"It's tracking" Term Report
Distributed Programming 2

Eugenio Gallea (s241876)
Stefano Roagna (s241845)

February 2018

Contents

1 Summary

The aim of this project has been the one to design a RESTful web service able to track the precence of vehicles in a area with restricted access and, based on this information, choose if guarantee or reject the access to other vehicles. If the access is granted, the system has to provide the newly-entered vehicle with a suggested path to follow. Otherwise, the vehicle is just rejected by the system with a message.
In order to be able to track the vehicle, the client application has to periodically send information about the current position. At any point in time, the vehicle can decide not to follow the suggested path, in this case the system has to be able to provide the client with a new route, if possible.

2 Architecture

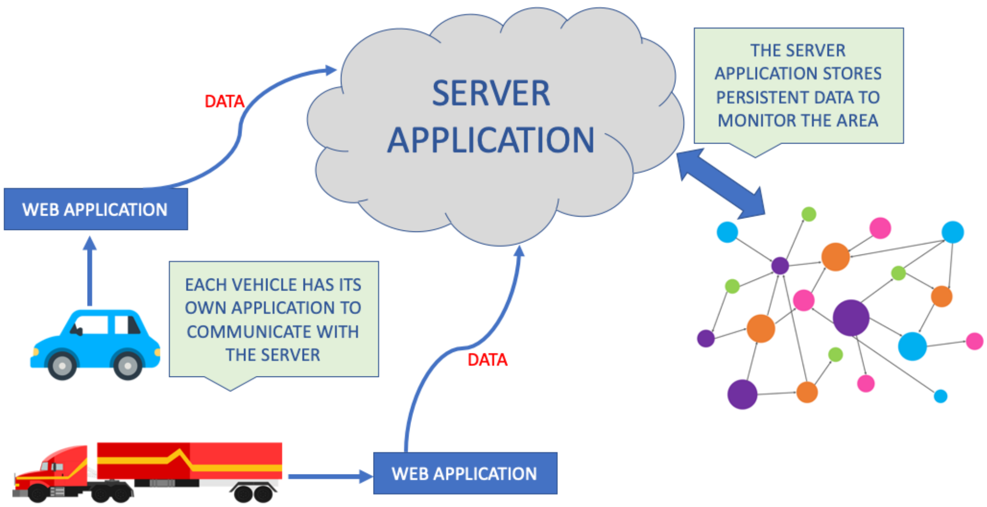

Figure 2.1: Architecture of the system.

The system consists of three principal components:

- Server: this component is responsible for the validation of data, supervision of the system and interaction with the database to store persistent data;
- Client: this component consist of an application, located on the vehicle, that communicates with the server and provide it with information about the vehicle (e.g. position, entry and exit time);
- Database: where are stored data about the current state of the area, i.e. actual capacity of places, position of vehicles and average times spent in each place.

3 Implementation

3.1 Used technologies

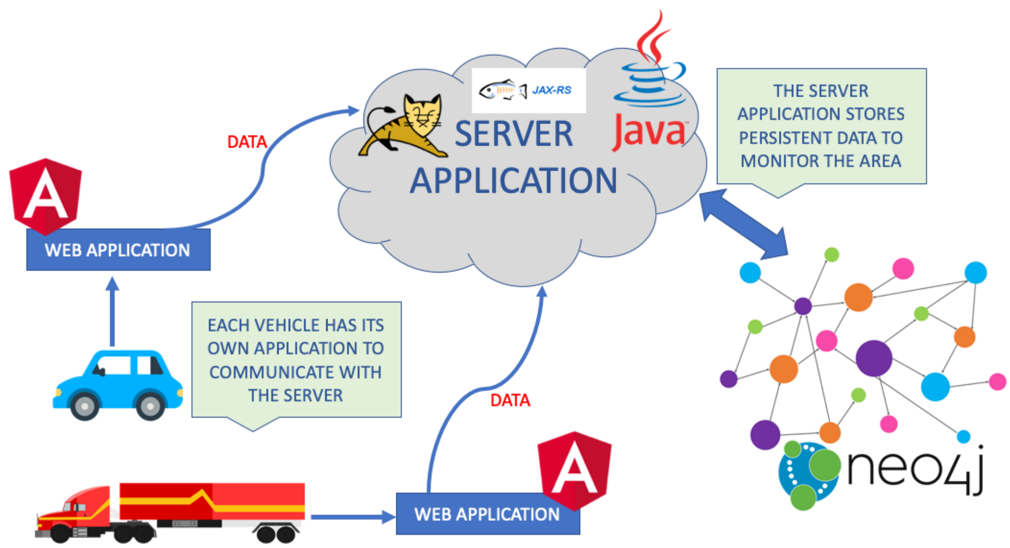

Figure 3.1: Architecture of the system with which technologies have been used for each component.

In figure 3.1 are presented the various components of the system along with the technologies used to achieve the objectives of each one. In particular:

- for the server application have been used various technologies:

  1. **Apache Tomcat** is an open source software that implements a *web server*, i.e. it manages the handling of Java Servlets, JSP. It is used to deploy web applications developed in Java [**tomcat**].
  2. **JAX-RS** has been used as a framework to develop REST application, given the used language has been **Java**.

- for the database component it has been decided to use **Neo4j** as storage for persistent data. Since what we are operating on (a map of places) can be modeled as connection between nodes, Neo4j, as a graph database, came in handy. Also it allowed to specify the type of relations between different nodes in the database as discussed in chapter 5.3.

- the client has been developed in **Angular**, a platform created to build applications with the web. Angular applications are equally used for mobile and desktop [**angular**].

