The Bottleneck Problem

Distributed Systems - Project 2019

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

Version 0.1, July 12, 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**.

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 crossing time t.

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).

For simplicity we assume that every car has the same dimensions expressed in an arbitrary length scale called **block** and **every measure is an integer** (s, 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 help 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).

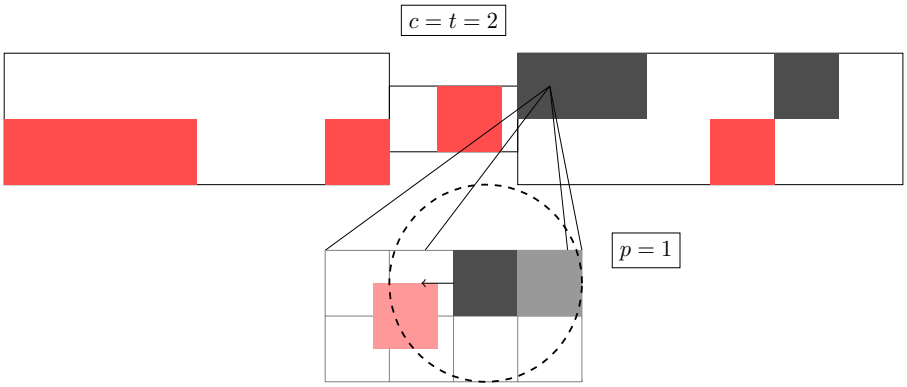


Figure 1.1: Example of a possible situation

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**.
- The maximum amount of cars that can cross the bridge at the same time, in the same direction (a **block**), is equivalent at 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 cant communicate and/or move any more) it will be removed by a tow truck.
- provide a **UI** in order to monitoring the **simulation**.
- 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 doesnt 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 that is when a car breaks down and so it 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.
- **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**).

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 of a peer-to-peer layer 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 an API for the clients.
- **Client** a simple web page used to monitoring the simulation state; uses the web service APIs to render a graphic interface that must be constantly syncronized.
- **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 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 syncronize its local timing using Berkeley algorith
- **Distributed DBMS**, an instance of Mnesia DBMS distributed on each container 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

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 lenght 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 syncronize themselves (for the syncronization 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 first car that has reached the bridge; once the current leader cross the bridge there is a new election of the car on the other side of the bridge (if one is present).

3.3 Syncronization problem

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

Solution

So we need to syncronize 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 syncronized). 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 (or the average time of the first block of incoming cars).

If some new unsyncronized cars appear in block the syncronization process takes into consideration the average RTT (according to the Berkeley algorithm).

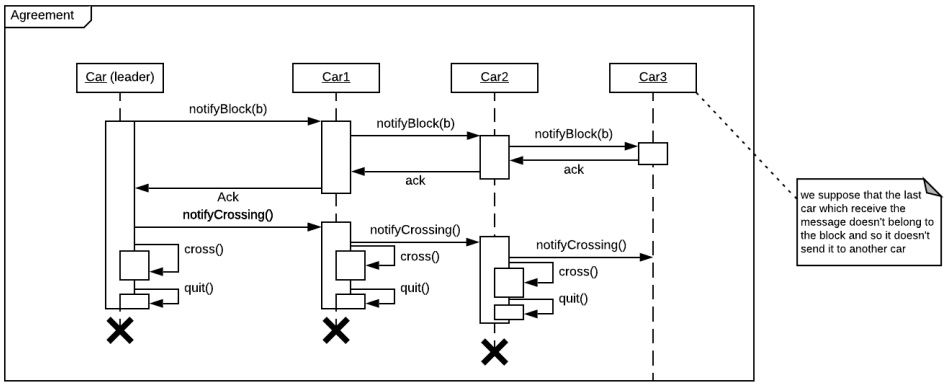
3.4 Agreement

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

Solution

Before crossing the bridge, the leader propagates a message to the first n cars behind him (where n is the capacity of the bridge) telling them to cross the bridge and the identity of the first car B on the other side. Each car, who receives the message, checks if its arrival time is less then B's: if so, the car can cross the bridge too, otherwise waits for its turn.
The leader will always be elected by himself. Suppose the car A, that is not leader and that has not the permission to cross, arrives at the last position before the bridge. At that point A checks if there are any cars in front of her:

- if there is a car whose arrival time is greater then A's and is on the opposite side then A is the new leader;
- if there isn't any cars then A is the new leader

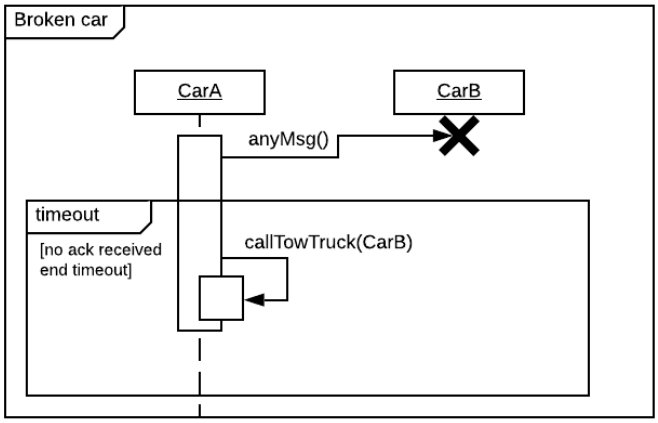


3.5 Failures

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 RTT) it is assumed that the receiver has a failure (and so the tow truck will be call). Then the car have to wait to be removed and then notify the first car behind that she has been removed after the tow truck timeout. The car has also to tell the name of the car in front of her so that the rear one can communicate with the next one.
Every time something like that happens, the remaining cars must update their adjacent lists.



3.6 MIM attack

The system must withstand a MIM attack.

Solution

Each message will be encrypted (HTTPS).

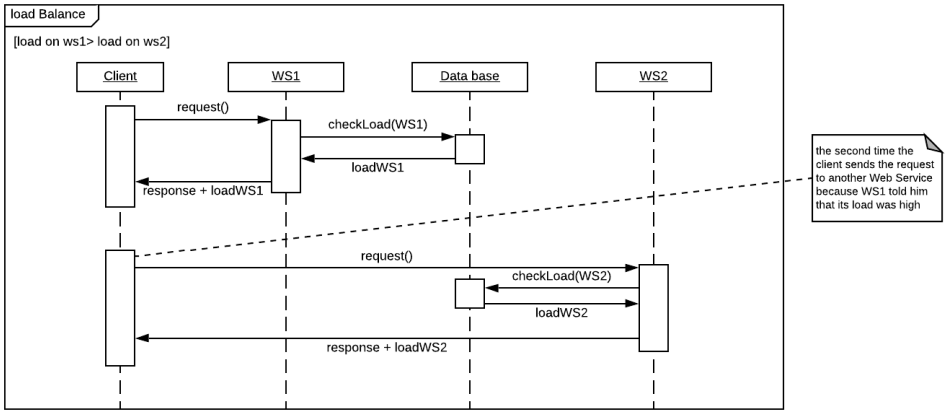
3.7 Scalability

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

Solution

Assume that the available machines are defined in an arbitrary way in the initialization phase; on each machine some docker containers will be raised and foreach one of those an arbitrary number of web services and car processes will be spawned.

[Bonus requirement, not necessary] We will design an embedded load balancer in order to avoid the web services overload (considering the number of calls).

