# Architecture and Negotiation Protocols for a Smart Parking System

**Bruno Rafael Alves - 39455**

Dissertation presented to Escola Superior de Tecnologia e de Gestão de Bragança to obtain the grade of Master in Sistemas de Informação.

Work oriented by:

Prof. André Pinz Borges

Prof. Gleifer Vaz Alves

Prof. José Eduardo Fernandes

Prof. Paulo Leitão

Bragança

2018-2019

# Arquitetura e protocolos de negociação para sistema de estacionamento inteligente

**Bruno Rafael Alves - 39455**

Dissertação apresentada à Escola Superior de Tecnologia e de Gestão de Bragança para obtenção do Grau de Mestre em Sistemas de Informação.

Trabalho orientado por:

Prof. André Pinz Borges

Prof. Gleifer Vaz Alves

Prof. José Eduardo Fernandes

Prof. Paulo Leitão

Bragança

2018-2019

# Dedication

I dedication this work to my family, without them it wouldn't be possible. They always support me and that has no price.

# Special Thanks

I thanks to my family that is always with me, even at distance, giving me the strength to continue, my parents were phenomenal and always find a way to help with what they could and couldn't. My siblings that at distance become closer. At last but not least, my grandmother that encouraged me in every important decision. The friends that I made here in Portugal specially The Special Group composed by Avatar, Isis, Stefany and Tamiris and the ones that I left in Brasil but I know that is for life. All of you are special in a single way. A special thanks to my teachers and mostly to my advisers, André Koscianki that introduce me in the academic research, André Pinz and Gleifer that even at distance helped a lot in this work, Paulo Leitão who I was always at contact and received me with opened arms at the Instituto Politécnico de Bragança.

# Abstract

Smart City uses emerging technologies to improve citizens' quality of life. A branch of this topic is the Smart Parking, where the parking system implements intelligent mechanisms to simplify to the searching of parking spots and consequently decrease the traffic of cars. This work proposes an architecture using Multi-Agent System (MAS), enhanced with some holonic systems principles, that is capable to be applied to different range of parking systems, e.g., considering trucks, cars, or bicycles.

Being a distributed architecture, a special attention is devoted to study the negotiation protocols that will regulate the behavior of autonomous and cooperative actors in the system, namely drivers and parking spots, during allocation process of parking spots to drivers. For this purpose, the Contract Net Protocol (CNP), English Auction, Dutch Auction and Faratin Auction were the tested, being the CNP the selected protocol for this problem. Also addressing the distributed nature of the system, some efforts were focused on the security of the messages exchanged between the agents was proposed using Secure Socket Layer (SSL).

The proposed multi-agent systems architecture was implemented using JADE (Java Agent DEvelopment Framework), which is a FIPA-compliant agent development framework that simplifies the development of agent-based applications. The exchange of messages follows the FIPA-ACL protocol using the CNP protocol for the negotiation. The communication between the agents and the User Interface is performed through the use of Message Queuing Telemetry Transport (MQTT) protocol.

**Keywords:** Smart Parking, Multi-Agent System, Architecture, Negotiation Protocols.

# Contents


**1 Introduction** **1**

1.1 Context . . . 1

1.2 Objectives . . . 3

1.3 Scope and Limitations . . . 4

1.4 Document Structure . . . 5

**2 Concepts, Technologies and Tools** **7**

2.1 Multi-Agent System . . . 7

2.2 Negotiation . . . 9

2.2.1 Contract Net Protocol Base . . . 11

2.2.2 English Auction Base . . . 11

2.2.3 Dutch Auction Base . . . 12

2.2.4 Faratin Auction Base . . . 14

2.3 JADE . . . 15

2.4 Message Queuing Telemetry Transport (MQTT) . . . 16

2.5 UML Diagrams . . . 18

2.6 Cryptography . . . 21

2.6.1 Rivest, Shamir, and Adleman (RSA) Algorithm . . . 22

2.6.2 Advanced Encryption Standard (AES) Algorithm . . . 23

2.6.3 SSL and Certificate Authorities . . . 24

**3 Architecture of the Smart Parking System** **26**
3.1 Base Architecture . . . . . 27
3.1.1 Use Case Diagram . . . . . 30
3.1.2 Class Diagram . . . . . 31
3.1.3 Activity Diagram . . . . . 36
3.2 Security Focused Architecture . . . . . 40
3.2.1 Proposal Roles . . . . . 40
3.2.2 UML Architecture . . . . . 41
3.3 Considerations . . . . . 43

**4 Negotiation Protocols** **44**
4.1 Assumptions . . . . . 44
4.2 Contract Net Protocol (CNP) . . . . . 45
4.3 English Auction . . . . . 47
4.4 Dutch Auction . . . . . 49
4.5 Faratin Auction . . . . . 51

**5 Comparative Analysis of the Negotiation Protocols** **53**
5.1 Simulations . . . . . 53
5.1.1 Scenarios . . . . . 54
5.2 Results . . . . . 56
5.2.1 Analysis of the Average Price Paid by the Driver . . . . . 57
5.2.2 Analysis of the Distance to the Desired Parking Place . . . . . 58
5.2.3 Analysis of Operating Parameters . . . . . 60

**6 Prototype Implementation** **62**
6.1 Base Architecture . . . . . 63
6.1.1 JSON . . . . . 64
6.1.2 Main Class . . . . . 66
6.2 Deployment of AgentSpot . . . . . 68

6.3 Make Reservation . . . 70
6.4 Request Reservations . . . 73
6.5 Discussion . . . 74

**7 Conclusions and Future Work** **75**
7.1 Conclusions . . . 75
7.2 Future Works . . . 76
7.3 Published works . . . 77

**A Original Project Proposal** **A1**

**B Base Use Case Diagram** **B1**

**C Base Class Diagram** **C1**

**D Base Activity Diagrams** **D1**
D.1 CRUD . . . D2
D.2 PAY . . . D5
D.3 DRIVER . . . D6
D.4 PARKING . . . D7

**E Security Class Diagram** **E1**

**F Deployment of AgentSpots demonstration** **F1**

**G Make Reservation demonstration** **G1**

**H Request Reservations demonstration** **H1**

# List of Tables

2.1 Simple substitution table example . . . . . . . . . . . . . . . . . . . . . . . . . . 23
2.2 Occurrence amount substitution table example . . . . . . . . . . . . . . . . . 24
2.3 Simple bit-by-bit table example . . . . . . . . . . . . . . . . . . . . . . . . . . 24

5.1 Amount of Drivers and Spots per scenario. . . . . . . . . . . . . . . . . . . . . 54
5.2 Driver Profiles. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 55
5.3 Spot Profiles. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 55
5.4 Average results related to the negotiation process for the four negotiation strategies part 1. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 60
5.5 Average results related to the negotiation process for the four negotiation strategies part 2. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 60

6.1 msg_id : "7". This message is sent by the AdmSpot, and have the following parameters inside a JSON. . . . . . . . . . . . . . . . . . . . . . . . . . . . 69
6.2 msg_id : "8". This message is sent by the new AgentSpot, and has the following parameters inside a JSON. . . . . . . . . . . . . . . . . . . . . . . 69
6.3 msg_id : "1". This message is sent by the Driver, and have the following parameters inside a JSON. . . . . . . . . . . . . . . . . . . . . . . . . . . . 71
6.4 msg_id : "2". This message is sent by the AgentDriver to the Driver, and have the following parameters inside a JSON. . . . . . . . . . . . . . . . . 72
6.5 msg_id : "3". This message is sent by the Driver, and have the following parameters inside a JSON. . . . . . . . . . . . . . . . . . . . . . . . . . . . 72

6.6 msg_id : “4”. This message is sent by the AgentDriver to the Driver, and have the following parameters inside a JSON. . . . . . . . . . . . . . . . . . 72
6.7 msg_id : “5”. This message is sent by the AgentDriver to the AgentSpots, and have the following parameters inside a JSON. . . . . . . . . . . . . . . . 74
6.8 msg_id : “6”. This message is sent by the AgentSpots to the AgentDriver, and have the following parameters inside a JSON. . . . . . . . . . . . . . . . 74

# List of Figures

1.1 Overview of the work. . . . 5

2.1 Intelligent Agent schema adapted from [7]. . . . 8
2.2 CNP Sequence Diagram of messages [16]. . . . 12
2.3 English Auction Sequence Diagram of messages [16]. . . . 13
2.4 Dutch Auction Sequence Diagram of messages [16]. . . . 14
2.5 JADE Architecture [25]. . . . 17
2.6 Use Case Diagram example in UML diagrams. . . . 18
2.7 Class Diagram example in UML diagrams. . . . 19
2.8 Activity Diagram example in UML diagrams. . . . 20
2.9 Sequency Diagram example in UML diagrams. . . . 21

3.1 Overview of the Architecture. . . . 26
3.2 Architecture of the smart parking. . . . 27
3.3 Part of the Class Diagram focused on the enumerate Classes. . . . 32
3.4 Part of the Class Diagram focused on the abstract class 'person' (Real person in the system). . . . 32
3.5 Part of the Class Diagram focused on the customer (Driver). . . . 34
3.6 Part of the Class Diagram focused on the parking spots. . . . 35
3.7 Part of the Class Diagram focused on the agents . . . . 35
3.8 Part of the Class Diagram focused on the agents . . . . 41

4.1 Components of the Negotiation Protocols in a Smart Parking System. . . . 45

4.2 CNP Sequence Diagram of messages based on the FIPA standard [16]. . . . 46
4.3 English Auction Sequence Diagram of messages based on the FIPA standard [16]. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 48
4.4 Dutch Auction Sequence Diagram of messages based on the FIPA standard [16]. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 50
4.5 Faratin Auction Sequence Diagram of messages based on the article [23] . . 52

5.1 Results of the price paid by the Driver for the four negotiation protocols divided per scenario. . . . . . . . . . . . . . . . . . . . . . . . . . . . 57
5.2 Results of the price paid by the Driver for the CNP and CNPDist protocols divided per scenario. . . . . . . . . . . . . . . . . . . . . . . . . . . . 58
5.3 Results of the distance to the desired parking place for the four negotiation protocols divided per scenario. . . . . . . . . . . . . . . . . . . . . . . . 59
5.4 Results of the distance to the desired parking place for the CNP and CNPDist protocols divided per scenario. . . . . . . . . . . . . . . . . . . . . 60

6.1 Chapter related to the work overview. . . . . . . . . . . . . . . . . . . . . . 62
6.2 Configurations to connect to some MQTT in MQTTBox. . . . . . . . . . . 65
6.3 CNP and MQTT protocols interactions. . . . . . . . . . . . . . . . . . . . . 70

B.1 Use case diagram. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . B1

C.1 Class diagram. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . C2

D.1 Create. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . D2
D.2 Delete. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . D2
D.3 Read information and parking spot. . . . . . . . . . . . . . . . . . . . . . . D3
D.4 Read vehicle and promotion. . . . . . . . . . . . . . . . . . . . . . . . . . D3
D.5 Update information and parking spot. . . . . . . . . . . . . . . . . . . . . D4
D.6 Update vehicle and promotion. . . . . . . . . . . . . . . . . . . . . . . . . D4
D.7 Pay. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . D5

D.8 Cancel a reservation. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . D6
D.9 Define default specification for parking spot allocation. . . . . . . . . . . . D6
D.10 Negotiate parking spot. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . D7
D.11 OCCUPY PARKING SPOT. . . . . . . . . . . . . . . . . . . . . . . . . . . D7
D.12 RELEASE PARKING SPOT. . . . . . . . . . . . . . . . . . . . . . . . . . . D8
D.13 GENERATE REPORT. . . . . . . . . . . . . . . . . . . . . . . . . . . . . D8

E.1 Class diagram focused in security. . . . . . . . . . . . . . . . . . . . . . . . E1

F.1 Initial and final using JADE GUI. . . . . . . . . . . . . . . . . . . . . . . . F1
F.2 Message flow. Read from bottom to top. . . . . . . . . . . . . . . . . . . . . F2

G.1 Middle state using JADE GUI. Notice, the carDriverAgent is in another container. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . G1
G.2 Message flow. Read from bottom to top. . . . . . . . . . . . . . . . . . . . . G2

H.1 read from bottom to top. . . . . . . . . . . . . . . . . . . . . . . . . . . . . H1

# Glossary

**AES** Advanced Encryption Standard. 23–25

**AI** Artificial Intelligence. 2, 7, 30, 31, 76

**CA** Certificate Authority. 3, 24, 25, 28, 40–42

**CM** Certificate Manager. 40–43

**CNP** Contract Net Protocol. 3–5, 11, 45, 49, 53, 55–59, 61, 62, 70, 73, 75, 76

**CNPDist** CNP Distance. 55, 56, 58, 59

**FIPA** Foundation of Intelligent Physical Agents. 3, 4, 11–13, 15

**IDE** Integrated Development Environment. 15

**IEEE** Institute of Electrical and Electronics Engineers. 4

**JADE** Java Agent Development Framework. 3, 4, 15, 63, 66, 75

**JSON** JavaScript Object Notation. 6, 63–65, 68, 70, 73

**MAS** Multi-Agent System. 2–7, 15, 27–29, 74–76

**MQTT** Message Queuing Telemetry Transport. 3, 4, 6, 16, 27, 28, 41, 63, 65, 66, 68, 70, 73, 76

**QoS** Quality of Service. 65, 66

**RFID** Radio-Frequency IDentification. 39

**RSA** Rivest, Shamir, and Adleman. 21–23, 25

**SSL** Secure Socket Layer. 3, 25, 26, 28, 40, 41

**UML** Unified Modeling Language. 18, 26, 75, 76

# Chapter 1

# Introduction

This Chapter intends to contextualize the problem of Smart Parking Systems, state the objectives of this work, and present the structure of the document.

## 1.1 Context

Nowadays, the cities are becoming very large, with the number of urban residents expanding every year. By 2050, 68% of the world's population is expected to be living in cities [1], leading to the emergence of some problems, namely pollution, waste, and traffic. To make the citizens' daily life more comfortable and convenient, Smart City [2] emerged. Smart City are cities that use emergent technologies to provide access to public information and services [3]. Particularly, citizens will get access to advanced facilities, for example, smart transportation facilities, smart electricity systems and smart applications for governance.

In terms of traffic, the parking problem can be crucial for the improvement of the Smart City concept. For instance, drivers expend almost 6 minutes trying to find a spot to park their vehicles in England [4]. In this way, parking that uses advanced technologies to improve its management and the provided services can contribute to a reduction of the traffic. Additionally, it might lead to a better profit for the parking.

On the other hand, Smart Parking Systems are not easy to build due to their dynamic and sometimes chaotic environments. In fact, the system needs to be able to deal with

a significant amount of drivers asking and receiving offers for parking spots, and at the same time, the parking needs to decide which requests will satisfy more both sides. Due to its dynamics, large-scale and often chaotic nature, the use of distributed systems can be helpful, being easier to divide the entire complex process into simpler micro-processes than having a single entity in charged by the entire process [5].

To do that, intelligent agents can be used to solve this problem. That is an entity in a system capable of being autonomous, communicate to other agents and possibly humans, react to the environment and take the initiative to reach this some goal [6]. In other words, it is an entity that can perceive and react to the environment using Artificial Intelligence (AI) (software capable of act or think humanly, or capable of act or think rationally [7]).

In particular, Multi-Agent System (MAS) [8] offers an alternative way to design such systems, by distributing the intelligence and control over a community of autonomous and cooperative agents that will cooperate to achieve their objectives. In the Smart Parking problem, the agents will represent the parking spots available in the system and the drivers of the vehicles, such as cars, bicycles and trucks. The use of MAS solutions provide scalability and flexibility. For example, the system continues operating under condition even with the increase of the number of agents, and also, the system can be adapted for different use cases, such as a car parking, a bicycle parking or considering both cases in the same system.

