Server based system for small cities traffic lights management

Francesco Ercolani

February 2024

Indice

1 Abstract

Almost everywhere, traffic lights are managed as independent units in each intersection. Each one has its own *state cycle* and it is synchronized with its corrisponding twin on the other side of the street.

The timing for each state might vary depending on the type of the street. This mechanism assures a stable management of the single intersection where average traffic is homogeneous. But what happens if certain streets have unpredictable peaks of traffic in certain time frames? Or what happens in case of exceptional events, such as a football game, that could cause certain city areas to be more trafficated rather than others?

All these problems could be solved with a dynamic management of the sempahores cycles based on data that tells which one requires more green light time than the others.

This project aims to develop a system where a central server acquires the data sent by sensors placed in the streets near the intersecionts and continuously process the timings each semaphore cycle will be based on. The main focus of the project is to show the interaction between these entities, by providing a simple interface that emulates a streetmap window which shows the vehicle movements in real time.

2 Goal/Requirements

The goal of the project is to develop a system which optimizes the traffic lights management exploiting the tools provided by a distribute organization.

2.1 Q/A

The questions and answers to clarify the functional requirements are listed as below:

- *How is the system started?*

  When the server is run, the application can be executed. Vehicles start appearing in the map, riding the streets and stopping whenever they encounter a red light. While the server is up the system works normally assigning the green light timings accordingly to the data received.

- *What happens if the application is run before the server*

  The traffic can still be run, but all the traffic lights will be off. Only by turning the server on first the system will start correctly.

- *How many servers are up?*

  There are 2 types of server:

1. Main server: is the server that computes the data and communicates with the entities to manage the semaphores.
2. Backup server: one or more servers that check whether the main server is up, to take its place if it goes down not to make the system stop working.

- *Who are the involved actors?*

  The involved actors are the user and the server administrator:

  - The user can see the vehicles traffic and the traffic lights operating, together with the data produced real-time.
  - The server administrator is able to crash or/and to restart the servers.

- *What is the purpose of having 2 types of server?*

  The system doesn't only aim at showing the potential of a server based traffic lights system, but also wants to provide a mechanism to assure resiliency. The possibility of the server administrator to crash the servers is a way to show the behaviour of the whole environment in those certain situations.

- *So what does it happen if the main server crashes?*

  In this situation, if there's a backup server up, it takes the main's place and immediately starts doing its job. The traffic lights will not be affected by this.

- *What does it happen if all the servers go down?*

  If all the servers go down, the traffic lights become "autonomous", so the green and red light timings become the same (20 seconds).

- *How are the vehicles controlled?*

  Since the whole project was impossibile to be created in a "real environment", the traffic of vehicles is simulated and many real-life factors are neglected. Each vehicle is generated to be autonomous and to make turns as random as possible to mock a real behaviour.

- *What are the entities that compose the communication network?*

  The entities involved are: traffic lights, sensors, servers. The interaction between them will be technically explained later. In simple terms, sensors data give information about how many vehicles are nearby the semaphores and, based on that, the server assigns the red/green light timings.

- *What kind of data is provided to the user?*

  The user is able to view data relative to the sensors and the semaphores:

  - the number of vehicles referred to a specific street and semaphore.

- the number of seconds assigned to each semaphore both for green and red light.

- *What is meant with "sempahore"?*

  Since the intersections are only 4 ways intersections, in each one there are 2 semaphores. One semaphore is to be considered as a couple of poles, one positioned in a direction of the street and the other one in the opposite direction, both synchronized to display the same color.

2.2 Scenarios

In this chapter the scenarios that the users encounter are shown in the form of use case diagrams.

2.2.1 System start scenario

As said in the previous chapter, the server must be started before the application. When more servers are started, they assume the role of backup servers since the main one is already running. The user starts the vehicles traffic only after choosing the number of vehicle and running the application.

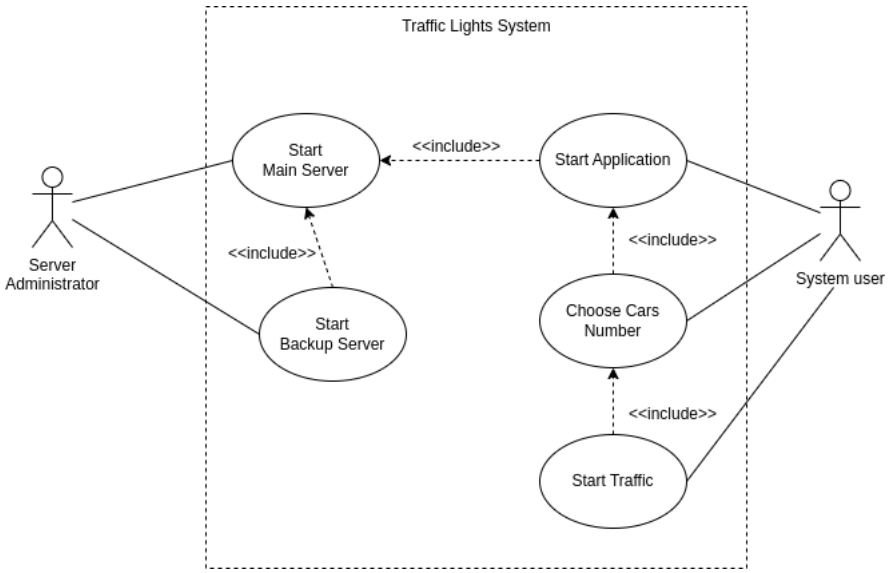


Figure 1: Use case diagram of the whole system start

2.2.2 Server crash/restart scenario

The server administrator has the power to crash and restart the servers. Since the servers recognize their role by themselves, when they are restarted the situation can be:

- main server is restarted, if there is no other server up, it remains the main server, and the whole system goes back to work normally.
- main server is restarted, but there was one or more backup server up when it crashed, so now the main server becomes a backup one. The system never stopped working.
- backup server is restarted, if the main server is down, it takes its place and becomes the main server. The system goes back to work.
- backup server is restarted, but the main was never down, so it doesn't take its place.

