"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 client application, of 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 sent periodically (every time a new place is entered) 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

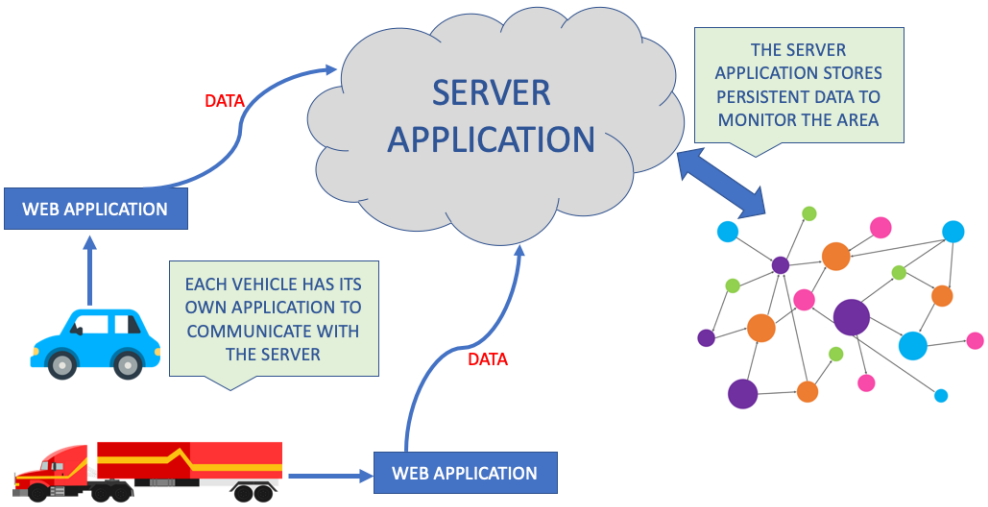

Figure 2.1: Architecture of the system.

As the figure 2.1 shows, 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: here the server application stores data about the current state of the area, i.e. actual capacity of places, position of vehicles, average times spent in each place.

3 Implementation

3.1 Used technologies

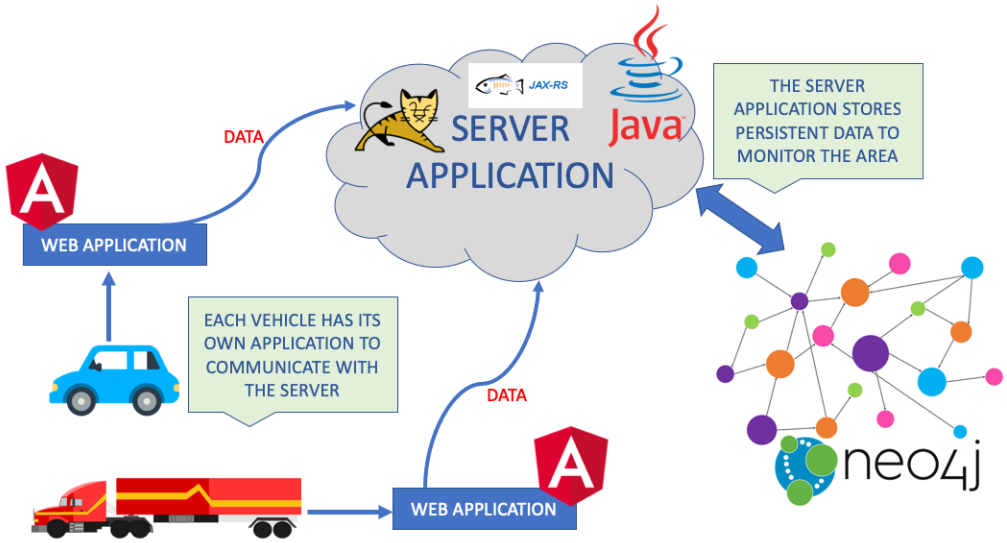

Figure 3.1: Architecture of the system with which technologies have been used for each component.

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 modelled 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.

