**Traffic Control System in the Connected Cars Context**

**(TCS-3C)**

LICENSE THESIS

|  |  |  |
| --- | --- | --- |
|  | Graduate: | **Andrei TUDORICA** |
|  |  |  |
|  | Supervisor: | **Prof. dr. eng. Rodica POTOLEA** |

**2019**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  | |  | |
| DEAN, |  | | HEAD OF DEPARTMENT, | |
| **Prof. dr. eng. Liviu MICLEA** |  | **Prof. dr. eng. Rodica POTOLEA** | |

Graduate: **Firstname LASTNAME**

**LICENSE THESIS TITLE**

1. **Project proposal:** *- care e problema pe care o rezolv*

**-** obiectivele

- concluziile (rezultate din ce mi-am propus)

1. **Project contents:** *(enumerate the main component parts) Presentation page, advisor's evaluation, title of chapter 1, title of chapter 2, …, title of chapter n, bibliography, appendices.*
2. **Place of documentation**: *Example*: Technical University of Cluj-Napoca, Computer Science Department
3. **Consultants**:
4. **Date of issue of the proposal:** November 1, 2018
5. **Date of delivery:** July 8, 2019

|  |  |
| --- | --- |
| Graduate: | \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ |
|  |  |
| Supervisor: | \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ |

**Declaraţie pe proprie răspundere privind**

**autenticitatea lucrării de licenţă**

Subsemnatul(a)**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**, legitimat(ă) cu \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ seria \_\_\_\_\_\_\_ nr. \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_   
CNP \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_, autorul lucrării \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_elaborată în vederea susţinerii examenului de finalizare a studiilor de licență la Facultatea de Automatică și Calculatoare, Specializarea \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ din cadrul Universităţii Tehnice din Cluj-Napoca, sesiunea \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ a anului universitar \_\_\_\_\_\_\_\_\_\_, declar pe proprie răspundere, că această lucrare este rezultatul propriei activităţi intelectuale, pe baza cercetărilor mele şi pe baza informaţiilor obţinute din surse care au fost citate, în textul lucrării, şi în bibliografie.

Declar, că această lucrare nu conţine porţiuni plagiate, iar sursele bibliografice au fost folosite cu respectarea legislaţiei române şi a convenţiilor internaţionale privind drepturile de autor.

Declar, de asemenea, că această lucrare nu a mai fost prezentată în faţa unei alte comisii de examen de licenţă.

In cazul constatării ulterioare a unor declaraţii false, voi suporta sancţiunile administrative, respectiv, *anularea examenului de licenţă*.

|  |  |  |
| --- | --- | --- |
| Data  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ |  | Nume, Prenume  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ |
|  |  |  |
|  |  | Semnătura |

Table of Contents

[Chapter 1. Introduction 1](#_Toc12908442)

[1.1. Traffic context 1](#_Toc12908443)

[1.2. Connected Cars Context 1](#_Toc12908444)

[Chapter 2. Project Objectives 2](#_Toc12908445)

[2.1. Problem Overview 2](#_Toc12908446)

[2.2. Project Objectives 2](#_Toc12908447)

[2.3. Requirements 2](#_Toc12908448)

[2.3.1. Functional Requirements 2](#_Toc12908449)

[2.3.2. Nonfunctional Requirements 2](#_Toc12908450)

[Chapter 3. Bibliographic Research 4](#_Toc12908451)

[3.1. Navigation Systems 4](#_Toc12908452)

[3.1.1. Proprietary Navigation Systems (Closed Source) 4](#_Toc12908453)

[3.1.2. Open Source Navigation Systems 4](#_Toc12908454)

[3.2. Maps and Data 6](#_Toc12908455)

[3.3. Routing 6](#_Toc12908456)

[3.3.1. Graph forms 6](#_Toc12908457)

[3.3.2. Algorithms 6](#_Toc12908458)

[3.4. Communication Methods 6](#_Toc12908459)

[Chapter 4. Analysis and Theoretical Foundation 7](#_Toc12908460)

[4.1. Motivation for choosing Itinero 7](#_Toc12908461)

[4.2. Conceptual Architecture of the System 8](#_Toc12908462)

[4.2.1. Live Processing Module 8](#_Toc12908463)

[4.2.2. Route Computing Module 8](#_Toc12908464)

[4.2.3. Modules 9](#_Toc12908465)

[4.2.4. Example of System build on the Conceptual architecture 9](#_Toc12908466)

[4.3. TCS-3C - Design 13](#_Toc12908467)

[4.3.1. Design 13](#_Toc12908468)

[4.3.2. Traffic Data 15](#_Toc12908469)

[4.3.3. Routing process 16](#_Toc12908470)

[Chapter 5. Detailed Design and Implementation 19](#_Toc12908471)

[5.1. Design 19](#_Toc12908472)

[5.2. Live Traffic Server 19](#_Toc12908473)

[5.2.1. Design 19](#_Toc12908474)

[5.2.2. Flow 24](#_Toc12908475)

[5.3. Route Computing Server 26](#_Toc12908476)

[5.3.1. Design 26](#_Toc12908477)

[5.3.2. Flow 28](#_Toc12908478)

[5.4. Traffic Simulator 31](#_Toc12908479)

[5.4.1. Design 31](#_Toc12908480)

[5.4.2. Flow 33](#_Toc12908481)

[Chapter 6. Testing and Validation 36](#_Toc12908482)

[6.1. Hardware 36](#_Toc12908483)

[6.2. Metrics 36](#_Toc12908484)

[6.2.1. System metrics 36](#_Toc12908485)

[6.2.2. Simulation metrics 36](#_Toc12908486)

[6.3. Scenario 36](#_Toc12908487)

[6.4. Results 37](#_Toc12908488)

[Chapter 7. User’s manual 38](#_Toc12908489)

[7.1. Environment 38](#_Toc12908490)

[7.2. Dependencies 38](#_Toc12908491)

[7.3. Route Computing Server 39](#_Toc12908492)

[7.4. Live Traffic Server 41](#_Toc12908493)

[7.5. Traffic Simulation 41](#_Toc12908494)

[7.6. Distribution 43](#_Toc12908495)

[Chapter 8. Conclusions 44](#_Toc12908496)

[Bibliography 45](#_Toc12908497)

[Appendix 1 (only if needed) 46](#_Toc12908498)

# Introduction

## Traffic context

## Connected Cars Context

## Problem Statement

## Structure of the project

# Project Objectives

## Problem Overview

## Project Objectives

The first objective of this project is to design and implement a system able to have a live overview of the Connected Cars Context. This system can be used in taking different actions related to traffic by several actors.

The second one is developing a simulator module in order to test the system. This simulator will mimic genuine traffic participants actions related to traffic. The system is will analyze and report the effect the traffic participants decisions have in the aforementioned context. Thereafter, this module allows the viewing of the traffic evolution under different parameters.

In order to achieve the proposed objectives the following steps need to be systematically followed:

* Design a high-level architecture of the system in its basic form
* Decide on an open source map interpreter and route planning solution
* Setup the open source project for testing and development
* Understand all the features and limitations the open source project has
* Apply the open source solution to all the basic modules of the system
* Design a simulator module
* Decide on a set of cost parameters used for the study
* Decide on a list of metrics to study the simulated context
* Implement the simulator
* Compute and study the results of the simulations under different parameters

## Requirements

### Functional Requirements

Based on the presented objectives, the functional requirements resemble the features of the project. Each requirement is shortly described, but more details about the design and implementation of these features can be found in chapters 4 and 5.

* Component able to generate routes based on different factors
* Component able to hold the status of the traffic and update it when necessary
* Component able to simulate the traffic context
* Simulator should be able to display statistics over the length of the entire simulation
* Simulator should output detailed statistics at the end of the simulation

### Nonfunctional Requirements

# Bibliographic Research

## Navigation Systems

A navigation system is an electronic system that facilitates the navigation. They are composed of both software and hardware. Navigation systems can be offline, online or a combination of both.

Offline navigation systems assume that all the data is on board of the using party and doesn’t need any kind of aid from remote components.

Online systems have components that are located in fixed points and the using party communicates with it in order to receive the navigation information needed.

The hybrid systems can have the map information stored locally but still request routes from a remote component with higher computational power.

### Proprietary Navigation Systems (Closed Source)

A proprietary software is a kind of software that doesn’t allow anybody but the developing team to see or modify the source code. This kind of software is usually developed by companies that sell that product or offer it for free but generate profit out of it.

The most used proprietary navigation systems are the ones that are offline. These come integrated with the car and are provided by the car manufacturer or can be purchased separately as GPS devices from separate manufacturers. These systems are able to provide routes, direction, traffic information (e.g. speed limits) even in the absence of an open internet connection.

On the other hand, with the development of the mobile devices, the online navigation systems got to increase their popularity and reach. By benefiting from internet connection these systems are able to offer the user a wider range of maps. They can also provide way more information on what concerns the current traffic status in certain areas, reviews and even street-level photos. The most relevant examples of online navigation apps are Google Maps and Waze. The fact online factor allows these apps to use user location in order to map the traffic and it allows them to use user data in order to show traffic information like traffic jams, police locations and many others. They offer the possibility to build a route in different ways: shortest distance, shortest time, least consumption, etc. Even if they have all these features, they still take into consideration only the user’s choice when it comes to routing. They do not try to improve the entire context, but only the requesting users’ route, even if they have one of the largest Connected Context when it comes to maps and navigation.

Considering the above mentioned, we are not able to use these systems directly in order to study and improve the Connected Cars Context.

### Open Source Navigation Systems

An open source software is a type of software that are licensed by their developers in such a way that it grants users access to the code and it allows them to study, change and distribute the software, no matter the purpose. Most of these softwares are developed in a collaborative manner, many individual developers helping the process of designing, writing and testing the product.

These kinds of projects are the best choice for somebody that wants to try new routing techniques or even trying a new approach in the context of navigation systems. Writing a navigation software from scratch is a huge project and usually not worth it as the number of open source projects is big enough that it is almost sure that one can find a good starting point for their approach. Most of these projects are designed in such a way that they can be built over.

For this project, from the research of the open source community projects, two of them stood out.

* *OsmAnd*

OsmAnd, presented in [1], is an open source software that runs on Android and iOS devices and is able to process OpenStreetMap data fully offline. OsmAnd has multiple navigation features like turn-by-turn voice guidance, traffic warnings (stop signs, pedestrian crosswalks, speed limits, etc.) and it has features for pedestrians and bicycle riders as well.

The software was developed by a team of over 500 members and the entire source code is available on GitHub [2] under a license that allows any developer to modify and distribute the project, as long as they do it under the same type of license.

The international community that takes part in the development of this product offered a huge collection of functionalities and elements on maps from over 40 countries, but it also brought a large set of undocumented features. The different coding styles approached by the developers affect the clarity of the projects code and makes it difficult for someone to build over it.

* *Itinero*

Itinero, presented in [3], is an open source project built for route planning that was originally designed for logistical optimization. Just as OsmAnd [2], it processes OpenStreetMap data fully offline and it offers step by step instructions in routing. It can provide routes for cars, bicycles, pedestrians, mixed or custom profiles. On top of that, it can be used both as a library and a routing server for other projects, making it perfect for the development of other traffic related projects.

The Itinero project was developed by Ben Abelshausen [4], who is a Belgian optimization algorithms and routing expert. The fact that this project is built by a single person brings ensures that the project is uniformly written and consistently documented.

## Maps and Data

## Traffic

## Routing

### Graph forms

### Algorithms

## Communication Methods

# Analysis and Theoretical Foundation

## Motivation for choosing Itinero

There are multiple reasons for which Itinero was chosen for developing this diploma project.

First of all, Itinero offers the possibility to be used as a library in any .NET project. That means the route computing is extremely customizable and can be used even for our very specific route computing criteria. It also means that the data structures used along the distributed components of this project are exactly the same, using as little effort to communicate as possible. The fact that Itinero is more of a tool than an application itself, allows the further development of a distributed system that uses it in all its components to be very easily extended.