A centralized monolithic solution is a usual approach for the Smart Parking System. Nonetheless, recently the adoption of MAS is being reported in the literature (see for example, [9] and [10]). However, in these works, the negotiation among the agents follows a centralized approach, which simplifies its implementation but limits the use of the MAS potentialities. In fact, during the negotiation among the agents to find a consensus, each type of agent has its objectives that usually are in conflict: drivers want to pay as little as possible and the parking spots want to receive as much as possible. To reach an agreement a negotiation protocol might be used [11].

In a MAS many agents may interact with each other to achieve a consensus in negotiations. In a Smart Parking the agents have different goals, to get the highest (driver)

and the lowest (parking) price. That can affect the negotiation strategy and consequently the system performance. The negotiation needs to be simple and fast enough and yet show a good balance for both the driver and the parking perspectives. A comparison between some negotiation protocols was made simulating the system with the Contract Net Protocol (CNP), the English Auction, the Dutch Auction and the Faratin Auction, following the Foundation of Intelligent Physical Agents (FIPA) standards. That said, this work also concerns about communication security and proposes, as an option, the use of Secure Socket Layer (SSL) and Certificate Authority (CA) to improve the security of message exchange between agents.

A way to create a MAS is using Java Agent Development Framework (JADE), a JAVA framework capable of creating agents and manage their communications. This framework may be responsible for the agents in the system, creating, deploying, managing the communications, and deleting them.

But, not only the communication between agents is necessary, some interface between the agents and the driver is needed. Message Queuing Telemetry Transport (MQTT) is a message protocol capable of making the agents communicate with some mobile applications for instance [12].

## 1.2 Objectives

The main objective of this work is to study negotiation protocols in a Smart Parking System based on MAS. For that end, an architecture for a Smart Parking System using MAS was proposed. A module for the architecture focused on the security of the system was also proposed. As we have mentioned, in a MAS the agents exchange a lot of messages to achieve their goals. Therefore, one of the concerns of security is the exchange of such messages. That is why we have defined how SSL and CA can be used to improve the security of the system.

Furthermore, a comparison of negotiation protocols for consensus problems in Smart Parking Systems was made to find the one who gets the best prices and expends fewer

resources. In particular, the CNP, the English Auction, the Dutch Auction and the Faratin Auction were chosen to make the comparison. These strategies were implemented under an agent-based Smart Parking System using JADE [13] [14], an agent-based framework for JAVA [15]. To create hem, the interaction protocols defined by the FIPA [16] (standards accepted by the Institute of Electrical and Electronics Engineers (IEEE) [17]) were used, and tested according to different scenarios. The evaluation has considered the level of satisfaction of the actors in the system based in the price paid by the Driver to reserve a parking spot and the distance between the desired parking place and the parking spot got by the driver, the scalability and the negotiation time.

Using the best protocol, a prototype of the Smart Parking System, based on the proposed MAS-based architecture and covering also the communication between the agents and the mobile user interfaces using MQTT, was implemented and tested.

As a Smart Parking architecture is composed of several parts, the focused here is the study of negotiation protocols for Smart Parking Systems. Furthermore, the work proposes a message exchange method and has a security concern. Those modules might be used in further works to create, study or implement a MAS.

## 1.3 Scope and Limitations

This work cares about the structure of the Smart Parking System and has its focus on the negotiation protocols. It covers the architecture and the prototyping of the System. It's assumed an interface with the Driver and with the gate on the spot. Furthermore, the work is based on simulations, that is no physical implementation of the system has been made.

Moreover, the project assumes a flat system, which means no hierarchy at any agent level (even in the security module), a Smart Parking System for cars only that can be easily adapted for other vehicles and no 24h parking is considered.

## 1.4 Document Structure

This work has three main modules represented in Figure 1.1. The MAS is the central part of this work and is explained in Chapter 2 being the base of the proposed Smart Parking System. Moreover, the modules are the *Architecture*, *NegotiationProtocols* and the *Prototype*, and one extension of the *Architecture*, the *Security* module.

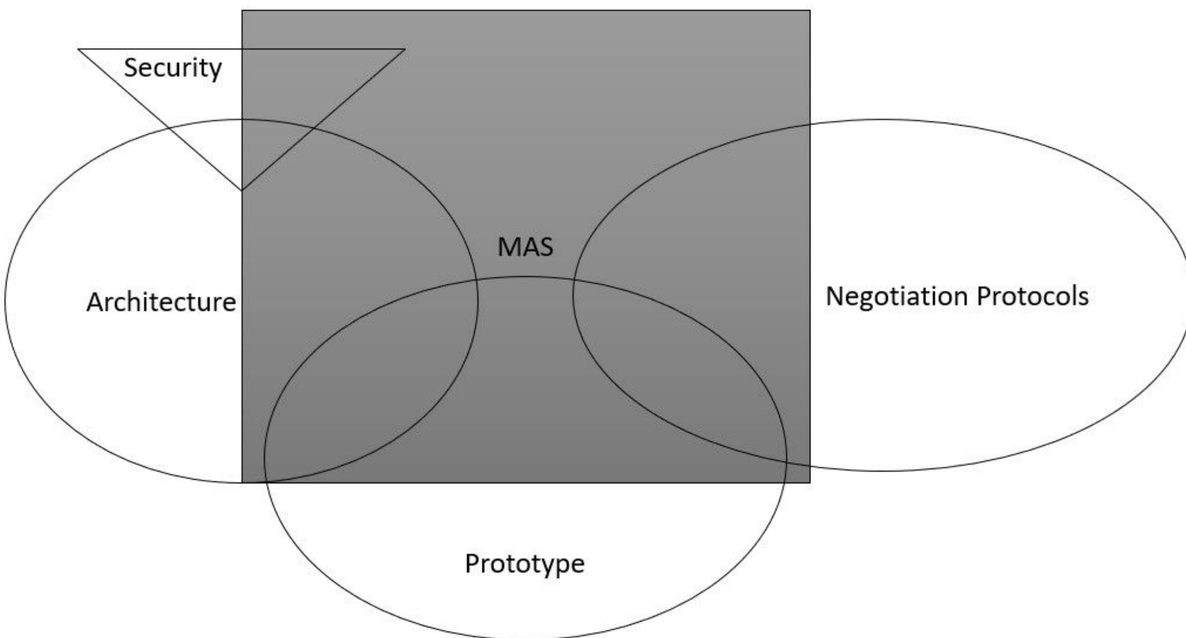


Figure 1.1: Overview of the work.

Each one of the modules will be the focus of this paper in one Chapter. The architecture of a Smart Parking System can be seen in the Chapter 3 as long with a security approach for these systems, it's intended to propose an architecture capable of being easily adapted for other scenarios then car parking, for instance, a bicycle parking or a holonic approach instead of a flat system.

The negotiation protocols in MAS systems are explained in Chapter 4 and 5, namely the CNP, English Auction, Dutch Auction and Faratin Auction. It will explain how to adapt the protocols for a Smart Parking System and will also make a comparison between them to find the best one for this scenario. On the other hand, an initial proposal on

how to build a Smart Parking System is exposed in Chapter 6 emphasizing the communication with the interface using MQTT and JavaScript Object Notation (JSON) for the standardize the data.

The remainder of this paper is organized as follows. Chapter 2 presents the concepts, technologies, and tools needed for a better understanding of the work. Chapter 3 explains the architecture presented here. Not only that, this Chapter proposes a module to improve the security of the system. Chapter 4 explains the protocols for the case of a Smart Parking System and how they were used in this work. Chapter 5 explains the protocols studied, shows how the tests were done and made a brief conclusion on that subject. Chapter 6 explains how the prototype was developed, focused on the message exchanged between the MAS and the user interface. Chapter 7 Makes an overview of the previous chapters and propose future works to improve the system.

# Chapter 2

# Concepts, Technologies and Tools

This chapter intends to explain some concepts and present some patterns used in this work. To get a better understanding of it, but a minimal knowledge of AI and object orientation is required, such as the difference between object and class.

## 2.1 Multi-Agent System

MASs are computer systems that have intelligent agents communicating with each other to find a solution to the problem [18].

An intelligent agent is a computer system that understands the environment through its sensors and acts through its actuators by itself to reach a previous goal [7]. In other words, it is a system capable of perceiving the environment and act on it without an explicit order.

An example is a system capable of turn on and off a cooler depending on its temperature. The system will need to perceive the temperature by itself and just with that information will choose if the cooler will be turned off or continues to work. The mechanism capable of turn on and off the cooler is the actuator on the system and the temperature sensor is the sensor of the system. The Figure 2.1 represents a generic agent.

As shown in [18], intelligent agents have some important characteristics, they are:

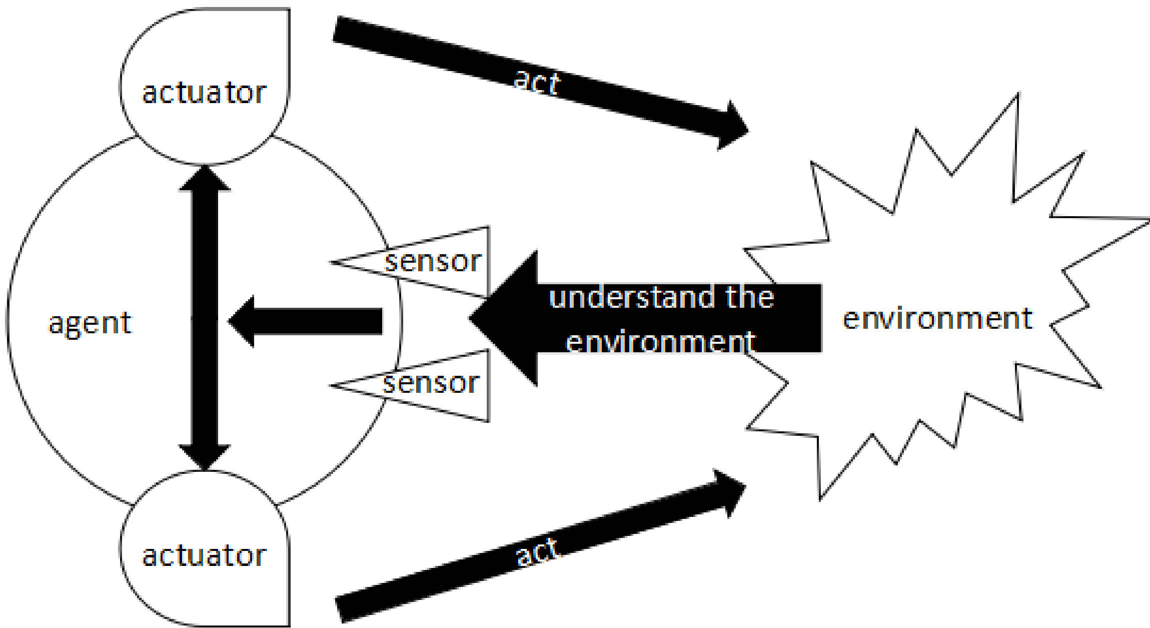


Figure 2.1: Intelligent Agent schema adapted from [7].

- Reactivity: Ability to understand the environment and act to events in a short time. For example, a smart car should stop a car if a tree falls in front of it.

- Proactivity: Ability to initiate something. For example, from time to time a satellite should take a picture of the Earth.

- Social ability: Ability to communicate with other agents (and possibly humans). For example, an alarm should be able to communicate with the coffee maker to prepare the coffee when the alarm rings.

As can be seen, the agent is a system capable of choosing what to do by itself. It is an autonomous system. That, in the case of a Smart Parking, is an advantage, because it can choose the best prices for the parking and the Driver in negotiations. When the parking is trying to get a higher price, the Driver wants to pay the lower price possible for the best spot.

Furthermore, the author also proposes three kinds of agents based on these abilities:

- Deductive reasoning agents: Agents that simply react to the environment. Can make more than one decision in a moment.
- Practical reasoning agents: Agents capable of simulating the environment in a memory. They use this memory to reach their goals.
- Reactive and hybrid agents: This is the union of the two types previously explained. They can plan and react fast enough to events.

For this work, the reactive and hybrid agent is the better option, since the parking is always receiving new visitors, so it needs to adapt fast enough to make a good offer. But also, it has time to calculate the best offer and decide which strategy use for the usual customers.

In the case of a Smart Parking, the multiagent system is an option to improve its performance since a person can just don't see a spot or even a Smart Parking at all. Furthermore, the price using this kind of system can be defined by the system itself, that is, the multiagent system will search for a parking spot with a price fair enough for the Driver and the parking.

The intention, in this case, is to get more customers for the parking and consequently more profit, help the Driver to get the lowest price for a parking spot, the fastest possible and with that agility, the traffic is supposed to decrease as well.

## 2.2 Negotiation

The interactions between agents have the objective to persuade other agents to execute some action, modify its action plan and make a deal [19]. For this to happen, a negotiation needs to occur. Negotiation can be defined as the effort made by two or more entities to achieve an agreement benefiting themselves [19]. As [18] says, there are four components in a negotiation:

- Negotiation set: Possible actions the agent can make. For example, the drive can search for a spot, refuse an offer or accept an offer and take the spot.

- Protocol: Possible actions the agent can make after the previous one. For example, after receives an offer, the Driver can accept it and take the spot or refuse it.

- Collection of strategies: Each agent has one and defines what the agent will do. For example, the car will try to get a spot closer to his destiny, no matter the price.

- Rule: Defines when the negotiation is over and what is agreed upon. For example, a negotiation time is defined, if it ends without an agreement between the agents, the Driver may try again with other parameters.

Furthermore, the negotiation between agents can be one of three types [18]:

- One to one: Negotiation between two agents.

- Many to one: An agent negotiating with more than one at the same time.

- Many to many: A lot of agents negotiate with a lot of agents at the same time. This model is hard to control and implement. In the worst-case n(n-1)/2 threads are needed at the same time.

In this work, the many to one negotiation was chosen because it is more credible in a Smart Parking System to happens. A single Driver negotiating with several spots.

To negotiate with several agents at the same time, an auction can be used. As [18] explains, there are many auctions and they can be divided by the number of participants in each side, that is the number of sellers and buyers and, also by the number of items in the negotiation as explained above:

- Unilateral negotiation: Only one agent is a seller, all the others are buyers. Or only one agent is a buyer and all the others are sellers.

- Bilateral negotiation: The model accepts more than one buyer and more than one seller at a time.

- Single item negotiation: Only one item is negotiated at a time.

- Multiple items negotiation: More than one item is negotiated at a time.

This work uses unilateral auctions and single item negotiations. The system is divided into two main groups, the buyers (Drivers) and the sellers (spots) and each negotiation occurs between a single buyer and several sellers trying to sell a single reservation.

In the literature, several negotiation strategies can be found, namely the CNP [20], English Auction [21], Dutch Auction [22] and Faratin Auction [23]. These four mechanisms have different characteristics and, the selection of the best one is dependent of the system requirements and application scenario.

### 2.2.1 Contract Net Protocol Base

The CNP auctions are examples of one-shot auctions and are the simplest auction we will consider. It consists of a single round where the best offer wins.

1. The auctioneer announces the auction.
2. The interested Drivers makes a proposal.
3. The auctioneer selects the best offer.

The FIPA has a standard for it be used by agents. As can be seen in Figure 2.2, the *Initiator* starts the auction sending a message for all the participants. After, the *Participants* can deny it or send an offer. When the *Initiator* receives all the answers, it selects the best one, informing all agents if it is the winner or not. The winner can now confirm, deny it or inform some problem.

### 2.2.2 English Auction Base

The English Auction is the most commonly known type of auction [18]. It consists of several rounds where the offers will become, in each turn, better for the auctioneer.