If all the servers crash, all the traffic lights go on "autonomous" mode.

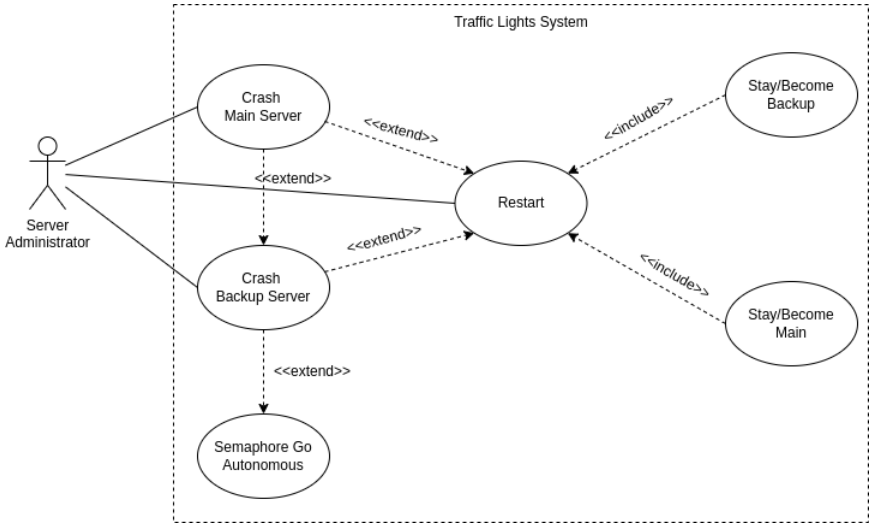


Figure 2: Use case diagram of the server crash/restart scenario

Since there might be more than one backup server, when the main crashes it is not predetermined which one of them could become the main.

2.3 Self-assessment policy

The system will be tested using the JUnit framework. The tests will be focused on the interaction between the involved entities. Less attention will be paid to the GUI aspects, the map and the motion of the vehicles logic.

3 Requirements Analysis

3.1 Implicit requirements

The project is fully developed in java and not in a real life context. This means that, for example, the servers are the same instance run more times after editing

the run configuration on *intelliJ IDEA*. In real life, they would be more seperate machines. This limitation requires the final user to follow the "rules" not to encounter unexpected behaviours.

The same goes for the sequence in which the operations are made. If the application is run before the server at the start, everything might seem working fine, but some bad behaviours can be encountered later. So, in order not to have problems, the user must be put in a situation which simulates the real life scenario.

3.2 Non-functional requirements

The project also has the goal to provide the usage of the system in a form which is user friendly and easily understandable. The user must not be confused about what is happening in the map.

Non relevant aspects of the project such as the "look" of the application and the features related to the roadmap and/or the vehicles movements won't be placed on the same level of the communication ones. The main focus will be on the distribution of the system.

4 Design

In this chapter will be shown the structure, behaviour and interaction of the project components with their respective diagrams.

4.1 Structure

The general architecture is client-server. The server is responsible for receiving the data from the sensors placed in the streets nearby the intersections, process it, and assigning the new timings to each semaphores when it receives the requests from them.

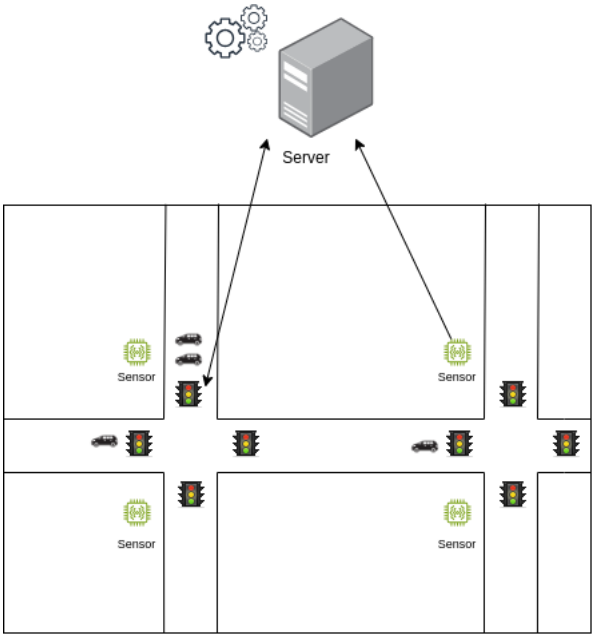


Figure 3: General schema of the structure of the system

The communication between the entities is made through TCP connections. In the specific, the server keep the ServerSocket opened listening to the connections made by the clients, and handles each one separately. Meanwhile the clients open (and close) a socket everytime a communication to the server is needed.