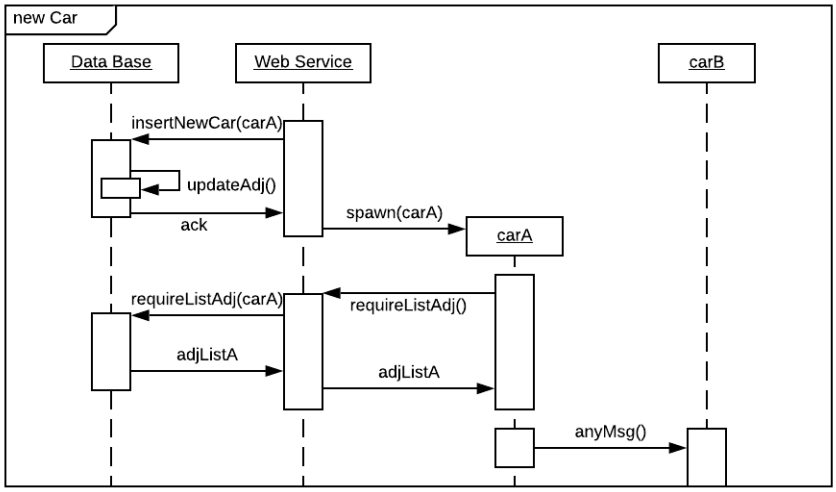


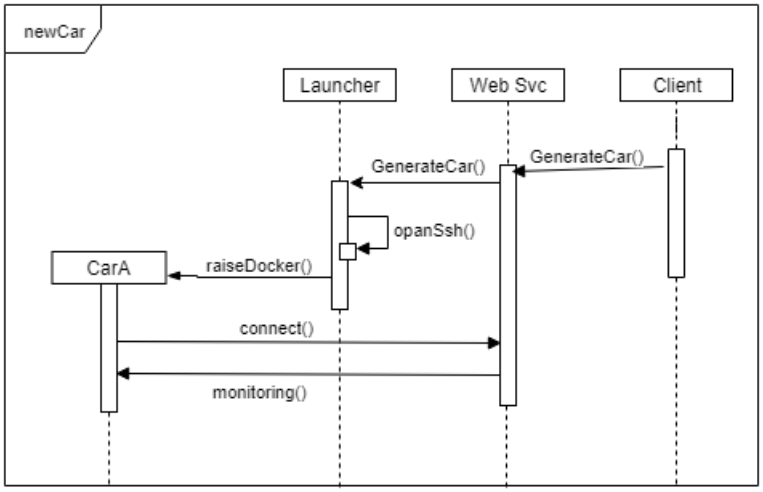
3.8 SPOF

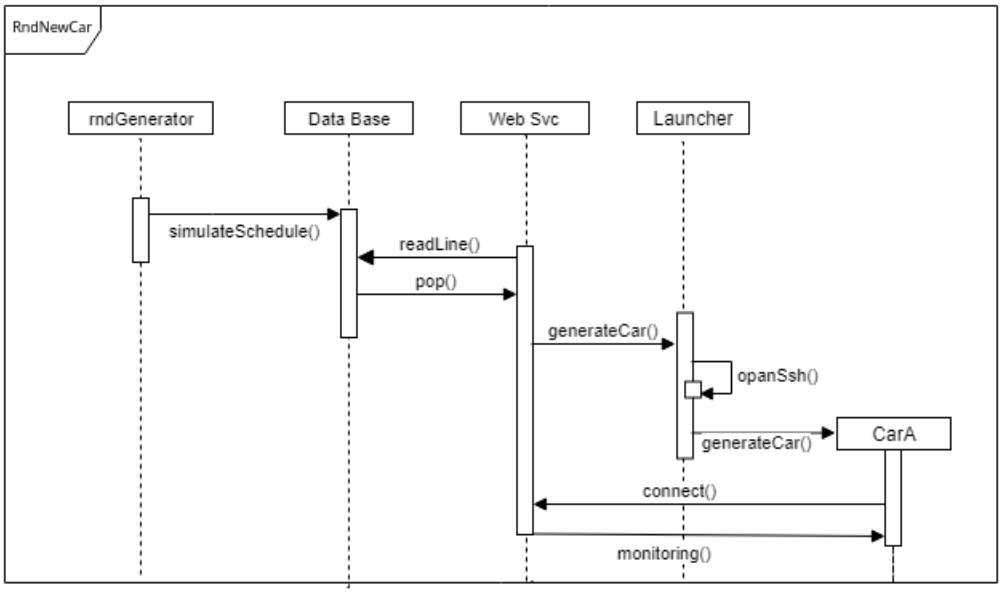
The system must be resilient wrt. failures (even multiple failures); so basically the architecture must be deeply **distributed** and **decentralized** (avoid any SPOF).