1. The *Initiator* announces a start value (bad for him and good for the Drivers).

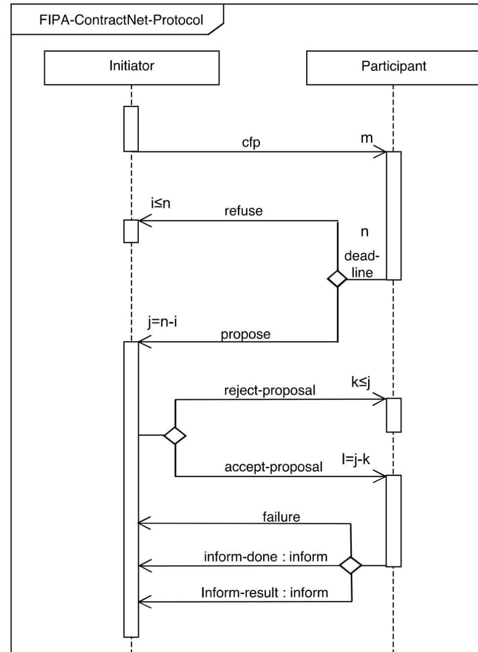


Figure 2.2: CNP Sequence Diagram of messages [16].

2. The interested *Participants* make an offer.

3. The *Initiator* changes the start price, making it better for himself, and offers again to all *Participants*.

4. The auction ends when no one else makes a new offer or the time is up.

The FIPA has a standard for it be used by agents. As can be seen in Figure 2.3, the *Initiator* starts the auction sending a message for all the participants and after, it requests the first offer informing the start price. The *Participants* can deny it or offer something. Now, the *Initiator* must decide if the auction is over or not. That cycle continues until no *Participant* sends a new offer or the time is up. When it's over, the *Initiator* informs the winner and the losers.

### 2.2.3 Dutch Auction Base

The Dutch Auctions are very similar to the English Auction. The main difference is the changes in each offer. That is, while in the English Auction in each turn the *Initiator*

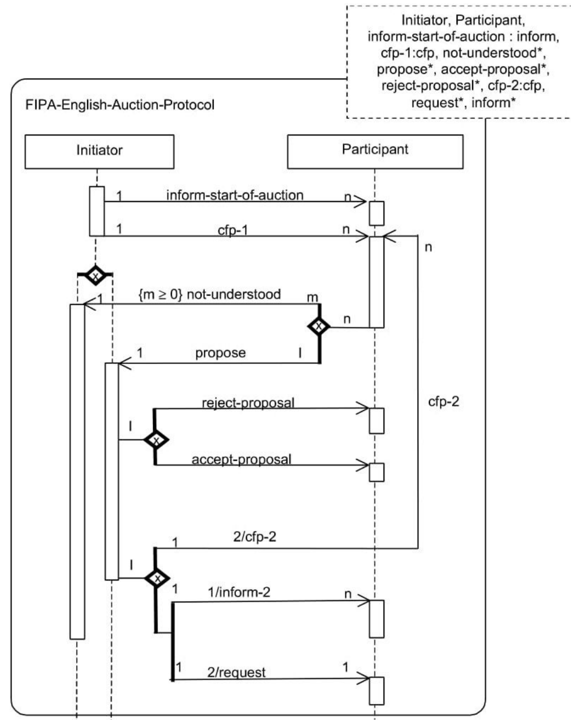


Figure 2.3: English Auction Sequence Diagram of messages [16].

increases the value of the offer for itself, in the Dutch Auction the negotiation starts with a very good value for the *Initiator* and in each turn it decreases the value of the offer for itself, but, at the same time, improves the value of the offer for the *Participants*.

1. The *Initiator* announces a start value (good for him and bad for the Drivers).

2. The interested *Participants* make an offer.

3. The *Initiator* changes the start price, making it better for the Drivers, and offers again to all Drivers.

4. The auction ends when one *Participant* makes an offer or the time is up.

The FIPA has a standard for it be used by agents. As can be seen in Figure 2.4, the *Initiator* starts the auction sending a message for all the *Participants* and after, it sends the first initial value. The *Participants* can deny it or make an offer. Now, the *Initiator*

must decide if the auction is over or not. That cycle continues until one *Participant* sends an offer or the time is up. When it's over, the *Initiator* informs the winner and the losers.

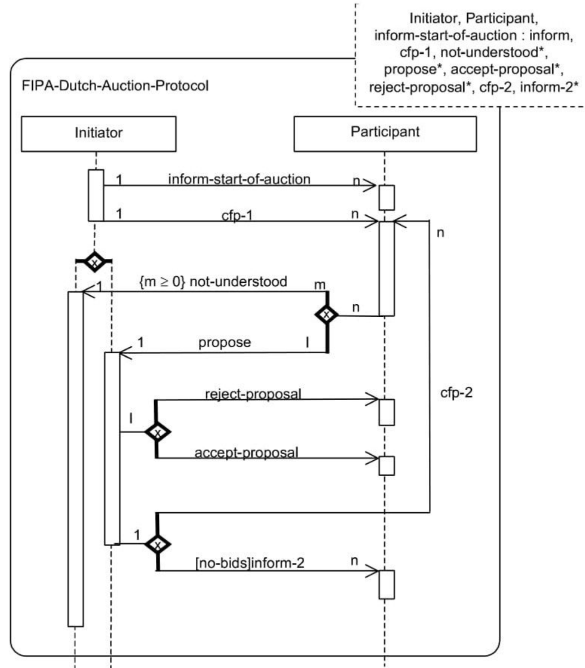


Figure 2.4: Dutch Auction Sequence Diagram of messages [16].

### 2.2.4 Faratin Auction Base

The Faratin Auction is the most different auction that will be considered. It consists of several rounds where the *Initiator* and the *Participant* will send an offer and sends a counteroffer. This cycle will be repeated until one of the sides agree with the offer received. Notice, this is a one to one negotiation.

1. The *Initiator* makes an offer.

2. The *Participant* accepts or sends a counteroffer.

3. The *Initiator* accepts or sends a counteroffer increased.

4. The auction ends when some of them accept an offer.

The FIPA hasn't a standard for it. In this case, an adaptation to be used by agents was made. The *Initiator* sends the information about the auction to the *Participants*. Each sent message will be a separated auction occurring in parallel. After, the *Initiator* sends the first offer. At the first time the *Participant* receives an offer, it can choose to not participate in the auction. Otherwise, it will accept or send a counteroffer. The *Initiator*, when receives the offer can also accept or makes a counteroffer. These cycles go on until one of them accepts the received offer or the time of the auction is up.

Notice, the offers are converging to a middle value. This conversion can be based on some strategies, like the time left to complete the auction or the offers made by the other, imitating it.

As mentioned before, maybe several auctions occurring in parallel for the same resource. The first one that reaches an agreement wins, interrupting the other auctions.

## 2.3 JADE

All this work was made with JAVA version 1.8.0_221 [15] using the Integrated Development Environment (IDE) Eclipse Photon [24]. That's because JAVA is a general propose language, supports object orientation and is well documented.

JADE is an agent-based framework that facilitates the implementation, debugging and maintenance of agent-based solutions by offering services like the white and yellow pages and the sniffer agent fully implemented in the JAVA language. This framework focused on the creation of MAS. It works like a middle-ware that complies with the FIPA specifications [16]. Furthermore, it has graphical tools that help to debug and deploy the system.

JADE has 6 main features in its architecture:

- **Agent** In the JADE environment it can namely, execute tasks and interact by exchanging messages.

- **Platform** The cyber environment created to the agents to live, that provides them

with basic services such as message delivery.

- **Container** A platform can be subdivided into one or more containers and, each one of them can be running on a different host. In that way, a distributed system can be created.
- **Main Container** It is a special container that must be on the platform. It must be the first to be created and it has two special agents created automatically: AMS and DF.
- **AMS** The authority agent in the platform. It is the only agent capable of management actions, such as create and kill other agents. But, other agents can ask this kind of action of it.
- **DF** The agent that provides the Yellow Pages service where agents can publish the services they provide and find other agents providing the services they need.

In Figure 2.5 a schema of the architecture can be seen. There are two platforms, two independent systems. The platform 2 has one container, the main container, and one agent (A5), beyond the AMS and DF. The second system in platform 1 has 4 agents beyond the DF and AMS. Not only that, but the system has three different hosts. They can be, for example, a windows system, a Linux system, and a MAC system.

One more important thing to be noticed is the Platform 1 also has a Main container since it is another system. At last, the systems can communicate with each other by the network, but notice the names of the agents are unique, even in different platforms to communicate properly.

## 2.4 Message Queuing Telemetry Transport (MQTT)

MQTT is an open publish/subscribe protocol. It is designed in such a way that its implementation on the client's side (agents) is very simple. All of the system complexities

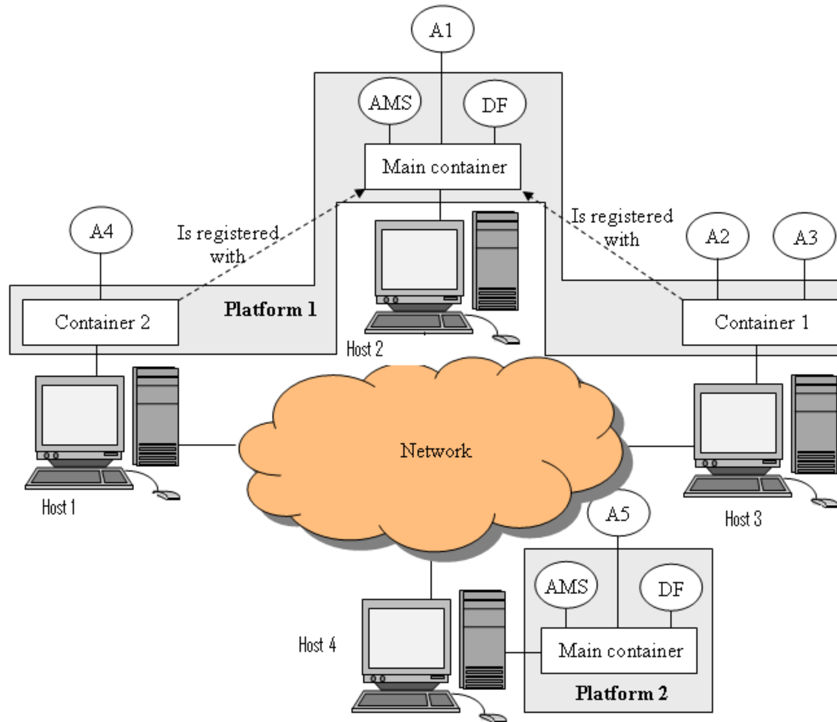


Figure 2.5: JADE Architecture [25].

reside on the broker's side [12]. It has four main concepts. Each agent in this work is a subscriber and a publisher.

- Topic: A channel that contains a list of subscribers. When a message is received, all those subscribers are notified.

- Message: Message sent by a publisher to some topic. The sender does not know the subscribers.

- Publisher: The entity that publishes a message to a topic.

- Subscriber: The entity that registers itself in a topic.

## 2.5 UML Diagrams

Unified Modeling Language (UML) is a standard language to model systems. It is used to describe the system and to facilitate the comprehension of the system by another person. Moreover, it documents the system. Those diagrams were the base for the architecture presented in this work. Since at the beginning of this project the architecture was not the priority, just some of the diagram was made. They were chosen when the need for some specification was needed. The used diagrams were:

- *Use Case Diagram:* The Use Case Diagram describes the functionalities of the system. It is useful to get the functional requirements of the system. It shows what the system can do or not. This diagram is composed of some entities. The most important are the Actors and the Use Cases [26].

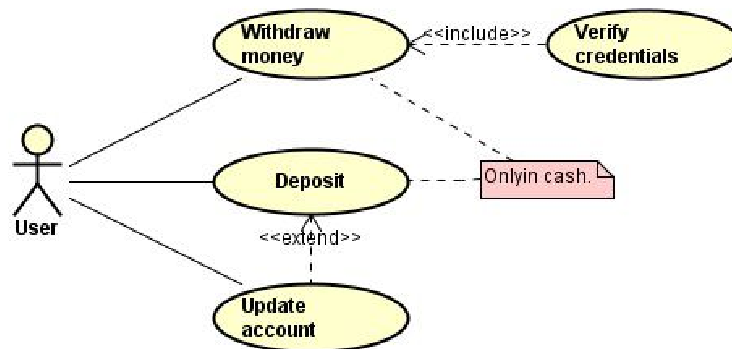


Figure 2.6: Use Case Diagram example in UML diagrams.

The Figure 2.6 is a full example of a Use Case Diagram. It represents a simple cash machine, where the User can Deposit money, Update the account and Withdraw the money. In more detail, when the account is updated, it can deposit money after that, instead of restart the system. Also, when the User Withdraw money it needs to verify its credentials. At last, both, deposit and withdraw money only work with cash.

- *Class Diagram:* The Class diagram represents the Objects needed to be represented

in the system, that is, the pattern used to represent it inside the system. Each class is represented by a box divided into three. The first section contains the name of the class. The second contains the attributes of the class. And, the third, contains the methods of the class, that is, the actions it can perform [26].

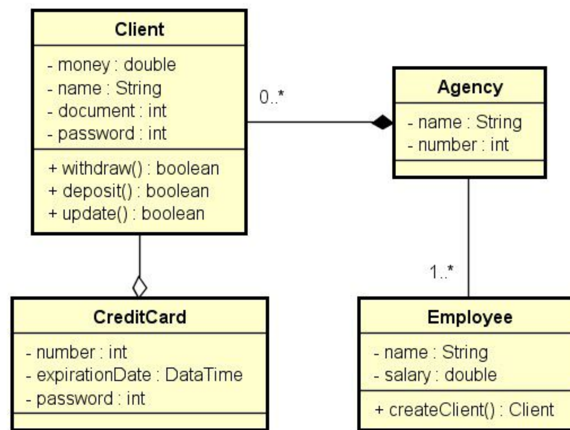


Figure 2.7: Class Diagram example in UML diagrams.

The Figure 2.7 is an example of Class Diagram. It represents a simple Bank that has 'Agencies' and each one of them has one or more 'Employee'. Notice that the 'Client' can only be created if the 'Agency' exists. Not only that but when a 'Client' is created a 'CreditCard' is also created.

- *Activity Diagrams:* Activity Diagrams describes in a flow chart the possible paths the system might take for each Use Case. In that way, a better understanding of the data flow is obtained. Not only that, the need for Objects and Actors are exposed.

  Figure 2.8 represents a simple Activity Diagram example of a account creation. The first action must be the request of the creation by the Client, then, the Agency requests the information that is given by the Client. Some validation of that information is made and depending on the result the agency can create the account (that is another Diagram) and inform it or just inform that the account couldn't be created.

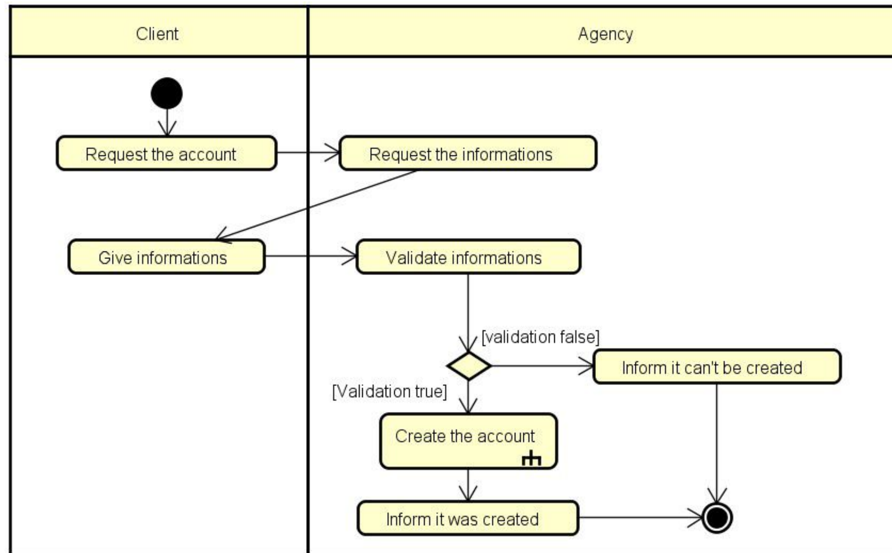


Figure 2.8: Activity Diagram example in UML diagrams.

- *Sequence Diagram:* While the Activity Diagram focuses on the application possibilities, the Sequence Diagram focus on a single process.

