**WEST UNIVERSITY OF TIMIŞOARA**
**FACULTY OF MATHEMATICS AND COMPUTER SCIENCE**
**BACHELOR STUDY PROGRAM: Computer Science**

# BACHELOR THESIS

**SUPERVISOR: Todor Ivașcu**
Prof./Conf./Lect. Dr. Firstname Lastname

**GRADUATE: Mihai Andrei Gherghinescu**
Firstname Lastname

**TIMIŞOARA**
**2023**

**WEST UNIVERSITY OF TIMIŞOARA**
**FACULTY OF MATHEMATICS AND COMPUTER SCIENCE**
**BACHELOR STUDY PROGRAM: Computer Science**

# Traffic Manager

**SUPERVISOR: Todor Ivașcu**
Prof./Conf./Lect. Dr. Firstname Lastname

**GRADUATE: Mihai Andrei Gherghinescu**
Firstname Lastname

**TIMIŞOARA**
**2023**

## Abstract

The paper introduces an intelligent system for traffic signal applications that is designed to be as adaptive as possible and combines multiple technologies. At base, it is programmed on an micro controller and it uses fuzzy logic, but can be upgraded. This can be done by linking other hardware components to it with the help of a configuration file. So we can utilize multiple technologies like image recognition or speed/radio detection and get the best from all of them.

The system aims to be used in real time traffic scenarios, as an affordable and upgradable option, to decrease the waiting time at crossroads. This would lead to shorter travel times from point A to point B and would have a significant impact not only on our daily lifes, but on the environment as well.

# Contents

## List of Tables

# List of Figures

# 1 Introduction

Intersections are the points where two or more routes overlap with one another. As a result, for drivers, they act as an obstruction for smooth traffic flow especially within urban areas where the infrastructure demands them the most. This leads to traffic interruption, congestion, and poor control and management of the traffic.

To minimize the amount of time lost at crossroads while maintining the safety of the drivers, Intelligent Transportation Systems (ITS) were developed. One major impact that can be seen as a result is the reduction of $CO_2$ car emissions. You may think that this wouldn't be relevant when the era of combustion engines comes to an end, but there are so many other bennefits apart from this. For example, the less time it takes for resources to travel from a provider to a manufacturer, the faster the production can start so it also has a huge impact on the global economy.

As it's occurrence is noticed the most in urban areas, the problem is currently being tackled in concordance with the smart city concept. A smart city is defined as an ultra-modern urban area that aims to improve the overall life quality of citizens. It is based on different arhitectural approaches that involve modernizing and upgrading our current environment by applying new concepts and technologies on top of the ones currently in use.

# 2 Overview of the Technology Developed

To better understand the problem we must first know what solutions were used to fix the issue through out the time and how they evolved.

The first solution that was put in use was created long time ago and required the use of actual human resources to control the traffic. Policemans were responsible to manage the traffic at junctions. Once with the technological era, we were able to remove the need of this kind of resources by creating autonomous systems that alternate between STOP and GO phases. The design of the systems was pretty simple and straightforward, we would use lights to signal those phases. Red lights were matched with the STOP phase, green ones with the GO phase. To prevent collisions and maintain drivers safety amber was introduced as well. It notifys the drivers that STOP and GO phases will soon switch and they should take the corresponding measures.

As the time passed by and the number of cars on the road increased drivers started experiencing traffic jams scenarios. Also there was clearly a better option then having a standard fixed time to change the lights. Many times the green signal was turned on even though there were no cars crossing that given road while on the other crossing roads there were actual drivers waiting. Something had to be done, a more efficient way had to be found, but at what cost? If we wanted to develop better solution we would need the help of expensive hardware components.

We reach the point in time when a new discovery was made: bugdet units based on microcontrollers, sensors and recievers [2] that were cable of running advanced algorithms (Figure 1). So all that was left was to develop new solution and use them to design an intelligent Traffic Light Control System (TSC)

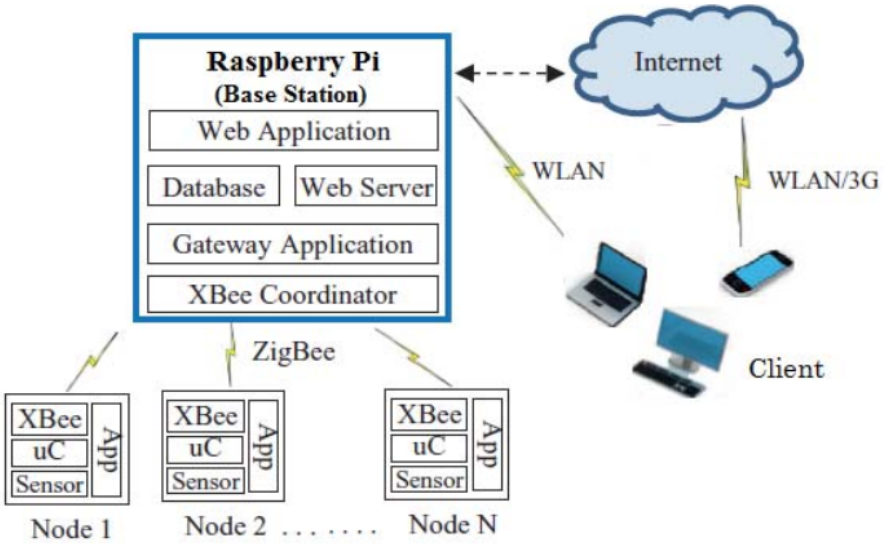



Figure 1: Rasberry PI based system ©

By the time writing, TSC is an active research topic that proved to be really challenging. Continuous work has been done on designing and developing intelligent traffic signal control systems that could address the issue.

So new methods and advanced systems based on fuzzy logic, evolutionary algorithms, image processing, neural network and many other algorithms have been proposed by the researchers to solve this TSC problem [7]. All of the methods aimed to reduce the overall waiting time and prevent traffic jams from occurring while maintaining drivers safety. In this section we will discuss about some of the methods that were put in to use and their ups and downs. After describing each and every tehnology we will have a brief comparison to highlight some of their characteristics.

## 2.1 Object recognition based systems

One way to determine traffic scenarios was done with the help of cameras. To be more specific we wanted to calculate the traffic waiting queue by using the images provided by the cameras. This problem was addressed by the help of vehicle tracking and image segmentation algorithms.

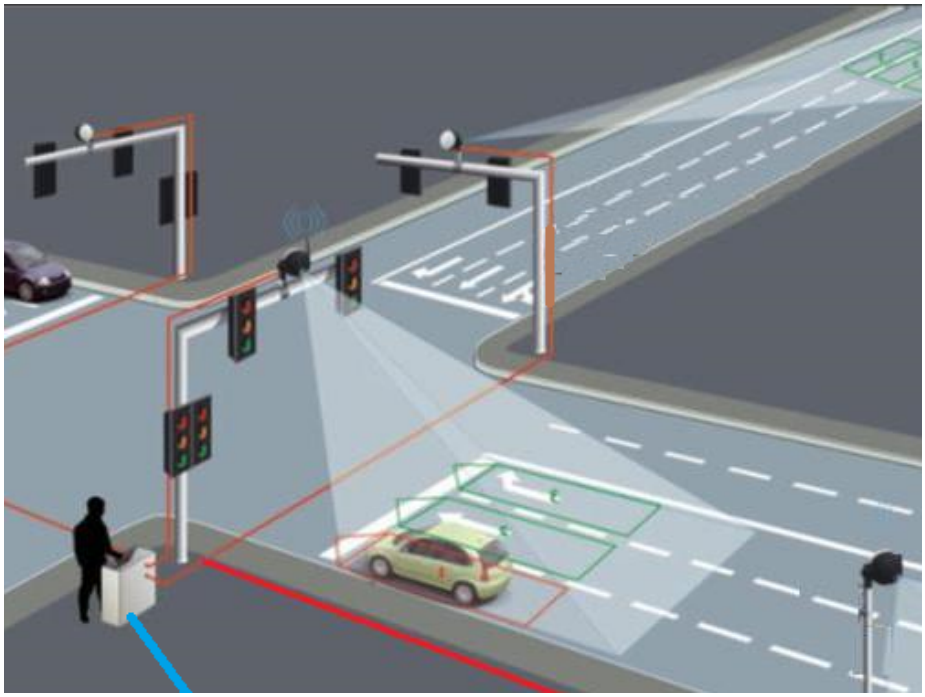


Figure 2: Object Recognition Based Traffic System Model ©

However, the techniques used to solve the problem become ineffective for real-time operations because of the computational complexity and longer execution time. Also, some are not even able to operate in bad weather conditions, so they were doomed to failure. Studies had shown that whenever visibility is reduced due to rain, fog or other external factors that the pecentage of vehicles that will be detected will significantly drop. [5]

Even if the problems stated above were known, neural networks systems have been developed with the help of object recognition algorithms. They were designed to predict the upcoming traffic volume from a X-min daylight traffic flow on an expressway. Like so we would be able to overcome the high computational complexity of the algorithm by avoiding recalculations. This lead to high memory needs, that could not be overcomed, so this approach is unsuitable for real-time systems.

To sum it up, unless the memory issue is somehow fixed in the near future they are not a viable option for tracking vehicles.

## 2.2 Vehicle sensor detection based systems

Another approach would be to collect data about the vehicles approaching junctions with the help of GPS sensors. (Figure 3)

One way to do it is to track the position, speed and direction of the given vehicles. Sadly, this method would work only for a road network, it can not solely control the traffic signal timing for just one junction. This is due to the high speeds that vehicles travel at and the time complexity of the main algorithm.

Another way to do it is to monitor the arrival and departure of vehicles at a junction. This can be done with the help of sensors and traffic servers. Like so we could use embedded technology to record the GPS data and send it to the traffic monitoring system through GSM/GPRS. The drawbacks of this method are the fact thats it involves very high implementation cost and, sadly, some vehicles can not be tracked using radio detection systems. This problem can be also aproached with the help of in-road sensors, but it would require even higher costs as you would need often change the sensors because roads often need to be rebuilt due to wear or ongoing costructions.

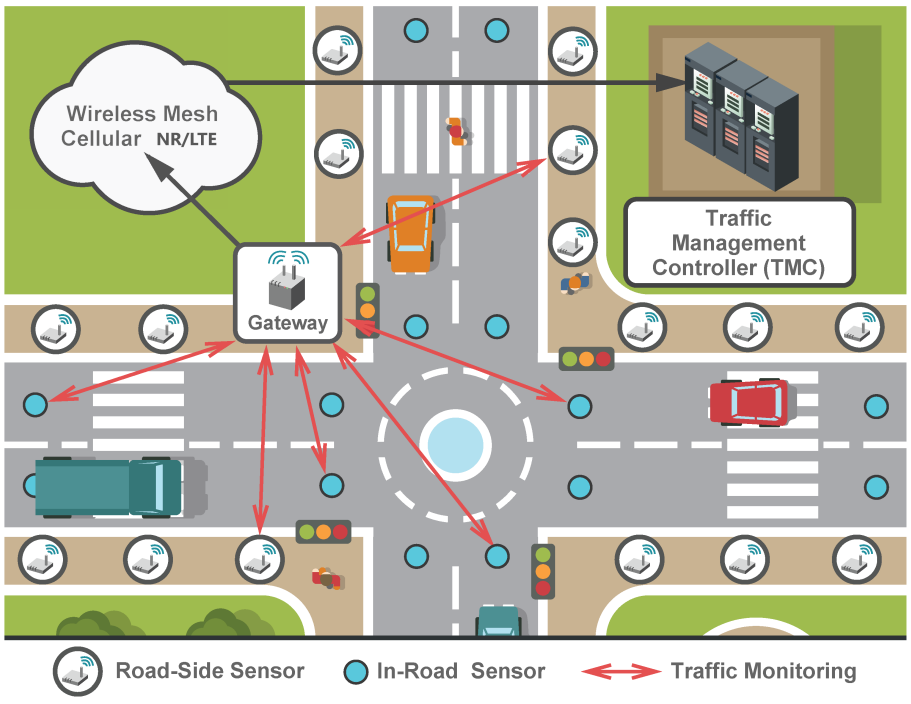



Figure 3: Sensor Based Traffic System Model ©

## 2.3 Traffic Lights Synchronization based systems

Traffic Lights Synchronization (TSC) [7] systems aim to minimize the number of STOP and GO occurrences by adapting the traffic light phases at junctions. It is a technique in which a vehicle traveling along one side of the road at a specific speed can continue to the opposite end of the road indefinitely by obtaining the maximum number of green lights at the intersections. Studies had shown that in comparison with other fixed time and non-synchronized traffic control strategies it reduces overall travel time by up to 39%. [**ALEKO2019**]

Just like most of the others methods, it collects and procces data from real traffic scenarios to determine the green light timer. When vehicles pass through a juction the green light timer for the following juctions decreases by a certain amount (Figure 4). As with the other roads present at the juction, the red light time increase to guarante the continuous motion of already travelling cars. This can be done for mutiple levels of traffic depending on how many junctions do you want to synchronize with one another, but must of the the times it was handled as a 2 level system.