- client – TODO

3.2 Communication between components

There are two communicationchannels in this architecture:

1. cleint $\leftrightarrow$ 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

5 Server application

5.1 Application organization

asdfghjkl

5.2 Z3

5.2.1 What is Z3

Z3 is a a theorem prover [**z3**]. It came in handy with our project, since it can be exploited to solve problems that requires the definition of constraints.
In particular, 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 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;

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 5.2 is shown an example 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;

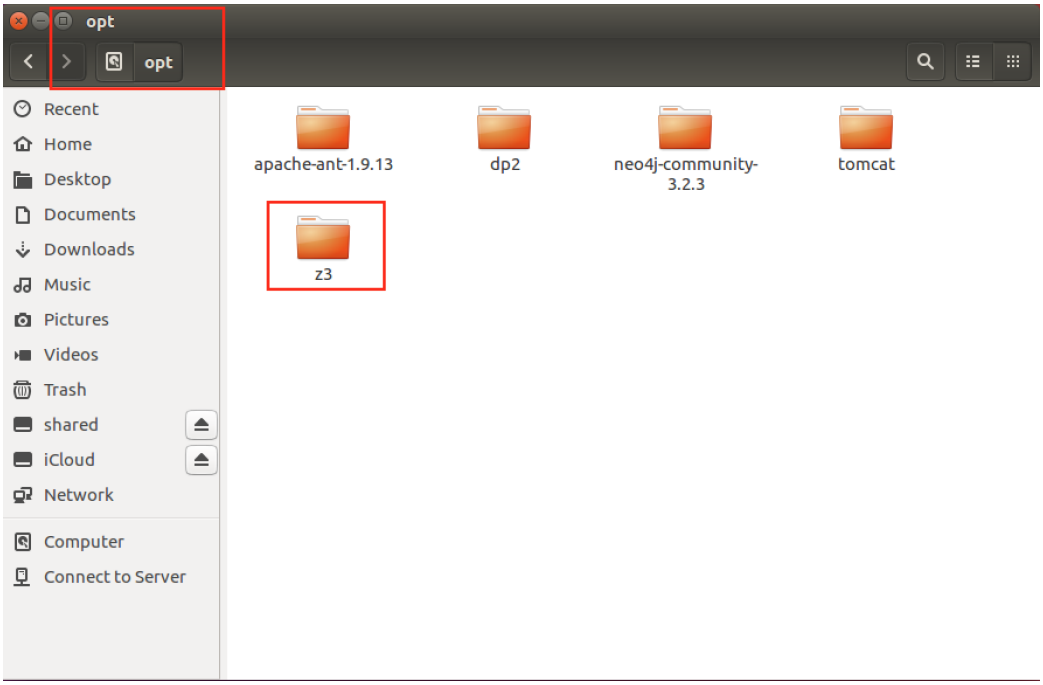

Figure 5.1: Example of where to place Z3 extracted libraries.

Figure 5.2: Example of how to define LD_LIBRARY_PATH environment variable.