Second of all, the way Itinero manipulates the map information is very well implemented, making it lightweight memory wise and fast even in the scenario of a distributed system, like in the case of this diploma project.  Every data structure offers custom serialization and deserialization, which makes Itinero really fast when it comes to loading and transferring data.

Moreover, the fact that Itinero was developed by a single person (with confirmed design, algorithmic and development skills) ensures that the project is uniformly written. The code flows are consistent and well implemented and coding style is the same in the entire project, so the entire software is comfortable to study and understand, even if it is really heavy when it comes to the content of the code.

To complete the above stated argument, Itinero comes with a good documentation, both in the wiki page and in code comments. Because of this, understanding it and finding answers to almost any technical questions regarding Itinero is not as difficult as it might be in the case of other projects. Besides that, as this diploma project required some really specific tools from Itinero’s kit, whenever something was not clear enough, Ben, the developer of Itinero, was kind enough to provide the needed help.

Last but not least, the technical stack used in the development of Itinero, .NET framework written in C#, completed the list of arguments for taking this decision.

## Conceptual Architecture of the System

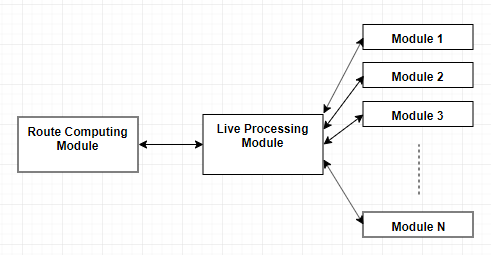


Figure 4.1 Conceptual Architecture of the TCS-3C

As a distributed system, the currently described project is built using modules. The main components are the Live Processing Module and the Route Computing Module. All other possible modules would communicate directly to the Live Processing Module.

### Live Processing Module

The Live Processing Module is the core of this system. It is the module that supports or controls the actions of every other component in the system. It also holds the current state of the traffic.

This module has 2 types of input: route requests and notifications relating traffic. These inputs are received from Active Modules (4.3.2.). This module can also request traffic information from Passive Modules, in order to keep the traffic status, it holds up to date.

When a route request is received, the Live Processing Module requests the latest traffic data from the available Passive Modules (4.3.2.), in order to ensure the reliability of the currently held traffic data. Then it makes a request to the Route Computing Server for a route and after it receives the route, it sends it to the requesting module.

### Route Computing Module

The Route Computing Module is the component of the system that takes care exclusively of computing routes, considering only the c2c infrastructure. It receives as input a route request and it returns the computed route based on the current status of the traffic. Whenever a route request is received, this module loads the traffic data that was updated by the Live Processing module.

### Modules

* Active Modules

Active Modules are the type of modules that take direct actions in or regarding the traffic context. These modules are usually Client applications for Traffic Researchers, Traffic Participants or Traffic Managers. These modules are the ones that provide the input to the Live Processing Module and they process the responses received.

* Passive Modules

Passive Modules are the type of modules that hold a traffic status of their own based on their own input and receive requests from the Live Processing Module. They provided their current status to the Live Processing Server and it will take this information into consideration when updating the general traffic status.

### Example of build system on the conceptual architecture

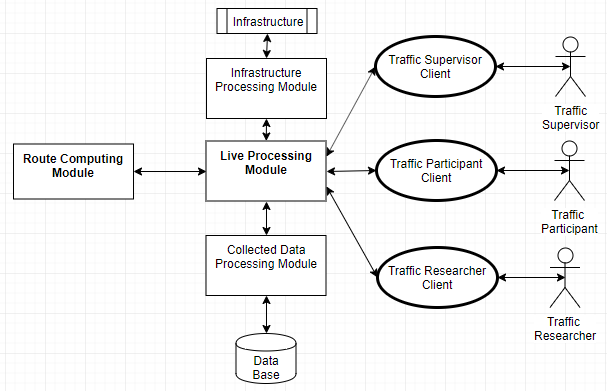


Figure 4.2 Example of architecture based on the conceptual architecture

Figure 4.2. describes a system built on the conceptual architecture presented in the previous pages (4.3). The system above respects the structure proposed and has the main components and 5 other modules (3 active modules and 2 passive modules).

* + **Live Processing Module**

As described at point 4.2.1., the Live Processing Module is the core module of the system. It holds the current traffic status, updates it constantly and communicates with all the other modules of the system.

The Live Processing Module has 2 type of inputs. The first type of input is traffic status updates, received from the Traffic Participant Client Active Module, Traffic Manager Client Active and, on request, from the 2 Passive Modules of the system. The Live Processing Module uses these updates to keep the traffic status up to date.

The second type of input received are requests. The Traffic Participant Client can request routes and the Live Processing Server forwards the request to the Route Computing Module and after it receives the response it outputs that response to the Traffic Participant client. On the other hand, the Traffic Researcher Client can request for statistics on different topics of the traffic status. When that input is received, the Live Processing Server computes and outputs the requested statistics.

* **Route Computing Module**

As described at point 4.2.2., the Route Computing Module has the purpose to answer route requests with a route, based on the current status of the traffic, if it is possible to build one, or with an error message if the route cannot be computed.

**- Active Modules**

* **Traffic Participant Client**

The Traffic Participant Client Module is the client application that can be used by the Traffic Participant stakeholder. This module aids the Traffic Participant in the process of requesting a route.

After a route is received the module presents the Traffic Participant with the route, guides them and sends constant location updates to the Live Processing Module, so the traffic status can be updated.

* **Traffic Manager Client**

The Traffic Manager Client Module is the client application that can be used by the Traffic Manager stakeholder. It offers the Traffic Manager the possibility to bring changes to the traffic context, like blocking a certain area of the city for various events.

* **Traffic Researcher Client**

The Traffic Researcher Client Module is the client application that can be used by the Traffic Researcher stakeholder. It offers the Traffic Researcher the possibility to request the Live Processing Module for various statistics on what concerns the traffic.

**- Passive Modules**

* **Collected Data Processing Module**

The Collected Data Processing Module is the passive module that is responsible with storing the traffic data received or computed by the Live Traffic Server. This module has direct connection to a database in which it stores the collected data, and from which it retrieves and analyses traffic data.  It can provide the Live Traffic Module, on demand, with information regarding the overall usual status of the traffic at certain times/dates in the past.

* **Infrastructure Processing Module**

The Infrastructure Processing Module is the passive module that is responsible for communicating with the infrastructure systems, like live traffic cameras or proximity car counters, and build a status traffic model based on the information received from these systems. It is able to answer the Live Traffic Modules requests with the live status traffic computed using the traffic data from the infrastructure.

* A route request Use-Case, from the perspective of the Live Traffic Module, is presented in the following flow diagram.

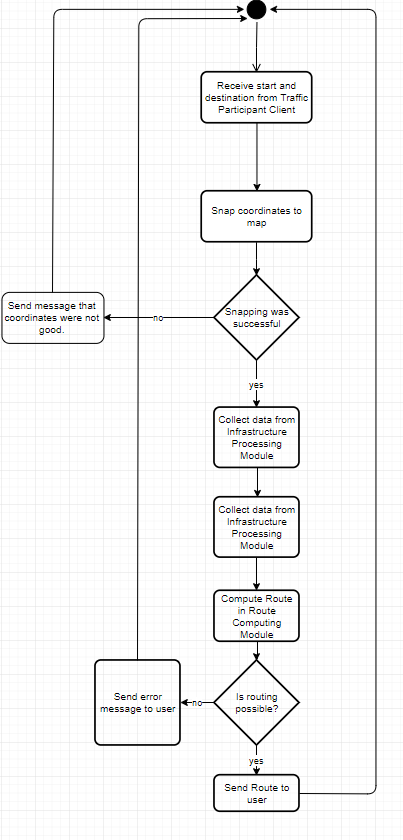


Figure 4.3 Flow diagram of route request Use-Case

## TCS-3C - Design

### Design

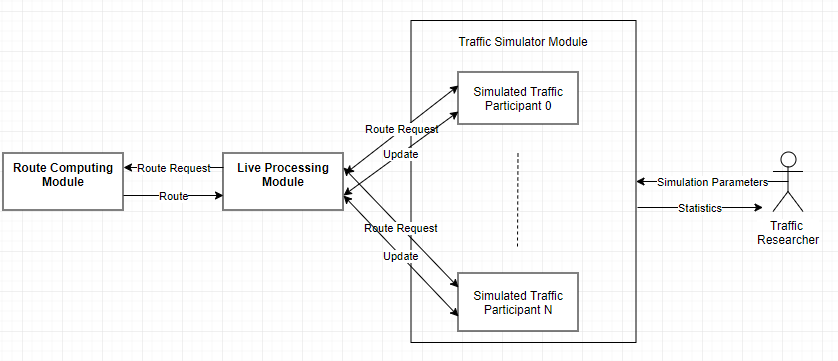


Figure 4.4 Design of TCS-3C

Figure 4.4. describes the architecture of TCS-3C. It respects the conceptual diagram described earlier (4.2.). It has the main modules, the Live Traffic Module and the Route Computing Module, and one active module, the Traffic Simulator Module.

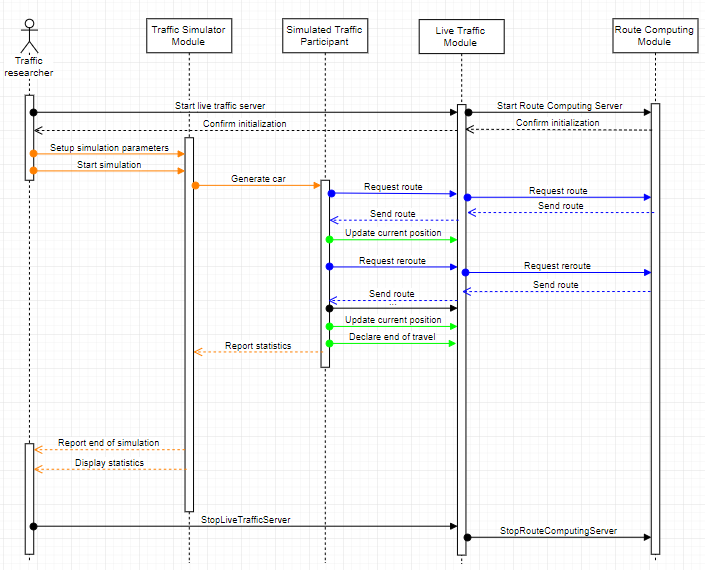


Figure 4.5 Sequence Diagram of a TCS-3C

Figure 4.5. describes a simulation in terms of inter-module communication in the TCS-3C system.

* **Live Processing Module**

As it was previously described, The Live Processing Module is the main module of the system. It has the purpose of connecting all other modules in the system. In this case, the Live Processing Module has a direct connection to the Route Computing Module and to the Traffic Simulator Module.

This module manages the request for routes received from the Simulator Module. Whenever a route request is received, the Live Traffic Module forwards it to the Route Computing Module, together with the current status of the traffic. It then expects a route and forwards it to the simulated traffic participant. These actions can be found colored blue in Fig 4.4.

It maintains the current status of the traffic by updating it periodically when location updates are received from traffic participants simulated by the Traffic simulator. These actions can be seen in color Green in figure 4.4.

* **Route Computing Module**

As described at point 4.2.2., the Route Computing Module has the purpose to answer route requests with a route, based on the current status of the traffic, if it is possible to build one, or with an error message if the route cannot be computed.

* **Traffic Simulator Module**

This module is the interface of the Traffic Researcher in TCS-3C. Its purpose is to generate a traffic context based on a set of parameters and create statistics on the effects of Traffic Participants choices, made in traffic. These effects can be studied by varying a threshold (0%-100%) of people taking the best route over the shortest one and comparing the total times of the simulated contexts.

It simulates traffic participants, giving each one of them a type of behavior, a starting location and an ending location. Each one of these traffic participants requests a route and starts following it, according to their behavior, updating their position and requesting reroutes when necessary.