  In the Figure 2.9 is possible to see all those elements. It represents a communication process with authentication. First, the 'agent a' sends a request to the 'agent b' to verify if it exists. When 'agent a' receives the message it requests the credentials. When the credentials arrived it makes a validation. If the credentials are invalid it ends the communication. If the credentials are valid it confirms the credentials and starts a dialog, sending and receiving messages. When the 'agent a' wants to close this communication channel it sends s message to end the communication.

Last but not least, a simplification on some Diagrams can be done. The Use Case Diagram may contain the word 'CRUD', it means the operations of 'Create', 'Read', 'Update' and 'Delete'. But it is only used when all the operations refereed. If the entity, for example, can 'Create', 'Read' and 'Update' but can not 'Delete', it will be expressed by a comment or in three separate entities in the Diagram.

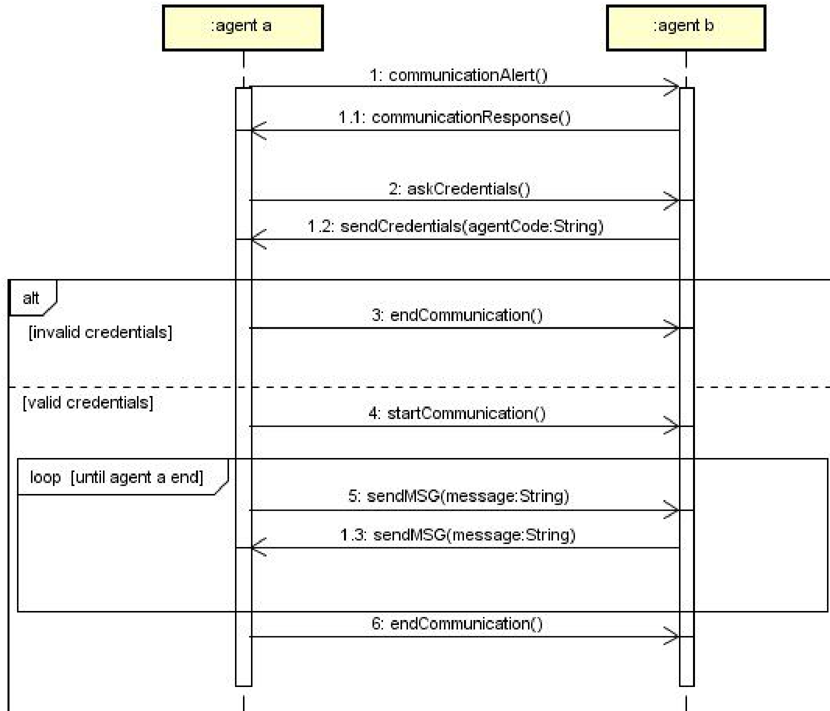


Figure 2.9: Sequency Diagram example in UML diagrams.

## 2.6 Cryptography

In 1978, Rivest, Shamir, and Adleman discovered the first practical public-key encryption and signature scheme, now referred to as Rivest, Shamir, and Adleman (RSA) [27]. And based on that, in 1991 the first international standard for digital signatures (ISO/IEC 9796) was created [28]. This protocol is based on the goals of the cryptography, explained in [28]. They are:

1. Confidentiality: A service that does not allow others but the ones previously authorized to see the content of the information. For example, physical protection to mathematical algorithms, which render data unintelligible, can provide confidentiality.

2. Data integrity: A service that prevents the data to be modified by an unauthorized person. To assure that, the detection of an unauthorized change in the data must be identifiable.

3. Authentication: A service related to the identification. Both sides must be able to identify the other when it receives a message. The information about the data should also be authenticated. Usually divided into two major classes: entity authentication and data origin authentication.

4. Non-repudiation: A service that ensures all the commitments or actions. When an entity tries to deny some previous action, a third trustworthy party is needed to verify and confirm the operations.

### 2.6.1 Rivest, Shamir, and Adleman (RSA) Algorithm

To achieve some of those goals, the RSA uses asymmetric (public) keys. In this scenario, every entity that can communicate with others must have two keys: one public key and private key. The second key must be kept secret. In RSA a pair of numbers, for instance, $(p, q)$, is given to everyone who wants to send a message for an agent $A$. Then, the message $M$ is converted to numbers and an operation is made as in equation 2.1 to encrypt it $(MI)$ [27]:

$$MI = M^p * (modq) \tag{2.1}$$

To decrypt the message, another pair of value is needed, for instance, $(d, q)$, called private key. Notice, the second value $q$ is the same. This key must be secured and not be shared, because, only with that, MI can be decrypted. In order to do that, the equation 2.2 is used.

$$M = MI^d * (modq) \tag{2.2}$$

How the RSA keys are chosen will not be explained in this work, since the objective is not to implement it, but understand how it works.

Using the RSA is possible to secure the message confidentiality and Data integrity. But is wise to take note that this method is not perfect. The pair of keys to decrypt

can be found, but it‘s necessary a lot of processing, may be taking even years to do it. To prevent that the keys must have an expiration date, that is, changing it from time to time.

In the end, if someone agent wants to speak with another, it needs to know the public key of it. In that way, only who knows the private key will be able to open that message, that is, the agent receiver.

### 2.6.2 Advanced Encryption Standard (AES) Algorithm

Just like the RSA algorithm, the Advanced Encryption Standard (AES) encrypts a message. Using the same key to encrypt and decrypt. This kind of key, capable of encrypting and decrypt, is called *symmetric*. The base of the algorithm is to use crypto primitives (reversible operations) [29].

The primitive operations are substitution, transposition, and bit-by-bit operation. The substitution operation substitutes a part of an array with other expressed in a table. This table can be one to one or one to many, for example, if there is a letter $a$ it can be in the table $b$ or have more then one option like $b$ and $g$. It doesn't matter the chosen one, it will be always possible to come back to the original text if the table is known. Another possibility is to have a table that shows the substitution value if the letter is found one time, then if it's found for the second time, and so on, like $a$ becomes $g$, but if it's found $a$ again, it becomes $p$. Those tables can be seen in table 2.1 and 2.2

| Input | Output |
|---|---|
| a | b |
| b | k |
| d | 5 |

Table 2.1: Simple substitution table example

The transposition operation represents the change of the order. For example, *abc* can becomes *bca* or *cba* or any other combination with the elements. While the bit-by-bit operation is a predetermined operation, based on a table and an array. The entered array (part of the message to be encrypted) makes a bit-by-bit operation with the previously

| Times | Input | Output |
|---|---|---|
| 1 | a | r |
| 2 | a | g |
| >2 | a | 7 |

Table 2.2: Occurrence amount substitution table example

defined array based on the table. For example, if a in the input array is 001101, the previous array is 101010 and a table of operations is the 2.3, it should result in the array 100111.

| | 0 | 1 |
|---|---|---|
| 0 | 0 | 1 |
| 1 | 1 | 0 |

Table 2.3: Simple bit-by-bit table example

With just those primitive operations the AES is a simple but very robust algorithm of encryption. Its schema follows the steps [30]:

1. Split the message into blocks of 128 bits and encrypt each one of those blocks.
2. Subkeys are generated based on the algorithm key for each operation in each round.
3. A XOR operation is made with one subkey and the block.
4. Each block goes for 10 rounds of encryption in a pipeline.
5. In each round, a series of operations are done as follows: substitution, transposition, substitution, and a XOR operation.

In the end, the blocks are united again generating a new encrypted text that can be decrypted using the same key and the exact reverse sequence described before.

### 2.6.3 SSL and Certificate Authorities

To guarantee that no outsider enters the system, the CA certificates all the entities inside the system. It is a database of certificates. The system can have more than one CA, each

one having a copy of the database. In that way, the system may support failures. The CA contains information about the entities, like their version, serial number, and the public key. Based on that, it generates a certificate for the entity.

The SSL [31] is used to cryptography and guarantee the authentication. SSL is a method that intends to only the expected agents receive a symmetric key.

At this point, it is assumed a system where all the entities are using RSA to communicate with each other, but CA. Different from the other entities, the CA uses its own private key to generate the messages to assure that any entity can open the certificates with the public key. In that way, it is nearly impossible to fake certificates.

SSL is a protocol followed when the intention is to open a secure channel by entity $A$ to communicate to entity $B$. The sequence below represents the steps needed. It is assumed the RSA algorithm in the communications as explained before.

1. Entity $A$ sends a request for the certificate of the entity $B$ to the agent $B$ and the CA.

2. Entity $A$ compares the received certificates.

3. Entity $B$ sends a request for the certificate of the entity $A$ to the agent $A$ and the CA.

4. Entity $A$ compares the received certificates.

5. If both entities authenticate the other, they exchange a symmetric key.

When those steps where complete, the SSL is created and all the messages exchanged between the agent $A$ and $B$ are made using AES, until the end of this thread of communication. When the communication is done, both agents ignore the key, to create another SSL in the futures communications.

# Chapter 3

# Architecture of the Smart Parking System

This chapter intends to expose the architecture created for the Smart Parking using UML Diagrams. Furthermore, a module of this architecture is proposed using SSL to improve security. Notice that this security approach is only a proposal method, all this work was done using the base architecture.

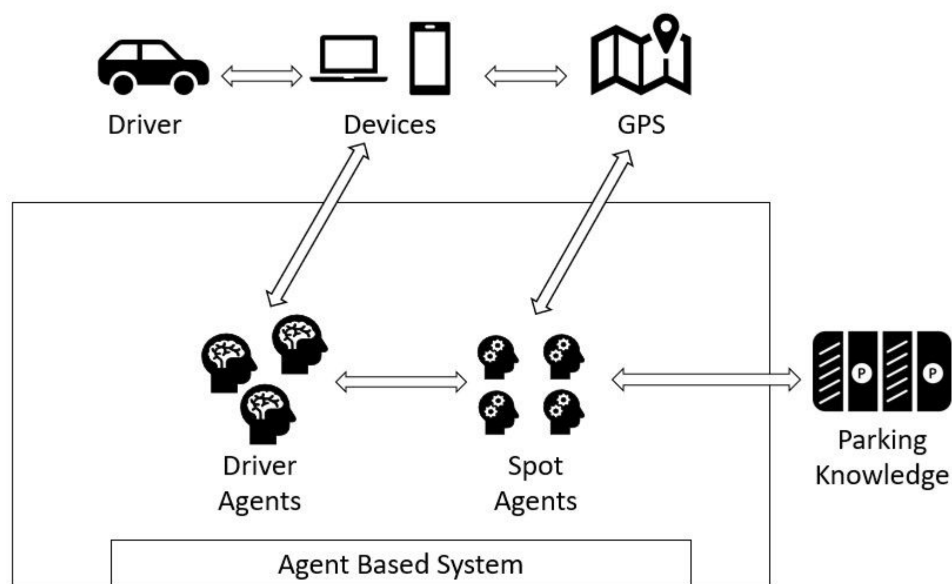


Figure 3.1: Overview of the Architecture.

The architecture has three main parts: The user interface (top of the Figure 3.1) representing the Driver with its phone and getting the GPS location from another service; The MAS (square of the Figure 3.1 and the focus of this work). It can have several agents representing the Drivers and the Spots; And the hardware (right part of the Figure 3.1) that covers the gates on the parking and the sensors to identify the vehicles in the spot. The communications between those parts were made using MQTT (more information about MQTT in Subsection 2.4).

## 3.1 Base Architecture

The Base architecture was made thinking in generalization. Use the same architecture doesn't matter the vehicles. It can be, for instance, a car parking, a bicycle parking or even a system with cars and bicycles.

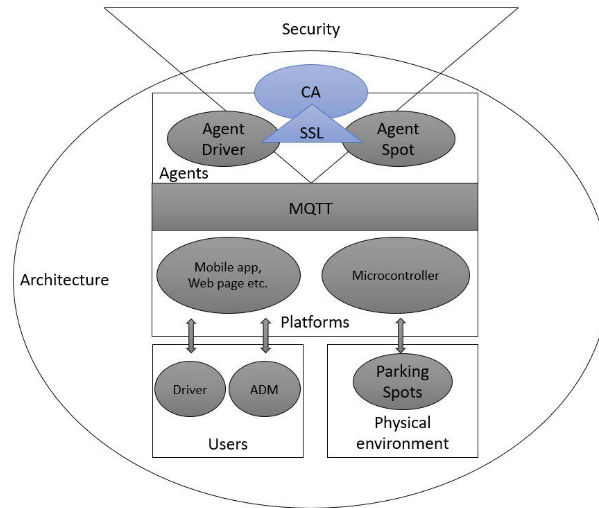


Figure 3.2: Architecture of the smart parking.

The Figure 3.2 explains the system as a whole. The Architecture is composed of five main modules and a security module as an option. The modules are:

- Users: The users of the system. They can be Drivers or AdmSpots.
- Physical environment: The real world where the Drivers and the spots are.

- Platforms: It was imagined as a microcontroller able to open and close the gate of the parking spot (or change any light that can indicate if it is occupied or not). It will not be detailed, because it's not the main propose of the work. And for the Driver and AdmSpot interactions, a mobile application.

- MQTT: How the interface and the MAS communicate with each other (in detail in Chapter 6).

- Agents: The core of the system, representing the Driver and the Spots (managed by the AdmSpot).

Furthermore, a security module is proposed focused on the communication between agents using SSL and CAs (trustworthy agents).

The main proposal of the system is to be intelligent, autonomous and also supports scalability and flexibility. Translating it to the architecture of this work, it must be capable of choosing the best price for both sides, manage the prices for each parking spot over time, be capable of increase and decrease the size of the system (create or exclude entities without reboot the system) and be simple to implement in other scenarios.

The MAS is a good way to do it, since it works with agents in a system, given the system autonomy. Additionally, MAS can have scalability, that is, the system can manage to create and delete agents (parking spot for example) without the need for the system to be restarted. Furthermore, it may be difficult to create a Smart Parking System because its complexity, since it may have several users, (Drivers and parking owners), but, with the MAS the project is divided into little pieces (agents), making it easier to create each one for them unify it all.

Moreover, the architecture was made using the same principle, dividing the system into functionalities. They are:

- The client (Driver) must be capable of managing its information in the system. It must be able to create, read, update and delete its information.

- The client must be capable of managing its vehicles in the system. It must be able to create, read the information, update the information and delete its vehicles.
- The client must be capable of request a reservation.
- The client must be capable of canceling a reservation.
- The client must be capable of recharging its credits in the system.
- The client must be capable of pay tickets it may have.
- The client must be capable of defining its preferences for parking spot reservation requests. To make it fast.
- The parking administrator must be capable of managing its parking spots. It must be able to create, read the information, update the information and delete its vehicles.
- The parking administrator must be capable of creating promotions for its parking spot.

Thinking in MAS and the functionalities the system will have, four actors appeared. These four entities are in control of the system, just they can actually do something on it. They are:

1. Driver: Represents the user of the mobile application.
2. AdmSpot: Represents the administrator of a parking.
3. AgentDriver: Represents the agent of the Driver in the system. The responsible agent for its information, negotiation, and representation itself in the system.
4. AgentSpot: Represents a single parking spot in the system. An AgentDriver can be responsible for more than one AgentSpot. In that way, the scalability of the system is guaranteed.

### 3.1.1 Use Case Diagram

The first Diagram presented in this work is the Use Case Diagram as can be seen in appendix B. As the first Diagram, it shows the functionalities and entities the system will have. With it, classes are derived.

The Driver can make 5 actions in the system. They are:

- Pay: It can recharge credits in the app or pay a ticket. The recharge method and money transaction were not one of the main interests at the beginning of the project, so it will not be focused.

- CRUD vehicles: It will be possible to 'CRUD' the vehicle information, such as which type of vehicle, and a nickname for it.

- Cancel a reservation: It can cancel a reservation made for a mistake, for example.

- CRUD personal information: It will be able to manage its information, such as login and password. The creation refers only to create an account in the system (login).

- Define default specification for Parking Spot allocation: Maybe the core of the application, this Use case represents when it searches for a Spot. But before that, it must define some parameters, just like the spot wanted and which vehicle it will use. When it defines it for the first time it is saved for future searches. The Driver can stop there or it can go to the Use Case 'Negotiate Parking Spot', the Use case that will search for a good spot.