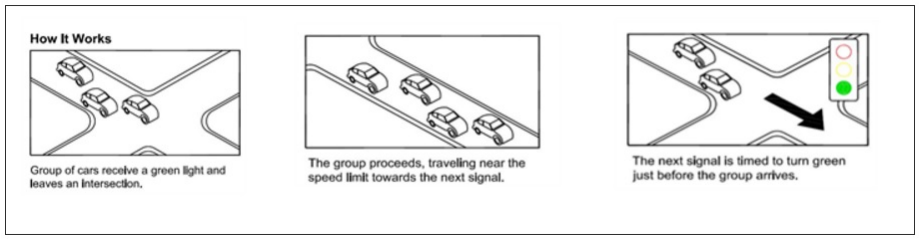



Figure 4: Traffic Signal Synchronization Model ©

Multiple approaches to implement this kind of systems were made with the use of varios AI algorithms and already known data collecting methods but the result are mostly the same. This method would lead to longer red signal time, but will reduce the overall waiting time when travelling long distances. One other advance to keep in mind is that, as result of this phenomenon, the wear of the vehicles will be drastically reduced.

The main disadvantages of this method is the fact that you will eventually have to prioritize one route. This would be problematic if multiple main rouds colide with one another. Also, the algorithm will not always provide an optimal solution as drives following sideroads may have longer waiting times. One more thing to keep in mind is that this method is reliant on drivers speed consistency. Most of the cases, there will be a decent amount of speeding vehicles that will not be able to pass multile junctions without any kind of stops as algorithm does not expect this from drivers side.

## 2.4 Fuzzy Intelligent Traffic Signal Control System

Fuzzy Intelligent Traffic Signal Control (FITS) [6] [3] systems were originally designed to mimic a human policeman in controlling traffic lights at an intersection. They are systems that take different types of traffic input and apply fuzzy rules to manage the traffic. Like so data is converted into fuzzy truth values between 0 and 1. For instance, the green light extension time can be modeled by a number of fuzzy sets including "none", "short", "moderate" and "long". The only thing that you are left to do is to define your membership functions for the given set. For example we can use the following functions to do so:

$$"none" - f(q) = max(min((10 - q)/10, 2), 0) \tag{1}$$

$$"short" - f(q) = max(min(q/10, (20 - q)/10), 0) \tag{2}$$

$$"medium" - f(q) = max(min(q/5, (30 - q)/5), 0) \tag{3}$$

$$"long" - f(q) = max(min((50 - q)/10, q/2), 0) \tag{4}$$

Like so we would alternate base on other fuzzy values or randomly, between choosing "none", "short", "moderate" and "long" fuzzy values, to determine the time that out green light should last using.

Depending on the actual improvements of the traffic flow the algorithm will adapt, altering it's member functions and decreasing/increasing the odds to chose one specific fuzzy value from the set. One big advantage that this would lead to is that FITS mitigates the negative effects of detection malfunction by predicting the traffic states at the whole intersection by simulating real time traffic scenarios. So FITS may be able to run withot any kind of additional hardware, leading to huge cost advantages.

Despite all of the bennefits presented above, as with any AI based algorithm, it takes a lot of time for the algorithm to improve itself and the upgrade. Also the chance to upgrade is very reliant on traffic conditions. Furthermore, even if the algorithm has been running for a long time, it still can choose a non optimal solution that may or may not lead to traffic jams.

## 2.5 Dedicated Short-Range Communications systems

To explain why this kind of systems would work, first we have to understand what are Dedicated Short-Range Communications (DSRC) [8] [7]. DSRC are one-way or two-way wireless communication channels specially designed for use in automobiles that are mostly used by ITS to comunicate with other vehicles or infrastructure technology. They operate on the 5.9 GHz band of the radio frequency spectrum and are effective over short to medium distances.

One technique that was developed with the help of DSRC is the Virtual Traffic Light (VTL) approach. VTL is a biologically-inspired approach to traffic control that relies on Vehicle-to-Vehicle (V2V) communication by using the Signal Phase and Timing (SPaT) message and Basic Safety Message (BSM) from the DSRC OBU. (Figure 5)


Figure 5: Dedicated Short-Range Communications Radio ©

The radio is capable of broadcasting several messages defined by the SAE 2735 [1] protocol with the most important beeing BSMs. This kind of messages contain vehicle's current information (GPS location, speed and where they are heading) that are associated with a temporary ID. As a result, by broadcasting BSM messages we are able to detect the vehicles approaching the intersection in a continuous manner, unlike traditional methods that only detect the presence of the vechile using loop detectors.

You can image the cars beeing "moving routers", and the junctions beeing "stationary routers". Whenever one "gateway" or in our case one traffic route is overfilled with vehicles it blocks the others and starts letting them pass just like RIP protocol. Like so we are not actually introducing new sorts of technology that may or not fail, we just adapt already existing algorithms that proved to be great solutions to fit one specific use case.

Figure 6: DSRC Based System Model ©

With the help of this approach we can drastically reduce the implementation costs as it does not require any kind of additional expensive hardware components. Furthermore, it is robust against weather conditions and easy to mantain. Extensive simulations have shown that this kind of technology can reduce daily commute time in urban areas by more than 30%. Different aspects of VTL technology, including system simulation, carbon emission, algorithm design and deployment policy have been researched in the last few years. [4]

The drawback of this approach is that it requires vehicles to support DSRC technologies, which is not possible at the present time. Furthermore, V2V communications in VTL could come across partial obstruction by a physical object present in the innermost Fresnel zone (non-LoS conditions), which makes rapid decision making extremely challenging. However, even if DSRC technology isn't totally supported by vehicles and we might colide with non-Los conditions, it proved to be effective in reducing the average waiting time at juctions. So, maybe the next step on improving traffic flow, would be to create an infrastructure for this kind of technologie, but only time will tell. In the mean time the best thing we can do is to develop a cost-effective transition scheme between current traffic control systems and VTL systems.

# 3 A new flexible economic approach

We belive that the future of traffic light management will be based on DSRC like signals. No matter if it will still be based on DSRC, 5G or even 6G will take lead, we aim to provide a some kind of software system that would be able to handle any kind message based tehnology. Also, because at the given moment, the infrastructure for DSRC systems hasn't been fully developed and deployed we want to provide a economic approach to the given problem that would also bennefit on the already working systems, but also can simulate the already working traffic light management protocol.

Even if DSRC will be fully deployed on a globas bases, there will still be alot of poor countries that will have a hard time upgrading their infrastructure. Because of that, we wanted to be able to have a working system even without investing any kind of money. Thing about a PC for example. Even one of the worst ones can run an OS and if you want to updgrade them, to make it run faster or handle a newer software, you would just add or replace a component with a better performing one. Not only you will be able to add "performance boosts" as needed to your given infrastructure but think about the scenarios when something goes bad? What if one of your component goes down and for example the traffic light will be stuck on red until someone comes and fixes it. That would lead to a HUGE traffic jam.

We also want to be able to take advantage of already working DSRC compatible vechiles without adding any kind of new hardware and be able to disable the additional protocols and move to a fully DRSC based system if needed.

Sounds great, right? This is how we designed it. The system will have **4 major components**: **2 types of clients**: one stationary one and one moving one, "attached" to our vechiles; **2 types of servers**: one that would manage the position of the clients and redirect them to the corresponding servers and one that would handle all of the messages and change the actual traffic lights. For the simplicity on things we will name them as follows:

- **Traffic Observer (TO)** - the stationary client.
- **Vehicle Tracker (VT)** - the moving client
- **Junction Main Server (JMS)** - the server that handles the traffic lights
- **Proxy** - the server that will guide the clients to the corresponding servers

The system would work as follows: We would constanly have a TCP-IP connection between our clients and the corresponding JMS. Once they are connected clients would send various data about the traffic that would be further proccesed by the JMS and traffic lights would be changed accordingly. The TO would be responsible for sending data about the traffic on it's road and the VT would be responsible for sending it's own data to the server (Figure 7). We also handled the cases when there is an emergency ongoing like any kind of special vehicles crossing the junction but we will talk later about this because it requires some more attention as it can be easily exploited if not handled right. Just imagine any kind of cars beeing treated like an ambulance on duty, you would just be able to pass freely any kind of junction without ever waiting at the red light.

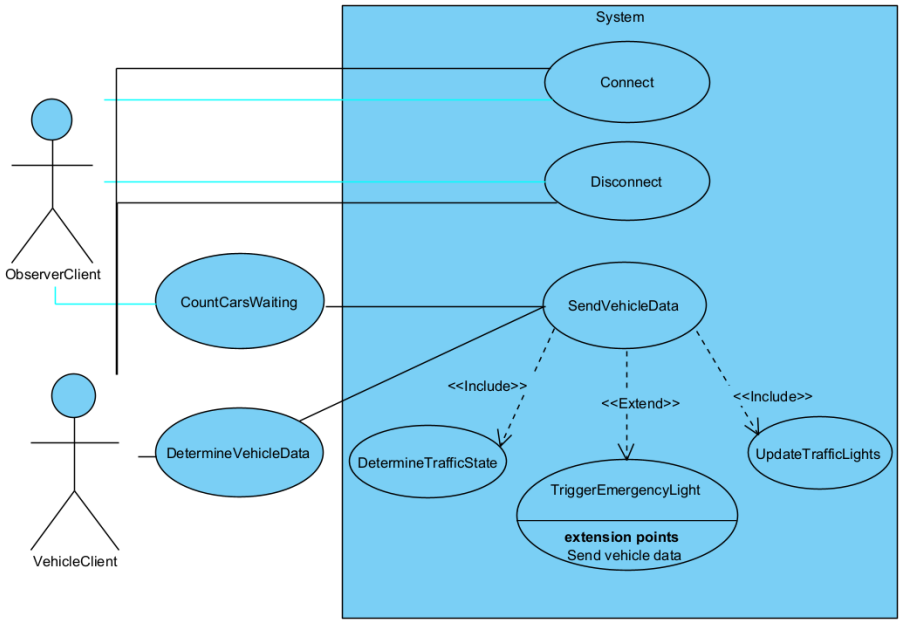



Figure 7: All UC Diagram

## 3.1 Proxy

The first problem that needs to be addressed is the fact that VTs will constanly have to switch between JMSs, but how do we know what server to connect to? We do not want to connect to each and every server in the covered area and start broadcasting mostly useless message. The answear would be that each client will know the main server off the system, that would act just like a proxy and redirect them to the corresponding JMS. For that we would need to store the JMS coordinates in one or multile database. The VT would need to just send its own coordinates and the direction they are heading, as a result the proxy would provide de IP address of the corresponding JMS (Figure 8).

As we want this to be scaled on a global level, the system supports horizontal scalling, as we can just stack proxys one on top of the other and instead of having stored only to the JMS coordinates we will also store all the proxys that are in the subarea of the owner. To not overload any kind of servers we also implemented a load balancing mechanism that prevents this kind of issue.

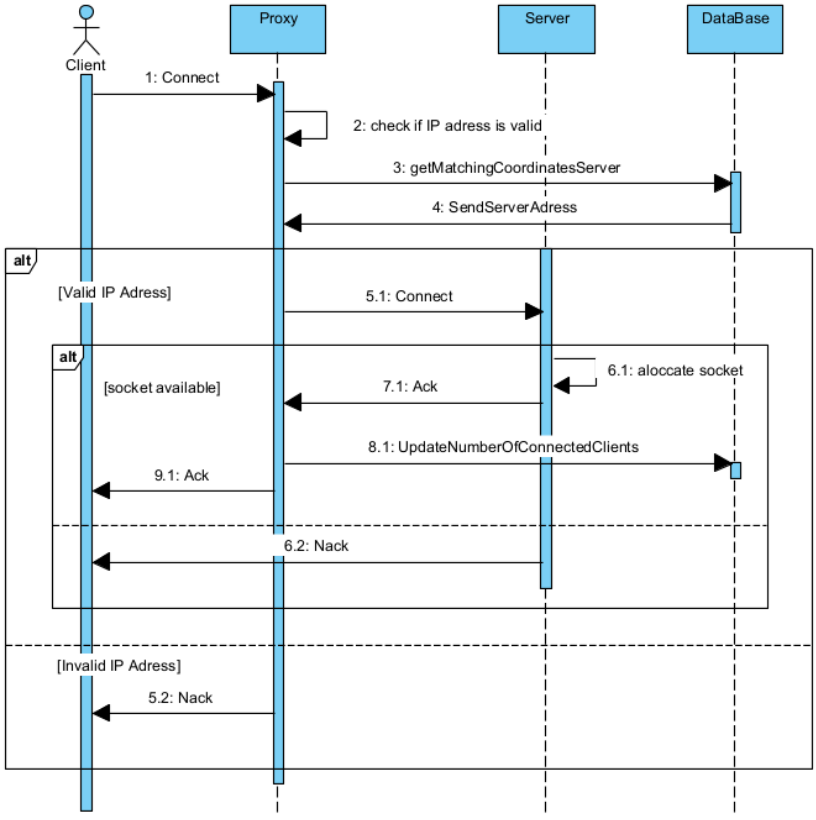



Figure 8: UC: Connect

One thing this can lead to is a better surveillance of the vechiles that aren't suppose to be on the road in the first place. The IP address of a given vechile can be "banned" and if it tries to connect to the JMS the message will be ignored and not only the JMS will not count the waiting vechile, sometimes leading to longer waiting times for the individual that is driving the vechile and broke the law, but also like this we would have the proof that he broke the law so we would be able to further punish him. We can lower the number of cars driving on the public roads that do not respect safety norms and reduce the number of accidents. For this to work we will need to also disallow owners to change their hardware without any kind of approval as a new radio would lead to a different IP address and this would be problematic.