During the simulation process, the module gathers statistics from the simulated traffic participants and displays them to the user at different points in the simulation and at the end of it.

These actions can be seen in Fig 4.4. colored orange.

### Traffic Data

Previous points stated that the Live Traffic Module holds the traffic status in order to be used when the routing is done. This traffic status is a combination of the already existent map information and the traffic data collected by the Live Traffic Module. Besides the basic data related to the maps, presented at 3.x(TAG), in order to hold the traffic related data, each edge has two additional characteristics:

* car-count: The number of cars that can be found on that edge at the time of the query
* custom-speed: A speed that was computed for that edge

An example can be found in the following image.

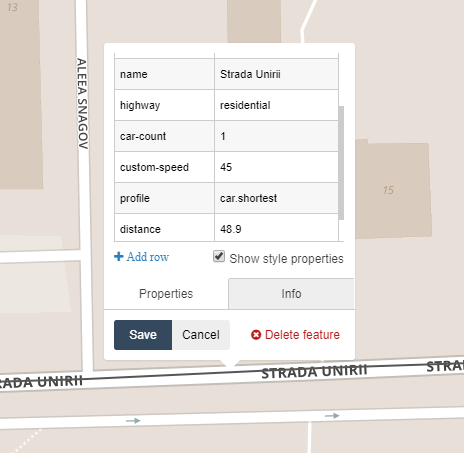
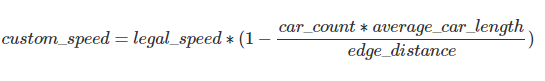


Figure 4.6 Screenshot of partial information recorded about an edge on the map

Normally, the speed associated with an edge is hardcoded and is equal to the speed limit legally imposed for that edge. In the case of TCS-3C, besides the legal speed limit, the number of cars present at one time on that edge and their average length influences the speed that can be reached on that edge. The function that determines the new speed can be easily changed in the system and the current form of this function is the following:



For example, in the picture above the legal speed limit is 50 km/h in that area, the number of cars on that edge is 1, the average car length is 4 meters and the edge distance is 48.9 meters. That gives us, using the formula above, a custom speed of 45.7 km/h, which is rounded to 45.

This function dictates the new speed for every edge that will be used for routing.

### Routing process

The routing process takes place in the Route Computing Module.

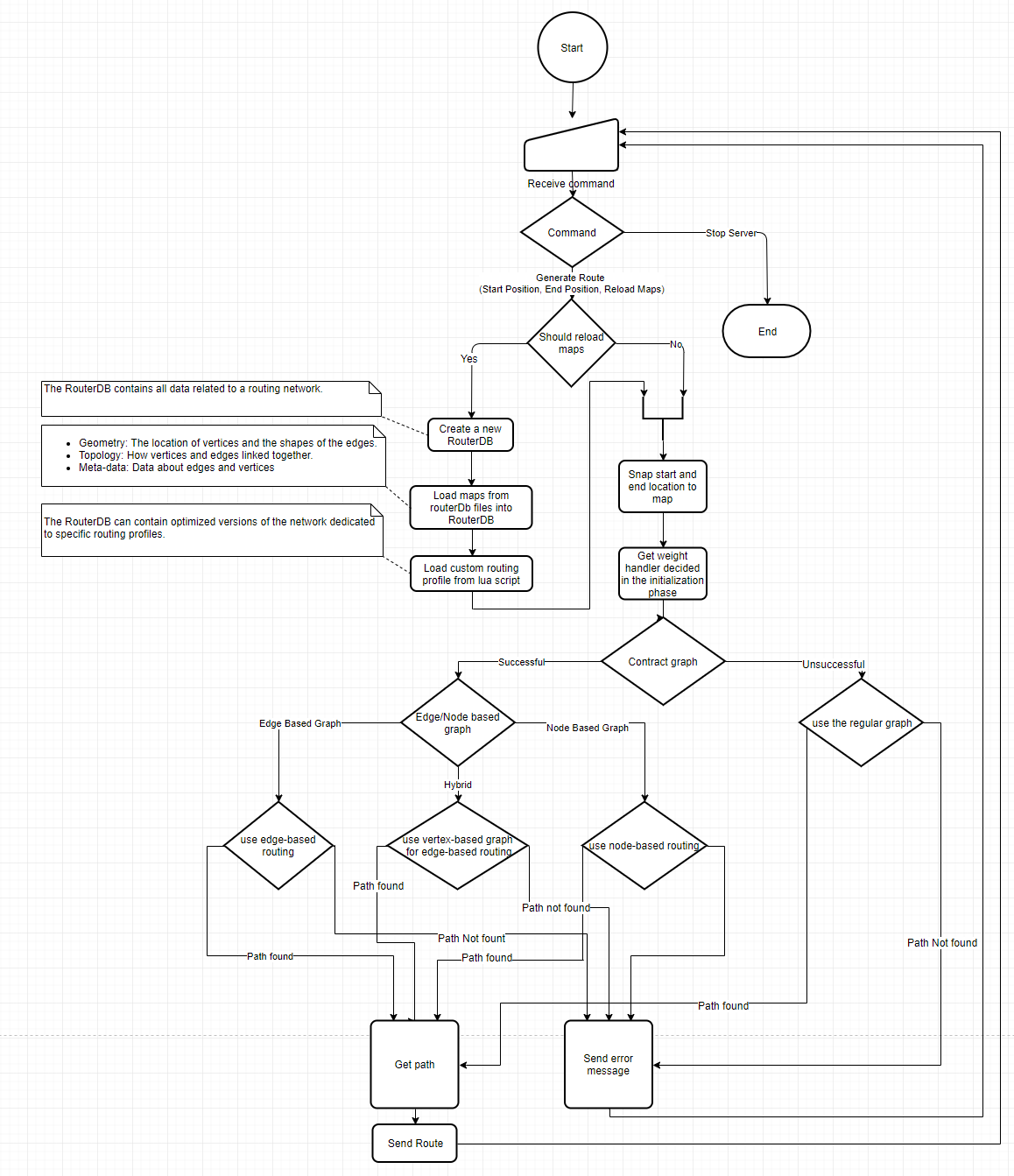


Figure 4.7 Flow Chart of route computing process

The first steps of the process include deciding whether the traffic status should be reloaded or not. If there were enough changes in the traffic context, the routerDb is reloaded.

The second part of the routing process is validating the start and end locations. It consists of checks whether the two pairs of coordinates are included in the map that is currently concerned. If the points are inside the map, the next step is checking if these points can be snapped to the road network in the map or if they are too far.

After the input data is validated, the next step is trying to contract the graph. This step minimizes the size of the graph so less effort is required in the route computing process. This process removes from the routerDb the information that is not used for the current routing profile. For example, if the current routing is only done for cars, all information about pedestrians and bicycles can be removed from the routerDb, as it won’t be used. This step also removes the nodes that have under three neighbors. The contraction also includes transforming the basic graph into an edge-based graph, if possible. This action would make the edges the main point of interest in the routerDb and the nodes would become edges. This would allow speed up in the routing process and would make it easier to model turn restrictions.

The last part of the process is the actual routing. Route computing is done using a custom bidirectional Dijkstra’s algorithm, for each form of the graph. This algorithm There are 2 types of routing in TCS-3C:

* shortest - a bidirectional Dijkstra’s algorithm adaptation that takes as weights the length of the edges and the legal speed limit. This type of routing is used by the greedy Traffic Participants.
* best - a bidirectional Dijkstra’s algorithm adaptation that takes as weights the length of the edges and the custom speed computed for each edge (4.3.2)

# Detailed Design and Implementation

## Components Architecture

In terms of implementation, the TCS-3C is built using Microsoft’s technical stack. The Live Traffic Server and the Route Computing Server are both .NET Core 2.1 APIs, while the Traffic Simulator is a Console Application in .NET Framework 4.6.1. In this chapter each one of these elements will be discussed in terms of implementation and the communication between these modules.

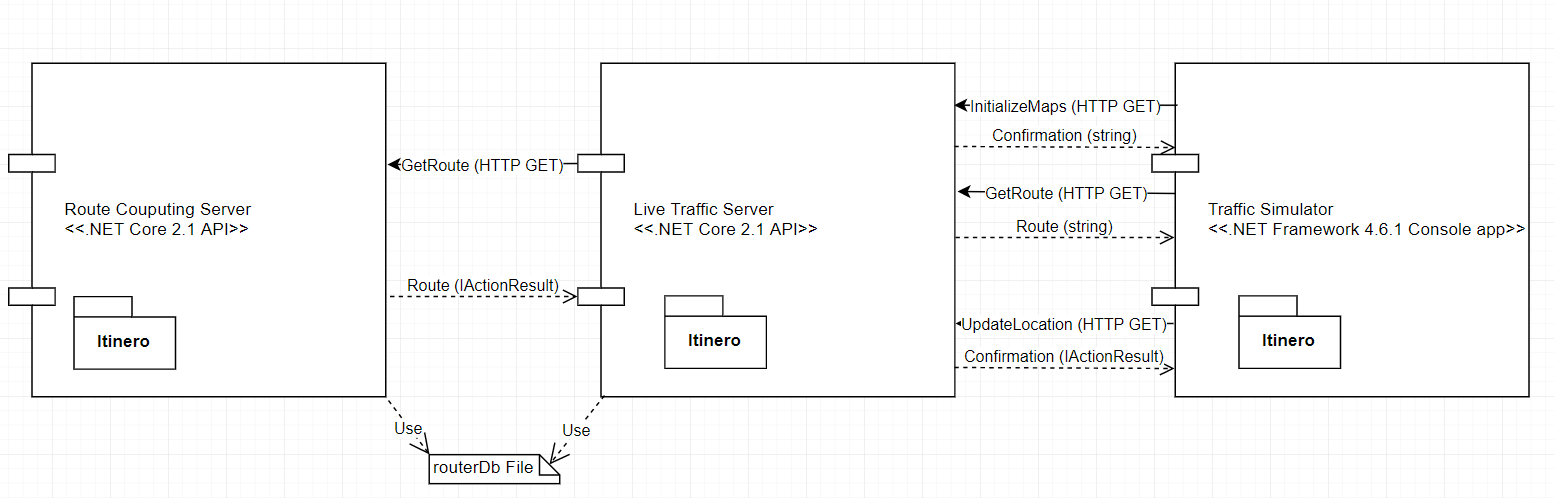


Figure 5.1 High Level design of TCS-3C

In Figure 5.1, a high level design of the system is described. At first sight, it is visible that all the modules in the system have the Itinero package. This package helps keep the datastructures similar over the entire system.

In terms of communication, as discussed at point 3.4, the TCS-3C presents an inter module communication based on the HTTPS protocol. Each one of the main components exposes several endpoints that are accessed via HTTP requests. The only exception concerning the communication is the way the Live Traffic Server communicates the map status: it is done using the routerDb file.

## Live Traffic Server

### Design

The Live Traffic Server is a .NET Core 2.1 API that, as described at point 4.3 is the core server of the application.

