The Bottleneck Problem

Distributed Systems - Project 2019

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

Version 0.1, May 28, 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).

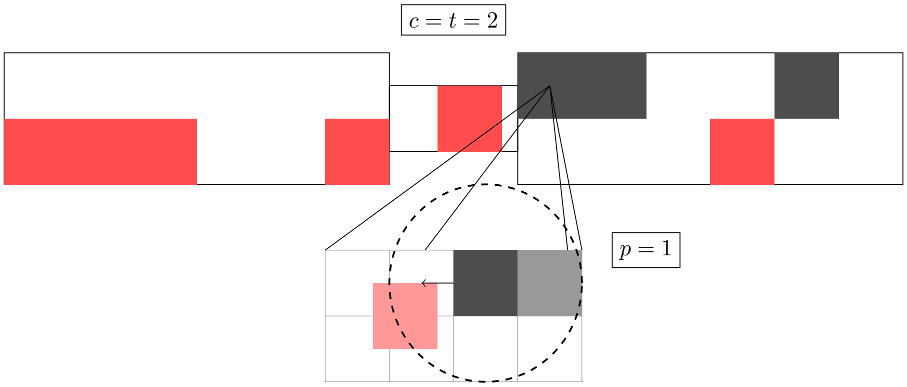


Figure 1.1: Example of a possible situation

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. Finally c sends to the furthest car reachable in the queue its arrival time (and the message is propagated along the peer chain to the current leader).

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).

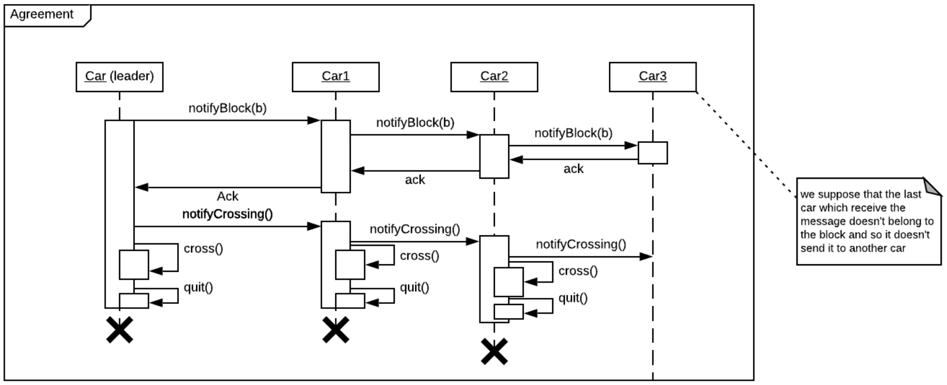
3.4 Agreement

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

Solution

The leader is concious of the current queue state in terms of arrival time of the cars; so it decides how may cars have to cross the bridge in this turn (accordingly with the bridge capacity c and FIFO ordering).

Finally the leader notifies to the involved cars its decision and, after an acknowledgement (agreement), the block starts crossing.

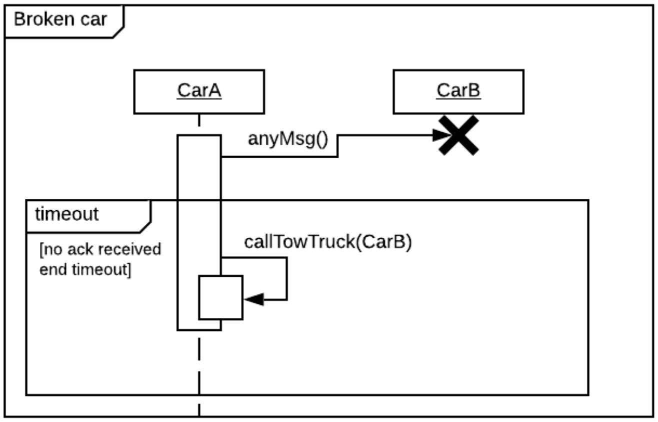


3.5 Failures

In case of failure another car (or the same car) has to call a **tow truck** for help.

Solution

Each car recursively checks if other reachable cars are safe; if a check message hasn't any response before a certain **timeout** (depends on the RTT) is assumed that the receiver has a failure (and so the tow truck will be call).



3.6 MIM attack

The system must withstand a MIM attack.

Solution

Each message will be encrypted (HTTPS).

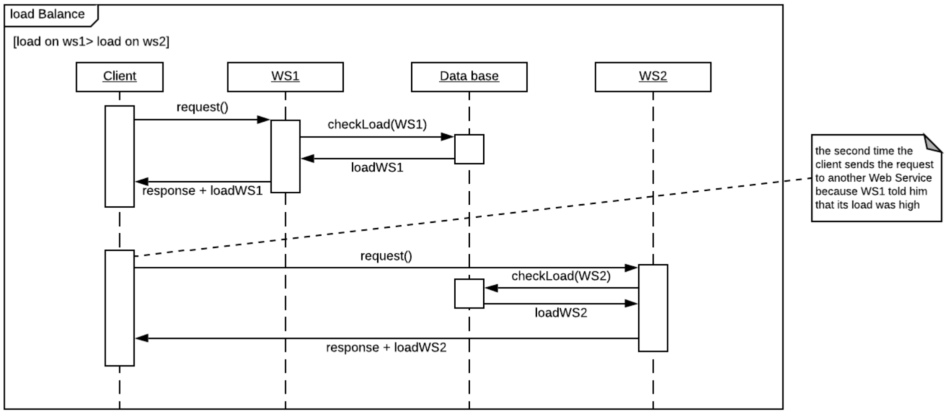
3.7 Scalability

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).

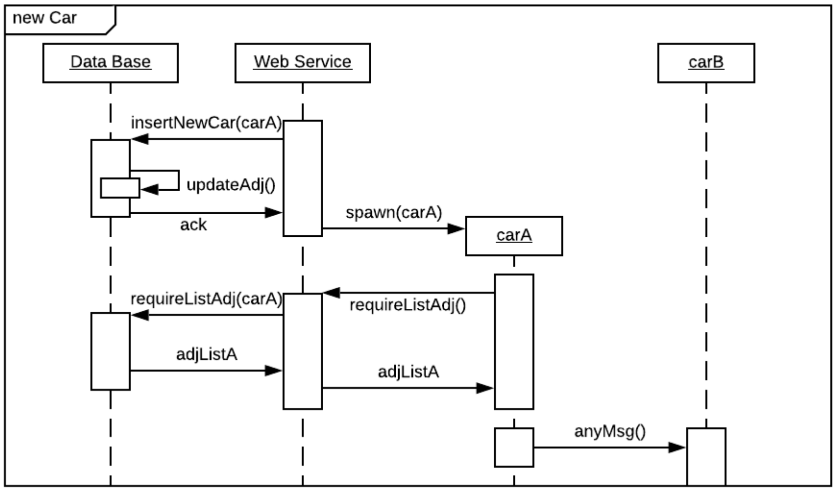

load Balance
[load on ws1> load on ws2]
Client
WS1
Data base
WS2
request()
checkLoad(WS1)
loadWS1
response + loadWS1
the second time the client sends the request to another Web Service because WS1 told him that its load was high
request()
checkLoad(WS2)
loadWS2
response + loadWS2

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.



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

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

Important choices about implementation should be described here; e.g., peculiar data structures.

Chapter 5

Validation

Check if requirements from Chapter **??** have been fulfilled. Quantitative tests (simulations) and screenshots of the interfaces are put here.

Chapter 6

Conclusions

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

Appendix A

Appendix

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

Evaluation