This implementation makes a bidirectional communication possible and this permits, beyond the consequences just listed, not to strictly respect the classical server schema where the clients necessarily send requests and the server responses. In fact, the sensors actually send data on their own without previously receiving a request.
Now let's go deeper into explaining how the server works, and how the clients (sensors and semaphores) communicate with it.

4.1.1 Server

As said, the server can be of two types, main or backup. Let's see a general schema:

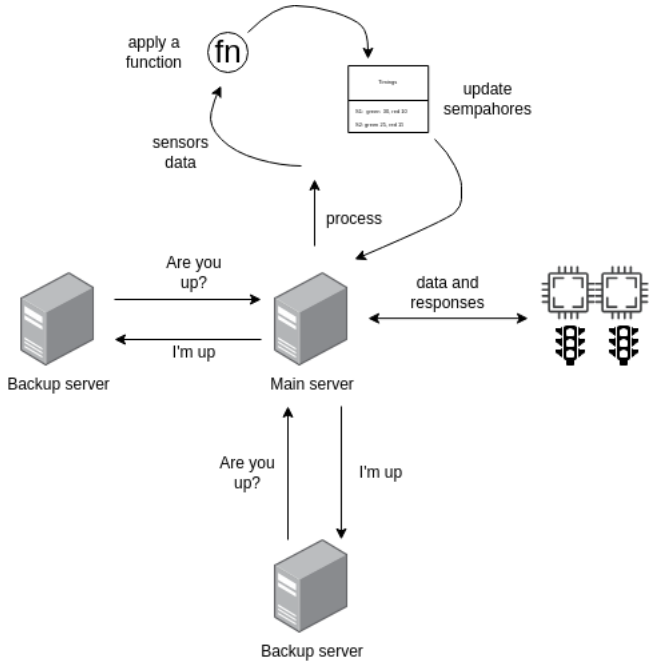


Figure 4: General schema of the structure of the system

The core of the system can be synthesized in the schema of the figure above. While the backup servers are kept alive by the send of requests, the main server receives data from the sensors and computes it applying a function, and updates a table where the green and red light times are re-assigned to each corrisponding semaphore.

Let's see in detail how this is implemented. When turned on, the server starts the GUI and set its type based on if there's, or not, already an opened *server socket*. If no server socket was opened yet, the server beceomes the main server, and starts listening to connections made to that specific socket and port.
When a connection is made, the server creates a `ServerHandler`, which is a thread which deals with the requests, and then goes back to listen to new connections. This goes on until the serversocket is closed.
The server handler initializes the `TimingProcessor`, the class that is going to calculate the timings, and the `Gson` for the serialization and deserialization. Let's see the code of the `run()` method to better understand the job of the handler:

```java
public void run() {
    try (socket) {
        var request = unmarshallRequest(socket);
        var response = computeResponse(request);
        marshallResponse(socket, response);
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

Firt, the request is unmarshalled, then the response is computed this way:

```java
private Response<?> computeResponse(Request<?> request) {
    String req = gson.toJson(request);
    try {
        switch (request.getMethod()) {
            case "status":
                return new EmptyResponse(ServerStatus.OK,
                    "Server is UP");
            case "sensors_data":
                timingProcessor.processTimings(
                gson.fromJson(req, SensorsData.class)
                .getArgument());
                return new EmptyResponse(
                    ServerStatus.OK, "Ok");
            case "timings":
                return new TimingsResponse(
                        ServerStatus.OK,
                        "Timings obtained",
                        timingProcessor.getTiming(
                        gson.fromJson(req,
                            TimingsRequest.class)
                        .getArgument()));
            default:
                return new Response<>();
        }
    } catch (Exception e) {
        return new Response<>(ServerStatus.SERVER_ERROR,
        e.getMessage());
    }
}
```

Based on what type of request it is, the handler returns its corrisponding response. The requests can be:

- status request: sent to know whether the server is still up or crashed. An empty response is sent to state that everything is ok.
- sensors_data: not an actual request, but the sensors data, which include the number of vehicles in specific sectors of the streets nearby the intersections. Server responds with an empty response.

- timing: sent by the semaphores to change their timings based on the sensors data hold by the server. Server responds with a map, where each one has the semaphore id as key and the green light time as value. The red light timing is the difference between the total semaphore cycle time (40 seconds) and the green light time.

Let's now go back to `Server` class to understand the process of crashing/restarting and server type change. As said before, when turned on, a server becomes the main server or the backup in accord to the ServerSocket status. So the two scenarios are:

- server is turned on and ServerSocket is closed: server is started as main and starts listening. If the server administrator presses the crash button, the ServerSocket gets closed and an exception get caught inside the infinite loop of the main server. After crashed, the server goes into a `crashState()` which is a `while loop` that stops when a flag is put to false. The flag is managed by the click of the "crash" and "restart" buttons.

- server is turned on and ServerSocket is opened: server is started as backup and starts sending requests about the status of the main server with the help of a `ClientHandler`. If at a certain point the request fails, it tries to obtain the ServerSocket to become the main.

To conclude, the package `presentation` contains all the serializers and deserializers plus the classes `Request` and `Reponse` and each respective extension classes. `Gson` was used as library.

4.1.2 Clients

The entities that interact with the server assume the role of clients. These entities are the sensors and the sempahores. All the aspects relative to the map creation will be left out. When the application is started, the following sequence of operations is done:

1. Main is executed, controller is started.

2. The controller creates the `StreetMap`, the instance of the class responsible for the creation of all the elements of the streetmap with all its logic: horizontal and vertical streets, the sempahores, the intersections.

3. The controller starts the view, the `VehicleViewer` (the class that call the method of the view for the updates), and the controllers. The controllers started are:

    - `NetworkController`: sends the sensors data and keeps the status requests up every 4 seconds. The communication is managed through a `ClientHandler`.

- `SensorsController`: since the sensors are simulated as they simply provide the number of vehicles in certain sections of the streets, this controller is the entity that manages the increment and decrement of the data for each sensor as if they were autonomous. The controller also provides the method `getSensorsIntersections()` needed by the `NetworkController` to send the data to the server.

4. When the button "start" is pressed, after chosen the number of vehicles, the `Controller` starts the `TrafficController`. This controller manages the creation and death of the vehicles (each vehicle is separate thread).

Now let's deepen a few points. The `ClientHandler` is the class that manages the client connections. Everytime a request is to be made, a socket is opened and the request is marshalled:

```
1 public <T, R> R rpc(Request<T> request,
2     Class<? extends Response<R>> responseType) {
3     try (var socket = new Socket()) {
4         socket.connect(this.socket);
5         marshallRequest(socket, request);
6         return unmarshallResponse(socket, responseType);
7     } catch (IOException e) {
8         throw new IllegalStateException(e);
9     }
10 }
```

The response from the server can be:

- `ok`: everything was received correctly.
- `SERVER_ERROR`: an error occurred and a message is displayed.

Given an intersection, the traffic will be governed by 2 semaphores which need to be synchronized. When one is red, the other one must be green, and viceversa. This is why when the streetmap is created, the `SemaphoreCouple` instances are started as threads, and not the single sempahores.
After the `SemaphoreCouple` is started, the client handler sends the request for the `timemap`, which is gonna be partially of totally null because no data is available yet. The 2 methods are executed:

- `startStateCycle()`: starts a thread which sets the three colors (red and green variable based on sensors data, yellow 4 seconds) in a loop.
- `startTimingsRequests()`: starts a loop where a request for new timings data is sent every x seconds.

4.1.3 Vehicles

Each vehicles is a separate thread. Within the class `Vehicle` there's all the logic that a vehicle follows such as the speed, the distance between them, the random street change when an intersection is encountered, the stop and go based on the semaphore state etc...
The vehicles are manages by `MapContext` class, which is responsible for the creation of the vehicles and the add of each one to the data structure. The positions of the vehicles are retrieved by calling the method `getAllPositions()`.
The TrafficController uses this class for the dynamic add/remove of each vehicle inside the map.

Also, a thread `VehicleViewer` is started at the start of the application, and its job is to update the GUI every x milliseconds (based on the chosen `FRAMES_PER_SEC`) by calling the GUI method `updateVehiclesPosition()`:

```
while (!stop) {
    long t0 = System.currentTimeMillis();
    mapPanel.updateVehiclesPosition(
        mapContext.getVehicles());
    long t1 = System.currentTimeMillis();
    //log("update pos");
    long dt = (1000 / FRAMES_PER_SEC) - (t1-t0);
    if (dt > 0){
        try {
            Thread.sleep(dt);
        } catch (Exception ex){
        }
    }
}
```

4.1.4 Project organization

The project follows the pattern MVC and it is divided this way:

- Model: it is composed of two main package
    1. `elements`: containing all the classes for the map logic, such as the intersections, the vehicles, the streets and the map
    2. `tfmanagement`: containing the classes for the actual management of the traffic. This package contains the client-server logic.
- View: all the GUI code is inside this package
- Controller: contains the main controller, and the single controllers for managing the communication, the sensors and the traffic. Details will be explained later.

In the next figure we can see a UML diagram of the server package involving the classes previously cited:

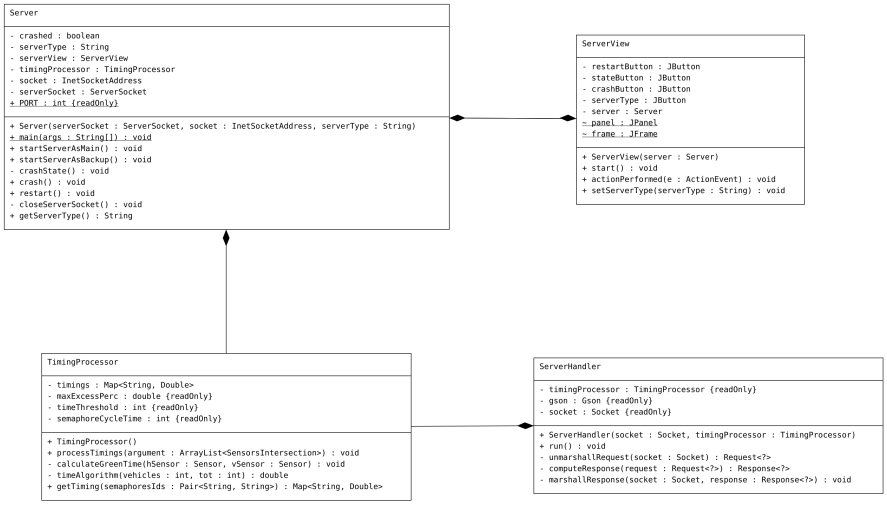


Figure 5: Server package UML diagram

In the next figure we can see a UML diagram of the clients involved:

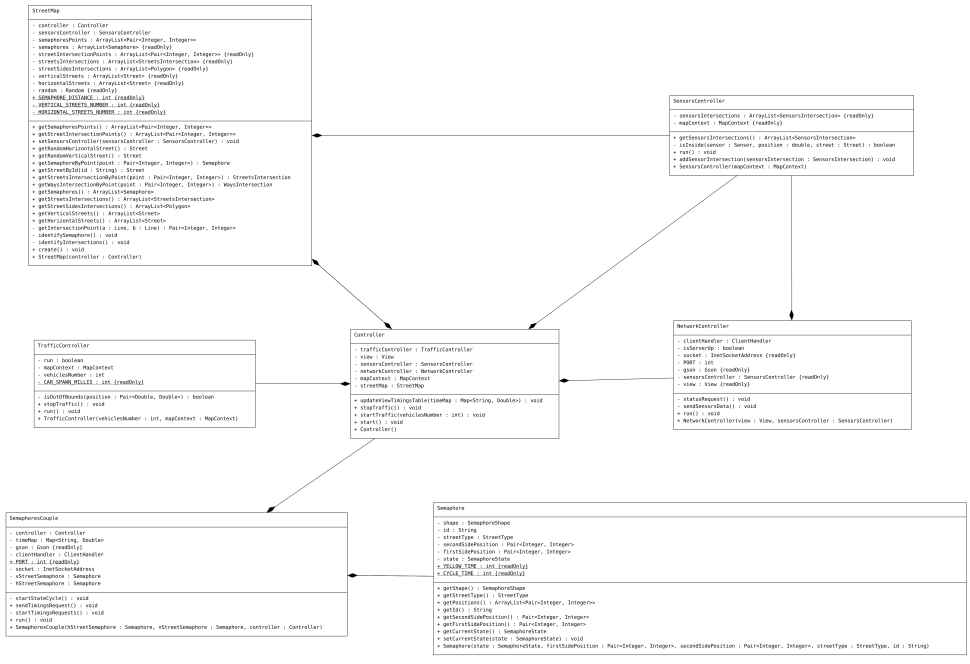


Figure 6: Clients UML diagram

4.2 Behaviour

In this chapter we are going to analyze the behaviour of the system. In the figure 7 we can see the state chart diagram of the system user point of view.

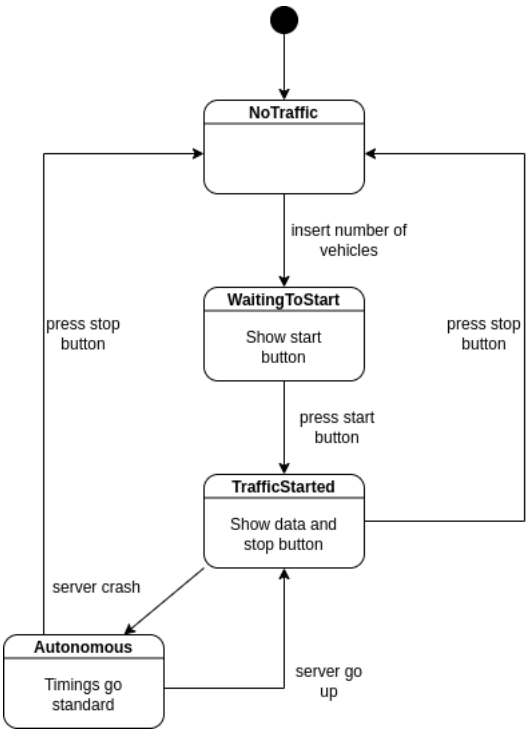


Figure 7: state chart diagram of the system user point of view

From the system user point of view, assuming the server has been started, the map with no traffic is what is first seen. When the user inserts the number of vehicles and starts the traffic, the vehicles start appearing in the map and the data is shown in the right panel. Also the stop traffic button gets enabled. At this point, if no server is reachable anymore, the system goes autonomous and the timings reset to default timings (20 sec for red, 20 for green). If the server gets reachable again, the system goes back to work, if the stop button is pressed, the vehicles stop appearing in the map.

Now let's see the state chart from the server adminstrator point of view taking 2 servers as example since the real case, where can be more than 2 servers, only introduces competition in which backup server may become the main first:

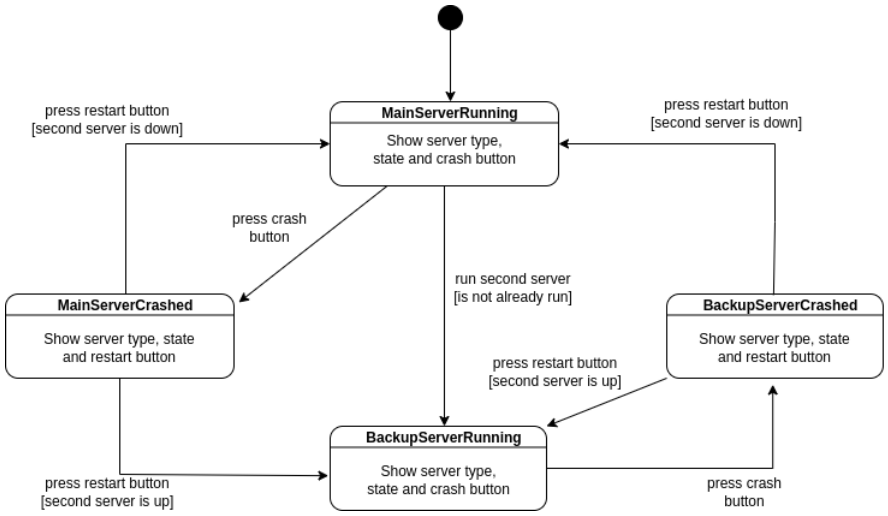


Figure 8: state chart diagram of the server administrator point of view

From the server administrator point of view, when the first server is started, it becomes the main server. The main server starts communicating and calculating the timings. If the crash button is pressed, the main server crashes. If the second server is started, it becomes the backups server and starts listening to the main. If the main server is crashed and gets restarted, if the backup is up, it takes its place and the backup becomes the main.

4.3 Interaction

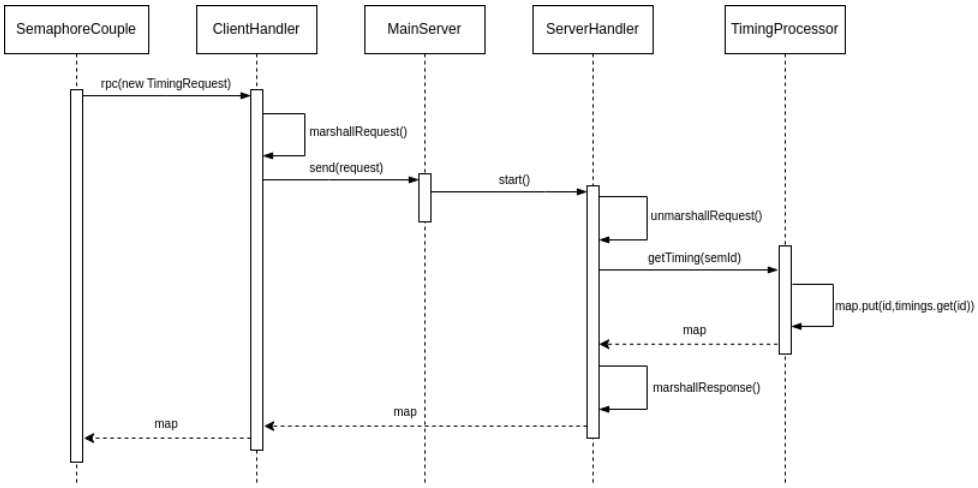


Figure 9: sequence diagram of the interaction between the semaphores and the server

In the figure 9 it is shown the sequence diagram of the interaction between the semaphores and the server for the obtaining of the timings. The request is made by the `SemaphoreCouple`, which is handled by the `ClientHandler`. The main listens to the connection and starts a `ServerHandler` thread. The handler gets the specific timings from the `TimingProcessor`, and then returns the map (the key is the semaphore id and the value is the green light time).

Let's now see the interactions between the `NetworkController` and the server for the sensors data send:

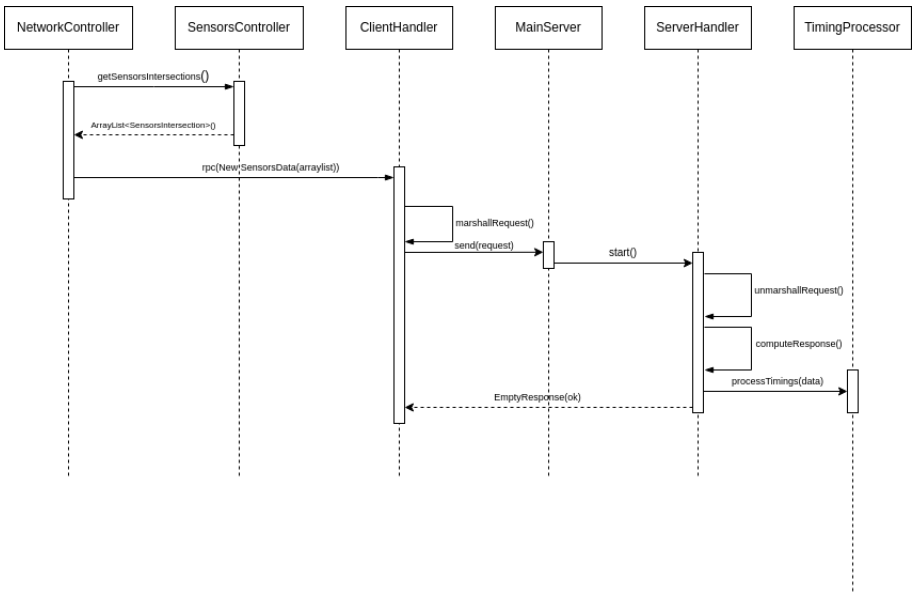


Figure 10: sequence diagram of the interaction between the NetworkController and the server

The `NetworkController` gets the arrayList of the `SensorsIntersections`. A SensorIntersection is an object which contains the sensors of the 2 interesected streets. It was created to simplify the management of the sensors within each streets and intresection. The array is then sent, handled by the `ClientHandler`, to the Main server. The server starts the handler which takes the data and sends it to the **TimingsProcessor** to process it. An "ok" response is sent if everything went fine.

To conclude, let's see the interaction between the main and the backup server:

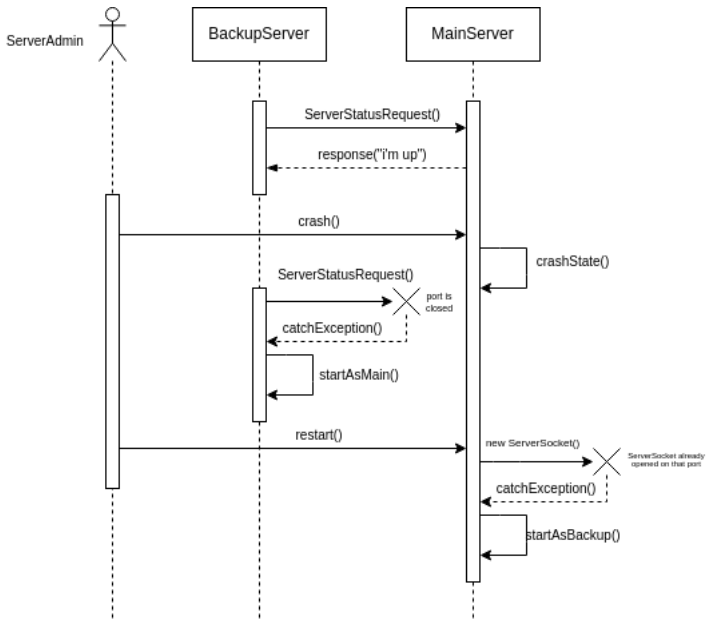
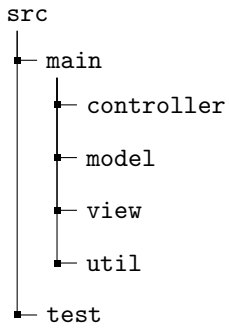


Figure 11: sequence diagram of the interaction between the main and the backup server

