The Bottleneck Problem

Distributed Systems - Project 2019

Edoardo Lenzi, Talissa Dreossi
DMIF, University of Udine, Italy

Version 0.1, July 27, 2019

Abstract

The aim of this project is the analysis, the implementation and testing of a distributed solution for the **bottleneck problem** (also know as *The Monkeys Problem*, in operative systems theory).

Basically the problem consists in a two-way road with a **bridge** (bottleneck) in the middle, the bridge has a certain **maximum capacity** and can be crossed in only **one direction at a time**.

The bridge is **very risky** cause it is located in a remote place where there is nothing that can prevent **car crashes/congested traffic** but cars.

Note that **cars** here are **autonomous systems** without any human driver inside and can send messages with others adjacent cars with some wireless technologies (ie. bluetooth, wifi, ...) in order to solve the situation.

The projet requires the implementation of a **simulator/business logic** for the environment setup and a **web server** that will expose the simulation state with some API for any further **UI application**.

Contents

Chapter 1

Introduction

In this chapter we are going to describe accurately the problem and provide a possible solution (from the point of view of a Distributed Systems designer).

1.1 The Problem

The bridge has a **maximum capacity** $c \geq 1$ and a certain length l.

Cars can send messages to adjacent cars (to the car in front and to the rear one); cars can also have the ability to speak with more than two other cars depending on the **power of their transmission medium** $p \geq 1$.

The initialization of a communication between two cars can be done unsing an **environment process** that returns the required references needed to start the message exchange.

So basically a car can only send messages to the p cars in front and to the p cars behind. We assume that **the bridge length doesn't constitute an impediment** for the communications (so even if the bridge was very long, however, the cars can send messages as if they were close).

By default every car has the same dimensions (but can be setted to an arbitrary **size**) expressed in a custom length scale called **block** and **every measure is an integer** (l, c, p, etc.).

We assume that cars can have a failure at any moment and there are only three types of failures:

- **mechanical failure** (the car cannot move but can send messages to the other cars)
- **link failure** (the car can move but cannot send any message)
- **system failure** (the car cannot move and cannot send messages)

We assume that a link failure is equivalent to a system failure cause the car cannot take any decision without the agreement of the others.

In case of engine failure or system failure the car or another helping car must call a tow truck in order to remove the broken car (in this case we have to wait an **elimination time** e).

In case of failure the car behind has to wait until the tow truck removes the broken car.

We assume that a car cannot be malicius (must follow the algorithm) but the communication channel isn't secure so it is exposed to a **MIM** (man in the middle) attack (drop messages, edit messages, message injection, ...).

Finally we have to consider that in a distributed system a global up-to-date state or a global timing cannot exist; this implies that each car can have a partial and **inconsistent view** of the global environment and a **time drift** from the global time (for this reason the system cannot guarantee the FIFO ordering at all).

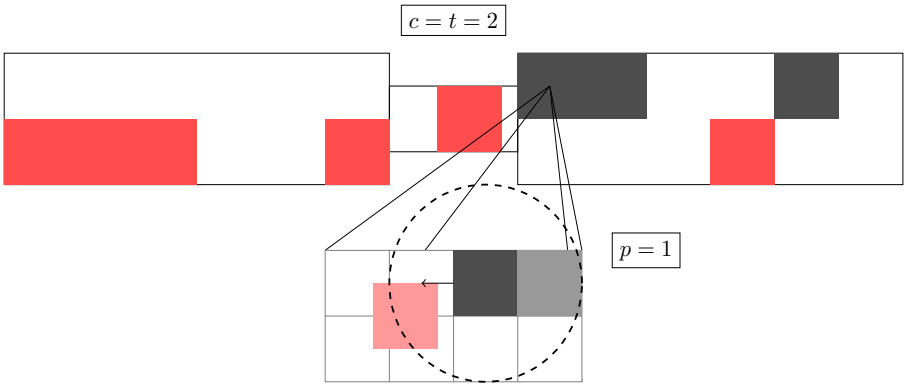

Figure 1.1: Example of a possible situation

1.2 Requirements

The main requirements to be met are:

- **Fairness**
- **Fault tolerance**
- **Without starvation**
- **Without deadlock**

Chapter 2

Analysis

In this section we will discuss the requirements which our system must satisfy discriminating them between functional and non functional.

2.1 Functional requirements

The system will be able to:

- **Coordinate the traffic** among the bridge following the rule that it can be crossed only in one way per turn. So it never happens that two cars crossing the bridge at the same time, in opposite ways.
- **Avoid** any kind of **accident** (also between enqueued cars).
- The maximum amount of cars that can cross the bridge at the same time, in the same direction (a **block**), is equivalent to its capacity c.
- The decision of who can cross the bridge will be taken by the cars themselves through a communication (**agreement**) that they instaurate without any human help.
- If a car is broken (it can't communicate and/or move any more) it will be removed by a **tow truck**.
- provide a **UI** in order to monitoring the **simulation** and manage **settings**.
- generate new cars.

2.2 Non functional requirements

We can distinguish the non functional requirements in the following macro areas:

- **Safety**: the system guarantees a correct use of the bridge, avoiding any sort of accident between two or more cars. In fact it ensures that the bridge can be crossed only by cars that are moving in the same direction. Notice that we are not excluding the possibility of a car to break down (the cause of crashing is something that doesn't belong to the task of the system).
- **Security**: (bonus requirement, not necessary) the system avoids man in the middle attacks using HTTPS.
- **Strength**: the system can work also in particular cases such as when a car breaks down and needs to be removed. This situation does not lead to an accident because the other cars can find out if a car is broken.
- **Scalability**: cars, clients and web services. The project requires that cars can scale arbitrarily and there is a single web service; anyway our architecture can be easily modified in order to have an arbitrary number of web services (cause we use Mnesia [**?**], a distributed database and cars are bounded to a particular web service dynamically).
- **Consistency**: due to the relative point of view of each process this requirement is not guarantee at all.
- **Starvation**: if a car requires to cross the bridge, its request will be satisfied eventually; it will never happen that a car waits for its turn forever.
- **Fairness**: if a car A has arrived before car B, then car A will be the first one who cross the bridge (**FIFO ordering**). Such as consistency there are scenarios in which, due to concurrency, the simulation generates two or more cars in the same instant and the requests arrive at the web service in random order (this isn't a huge issue cause in the reality two cars cannot be spawned in the scene in the same instant).

Chapter 3

Project

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

3.1 Logical architecture

The logical architecture is composed by the following component:

- **Web Service** a node that provides environment primitives to the car processes (list of adjacent cars) and, at the same time, can send queries to the DB component and exposes a **RESTful API** [**?**] for the clients.
- **Client** a simple web page used to monitoring the simulation state and manage simulation settings; uses the web service APIs to render a graphic interface that must be constantly synchronized (polling).
- **Car** a process that is the abstraction/simulation of a real car;
  - ▷ it can become a leader in order to schedule the crossing order for the next turn
  - ▷ it must check the health state and position of the other adjacent cars (calls the tow truck in case of failure)
  - ▷ it can see environment details calling a web service
  - ▷ it has to synchronize its local timing using Berkeley algorithm
- **Distributed DBMS**, an instance of Mnesia DBMS paired with the web service instance (in a real distributed scenery there will be an instance of Mnesia foreach web service in order increase redundancy and robustness).
- **Docker Container** wraps one or more car and/or web service instances and is designed to be interconnected with all others containers (using a **docker network**).

3.2 Protocols and algorithms

Following the *divide et impera* philosophy now we are going to split the problem into some subproblems and solve them. We also provide a simplified sequence of UML (sequence) diagrams in order to describe the workflow/communication patterns.

Starvation/deadlock and order method

Ideally a car that reaches first the queue must pass before other incoming cars (**FIFO**).

We have to **avoid starvation** (ie. when a car waits infinite time cause the opposite queue has infinite length and it never has the priority) and **deadlock** (cars aren't able to reach the agreement).

Solution

In order to avoid **starvation** in a first stage cars synchronize themselves (for the synchronization process we follow the **Berkeley algorithm**) in terms of local timinig and in a second stage there will be a **leader election** and the leader decides the crossing order.

Using this method we also avoid the possibility of a **deadlock** as long as the leader is running.

In a certain instant the elected leader is the car that has reached the bridge and present a lower arrival time wrt. the car on the opposite side (if present); once the current leader cross the bridge there is a new election.

3.3 Synchronization problem

Every car has a **local timing** that can **drift** out from the global timing.

Solution

So we need to synchronize the incoming cars using the **Berkeley algorithm**.

A new incoming car calls the *environment:get_adjacent_cars(p)* method in order to get the name of the adjacent cars. The new car c sends a message to the nearest car n in order to get the current time (we assume that n was already synchronized).

With this information c is concious of the RTT and its local drift from the global time; so basically the global timing is the timing of the first leader.

3.4 Agreement

The current leader identifies the block of cars that will cross the bridge and notifies this decision.

Solution

Before crossing the bridge, the leader A propagates a message to the first n cars behind him (where n is the capacity of the bridge). The message that is received tells them to cross the bridge if their arrival times are lower than the arrival time of B, the car on the opposite side (if present). It could be possible that only a part of them will pass the bridge while some other have to wait for their turn.

Each car, who receives the message, checks if its arrival time is less then B's: if so then the car can cross the bridge too, otherwise waits for its turn.

The leader will always be elected by himself. Suppose that the car A, which is not a leader and hasn't the permission to cross, arrives on the bridge side.

At that point A checks if there are any other car in front of her:

- if there is a car whose arrival time is greater then A's then A is the new leader;
- if there isn't any car, A is the new leader.

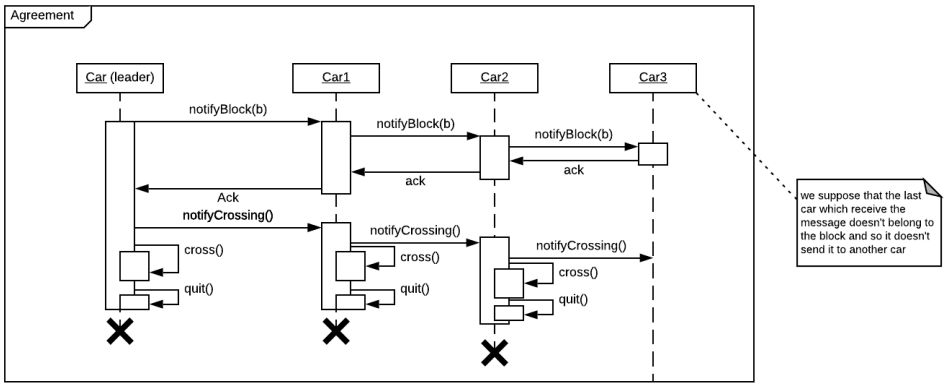

Figure 3.1: Agreement

3.5 Failures

In case of failure, of both engine and communication system, another car has to call a **tow truck** for help. If only the engine has crashed then the car itself can call the **tow truck**.

Solution

Each car recursively checks if other reachable cars are safe; if a check message hasn't any response within a certain **timeout** (depends on the max_RTT al-

lowed) it is assumed that the receiver has a failure (and so the tow truck will be called).

Once the car was removed the tow truck notifies the adjacent cars; anyway if the car is able to send messages notifies autonomously its removal to the adjacent cars.

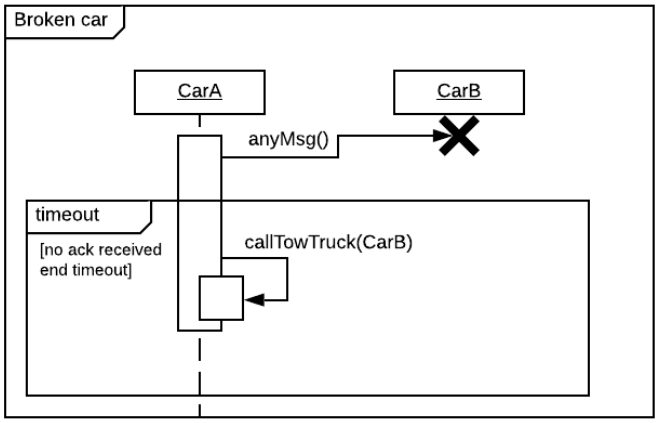

Figure 3.2: Car crash type 1 (engine only)

3.6 MIM attack

Bonus requirement, not necessary

The system must withstand a MIM attack.

Solution

Each message will be encrypted (HTTPS).

3.7 Scalability

The system must **scale wrt. the load** (the number of cars and clients can scale arbitrarily).

Solution

Erlang allows huge numbers of processes on the same machine so we will test everything locally.

Bonus requirement, not necessary

Anyway if we assume that the available machines are defined in the initialization phase, it will be easy to rise up instances of cars and/or web services on different machines.

Can be also designed an embedded load balancer in order to avoid the web services overload (considering the number of calls).

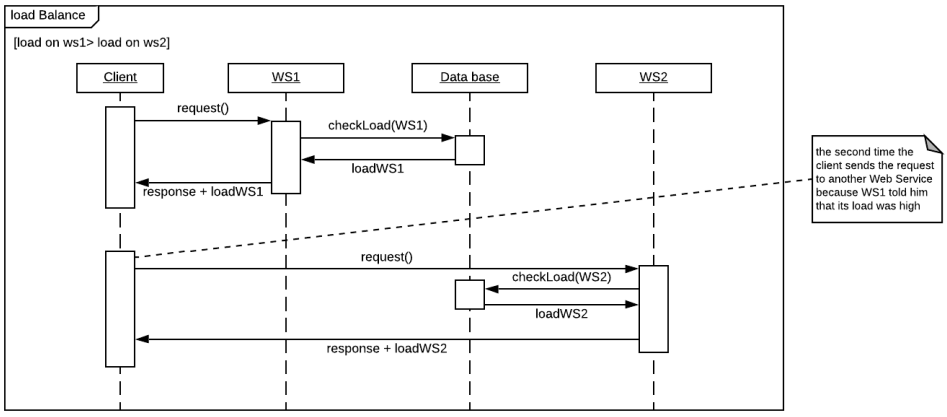

Figure 3.3: Load balance

3.8 SPOF

The system must be resilient wrt. failures (even multiple failures); so basically the architecture must be deeply **distributed** and **decentralized** (avoid any SPOF).

Solution

The scalability requirement imposes that every macro-component must be scalable so the architecture provides a **peer-to-peer layer** of cars.

The only SPOF in this context can be the *environment* component (which is embedded into the web services) that allows for a new spawned car to start a message exchange with the adjacent cars.

This issue can be solved with a peer to peer layer of web services and some istances of a **distributed DBMS** like Mnesia [**?**] (but isn't a project requirement).

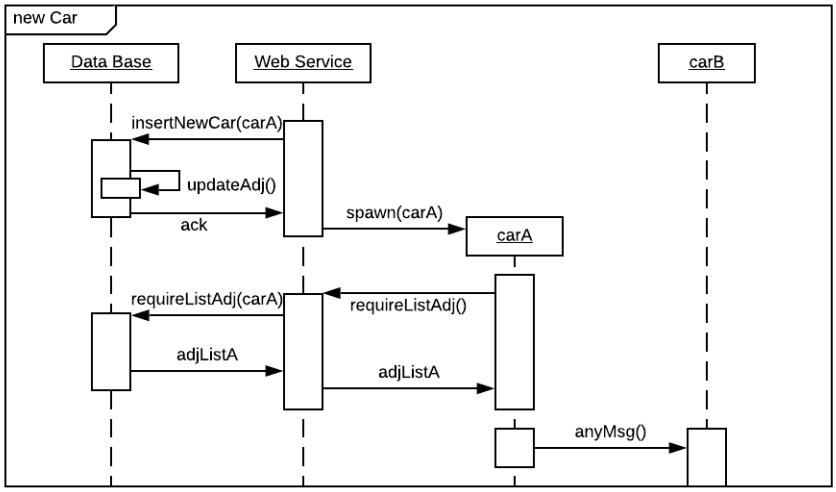

Figure 3.4: Creation of a new car

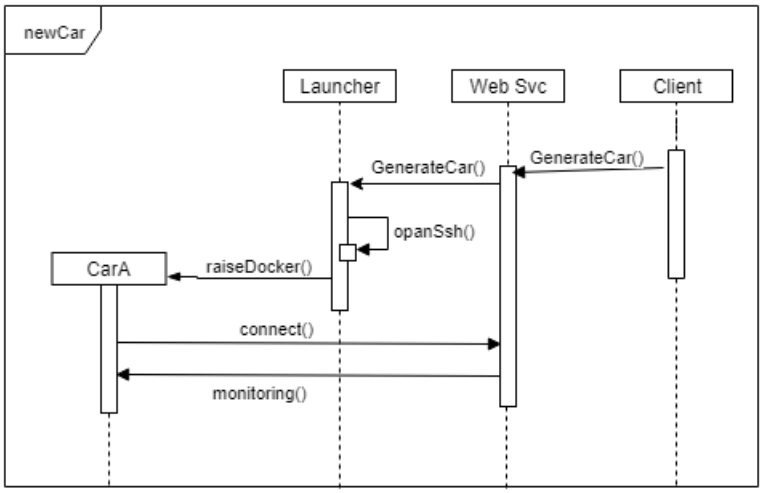

Figure 3.5: Client request of new car

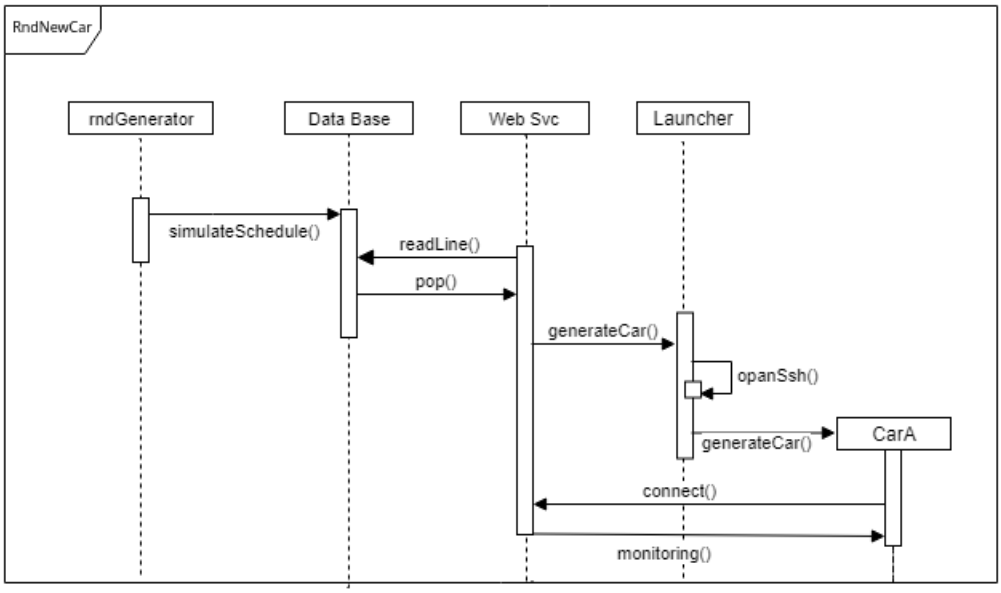

Figure 3.6: Random creation of new cars

3.9 Algorithms

Concurrency

Due to the asynchronicity, concurrency and time drifting is really hard to keep consistent the state of the cars layer with the simulation on the client layer. One way to keep an exact consistency can be to structure the algorithm on turns and only at the end of the turn, after the reaching of a global agreement, launch a transactional update on the database layer. This way is for sure really safe but implies the implementation of some logic and a huge message exchange that overloads the network.

A simpler and less expensive idea (greedy) can be to leave a temporal inconsistency between layers state but avoid anomalous behaviours on the simulation side.

Denote with $A \to B$ the partial order relation[1] between two cars A and B that are running from left to right and A **is immediately behind** B.

So assume that in a certain instant t_0 (measured from an external and omniscient point of view) A is in position p_{A0} and B in p_{B0} and the database

[1] To be precise $\to$ isn't a partial order relation cause it isn't reflexive (a car cannot be behind itself) but only transitive and antisymmetric.

layer state is consistent with the car layer state:

$$state_{DB} = state_{Car} \tag{3.1}$$

Now before the next instant t_1 car A moves forward of one block and car B do the same after a while; we said after a while cause, as described by Lamport, time is a continuous measure but clocks have a finite precision so ticks are discretizations of the time concept.

For this reason if we consider a turn/tick of one second between t_0 and t_1 we realize that in this interval some actions can be made so consider the case in which A and B before t_1 have send two messages to the DB layer $A : p_{A1}$ and $B : p_{B1}$.

Now from the point of view of the database layer we cannot made any assumption about the incoming messages cause one or both of them can be lost and/or the arrival order can be different from the sending order.

The more the third layer cannot have a consistent vision and, for example, can render the movement of B before the transition of A causing a visual crash (never happened).

I) Anomalous behaviours like that can easily be avoided ensuring that for each car must hold an order relation $\rightarrow$ and both movements and updates must be send respecting this relation.

In this way it is possible that the simulation renders A in position p_{A1} and B in position p_{B0} but this isn't a huge issue cause doesn't led to a crash so can be overlooked.

Berkeley

We have used the Berkeley algorithm in order to synchronize new spawned car processes.

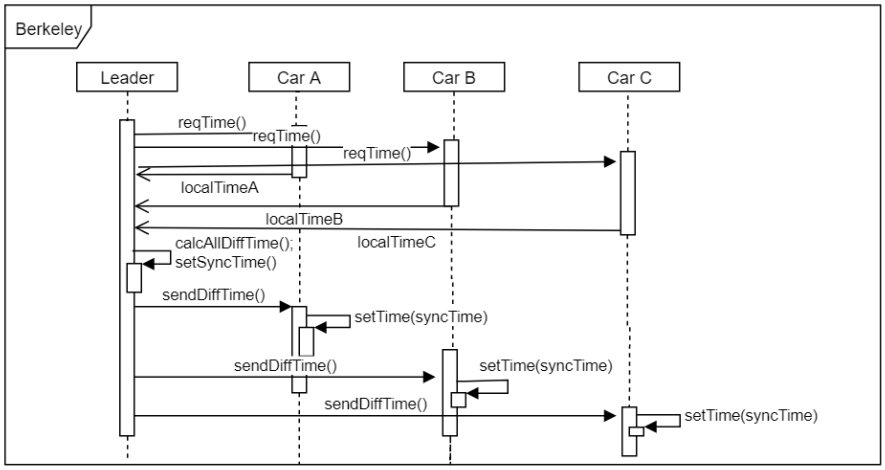

Figure 3.7: Berkeley algorithm

3.10 Physical architecture and deployment

The deployment and start up consists in the following steps:

1. Get the SSH credentials and IP of each computer involved in the simulation and fill the **settings files** (*environment.json*)
2. Launch the *make run* command
3. With a **web browser** any client can connect to the web service API and manage/monitoring the **simulation state**

We adopt **Git** for code versioning with a **feature-oriented** branching strategy with a **PR** foreach merge.

The code required for the simulation is taken directly from the project **repository** hosted on **GitHub** [**?**].

3.11 Development plan

The proposed architecture is a **P2P** architecture flanked by a **client-server** architecture (**three-tier architecture**: client, web service and DBMS).

We will follow the **Agile** philosophy, splitting the development process in sprints of one week and adopting the **TDD** [**?**] methodology.

Chapter 4

Implementation

In this chapter we will explain how the system was implemented, so we will describe technologies/languages, data structures, patterns and best practices used.

4.1 Assumptions and observations

Starting from the instantiation of a new car process we will describe the main aspects of a **car life-cycle**. We have to define new variables and notations; first of all we define a first layer cars as the set of car processes, this layer sends periodically updates to a second layer of web service and database. Finally a third layer of clients make polling on the second layer.

As we will show, we model a car as a **finite state machine** with the following main states: sync, normal, leader, dead.

One of the main problems of the project task is that there are some non trivial aspects that do not emerge in the initial analysis but are of fundamental importance; this happens cause the analysis takes into consideration only the problem that is located on an higher level with respect to the implementation and the architecture proposed.

First of all from the problem analysis is clear that the bridge is a shared resource so the problem can be solved with the solution exposed by Lamport in its article [**?**] about total ordering and synchronization.

But this isn't the best choice cause total ordering requires a huge exchange of messages and, for this problem, is more than necessary; we decide to implement a custom algorithm based on the idea of locks.

Also leader election/distribute agreement algorithms are superfluous cause bridge access isn't contended by every car in a certain instant but, at most, by two concurrent cars on the opposite side; so a simple message exchange will be enough for the leader election.

4.2 Tools

Language

The best choice between the languages exposed during the course seems to be **Erlang** [**?**]; in fact **Erlang OTP** allows to handle concurrency and distributed programming moreover provides some **behaviours** really useful for this project:

- **gen_statem behaviour** allows to create a **finite state machine** based on events (that fits perfectly with the nature of the problem);
- **gen_server behaviour** allows to create a **web server**; anyway for the web service implementation we prefer to use a more rich and well documented **web framework** called **Cowboy** [**?**]

Web service

Thanks to Cowboy we were able to implement a web service that exposes a **RESTful API** for both cars and simulation clients; we use **JSON** [**?**] as standard data interchange format for any interaction with the web service APIs.

The main reason of this decision is that it is, nowadays, a standard de facto and also versatile/**independent from the language** used (we use a library called **Jiffy** [**?**] for serialization/deserialization of data).

Patterns and best practices

The web service architecture follows the idea on the base of the **MVC pattern**; views, models and controllers were splitted and located in different files.

The **front-end tier** is structured in views, stored in the same folder of the back-end code (for scope reasons). Each view can calls the server calling *http-client.js* functions that use **AJAX** [**?**] for HTTP calls.

The **back-end tier** follows also the **repository pattern** and **CRUD** [**?**] logic for what concerns the interaction with Mnesia. We decide to use the repository pattern in order to feel free, at any time, to change DBMS/connect with more than one DB without rewrite business logic code.

Mnesia is a no-SQL DBMS that basically works in a functional way (it is written in Erlang) with tuples so it **isn't necessary any kind of ORM** [**?**] in order to map query results with Erlang data structures (and also avoid some issues of SQL like "hard-coded" queries and SQL-injection [**?**]).

Moreover every query, in particular **commands** (non idempotent query defined in CQRS [**?**] pattern), is made with a **transaction** so concurrency doesn't constitue an issue for this architecture.

An HTTP call flow is really simple and clear; it starts from the **routing module** *web_service_app.erl* that forwards the request to the right **controller**. In

this passage we can also interpone some **middlewares** but the project doesn't require any.

Controllers are responsible for data unmarshalling and call **business logic routines**.

The entire logic of the web service is stored into the **service layer** that has the duty to interact with the database using the interfaces exponed by the repositories and returns to the controller the expected results. For this reason in order to **test** the business logic is enough to test the utility functions used and exponed by the services.

The module dependencies are: *cowboy*, *jiffy* and *rebar3_run* plugin; **dependency management** was delegated to **rebar3** [**?**] that uses internally HEX [**?**] package manger.

UI

UI is implemented using some famous front-end libraries collected with **npm** [**?**] package manager.

Client UI can be used in any context so must have a **responsive layout** in order to fit any screen size; we use **Bootstrap** [**?**] and JQuery [**?**] in order to create a fluid layout.

Moreover we though that the aesthetics of the simulation could be more appealing using **ThreeJS** [**?**] so we have injected in the main view a **3D canvas** where cars are rendered as blocks on a 2D street plane.

Car

Car is structured as a **finite state machine** with events, it has a **supervisor** that initializes the FSM and manages the interactions with the other cars and with the web service.

The initial idea was to exclude the supervisor and call another car directly sending a message; this idea fails cause *gen_statem* behaviour doesn't allow to send/receive messages such as a normal node.

So the behavior forces to **use the supervisor as an interface** with the FSM moreover we want to realize an asynchronous system for message exchange also capable to keep track of message RTT and other parameters.

The solution was the creation of a little **timer subprocess** that manages the message exchange; this way was applicable but also a little complex because a message between two cars requires 8 hops (4 hops for the request and 4 hops for the response) so we implement a **pseudo multi-layer encapsulation** in order to create a generic asynchronous communication infrastructure.

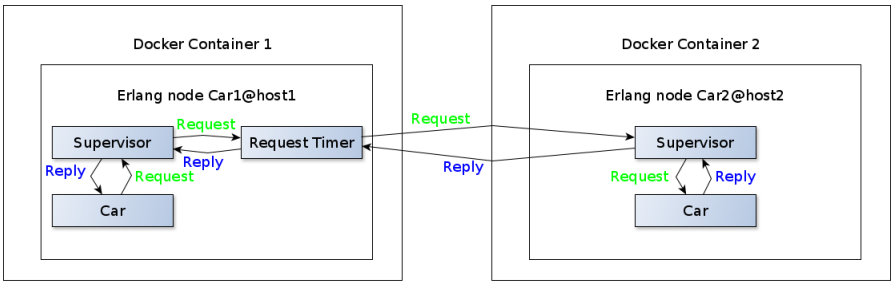

Figure 4.1: Message exchange between two cars

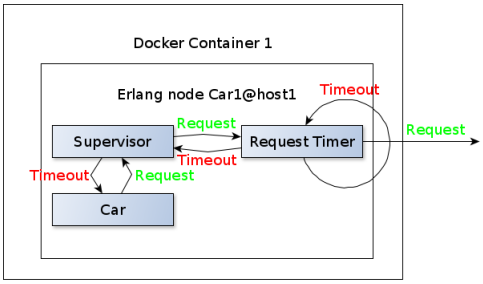

Figure 4.2: A message without any reply triggers a timeout event

4.3 Docker

At the end of the implementation the Erlang scripts were builded inside a docker image ready to be instantiated in a new container.

Cars aren't aware, at compile time, of the **parameters** that characterize them (size, power, global settings, web server IP, etc.) so we exploit the docker feature to pass arguments with the *run* call in order to **initialize the car**.

Now the web service is sandboxed in a Docker container so in order to be able to launch bash commands (ie. to raise a new car instance) outside its scope we set up an **SSH tunnel** [**?**] toward the target host.

4.4 Data structures

As described in Chaper 1, cars and bridge must be aware about their characteristics.

In order to do that we have decided to create an environment file (*environment.json*) that collects the default environment settings, flanked by the UI that allows to update some settings.

Environment

The environment defines different attributes. The most important are the following:

- *bridge_capacity*: it defines the maximun number of cars that can cross the bridge at the same time
- *bridge_length*: it defines the length of the bridge
- *tow_truck_time*: time necessary for the tow truck to arrive and remove the dead car
- *max_RTT*: it is the maximun RTT that could happen

These attributes will never be modified during the simulation.

Car

As for the environment, each car A is modelled exploiting several attributes. The most important are the following:

- *side*: if equal to 1 means that the car is on the right side, if equal to -1 it means that it is on left side
- *speed*: it defines the distance that the car goes through in one turn
  it has a maximun value specified in **environment.js**
  it is initialized with 0 to prevent crashing
- *crossing*: if true it means that the car is on the bridge
- $crash_type$: if equal to zero, the car won't crash, if equal to one the car will have an engine only crash and if equal to 2 the car will have both engine and system crash
- *adj*: it contains the lists of front and rear cars
- *state*: it defines the current state of the car
  it could be **init**, **sync**, **normal** or **dead**
- *position*: it indicates the distance between the car and the bridge or the car and the end of the bridge
  it is initialized as $pos(B) + 1 * side(B)$, where B is the first front car, if B is on the same side of A, or as $-pos(B)$ if B is on the other side
  when negative it means that the car is on the left side, otherwise on the right side
- *arrival_time*: it defines the time when the car has arrived in the queue

Adj

The *adj* is another data structure. It is composed by:

- $front\_cars$: it's a list of cars that contains all the car ahead that are rechable
  the first element is the next car
- $rear\_cars$: it's list of cars that contains all the car behind that are rechable
  the last element is the previous car

Each car knows also the *bridge_capacity* and *bridge_length* that are imposed by the environment and cannot change during the simulation.

We choose to model the car as a finite state machine because the car can switch between four main different states.

Init

Init isn't properly a state cause cannot receive external events, is used only for $Data$[1] initializations.

Sync

After FSM initialization the car reaches **sync state**, the main goals of this state are:

- Synchronize with the other cars
- Set a consistent arrival time with the queue state
- Set an initial position (safe, avoid spawn on the other cars)
- Send an update to the DB layer with the informations above

In order to perform a synchronization using Berkeley algorithm the new car needs to know a reference of the last car in the queue which is exposed by an end point; in order to avoid concurrency issues when a car calls the end point it becomes automatically the last car in the queue (with a transaction query).

This implies that any car is chained with the car in front of her and any other new car, from now on, cannot be interponed betweent them.

If everything goes in the right way, at the end of the synchronization the new car (A) is located near the car in front of her (B) and the arrival time of A must be higher then that of B.

[1] **Data**: we denote as $Data$ the state persistent variables

We have also to guarantee **I** property so a car in sync state can synchronize itself with the database layer only after the car in front of her. Define $A : check(B)$ a check message from A to B that asks for the current B state; the message reply contains every relevant information about B also its synchronization state, so is enough to wait until a check response state that B was just synchronized.

If a check call doesn't have any reply, a timeout event will be triggered after max_{RTT} ms. Once a car receives a timeout event calls a special process tow truck that has the duty of remove the broken cars.

Once a car is removed by a tow truck the garbage process returns to the caller an update event that is then propagated to every other cars. For this kind of events the order isn't relevant, so even if we receive two concurrent updates of near cars we haven't to straggle to ordering them.

Normal

Sync was a static state, the car declares its initial position but has $speed_A = 0$; normal is instead the state in which the car remains for the most of its life time and manages every car movement.

Assume that between two cars $A \rightarrow B$, in a certain instant t_0, there is a safety distance d_{AB0}; the car behind know this information because has sended a check message to the car in front of her and at instant t_1 receives a response.

