test suites using JUnit to cover every exposed system operations as much as possible. Each test suite is composed of several test cases, each of which tests the behavior of exactly one operation with several random inputs and expected outputs. This technique uses our TCP implementation as a black-box and analyzes the system behavior from an external standpoint. Hence every test suite connects to the TCP server using a client Socket, writes to the output stream and listen to the input stream. It creates the appropriate request object with the random parameters and analyzes the responses to validate the appropriate behavior.

Below is an exhaustive list of every test suites and their unit tests:

1. **CarTestSuite**
   (a) testAddQueryCar() : tests adding a car at a random location
   (b) testDeleteCar() : tests deleting a car at a random location
   (c) testQueryCarPrice() : tests querying the price of a car at a random location

(d) testReserveCar() : tests reserving a random car for a random customer

2. **FlightTestSuite**
   (a) testAddQueryFlight() : tests adding a random flight
   (b) testDeleteFlight() : tests deleting a random flight
   (c) testQueryFlightPrice() : tests querying the price of a random at a random location
   (d) testReserveFlight() : tests reserving a random flight for a random customer
3. **RoomTestSuite**
   (a) testAddQueryRoom() : tests adding a room at a random location
   (b) testDeleteRoom() : tests deleting a room at a random location
   (c) testQueryRoomPrice() : tests querying the price of a room at a random location
   (d) testReserveRoom() : tests reserving a random room for a random customer
4. **CustomerTestSuite**
   (a) testNewCustomer() : tests creating a new random customer
   (b) testNewCustomerWithId() : tests creating a new random customer with a fixed id
   (c) testDeleteCustomer() : tests deleting a random customer
   (d) testItinerary() : tests booking an itinerary for 2 random flights with car hotel at a random location

We used Gradle as a build tool for Java in order to compile and execute the test suites. See the README file to run the unit tests.

Here is an example of the testReserveCar() test case. Does the following checks:

- adds a random car;
- adds a customer with a given id;
- reserves the car to the customer;
- makes sure that there is one less car available;
- make sure that the customer has the car on his bill.

```
private static final String RESERVE_CAR_LOC = String.valueOf(TestUtils.newUniqueId());
private static final int RESERVE_CAR_PRICE = 150;
private static final int RESERVE_CAR_NUM = 10;
private static final int RESERVE_CAR_CUST_ID = TestUtils.newUniqueId();
@Test
public void testReserveCar() throws IOException, ClassNotFoundException {
    // add a car
    AddCarsRequest addCarReq = new AddCarsRequest(RESERVE_CAR_LOC, RESERVE_CAR_NUM, RESERVE_CAR_PRICE);
    TCPResponse resp = TestUtils.send(addCarReq);
    assertEquals(SUCCESS_FAILURE_RESPONSE, resp.type);
    assertEquals(true, resp.asSuccessFailureResponse().success);

    // add a customer with given id
    NewCustomerWithIdRequest newCustReq = new NewCustomerWithIdRequest(RESERVE_CAR_CUST_ID);
    TCPResponse newCustResp = TestUtils.send(newCustReq);
    assertEquals(SUCCESS_FAILURE_RESPONSE, resp.type);
    assertTrue(newCustResp.asSuccessFailureResponse().success);

    // reserve the car
    ReserveCarRequest reserveCarReq = new ReserveCarRequest(RESERVE_CAR_CUST_ID, RESERVE_CAR_LOC);
    TCPResponse reserveCarResp = TestUtils.send(reserveCarReq);
    assertEquals(SUCCESS_FAILURE_RESPONSE, reserveCarResp.type);
    assertTrue(reserveCarResp.asSuccessFailureResponse().success);

    // make sure the car has one less avail.
    QueryCarRequest queryCarReq = new QueryCarRequest(RESERVE_CAR_LOC);
    TCPResponse queryCarResp = TestUtils.send(queryCarReq);
    assertEquals(INTEGER_RESPONSE, queryCarResp.type);
    assertEquals(RESERVE_CAR_NUM - 1, queryCarResp.asIntegerResponse().value);

    // make sure the customer has the car on his bill
    QueryCustomerInfoRequest queryCustInfoReq = new QueryCustomerInfoRequest(RESERVE_CAR_CUST_ID);
    TCPResponse queryCustInfoResp = TestUtils.send(queryCustInfoReq);
    assertEquals(STRING_RESPONSE, queryCustInfoResp.type);
    assertTrue(queryCustInfoResp.asStringResponse().value.contains(
        String.valueOf("car-" + String.valueOf(RESERVE_CAR_LOC)))
    );
}
```

Here is a snapshot of all the test passing: