

# The Bottleneck Problem

Distributed Systems - Project 2019

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

Version 0.1, July 20, 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

|  |           |
|--|-----------|
| <b>Contents</b>                                    | <b>2</b>  |
| <b>1 Introduction</b>                              | <b>3</b>  |
| 1.1 The Problem . . . . .                          | 3         |
| 1.2 Requirements . . . . .                         | 4         |
| <b>2 Analysis</b>                                  | <b>5</b>  |
| 2.1 Functional requirements . . . . .              | 5         |
| 2.2 Non functional requirements . . . . .          | 5         |
| <b>3 Project</b>                                   | <b>7</b>  |
| 3.1 Logical architecture . . . . .                 | 7         |
| 3.2 Protocols and algorithms . . . . .             | 8         |
| 3.3 Synchronization problem . . . . .              | 8         |
| 3.4 Agreement . . . . .                            | 8         |
| 3.5 Failures . . . . .                             | 9         |
| 3.6 MIM attack . . . . .                           | 10        |
| 3.7 Scalability . . . . .                          | 10        |
| 3.8 SPOF . . . . .                                 | 11        |
| 3.9 Physical architecture and deployment . . . . . | 14        |
| 3.10 Development plan . . . . .                    | 14        |
| <b>4 Implementation</b>                            | <b>15</b> |
| 4.1 Tools . . . . .                                | 15        |
| 4.2 Docker . . . . .                               | 17        |
| 4.3 Data structures . . . . .                      | 18        |
| <b>5 Validation</b>                                | <b>23</b> |
| 5.1 Functional requirements . . . . .              | 23        |
| 5.2 Non functional requirements . . . . .          | 24        |
| <b>6 Conclusions</b>                               | <b>25</b> |
| <b>A Appendix</b>                                  | <b>27</b> |

# 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 using 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 set 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 malicious (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).

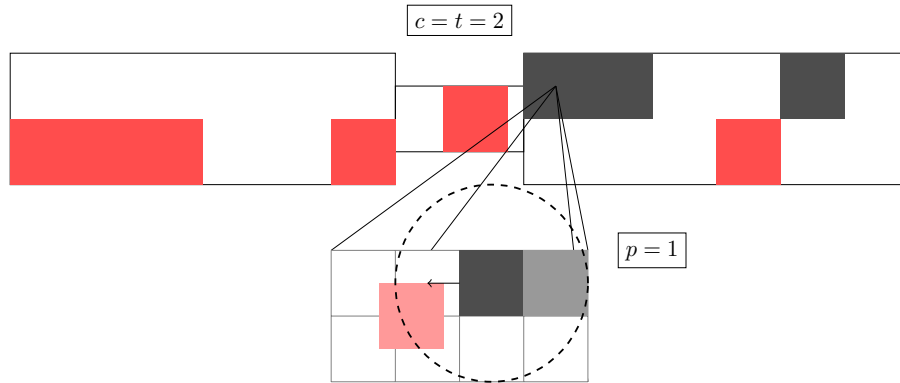


Figure 1.1: Example of a possible situation

## 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 have 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 timing 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 **drifts** 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 conscious 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

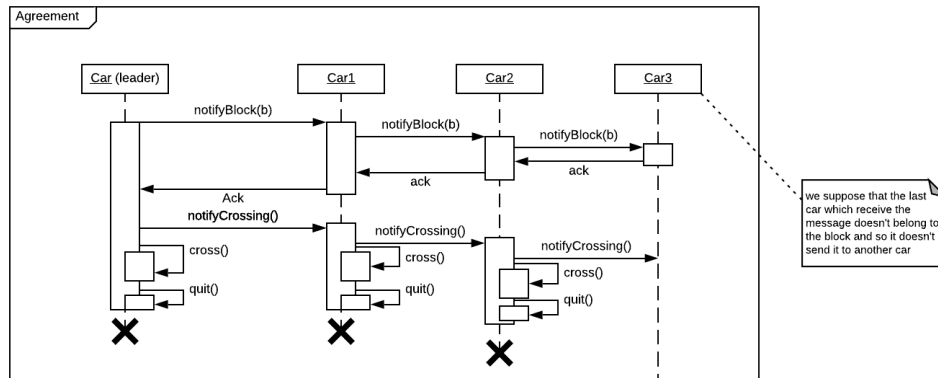
Before crossing the bridge, the leader  $A$  propagates a message to the first  $n$  cars behind him (where  $n$  is the capacity of the bridge) telling them to cross the bridge if their arrival times are lower than the arrival time of  $B$ , the car on the opposite side (if present).