Now A is free to move forward (because B can only move forward) of:

$$\begin{aligned} &min\{\ d_{AB0},\ max\_speed * travel\_time\} \\ &p_{Bi} \leq p_{Bj}, \quad i < j \end{aligned} \tag{4.1}$$

Where max_speed is an arbitrary constant that constitute an upper bound for the speed and $travel\_time$ the time passed from the last car transition.

Such as for sync also in this context we have to guarantee **I** property, so a car behind can move only after the car in front is synchronized with the DB layer.

Once a car reaches the bridge side and the car on the other side has an higher arrival time, the first car becomes the new leader.

Leader

Leader state is a static state such as sync, in this state the car notifies to the first i rear cars the arrival time of of the car on the opposite side of the bridge (where i is the bridge capacity) with a crossing message.

After the crossing propagation leader returns in the normal state in order to crossing the bridge.

This simple leader election is safe, in the worst scenario the car in front can present a wrong arrival time due to a synchronization error but this at most cause the break of the FIFO order but never a car accident.

Dead

Once a car reaches the end of the bridge goes in the dead state and, after the sending of a global update event, the process stops.

A car can also reach dead from normal and leader states in case of an unexpected crash; in this case the car has to wait for a tow truck that removes it from the street.

Communication

It is important to underline how two cars can send message to each other. First of all, each car has a supervisor and, when a car A wants to send something to another car B, this happens:

1. A sends a message to hers supervisor S_A
2. S_A calls a process, we can refer at it as *Timer*, which is used to send a timeout if the supervisor didn't receive a response from the other within a certain time. In that case S_A assumes that B is dead and so A will call a tow truck to remove it.
3. S_A sends it to the supervisor of B S_B
4. S_B sends an event to B which contains the message of A
5. B sends a message to S_B whit the response
6. S_B sends it to S_A, going through the *Timer* as S_A has done at point 2
7. finally S_A sends another event to A with the response of B, to which A reacts appropiately

Chapter 5

Validation

In this section we will talk about the testing part of the project. In particular our purpose was to show that the system satisfy the requirements listed in **??**.

At first, in order to do that, we wrote some test that consider different case for each state of the car:

a. on each side, and on the bridge itself, there aren't any cars, except the one we are testing;

b. on the same side of our car there is another one in position -1 or 1 depending on the side, while on the other one there is nobody;

c. on the opposite side of our testing car there is another.

At a second time we decided to implement several scenerys in order to see what will actually happen. In the following sections there are also some significant screenshot we took from the UI while running different scenerys.

Tests

Our tests check the correct flow of events. In other words we wanted to find out if the car in a particular situation (defined by the environment) reacts to it sending and receiving the expected events. The car should also act as we wanted so that there is no accident, deadlock, starvation and so on.

5.1 Functional requirements

We were able to check if the system can:

- **Coordinate the traffic**.
- **Avoid** any kind of **accident**.

- Avoid situation where more than n cars, where n is the capacity, cross the the bridge at the same time.
- Make cars cars interchange messages due to decide correctly who can cross, after reached an (**agreement**), the bridge.
- Detect **broken** cars and remove them calling a tow truck.
- generate new cars.

5.2 Non functional requirements

For the non functional requirements we tested that:

- **Safety**: the system guarantees that the bridge can be crossed only by cars that are moving in the same direction, but also that if a car broke, the others are able to detect that and so they stop before crashing into it.
- **Strength**: the system is usable even if a car breaks down and needs to be removed.
- **Scalability**: incrementing number of cars, clients and web services, the system don't affect too much the whole system.
- **Starvation**: when a car requires to cross the bridge, the request will be satisfied eventually.
- **Fairness**: the crossing order is a **FIFO ordering**.

In order to detacht any type of failure we defined different *scenerys*: these are json files which instantiate different number and type of cars. The following are the property of each scenery:

scenery **1** :
there are two cars, on the same side (left), while there is no one on the other side. The cars don't arrive at the same time but there is a delay of 2000ms between them.
The correct behaviour should be that the first car that reach the bridge will cross it (the other one will wait and then cross the bridge too)

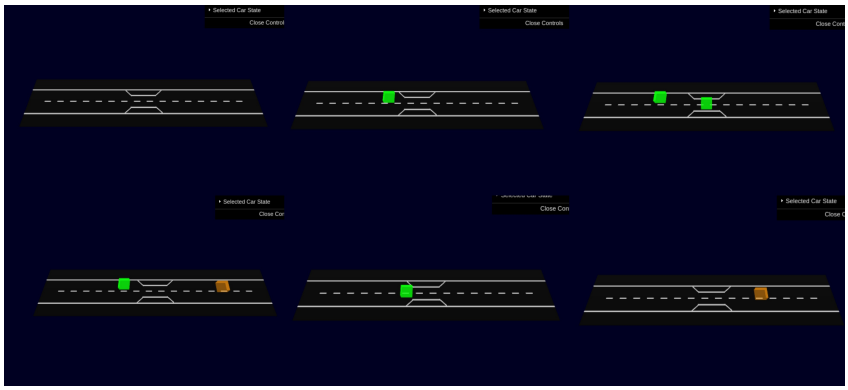

Figure 5.1: scenery 1

scenery **2** :
there are four car from the same side all created at the same time
The correct behaviour should be that the cars are able to synchronize themselves and then cross the bridge according to their *arrival_time*

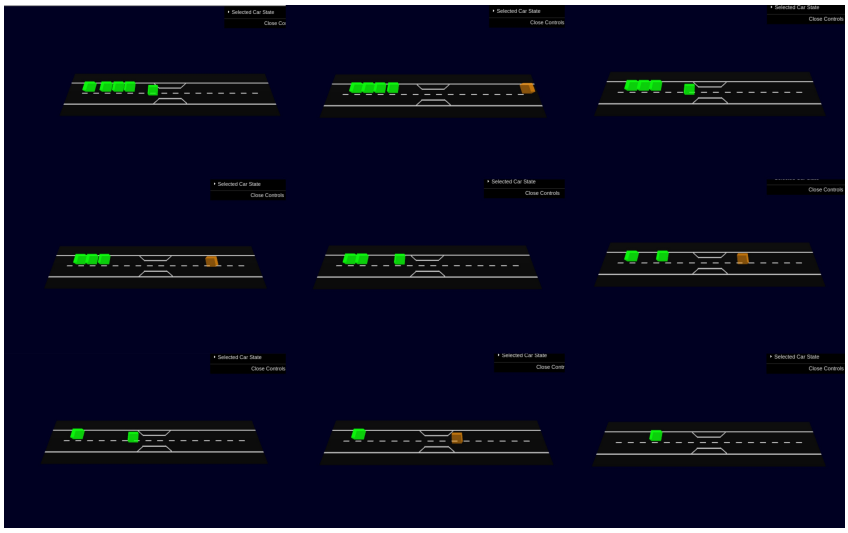

Figure 5.2: scenery 2

scenery **3** :
there are four car on left side and one on the right side (there is no delay between them).
The correct behaviour should be that the cars will cross the bridge in the correct order without crashing with the car on the other side (that will wait for its turn accordingly to its arrival time)