Once passing the area of coverage of that given junction we would also need to disconnect from that given JMS (Figure 9).

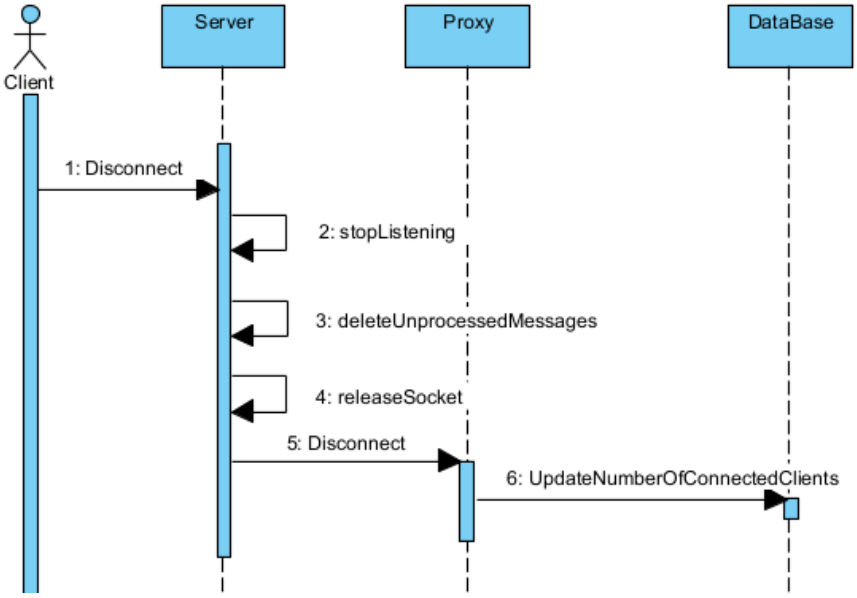



Figure 9: UC: Disconnect

## 3.2 Traffic Observer

The second thing we need to do is provide a way to detect the waiting vechiles withot the help of DSRC messages. This can be done in mutiple ways, but for the simplicity of things and to keep costs as low as possible we went with an camera based system. The way it works is as follows: a camera is mounted on the road we want to keep track of, that is connected to a microcontroller that can run our software.

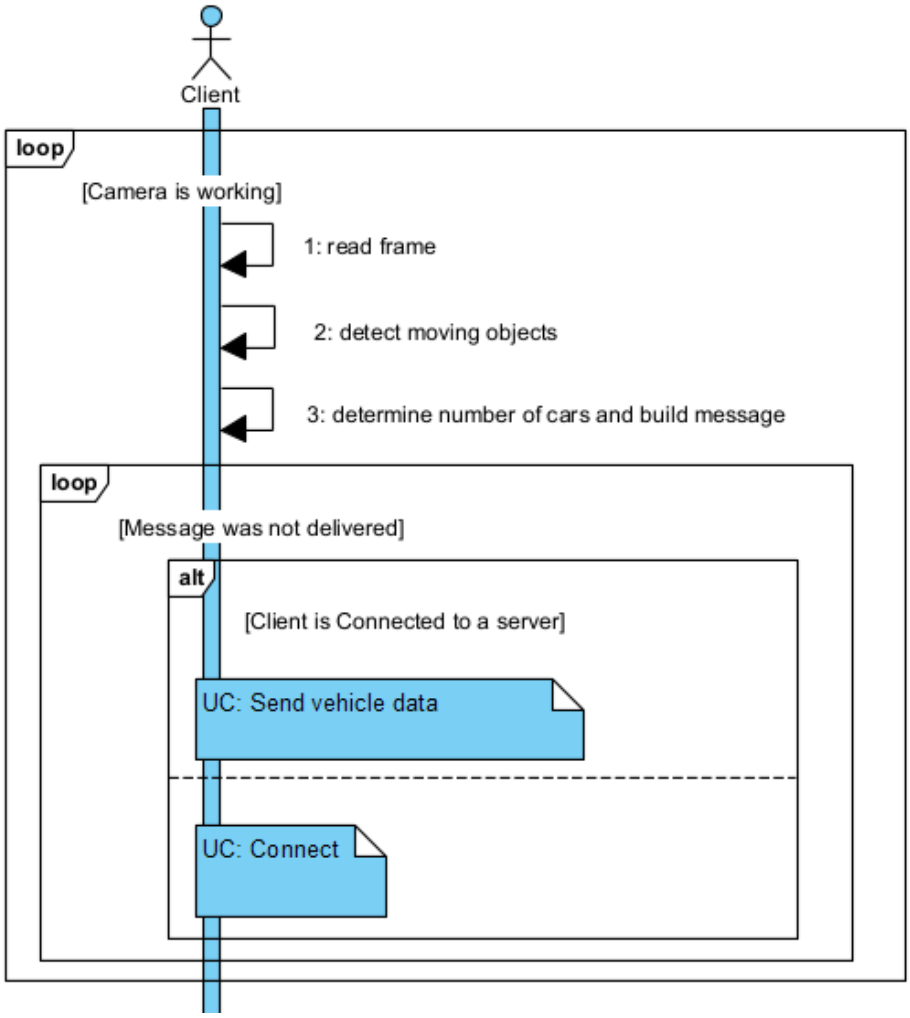
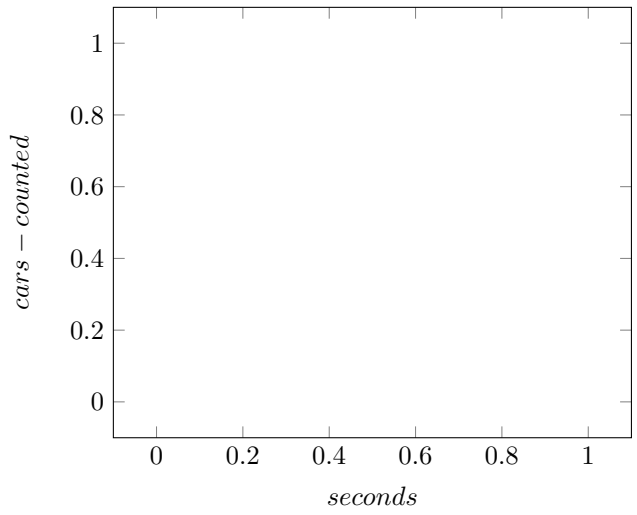



Figure 10: UC: Find cars from provided images

With the help of the provided images we first and foremost detect the moving objects on the given frames and second of all determine if that moving object is or is not a vechile (Figure 10). Like this we can speed up the image detection process as we will have less pixels to keep track off. The car detection speedup of the movement detection can be seen in Figure 3.2. The blue line represents the amount of cars detected with it running and the red one without it.




## 3.3 Vehicle Tracker

We want to be able to handle DSRC signals. But how can we achieve that? We do not want to add additional hardware requirments as this will just make the deployment of our app much much slower and more costly. Well, we make use of already integrated hardware components, to be more specific we make use of the radios and GPS trackers. With the use of the GPS we get the coordinates of the vehicle as well as it's heading. The radio allows us to establish TCP-IP connection and send messages. So the only things that is left to do is to download our app and keep it running while travelling.

To identify the vehicles we allocate them an IP adress when installing the app. But how can we make sure the user isn't amble to freely change their adress? We would want an "immutable storage space" to achieve that. To do so we require the manufacturer to encrypt their storage data so that only they will be able to modify it if needed. We just talked about the fortunate case, if for example we just unplug the radio module from the car how would we be able to send data in the system? Well we couldn't, but thats not necessarily what we want to achive. We want to prevent the sending of invalid data to the system and we achived that for now, when it comes to actual vehicles. Other faulty scenarios like driving a car without a radio, basically in "incognito mode", should be punished by the law and is something that we just shouldn't handle.

### 3.4 Junction Main Server

To handle all of those messages and change the actual flow of traffic we need to have a main command unit that acts as a server. Now that we know about all the components of the system we can start vizualing it(Figure 11).