Notice that the AgentDriver has two Use Cases, but it is for future approaches that may be using some AI. For instance, the system can predict, with the actual time, where the Driver is most likely to go. Or even, the Driver schedule for the agent searches for a time that maybe the prices are lower.

The important part of the 'Negotiate Parking Spot' Use Case is that not always the wanted spot by the Driver will be available, so it might return another option based on

the search parameters. Either way, the Driver will always have a chance to deny the negotiation.

Another important Actor is the AdmSpot, it will be in charge of two main things, manage its parking spots and create promotions for them. Furthermore, it can generate a report of a spot parking.

On the other hand, the AgentSpot will be able to close and release the parking spot, and, also, offers the promotions in case of an AI is used on it.

### 3.1.2 Class Diagram

The second Diagram presented in this work is the Class Diagram as can be seen completely in the appendix C. It is a huge Diagram, and because of that, it was divided into some parts to get a better understanding. In total the Diagram has been divided into 5 screenshots related to the 'types created for the system', to the 'Persons in the system', to the 'Driver', to the 'parking spot' and to the 'agents in the system'. Moreover, almost all Classes in the system has an id. That makes the identification of objects in the system easier.

The first Figure (Figure 3.3) refers to some classes that work like attributes. That is, they simply describe literal values: 'ParkingSpotType' and 'PayMethod'. They can have the values presented in the diagram but they can also be modified depending on the scenario. The first Class represents the Spot type, which type of vehicle it can park ('HANDICAPPED' is used when it doesn't allow a vehicle to park, for instance, it needs to be repaired). The second Class represents the payment methods that the system allows the user to make.

The second Figure refers to an abstract Class, the 'Person' Class (Figure 3.4). It is the base representation of a real person in the system. It must have a name, an ID, a login, a password, and an e-mail. The Classes 'Administrator' and 'Customer' inherit from the Person. The 'Administrator' has only one more attribute, the 'accessType', an attribute that may be used in the future to improve the system with other access levels. At this point, the methods are the important part, they are, the "'CRUD' spot" and the

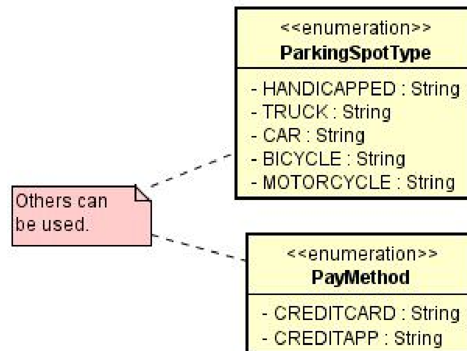


Figure 3.3: Part of the Class Diagram focused on the enumerate Classes.

"'CRUD' promotion", just like in the Use Case Diagram. That is, the object of this Class represents the actor AdmSpot in the system.

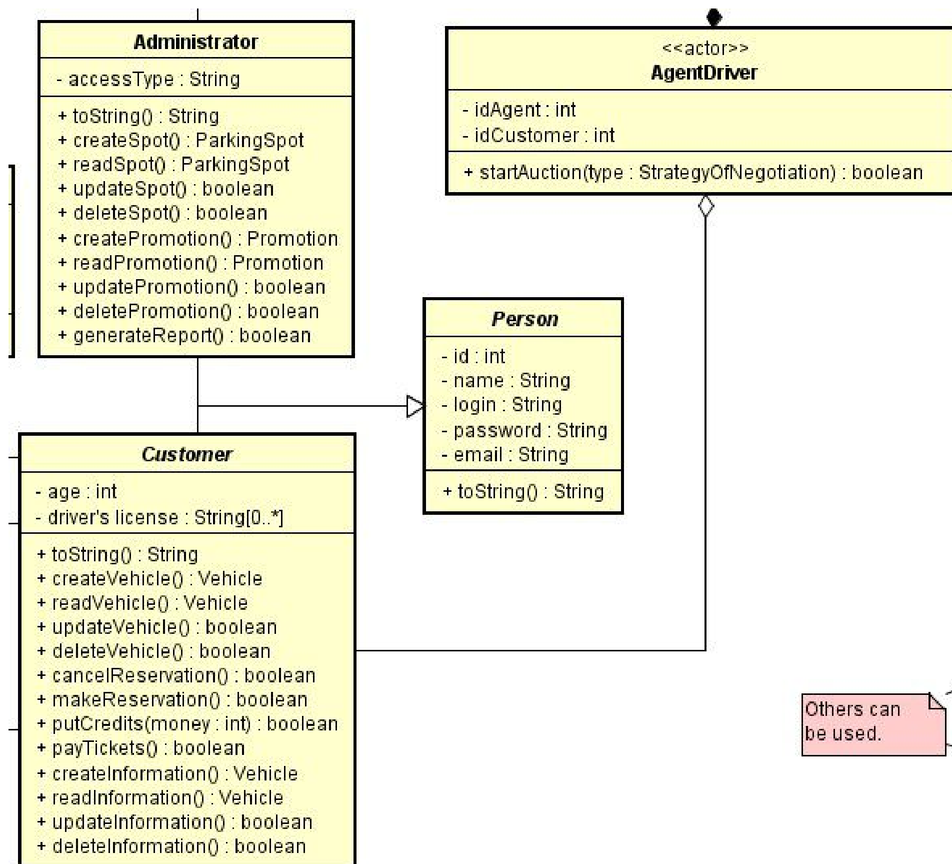


Figure 3.4: Part of the Class Diagram focused on the abstract class 'person' (Real person in the system).

It has 2 more attributes, the age and the licenses, the attributes of the ’Customer’ related to its vehicles. Notice there can be 0 or multiple licenses, since the Driver can ride only bicycles not needing a license, or, it can ride other vehicles like cars and motorcycles needing one or more licenses.

The ’Customer’ can make some actions in the system, they are, the ’CRUD’ of vehicles, cancel or create a reservation, recharge the credits in the app, pay tickets and the ’CRUD’ of information. Notice, all those methods at the end call the AgentDriver to put or request the data in the system. On the other hand, the ’makeReservation’ method selects some attributes and calls the AgentDriver to start the auction and exchanging messages in the middle of the process.

The ’AgentDriver’ refers to the AgentDriver in the Use Case Diagram, and can only exist if a ’Customer’ exists. This Class represents the Driver in the system, like an assistant. It contains its id and the id of the ’Customer’. It can also perform some actions, but it will be explained later when all the agents in the system will be detailed together (Figure 3.7).

The third Figure refers to the ’Customer’ (Figure 3.5). It may have tickets, vehicles, and reservations for some of the vehicles. It can make all that was specified for the Actor Driver in the Use Case Diagram. On the other hand, the ’Vehicle’ Class is important to be mentioned here. It might be different vehicles, like a car, bicycle or a truck, but not only that, this part can be modified depending on the scenario. Furthermore, all vehicles have a default specification. On the first time, the Driver requests a reservation for that vehicle is saved just like in the Diagram.

The ’Reservation’ Class can only be an object in the system if there is a ’Customer’, a vehicle and a ’ParkingSpot’ (when Figure 3.6 is explained, the relation between entities will be visible), since the Driver makes the reservation for a car in some parking spot.

The fourth Figure refers to the parking spot (Figure 3.6). It is possible to see that the ’ParkingSpot’ Class can only exist if there is a ’ParkingSet’, and the ’ParkingSet’ can only exist if there is a ’Parking’. Not mentioned until now, the ’ParkingSet’ represents an area or a sector in the parking, very common on shopping for example. The ’promotion’,

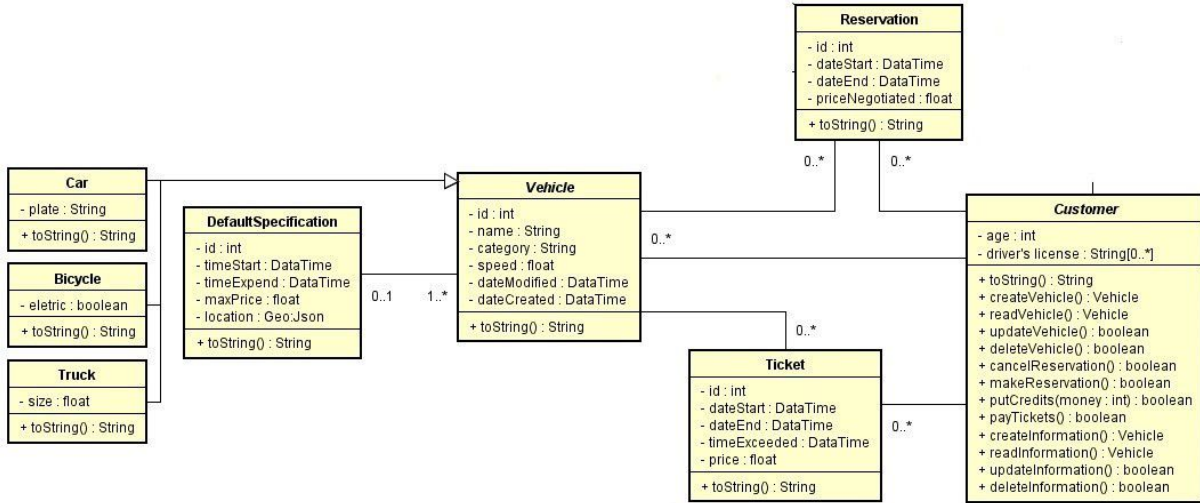


Figure 3.5: Part of the Class Diagram focused on the customer (Driver).

despite referring to a single spot, in the future it may be used by all the parking spots in the same parking set, or even in the same parking. Furthermore, the system should be able to handle multiple parking.

The Class 'Geo:Json' refers to the GPS coordinates or some other way to locate the spots in the real world. Another important attribute is the 'auction', despite it be an integer, it represents the protocols used to negotiate (more about it in Chapter 5). The parking also has a gate of some indicator that it is available or not. It can also represent a gate for a sector in the parking, explaining the one or many relations with the 'ParkingSpot'.

Lastly, the Spot may have a 'Reservation' as explained before, and it also has an 'AgentSpot', but not an administrator. Notice the Administrator doesn't change directly the 'ParkingSpot', it requests to the 'AgentSpot' to do it to check the inputs and update the database of the 'AgentSpot' itself. Each AgentSpot represents a single parking spot. The Administrator, on the other hand, maybe responsible for more than one AgentSpot.

The fifth and last Figure refers to the agents in the system (Figure 3.7). Maybe the most important part for this work, it represents the AgentSpot and the AgentDriver. Notice, only the 'AgentDriver' can start a negotiation.

Furthermore, both will have a 'StrategyOfNegotiation' when the negotiation starts,

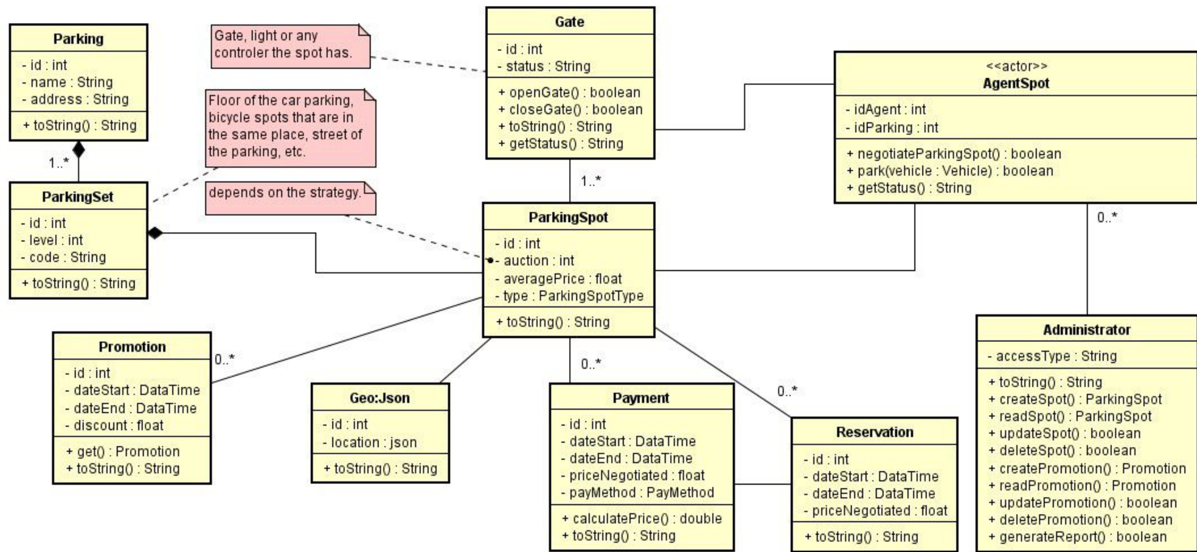


Figure 3.6: Part of the Class Diagram focused on the parking spots.

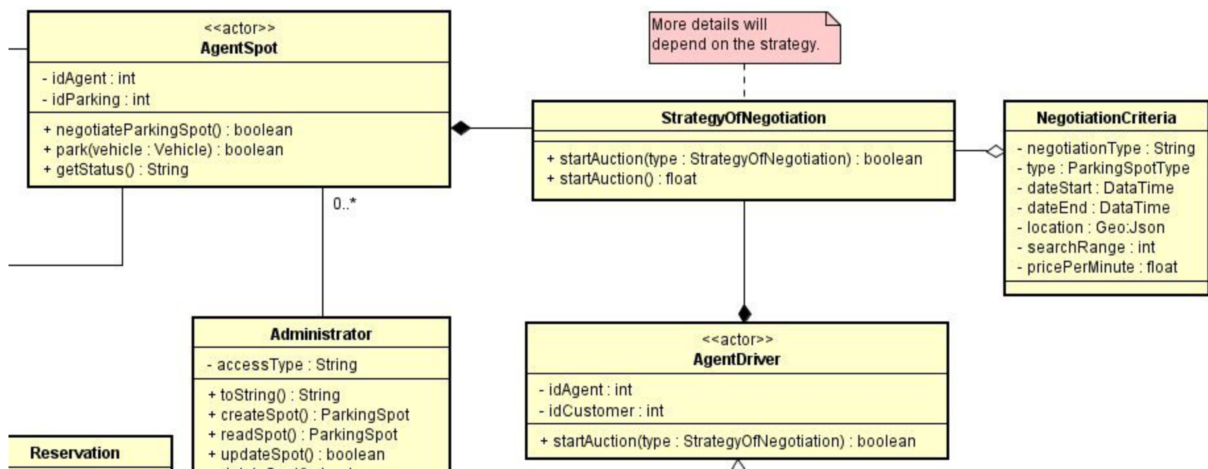


Figure 3.7: Part of the Class Diagram focused on the agents

but they will have opposite interests, but it may be different for each protocol used to negotiate (more about it in the Chapter 5). And every time a negotiation starts, the Driver will inform what spot it intends to get, generating an object of the Class 'NegotiationCriteria'.

### 3.1.3 Activity Diagram

The last pack of Diagrams presented in this section refers to the Activity Diagrams. Each one representing the flow chart of a Use Case. In total it has 11 Use cases and 13 Diagrams explained here. That's because there are similar 'CRUD'. All the 'Create' and the 'Delete' Diagrams are basically the same. On the other hand, the 'Read' and 'Update' Diagrams of the 'Personal Information' and 'Parking Spot' are different from the 'Vehicle' and the 'Promotion'. The sections below explain all the Diagrams that can be found in the appendix D.

#### Create

The Diagram on the appendix D.1 refers to the creation of vehicles, personal information, parking spots, and promotions. The first step is the acquisition of new data. Validation of the data is made to check if there is any field with some wrong format or if already have this information in the system.

If some error occurs in the validation process, the system informs it and asks if the user wants to try it again. When the data is inserted into the system to create a vehicle, it will be checked if it is, for instance, a category of vehicle accepted by the system. When the data is validated a last verification with the user is made to create and save the new data.