3.2 Communication between components

There are two communication channels in this architecture:

1. client $\leftrightarrow$ server: the server is able to speak both XML and JSON, but for this particular project has been decided to use mainly JSON;

2. server $\leftrightarrow$ database: in this case the server uses **Cypher** query language (specific for Neo4j) in order to run queries in the database. As a result it is obtained a pseudo-json from which it is possible to extract the desired data, asked in the query.

4 Data model

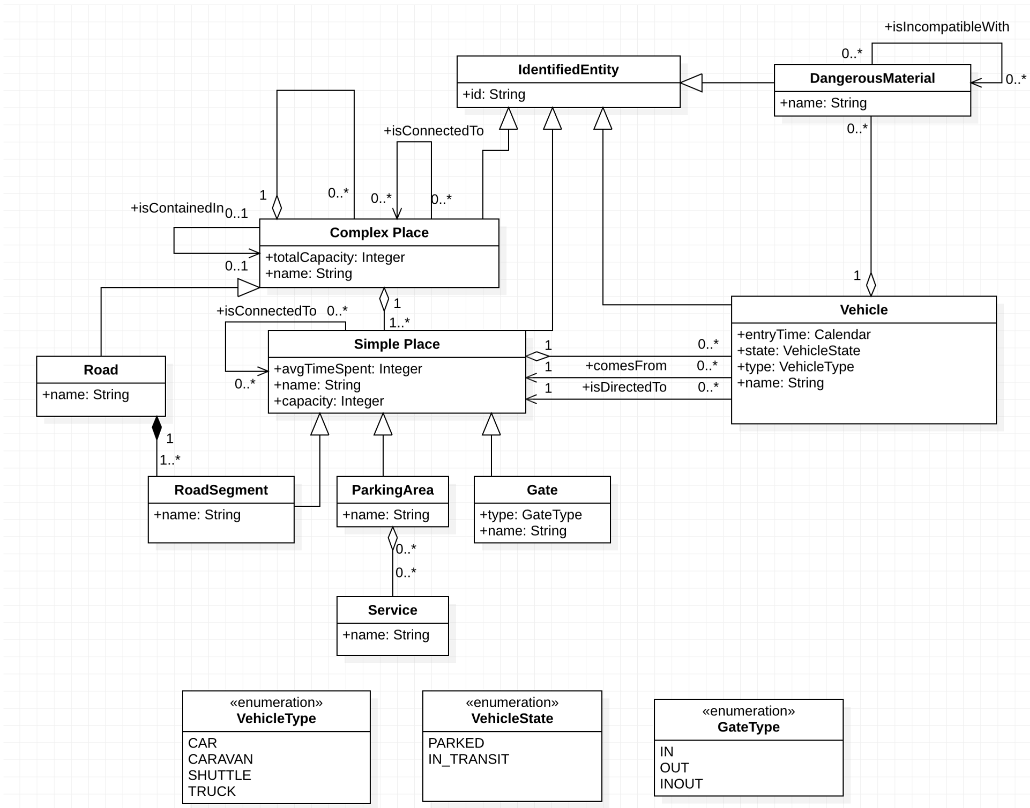

Figure 4.1: Data model for the application.

In figure 4.1 it is represented the UML model that describes the data handles by the application. Each node of the map can be described as a simple place. Each place has a maximum capacity, that represents the maximum number of vehicle allowed to be in the same time in that place. This property of the places will be exploited as constraint during the evaluation of a path for the entering vehicles. A particularity of this model is the presence of complex places. They have been defined in such way that they can contain a set of simple places or other complex places. For this particular

project has been defined only one type of complex place, but if, for future purposes, will be necessary to add other particular complex places, it will be possible to extends the ComplexPlace type.
All the information regarding the schema representing the model, are represented in the file */RNSService/rns/WebContent/WEB-INF/classes/xsd*. The Java classes used in the server application are generated from this file using JAXB.

5 Server application

5.1 Application organization

The server application has been developed and organized according to the multi-tier application paradigm. Such paradigm is shown in figure 5.1 and

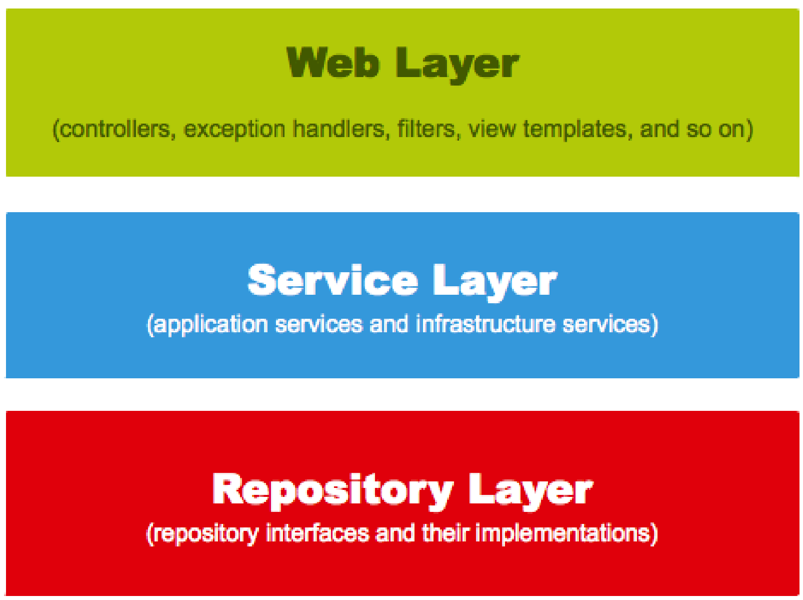

Figure 5.1: Subdivision of layers of multi-tier application.

it states that a web application has to be subdivided in different layers, each one of them with its own responsabilities. It's an approach similar to the *divide et impera* one, where a bigger problem is divided into sub-problems to achieve a solution. Similarly in this case, each layer works indipendently from the other and if they need to communicate, they do it through interfaces connecting them.
The organization of the layers in this project has been done this way:

- web (presentation) layer: all classes whose name ends with *Resource* or *Validator*. They are located in package *it.polito.dp2.rest.rns.resources* and they contain the definition of endpoints or filters used for the REST API to which clients will connect;

- service layer: this layer is the one responsible of all the business logic of the application and preparation of the information for possible queries to the database. All the logic of the developed application is contained in **RNSCore.java** class (itself located into *it.polito.dp2.rest.rns.resources* package). To achieve all objectives it is supposed to, it relies on the usage of some utility classes located into package *it.polito.dp2.rest.rns.utility* and to communicate possible errors to the presentation layer it utilizes custom exceptions, contained into package *it.polito.dp2.rest.rns.exceptions*;

- repository (database) layer: this layer has to handle all the interaction with the database. In this case it has been decided to use as a persistent storage Neo4j graph database [**neo4j**]. This part is discussed more in details in section 5.3.

To test the correctness of the developed application have been used two approaches:

1. the development of the client application is a test itself since it simulated a vehicle entering the system and performing random operations as discussed in chapter 6;

2. some basic junit test have been developed and added in package *it.polito.dp2.rest.rns.test.**. In particular this package is subdivided into other packages *it.polito.dp2.rest.rns.test.tests* and *it.polito.dp2.rest.rns.test.client*: the first contains all the function performing check and assertions on the result of certain operations, the second instead contains functions that instantiate clients to perform requests to specific endpoints.

As they are now, the tests check: if the vehicle that can be added is added correctly and all the information are preserved (POST of a vehicle and GET of the same vehicle); if a client tries to add a vehicle with as entry a place with not enough capacity, receives an error response; if a client tries to add twice a vehicle with the same id, receives an error response.
All the endpoint a client can exploit are visible in swagger documentation once the service is up and running. In order to access the full description of it from a browser, it is necessary to follow the link *Swagger documentation* visible when connecting to the homepage of the service at *http://localhost:8080/rns/*.

5.2 Z3

5.2.1 What is Z3

Z3 is a a theorem prover [**z3**]. We used it to implement the optimization algorithm in the server logic. Z3 is composed of two components:

1. the *OptSMT* module that is used to solve problems regarding the optimization of classical linear arithmetic objective functions (e.g. Knapsack problem [**knapsack**]);

2. the *MaxSMT* (actually a collection of MaxSAT solvers) module that we're most interested in, because it allows the definition of soft contraints in order to evaluate a solution.

5.2.2 Model

In order to define a model for Z3 to work on, it has been extracted a graph from the actual map, with updated capacities and position of vehicles.
The criteria for a node to be selected as a possible node are:

- the node doesn't contain any dangerous material.

- the node contains some vehicles carrying dangerous materials, but they are compatible with the one of the new vehicle.

Mathematical Model

The obtained mathematical model is the following.

Objective Function

$\min \; \sum_{i=1}^{N} y_i t_i$

Variables

N: number of nodes of the system

$y_i \in \{0, 1\}$: a generic node i

$c_i \geq 0$: the capacity of the node i

$t_i \in \mathbb{N}$: the weight average time spent in the node i

$y_{ij} \in \{0, 1\}$: a connection from node i to node j

Constraints

$c_i \geq y_i$

$\sum_{i \in N} y_{si} = 1$, where s is the origin

$\sum_{i \in N} y_{id} = 1$, where d is the destination

$\sum_{i,j \in N} y_{ij} - \sum_{i,j \in N} y_{ji} = 0$, where y_{ij} are the incoming and y_{ji} the outcoming connections

Final result

The final output of the z3 model consists in a set of boolean variables. If the value is **true**, that means that the node is to be considered in the path.
In the snippet 5.1 there is an example for a path evaluated for a newly entered vehicle.

```
(define-fun y_a02-01 () Bool true)
(define-fun y_ss02 () Bool true)
(define-fun y_a02-02 () Bool false)
(define-fun y_ss01 () Bool false)
(define-fun y_a01-01 () Bool false)
(define-fun y_a01-02 () Bool true)
(define-fun y_a01-03 () Bool false)
(define-fun y_ss03 () Bool true)
(define-fun y_g05 () Bool true)
(define-fun y_g04 () Bool true)
```

Snippet 5.1: Final result that need to be parsed to retrieve ne node ids

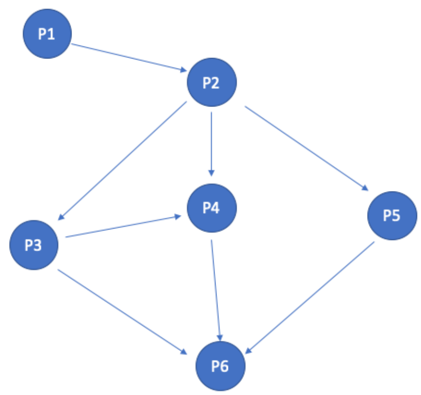

Figure 5.2: Example graph.

5.2.3 Example

In this subsection will be presented an example of how the server will compute the correct path for a new vehicle that wants to enter the system. The initial graph that is given Z3 to work on is the one in figure 5.2. Here two assumptions are made:

1. node **P5** contains a vehicle that carries a dangerous material not compatible with the one carried by the new vehicle that wants to enter;
2. node **P4** has not enough capacity to accept another vehicle.