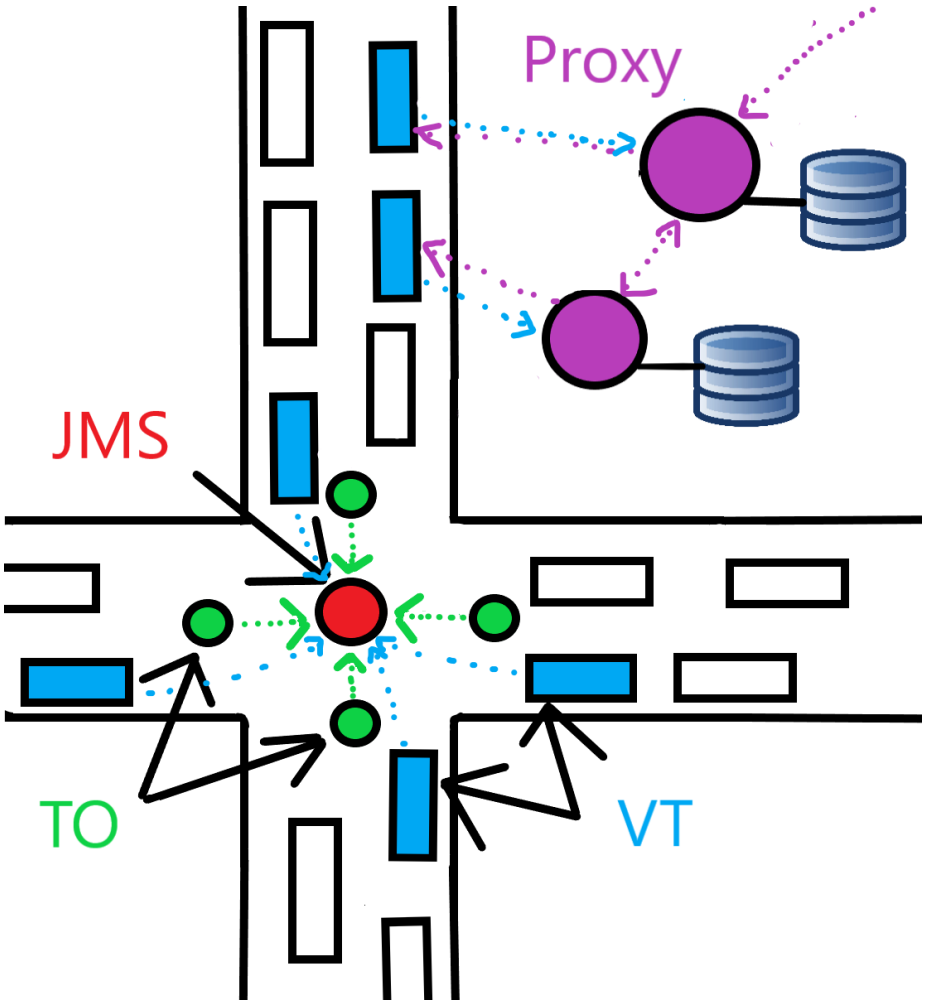



Figure 11: System sketch

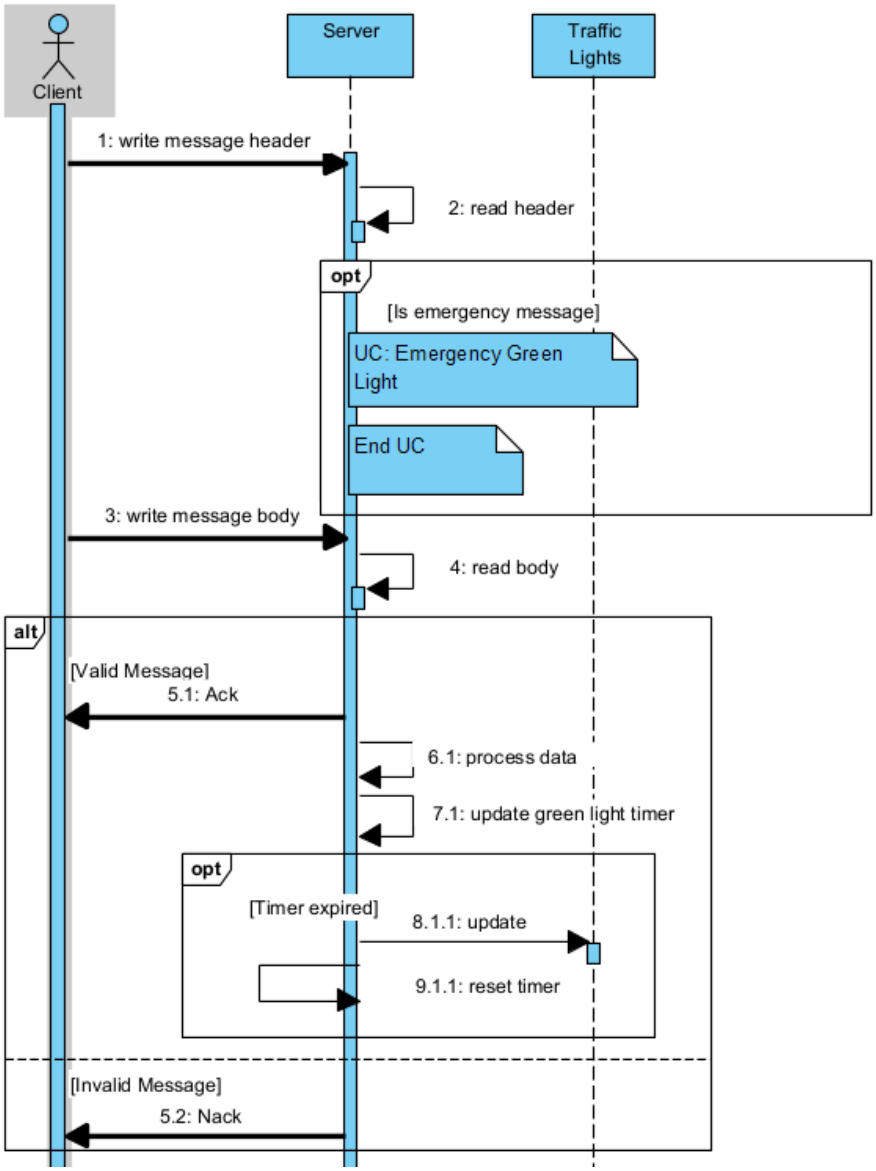



Figure 12: UC: Send vehicle data

The mechanism works mostly like any other TCP-IP mechanisms. We read the header of the message to determine the type of message recieved and the size of the actual message then based on the info we procces the data and update the traffic state. You may ask what happens if a car is detected by the TO as well as it's data is sent to the JMS trough a VT. Well, because of this scenario we chose to aproximate the traffic based on the number of cars determined by the

TO and the number of VT that sent messages to the JMS. Like this we make sure that, if one of the 2 components fail the system will still work. Also because noone is suppose to wait to long for the green light we can set a maximum timer for the red light. Imagine beeing alone on one road waiting for the green light and having thousands of cars crossing the intersection from another road, you would have to wait ages for you to be able to cross. This prevents that and acts just like an "aging mechanism" (Figure 12).

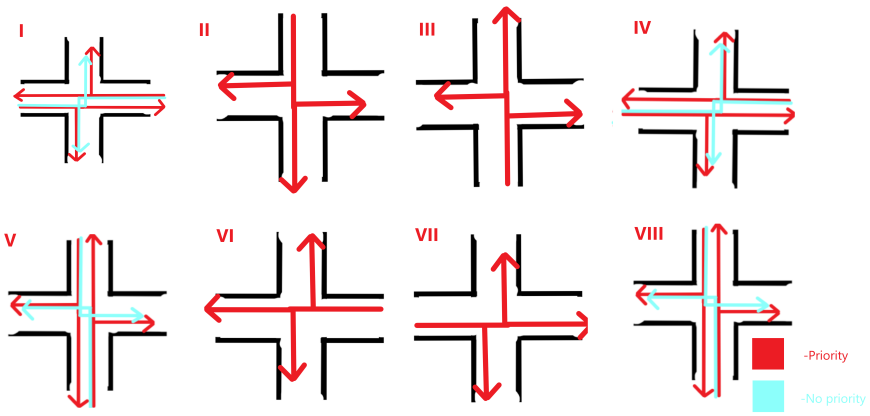



Figure 13: Junction Phazes

As far as the main algorithm for the traffic management goes we have 8 phazes of the traffic (Figure 13). The most basic scenarios is that the traffic will follow that 8 phaezes going trough phaezes 1 to phaze 8 in order. That's what is already implement for the current way the traffic works, we want to also be able to jump from one phaze to any other at any given time, to be able to shorcut the system. For that we are basing on several things.

First and foremost we need to take into account the number of cars waiting. We do that by combing the results from the VT and TO and take their maximum. Because in the TO scope we use Image Recognition and it may sometimes fail we will be using both entering and leaving counts and use ML(Machine Learning) to determine the actual number of cars that crossed the road. The algorithm will want to make the entering vehicles at least as high as the leaving ones, so we will be using the following equation: $vehicles = max(x, f(y))$ ;x being the number of the vechiles from the VT, y the number of vehicles from the TO, and f beeing the function constantly beeing changed by our ML.

Second of all we will need to fully understand our traffic pahzes, why the are required and how to determine the phaze we want to switch to. For that we first described the phazes based on the acting vehicles.

- PHAZE I: E + W waiting vehicles
- PHAZE II: N waiting vehicles
- PHAZE III: S waiting vehicles
- PHAZE IV: E + W waiting vehicles (same with PHAZE I)
- PHAZE V: N + S waiting vehicles
- PHAZE VI: E waiting vehicles=
- PHAZE VII: W waiting vehicles
- PHAZE VIII: N + S waiting vehicles (sane with PHAZE V)

Now we want to know when to switch from one phaze to another. For that we will have timers for each direction: E, W, N and S. At first all the timers will be set to 300 sec (5 min) waiting time, but the waiting time will be changed by the use of another ML updated function. This will be further described as right now, the reasons behind it will not be fully understood.

Now, to start the whole mechanism we would want to just start the traffic as usual, having the normal order of phazez running and for us to switch to and abnormal phaze we'll just need one or more timers to expire. As any other timer, it will decrease normally, but also when a car starts waiting at the crossroad timer will be decreased by f(x), x beeing the time left for the timer to expire. We want to have as little cars waiting at the junction and as many cars crossing. For that we use yet another ML algorithm to update the function that determines how much time will be taken from the timer. The criteria for it will be that we want to stick to a not that low but not that high fix number of cars. For now we will be wanting to have at least 10 cars crossing at once. The check for the jump to an abnormal phaze will be done ONLY when GREEN LIGHT ENDS.

The green light can be intrerupted only be emergency states(will talk about this later) and is based on yet another ML algorithm. To maintain traffic safety conditions, we have a fix time set for the drivers to notice the green light switch, to be prices 5 sec. You may think that this is very low, but it can actually be extended to 15 sec as the maximum number of phazes we can go trough without any cars waiting is 3 and we want to keep wasted time as low as possible, that's the main reason of creating the system so any bigger time perios would be a waste. The green light time will take the following form: $time = 5 + f(x)$ where x is the number of cars waiting and f is the function updated by the ML algorithm. The criteria of the algorithm is to make as close to x cars pass the junction during the green light and is really usefull when it comes to the speed the vechiles are moving on different roads.

Last but not least we need to define the way our algorithm jumps from one phaze to another, for that we will use the following example (Figure 14):

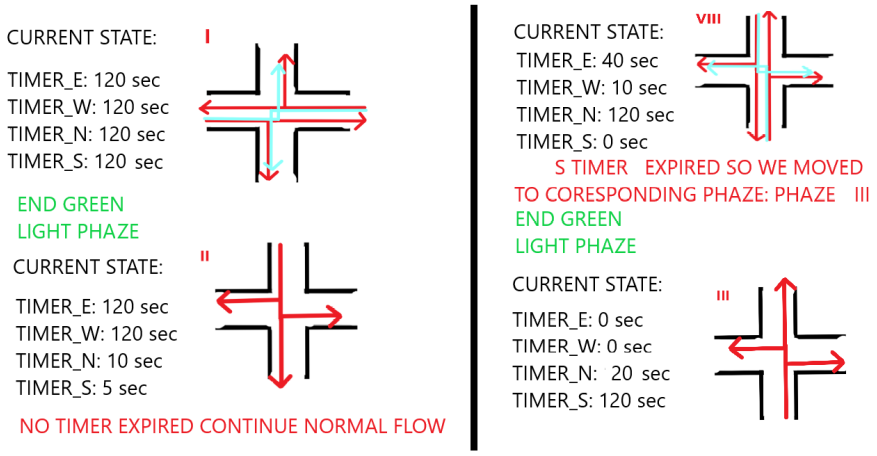



Figure 14: Phaze switching example

At the end of every green light phaze we check the timers. If any of them expired we jump to it's corresponding phaze. If, for example the following state coresponds to the phaze we want to jump to, we do not short circuit the normal flow of things. Every time we are trough green light phazes it's timer is frozen and then reset to it's maximum value.

We need to talk also about the situations we want to avoid as much as possible. This are caused as a result of letting the cars from only one road cross (phazes II, III, VI and VII)(Figure 15). As mentioned before we use FIFO logic and take the phaze that corresponds to the first timer(s) that expired as a fix.

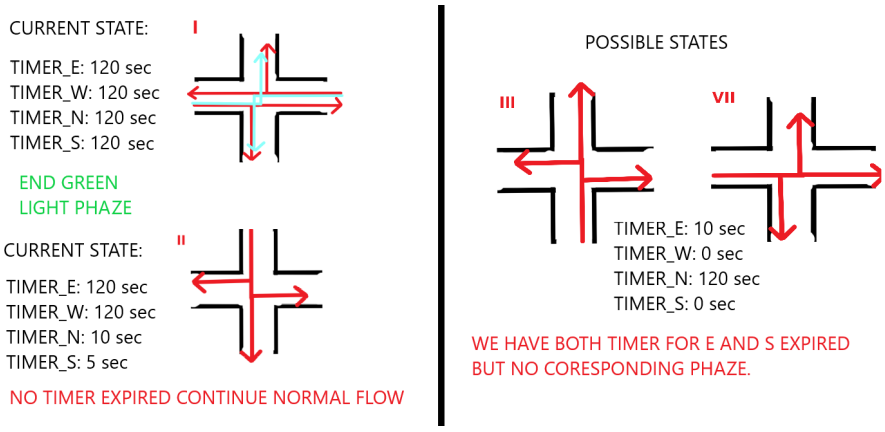



Figure 15: Faulty phaze switching

### 3.4.1 Emergency states

Because every lifes matter we wanted to make sure that the special vechiles like ambulances, police cars and fire trucks crossing the junctions will be treated differently. For that, whenever one of those vehicle is detected our server turns on the green light corresponding to the lane they are following and keeps it on until it crosses the junction. If we are already in an so called emergency green light state, the procces will just be queued and be restarted when the already incoming emergency vehicle passes. (Figure 16).