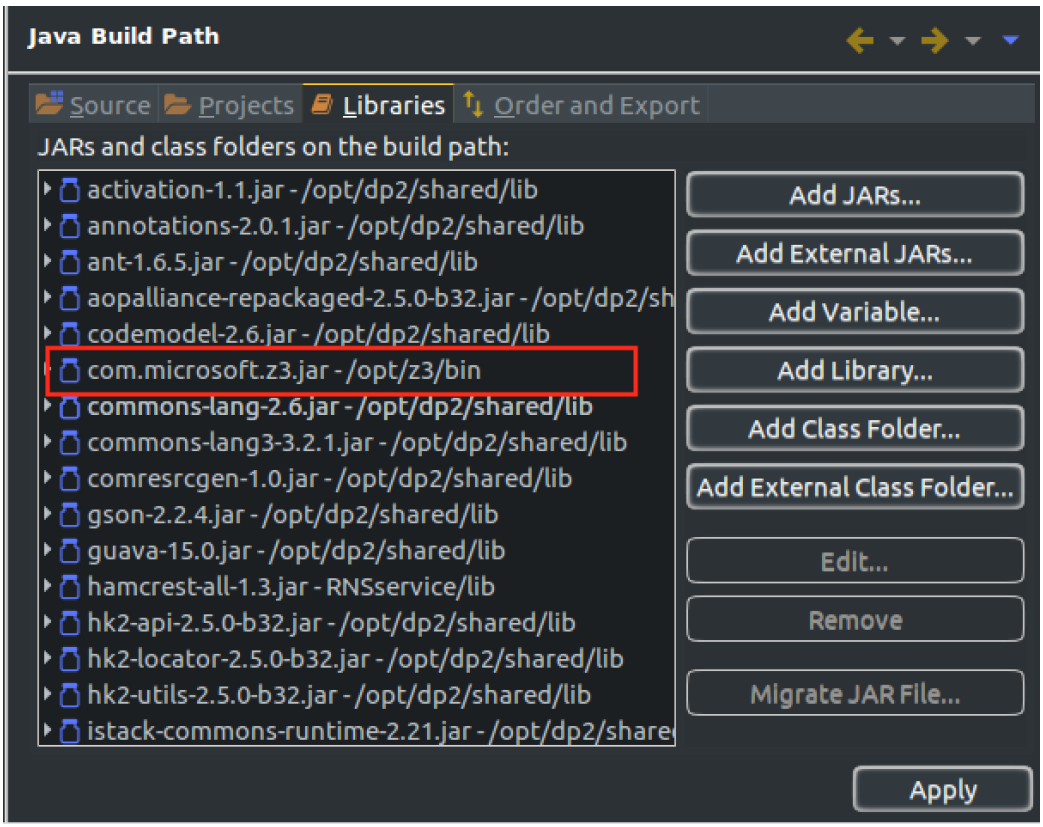

Figure 5.3: Example of how the project build path should look like.

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.
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).

5.2.3 Model

In order to define a model for Z3 to work on, it has been extracted a graph from the actual map stored in the database. The procedure can be subdivided

in various steps:

1. creation of a context;
2. extraction of the viable nodes from the map;
3. definition of constraints to select correct nodes the vehicle can traverse;
4. definition of soft constraints.

Since the map in not known in advance, in order to traverse the whole graph has been used **recursion**.
The definition of the context and creation of the optimizer class, has been done as shows code snippet 5.1.

```
1  ctx = new Context();
2  mkOptimize = ctx.mkOptimize();
```

Snippet 5.1: Initialization of context

The context allows the creation of boolean constants, that are assigned a value from the optimizer during its evaluation of feasibility of the problem (based on constraints that have previously been defined).

Pruning of nodes from the map

First step is the one to retrieve the nodes from the database and select the ones which don't contain any material or, if vehicle containing materials are present in that node, they have to be compatible with the one carried by the new vehicle entering the system.
After selecting these nodes, it has to be defined a boolean expression for each one of them. Such expression will result **true** if the node has been selected for the path, **false** otherwise.

Definition of boolean variables

In order to define boolean variables for the nodes, some other informations are required:

- boolean variable of the previous node we're coming from in recursion. If that is true, the current node, **may be** taken into the path;

- boolean variable of the connection between the previous node and the current one. If it is true, and the previous node as well, then the current node **is** taken in the path.

```
BoolExpr y_curr = ctx.mkBoolConst("y_" + current.
    getId());
```

Snippet 5.2: Definition of a boolean expression

The code snippet 5.2 shows how the boolean variables for the nodes have been defined.
Once this has been defined it is necessary to derive a variable to determine whether the connection between the previous node and the current one is taken. This step is shown in the snippet 5.3.

```
this.connectionsBool.put("z_" + prevId + "_" + current
    .getId(), ctx.mkAnd(y_prev, y_curr));
```