#### Delete

The Diagram on the appendix D.1 refers to the deletion of vehicles, personal information, parking spots, and promotions. The first step is to request and validate the password.

After that, the last verification is made to delete it. Notice, it is assumed that the item to be deleted is selected.

### Read Information and Parking Spot

The Diagram on the appendix D.1 refers to the read of personal information and parking spots. The first step is to request and validate the password. After that, the information about the object is loaded and shown. Notice there is no ID request on this Diagram because it refers to itself, not another object in the system.

### Read Vehicle and Promotion

The Diagram on the appendix D.1 refers to the read of vehicles and promotions. The first step is to request and validate the password. After that, the system requests the ID of the object. If the ID doesn't exist or the user doesn't have permission to read it the system informs the error and goes back to the password request. Otherwise, the data is loaded and shown.

### Update Information and Parking Spot

The Diagram on the appendix D.1 refers to the update of personal information and parking spots. The first step is to request and validate the password. After that, the data is loaded to the user to identify and modify it. Then, a last verification of the updated values is made. Notice there is no ID request on this Diagram because it refers to itself, not another object in the system.

### Update Vehicle and Promotions

The Diagram on the appendix D.1 refers to the update of vehicles and promotions. The first step is to request and validate the password. After that, the system requests the ID of the object. If the ID doesn't exist or the user doesn't have permission to read it the

system generates an error. Otherwise, the data is loaded and shown to the user to make the desired modifications. When it is done, the updated data is validated and saved.

### Pay

The Diagram on the appendix D.2 refers to the insertion of credits in the system or the payment of tickets. The first step is to choose one of the two options: put credits or pay a ticket.

If the user chooses the payment of the ticket, the system loads them and shows to user select which one it wants to pay. More than that the system requests the payment method. When it pays (with credits or real money) the system update the agents involved with this transaction.

On the other hand, if the user selects to recharge its credits the system request the amount and the payment method (with a credit card, bank transfer, etc.). In the end, the system updates itself.

### Cancel a Reservation

The Diagram on the appendix D.3 refers to the cancellation of a reservation. The first step is to load the unfinished reservations of the Driver. The Driver selects one to cancel and a request to that spot is sent. After loading that data, a final confirmation is required. If the Driver confirms it the credits are returned and the reservation is deleted.

### Define Default Specification

The Diagram on the appendix D.3 refers to the definition of a default specification to search for a parking spot. The first step is to request the specifications. After validation, the Driver can go straight to the search or just quit the process.

### Negotiate Parking Spot

The Diagram on the appendix D.3 refers to the negotiation of a parking spot. The first step is to check if the Driver has something pending (tickets unpaid). After that, the default specifications are loaded (if the Driver has already made a reservation before, that is used) and the system asks if there is some change to be done before the search.

The specifications are validated (if the Driver inserted, for instance, a valid wanted spot) as long as the available credits and the process of negotiation starts (Depending on the protocol it might change, more about it in Chapter 5). When the negotiation is done, the payment is made along with the reservation.

### Occupy Parking Spot

The Diagram on the appendix D.4 refers to when a Driver wants to park its vehicle in some parking spot. The first step is to load the data related to that time and spot. The system gets the ID of the vehicle somehow (camera, Radio-Frequency IDentification (RFID), the AgentDriver informs it, etc). After that, the spot verifies if this vehicle is supposed to park, indicating it.

### Release Parking Spot

The Diagram on the appendix D.4 refers to when a Driver wants to get its vehicle out of some parking spot. The first step is to request it. A verification is made based on the current time and the time expected to the vehicle leave the spot. Based on that a ticket may be generated. Anyway, the gate opens (if there is one).

### Generate Report

The Diagram on the appendix D.4 refers to the generation of a report of a parking spot. The first step is to request and validate the password. Then, some specifications can be made, like the time interval in which the user wants to get the report. In the end, a report is generated.

## 3.2 Security Focused Architecture

With cryptography, it‘s possible to the CAs store public keys aside with the certificates of the agents in the system. Also, the CAs will not be needed to handle the whole system data, just a small part of it.

In the end, the system will have several CAs grouped in regions or sectors holding the public key shared from the agents and the certificates. Allowing the use of SSL and ensuring fault support. That can guarantee the security and integrity of the system as a whole.

Moreover, another important issue is the expiration of the keys, from time to time, all the keys must be changed to improve the security of the system as explained before. That process can be made automatically and be sent to one CA to share it with other CAs in the same region.

### 3.2.1 Proposal Roles

Initially, the first deployed CA in the system will certificate other entities in the system, including other CAs, not relying on only one. Furthermore, it may make the system faster, because of the multiple CAs responding to several agents instead of only one responding to all of them. Also, if the system has only one CA, it might be far from some of the agents increasing the latency.

As an alternative to reduce infrastructure costs, some of them can be a parking lot that already has some structure, for instance, a parking with a database of clients. But, most importantly, it must be trustworthy to the system. To validate it, a Certificate Manager (CM) is needed. The CM is a person responsible to validate entities and update the database of some CA to that entity be accepted by the system. After the system accepts the new entity, the CA is not needed anymore.

On the other hand, all the agents deployed in the system will need to be launched with at least one CA listed on its memory no matter if it's a AgentSpot or a AgentDriver. Also, when a new agent gets in the system, it must be certified by one CA. After that the

CA can inform the others automatically.

Another improvement to the system is having the CAs responsible for an area or a region, not the entire system. It will allow the CAs to deal with it with less memory and processing. But, with it, a problem comes up, when a Driver from other sector requests something. To manage that, the CAs from different areas must communicate. MQTT (a message protocol capable of alert when something important arrives, more about it in the Subsection 2.4) may be used to request something from the outside or the known area. With this system, the integrity of a region will not be compromised, since it will have more than one CA there.

### 3.2.2 UML Architecture

Figure 3.8 refers to the part of the Use Case Diagram of the base architecture that had suffer some changes. The main difference is the addition of two actors and four use cases. The actors represent the CM ('CAManager') and the agents of the several CAs (CA). With those Use Cases, all the CA operations can be established and a secure channel can be made (SSL). The four new use cases are:

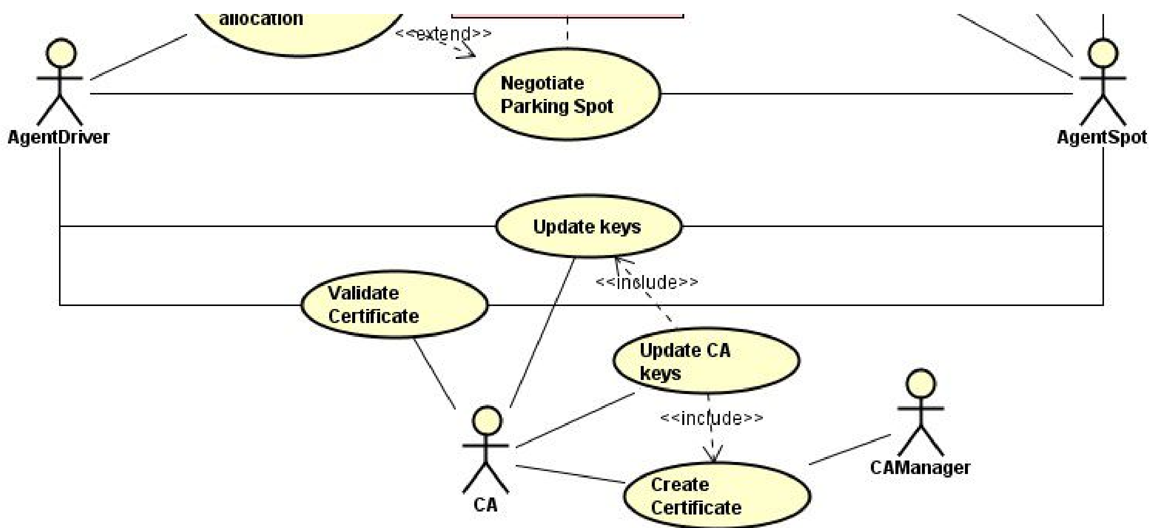


Figure 3.8: Part of the Class Diagram focused on the agents

Notice the CM and the AdmSpot are ideally not the same person. That is because the

CM is responsible for all the systems, which may include more than one parking, while the AdmSpot is responsible for only its parking.

- Create Certificate: When a new agent or CA is deployed in the system, the CM needs to add it to one CA database, generating a certificate.

- Update CA keys: The CA informs all the CAs in the same area or region to update their database, informing the alterations.

- Validate Certificate: This Use Case is used, for instance, when an agent $A$ requests the certificate of an agent $B$. The CA returns to the agent $A$ the requested certificated if it has one, and also, validate the certificate of the agent $A$. If it is an intruder, some actions can be done.

- Update keys: When the agent is created or a key expires it is needed to create a new pair of keys and that is what this use case does.

The appendix E shows the Class Diagram differences. The new Classe 'Manager' represents the CM, the 'AgentCA' represents the CA agent and the 'Certificate' represents the certificates. The first Class is, the login of the CM in the system and it can administrate more than one CA. The Class 'AgentCA' is the CA agent in the system responsible for creating and validating certificates. Also, all agents now have a public and a private key, as well as a method to update it. The AgentDriver and the AgentSpot have one similar method, the 'validateCertificate', but instead on validate, they request one CA to do it.

At last, there is the Class 'Certificate' that represents the certificate of the agents in the system. Each agent has its own certificate generated by the CA. Even the 'AgentCA' has one. The difference is that the 'AgentCA' has the certificates of others.

## 3.3 Considerations

This architecture uses autonomous agents. The intelligent agents negotiate to get the best price autonomously. Even in the security-focused architecture, after the entity enters the system and is registered by a CM the system can continue to run by itself.

The proposed architecture can also be easily adapted for other scenarios. For instance, it supports one or many car parking, since it might have several AdmSpots. Moreover, it can have several vehicles just changing the 'ParkingSpotType' Class and by changing the structure of the agents is possible to make a holonic system based on it.

# Chapter 4

# Negotiation Protocols

This chapter intends to expose some assumptions and explain how the auctions were adapted for the case of a Smart Parking System. The Figure 4.1 represents an overview of the negotiation protocols. It's possible to see the Strategy used by both agents (Drivers and Spots), that is, both must agree on how to negotiate. Moreover, the resources they will be negotiating, the money and the spot are used to get an agreement. The money represents the value of the spot, while the spot itself has a value depending on the distance between it and the spot wanted by the Driver.

## 4.1 Assumptions

The agents that will perform the auctions will only know about themselves and the environment. The AgentDrivers, for instance, will not have any information about other AgentDrivers. Furthermore, it will only know about the auction it is participating in. On the other hand, the AgentSpots will not know about other AgentSpots and will know only about the auctions it is participating in.

Since in the Smart Parking System the Driver and the AgentSpots have opposite interests (while the Driver wants the smallest price, the AgentSpot wants the highest), the cooperation in such system will be in terms of a negotiation to find a consensus, pleasing both parts (a reasonable price for both agents) or denying the negotiation. In other

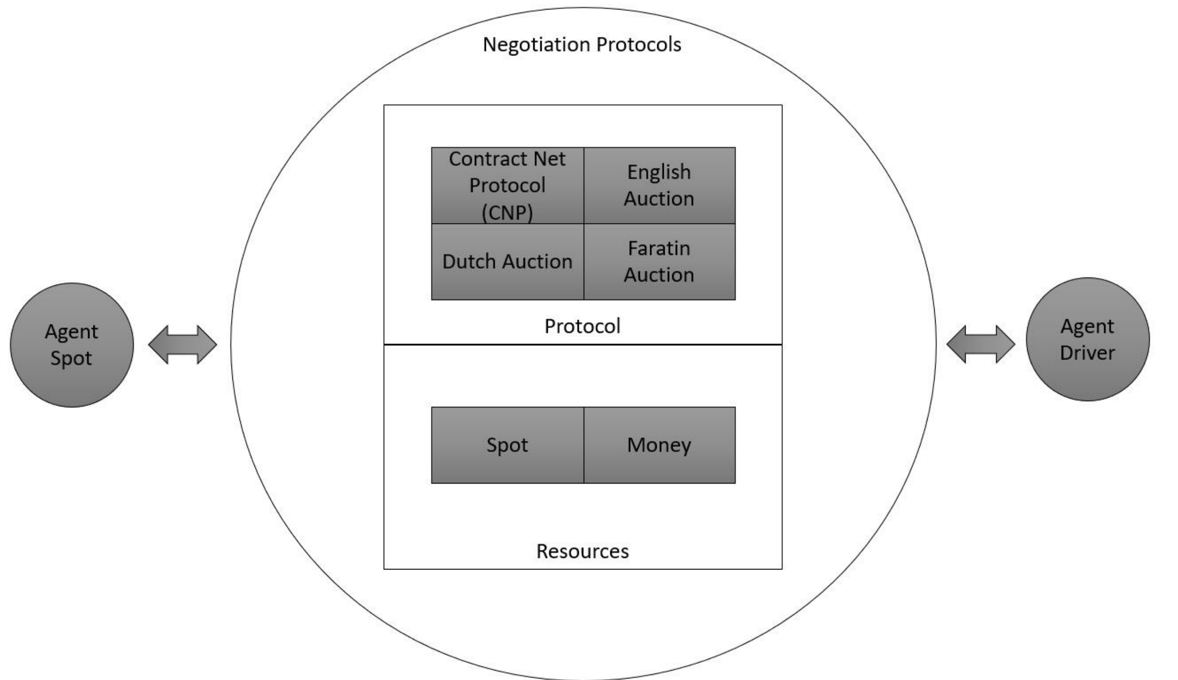


Figure 4.1: Components of the Negotiation Protocols in a Smart Parking System.

words, they will try to get a consensus, if this not happens in a period, the negotiation ends without alterations. That allows the Driver to create another auction with different parameters.

In this case of study, the *Initiator* is the Driver. Therefore, it will try to get the lowest price to buy a reservation. The remaining sections explain each of the four protocols, namely the CNP, English Auction, Dutch Auction and the Faratin Auction.

## 4.2 Contract Net Protocol (CNP)

The figure 4.2 represents the Sequence Diagram of the CNP made for the case of a Smart Parking System. Initially, the AgentDriver initiates the negotiation by announcing the auction to the AgentSpots. This announcement message will contain the eligibility specifications the participant agents will have to satisfy to participate in the auction and the specification of the task.

When an AgentSpot receives an announcement message, it verifies if the required

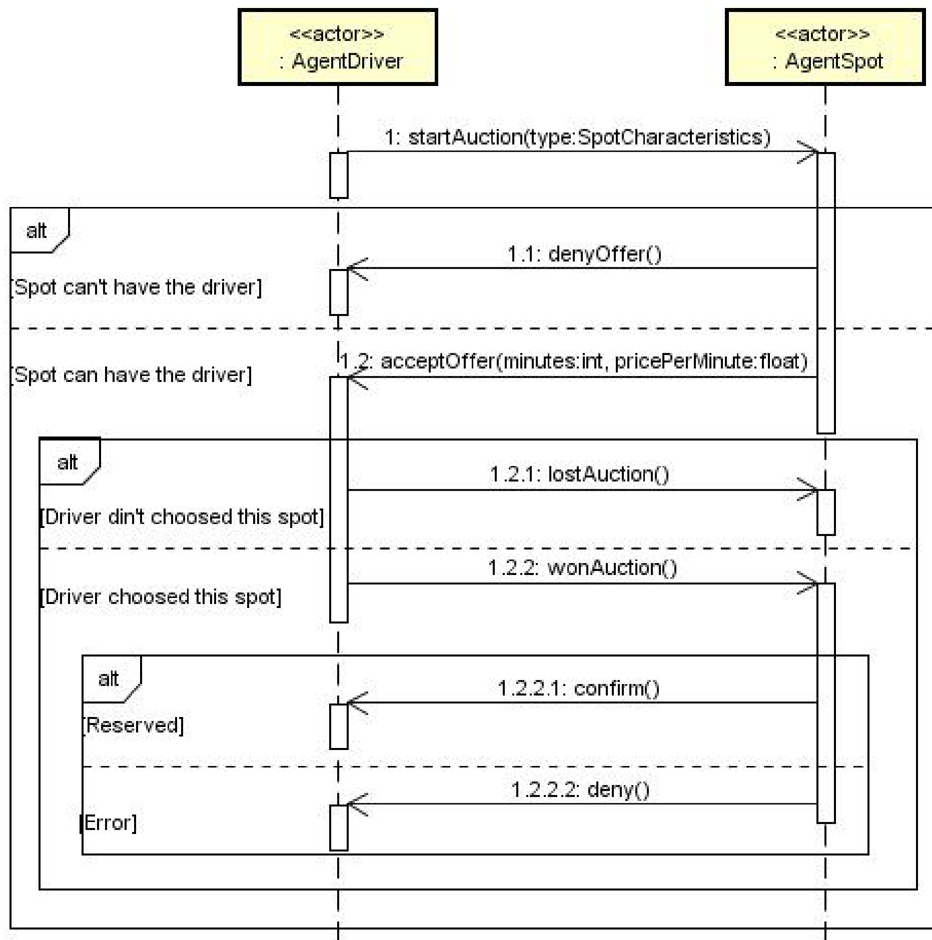


Figure 4.2: CNP Sequence Diagram of messages based on the FIPA standard [16].

specifications can be satisfied. If not, it sends a message saying so and getting out of this auction. On the other hand, in affirmative cases, it sends a bid proposal indicating its conditions (price) to participate in the auction.