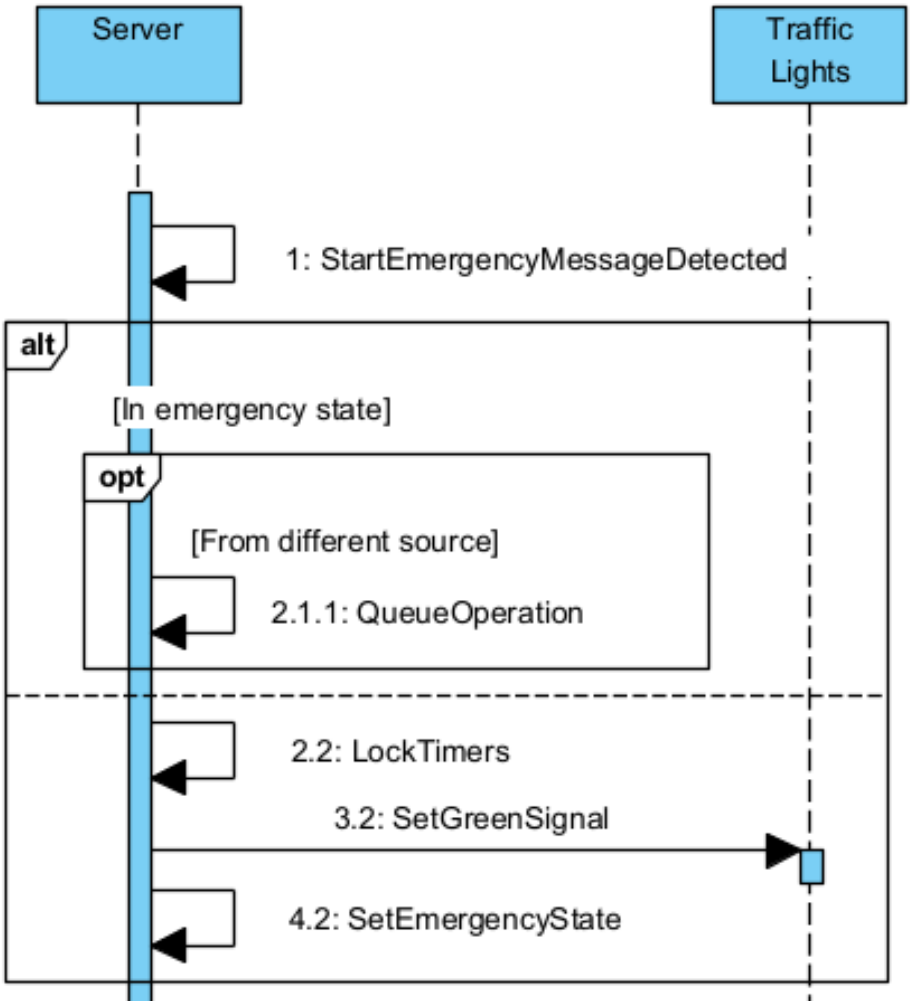



Figure 16: UC: Emergency Green Light Start

Of course when finishing with an emergency state we should try to first of determine if another one should be started right after or we should come back to the normal flow of traffic. Also we should restart the maxium wait time timers, as they were frozen during the whole emergency state. (Figure 17).

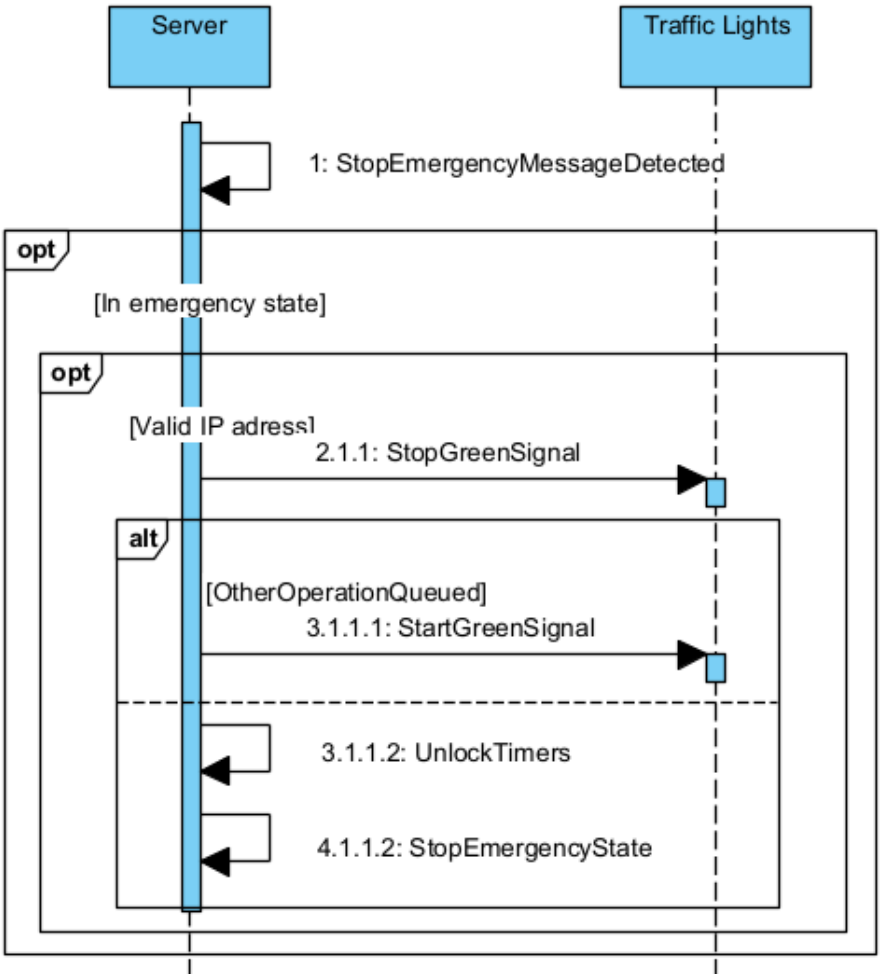



Figure 17: UC: Emergency Green Light Stop

# 4 Implementation details

The whole system was developed using C++17, Boost, GLFW, MSVC WinAPI, OpenCV, Python, Tensorflow and MySQL. The system itself is treated as a big project and splitted into multiple submodules: 3 static librarys(Common, CarDetector and IPC) and 6 executables (Proxy, JMS, VT, TO, TestingExe and ModelTrainig). This section aims to explain in detail what each and every one of this submodules were meant to do, how they were implemented some examples of using them and the main features that they provide. To compile and test the project itself I chose to use Microsoft Visual Studio 2022 MSVC as it provided a really quick and easy way of debugging my applications, but the code itself, if there is to be linked with CMake all can be run on any platform, appart from the testing module, as it is using Windows.h library to spawn and handle processes. Also, apart from the project itself, to train the car model I choose to use Python and Tensorflow as they were the most reliant AI tools at the given moment.

```
// using CreateProcess
bool createProcessFromSameDirectory(const command_line& cmd);
void closeAllProcesses()
{
    for (const auto& pi : g_runningProcesses)
    {
        HANDLE processHandle = NULL;
        processHandle = OpenProcess(PROCESS_TERMINATE | PROCESS_ALL_ACCESS,
            FALSE, pi.dwProcessId);
        if (processHandle == NULL) {
            std::cerr << "Error opening process: "
                << GetLastErrorAsString() << std::endl;
            continue;
        }
        // Wait until child process exits.
        WaitForSingleObject(pi.hProcess, 5000);

        if (!TerminateProcess(pi.hProcess, 0)) {
            std::cerr << "Error terminating process: "
                << GetLastErrorAsString() << std::endl;
        }


        // Close process and thread handles.
        CloseHandle(pi.hProcess);
        CloseHandle(pi.hThread);
    }
}
```

## 4.1 Common

The main ideea of the common submodule is to act as a helper library that may be reused in multiple other projetcs and it was developed in the means of helping with multiple problems:

- multi threaded concurrency
- splitting tasks to threads, as some executables run on more then 6 threads
- handling command line arguments
- handling signals
- logging in a multi thread environment
- the need of timers that trigger actions when they expire
- reading from a config file
- managing GPS geocoordinates
- model for DB tables

The main problems that I aimed to fix with this submodule are the multi threaded issues. To do so I implemented thread safe structures, a thread pool and altered an already existant multi thread synchronized logger implementation . Like this I avoided concurrency problems when handling data and fixed the problem regarding the big number of threads that may cause performance issues for processors with less then 6 cores as well as managing to have more readable logs for debugging purposes.

```
template<typename T>
class ThreadSafeQueue
{
protected:
    std::mutex mutexQueue_;
    std::queue<T> queue_;
    LOGGER("TSQ");
public:
    ThreadSafeQueue() = default;
    ThreadSafeQueue(const ThreadSafeQueue&) = delete;
    virtual ~ThreadSafeQueue() { clear(); }

    std::optional<T> pop()
    {
        if (queue_.empty())
        {
            LOG_WARN << "Tried to extract data from queue when it was empty";
```

```
            return {};
        }
        std::scoped_lock lock(mutexQueue_);
        auto t = std::move(queue_.front());
        queue_.pop();
        return t;
    }

    void push(const T& operation)
    {
        std::scoped_lock lock(mutexQueue_);
        queue_.push(std::move(operation));
    }

    const bool empty()
    {
        std::scoped_lock lock(mutexQueue_);
        return queue_.empty();
    }

    const size_t size()
    {
        std::scoped_lock lock(mutexQueue_);
        return queue_.size();
    }

    void clear()
    {
        std::scoped_lock lock(mutexQueue_);
        while (!queue_.empty())
        {
            queue_.pop();
        }
    }
};
```

## 4.2 IPC

To be able to send messages over the network I needed to create an IPC environment. For that I used Boost Asio to more easily handle connect/disconnect actions. This module has 4 main template classes that are further used by all the executables: Client, Connection, Server and Message.

The client always owns one connection, the one between him and the server and does idle work until required to send/recieve messages. I chose to do the connection operation and the message recieving syncronosly, while proving a timeout to the functions.

```
template<typename T>
class Client
{
private:
    common::utile::ThreadSafePriorityQueue<OwnedMessage<T>> incomingMessages_;

protected:
    boost::asio::io_context context_;
    std::thread threadContext_;
    std::mutex mutexUpdate_;
    std::condition_variable condVarUpdate_;
    std::unique_ptr<Connection<T>> connection_;
    std::atomic<bool> shuttingDown_ = false;
    boost::asio::io_context::work idleWork_;
    LOGGER("CLIENT");

public:


    Client() : idleWork_(context_)
    {
        threadContext_ = std::thread([this]() { context_.run(); });
    }


    virtual ~Client() noexcept
    {
        shuttingDown_ = true;
        LOG_INF << "Server shutting down";
        disconnect();
        stop();
    }
```

```
bool connect(const utile::IP_ADRESS& host, const ipc::utile::PORT port)
{
    if (!utile::IsIPV4(host))
    {
        LOG_ERR << "Invalid IPV4 ip adress: " << host;
        return false;
    }

    try
    {
        boost::asio::ip::tcp::resolver resolver(context_);
        boost::asio::ip::tcp::resolver::results_type endpoints =
            resolver.resolve(host, std::to_string(port));

        connection_ = std::make_unique<Connection<T>>(
            Owner::Client,
            context_,
            boost::asio::ip::tcp::socket(context_),
            incomingMessages_,
            condVarUpdate_);

        return connection_->connectToServer(endpoints);

    }
    catch(const std::exception& e)
    {
        LOG_ERR << "Client exception: " << e.what() << '\n';
        return false;
    }
}




void disconnect()
{
    if (isConnected())
    {
        connection_->disconnect();
    }
    connection_.reset();
}
```

```
void stop()
{
    context_.stop();

    condVarUpdate_.notify_one();
    if (threadContext_.joinable())
        threadContext_.join();
}



bool isConnected()
{
    return connection_ && connection_->isConnected();
}



bool answearRecieved()
{
    return !incomingMessages_.empty();
}



std::optional<std::pair<OwnedMessage<T>, bool>> getLastUnreadAnswear()
{
    return incomingMessages_.pop();
}



common::utile::ThreadSafePriorityQueue<OwnedMessage<T>>& getIncomingMessages
{
    return incomingMessages_;
}



void send(const Message<T>& msg)
{
    if (isConnected())
        connection_->send(msg);
}
```

```
    bool waitForAnswear(uint32_t timeout = 0)
    {
        if (timeout == 0)
        {
            std::unique_lock<std::mutex> ulock(mutexUpdate_);
            condVarUpdate_.wait(ulock, [&] { !incomingMessages_.empty() || shutti

            return !shuttingDown_;
        }

        std::unique_lock<std::mutex> ulock(mutexUpdate_);

        if (condVarUpdate_.wait_for(ulock, std::chrono::milliseconds(timeout * 1
        {
            return !shuttingDown_;
        }
        else
        {
            LOG_ERR << "Answear waiting timedout";
            return false;
        }
    }

    uint32_t getId()
    {
        return connection_->getId();
    }
};
```