Solution

The scalability requirement imposes that every macro-component must be scalable so the architecture provides some **peer-to-peer layers** (web services layer, cars layer). 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. A possible solution can be using a **distributed DBMS** like Mnesia [**?**] in order to keep track of the queue state.

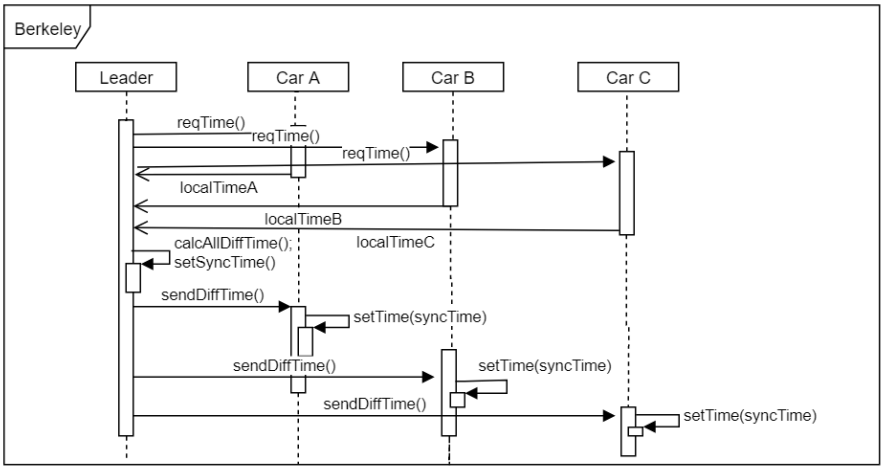

new Car
Data Base
Web Service
carB
insertNewCar(carA)
updateAdj()
ack
spawn(carA)
carA
requireListAdj(carA)
requireListAdj()
adjListA
adjListA
anyMsg()


newCar
Launcher
Web Svc
Client
GenerateCar()
GenerateCar()
opanSsh()
CarA
raiseDocker()
connect()
monitoring()



Algorithms

We have used the Berkeley algorithm in order to syncronize 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 of each computer involved in the simulation
2. Execute a launcher bash script that will open an ssh tunnel with each computer and raise the Docker Containers with an arbitrary number of web services
3. A random generator or a test procedure inserts into the DB a simulation schedule (the web service p2p layer starts automatically the simulation)
4. With a web browser any client can connect to the web service API and monitoring the simulation state

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).

Chapter 4

Implementation

In this chapter we will explain how the system was implemented, so we will analize the languages, the data structures that were used and something more, inherent to that.

4.1 Language

To implement the project Erlang language was used. The decision was made due to the nature of the language itself.
In fact, Erlang allows to handle concurrency and distributed programming: programs that can handle several threads of execution at the same time. Actually each processo only swaps between jobs so fast, so it only gives the illusion of running them all at the same time. What is great about Erlang is that it make it easy to create parallel threads of execution and it also permits these threads (called process because they do not share data) to communicate with each other.
Furthermore Erlang can models finite state machine thanks to the gen_statem behaviour and that fitted perfectly with the nature of the problem.

The standard data interchange format used is JSON. The main reason of this decision is that it is used for transmitting data between a web application and a server. Moreover JSON files are lightweight, text-based and human-readable, so we tought that would be good for our goal.

4.2 TODO– ?docker?

Details about the implementation: every choice about platforms, languages, software/hardware, middlewares, which has not been decided in the requirements.

4.3 Data structures

As described in chaper 1, cars and bridge must be aware about some of their characteristics. In order to do that we decided to implement the car with a new data structure as follows:

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 maximun speed allowed
- *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 lenght 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 maximun 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

- *syncrhonized*: boolean
  if true it means that the car is syncrhonized

- *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 lenght of the bridge

- *max_speed*:
  the maximun 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 maximun 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 syncrhonized it changes its state in **sync**: the car call the web service to receive the list of adiacents 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 thee car can change its state to **normal**.
In **normal** the car still sends repeatedly a check event to the front car in order to regolate hers 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 then 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.
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 appropiately

Adj

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

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

What has been done with respect to what has been promised in Chapters 1 and 2, and what is left out.

Appendix A

Appendix

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

Evaluation