Distributed Systems:
Crossing Intersection with Autonomous Vehicles

Antonio Toncetti
Gabriele Venturato

DMIF, University of Udine, Italy

September 10, 2019

Abstract

The aim of this project is to provide an implemented solution to the problem of autonomous vehicles crossing an intersection. Although the solution relies on some simplifications, it can be further elaborated to work in a real-case scenario.

The solution proposed in this report is meant to be more general and as modular as possible, in order for it to be possibly extended in a concrete situation.

Chapter 1

Introduction

The problem to solve is the one in which some autonomous vehicles have to cross an intersection without being involved in road accidents.

The idea is to solve the problem for a generic intersection. The autonomous vehicles can not rely on a central server, they have to cooperate with each other to cross the intersection by taking decisions which ensure a *fair* and *safe* policy. In particular there are two components in the proposed solution: vehicles and the environment. The latter is necessary in this context in order to simulate sensors that are usually inside autonomous vehicles which allows them to interact with the environment (e.g. proximity sensors, GPS, cameras, etc. . .). The system is *fault tolerant*, but neither byzantine processes nor cybersecurity hazards are taken in consideration, for simplification purpose.

The report starts by analysing the project: Chapter 2 is devoted to use cases, functional and non-functional requirements, and system assumptions too. Chapter 3 contains the description of the general architecture, and specific algorithms.

Following chapters aim to describe details of the implementation, and validation w.r.t. requirements.

Chapter 2

Analysis

This chapter describes in detail some fundamental assumptions about the system, as well as functional and non-functional requirements.

2.1 Use cases

2.1.1 Vehicle at an intersection

Brief Description

An Autonomous Vehicle (AV) is approaching, entering and leaving a non empty intersection. The use case begins with the AV approaching an intersection and ends with the AV having left the intersection in the desired direction.

Actors

- Autonomous Vehicles (AVs)
- Environment

Preconditions

- There is more than one AV at the intersection.
- AVs can be both active and inactive.
- The AV approaching the intersection is active.
- The AV knows the direction (exit) to take.

Scenarios

Main scenario

1. The AV is travelling along a road leading to an intersection.
2. The AV is approaching the intersection.
3. The AV stops at the entrance to the intersection.
4. The AV signals its intent to other AVs at the intersection.

5. The AV agrees with the other participants on how to solve the intersection.
6. The AV initiates the turn procedure towards the desired exit.
7. The AV is in the intersection travelling to an exit.
8. The AV leaves the intersection through an exit.
9. The AV signals that it has successfully left the intersection.
10. The AV continues to travel along its path.

Alternative scenario

1. 1, 2 The AV finds an active AV in front and gets in the queue.
2. 1, 2 The AV finds a faulty AV in front and overtakes him.
3. 6. 7 The AV finds a faulty AV and awaits for its removal.

Postconditions
At any point in its journey the AV can become inactive due to a mechanical failure or a software failure.

2.1.2 Vehicle mechanical failure

Brief Description
When an Autonomous Vehicle (AV) has a mechanical failure but the software is still working properly. The use case begins when the failure happens, and ends with the recovery of a normal situation.

Actors

- Autonomous Vehicle (AV)
- Environment

Preconditions

- There is at least one AV with a mechanical fault.
- There are possibly other AVs in the intercept network.
- Any other AV can fail at any moment.

Scenarios

Main scenario

1. The AV detects the mechanical fault.
2. The faulty AV communicates its state to the others in the network and to the environment.
3. Faulty AV awaits until it is removed.
4. Before leaving, communicate to the others (and the environment) the resolution.

Postconditions
The network does not contain the faulty AV anymore.

2.1.3 Vehicle software failure

Brief Description

When an Autonomous Vehicle (AV) has a software failure. This implies a mechanical failure, because we assume that a software failure stops the AV. The use case begins with the fault having happened, and ends with the recovery of a normal situation.

Actors

- Autonomous Vehicle (AV)
- Environment

Preconditions

- There is at least one AV with a software fault.
- There are possibly other AVs in the intersection network.
- Any other AV can fail at any moment.

Scenarios

Main scenario

1. One of the other AVs in the network detects the fault.
2. The AV which detects the fault communicates to others in the network which AV has just faulted.
3. Everyone awaits for the faulty AV to be removed.
4. The Environment communicates to the AVs in the network the occured removal.