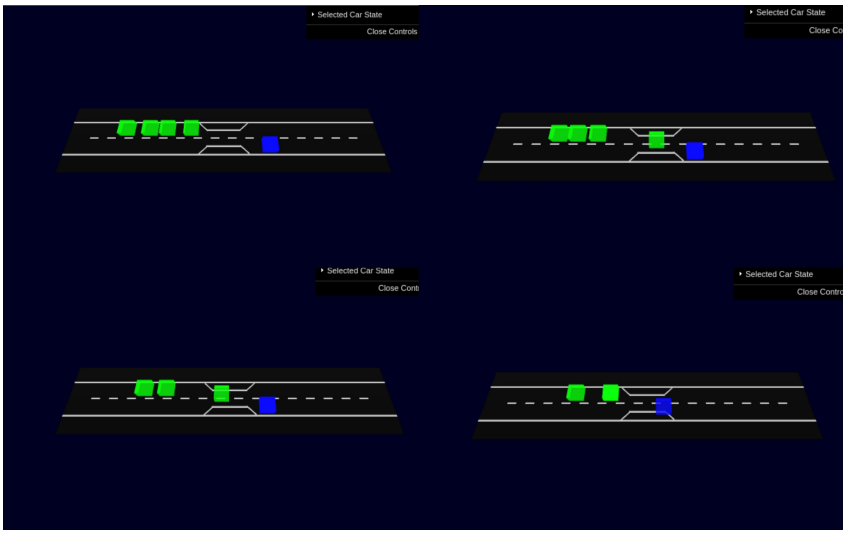

Figure 5.3: scenery 3

scenery **4** :

there are three cars, with no delay, that are on the same side but there is one that is going to have a system crash.
The correct behaviour should be that the two cars that won't have any type of crash will cross the bridge even if there is one in front of them that has crashed. Due to the fact that the *crash_type* for the broken one is 2, the other cars (at least one of them) have to call the tow truck for it; after it has been removed the cars can continue crossing the bridge

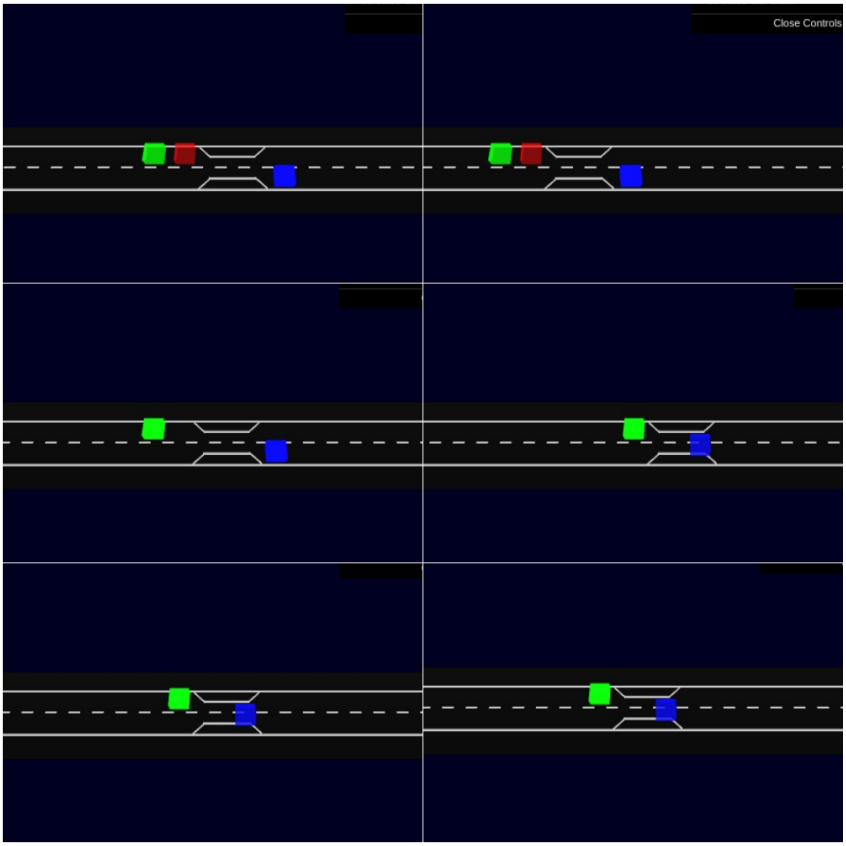

Figure 5.4: scenery 4

scenery **5** :

there are five cars that are on the same side and all of them have a crash; three of them have system crash, while the other have only engine crash and there is a delay between the car with *crash_type* 2 and the others. The correct behaviour should be that the cars that have only engine crash will call the tow truck for themselves and for the cars that they can reach (because if a car has system crash cannot call a tow truck for itself). For this reason if there is a car that has a system crash after that the cars, which had an engine crash, have already been removed by the tow truck, it will stuck there, waiting for a new car to call the tow truck for it

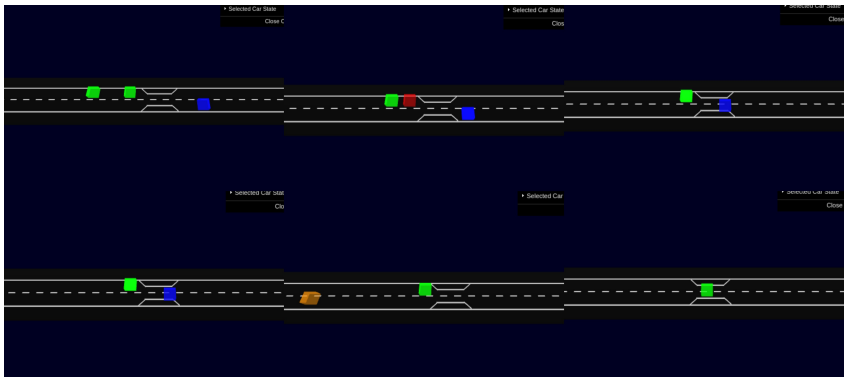

Figure 5.5: scenery 5

scenery **6** :
there are two cars that will have a crash with *crash_type* 2 and another one that will not crash and that will arrive after that the previously cars have crashed due to the delay between them. All of them are on the same side of the bridge.
The correct behaviour should be that the cars with the crash will wait there until the other one arrive and call a tow truck for them. After they have been removed the last car can cross the bridge

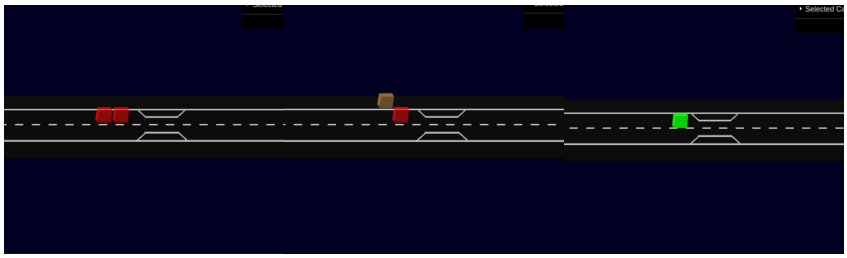

Figure 5.6: scenery 6

scenery **7** :
it is the same as the above but there are only two cars and them are on different sides; the sane one has a delay.
The correct behaviour should be that when it arrives, it will call the tow truck for the other and then cross the bridge (after the other has been removed)