A connection is the linking between a server and a client, handling the reading and writing of messages, as well as notifing the owners of a message arrival. It runs on 2 threads, one for writing and one for reading and every operation is done syncronosly to avoid unwanted behaviour like reading from another message.

```
template <typename T>
class Connection : public std::enable_shared_from_this<Connection<T>>
{

protected:
    const Owner owner_;
    std::thread threadRead_;
    std::thread threadWrite_;
    std::mutex mutexRead_;
    std::mutex mutexWrite_;
    std::condition_variable condVarRead_;
    std::condition_variable condVarWrite_;
    std::condition_variable& condVarUpdate_;
    boost::asio::io_context& context_;
    boost::asio::ip::tcp::socket socket_;
    common::utile::ThreadSafePriorityQueue<OwnedMessage<T>>& incomingMessages_;
    common::utile::ThreadSafePriorityQueue<Message<T>> outgoingMessages_;
    Message<T> incomingTemporaryMessage_;
    std::atomic_bool isReading_;
    std::atomic_bool isWriting_;
    std::atomic_bool shuttingDown_ = false;
    uint32_t id_;
    std::string ipAdress_;
    LOGGER("CONNECTION-UNDEFINED");


private:


    struct compareConnections {
        bool operator() (const Connection<T>& a, const Connection<T>& b) const {
            return a.getId() < b.getId();
        }
    };
```

```
    bool readData(std::vector<uint8_t>& vBuffer, size_t toRead)
    {
        size_t left = toRead;
        while (left && !shuttingDown_)
        {
            if (shuttingDown_)
                return false;

            boost::system::error_code errcode;
            size_t read =
                socket_.read_some(
                    boost::asio::buffer(
                        vBuffer.data() + (toRead - left), left), errcode);

            if (errcode)
            {

                LOG_ERR << "Error while reading data err: "
                    << errcode.value() << errcode.message();
                disconnect();
                return false;
            }
            left -= read;
        }
        return true;
    }


public:
    Connection(Owner owner,
        boost::asio::io_context& context,
        boost::asio::ip::tcp::socket socket,
        common::utile::ThreadSafePriorityQueue<OwnedMessage<T>>& incomingMessage
        std::condition_variable& condVarUpdate) :
        owner_{ owner },
        context_{ context },
        socket_{ std::move(socket) },
        incomingMessages_{ incomingMessages },
        condVarUpdate_{condVarUpdate},
        isWriting_{false},
        isReading_{false},
        id_{0}
    {
        threadWrite_ = std::thread([this]() { writeMessages(); });
        threadRead_ = std::thread([this]() { readMessages(); });
    }
```

```
virtual ~Connection() noexcept
{
    shuttingDown_ = true;

    condVarRead_.notify_one();
    if (threadRead_.joinable())
        threadRead_.join();

    condVarWrite_.notify_one();
    if (threadWrite_.joinable())
        threadWrite_.join();

    disconnect();
}



Owner getOwner() const
{
    return owner_;
}



uint32_t getId() const
{
    return id_;
}



std::string getIpAdress() const
{
    return ipAdress_;
}
```

```
bool connectToServer(
    const boost::asio::ip::tcp::resolver::results_type& endpoints)
{
    if (owner_ == Owner::Client)
    {
        LOG_SET_NAME("CONNECTION-SERVER");
        std::function<void(
            std::error_code errcode,
            boost::asio::ip::tcp::endpoint endpoint)> connectCallback;
        if (isReading_)
        {
            connectCallback = [this](
                std::error_code errcode,
                boost::asio::ip::tcp::endpoint endpoint)
            {
                if (errcode)
                {
                    LOG_ERR << "FAILED TO CONNECT TO SERVER: "
                        << errcode.message();
                    socket_.close();
                }
            };
        }
        else
        {
            connectCallback = [this](
                std::error_code errcode,
                boost::asio::ip::tcp::endpoint endpoint)
            {
                if (!errcode)
                {
                    LOG_DBG <<"Started reading messages:";
                    condVarRead_.notify_one();
                }
                else
                {
                    LOG_ERR << "FAILED TO CONNECT TO SERVER: "
                        << errcode.message();
                    socket_.close();
                }
            };
        }

        std::error_code ec;
        boost::asio::ip::tcp::endpoint endpoint;
        try
```

```
        {
            endpoint = boost::asio::connect(socket_, endpoints);
        }
        catch (boost::system::system_error const& err)
        {
            ec = err.code();
        }

        connectCallback(ec, endpoint);
        if (socket_.is_open())
        {
            isReading_ = true;
            ipAdress_ = socket_.remote_endpoint().address().to_string();
            return true;
        }

        return false;
    }
}



bool isConnected() const
{
    return socket_.is_open();
}



void disconnect()
{
    if (isConnected())
    {
        boost::asio::post(context_, [this]() { socket_.close(); });
    }
}



std::shared_ptr<Connection<T>> get_shared()
{
    if (shuttingDown_)
        return nullptr;

    return this->shared_from_this();
}
```

```
bool connectToClient(uint32_t id)
{
    if (owner_ == Owner::Server)
    {
        if (socket_.is_open())
        {
            id_ = id;
            LOG_SET_NAME("CONNECTION-" + std::to_string(id_));
            std::function<void()> connectCallback;
            if (!isReading_)
            {
                connectCallback = [this]()
                {
                    LOG_DBG <<"Started reading messages";
                    condVarRead_.notify_one();
                };
            }
            else
            {
                connectCallback =[this]()
                {
                };
            }
            isReading_ = true;
            connectCallback();
            return true;
        }
        return false;
    }
    return false;
}

void readMessages()
{
    std::unique_lock<std::mutex> ulock(mutexRead_);
    condVarRead_.wait(ulock,
        [this]() { return isReading_ || shuttingDown_; });
    ulock.unlock();

    while (!shuttingDown_)
    {
        std::scoped_lock lock(mutexRead_);
        if (!readHeader()) { break; }
```

```
            if (!readBody()) { break; }
            addToIncomingMessageQueue();
            condVarUpdate_.notify_one();
        }
        isReading_ = false;
    }

    bool readHeader()
    {
        std::vector<uint8_t> vBuffer(sizeof(MessageHeader<T>));

        if (!readData(vBuffer, sizeof(MessageHeader<T>))) { return false; }

        std::memcpy(
            &incomingTemporaryMessage_.header,
            vBuffer.data(),
            sizeof(MessageHeader<T>));
        LOG_DBG << "Finished reading header for message: "
            << incomingTemporaryMessage_;
        return true;
    }

    bool readBody()
    {
        std::vector<uint8_t> vBuffer(incomingTemporaryMessage_.header.size * siz

        if (!readData(vBuffer, sizeof(uint8_t) * incomingTemporaryMessage_.heade

        incomingTemporaryMessage_ << vBuffer;
        LOG_DBG << "Finished reading message: " << incomingTemporaryMessage_;
        return true;
    }

    void addToIncomingMessageQueue()
    {
        if (owner_ == Owner::Server)
        {
            const auto& pair = std::make_pair(
                OwnedMessage<T>{get_shared(), incomingTemporaryMessage_},
                incomingTemporaryMessage_.header.hasPriority);
            incomingMessages_.push(pair);
        }
        else
        {
            incomingMessages_.push(
                std::make_pair(
```

```
                    OwnedMessage<T>{nullptr, incomingTemporaryMessage_},
                    incomingTemporaryMessage_.header.hasPriority
                ));
        }
        incomingTemporaryMessage_.clear();
        LOG_DBG << "Added message to incoming queue";
    }



    void send(const Message<T>& msg)
    {
        std::function<void()> postCallback;
        std::pair<Message<T>, bool> pair = std::make_pair(msg, msg.header.hasPri
        outgoingMessages_.push(pair);
        LOG_DBG <<"Adding message to outgoing queue: " << msg;
        if (isWriting_)
        {
            postCallback = [this, msg]()
            {
            };
        }
        else
        {
            postCallback = [this, msg]()
            {
                LOG_DBG <<"Started writing messages";
                condVarWrite_.notify_one();
            };
        }
        isWriting_ = true;

        if (isConnected())
        {
            boost::asio::post(context_, postCallback);
        }
        else
        {
            LOG_WARN << "Failed to post message, client is disconnected";
        }
    }
```

```
    void writeMessages()
    {
        while (!shuttingDown_)
        {
            std::unique_lock<std::mutex> ulock(mutexWrite_);
            condVarWrite_.wait(ulock, [this]() { return isWriting_ || shuttingDo
            ulock.unlock();

            while (!outgoingMessages_.empty())
            {
                std::scoped_lock lock(mutexWrite_);

                if (shuttingDown_)
                    break;

                auto outgoingMsg = outgoingMessages_.pop();
                if (!outgoingMsg)
                {
                    LOG_ERR << "Failed to get image from queue";
                    return;
                }
                const auto& outgoingMessage = outgoingMsg.value().first;

                LOG_DBG << "Started writing message: " << outgoingMessage;
                if (!writeHeader(outgoingMessage)) { outgoingMessages_.clear();

                if (outgoingMessage.header.size > 0)
                {
                    if (!writeBody(outgoingMessage)) { outgoingMessages_.clear()
                }
                else
                {
                    LOG_DBG << "Finished writing message";
                }
            }
            isWriting_ = false;
        }
    }
\pagebreak
```

```
    bool writeHeader(const Message<T>& outgoingMessage)
    {
        boost::system::error_code errcode;
        boost::asio::write(socket_, boost::asio::buffer(&outgoingMessage.header,

        if (errcode)
        {
            LOG_ERR << "Failed to write message header: " << errcode.message();
            disconnect();
            return false;
        }
        LOG_DBG << "Finished writing header";
        return true;
    }



    bool writeBody(const Message<T>& outgoingMessage)
    {
        boost::system::error_code errcode;
        boost::asio::write(socket_, boost::asio::buffer(outgoingMessage.body.dat

        if (errcode)
        {
            LOG_ERR << "Failed to write message body: " << errcode.message();
            disconnect();
            return false;
        }
        LOG_DBG << "Finished writing message";
        return true;
    }
};
```

The server runs on 2 separate threads: one for boost asio work and one for updating when a new message is recieved. Same with the other classes, everything is done syncronosly except for accepting new connections.

```
template<typename T>
class Server
{
protected:
    common::utile::ThreadSafePriorityQueue<OwnedMessage<T>> incomingMessagesQueu
    boost::asio::io_context context_;
    std::thread threadContext_;
    std::thread threadUpdate_;
    std::condition_variable condVarUpdate_;
    std::mutex mutexUpdate_;
    std::mutex mutexMessage_;
    boost::asio::ip::tcp::acceptor connectionAccepter_;
    common::utile::ThreadSafeQueue<uint32_t> availableIds_;
    std::map<uint32_t, std::shared_ptr<Connection<T>>> connections_;
    std::atomic<bool> shuttingDown_ = false;
    LOGGER("SERVER");

public:

    Server(const utile::IP_ADRESS& host, ipc::utile::PORT port):
        connectionAccepter_(context_)
    {
        if (!utile::IsIPV4(host))
        {
            throw std::runtime_error("Invalid IPV4 ip adress: " + host);
        }

        boost::asio::ip::tcp::endpoint endpoint(
            boost::asio::ip::address::from_string(host), port);
        connectionAccepter_.open(endpoint.protocol());
        connectionAccepter_.set_option(
            boost::asio::ip::tcp::acceptor::reuse_address(false));
        connectionAccepter_.set_option(
            boost::asio::ip::tcp::acceptor::broadcast(false));
        connectionAccepter_.bind(endpoint);
        connectionAccepter_.listen();

        for (uint32_t id = 0; id < 1000; id++) { availableIds_.push(id); }
        threadUpdate_ = std::thread([&]()
            { LOG_INF << "START UPDATING"; while (!shuttingDown_) { update(); }}
    }
```

```
virtual ~Server() noexcept
{
    shuttingDown_ = true;
    LOG_INF << "STOPING SERVER";
    stop();
}



bool start()
{
    try
    {
        waitForClientConnection();
        threadContext_ = std::thread([this]() { context_.run(); });
    }
    catch(const std::exception& e)
    {
        LOG_ERR << "Server exception";
        logger_ << e.what() << '\n';
        return false;
    }

    LOG_INF <<"Server Started\n";
    return true;
}



void stop()
{
    LOG_DBG << "Stopping context";
    context_.stop();

    if (threadContext_.joinable())
        threadContext_.join();

    LOG_INF << "Stopping thread update";
    condVarUpdate_.notify_one();
    if (threadUpdate_.joinable())
        threadUpdate_.join();

    LOG_INF <<"Server Stopped\n";
}
```

```
void update()
{
    std::unique_lock<std::mutex> ulock(mutexUpdate_);
    LOG_DBG << "UPDATING: Waiting for incoming message";
    condVarUpdate_.wait(ulock, [&] { return !incomingMessagesQueue_.empty()

    if (shuttingDown_)
    {
        return;
    }

    LOG_DBG << "UPDATING: Handling new message";
    auto maybeMsg = incomingMessagesQueue_.pop();

    if (maybeMsg.has_value())
    {
        auto msg = maybeMsg.value().first;
        onMessage(msg.remote, msg.msg);
    }
}
// ASYNC OK
void waitForClientConnection()
{
    connectionAccepter_.async_accept(
        [this](std::error_code errcode, boost::asio::ip::tcp::socket socket)
        {
            if (!errcode)
            {
                LOG_INF << "Attempting to connect to " << socket.remote_endp
                std::shared_ptr<Connection<T>> newConnection =
                    std::make_shared<Connection<T>>(
                        Owner::Server,
                        context_,
                        std::move(socket),
                        incomingMessagesQueue_,
                        condVarUpdate_);

                if (onClientConnect(newConnection))
                {
                    auto idCounter = availableIds_.pop();
                    if (!idCounter.has_value())
                    {
                        LOG_ERR << "Server doesn't support any more connecti
                        return;
                    }
```

```
                        connections_[idCounter.value()] = std::shared_ptr<Connec
                        connections_[idCounter.value()]->connectToClient(idCount

                        LOG_INF << connections_[idCounter.value()]->getId() << "
                    }
                    else
                    {
                        LOG_WARN <<"Connection has been denied";
                    }
                }
                else
                {
                    LOG_ERR << "Connection Error " << errcode.message();
                }

                waitForClientConnection();
            });
    }

    void messageClient(std::shared_ptr<Connection<T>> client, const Message<T>&
    {
        std::scoped_lock lock(mutexMessage_);
        if (client && client->isConnected())
        {
            client->send(msg);
        }
        else if (client)
        {
            onClientDisconnect(client);
            connections_.erase(client->getId());
            availableIds_.push(client->getId());
            client.reset();
        }
    }

protected:
    virtual bool onClientConnect(std::shared_ptr<Connection<T>> client)
    {
        return false;
    }

    virtual void onClientDisconnect(std::shared_ptr<Connection<T>> client) {}

    virtual void onMessage(std::shared_ptr<Connection<T>> client,
        Message<T>& msg) {}
};
```