After receiving the replies or the time is up, the AgentDriver evaluates all arrived bid proposals and decides which one is accepted according to its selection criteria. If none of them satisfies the Driver requirements it informs all the participants that it lost and the process starts again. Otherwise, The AgentDriver informs the winner and the losers. When the winner AgentSpot is notified about it, it makes a new verification on the conditions, if there is no problem it makes the reservation. Either way, the AgentSpot informs the AgentDriver if the negotiation is completed or not. Restarting all the progress or finishing it.

The CNP approach leads to a fast, but sub-optimal solution due to its spatial and temporal myopic. Spatial myopia means that the information of the state of other initiator agents is not used during the construction of a bid proposal, while temporal myopia means that the information of subsequent tasks is not used either in the bidding or in the award selection [32].

## 4.3 English Auction

The English Auction tries to achieve the consensus between the agents by changing the price over each turn, starting with a bad price for the initiator (the farther from 0 the worst) and making it better, until none of the participant agents accepts more changes in the price [21]. Figure 4.3 represents the Sequence Diagram that exposes the messages exchanged between the agents in this process.

Similarly to the CNP schema, the AgentDriver initiates the negotiation by announcing the auction to the AgentSpots that contain the eligibility specifications. This message contains the task specification. As soon as the messages are sent, the AgentDriver sends an offer with the price to all participant agents. When a participant agent receives an auction announcement message with the requested specifications, and, then the price, it

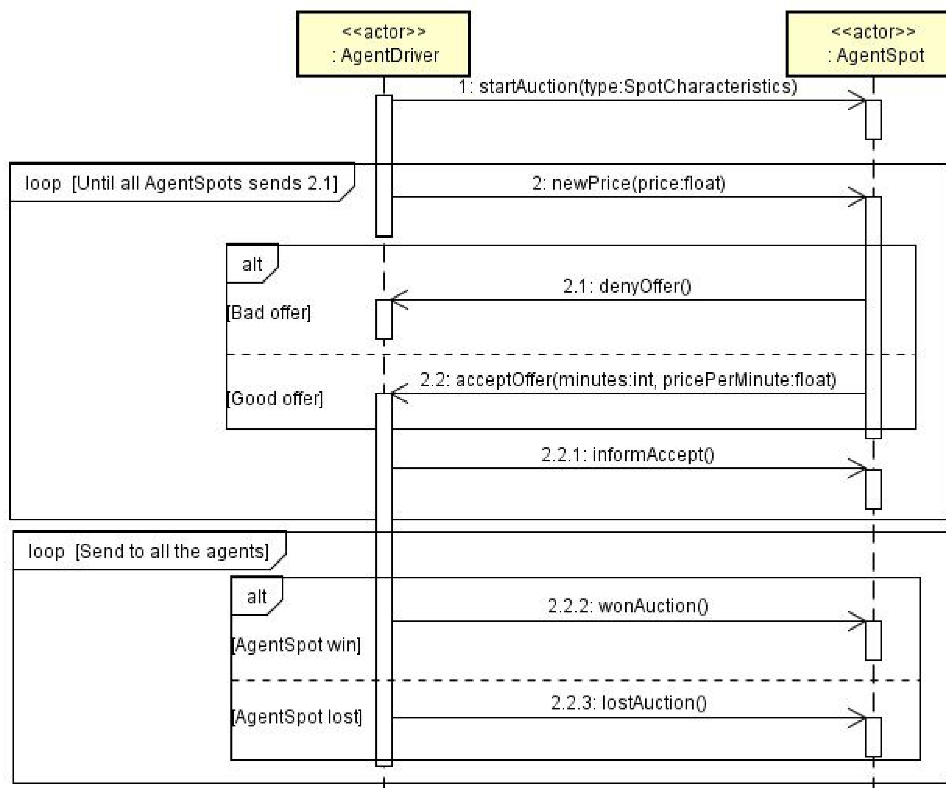


Figure 4.3: English Auction Sequence Diagram of messages based on the FIPA standard [16].

makes verification of the conditions. If they can be satisfied with a proposal it sends a message saying so. Otherwise, the announcement is rejected.

If at least one AgentSpot accepts the offer, the previous process will be repeated until no one accepts it. The expectation is that the price can be decreased with the new offer (better for the Driver) and one agent might accept the offer (the message "2.2.1:informAccept" is saying the offer was accepted, but the auction is not over). After no agent agrees with the offer, the AgentDriver sends a message to all the participating AgentSpot inform if it is or not the winner according to the Driver preferences.

The main problem of this auction is the number of messages exchanged, which means that the increase in the number of participants will significantly increase the number of messages and consequently require more time to achieve the consensus. Different from the CNP this auction has several rounds and only finishes when the time is up or none of the participants accept the new offer.

## 4.4 Dutch Auction

Similarly to the English Auction, the Dutch Auction tries to achieve the consensus between the agents by changing the price over each turn [22]. But, instead of starting with a bad price for itself, the Dutch Auction starts with a very good price (the closer to 0 the better), and in each turn, the price will be changed until some participant agent accepts it. Figure 4.4 represents the Sequence Diagram that exposes the messages exchanged between the agents in this process.

Initially, the AgentDriver sends a message to the AgentSpots announcing the start of the auction, which includes the eligibility specifications that the agents will have to satisfy to participate in the auction. After that, the AgentDriver sends an offer with the expected price to all participant agents. The AgentSpots checks the conditions and the price, accepting or refusing it.

If some AgentSpot accepts the offer, the AgentDriver sends a message to the selected agent confirming the agreement and then, informs all the agents if it's the winner or not.

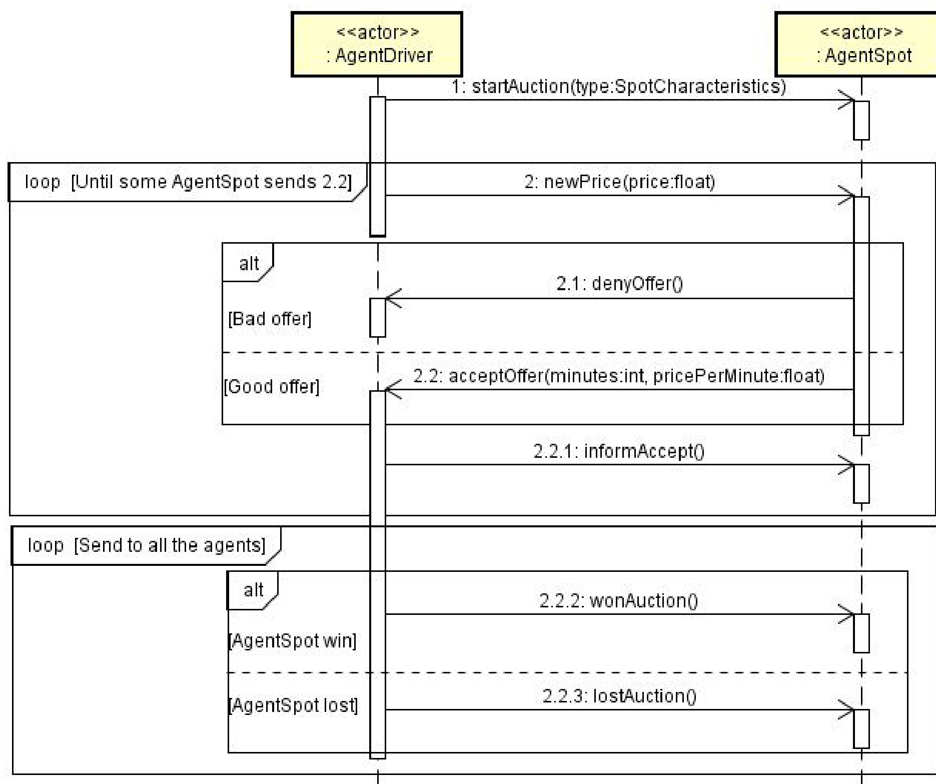


Figure 4.4: Dutch Auction Sequence Diagram of messages based on the FIPA standard [16].

Otherwise, it means that the price is not good enough for the participant agents, so it repeats the procedure until one agent accepts the price. If the price reaches a value that is not interesting anymore for the Driver or the negotiation reaches the time limit, it is informed to the participants and the negotiation can restart with new conditions.

## 4.5 Faratin Auction

Different from the previous auctions, the Faratin Auction tries to achieve the consensus through a series of offers and counteroffer [23]. Each side will start with a good offer for itself (AgentDriver will offer something close to 0, while the AgentSpot will offer something far from 0), then it will make it worst in function of something (time was the chosen metrics, since the AgentDriver has a time limit to complete the auction). Figure 4.5 represents the Sequence Diagram that exposes the messages exchanged between the agents in this process.

To start the auction, Just like the previous protocols, the AgentDriver sends a message to the AgentSpots announcing the start of the auction, which includes the eligibility specifications that the agents will have to satisfy to participate the auction. After, the AgentDriver sends the first offer with a price based on the Driver specifications, the time left in the auction and the position of that AgentSpot. On the other hand, when the AgentSpot receives it, it makes a verification on the attributes and price. If it is not a good offer, it calculates a counteroffer based on those attributes and the time left in the auction. That cycle of offer and counteroffer continues until they reach an agreement or the time is up.

If the time is up, the AgentDriver informs everyone about it and might restart with other attributes. Otherwise, no matter who sends the 'acceptOffer' message, the Agent-Driver sends the next message to all the participants informing if it lost or not. When the winner AgentSpot receives this message it makes a final verification to make the reservation. Then, it will inform the AgentDriver.

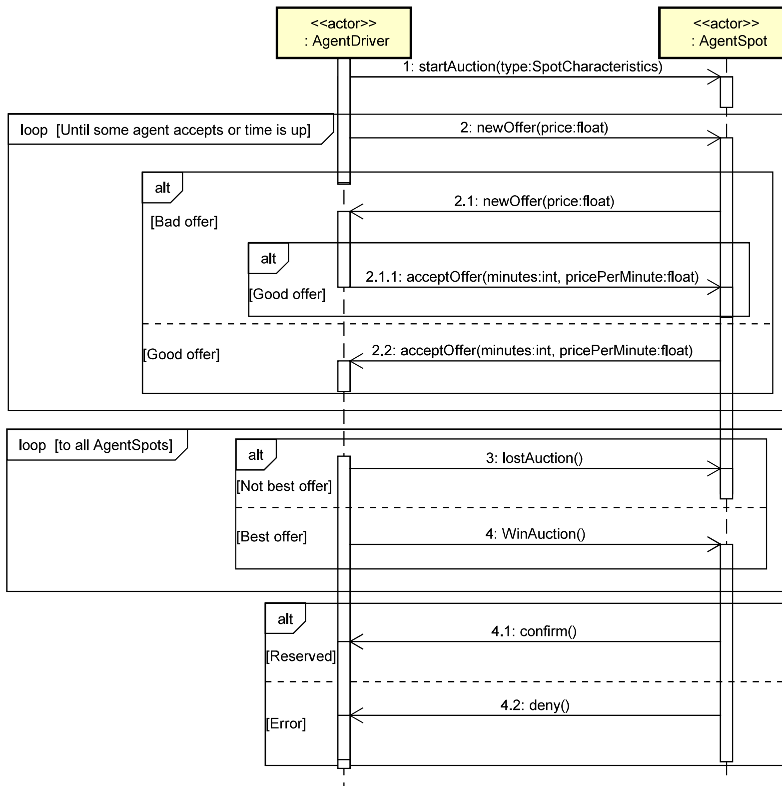


Figure 4.5: Faratin Auction Sequence Diagram of messages based on the article [23]

# Chapter 5

# Comparative Analysis of the Negotiation Protocols

This chapter intends to test the CNP, English Auction, Dutch Auction and the Faratin Auction according to different scenarios. The evaluation has considered the level of satisfaction of the agents in the system, the scalability and the negotiation time.

## 5.1 Simulations

This section details the experimental tests developed to analyze the different negotiation protocols for the Smart Parking problem. The agent-based parking system, and particularly the four referred negotiation protocols, was implemented using the JADE framework [13]. The agent-based model was developed by implementing the behavior of the driver and spot agents, and the four described negotiation strategies following the Interaction Protocol Specifications defined by FIPA [16] and the article [23].

Since the code related to the protocols is the same as Chapter 6 it will not be focused here. Instead of it, graphics and tables will be shown to present the results.

An important issue considered during the agents' implementation was related to the mechanisms for the price generation, where the average price is selected randomly between 10 and 100 for each AgentSpot. These values represent the highest price that a

AgentDriver can pay for a spot and the lowest value the AgentSpot will accept. Also, only integer numbers were accepted to be easier to visualize and converge faster to a consensus.

The agent-based Smart Parking was running in an Aspire F5-573G, Intel core i5 7200U, 8GB RAM DDR4 and SSD in a Windows 10.

### 5.1.1 Scenarios

The behavior of each negotiation protocol was tested taking into consideration 9 scenarios build up the variation of the number of available parking spots and Drivers, considering three sets: small (50), normal (175) and large (300). We are considering small and medium car parking, even the 300 spots parking will have something around $3750m^2$ when we considered just large spots with $2{,}5m^2$ length and 5m width [33].

These scenarios, ranging from 50 Drivers and 50 parking spots to the 300 Drivers and 300 parking spots named from $A$ to $I$, just like is shown in the Table 5.1, were experimentally tested 40 times each one to get an average. These numbers were chosen to represent scenarios where are spots left (for instance $G$), the same amount of spots and Drivers (for instance $E$) and too few spots (for instance $C$).

Table 5.1: Amount of Drivers and Spots per scenario.

| Drivers \ Spots | 50 | 175 | 300 |
|---|---|---|---|
| 50 | A | B | C |
| 175 | D | E | F |
| 300 | G | H | I |

Since the Smart Parking System is a dynamic system, it was considered that all the AgentSpots are available from the beginning, but the AgentDrivers are not created all at the same time since it's not been considered a 24h parking. Each AgentDriver has a possibility of 10% to be created inside a loop just like in a real case. The parking time can range from 1ms to 100ms, and the searched area for parking spot by a Driver is at a maximum of 50% of the whole parking because of its size not be that big. To simulate that, an array with all the AgentSpots was made.

The AgentDrivers and AgentSpots were classified into three categories reflecting their role in the negotiation process. The Table 5.2 refers to the AgentDriver profiles while the Table 5.3 refers to the AgentDriver profiles. The $Minprice$ reflects the minimum price the agent will accept in a negotiation. The $Maxprice$ refers to the maximum price the agent will accept in a negotiation. The $Pricechange$ reflects the value that will be changed in the round of the negotiations compared to the previous offer when needed.