Each car, who receives the message, checks if its arrival time is less than  $B$ '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 and whose arrival time is greater than  $A$ 's then  $A$  is the new leader;
- if there isn't any car,  $A$  is the new leader.



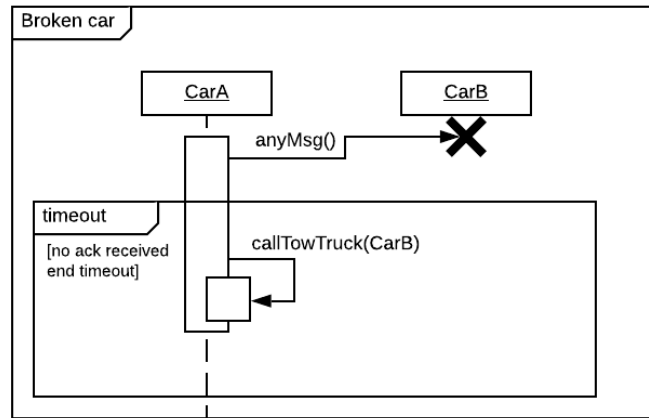
### 3.5 Failures

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  $max\_RTT$  allowed) it is assumed that the receiver has a failure (and so the tow truck will be call).

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.



### 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).

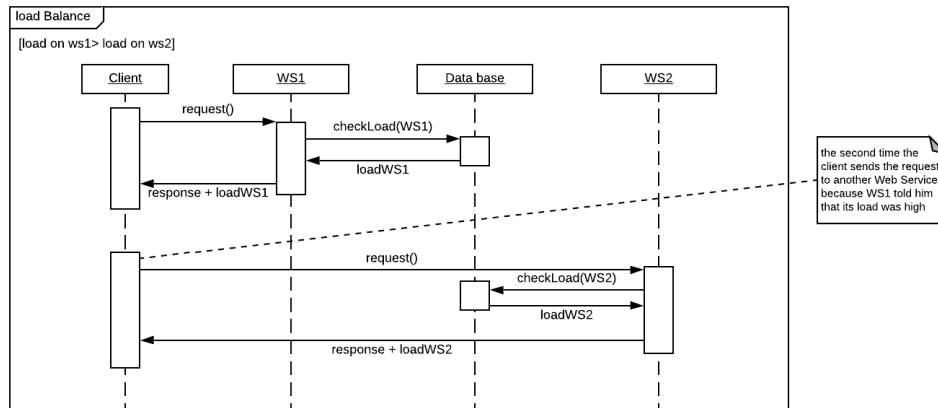
#### Solution

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).



### 3.8 SPOF

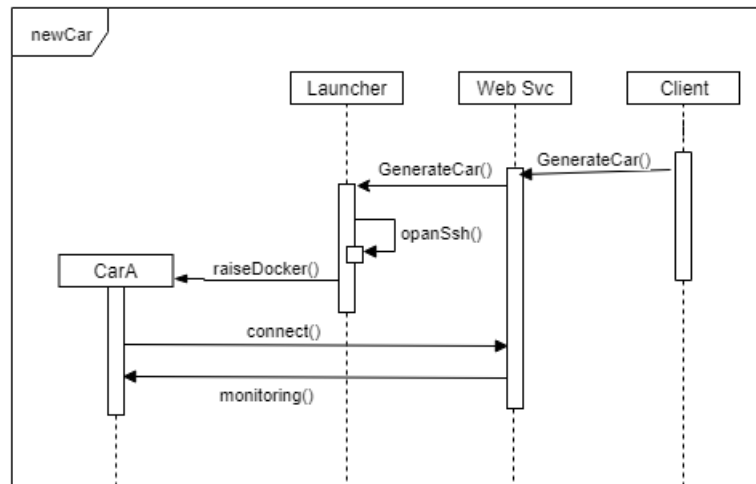
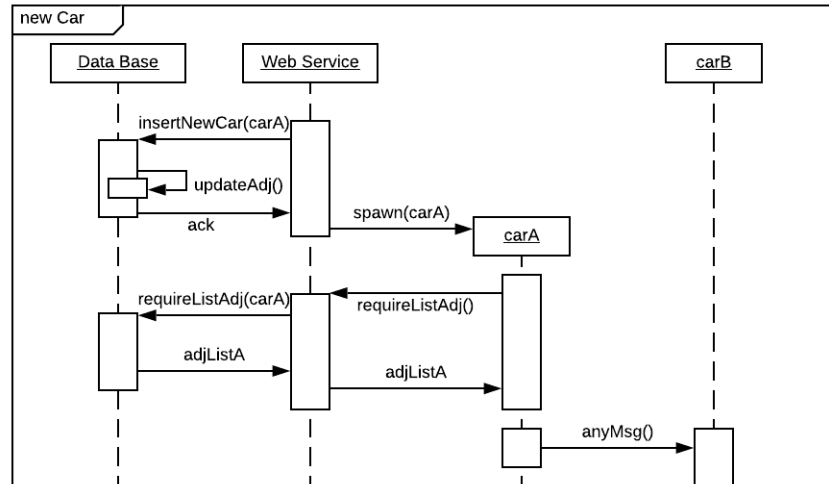
The system must be resilient wrt. failures (even multiple failures); so basically the architecture must be deeply **distributed** and **decentralized** (avoid any SPOF).

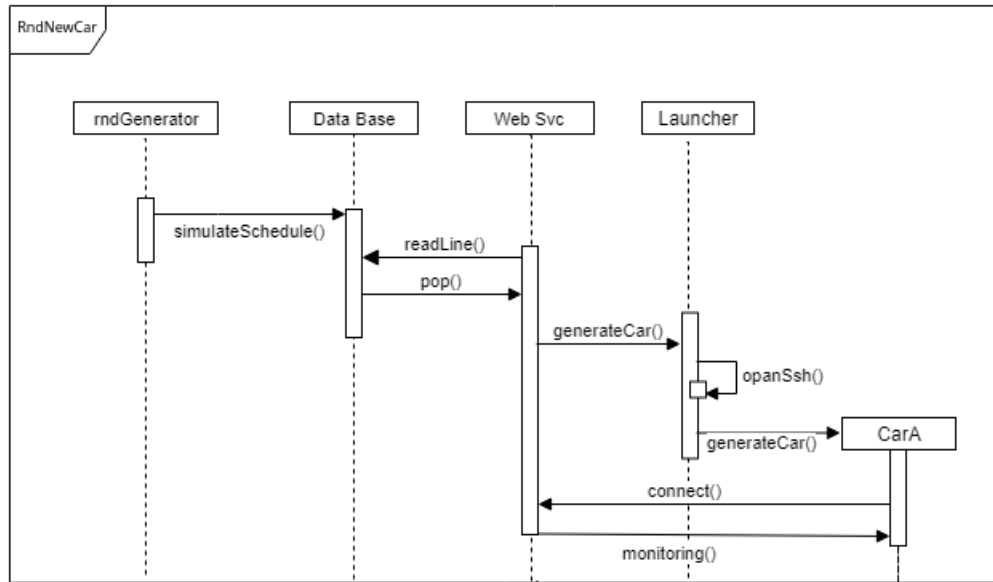
#### Solution

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.