Snippet 5.3: Definition of a boolean expression for connection between nodes

The variable for the connection has been defined as the AND between the boolean variable of the previous node and the current one, so that it will result **true** only if both are **true**.
Last variable it has to be defined is the one responsible for the capacity constraint. It hasn't been used a *BoolExpr* like in the previous cases for this one, instead an *ArithExpr*. Such variable has been defined as the subtraction between the current capacity of the node and 1 as code snippet 5.4 shows. In this way, if the arithmetic result is less than 0, the boolean expression defined for the node has to be forced to false, implying that the node has not to be considered.

```
ArithExpr leftSide = ctx.mkSub(ctx.mkInt(
    actualCapacity), ctx.mkInt(1));
```

Snippet 5.4: Definition of arithmetic expression for capacity constraint

Definition of hard constraints

Once everything has been created and initialized, it is necessary to specify to the optimizer the policies to assign value (true or false) to the different created variables.

First of all the constraint on the capacity has to be defined. This is done the way code snippet 5.5 shows.

```
1 ArithExpr capacity = this.nodesCapacity.get(node.
     getKey());
2 mkOptimize.Add(ctx.mkImplies(ctx.mkLt(capacity, ctx.
     mkInt(0)), ctx.mkNot(node.getValue())));
```

Snippet 5.5: Definition of capacity constraint for the optimizer

In order words if *capacity* is Less Than 0, we force to negate the boolean expression of the node.
Then the conditions for the nodes have to be defined:

- if the node is the source from which the path has to be started, then it has to be true. Since it has no incoming connection from previous nodes (by being the first one), no other constraints have to be defined 5.6;

```
1     mkOptimize.Add(ctx.mkEq(node.getValue(), ctx.
        mkBool(true)));
```

Snippet 5.6: Defintion of constraint for the source node

- if the node is the destination of the vehicle, then it has to be true, and it has to have only one incoming connection. No outgoing connection because it is the last node to be visited 5.7;

```
1     mkOptimize.Add(ctx.mkEq(node.getValue(), ctx.
        mkBool(true)));
2     mkOptimize.Add(ctx.mkImplies(node.getValue(),
        ctx.mkAtLeast(conns.stream().toArray(BoolExpr
        []::new), 1)));
3     mkOptimize.Add(ctx.mkImplies(node.getValue(),
        ctx.mkAtMost(conns.stream().toArray(BoolExpr
        []::new), 1)));
```

Snippet 5.7: Defintion of constraint for the destination node

The node boolean value has to be forced to **true**, like in the source node case. Since it is true, exactly one (combination of at least one constraint and at most one constraint) connection between all incoming connection (conns is a list containing all the boolean variables for the connection incoming define like in code snippet 5.3) has to be selected;

- if it is a generic node it has to be inferred that if this node has been selected (boolean variable of the node set to **true**), then one incoming connection has to be selected as shown in code snippet 5.8.

```
1    mkOptimize.Add(ctx.mkImplies(node.getValue(),
        ctx.mkOr(conns.stream().toArray(BoolExpr[]::
        new))));
```

Snippet 5.8: Defintion of constraint for a geneeric node

The selection of only two connection per node (one incoming and one outgoing) will be forced by the last node and the definition of soft constraints.

Definition of soft constraints

The only soft constraint that need to be added to the optimizer is the one on the minimum amount of time predicted for the traversing of the route. This is shown in code snippet 5.9

```
1  for(Entry<String, BoolExpr> node : this.nodes.entrySet
      ()) {
2    SimplePlaceReaderType place = Neo4jInteractions.
       getInstance().getPlace(node.getKey());
3    mkOptimize.AssertSoft(ctx.mkNot(node.getValue()),
       place.getAvgTimeSpent().intValue(), "latency");
4  }
```

Snippet 5.9: Defintion of soft constraint on minimum traverse time

In other words, for each node in the graph, we retrieve information about the place (i.e. average time spent in each place), then we infer that the probability of considering such node in the path is inversely proportional to the average time spent in the node.