The first step that it has to be performed is selecting the origin and destination in the graph (in this case **P1** and **P6**) and infer in Z3's optimizer an hard constraint that state that both these nodes has to result **true** in order to verify the correctness of the solution. The selection of origin node (red) and destination node (green) is shown in fugire 5.3. The next step is pruning from the actual graph, in order to input a correct map to Z3, all the nodes that don't meet the dangerous material constraint. Such constraint states that for a node to be selected it has not to contain any vehicle carrying materials not compatible with the one carried by the vehicle that wants to

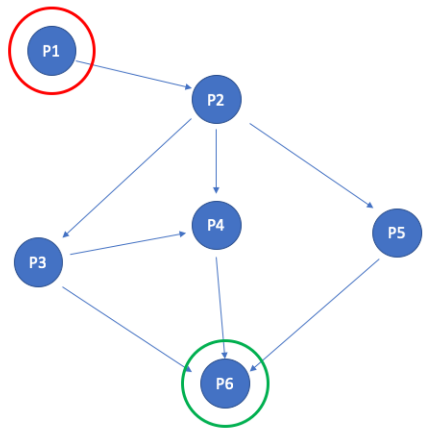

Figure 5.3: Selection of origin (red) and destination (green).

enter the system. Coherently with assumptions, node **P5** is pruned out from the graph, since it contains a vehicle with materials not compatible with the new entering (figure 5.4). Once a correct graph has been retrieve for Z3, the next phase is to define boolean expressions for the optimizer. To achieve that, it is necessary to recur the whole graph in order to explore every node. For each one of them, different expressions has to be defined:

- a boolean expression for understanding if a node is considered in the final path, that will result **true** only if one of its incoming connection's boolean expression is **true**;
- a boolean expression for the capacity of the node, that will result to true only if the capacity of the node is greater or equal than 1. If this expression is false, it will invalidate the node as a whole, making it not suitable for the route;
- a boolean expression for each one the connection betwee the node and the previous ones. This is not an actual boolean expression, because it is derived from the AND between the two boolean expressions of the nodes at the end of the connection.

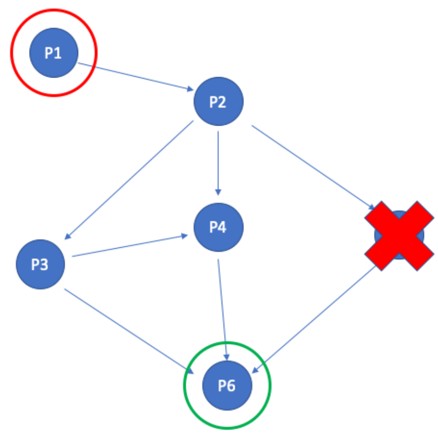

Figure 5.4: Pruning of a node due to dangerous materials constraints.

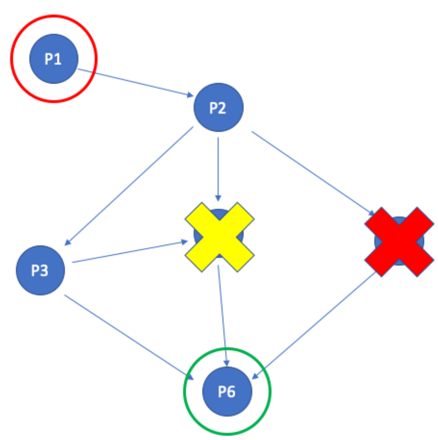

Figure 5.5: Pruning of a node due to capacity constraints.

Given the definition of such expressions, since the capacity of node **P4** doesn't meet the requirements as specified above, the corresponding boolean expression of the node is invalidated by the optimized, making it not a suitable node for the evaluation of a path as shown in figure 5.5. The resulting

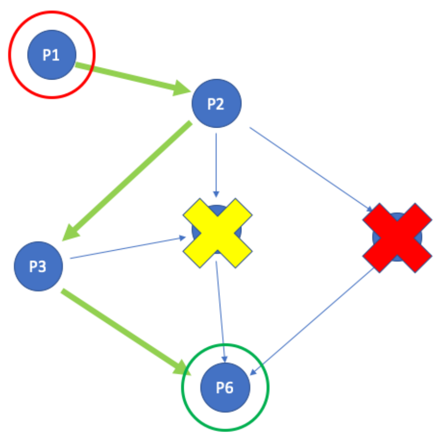

Figure 5.6: Evaluation of path from source to destination.

path produced by Z3 is the one shown in figure 5.6. In the case multiple paths are present from source to destination, the optimizer will choose the one with the minimum traverse time, using the min time constraint define as subsection 5.2.2 shows.

5.3 Neo4j

As a persistent backend storage, it has been decided to use Neo4j [**neo4j**]. The main reasons behing this choice are:

- ease of use;
- a graph database is very handy when working with maps;
- neo4j is the leader in the development of graph database frameworks.

The developed library for interfacing with the database is located in package *it.polito.dp2.rest.rns.neo4j* and it is based on the usage of Neo4j drivers and Cypher query language, that is one of the two way to go suggested by Neo4j documentation (Cypher and drivers [**neo4jcypher**] or HTTP rest API [**neo4jhttprest**]).
The package contains two classes:

1. class **Neo4jInteractions.java**: it is used to to start/close a session with the database, execute a query, retrieve the results. It basically offers a set of function to allow the service layer of the server application to store, retrieve, delete and update data;
2. class **StatementBuilder.java**: this class is responsbile of providing a set of functions that create statements (a.k.a. queries), to be run in the database through the driver, depending on some parameters that are given.