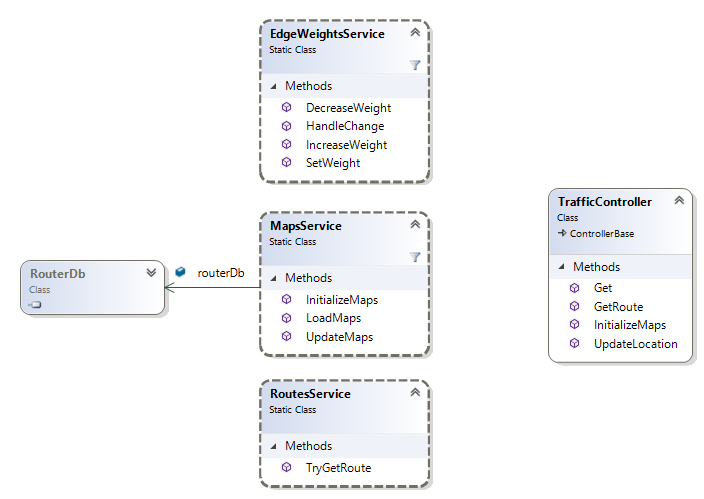


Figure 5.2 High level class diagram of Live Traffic Server (from Visual Studio)

Figure 5.2 presents the high level class diagram of the Live Traffic Server. For the current version of TCS-3C the high level architecture is rather simple. It contains four main classes:

**Traffic Controller**: is an extension of the ControllerBase class in .NET Core. It is responsible with exposing the endpoints and handling the requests. There are four endpoints exposed in the Traffic Controller:

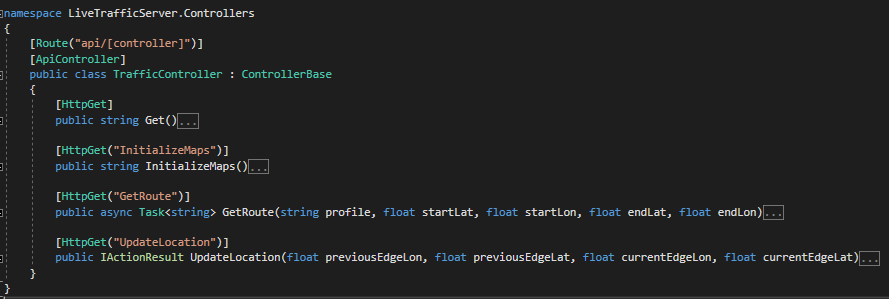


Figure 5.3 Screenshot of the Traffic Controller code, methods colapsed to deffinitions

Figure 5.3 presents the endpoints of the Traffic Controller.

The first one is a test GET endpoint that returns „Live Traffic Server is Running. Waiting for commands.” if the server is running.

The second one is the InitializeMaps GET endpoint which calls the MapsService.InitializeMaps() method. It returns a confirmation string.

The third one is the GetRoute endpoint which receives five parameters: the routing profile (shortest or best), latitude and longitude of the start and end location. This endpoint delivers the resonsability to the RoutesService by calling the TryGetRoute method in the service with the received parameters. It returns the route received from the RoutesService as a string.

The fourth endpoint is UpdateLocation. It receives as parameters the coordinates of the middle of the edge a Traffic Participant is currently leaving and the coordinates of the middle of the edge the participant is moving to. It calls the EdgeWeightsService.HandleChange method with the received parameters. It returns an OK(„succesful”) ActionResult if the update was succesful of a BadRequest(errorMessage) otherwise.

* **MapsService**

The Maps Service is a static service that is responsible with maintaining the current status of the maps and handling all the actions performed with it.

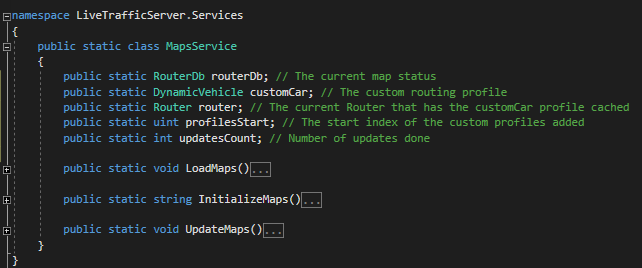


Figure 5.4 Screenshot of the MapsService code, methods colapsed to definitions

Figure 5.4 presents the main elements of the MapsService. This service has five static members and three static methods.

The first static member is the routerDb. This object holds the entire map and profiles associated with each edge. It is initialized in the LoadMaps or in the InitializeMaps method.

The second static member is the customCar DynamicVehicle. It is used for snapping points to the map and contracting of the graph (4.3.3). This is loaded in the InitializeMaps method from the custom .lua script.

The third static member is the router. This member chaches the costomCar profile and it is used for snaping points to the map.

The profilesStart static member holds the index where the custom edge profiles start.

The updateCount static member holds the number of updates done. This member is used for the Live Traffic server to know when to Update the maps in the routerDb file and flag the Route Computing server that it should reload the maps aswell. The maps are updated and this member resets when the UpdatesNecesaryForMapRefresh parameter in Constants.cs is smaller than the number of updates.

The first method is LoadMaps. It is used to load the routerDb from a file.

The second static method is the InitializeMaps method. It loads a new routerDb directly from the pbf format of the map. On the loaded map it adds 500 new custom edge profiles. One for each combination of car-count (0-99) and custom-speed (1-50). It initializes the customCar routing profile, caches it into the router and updates the routerDb file, by calling the UpdateMaps method.

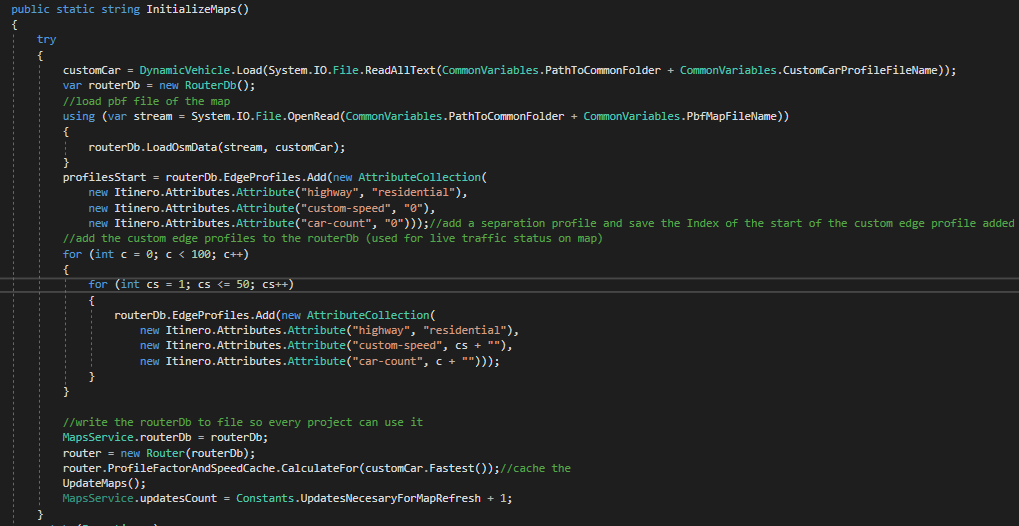


Figure 5.5 Screenshot of InitializeMaps method of the MapsService

Figure 5.5 presents the code described in the previos paragraphs.

The last static method is the UpdateMaps. It is used to serialize the routerDb and write it into the routerDb file.

* **RoutesService**

This static class is has the responsability to handle the routing requests. It has only one static asyncronous method TryGetRoute. This method receives as parameters form the TrafficController a routing profile name („shortest” or „best”) and the coordinates of the start and end point of the request.

Firstly, if the updatesCount in the MapsService is greater or equal than the UpdatesNecesaryForMapRefresh parameter in Constants.cs, The Routes Service calls the UpdateMaps method of the MapsService and flags that the following route request to the Route Computing Server should notify the changing of the routerDb file.

Then, it builds a HTTP request and calls the Route Computing Server for a route. In the case of a BadRequest it updates the maps again and tries a new call by flagging the Route Computing Server to refresh the maps it has. The last step is returning the response received from the Route Computing Server to the TrafficContriller. The code can be seen below.

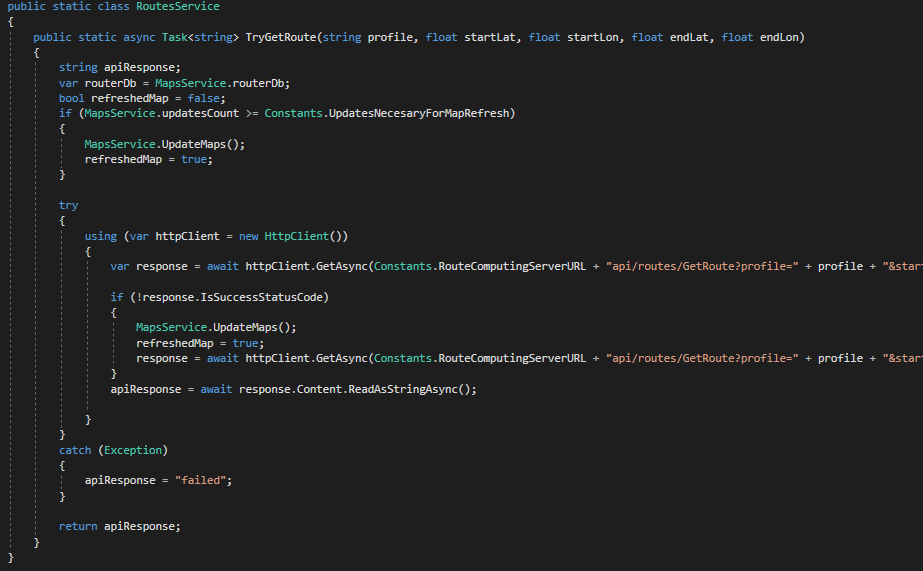


Figure 5.6 Screenshot of the RoutesService code

* **EdgeWeightsService**

The EdgeWeightsService static class is responsible with managing all the actions taken in the map at edge level. This is where the status of the traffic is updated by „moving” cars through the virtual edges.

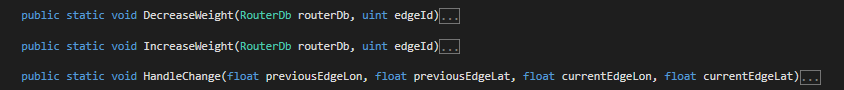


Figure 5.7 Screenshot of EdgeWeightsService methods colapsed to definitions

Figure 5.7 presents the three static methods of the EdgeWeightsService. A change in the traffic context is the moment when a Traffic Participant goes from one edge to another. To deonte that, the TrafficControllers’ UpdateLocation endpoint is called with two coordinates: the middle of the edge the Traffic Participant is leaving and the middle of the edge the Traffic Participant is joining. The TrafficController calls the HandleChange static method of the EdgeWeightsService.

The HandleChange static method takes as arguments two sets of coordinates (previousEdgeLon, previousEdgeLat) and (currentEdgeLon, currentEdgeLat). If the previous coordinate is (0,0) that means a new traffic participant is joining the traffic. If the previous coordinate is not (0,0) but the current one is, that means a traffic participant just left the traffic. If none of the 2 coordinates are (0,0) that denotes only the action of an already existing traffic participant. The HandleChange method snaps the nonzero coordinates to the map and gets the Id of the edges taken into consideration. The next step is decreasing the weight of the previous weight, as a car just left that edge, and increasing the weight of the current edge. This is done using the DecreaseWeight and IncreaseWeight static methods.

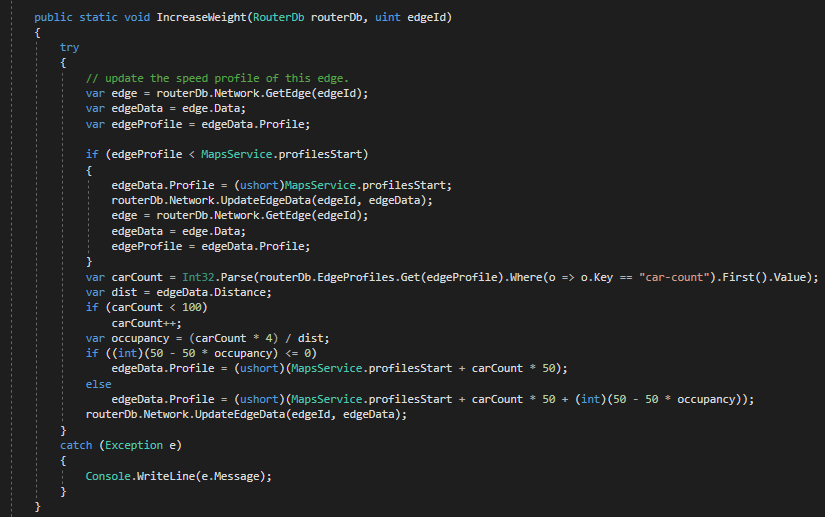


Figure 5.8 Screenshot of the IncreaseWeight method of the EdgeWeights static class

The IncreaseWeight and DecreaseWeight work in a similar manner. They receive as parameters the routerDb and an edge Id. First they retreive the profile of the edge. If this profiles Id is lower than the profilesStart variable in the MapsService it means that is hasn’t been visited before and it is assigned the first custom edge profile (car-count =0, custom-speed=0). The next step is to compute the new number of cars, by retreiving the car-count value and increasing/decreasing it. By aplying the formula described at 4.3.2, a new custom-speed is computed. Using the new car-count and custom-speed, the Id of the new edge profile is computed and assigned to the edge.