Alternative scenario

1. On 1 can happen that there is no other AVs, in that case the faulted AV stays into the intersection until someone comes to the rescue.

Postconditions

The network does not contain the faulty AV anymore.

2.2 Assumptions

Some general considerations are here presented. First of all it is assumed a situation in which each autonomous vehicle knows its destination and the roads it is going to travel, since we can assume that each autonomous vehicle has a GPS device on board.

Moreover it is assumed, for sake of simplicity, that all vehicles have the same dimension — or better: that each vehicle can fit into a single position of the internal model used to represent the roads. Moreover, common physical quantities (like weight, speed, acceleration, etc. . .) are omitted. Instead, the

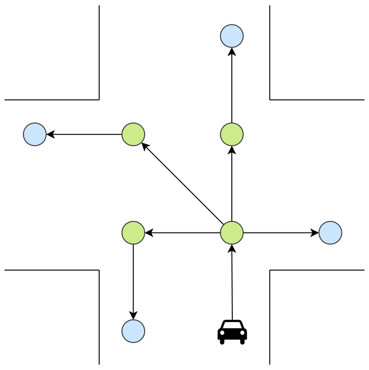

Figure 2.1: Example of predefined movements that a vehicle can follow in crossing an intersection.

autonomous vehicles move in unit steps governed by the internal model.

Further assumptions are:

- *Faults*: at any moment a failure can arise in vehicles — software or mechanical. A mechanical failure does not compromise the software abilities, but a software failure implies a mechanical failure. So we can assume that if the software fails, the autonomous vehicle stops and goes in "emergency mode". It is also assumed that if a crash happens, it must be managed by removing the faulty vehicle with a tow truck (or something similar).

- *Moves*: the path that a vehicle can take inside the interception is predefined and known a priori. From each position there is a unique path to each destination, and no one can modify or choose a different path once decided. A visual representation of this assumption can be found in Figure 2.1, in which green dots represent positions occupied while crossing, and light-blue dots are the destinations. Once a vehicle reaches its destination, it is considered "satisfied", and can not fail anymore, it can only move forward on its own way. Reversing is not allowed.

- *Message Delivery*: each vehicle can communicate with its immediate neighbors, i.e. the vehicle in front, the one behind it, and if it is at the verge of the crossroads, it can communicate with the others at the verge, and with crossing vehicles. Therefore it's assumed that there exist an upper bound time in message passing between two vehicles. This is justified by the fact that all connection are almost direct, without routers or broker that can cause bottle-necks or network congestion.

2.3 Functional requirements

The solution provides two main modules: *vehicle* and *environment*. Also a *vehicle generator* is provided, in order to test and validate the solution proposed.

2.3.1 Vehicle

Each vehicle has an internal state at any moment. Moreover it:

- knows its position and what happens around, through the environment (in a real context this is handled by sensors like GPS and proximity sensors): it can detect if there are other vehicles trying to cross, or if there is a vehicle in front or behind it.
- communicates with its neighbors by sharing: the direction it wants to travel, its internal state, the next position he is about to reach.
- if it is in the queue, it knows nothing else than its internal information and that it is taking part in a queue with someone before and possibly behind. It can communicate only with these two.
- if it is at the head of the queue, it can connect with other vehicles (that are at the head of other queues) that are crossing the intersection.

So, main functions offered by this module are:

- `startup()` to start the vehicle.
- `stop()` to stop the vehicle.
- `cause_mechanical_failure()` and `cause_software_failure()`, to cause failures; these are useful to test simulating failures.
- `send_message(To,Msg)` to communicate with other vehicles.
- `move()` to move one step forward.
- `check_next_position()` to check if the position it is willing to move is free.

2.3.2 Environment

The environment represent the vehicle ability to use its sensors. It knows the intersection shape and dimension; it knows vehicles approaching the crossroads and the ones in the queues; it provides vehicles with all the information they need to safely circulate within the environment.

So, main functions offered by this module are:

- `is_position_free(Pos)` to simulate proximity sensors; check if a specific position is free.

- `update_position(Vehicle,OldPos,NewPos)` to update its internal state moving the vehicle from the old position into the new one.
- `get_route(StartPos,DestPos)` to simulate the GPS; a vehicle that is going to start its journey ask the environment which is the route it has to travel.
- `get_participants()` to simulate proximity sensors, cameras, and antennas; to ask the identity of vehicles at the verge of the intersection.

2.4 Non functional requirements

The system does not handle byzantine processes nor cybersecurity issues.

- *Safeness*: there can not be more than a vehicle in the same position at the same time;
- *Liveness*: if a vehicle approaches the intersection and is waiting to cross it, eventually it will cross it;
- *Fairness*: if a vehicle approaches the intersection and is waiting to cross it, there exists a bound to the waiting time;
- *Fault Tolerant*: the system is tolerant to mechanical and software failures;
- *Distributed*: there is no central server, vehicles have to coordinate each other.

Chapter 3

Project

This chapter is devoted to the description of the general architecture, and specific algorithms.

3.1 Logical architecture

The solution relies on two main modules, and a secondary one for tests.

- *Vehicle*: it is represented as a finite state automaton. A graphical representation can be seen in Figure 3.1. Each state represents a situation in which the vehicle can find itself. A vehicle is in the *Init* state when it has just approached the intersection. It is in the *Ready* state when it is at the verge and has identified the participants. It is in *Election* when it is performing the election algorithm to choose who will be the one to cross. *Crossing* when it is crossing the interception, and *Terminate* when it has finished the crossing. When a mechanical failure happens, it notifies the event to its neighbors and then terminate; so a crash state is not needed.

- *Environment*: it represents the physical environment accessible through autonomous vehicle sensors.

- *Vehicle Generator*: it is a module aimed to (pseudo-)randomly generate vehicles that approach the intersection, and it is used to test and validate the solution proposed.

3.2 Protocols and algorithms

This section describes the algorithm to solve the core part of the project. Moreover, some peculiar situation sequence diagrams are presented.

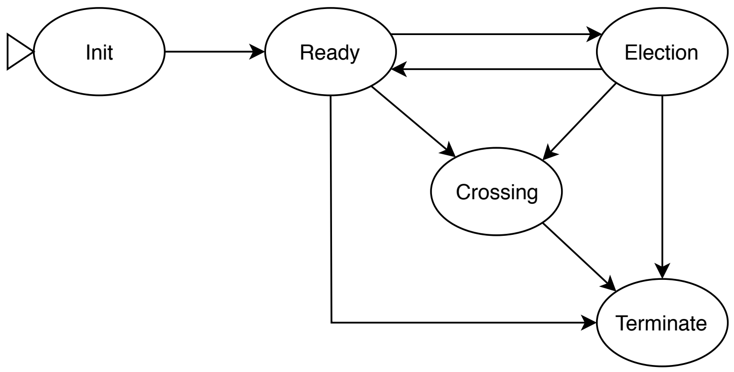

Figure 3.1: Vehicle as a finite state automaton.

3.2.1 Intersection Crossing Algorithm

Vehicles queue up along the roads leading to an intersection awaiting for their turn to cross. The first vehicle of a queue is the vehicle at the verge of entering the intersection and we will call such vehicle a *participant* of the intersection crossing algorithm.

Note that this algorithm applies to generic intersections, just by adapting the representing graph. Moreover, it trivially guarantees the *fairness* and *liveness* requirements (thus it is also *deadlock-free*), thanks to its round robin characteristic.

The algorithm maintains the following invariant through execution:

- one working (i.e. without faults) vehicle per time can be crossing the intersection
- only the leader vehicle can cross the intersection

Algorithm description:

1. The participants (first vehicles in their respective queues) are willing to cross the intersection;
2. They start a slightly modified version [1] of the Bully Algorithm to elect a leader: it terminates with the leader L, and the next leader L', which is the first vehicle after him in a clockwise manner;
3. L begins to cross the intersection;

[1] In the canonical version of the Bully Algorithm, every participant is supposed to know other IDs a priori. The version used here does not rely on this assumption. Still, it guarantees the same outcome, even w.r.t. non functional requirements (safety and liveness).

4. After L has successfully crossed the intersection, the participants are informed;
5. L' identifies the next leader L'' by choosing the first vehicle after him in a clockwise manner;
6. The lead passes to L' informing every participant that it's the new leader;
7. The algorithm repeats from 3, where $L = L'$ and $L' = L''$.

Additional details:

- After the leader election, every participant knows who is the current leader, but only the leader knows who is the next one. In this way if the leader dies, it is necessary to perform a new election;
- Once the leader starts crossing, the vehicle behind it (if any) becomes a new participant and asks to join the algorithm. Only the leader answers, providing its identity.
- If a vehicle approaches an empty intersection it automatically elect itself as leader and starts the crossing.

A leader election starts only when new participants do not receive an answer when asking to join the algorithm; or when, after the leader is gone (after completing the crossing, or due to a failure), remaining participants does not receive the message from the candidate, communicating it is the new leader.

3.2.1.1 Managing abnormal cases

All participants monitor only the leader, if it fails, everyone is expecting a message from the candidate, telling it is the new leader. If this not happens, a new election is started. Moreover:

- If the leader fails, the vehicle in clockwise order after him becomes the new leader — identifying also the next candidate leader — and the algorithm restarts from 3.
- If the leader fails while crossing, a timeout T is needed before it is removed from the intersection. Meanwhile a new election can start, and the new leader can start crossing, but it has to stop if the position in front of him is still occupied from the previous faulty leader, until the position has been freed.
- Some peculiar situations can happen for example when a new vehicle arrives at the interception while the leader is gone and it is passing the leading to the candidate. It the new vehicle arrives at an unfortunate moment, it could receive no answers when asking who is the leader, and so starting a new election. The risk is that the candidate can become leader, and after a while someone else can win the election, stealing its

role of leader. This can lead to disastrous scenarios, in which more than a vehicle start crossing. To avoid this, it is *necessary* that a candidate receiving the role from the previous leader, in case of a sudden election, it turns out to be the winner. So, no one can steal the leading from anyone else. This is guaranteed with an appropriate implementation, described in the next chapter.

3.2.2 Sequence Diagrams

This section presents some sequence diagrams of relevant case situations.

3.2.2.1 Normal execution of Intersection Crossing Algorithm

The diagram in Figure 3.2 represent the normal execution of the algorithm described above. It starts at the end of the Bully Algorithm, with the AV_4 communicating it is the leader. It is assumed the queues are enumerated from 1 to 4 in clockwise order.

The leader crosses the intersection by asking the environment if the position in front of it is free, and continues to cross until it reaches its destination.

It is important to note that when the leader starts crossing, AV_5 — which is the one following AV_4 in the queue — moves forward. Then AV_5 is at the verge of the intersection and can communicate with the other participants in order to compete for the crossing.

3.2.3 Dealing with AV failures

When the leader dies because of a software failure, it can not promote the candidate. So a new election is needed. The situation is represented by the sequence diagram in Figure 3.3

The last situation in Figure 3.4 represents a failure of the candidate leader. When the leader finish the crossing, everyone wait a timeout T for a message from the candidate. Since it is not alive, the timeout runs out, and a new election is performed.

3.3 Physical architecture and deployment

The architecture is quite simple, a graphical representation is displayed in Figure 3.5. Each Autonomous Vehicle (AV) is a physical node. The Environment represents the world sensed by AVs, so it is on a separate node, together with the vehicle generator. AVs can ask the environment for information, and they can communicate with their neighbors [2].

[2]For a definition of "neighbors", see in Chapter 2, in assumptions section. Where the message delivery assumption is described .

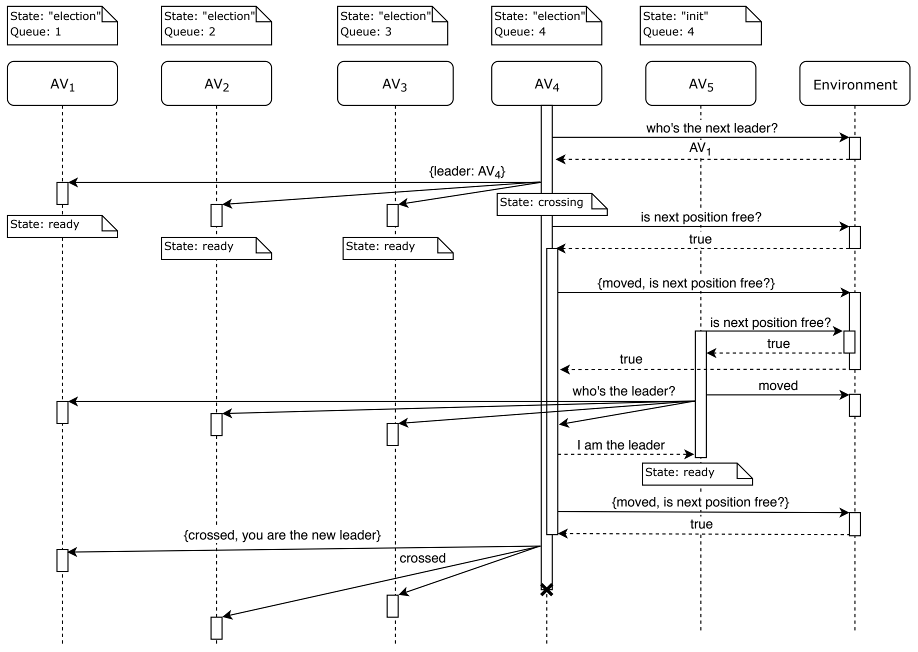

Figure 3.2: Sequence diagram on a normal execution of the Intersection Crossing Algorithm described above.

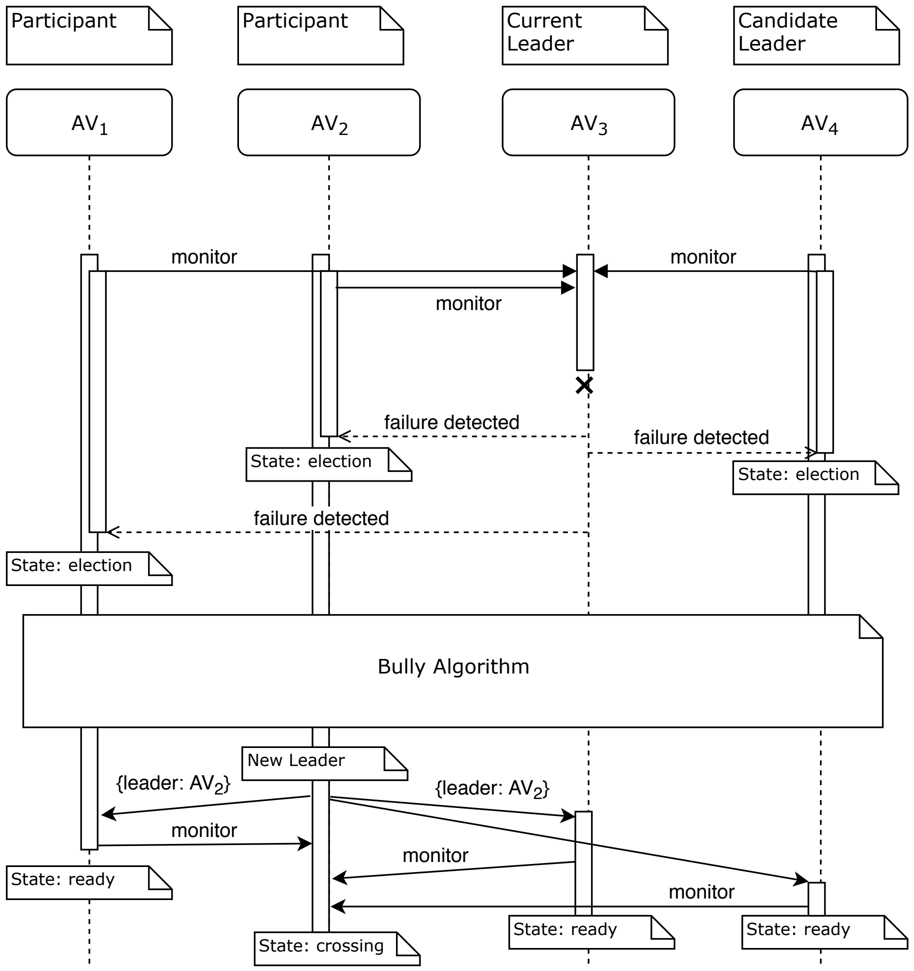

Figure 3.3: Sequence diagram representing a generic leader failure.

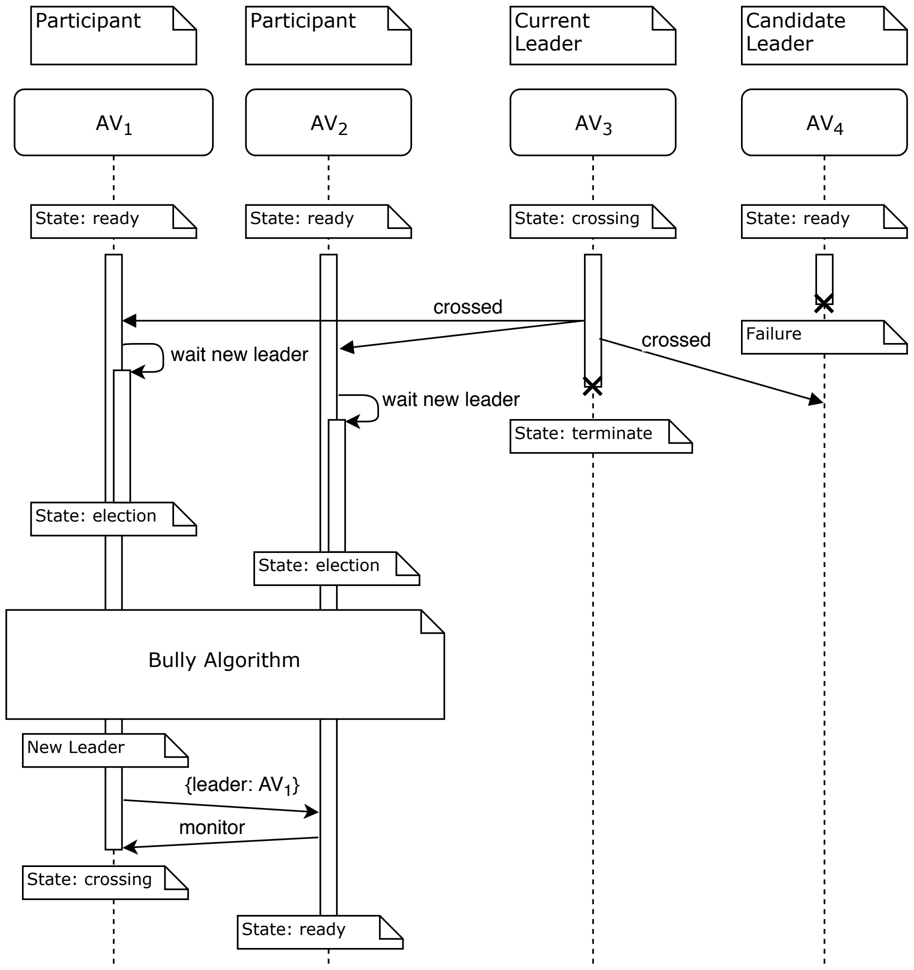

Figure 3.4: Sequence diagram of a candidate leader failure.

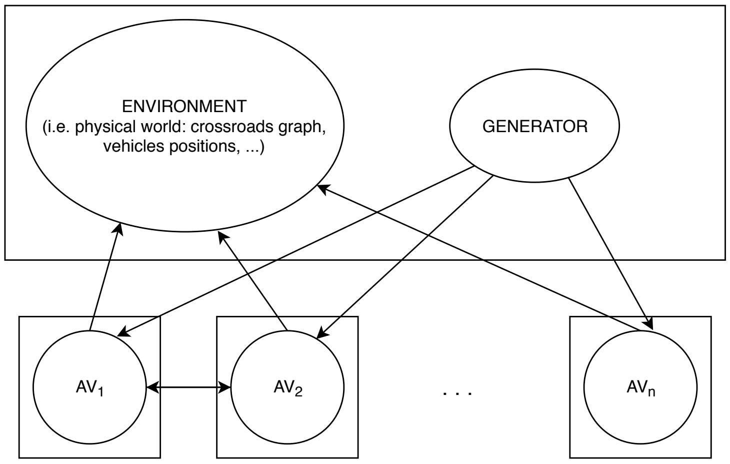
ENVIRONMENT
(i.e. physical world: crossroads graph, vehicles positions, ...)
GENERATOR
AV$_1$
AV$_2$
. . .
AV$_n$

Figure 3.5: Physical architecture: Environment, Generator, and Autonomous Vehicles (AV).

Chapter 4

Implementation

The project is developed in Erlang.

4.1 Getting Started

4.1.1 Project Structure

4.1.2 Hands On

4.2 Development Details

4.2.1 Intersection Coordination

4.2.2 Vehicle Generator

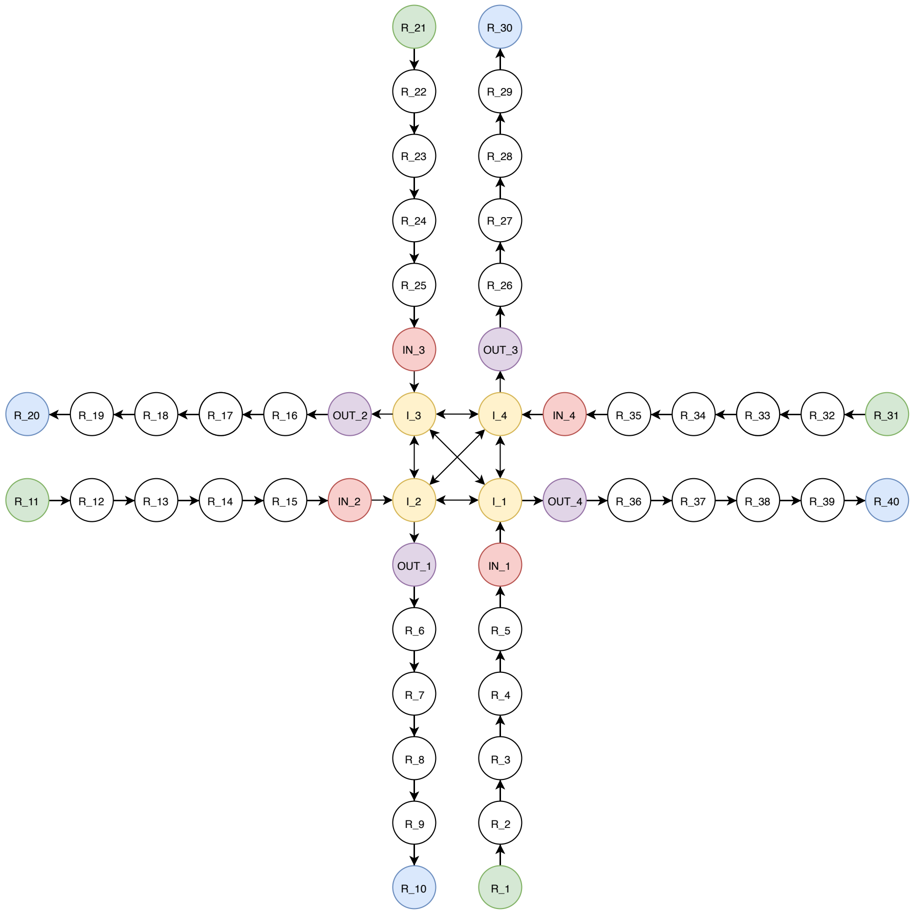

Figure 4.1: Graphical representation of the intersection graph contained in the configuration file, and used to test the project. Nodes with name R_x represents roads. Details: in green start nodes (R_1, R_11, R_21, R_31), in blue destination node (R_10, R_20, R_30, R_40), in yellow intersection internal nodes (I_x), in red intersection entrance nodes (IN_x), in purple intersection exit nodes (OUT_x).

Chapter 5

Validation

The validation part consisted in three main phases. First of all the project was tested with few vehicles generated singularly into different shells, with Erlang nodes created on local machine. After the initial validation, simulating peculiar situation manually (software and mechanical failures, vehicles arrival at specific times, ...), the software has undergone numerous tests with the help of the vehicle generator described in the previous chapter, even this with Erlang nodes on local machine. Finally, exploiting Docker facilities, we tested the project even in distributed containers on different hosts in a virtual network.

The project has been tested on Fedora 30 with Erlang/OTP 21, and on Ubuntu 18.04 LTS with Erlang/OTP 22.

After careful planning and developing, and after all these tests phases, we are confident of providing a reliable and sound solution. In next sections is described how to reproduce the tests. All the scripts provided have been tested only on cited above operative systems, same results are not guaranteed in Mac OS or Windows.

