The Bottleneck Problem

Distributed Systems - Project 2019

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

Version 0.1, May 25, 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 **length** $l \geq 1$.

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** $p \geq 1$.

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).

Moreover a car has a certain **speed** $s \geq 1 \frac{block}{s}$ that determines the bridge crossing time and the time to reach the first obstacle (another enqueued car or the bridge).

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:

- **engine 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).

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.

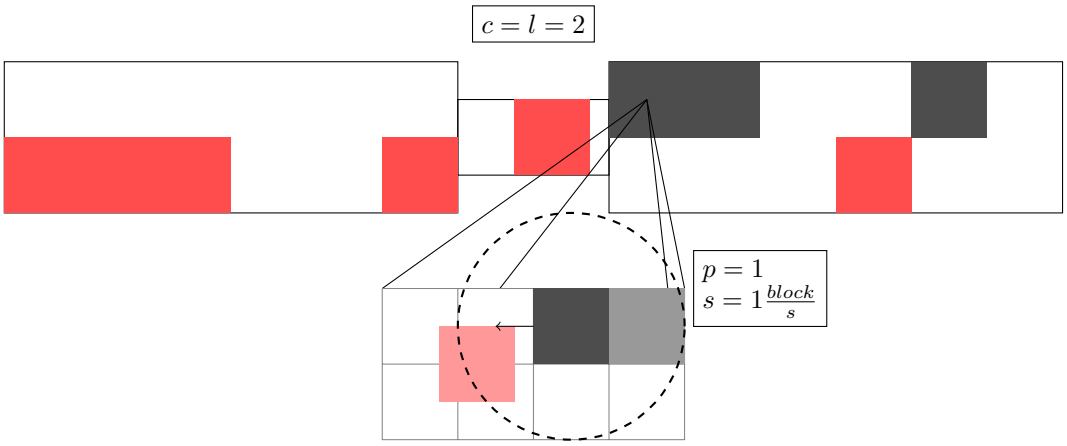

Figure 1.1: Example of a possible situation

1.2 Subproblems

Following the *divide et impera* philosophy now we are going to split the problem into some subproblems and solve them.

Starvation/deadlock and order method

[TODO] 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 reaches 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).

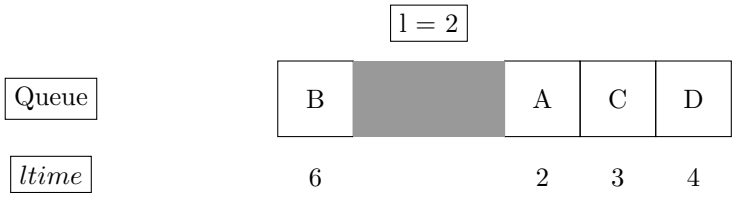

Figure 1.2: Minimizing global waiting time: A-C-D-B. $gtime = 15$

If on the right side we have an infinite queue then B waits forever and the *gtime* is the lowest.

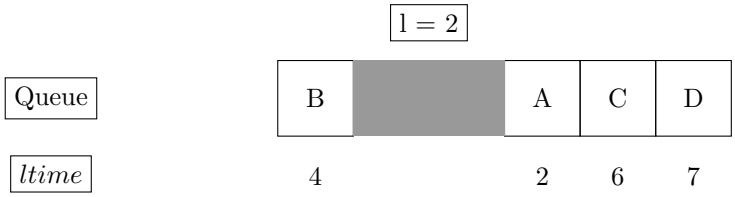

Figure 1.3: FIFO order: A-B-C-D. $gtime = 19$

FIFO order avoids deadlock and starvation but doesn't minimize *gtime*. The idea is then find the right trade off between minimizing *gtime* and minimizing *ltime* locally (starvation).

The concept of equity comes to the rescue cause

1.3 Syncronization problem

Every car has a local timing that can drifts out from the global timing so we need to syncronize them using the Berkeley algorithm.

Solution

A new incoming car calls the *get_adjacent_cars(p)* method in order to get the name of the adjacent cars.

[TODO] capire che c'e' prima e chi c'e' dopo

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).

1.4 Agreement

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

1.5 Communications

1.6 Solution

Chapter 2

Analysis

In this chapter, we describe in detail functional and non-functional requirements of a solution for the problem.

2.1 Functional requirements

Which functions must be offered to users / other programs? Which are the input data and the output data? Which is the expected effect?

2.2 Non functional requirements

Everything about mode and transparencies: availability, mobility, security, fault tolerance, etc.

Are there execution time bounds? Minimum data rates?

If requested, specific platforms/languages/middlewares requirements for the implementation can be decided here. (E.g.: if the project is on a SOA, we may request that functions are offered via SOAP or RESTful services).

Chapter 3

Project

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

3.1 Logical architecture

Describe the components of your systems: modules/objects/components/services. For each component, describe the functionalities it implements, and by who is used.

3.2 Protocols and algorithms

Communication between components. UML sequence diagrams go here.

Also, put here a detailed description of distributed algorithms used to solve specific problems of the project.

3.3 Physical architecture and deployment

Which nodes and platforms involved, and where each component is deployed.

3.4 Development plan

Since it is difficult to predict just how hard implementing a new system will be, you should formulate as a set of "tiers," where the basic tier is something youre sure you can complete, and the additional tiers add more features, at both the application and the system level.

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