### Flow

The live Traffic Server has three main flows, associated with the endpoints exposed by the Traffic Controller: InitializeMaps, UpdateLocation and GetRoute. These flows take different code paths through the previously described Classes and methods in order to return an answer.

* **The InitializeMaps flow**

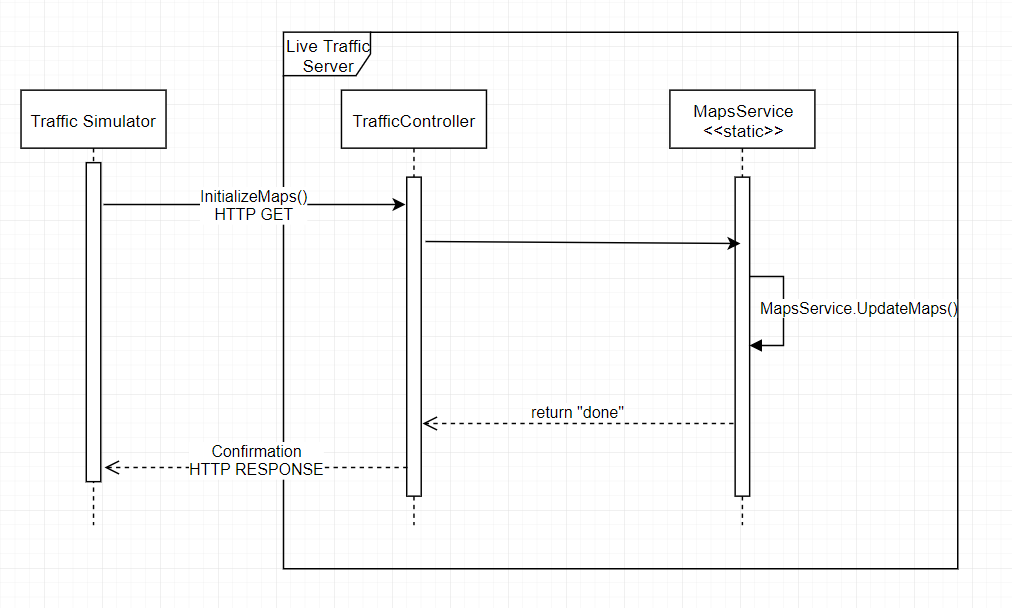


Figure 5.9 Flow Diagram of the InitializeMaps use-case in the Live Traffic Server

In Figure 5.9 the code flow of the InitializeMaps use-case is described. As stated earlier, the command is received via HTTP GET by the TrafficController. The controller delivers the responsability to the MapsService which handles the command as described at point 5.2.1.

* **The GetRoute flow**

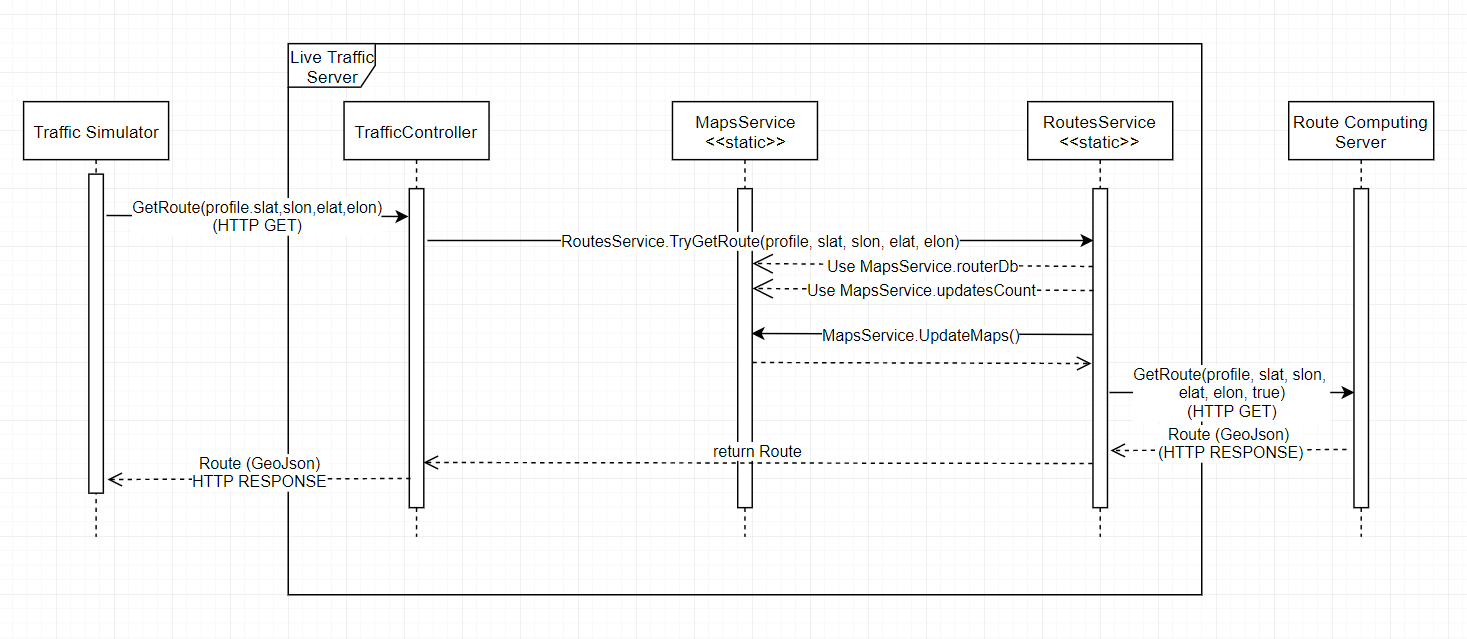


Figure 5.10 Flow diagram of the GetRoute use-case in the Live Traffic Server

In Figure 5.10 the code flow of the GetRoute use-case is described. As stated in 5.2.1, the command is received by the TrafficController via HTTP protocol. The controller delivers responsability to the RoutesService by using the TryGetRoute method. The service uses the routerDb and updatesCount variables from the MapsService and, if necesarry, it updates the routerDb file using the UpdateMaps method in the MapsService. The server then calls the Route Computing server via HTTP GET and returns the response received from it to the controller.

* **The UpdateLocation flow**

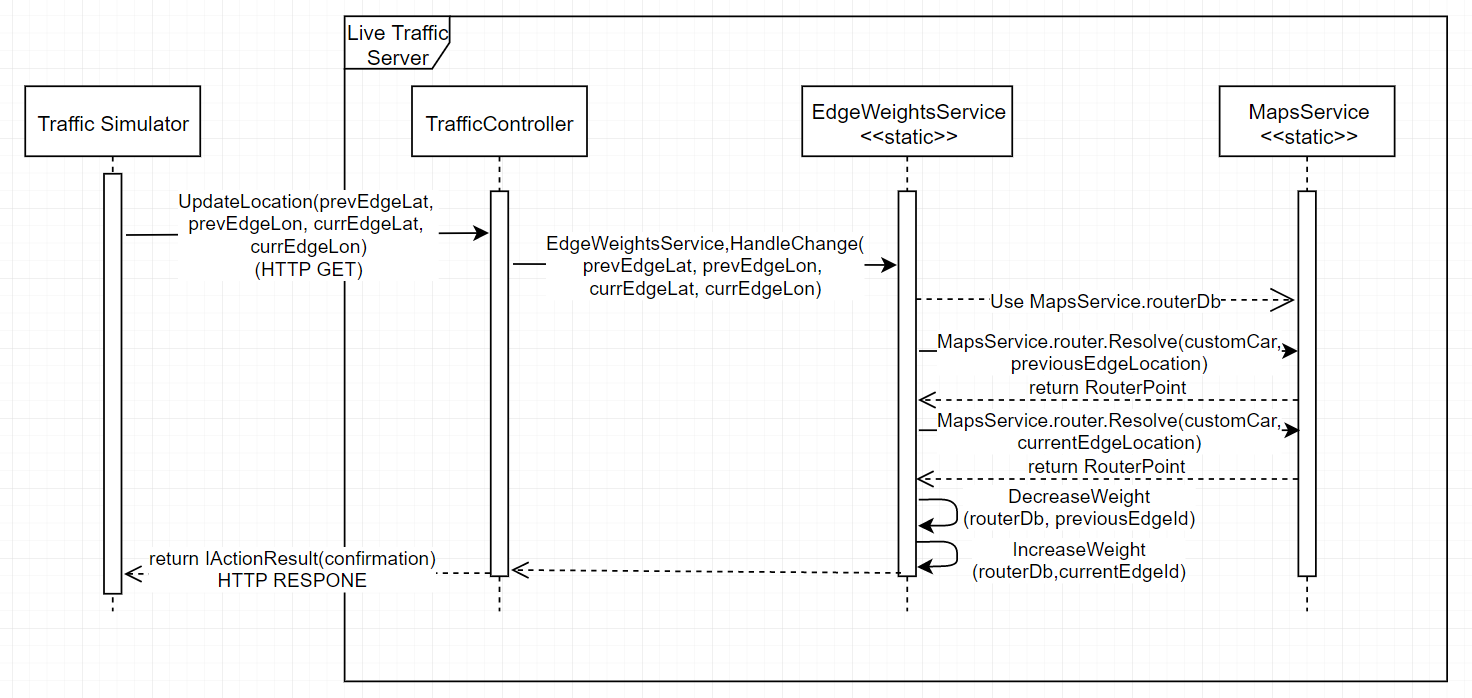


Figure 5.11 Flow diagram of UpdateLocation use-case in the Live Traffic Server

In Figure 5.11 the code flow of the UpdateLocation use-case is described. As stated earlier, the command is received via HTTP GET by the TrafficController. The controller delivers the responsability to the EdgeWeightsService, by calling he HandleChange method. Using the routerDb and the router from the MapsService, the EdgeWeightsService snaps the input coordinates to the map, decreases the weight of the previous edge and increases the weight of the current edge.

## Route Computing Server

### Design

The Route Computing Server is a .NET Core 2.1 API that, as described at point 4.2.2, is the component that is responsible with managing routing requests from the Live Traffic Server.

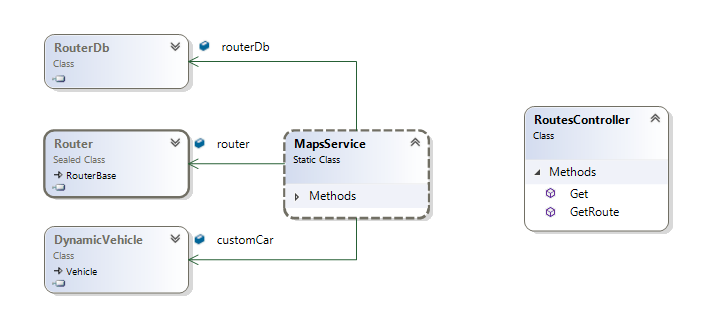


Figure 5.12 High level class diagram of Route Computing Server (from Visual Studio)

In Figure 5.12 the Route Computing Server is presented from a high level perspective. This diagram does not include the class structure behind the RouterDb class, as this diagram presents only the API project. That makes the high level class diagram rather simple.