The access from service layer to database layer is achieved through an object of type **Neo4jInteractions**. It has been developed using a *singleton* pattern in order to have only one instance of this object for the whole application. Such pattern consists in the definition of a private constructor and a private static instance of an object of the same type of the containing class. This is instance is made accessible through static methods.
In the database, each neo4j node corresponds to a place. Since the defined model for the project stated that even connection between places are to be considered places themselves, the relationships between nodes that neo4j offers have been exploited in a different way. What relationships have been used is to describe two things:

1. the type of relationship existing between the nodes (container, connection, ...);

2. the direction in which is possible to traverse the two nodes. Let's assume there exist two nodes, *node1* and *node2*, that are strongly connected (which means there is a relationship from *node1* to *node2* and one the other way around), a vehicle is able to go either from *node1* to *node2* or from *node2* to *node1*. If there is only one relation, for instance from *node1* to *node2*, the vehicle will only be able to traverse the nodes the same direction the relation is pointing (in this case *node1* → *node2*).

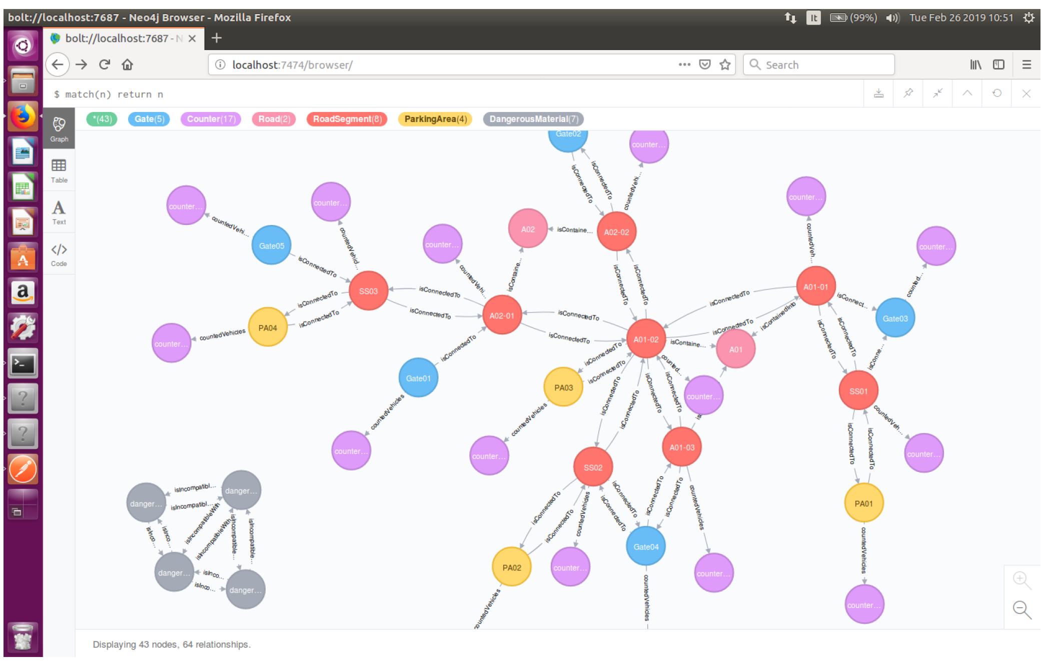

Figure 5.7: Example of graph of the system.

In figure 5.7, it is represented a possible description of a system where there are present gates, parking area, road segments and road. Other than that, worth of a notice is the presence of some particular nodes:

- **counter** nodes (*purple* nodes in figure 5.7): this kind of nodes are used to track information about places. There is a counter node for each place and it is responsible of keeping the number of reservations for that place and the number of vehicles that have passed through that place;

- **dangerous material** nodes (*gray* nodes in figure 5.7): it is present a node, for each dangerous material known by the system. Between these nodes exist relations of incompatibility, in order to allow the system to recognize if a material is incompatible with another.

6 Client

The client is a web application developed in Angular. It is designed to choose and simulate the vehicle journey inside a given map.

6.1 Client architecture

The client is composed by two pages:

- Home page: where it is possible to choose the path
- Route page: where it is possible to see the route and the next directions to take.

The application uses a service to make HTTP requests to the server to retrieve the information needed and another one to share data inside the client.

6.2 Home page

In the home page is displayed the actual position of the vehicle in the system, if it has one, as shown in figure 6.1.

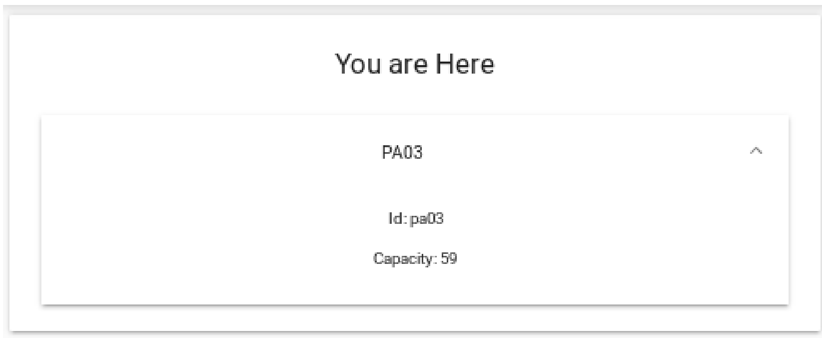

Figure 6.1: Actual position in the system.

All info of the places are retrieved using the API of the server */rns/webapi/rns*, starting point to obtain a map. It is then possible to decide the origin and destination of the vehicle, in order to retrieve an authorization from the server and a path that should be followed, from the list as per figure 6.2.

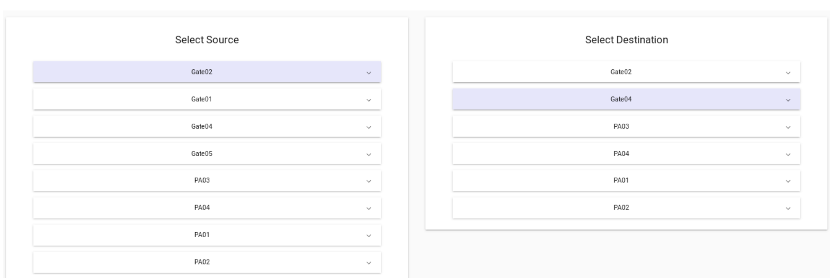

Figure 6.2: Possible choices of source and destination.