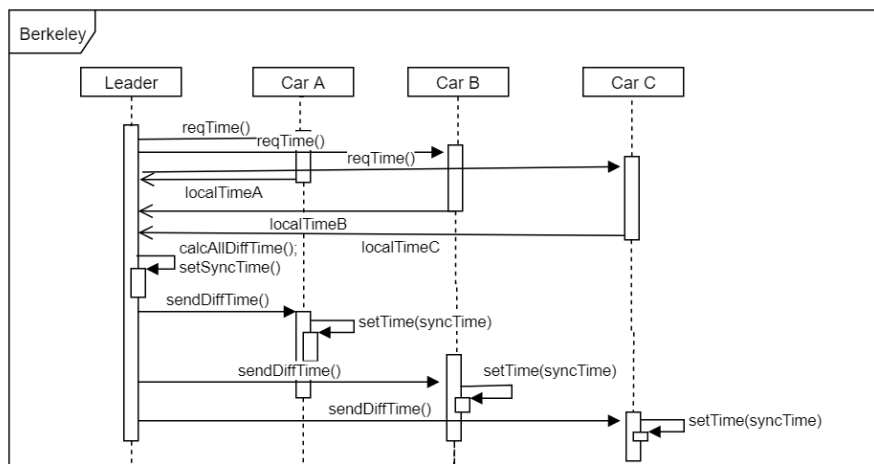
This issue can be solved with a peer to peer layer of web services and some instances of a **distributed DBMS** like Mnesia [?] (but isn't a project requirement).





## Algorithms

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



### 3.9 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** [?].

### 3.10 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 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 call 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 constitute 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 interpose 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 exposed 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 exposed 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 thought 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.

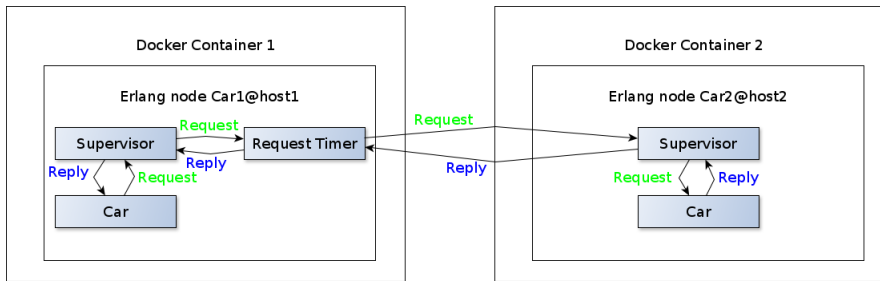


Figure 4.1: Message exchange between two cars

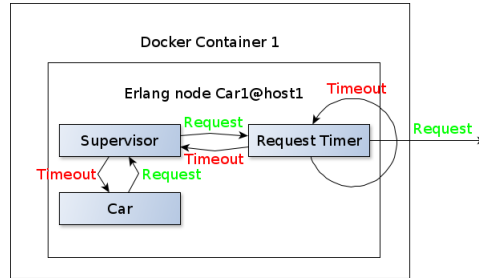


Figure 4.2: A message without any reply triggers a timeout event

## 4.2 Docker

At the end of the implementation the Erlang scripts were built 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.

### 4.3 Data structures

As described in Chapter 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

To model the environment the following attributes, which are set in *environment.js* for once, will be used:

- *host*:  
Web service IP:PORT
- *max\_speed*:  
the maximum speed allowed
- *bridge\_capacity*: number  
it defines the maximum number of cars that can cross the bridge at the same time
- *bridge\_length*: number  
it defines the length of the bridge
- *tow\_truck\_time*: number  
time necessary for the tow truck to arrive and remove the dead car
- *max\_RTT*: number  
it is the maximum RTT that could happen

These attributes will never be modified during the simulation.