The server contains 2 main classes: the controller called RoutesController and the static service that handles the maps, called MapsService.

* **RoutesController**

The RoutesController class is an extension of the ControllerBase class in .NET Core. It is responsible with exposing the endpoints and handling the requests. There are two endpoints exposed by the RoutesController.

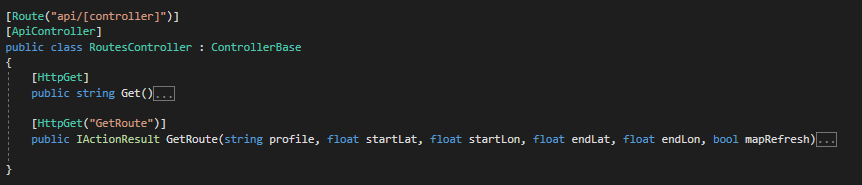


Figure 5.13 Screenshot of RoutesController endpoints code, colapsed to definitions

Figure 5.13 presents the deffinitions of the endpoints in RoutesController as a screenshot from Visual Studio.

The first endpoint, Get(), is a test endpoint. It returns the message "Route Computing Server is Running. Waiting for commands." if the server is running.

The second endpoint, GetRoute, is responsible with receiving and handling route requests. It receives as parameters a routing profile, “shortest” or “best”, four float values that represent the coordinates of the start and end locations of the requested route and a flag announcing the server whether it should refresh its current map status or it should route using the currently loaded one.

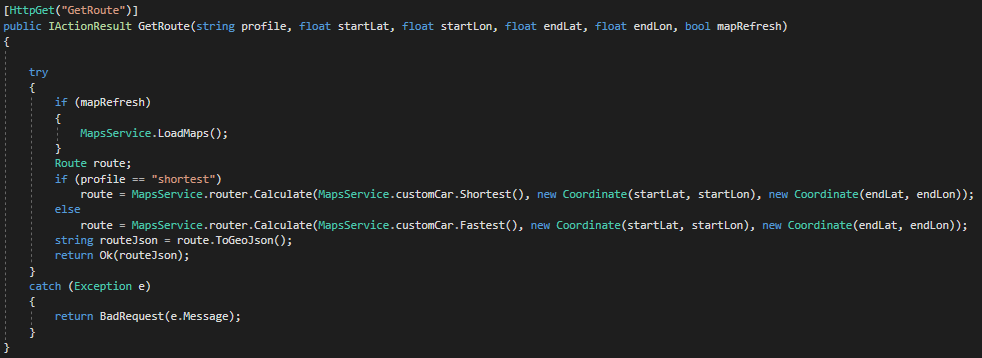


Figure 5.14 Screenshot of the code in the GetRoute endpoint

Figure 5.14 presents the code of the GetRoute andpoint in a screenshot from Visual Studio.

The first thing done is checking wether the mapRefresh flag is active or not. If it is, the LoadMaps method from the MapsService is called, in order to reload the maps.

The second step is computing the route. Based on the chosen routing profile, the controller calls the Calculate method of the router member in the MapsService and sends as parameters the routing profile chosen by the caller, and the two coordinates for the starting and ending point of the route. It then serializes the received route into a GeaoJson format end returns it to the caller with status OK.

All these steps are surrounded with a try/catch clause, so in case any of the steps fail, the error message is returned to the caller with status BadRequest.

* **MapsService**

The Maps Service is a static service that is responsible with loading and maintaining the current status of the maps.

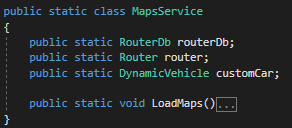


Figure 5.15 Screenshot of the members and method of the MapsService

Figure 5.15 presents the members and the method definition of the Maps service as screenshot from Visual studio. The Maps service has three static members and a static method:

* **routerDb static member**

The routerDb static member, of type RouterDb (from Itinero TAG), is the member that holds all the map and traffic information. It is loaded at the start of the server and is updated constantly using the LoadMaps method.

* **router static member**

The router static member, of type Router (from Itinero TAG), is the member that is used for routing in the routerDb. It is reinstantiated every time the maps are updated in the LoadMaps static method.

* **customCar static member**

The custom car static member, of type DynamicVehicle (from Itinero TAG), is the member that defines the profiles of the routings done in the routerDb. This is loaded from a custom Lua script.

* **LoadMaps static method**

The LoadMaps static method is responsible with refreshing the current routerDb by loading the new sattus from the routerDb file.

* Other functionalities

All the functionalities concerning snaping and routing are implemented in Itinero and are described in TAG.

### Flow

The Route Computing Server has one main flow: the GetRoute flow. This flow Starts when a route request is received by the RoutesController. An overview of the flow can be seen in Figure 5.16.

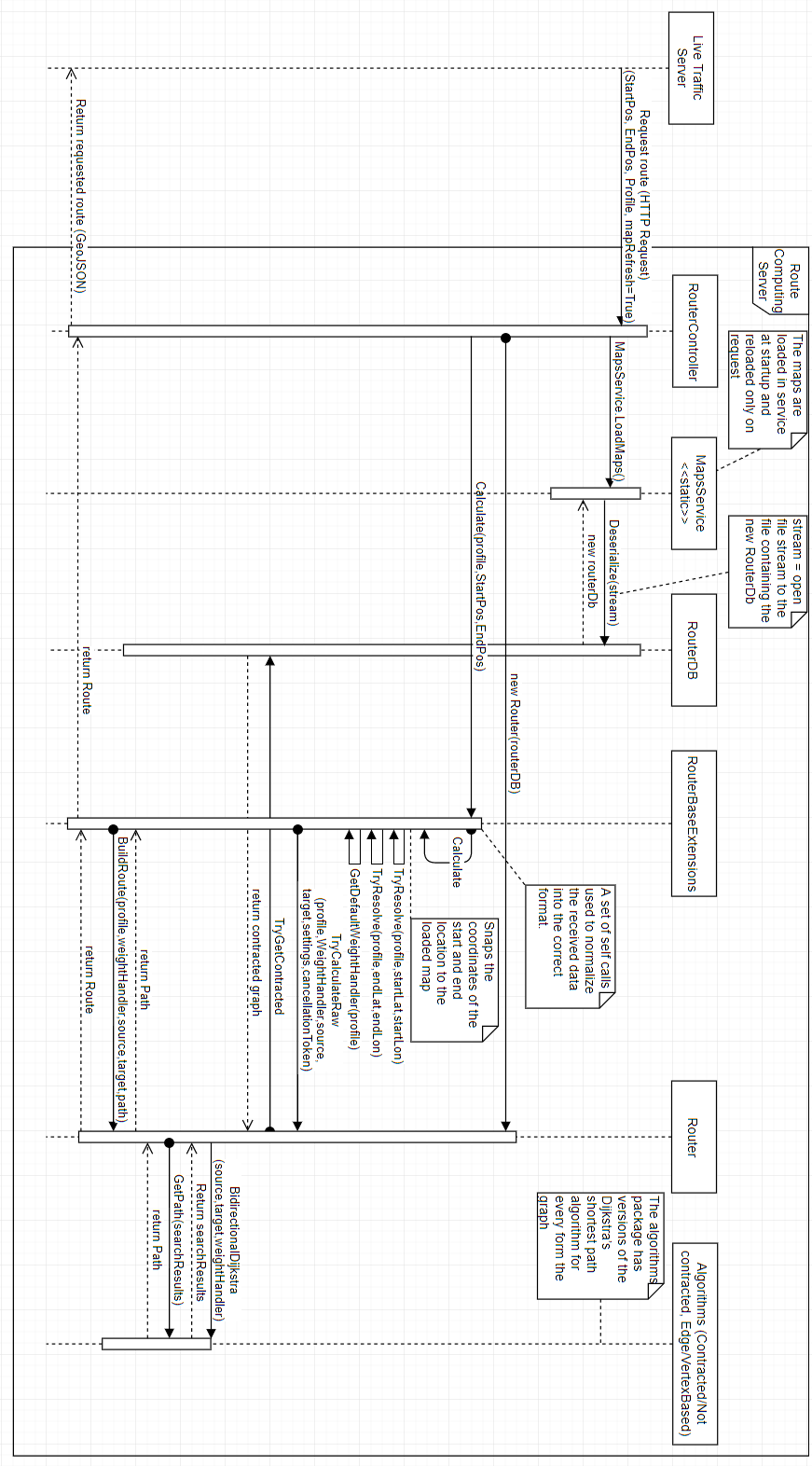


Figure 5.16 Flow diagram of a route request with map refresh

Figure 5.16 is a flow diagram of the code path in the Route Computing Server for a route request. In the use case described the mapRefresh flag in the request is set to true, so the maps have to be reloaded from the routerDb file before starting the route computation.

## Traffic Simulator

### Design

The Traffic Simulator is a .NET Framework 4.6.1 Console Application that is responsible with creating a virtual traffic context. The simulator creates virtual Traffic Participants that mimic the actions real Traffic Participants: they request routes, update their positions and ask for reroutes periodically.

The Traffic Simulator also gathers statistics from the traffic participants and processes them in order to output System and Simulation Statistics (6.4).

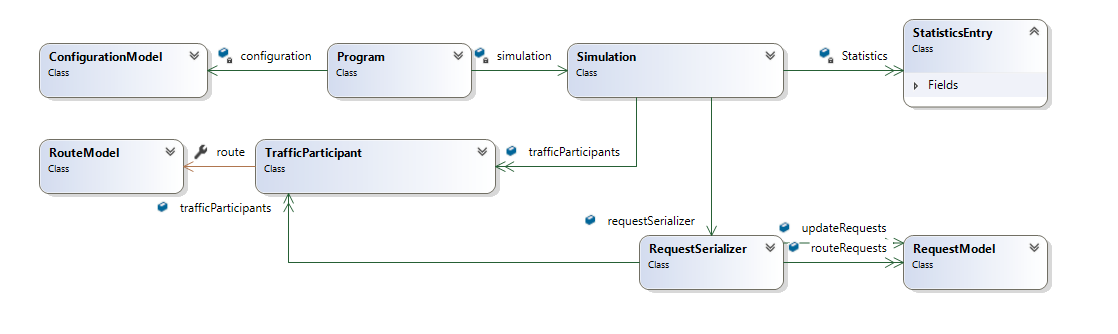


Figure 5.17 High level class diagram of Traffic Simulator (from Visual Studio)

In Figure 5.17, the Traffic simulator is described using a high level class diagram. The simulator is composed of four main classes and uses 4 types of models.

* **The Program class** is the first class that is loaded when the simulator is started.

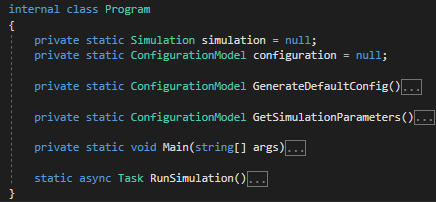
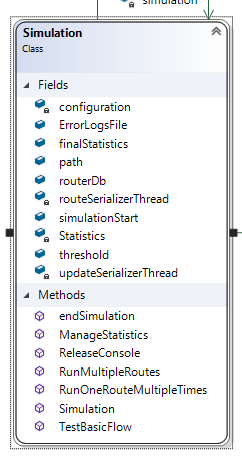


Figure 5.18 Screenshot of the structure of the Program class, methods colapsed to definitions

In Figure 5.18 the code of the Program class is presented usign a screenshot from Visual Studio. The class contains two static members and three static methods.