Also dangerous material are displayed at the end of the page, where it is possible to select only compatible materials.

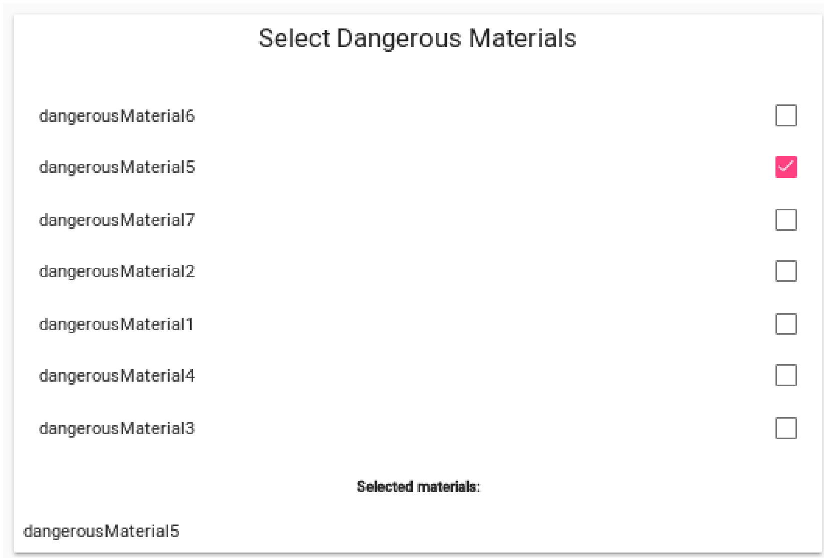

Figure 6.3: Possible choices of source and destination.

It is then possible to create a POST request using JSON with all the information collected, that are summarized as seen in figure 6.4.

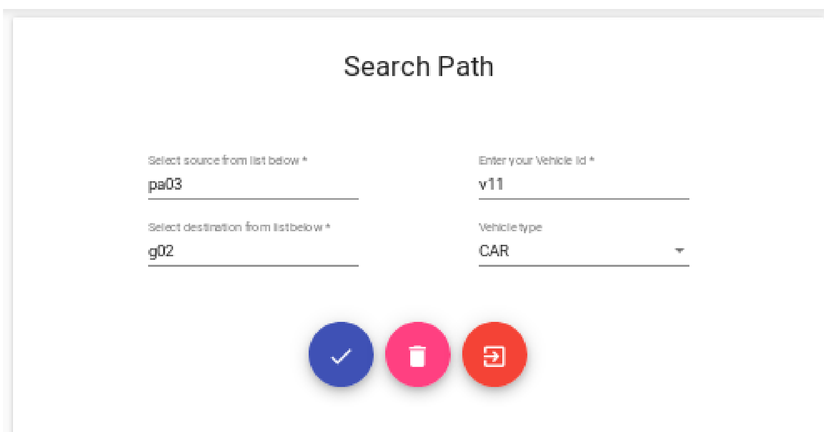

Figure 6.4: Possible choices of source and destination.

6.3 Route page

The client then redirects to the route page, where it is simulated the journey of the vehicle, by randomly choosing, at every crossroad, the next road to take. It is the used a PUT request to update the position of the vehicle on the server. The design was implemented in such way to test the correct behavior of the server in case of error. The probability to chose a wrong road is constant and defined as 90%. In this eventuality the server recompute a path and the client is updated.

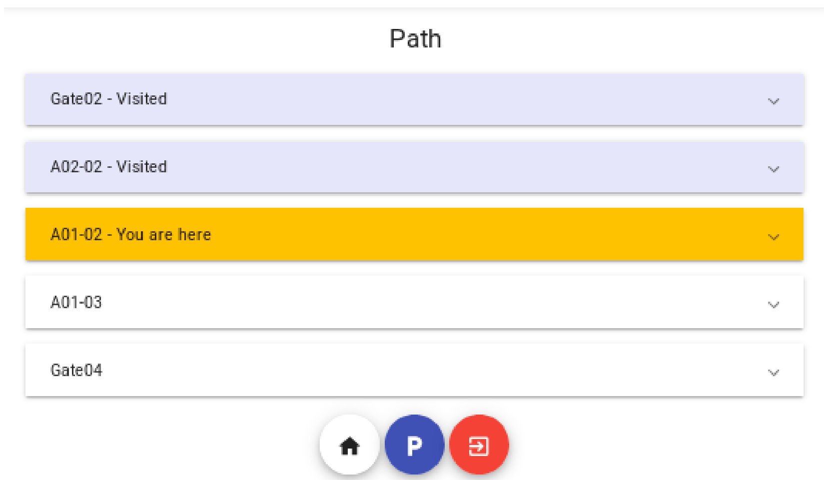

Figure 6.5: Simulation of the path of the vehicle.

7 Project setup

In this section will be described the necessary steps to have the application up and running.