In this sequence diagram, the details about the handlers is skipped to focus on the interaction between the servers. While the main is running, the backup keeps sending the requests about its status. When the admin crashes the main server, and the backup sends the status request once again, an exception is caught because no ServerSocket on that port is open. The backup understands that the main is now down, and becomes itself the main server. When the admin restards the ex-main server, it tries to open a ServerSocket on that same port, but failing and understanding that the backup has taken its place. It finally becomes the backup.

5 Implementation Details

The entire project was made in Java and is divided into diffrent modules:



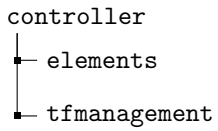
The whole project was made following the *MVC* [1] pattern. Both the GUI and the communications between the elements were made using the java built-in packages. The only external library that was used is *Gson* [2] for the serialization and deserialization.

5.1 controller

This package cointains all the controllers which have the job to connect the view to the model and to manage part of the communication.

5.2 model

In the model package are contained all the classes for the actual creation and management of the elements, and the classes for the traffic management. For the networking communication the package `java.net` [3] was used



5.3 view

The GUI is all inside the package view. For the GUI development it was used *Java Swing* [4] and *Java AWT* [5].

5.4 util

This package contains util classes used for specific operations.

5.5 test

All the tests are inside this package.

6 Self-assessment / Validation

Being the whole project not finalized to the human interaction (e.g. a game), the tests were made to execute specific tasks that could lead to problems. The tests were made with *JUnit 5* [6] framework. In the following image we can see the implemented tests:

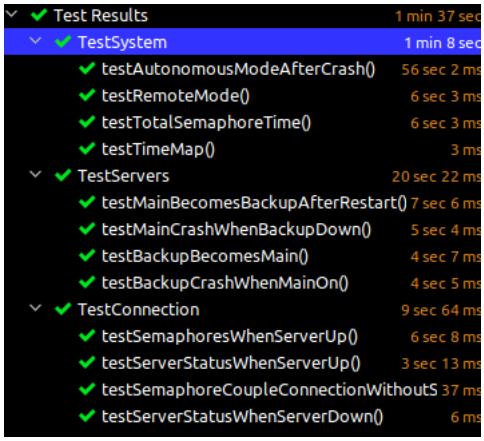


Figure 12: JUnit implemented tests

Also, for *continuous integration* purposes *Gitlab CI/CD pipelines* were exploited. Thanks to that, everytime a new push is made a new pipeline is created follwing the instruction of the `.gitlab-ci.yml` file. The content of the file is the following:

```
image: gradle:latest

build:
  stage: build
  script: gradle classes testClasses

test:
  stage: test
  script: gradle test
  artifacts:
    when: always
    reports:
      junit: '**/build/test-results/test/**/TEST-*.xml'
```

7 Deployment Instructions

To properly run the system it's enough to open 3 (if you want to have only one backup server), ore more terminals in the project root (execute the commands in the fopllowing order):

1. One for the first server (main server). To run it execute the command:
   `./gradlew runServer`
2. One for the application. To run it execute the command:
   `./gradlew runApp`
3. One or more for the backup servers. To run it execute the command:
   `./gradlew runServer`

The connections will be on `localhost` port `2000`.

8 Usage Examples

In this chapter we'll see a usage example with one backup server.

When the servers are run, the two instances of the GUI are shown as in figure 13, one for the main server and one for the backup server, showing their state and the button to crash them.

Figure 13: Main and backup server GUIs

When the application is run, but the traffic has not started yet the situation is the one shown in figure 14.

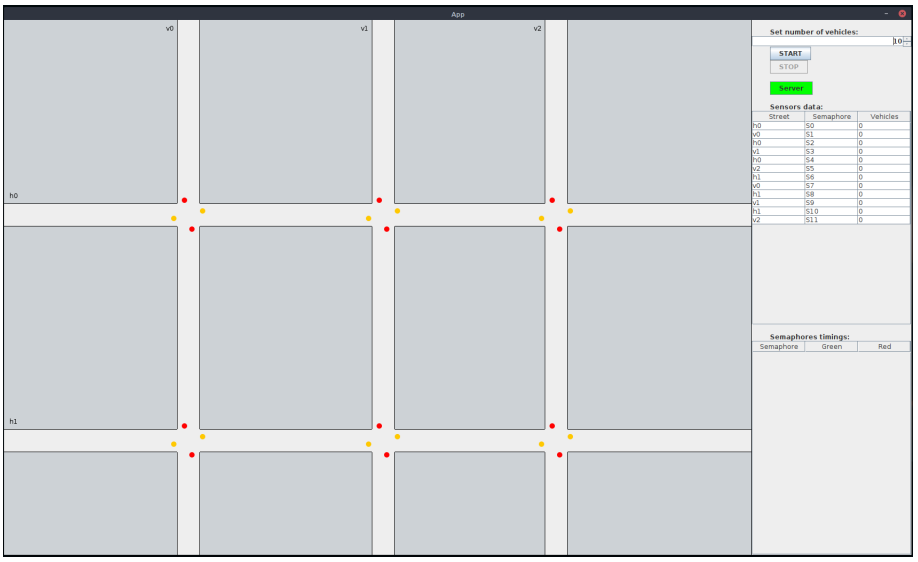


Figure 14: Application with no traffic

When the button *start* is pressed, the vehicles start appearing in the map and the data is shown in the right panel (figure 15).

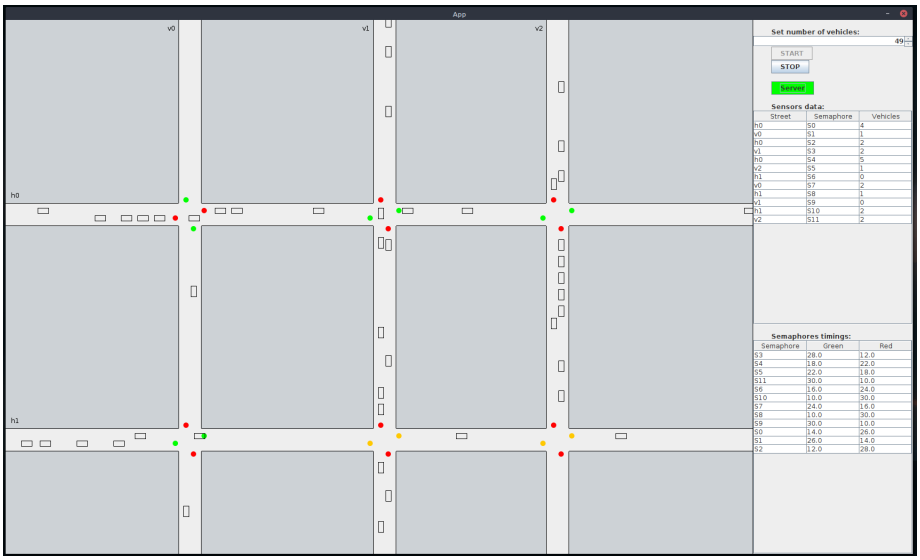


Figure 15: Application with traffic

At this point the main server is crashed, and the backup takes the control.

Figure 16: Main server crash

Figure 17: Both servers crash

At this point the main server is crashed, and the backup takes the control. The semaphores are still receiving data. If both servers crash (figure 17), the semaphores go in autonomous mode (figure 18) and the timings are set to 20 seconds. This is going to cause congestion situation in busy road where more time for green light is needed.

Figure 18: Application when servers are down

Now one server is restarted, and the system goes back to work normally:

Figure 19: Server restart

9 Problems

Vehicles do not give way to the right, and this may lead to an overlap between two or more vehicle. Since this is not important for the purposes of the project, and it's not a frequent event, it was not implemented in the project.

Screen is not resizable, the height and weight of the application window are set inside the object `MapDimension` to respectivel 1000 and 1400. If a smaller screen is used, this will lead to visual problems.

10 Conclusions

The aim of the project was to create a system that could manage a small city traffic lights network in the most efficient way. This goal has been achieved by creating a distributed system able to reflect a real life scenario.

10.1 Future Works

Future works include:

- fixing the problems explained in the previous section
- implementation of a "richer" map with more types of streets, different vehicles (with different speed, acceleration etc..) such as *A/B street* [7]
- implementation of a "more real" version of the system by using one or more *Raspberry Pi* [8] as servers

10.2 What you learned

The development of this project has contributed to enrich my knowledge in distributed systems and to practice my designing skills.

Bibliography

[1] *MVC*. URL: https://en.wikipedia.org/wiki/Model-view-controller.

[2] *Gson*. URL: https://github.com/google/gson.

[3] *java.net*. URL: https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/net/package-summary.html.

[4] *Java Swing*. URL: https://docs.oracle.com/en/java/javase/17/docs/api/java.desktop/javax/swing/package-summary.html.

[5] *Java AWT*. URL: https://docs.oracle.com/en/java/javase/17/docs/api/java.desktop/java/awt/package-summary.html.

[6] *JUnit 5*. URL: https://junit.org/junit5/.

[7] *A/B street*. URL: https://a-b-street.github.io/docs/software/abstreet.html.

[8] *Raspberry Pi*. URL: https://www.raspberrypi.com/.