* simulation – instance of the Simulation class that represents a simulation with the parameters fond in the configuration
* configuration – instance of the ConfigurationModel class that holds all the information about the curently run simulation
* GenerateDefaulConfigs – method that generates a basic configurations file
* GetSimulationParameters – method that reads the simulation parameters from the SimulationParameters.json file and popultes the configuration field
* Main – method that is run when the project is built
* SunSimulation – method that instantiates the simulation parameter and runs the proper kind of simulation based on the simulation parameters



* **The Simulation class** contains all the members and methods necessary for a simulation. It also contains separate methods that define the flow for every type of simulation. A screenshot of the code of the Simulation with the methods colapsed to definitions can be found in TAG. The class contains 10 fields and 7 methods. The most important ones are:
  + trafficParticipants – list of TrafficParticipant instances that represent the simulated traffic participants
  + routeSerializerThread
* **The TrafficParticipant class**
* **The RequestSerializer class**

### Flow

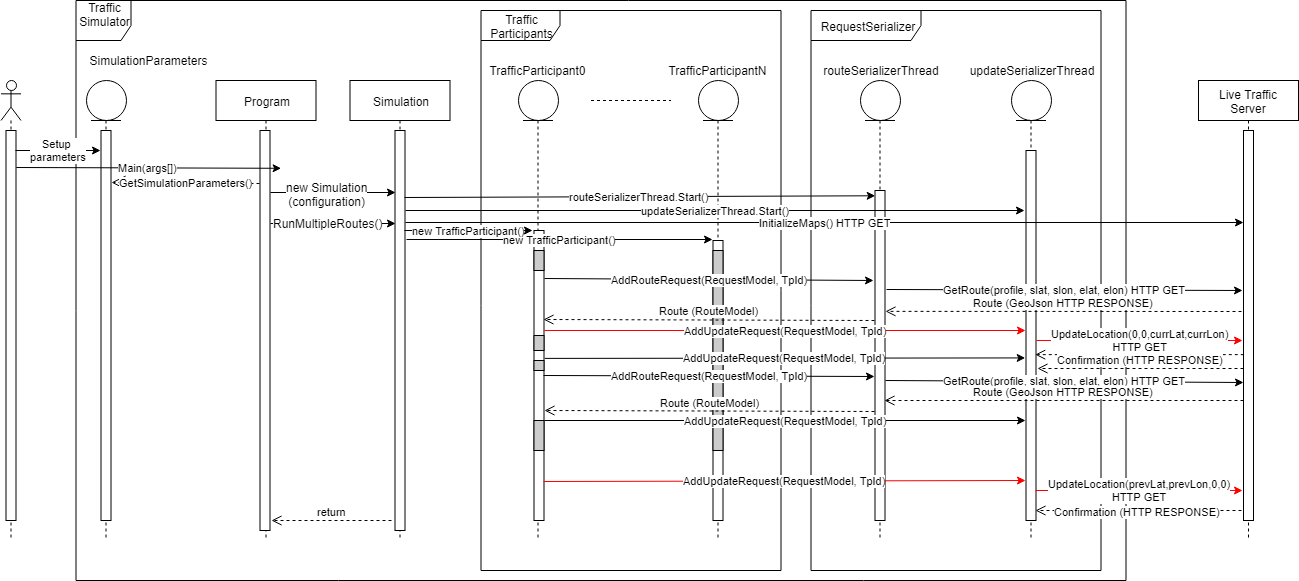


Figure 5.20 Sequence Diagram of the Traffic Simulator code

# Testing and Validation

## Hardware

Testing was done using 3 different setups:

1. Intel Core i5-4460 3.20 GHz, 16 GB RAM, 100 GB free SSD storage
2. Intel Core i7-4720 2.60 GHz, 16 GB RAM, 60 GB free SSD storage
3. Intel Core i5-6700 2.60 GHz, 16 GB RAM, 100 GB free SSD storage

## Metrics

In order to study the efficiency of the presented project, metrics should be defined for both the system itself and the effects of the system in the connected cars context, computed using the implemented Traffic Simulator (4.3.1.).

### System metrics

The system metrics are the characteristics of the system, measuring the way this works. These metrics would show the capabilities of the system in any module configuration. They focus especially on the Live Traffic Server and on the Route Computing Server.

* Live Traffic Server route request response time: The time it takes for the Live Traffic Server to respond to a request for a route.
* Route Computing Server route request response time: The time it takes for the Route Computing Server to receive a route request, compute that route and respond to the request.
* Live Traffic Server Update request response time: The time it takes for the Live Traffic Server to respond to an Update Request.

### Simulation metrics

The Simulation metrics are the characteristics of the results obtained by running different scenarios in the Traffic Simulator. These metrics are used to analyse and compare different approaches of implementation or different traffic scenarios.

* Simulation length: the amount of time it took for the Traffic Simulator to finish the job
* Total time: the sum of the durations of all routes simulated
* Average route time: the average duration of the routes simulated
* Average speed: the average speed of the Traffic Participants simulated
* Average touched features: the average number of edge sets touched by the Traffic participants while following the simulated routes
* Average route request wait time: the average time the simulated traffic participants had to wait for one route request
* Average update request wait time: the average time the simulated traffic participants had to wait for one update request

## Scenarios

## Results

# User’s manual

This chapter comprises the installation of all the components of the developed system as well as methods to test each one of these components. Furthermore, all the development and testing were done on only one machine at a time, with some supplementary steps the system can be easily distributed. The choice of testing only on one machine at a time was made so the speed of the simulation was as big as possible.

## Environment

For running of the current version of the system the user needs the Visual Studio IDE. The development was done using Microsoft Visual Studio Express 2017. Using a different version of Visual Studio might create issues. .NET Core 2.0 was used in the development of the Live Traffic Server and Route Computing server so the user must make sure the .NET Core SDK is installed, by checking the “.NET Core cross-platform development” in the installation/modifying wizard of Visual Studio.

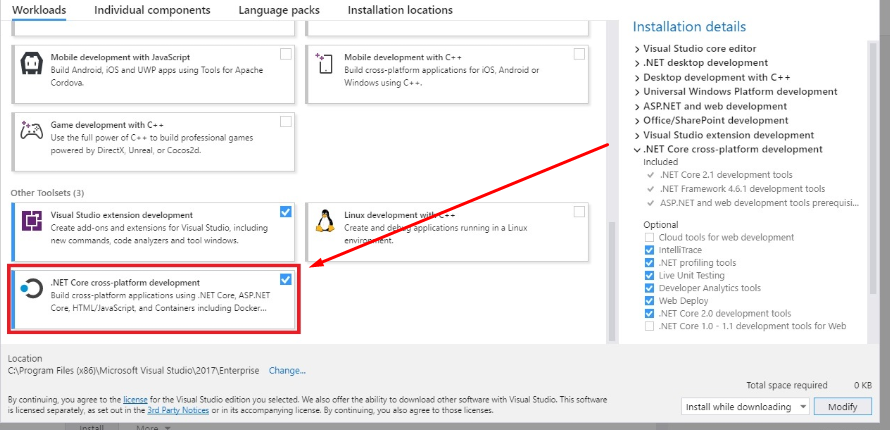


Figure 7.1 Visual Studio installation wizzard

## Dependencies

Each module of the system uses a few external elements. These elements have to be present in order for the system to work. In the case of a distribution of the system over multiple machines the following steps must be done for each machine.

* Download the project from [5]
* Create a folder destinated for map related files
* Create a folder destinated for the results of the simulation
* Download the map of the country of focus from [6]
* Reduce the map to the specific area of focus using instructions from [7] (the smaller the area, the faster the system)
* Add the .pbf file of the area to the folder created at step 1.
* Make sure that the CommonVars.cs and SharedAssemblyVersion.cs files are in the same folder with the components of the system on the machine.

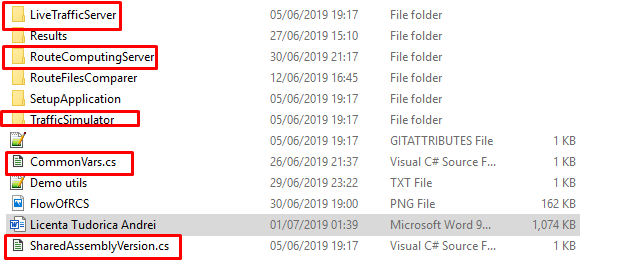


Figure 7.2 Screenshot of the folder containing components and common files

* Open the CommonVars.cs file
* Change the value of “PathToCommonFolder” to be the path of the folder you created for the maps
* Change the value of “PathToResultsFolder” to be the path of the folder you created for the results
* Add the custom-car.lua file, that can be found in the root folder of the system, in the folder destinated for maps
* Change the “PbfFileName” value to be equal to the name of the final map to be used
* Change the “RouterDbFileName” to match with the target area
* Run the SetupApplication found in the same folder with all the components

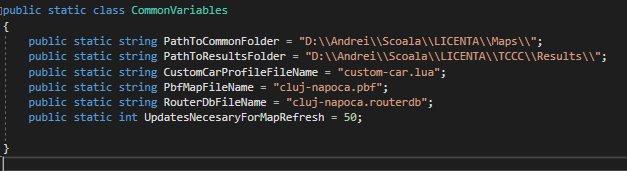


Figure 7.3 Screenshot of CommonVars.cs in current version

## Route Computing Server

The Route Computing Server code can be found in the “RouteComputingServer” folder of the downloaded system. After all the steps at 7.2. are completed, run the .sln file and compile the project. This should open a page saying the Route Computing Server is running and waiting for commands. Testing can be done by building a link containing 2 locations on the chosen map.

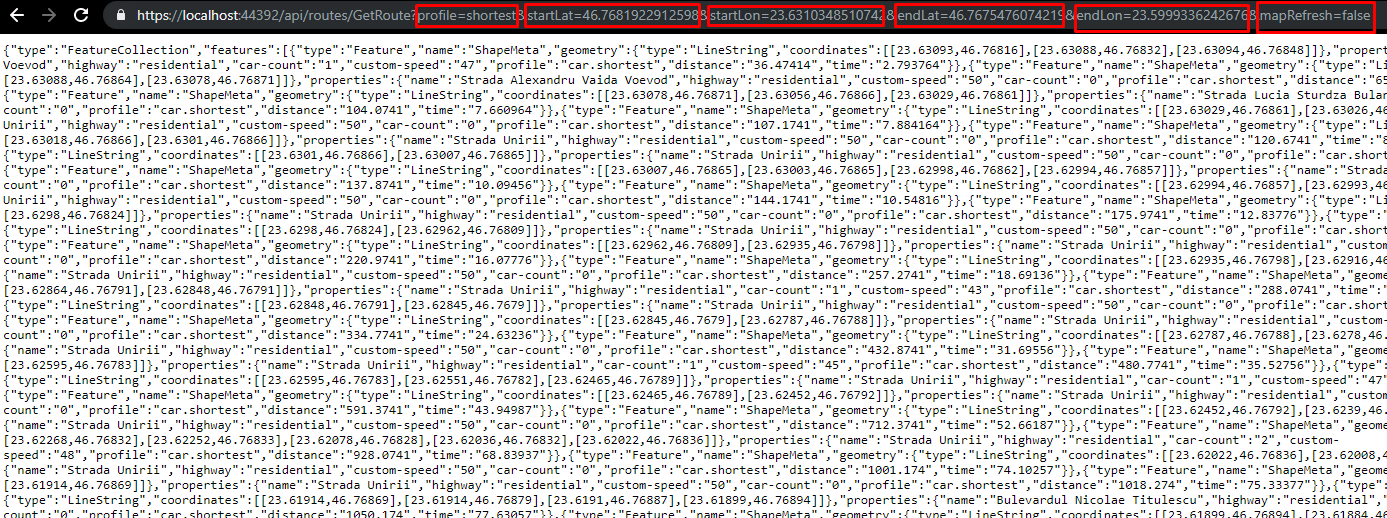


Figure 7.4 Screenshot of test run for the Route Computing Server

Figure 7.4. presents an example of usage for the following link:

https://localhost:44392/api/routes/GetRoute?profile=shortest&startLat=46.7681922912598&startLon=23.6310348510742&endLat=46.7675476074219&endLon=23.5999336242676&mapRefresh=false. The response of the Route Computing Server is a GeoJson string containing a route. By pasting this into [8] the visualization of this route is obtained.