7.1 Necessary libraries

The setup of the necessary libraries and framework has been done simmetrically to the one performed in DP2 virtual machine available on the course website. Only difference is the addition of Z3 library for the machine, the setup of it is shown in section 7.2. In figure 7.1 is shown the actual setup used for the machine. Additional library like neo4j drivers and junit are provided in the

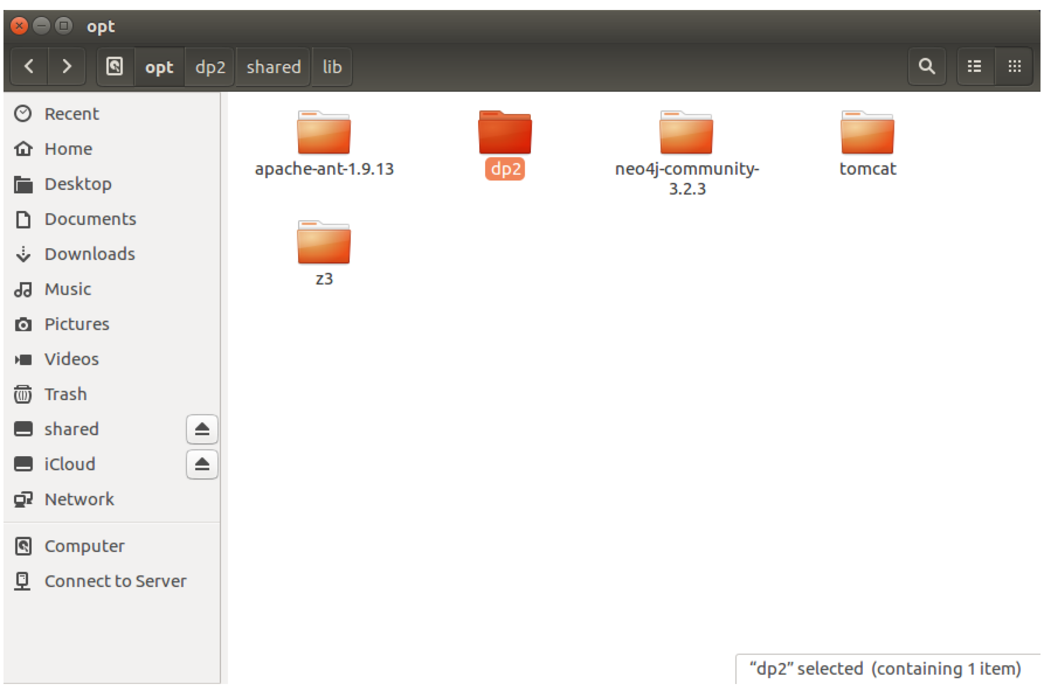

Figure 7.1: Folder */opt* of the machine used as development environment.

lib and *lib-src* folder of the application project. They are already included in classpath during compilation launched through ant scripts. If errors are signaled by Eclipse when opening the project, probably they should be added to build path. After the addition it is suggestable to refresh and clean the project.

7.2 Setup of Z3 in machine running Ubuntu

The development enviroment that has been used is **Ubuntu**. In order to use Z3 library in such operating system, there are a couple of steps one has to follow.

1. download the prebuilt version of the library from the offical GitHub repository `https://github.com/Z3Prover/z3/releases`;

2. once extracted the files, we have to place them in a specific location in which we will command other applications to look for the needed classes;

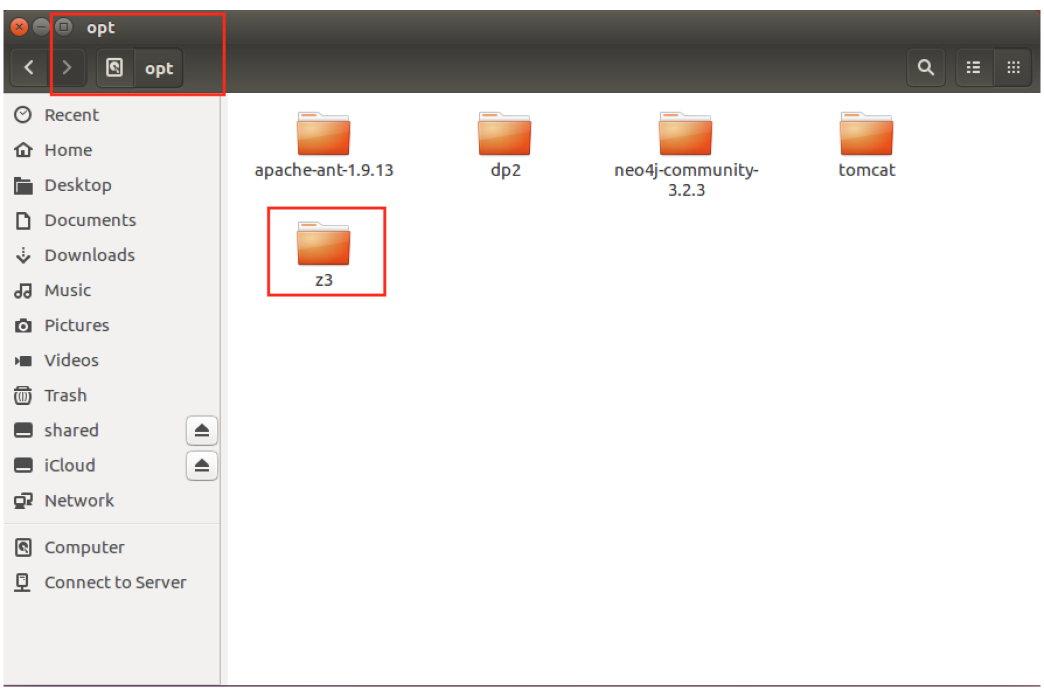

Figure 7.2: Example of where to place Z3 extracted libraries.

3. after everything has been placed in the chosen location, we have to **define** an environment variable which will allow the Java application to know where the Z3 library is located. For Ubuntu such variable is LD_LIBRARY_PATH. This variable must point to the location of the bin folder of the extracted Z3 library; In figure 7.3 is shown an example

Figure 7.3: Example of how to define LD_LIBRARY_PATH environment variable.

   of definition of the LD_LIBRARY_PATH environment variable.
   In this case has been define globally for the whole machine, that means it has been inserted in the file */etc/environment*, but the same result could have been achieve defining it locally for the used in */Home/.bashrc* file;

4. last step, is to add the Z3 .jar file to the build path of our project.