Final result

The final model from the graph is very simple, only a set of boolean variables representing each one a node in the graph. If the value is **true**, that means that the node is to be considered in the path, otherwise not.
In the snippet 5.10 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 5.10: Final result that need to be parsed to retrieve ne node ids

5.2.4 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

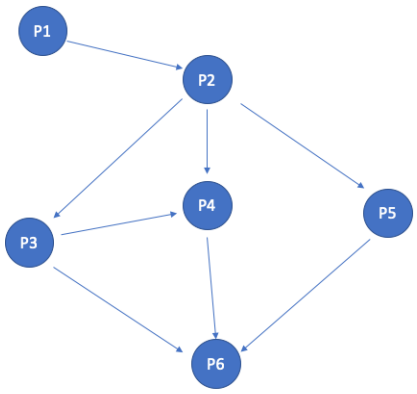

Figure 5.4: Example graph.

initial graph that is given Z3 to work on is the one in figure 5.4. 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 5.5. The next step is

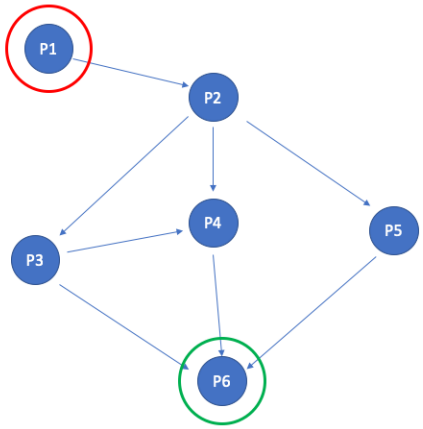

Figure 5.5: Selection of origin (red) and destination (green).

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 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 5.6). 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:

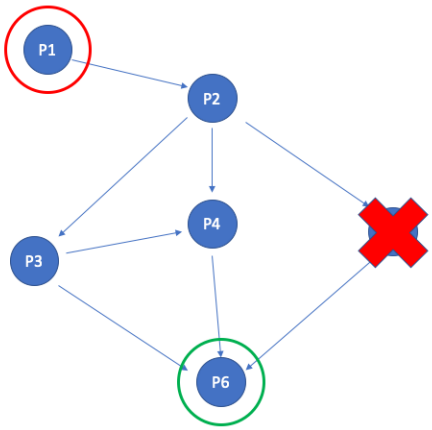

Figure 5.6: Pruning of a node due to dangerous materials constraints.

- 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.

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 5.7. The resulting path produced by Z3 is the one show in figure 5.8. In the case multiple paths are present from source to destination, the optimizer will choose the one with the

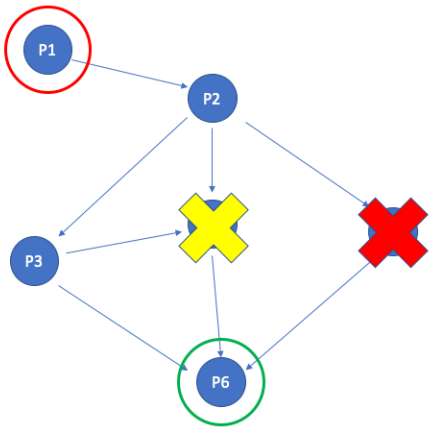

Figure 5.7: Pruning of a node due to capacity constraints.

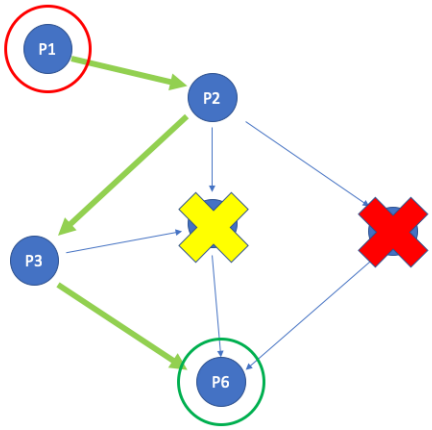

Figure 5.8: Evaluation of path from source to destination.

minimum traverse time, using the min time constraint define as subsection

?? shows.

5.3 Neo4j

6 Client

7 Project setup

Bibliography

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

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

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

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