#### Car

Each car A is modelled exploiting the following attributes:

- *name*: number  
it identifies univocally the car
- *side*: number  
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

- *power*: number  
it defines the number of rear and front cars that the car can reach
- *size*: number  
it defines the lenght of the car
- *speed*: number  
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
- *position*: number  
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
- *crossing*: boolean  
if true it means that the car is on the bridge
- *synchronized*: boolean  
if true it means that the car is synchronized
- *delta*: number  
it is the difference between the local time and the global time
- *arrival\_time*: number  
it defines the time when the car has arrived in the queue
- *current\_time*: number  
it indicates the local time at each turn
- *adj*: another data structure  
it contains the lists of front and rear cars
- *state*: string  
it defines the current state of the car  
it could be **init**, **sync**, **normal** or **dead**

These above are the car metadata, while the following are settings and bridge metadata (these are taken from the environment definition but must be part of the car specifics too):

- *host*:  
Web service IP:PORT
- *bridge\_capacity*: number  
it defines the maximun number of cars that can cross the bridge at the same time

- *bridge\_length*: number  
it defines the length of the bridge
- *max\_speed*:  
the maximum speed allowed
- *tow\_truck\_time*: number  
time necessary for the tow truck to arrive and remove the dead car
- *max\_RTT*: number  
it is the maximum RTT that could happen

Note that *bridge\_capacity* and *bridge\_length* are imposed by the environment and cannot change during the simulation. Furthermore *name* and *power* are defined independently, while *delta* and *adj* depend on the other cars.

The problem allows to describe the car as a finite state machine. We choose this modality because the car can switch between four main different states. The first one is a so called **init**: the undefined attributes are computed here. It is assumed that in this state the car cannot receive any sort of event and that cannot crash.

When the car is synchronized it changes its state in **sync**: the car call the web service to receive the list of adjacent cars. When it is received the car knows how to communicate with the current front car and so can define its *speed* and *position* according to the state of the front car. Finally the car can change its state to **normal**.

In **normal** the car still sends repeatedly a check event to the front car in order to regulate her speed (and consequently the new position).

In particular, at every iteration, the car computes the following function to determinate the new *speed*:

$$\frac{pos(A) - pos(B)}{RTT + turn}$$

and the new *position*:

$$pos(A) + \frac{|pos(A) - pos(B)|}{RTT + turn}$$

where  $pos(x) = position$  of x.

Obviously this means that if the distance between A and B is zero, the car will change its speed to zero: this avoids crashing during the next turn even if the front car says that its *speed* is greater than zero. In fact, if B crashes immediately after sending its state, the other car assumes that B is moving and so she decides to move too ending up colliding with B.

When the car finally reach the position zero she become a leader and so its state would be the **leader** state. If there is a car B on the opposite side (that can be in the leader state too), they have to decide who will start crossing:

- if A's *crossing* attribute is true and her arrival time is less than B's, she start crossing the bridge
- if A's *crossing* attribute is false and her arrival time is less than B's, she change it to true
- if A's arrival time is greater than B's, she stops and waits her turn

When it's its turn, the car change its *position*, that will be equal to the bridge lenght, and its *speed*, that will be the maximun speed allowed. Before changing state to normal the car has also to propagate to the first n cars behind her (n is equal to the bridge capacity) to change *crossing* value to true. After that she moves and returns to the **normal** state.

Obviously there is another state called **dead** that could be reached from every state, except for **init**, when the car's engine or the car's engine and system crash or when the car completes the crossing. In the first case the car cannot move anymore and so somebody (the car itself or the first behind her) must call the tow truck to be removed. Anyway the car must send periodically a check message to the others in order to find out if one is dead.

Notice that if there is a car whose *speed* = 0 it may mean that the car is stationary, waiting the front car to move, or in **dead** status.

### 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 want 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 send 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 appropriately

## Adj

We used another data structure for the adjacent cars, called *adj*:

- *front\_cars*: list of cars  
it contains all the car ahead that are reachable  
the first element is the next car
- *rear\_cars*: list of cars  
it contains all the car behind that are reachable  
the last element is the previous car

## 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 2. 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 there is a car.

### 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**.



## Chapter 6

# Conclusions

The goal of our work was to develop a system that models the behaviour of a bridge. We managed to ensure the most of the requisite.

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 ensure is that nobody get 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 simplest 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 significantly 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).

??? something more ???



## Appendix A

# Appendix

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



# Evaluation