This is an automatic configuration and Tomcat will load the dynamic libraries in the bin folder by itself. If this is not sufficient, we need to manually put in the WebContent/WEB-INF/lib/jni the Z3 library and point LD_LIBRARY_PATH to that folder. Double check that the .jar file is present in the copied folder. Please make sure, in order to use the library in Tomcat, to have correctly set CATALINA_HOME variable, pointing to Tomcat folder and JAVA_HOME variable (provided you have installed Java on the machine).

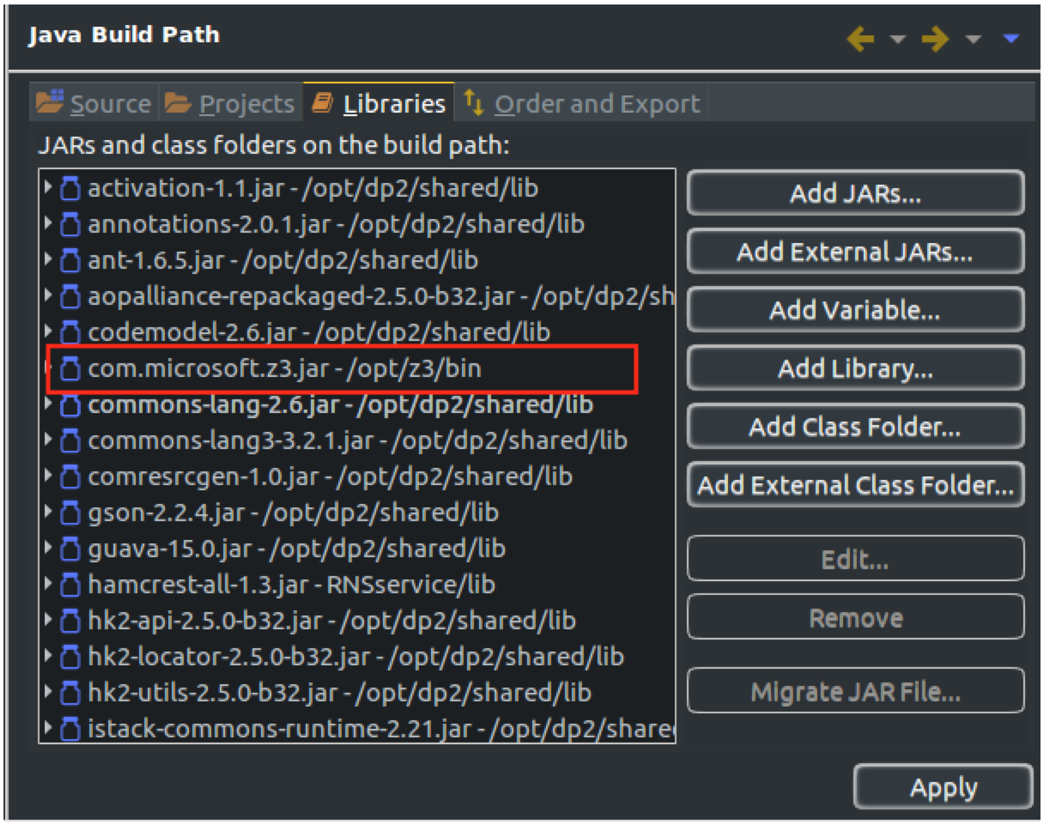

Figure 7.4: Example of how the project build path should look like.

7.3 Server application and tests

In order to launch and compile the server application (in the directory *RNSService*) and its tests, it is available a set of ant scripts. In particular:

- ant script **neo4j-build.xml** provides a set of target to launch/stop/restart&clean neo4j database;
- ant script **build.xml** provides all the target to start/stop tomcat, to deploy the application and to run tests, it relies upon another script to define the operation of such targets, that is **build-rns.xml**.

The targets can be either launched via Eclipse IDE or command line.
To setup the server up and running it is necessary to follow these step:

1. start Neo4j by using **start-neo4j** target of **neo4j-build.xml** script. From command line *ant 'start-neo4j' -f /path/to/neo4j-build.xml*;

2. start Tomcat by calling **start-tomcat** target of **build.xml** script. From command line *ant 'start-tomcat' -f /path/to/build.xml*;

3. deploy web service by calling **redeploy** target of **build.xml** script. From command line *ant 'redeploy' -f /path/to/build.xml*.

This last command embed in itself the generation of the bindings from XML Schema with JAXB and the compilation of all the necessary classes.
In order to launch the tests written for the service it is necessary to use **rns-tests** target of **build.xml** script. This target will launch the coompilation of all the classes necessary to the tests and launch them with JUnit.

7.4 Client application

To setup the client (in the directory *Client*) it is necessary to install Node.js, then it is possible to launch the application. To get Node.js, go to nodejs.org. These are the steps:

1. go to dir *Client*

2. install the angular CLI running the following command:
   npm install -g @angular/cli

3. get all node dependencies running the following command:
   npm install

4. launch the app running the following command:
   ng serve

Bibliography

[1] Apache Tomcat website, `http://tomcat.apache.org/`

[2] Neo4j website, `https://neo4j.com/`

[3] Neo4j Drivers documentation, `https://neo4j.com/docs/driver-manual/1.7/`

[4] Neo4j HTTP REST API documentation, `https://neo4j.com/docs/http-api/3.5/`

[5] Z3 Prover, `https://github.com/Z3Prover/z3`

[6] Knapsack Problem Wikipedia, `https://en.wikipedia.org/wiki/Knapsack_problem`

[7] Angular, `https://angular.io/docs`