To be able to have a common languages between all of our servers and clients we defined an easy to use message class. We mostly followed common standards, apart from the fact that the message header has a priorty boolean value.

```
template <typename T>
struct MessageHeader
{
    T type{};
    uint16_t id{};
    bool hasPriority = false;
    size_t size = 0;
};

template <typename T>
struct Message
{
    MessageHeader<T> header{};
    std::vector<uint8_t> body;

    size_t size() const
    {
        return body.size();
    }

    friend std::ostream& operator << (std::ostream& os, const Message<T>& msg)
    {
        os  << "ID:" << msg.header.id
            << " Size:" << msg.header.size
            << " HasPriority:" << msg.header.hasPriority
            << " Type:" << int(msg.header.type);
        return os;
    }

    template<typename DataType>
    friend Message<T>& operator << (Message<T>& msg, const DataType& data)
    {
        static_assert(std::is_standard_layout<DataType>::value,
            "Data can not be pushed");
        size_t sizeBeforePush = msg.body.size();
        msg.body.resize(sizeBeforePush + sizeof(DataType));
        std::memcpy(msg.body.data() + sizeBeforePush, &data, sizeof(DataType));
        msg.header.size = msg.size();

        return msg;
    }
```

```
template<typename DataType>
friend Message<T>& operator << (Message<T>& msg,
    const std::vector<DataType>& dataVec)
{
    static_assert(std::is_standard_layout<DataType>::value,
        "Data can not be pushed");
    size_t sizeBeforePush = msg.body.size();
    msg.body.resize(sizeBeforePush + (sizeof(DataType) * dataVec.size()));
    std::memcpy(
        msg.body.data() + sizeBeforePush,
        dataVec.data(),
        (sizeof(DataType) * dataVec.size()));
    msg.header.size = msg.size();

    return msg;
}



template<typename DataType>
friend Message<T>& operator >> (Message<T>& msg, DataType& data)
{
    static_assert(std::is_standard_layout<DataType>::value,
        "Data can not be poped");

    if (msg.body.size() < sizeof(DataType))
    {
        std::cerr << "WARN [MESSAGE] Insufficient data\n";
        return msg;
    }

    size_t sizeAfterPop = msg.body.size() - sizeof(DataType);
    std::memcpy(&data, msg.body.data() + sizeAfterPop, sizeof(DataType));
    msg.body.resize(sizeAfterPop);
    msg.header.size = msg.size();

    return msg;
}
```

```
friend Message<T>& operator >> (Message<T>& msg, std::string& data)
{
    if (msg.body.size() < sizeof(char) * data.size())
    {
        std::cerr << "WARN] [MESSAGE] Insufficient data\n";
        return msg;
    }

    size_t sizeAfterPop = msg.body.size() - sizeof(char) * data.size();
    std::memcpy(
        data.data(),
        msg.body.data() + sizeAfterPop,
        sizeof(char) * data.size());
    msg.body.resize(sizeAfterPop);
    msg.header.size = msg.size();

    return msg;
}



template<typename DataType>
friend Message<T>& operator >> (Message<T>& msg, std::vector<DataType>& data
{
    static_assert(std::is_standard_layout<DataType>::value,
        "Data can not be poped");

    if (msg.body.size() < sizeof(DataType) * dataVec.size())
    {
        std::cerr << "WARN] [MESSAGE] Insufficient data\n";
        return msg;
    }

    size_t sizeAfterPop = msg.body.size()
        - sizeof(DataType) * dataVec.size();
    std::memcpy(
        dataVec.data(),
        msg.body.data() + sizeAfterPop,
        sizeof(DataType) * dataVec.size());
    msg.body.resize(sizeAfterPop);
    msg.header.size = msg.size();

    return msg;
}
```

```
    friend Message<T>& operator << (Message<T>& msg, const std::string& data)
    {
        std::vector<char> convertedString(data.begin(), data.end());

        size_t sizeBeforePush = msg.body.size();
        msg.body.resize(sizeBeforePush + sizeof(char) * convertedString.size());
        std::memcpy(msg.body.data() + sizeBeforePush, convertedString.data(), si
        msg.header.size = msg.size();

        return msg;
    }


    void clear()
    {
        body.clear();
        header.id = 0;
        header.hasPriority = false;
        header.size = this->size();
    }

    Message<T> clone()
    {
        Message<T> copy;
        copy.header = this->header;
        copy.body = this->body;

        return copy;
    }
};

template<typename T>
struct OwnedMessage
{
    std::shared_ptr<Connection<T>> remote; // WE SEND WHOLE OBJECT OVER THE NETW
    Message<T> msg;
    friend std::ostream& operator << (std::ostream& os, const OwnedMessage<T>& m
    {
        os << msg.msg();
        return os;
    }
    OwnedMessage(const std::shared_ptr<Connection<T>>& remotec, const Message<T>
        remote{ remotec }, msg {msgc} {}
};
```

### 4.3 ModelTrainig

TO DO

## 4.4 CarDetector

TO DO: I am currently changing this part so will add it later, changing from cv::CascadeClassifier with a car model found online to generating my own

## 4.5 Proxy

The proxy assures the connection between the vechile trackers and the next junction main server they will encounter. Every single one is connected to they own database, that contains a list of junctions, proxys and the area covered by them. They way it works is preaty straight forward, if a client querys the proxy for the next junction it searches the database. If it finds a suiting junction then it sends the connection data, otherwise it redirects the vehicle to the closeste proxy. This process repeats itself until it finally reaches one valid junction or the car is out of coverage area (for example if the car is somewhere on a ship in the middle of the sea there is no reason for us to connect it to a junction).

```
class ProxyServer :
public ipc::net::Server<ipc::VehicleDetectionMessages>
{
private:
        common::db::ProxyPtr dbProxy_;
        std::unique_ptr<::utile::DBWrapper> dbWrapper_;
        LOGGER("PROXY-SERVER");

        bool isCoveredByProxy(
    ipc::utile::ConnectionPtr client,
    ipc::utile::VehicleDetectionMessage msg);

        ipc::net::ProxyReply buildProxyReply(
    ipc::utile::VehicleDetectionMessage& msg,
    const common::db::JunctionPtr& junction) const;

        ipc::net::ProxyRedirect buildProxyRedirect(
    ipc::utile::VehicleDetectionMessage& msg,
    const common::db::ProxyPtr& proxy) const;

        void rejectMessage(
    ipc::utile::ConnectionPtr client,
    ipc::utile::VehicleDetectionMessage& msg);

        bool isMessageValid(
    ipc::utile::ConnectionPtr client,
    ipc::utile::VehicleDetectionMessage& msg);

        void handleMessage(
    ipc::utile::ConnectionPtr client,
    ipc::utile::VehicleDetectionMessage& msg);

        std::optional<ipc::net::ProxyReply> getClosestJunctionReply(
    ipc::utile::VehicleDetectionMessage& msg);
```

```
        void redirect(
    ipc::utile::ConnectionPtr client,
    ipc::utile::VehicleDetectionMessage& msg);
public:
        ProxyServer(const ipc::utile::IP_ADRESS& host,
                const ipc::utile::PORT port,
                const common::db::ProxyPtr& dbProxy,
                const utile::DBConnectionData& connectionData);
        ProxyServer(const ProxyServer&) = delete;
        virtual ~ProxyServer() noexcept = default;

        virtual bool onClientConnect(
    ipc::utile::ConnectionPtr client) override;
        virtual void onClientDisconnect(
    ipc::utile::ConnectionPtr client) override;
        virtual void onMessage(
    ipc::utile::ConnectionPtr client,
    ipc::utile::VehicleDetectionMessage& msg) override;
};
```

## 4.6 GPSVechileTracking

VechileTracking submodule is a wrapper over the client class that also handles GPS interactions, it is capable of parsing NMEA GPGLL, GPGGA, GPRM and GPGGA formated data and extracting the latitude and longitude of the vehicle. With the help of the above present proxy it signal the JMS it's arrival and departure. It uses a predefined config file that auto updates, so that every time it is redirected to a proxy the data is saved and it starts with quering the last proxy it was conected to and in case of failure just moves up until reaching the inital global proxy.

```
class VehicleTrackerClient :
    public ipc::net::Client<ipc::VehicleDetectionMessages>
{
private:
    ipc::utile::MessageIdProvider<ipc::VehicleDetectionMessages> msgIdProvider_;
    GPSAdapter gpsAdapter_;
    std::thread threadProcess_;
    std::mutex mutexProcess_;
    std::condition_variable condVarProcess_;
    std::atomic_bool shouldPause_ = true;
    std::atomic_bool shuttingDown_ = false;
    std::atomic_bool isRedirected_ = false;

    std::stack<std::pair<ipc::utile::IP_ADRESS, ipc::utile::PORT>> lstVstedPrxys
    std::optional<std::string> signature_;

    bool isEmergency_;
    std::shared_ptr<common::db::Junction> nextJunction_;
    LANE followedLane_;

    LOGGER("VEHICLETRAKER-CLIENT");

    void process();
    bool switchConnectionToRedirectedProxy(ipc::net::ProxyRedirect& redirect);
    bool handleProxyAnswear(
        ipc::net::Message<ipc::VehicleDetectionMessages>& msg);
    bool queryProxy();
    bool setupData(ipc::net::ProxyReply& reply);
    bool notifyJunction();
    void waitToPassJunction();
public:
    VehicleTrackerClient() = delete;
    VehicleTrackerClient(
        const std::string& pathConfigFile,
        std::istream& inputStream);
```

```
    virtual ~VehicleTrackerClient() noexcept;

    bool saveDataToJson();

    bool start();
    void pause();
    void stop();
};


class GPSAdapter
{
protected:
        std::optional<GeoCoordinate<DecimalCoordinate>> lastCoordinates_;
        std::istream& inputStream_;
        std::thread threadProcess_;
        std::mutex mutexProcess_;
        std::condition_variable condVarProcess_;
        std::atomic_bool shouldPause_ = true;
        std::atomic_bool shuttingDown_ = false;

        void process();
        int hexStringToInt(std::string& value);
        int calculateCheckSum(std::string NMEAString);
        std::string getNextValue(std::string& NMEAString, size_t& start);
        std::optional<GeoCoordinate<DecimalCoordinate>> parseGPGLLString(
        std::string NMEAString);
        std::optional<GeoCoordinate<DecimalCoordinate>> parseGPGGAString(
        std::string NMEAString);
        std::optional<GeoCoordinate<DecimalCoordinate>> parseGPRMCString(
        std::string NMEAString);
        std::optional<GeoCoordinate<DecimalCoordinate>> parseNMEAString(
        std::string& NMEAString);
public:
        GPSAdapter() = delete;
        GPSAdapter(std::istream& inputStream);
        ~GPSAdapter() noexcept;
        std::optional<GeoCoordinate<DecimalCoordinate>> getCurrentCoordinates();

        bool start();
        void pause();
        void stop();
};
```

```
std::optional<GeoCoordinate<DecimalCoordinate>>
    GPSAdapter::parseGPGLLString(std::string NMEAString)
{
        //"$GPGLL", <lat>, "N/S", <lon>, "E/W", <time>(hhmmss.sss), "A/V",
    // "A/D/E/M/N" "*" <checksum> "CR/LF"(ignored in our case)
        GeoCoordinate<DecimalCoordinate> rez{};
        size_t start = 0;

        //$GPGLL
        std::string value = getNextValue(NMEAString, start);
        CHECK_IF_STILL_VALID_POSITION(start);
        if (value != "$GPGLL") { return {}; }

        // <lat>
        value = getNextValue(NMEAString, start);
        CHECK_IF_STILL_VALID_POSITION(start);

        auto latitude = common::utile::StringToDecimalCoordinates(value);
        if (!latitude.has_value()) { return {}; }

        // N/S
        value = getNextValue(NMEAString, start);
        CHECK_IF_STILL_VALID_POSITION(start);
        if (!(value == "N" || value == "S")) { return {}; }
        if (value == "S") { latitude = -latitude.value(); }

        // <lon>
        value = getNextValue(NMEAString, start);
        CHECK_IF_STILL_VALID_POSITION(start);

        auto longitude = common::utile::StringToDecimalCoordinates(value);
        if (!longitude.has_value()) { return {}; }

        //take E/W
        value = getNextValue(NMEAString, start);
        CHECK_IF_STILL_VALID_POSITION(start);
        if (!(value == "E" || value == "W")) { return {}; }
        if (value == "W") { longitude = -longitude.value(); }

        // <time>(hhmmss.sss) IGNORED
        value = getNextValue(NMEAString, start);
        CHECK_IF_STILL_VALID_POSITION(start);

        // A/V if V return {}
```

```
        value = getNextValue(NMEAString, start);
        CHECK_IF_STILL_VALID_POSITION(start);
        if (value == "V" ) { return {}; }

        // "A/D/E/M/N" useless so just skip it
        start++;
        CHECK_IF_STILL_VALID_POSITION(start);

        // "*" <checksum> if not matching return {}
        if (NMEAString[start] != '*') { return {}; }
        start++;
        CHECK_IF_STILL_VALID_POSITION(start);

        value = NMEAString.substr(start, NMEAString.size() - start + 1);
        if (calculateCheckSum(std::string(NMEAString.substr(1, start - 2)))
        != hexStringToInt(value)) { return {}; }

        rez.latitude = latitude.value();
        rez.longitude = longitude.value();
        return rez;
}
```