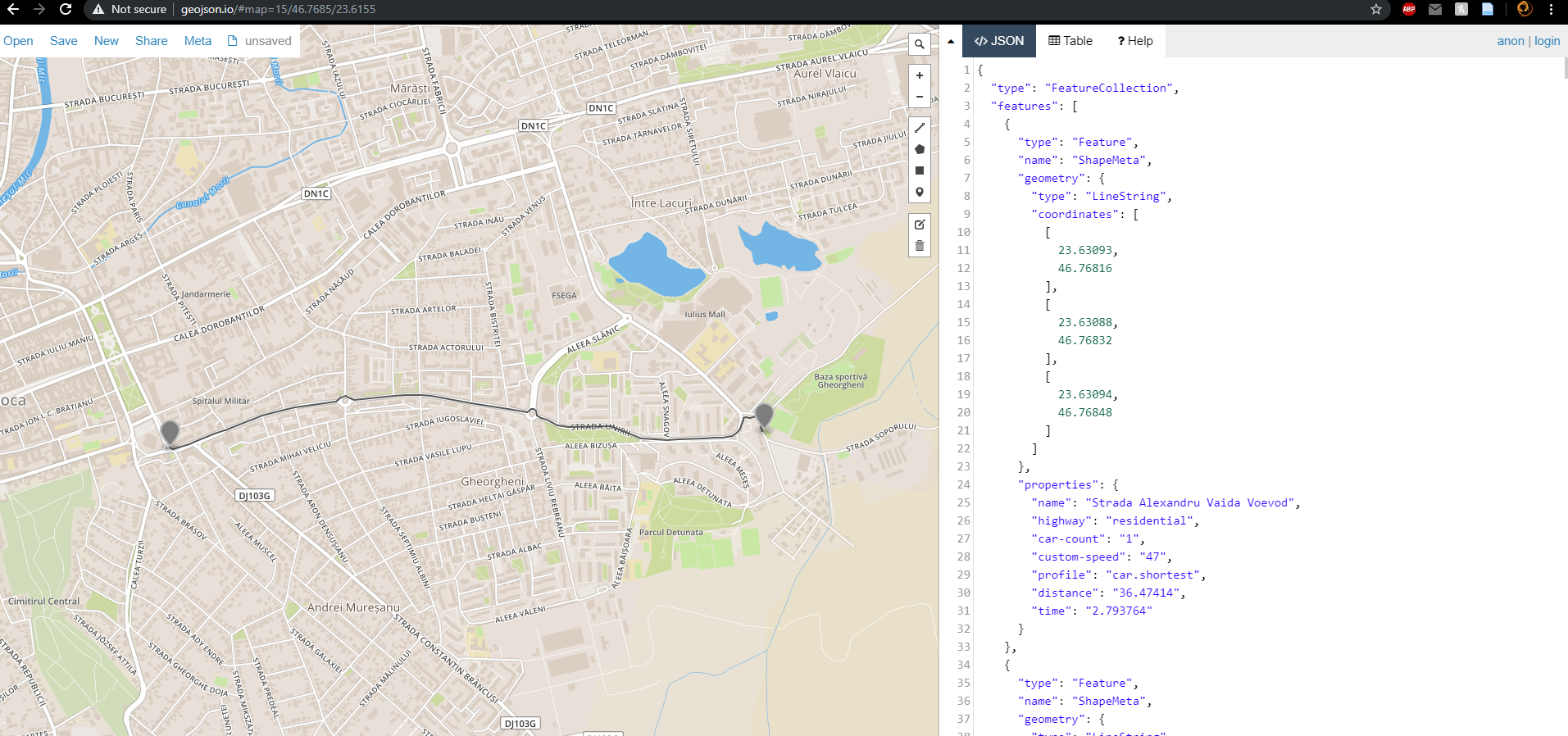


Figure 7.5 Screenshot of the visualization of the response received by the command in Figure 7.4.

For best performance the Route Computing server should be done in Release mode. This can be done by selecting „Release” in the run mode dropdown menu and pressing CTRL+F5.

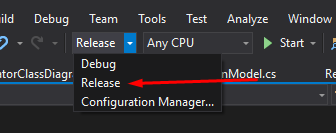


Figure 7.6 screenshot of run mode dropdown menu in Visula Studio

## Live Traffic Server

The Live Traffic Server code can be found in the “LiveTrafficServer” folder of the downloaded system. After all the steps at 7.2. are completed, open the .sln file. Open the Constants.cs file in Visual Studio and modify the value of “RouteComputingServerURL” so that it matches the URL where the Route Computing Server. In a local environment this is "https://localhost:44392/".

When the project is run a page saying “Live Traffic Server is Running. Waiting for commands.” should open.

The server initializes the maps by itself at start up. The functionality can be tested by building a link containing 2 locations on the chosen map just like the one for the Route Computing server.

Example: https://localhost:44351/api/traffic/GetRoute?profile=shortest&startLat=46.7681922912598&startLon=23.6310348510742&endLat=46.7675476074219&endLon=23.5999336242676

The visualization of the answer follows the same steps as the ones for the Route Computing Server.

For best performance the Route Computing server should be done in Release mode. This can be done by selecting „Release” in the run mode dropdown menu (Figure 7.6) and pressing CTRL+F5.

## Traffic Simulation

The Traffic Simulator code can be found in the “TrafficSimulator” folder. After all the steps at 7.2. are completed, open the .sln file. Open the SimulationParameters.json file in Visual Studio and modify the value of the “LiveTrafficServerUri” such that it matches the URL where the running Live Traffic Server can be found. In a local environment that link should be "https://localhost:44351". Modify the other simulation parameters so they match the desired simulation.

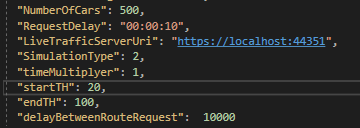


Figure 7.7 Example of simulations parameters.

Run the simulator in release mode to get live statistics about the simulation.

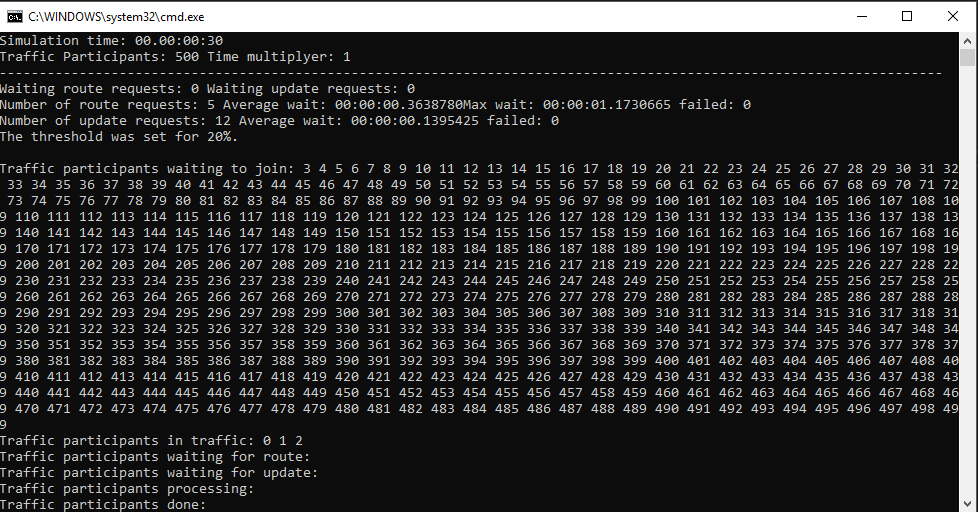


Figure 7.8 Screenshot of simulator run in Release mode

Run the simulator in Debug mode to follow the Simulation step by step. This might be really hard to followas the updates move quite fast.

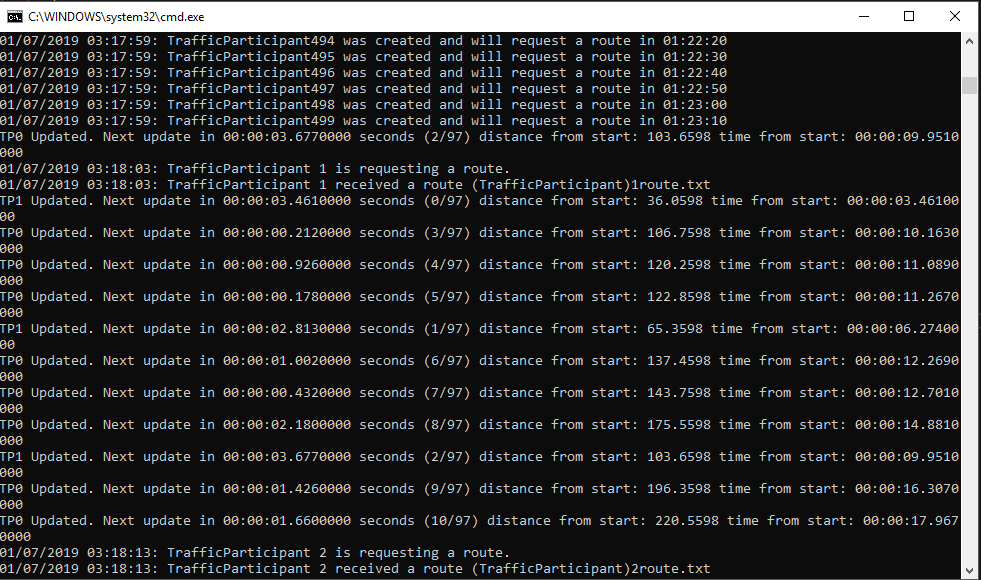


Figure 7.9 screenshot of simulator run in Debug mode

The results of the simulation can be found on the console during the simulation and at the end of it. Also, at the end of the simulation, in the Results folder that was created at 7.2, the simulator creates files for the entire simulation statistics and for each traffic participant.

For best performance the Simulator should be done in Release mode.

## Distribution

In order for the system to be distributed a couple of changes should be made.

First of all, the live traffic server’s code should be modified so it writes the routerDb in the route request to the Route Computing Server, not in a local file. Also, the Route Computings code shoud be modified such that the Route Computing receives a routerDb in the route request body and deserializes it in the local RouterDb structure.

The second part would be changing the IIS settings and making the server discoverable, just like in [9]. This can be done for both the Live Traffic Server and for the Route Computing Server.

# Conclusions and Future Work

About. 5% of the whole.

In this chapter you present:

* A summary of your contributions/achievements,
* A critical analysis of the results achieved,
* A description of the possibilities of improvements/further development.

Reamintesc problemele prin prisma solutiei (daca e posibil comparand cu alte solutii)

Concluzionez cu un statement puternic “in acest fel mi-am atins obiectivul de a …”

Future work

# Bibliography

|  |  |
| --- | --- |
| [1] | “OsmAnd Web page,” [Online]. Available: https://osmand.net/. |
| [2] | “OsmAnd Repository,” [Online]. Available: https://github.com/osmandapp/Osmand. |
| [3] | “Itinero Web Page,” [Online]. Available: https://www.itinero.tech/. |
| [4] | B. Abelshausen, “Linked In page,” Itinero, [Online]. Available: https://www.linkedin.com/in/benabelshausen/?originalSubdomain=be. |
| [5] | A. Tudorica, “Licence Thesis Repo,” [Online]. Available: https://github.com/andreitudorica/UTCN\_LICENTA. |
| [6] | “OpenStreetMap Data Extracts,” [Online]. Available: https://download.geofabrik.de/. |
| [7] | “OsmConvert Wiki,” [Online]. Available: https://wiki.openstreetmap.org/wiki/Osmconvert. |
| [8] | “GeoJson,” [Online]. Available: http://geojson.io. |
| [9] | “Remote Access to Local ASP.NET Core Applications,” [Online]. Available: https://blog.kloud.com.au/2017/02/27/remote-access-to-local-aspnet-core-apps-from-mobile-devices/. |

# Appendix 1 (only if needed)

…

Relevant code sections

…

Other relevant info (proofs etc.)

…

Published papers (if any)

etc.