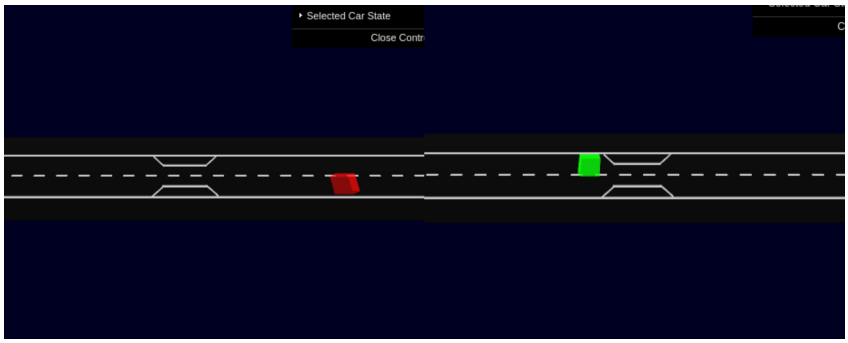

Figure 5.7: scenery 7

scenery **8** :
there are five cars on the same side but only one (with the longer delay) is sane.
The correct behaviour should be that the cars with crashes will wait to be removed by the tow truck called by the sane one (they have *crash_type* 2) and then the last car can cross the bridge

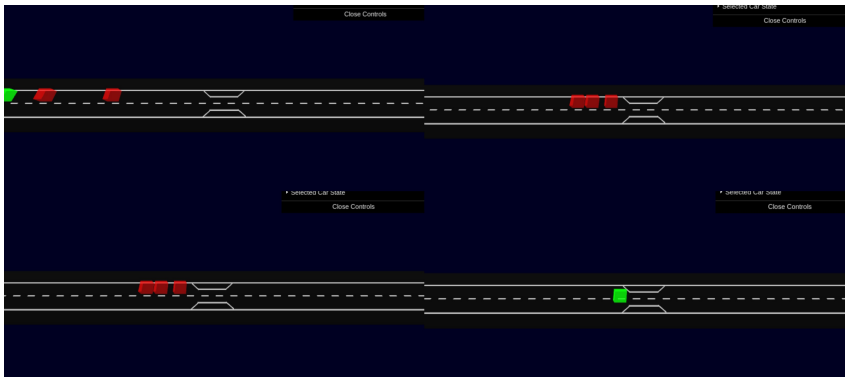

Figure 5.8: scenery 8

scenery **9** :
there are three cars, two from the same side. One of them has a system crash and a delay of 300ms.
The correct behaviour should be that the car with crash will wait to be removed by the tow truck called by one or both the others cars.

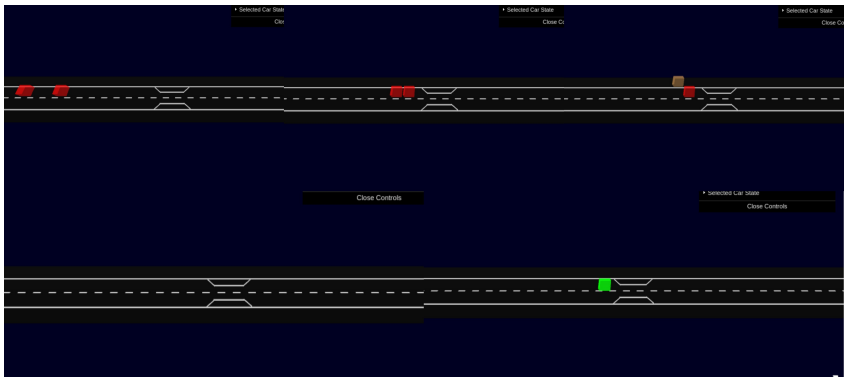

Figure 5.9: scenery 9

We tested also others scenarys:

- 9 other scenarys that are the same as the 9 already shown but with opposite side for each car (i.e. if a scenary previously listed has two car, one from right and one from left, in the new scenary the right one would be on the left and the left one on the right)
- 6 other scenarys that are the same as the 6 already shown (but only these with cars that will crash) but, in the new scenery, every car that has a *crash_type* 2 will have *crash_type* 1 and vice versa. This case is even symplier because the *crash_type* 1 allows the crashed car to call the tow truck by itself.

Chapter 6

Conclusions

The goal of our work was to develop a system that models the behaviour of a bridge. We managed to ensure most of the requisites.
In particular our model avoids any accident between two or more cars: this is really important because in a real world the first thing to ensures is that nobody gets hurt or damaged. For the same reason we made the cars able to detect any broken car. The case of the broken engine is the simpliest to handle because the broken car can still communicate with the others and so notify them she is not moving. Moreover the tow truck can remove the car so that the rear ones can eventually move again.
Similar to that was ensuring that nobody will wait forever to cross the bridge. In a realistic situation it is not contemplated that one or more cars could never cross the bridge.
The chosen strategy is FIFO order. This avoids starvation but isn't the best choice for performance: consider the case where one car arrives on the left side while on the right there are, for example, 20 cars, and that the bridge has capacity equal to 1 and length 30. The car on the left has to wait for all the other 20 on the right side: this is a very long time due to the fact that only one car per time can cross the bridge and to the length of the bridge. If that car could cross without waiting for all the others its waiting time would decrease significally while the waiting time for the others will increase only as if there were one car more before them. So one thing that can improve the system could be order the cars by minimizing the average waiting time.
Finally the system satisfy all the requisites of the bridge (capacity and one way cross).

Another important matter we have to discuss is the performance of the system. Everytime a car has to move it has to send many message to its adjacent cars: for this reason the cars cannot move in a fluent way but have to stop sometimes. Moreover we cannot visualize a new car if there

is already another one on the street that has *crash_type* 2, in fact, if it is broken it isn't able to tell the new car its position and so the new one has to wait until it is removed (notice that the tow truck will be called by the new car before going on the street). Pay also attention to the *max_RTT* that will be set because a short one could led to assume that some cars are dead even if they are not (only because their response to the check would arrive too late), but on the other hand a long one could be unnecessary (and so the cars would wait longer before making a decision).

The last aspect we would talk about is how the *crash_type* effects the performance of the system. As we could imagine a car that crash only because its engine is broken would be easier to handle. This because it can call by itself the tow truck. Moreover it can also keep sending messages to the other and so they can move and reach the last ammisible position without crashing with it. In other words the whole system works faster.

Appendix A

Appendix

In the Appendix you can put code snippets, snapshots, installation instructions, etc.

Evaluation