## 4.7 TrafficObserver

This submodule is just a wrapper over the client that combines cardetection functionalities. The main difference from the other client is that it's lane is predefined and it is identified by a keyword given inside the config file to the JMS.

```
class TrafficObserverClient :
    public ipc::net::Client<ipc::VehicleDetectionMessages>
{
private:
    ipc::utile::MessageIdProvider<ipc::VehicleDetectionMessages> msgIdProvider_;
    cvision::ObjectTracker carTracker_;
    common::utile::IObserverPtr observer_;
    std::function<void()> observer_callback_;
    std::string keyword_;

    LOGGER("TRAFFICOBSERVER-CLIENT");

    bool startTrackingCars();

    void stopTrackingCars();

    bool sendData(size_t numberOfCars, bool leftLane = false);

public:

    TrafficObserverClient(std::string keyword, std::filesystem::path videoPath);

    TrafficObserverClient(std::string keyword);

    ~TrafficObserverClient();

    void handleNewCarData();
};
```

## 4.8 JMS-TrafficManagement

To implement the phase transition I chose to create a state machine using Boost statechart predefined template. The traffic transition are represented by boost::statechart::events and and the traffic state are represented by boost::statechart::simplestate.

```
struct NormalTransition : sc::event<NormalTransition>
{
    NormalTransition() = default;
    virtual ~NormalTransition() noexcept = default;
};

struct JumpTransition : sc::event<JumpTransition>
{
    std::string nextTransitionName_;
    JumpTransition() = delete;
    JumpTransition(const std::string nextTransition) :
        nextTransitionName_(nextTransition)
    {}
    virtual ~JumpTransition() noexcept = default;
};

struct Stopped : sc::simple_state <Stopped, BaseState>
{
    typedef sc::transition<Start ,EWTransition> reactions;
};

// STATE I/IV
struct EWTransition : sc::simple_state <EWTransition, BaseState>
{
    typedef  mpl::list < sc::transition<NormalTransition,
        NTransition> > reactions;

    EWTransition()
    {
        std::cout << "State EW started";
    }

    virtual ~EWTransition()
    {
        std::cout << "State ended";
        outermost_context().resetTimers("EW");
        outermost_context().nextNormalState_ = "N";
    }
};
```

```
// STATE II
struct NTransition : sc::simple_state <NTransition, BaseState>
{
    typedef mpl::list < sc::transition<NormalTransition,
        STransition>> reactions;

    NTransition()
    {
        std::cout << "State N started";
    }

    virtual ~NTransition()
    {
        std::cout << "State ended";
        outermost_context().resetTimers("N");
        outermost_context().nextNormalState_ = "S";
    }
};
```

```
// STATE III
struct STransition : sc::simple_state <STransition, BaseState>
{
    typedef mpl::list <sc::transition<NormalTransition,
        EWTransitionCpy>> reactions;

    STransition()
    {
        std::cout << "State S started";
    }

    ~STransition()
    {
        std::cout << "State ended";
        outermost_context().resetTimers("S");
        outermost_context().nextNormalState_ = "EW";
    }
};
```

```
// STATE I/IV
struct EWTransitionCpy : sc::simple_state <EWTransitionCpy, BaseState>
{
    typedef mpl::list < sc::transition<NormalTransition,
        NSTransition> > reactions;

    EWTransitionCpy()
    {
        std::cout << "State EW started";
    }

    virtual ~EWTransitionCpy()
    {
        std::cout << "State ended";
        outermost_context().resetTimers("EW");
        outermost_context().nextNormalState_ = "NS";
    }
};




// STATE V/VIII
struct NSTransition : sc::simple_state <NSTransition, BaseState>
{
    typedef mpl::list <sc::transition<NormalTransition,
        ETransition>> reactions;

    NSTransition()
    {
        std::cout << "State NS started";
    }

    virtual ~NSTransition()
    {
        std::cout << "State ended";
        outermost_context().resetTimers("NS");
        outermost_context().nextNormalState_ = "E";
    }
};
```

```
// STATE VI
struct ETransition : sc::simple_state <ETransition, BaseState>
{
    typedef mpl::list <sc::transition<NormalTransition,
        WTransition>> reactions;

    ETransition()
    {
        std::cout << "State E started";
    }

    virtual ~ETransition()
    {
        std::cout << "State ended";
        outermost_context().resetTimers("E");
        outermost_context().nextNormalState_ = "W";
    }
};
```

```
// STATE VII
struct WTransition : sc::simple_state <WTransition, BaseState>
{
    typedef mpl::list <sc::transition<NormalTransition,
        NSTransitionCpy>> reactions;

    WTransition()
    {
        std::cout << "State W started";
    }

    virtual ~WTransition()
    {
        std::cout << "State ended";
        outermost_context().resetTimers("EW");
    }
};
```

```
// STATE V/VIII
struct NSTransitionCpy : sc::simple_state <NSTransitionCpy, BaseState>
{
    typedef mpl::list <sc::transition<NormalTransition,
        EWTransition>> reactions;

    NSTransitionCpy()
    {
        std::cout << "State NS started";
    }

    virtual ~NSTransitionCpy()
    {
        std::cout << "State ended";
        outermost_context().resetTimers("NS");
        outermost_context().nextNormalState_ = "EW";
    }
};


struct TrafficLightStateMachine :
    sc::state_machine<TrafficLightStateMachine, BaseState>,
    std::enable_shared_from_this<TrafficLightStateMachine>
{
public:
    std::string nextNormalState_ = "N";
private:
    // CONFIG DATA
    bool usingLeftLane_;
    boost::optional<common::utile::LANE> missingLane_ = boost::none;

    std::mutex mutexClients_;
    std::map<common::utile::LANE, ipc::utile::IP_ADRESS> laneToVTIPAdress_;
    std::map<common::utile::LANE, ipc::utile::IP_ADRESSES> clientsConnected_
    std::map<common::utile::LANE, common::utile::TimerPtr> laneToTimerMap_;
    common::utile::ThreadSafeQueue<JumpTransition> jumpTransitionQueue_;

    common::utile::ThreadSafeQueue<std::pair<common::utile::LANE,
        ipc::utile::IP_ADRESS>> waitingEmergencyVehicles_;

    uint16_t regLightDuration_;
    uint16_t greenLightDuration_;
    IObserverPtr greenLightObserver_;
    std::function<void()> greeLightObserverCallback_;
    common::utile::Timer greenLightTimer_;
```

```
    LOGGER("TRAFFICLIGHT-STATEMACHINE");

    uint16_t calculateTimeDecrease(
        const common::utile::LANE lane, ipc::utile::IP_ADRESS ip);
    void updateTrafficState();
    void updateGreenLightDuration();
public:
    TrafficLightStateMachine(const common::utile::model::JMSConfig& config);
    TrafficLightStateMachine(const TrafficLightStateMachine&) = delete;
    virtual ~TrafficLightStateMachine() noexcept = default;

    bool isUsingLeftLane();
    // BASED ON THE LANE WE WILL
    //DETERMINE WHAT PHAZE TO START:
    //CAN BE EITHER II, III, VI or VII
    // as they are the only ones that
    //allow the vehicles to move freely from one lane
    bool isVehicleTracker(
        const common::utile::LANE lane, ipc::utile::IP_ADRESS ip) const;
    boost::optional<common::utile::LANE> getVehicleTrackerLane(
        const ipc::utile::IP_ADRESS& ip);
    bool isLaneMissing(const common::utile::LANE lane) const;

    bool isClientValid(
        const common::utile::LANE lane, ipc::utile::IP_ADRESS ip);
    bool registerClient(
        const common::utile::LANE lane,
        ipc::utile::IP_ADRESS ip,
        bool leftLane);
    bool unregisterClient(ipc::utile::IP_ADRESS ip);

    bool startEmergencyState(
        const common::utile::LANE lane, ipc::utile::IP_ADRESS ip);
    bool isInEmergencyState();
    bool endEmergencyState(ipc::utile::IP_ADRESS ip);
    void freezeTimers(const std::string lanes);
    void resetTimers(const std::string lanes);
    void decreaseTimer(
        const common::utile::LANE lane, ipc::utile::IP_ADRESS ip);
    void greenLightExpireCallback();
    void queueNextStatesWaiting();

    bool registerVehicleTrackerIpAdress(
        const common::utile::LANE lane,
        const ipc::utile::IP_ADRESS ipAdress);
};
```

# References


[1] Dex R. Aleko and Soufiene Djahel. "An IoT Enabled Traffic Light Controllers Synchronization Method for Road Traffic Congestion Mitigation". In: *2019 IEEE International Smart Cities Conference (ISC2)*. IEEE, Oct. 2019. DOI: 10.1109/isc246665.2019.9071667.

[2] Akshay D. Deshmukh and Ulhas B. Shinde. "A low cost environment monitoring system using raspberry Pi and arduino with Zigbee". In: *2016 International Conference on Inventive Computation Technologies (ICICT)*. IEEE, Aug. 2016. DOI: 10.1109/inventive.2016.7830096.

[3] Junchen Jin, Xiaoliang Ma, and Iisakki Kosonen. "An intelligent control system for traffic lights with simulation-based evaluation". In: *Control Engineering Practice* 58 (Jan. 2017), pp. 24–33. DOI: 10.1016/j.conengprac.2016.09.009.

[4] Till Neudecker et al. "Feasibility of virtual traffic lights in non-line-of-sight environments". In: *VANET '12: Proceedings of the ninth ACM international workshop on Vehicular inter-networking, systems, and applications* (June 2012). DOI: 10.1145/2307888.2307907.

[5] Marcel Sheeny et al. "RADIATE: A Radar Dataset for Automotive Perception in Bad Weather". In: *2021 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, May 2021. DOI: 10.1109/icra48506.2021.9562089.

[6] K.T.K. Teo, W.Y. Kow, and Y.K. Chin. "Optimization of Traffic Flow within an Urban Traffic Light Intersection with Genetic Algorithm". In: *2010 Second International Conference on Computational Intelligence, Modelling and Simulation*. IEEE, Sept. 2010. DOI: 10.1109/cimsim.2010.95.

[7] Ishu Tomar, Indu Sreedevi, and Neeta Pandey. "State-of-Art Review of Traffic Light Synchronization for Intelligent Vehicles: Current Status, Challenges, and Emerging Trends". In: *Electronics* 11.3 (Feb. 2022), p. 465. DOI: 10.3390/electronics11030465.

[8] R. Zhang et al. "Increasing Traffic Flows with DSRC Technology: Field Trials and Performance Evaluation". In: *IECON 2018 - 44th Annual Conference of the IEEE Industrial Electronics Society*. IEEE, Oct. 2018. DOI: 10.1109/iecon.2018.8591074.