To perform tests is necessary to have in mind the graph of the interception presented in Figure 4.1. Moreover, for the test with the generator and with docker, the scripts will not exit at the end of the simulation, because we can't understand when all vehicles have finished, but you can easily check that if for about a minute nothing is printed, then the simulation is over, and you can kill the script with `ctrl+C` and then `(a)bort`.

5.1 Simple Tests

To use the code proposed here is necessary to execute `make all`. Some scripts are provided in the direcotry `test/`. Note that is is important to execute the script provided from the project root.

First of all it is necessary to start the environment. To do it open a shell and run:

```
./test/start_environment.sh
```

You should se now an Erlang shell with the message "Env started!".

Now it is time to produce vehicles: pretty easy, open another shell and run:

```
./test/start_vehicle.sh 1 R_1 R_30
```

Where, the first parameter is the index of the vehicle to generate — note that if you insert x the erlang node will be `v`x`@hostname`. The second and the third parameters are the start position, and the destination position, chosen from the graph in Figure 4.1. One can also start a vehicle from other positions chosen from R_x or IN_x, to test different situations.

You will see that in every vehicle shell is printed what it is doing, and more information can be found in the vehicle logs inside the `log/` directory.

5.2 Vehicle Generator

To use the code proposed here is necessary to execute `make all`. To start the vehicle generator can be used the script inside the `test/` directory, as follows:

```
./test/start_generator.sh <vehicle number> <fail ratio> \
[relaive sw fail ratio] [max fail timeout (ms)]
```

for example you can try:

```
./test/start_generator.sh 10 0.2 0.5 20000
```

Where:

- `<vehicle number>` is mandatory and represent the number of vehicles that will be generated.
- `<fail ratio>` is mandatory and represent the percentage (expressed with a number $0 \leq x \leq 1$) of failures that will be generated.
- `[relaive sw fail ratio]` is optional and represent the ratio of software failures caused by the generator. (e.g. if set to 0.5, half faults will be software, and the other half will be mechanical).
- `[max fail timeout (ms)]` represent the maximum time in milliseconds within which the fault will be caused, considering the moment when the vehicle starts. (e.g. if set to 20000, the vehicle can fail after a maximum of 20s from its start).

To stop the execution press `ctrl+C` and then `(a)bort`. The information here are printed all in the same shell, so it is less clear what is happening, but the logs for each vehicle can be found in the `log/` directory.

5.3 Docker

To use the code proposed here is necessary to execute `make docker`. Note that with the tests proposed here is required to have docker installed and configured properly. Some containers, and a network, will be created. In `docker/` directory are provided some script to easily run the tests and also to stop and clean the containers and network created.

The command to run is:

```
./docker/start_docker.sh <vehicle number> <fail ratio> \
[relaive sw fail ratio] [max fail timeout (ms)]
```

for example you can try:

```
./docker/start_docker.sh 10 0.2 0.5 20000
```

Where the parameters are the same of the vehicle generator described in the previous section. Note that in docker you will se only messages printed from the generator. To see what happens to the vehicles you can look at the `docker_log/` directory, where all vehicles log are generated. Further, can be necessary to provide root permission to delete `docker_log/` directory, because the files are generated by root user inside the containers, so the proprietary user is still root. After stopping the simulation, everything can be cleaned with:

```
./docker/clean_docker.sh
```

Chapter 6

Conclusions

The proposed solution focuses on a very simplified version of what a real solution could be. In particular the aim was to produce a core solution which satisfies the requirements but that also does not contain more than necessary assumptions, and in particular that does not rely on non realistic ideas that can lead future development to a blind spot.

6.1 Partial implementation and future work

Some advancement that can be pursued starting from our solution are provided here. Some of them are easier than other, but due to lack of time we did not managed to deepen them:

- Try different interception graphs;
- Centralize some configuration parameters in such a way the solution can be tuned without recompiling;
- Exploit code change feature offered by Erlang;
- Develop a graphical interface of the environment to better understand how vehicles are moving;
- Add vehicles with priorities, e.g. emergency vehicles;
- Dynamically modify paths inside the crossroad if some position becomes unavailable;
- Let more than one vehicle per time cross the intersection, maybe with an interface to a planner in order to find the best route (exploiting epistemic planning could be an interesting option).