Table 5.2: Driver Profiles.

| Driver | Min price | Max price | Price change |
|---|---|---|---|
| Conservative | 0 | 40 | 5 |
| Moderate | 0 | 75 | 10 |
| Aggressive | 0 | 100 | 15 |

Notice, the column $Minprice$ of the Table 5.3 is in percentage referring to the amount that will be subtracted from the average value of the AgentSpot. For instance, the minimum accepted by a Moderate AgentSpot that has an average price of 50 is 45.

Table 5.3: Spot Profiles.

| Spot | Min price | Max price | Price change |
|---|---|---|---|
| Conservative | 5% | 100 | 5 |
| Moderate | 10% | 100 | 10 |
| Aggressive | 15% | 100 | 15 |

The distribution of profiles in the agents follows a normal distribution, with the system having 30% of conservative agents, 40% of moderated agents and 30% of aggressive agents.

Another simulation was made to compare the importance of the metrics in the negotiation. For that simulation, the distance between the spot desired by the Driver to the spot offered is the only metric the agent will take into consideration to see the other parameters' behavior. Moreover, the CNP was chosen to be the base of this simulation, because it does not trade, just accept the best option. In this case, the closest to the desired spot, no matter the price. For the rest of this work, this simulation will be called CNP Distance (CNPDist).

These simulations intend to get a lot of scenarios to see how the system will react

with each one of the negotiation strategies. The covered scenarios were very embracing allowing, in the end, to conclude the best negotiation approaches for each scenario. However, an important remark should be considered: these scenarios consider that at the beginning, the parking is empty, with no cars at any parking spot. That means 24 hours of parking is not considered in this work.

The metrics defined to evaluate the negotiation protocols are mainly related to the price paid by the Driver to reserve a parking spot and the distance between the desired parking place and the parking spot got by the driver. The achieved results will be evaluated under these two parameters considering the previously described scenarios.

Not only that, but the four negotiation strategies are also evaluated taking into consideration the number of messages exchanged during the negotiation process, the time required to conclude the negotiation and the difference between the highest and lowest price paid in the simulations.

## 5.2 Results

This section analyses the results from the experimental testing of the four implemented negotiation protocols, namely the CNP, English Auction, Dutch Auction and Faratin Auction, considering the scenarios previously described. After that, the CNPDist is compared with the CNP. Initially, the average price paid and the distance to the desired parking place are analyzed, then, other parameters are also compared, namely the number of exchanged messages, the negotiation time and the difference between the maximum and the minimum price paid in each protocol.

$$\Delta D = |Iw - Io| \tag{5.1}$$

Notice, as explained before, the distance between the spots is calculated by the difference between the index of the spots in the array. The Equation 5.1 shows how the distance is calculated, where the $Iw$ is the index of the wanted spot and the $Io$ are the

index of the spot obtained.

### 5.2.1 Analysis of the Average Price Paid by the Driver

The results for the price paid by the Driver for the four negotiation protocols are illustrated in Figure 5.1. The achieved results for the four strategies follow the same behaviour, with the price remaining stable between 175 and 300 parking spots in the system, and rising for scenarios considering less than 175 parking spots, which means that when the Driver agents are competing for a limited amount of parking spot, the price they need to pay is slightly higher. In fact, since there are more options for the Driver to choose the parking spot, the Driver prefers the cheapest ones (i.e. the selection function tries to minimize to price to pay), with the AgentSpot needing to reduce the proposal prices to remain competitive.

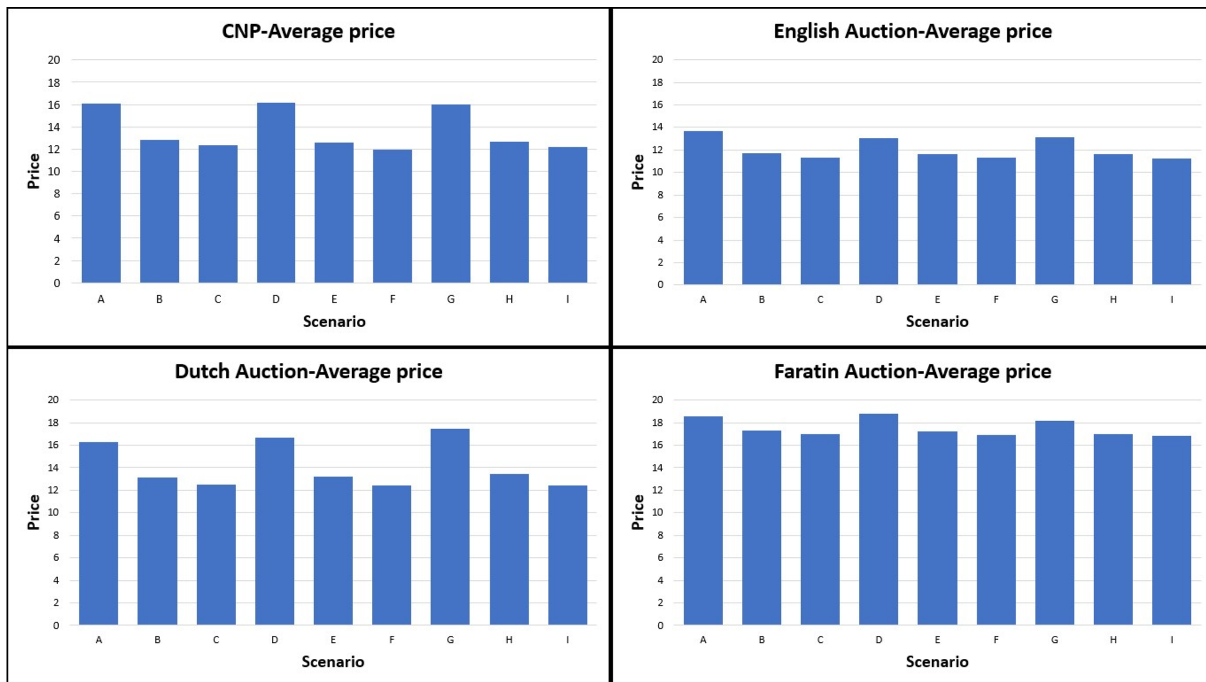


Figure 5.1: Results of the price paid by the Driver for the four negotiation protocols divided per scenario.

The price values for the CNP and English Auction strategies seem to be dependent only from the number of the spot agents, while in the case of the Dutch Auction and

the Faratin Auction, the values are dependent of the number of Driver and AgentSpot. The English Auction reached the lowest prices from the four tested strategies. And the Faratin Auction reached the highest prices, despite that, this protocol had the minimum price variation in the tested scenarios.

Moreover, the Dutch Auction tends to increase the average price as higher the amount of Driver when there are few spots when the English Auction tends to decrease the average price in the same scenarios (A, D and G).

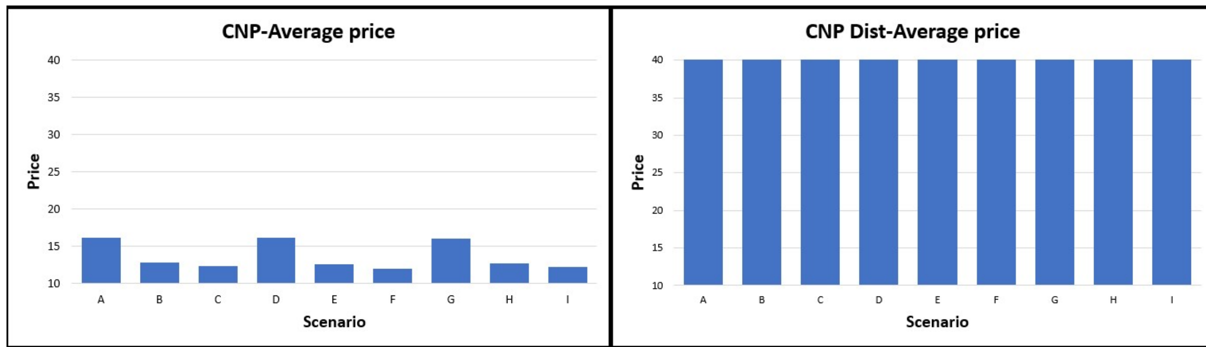


Figure 5.2: Results of the price paid by the Driver for the CNP and CNPDist protocols divided per scenario.

On the other hand, Figure 5.2 represents the average price obtained between by the CNP and the CNPDist simulation. It's possible to see the price increased a lot, almost four times compared with the CNP. That's because the Driver in the CNPDist doesn't care about the price at all.

### 5.2.2 Analysis of the Distance to the Desired Parking Place

The results of the distance of the assigned parking spot to the desired parking place (another parking spot) for the four negotiation protocols are illustrated in Figure 5.3. The observation of these results shows quite similar behavior for the four strategies. This distance is mainly dependent on the variation of the number of parking spots, increasing as higher is the number of parking spots. On the other hand, the distance almost does not change with the variation of the number of drivers in the system.

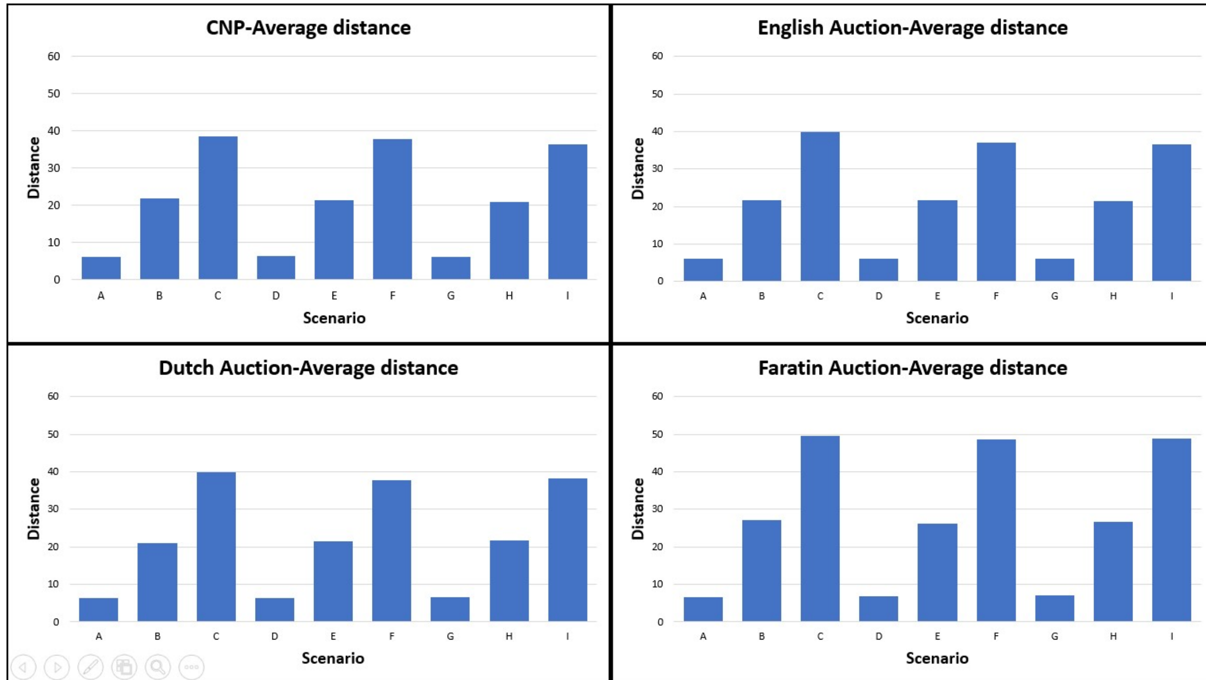


Figure 5.3: Results of the distance to the desired parking place for the four negotiation protocols divided per scenario.

Furthermore, when are more AgentDriver in the system, the average distance has a slight change, making the average go down. That difference may occur because of the weight of the overlap (when a spot wants a spot already taken) in the average. In other words, when an overlap occurs in a system with few overlap cases (scenarios A, D, and G), it may have a high weight when compared to a system with a lot of AgentDriver getting the wanted pot (scenario C, F and I).

Moreover, the English Auction, Dutch Auction and mainly the CNP auctions tend the average distance be around 12% of the amount of available AgentSpots in every scenario.

The the Figure 5.4 shows the average distance for the CNP and the CNPDist protocols. Just as expected, in the CNPDist tends to 0, when it can't get the spot wanted the Driver gets the next closest parking spot. That may happen because of the budget the Driver has or because the AgentSpot already has a reservation for that time.

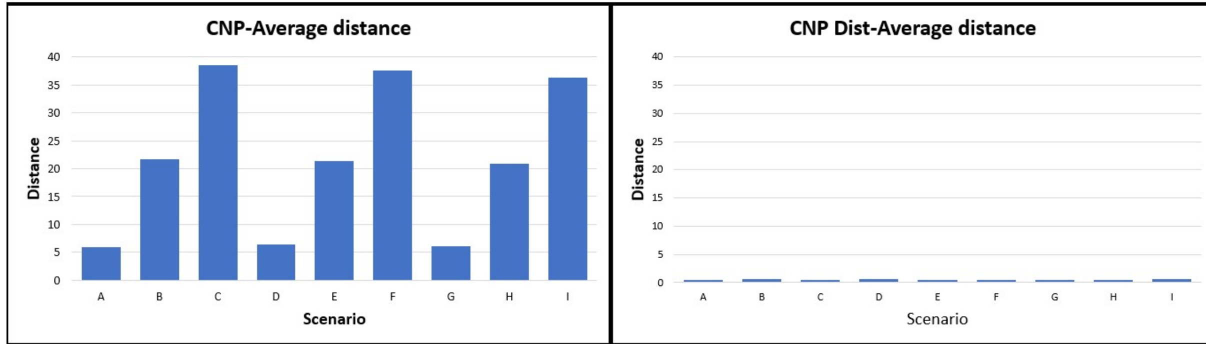


Figure 5.4: Results of the distance to the desired parking place for the CNP and CNPDist protocols divided per scenario.

### 5.2.3 Analysis of Operating Parameters

In addition to the analysis of the price and distance parameters, the average of some other parameters is also exposed in this section, namely, the number of messages exchanged in each negotiation strategy, the time expended in the negotiation in milliseconds and the difference between the highest and the lowest agreed prices. To make the table even more completed, the average price and distance of the four protocols were also included. The achieved results are summarized in Table 5.4 and 5.5.

Table 5.4: Average results related to the negotiation process for the four negotiation strategies part 1.

| | Messages | Time (ms) | Distance | Distance standard deviation |
|---|---|---|---|---|
| CNP | 224 | 15 | 22 | 25.46 |
| English | 1340 | 207 | 22 | 25.71 |
| Dutch | 520 | 42 | 22 | 26.12 |
| Faratin | 322 | 22 | 28 | 30.99 |

Table 5.5: Average results related to the negotiation process for the four negotiation strategies part 2.

| | Price | Difference max/min | Price standard deviation |
|---|---|---|---|
| CNP | 14 | 61 | 5.38 |
| English | 12 | 78 | 5.01 |
| Dutch | 14 | 76 | 5.97 |
| Faratin | 18 | 51 | 4.84 |

As observed, the number of exchanged messages between the agents during a negotiation process was significantly different between the four protocols, in favor of CNP. In fact, the number of exchanged messages in the CNP strategy is approximately only 17% of the number presented by the English Auction, 43% of the value presented by the Dutch Auction and 69% of the Faratin Auction. Consequently, the CNP strategy presents the lowest average time to obtain a parking spot, as initially expected. Surprisingly, the number of exchanged messages in the Dutch Auction is significantly lower than in the English Auction. That occurs because the convergence point is much closer to the initial price value established by the Dutch Auction than in the English Auction (note that the starting point in the Dutch Auction is 10 and for the English Auction it might be 40, 70 or 100 depending on the profile, and the convergence value is around 14, as illustrated in Table 5.5).

In spite of presenting the lowest average price, the English Auction has the highest difference between agreed prices (last column of the table). On the other hand, Faratin Auction is the most deterministic negotiation strategy.

Summarizing, the English Auction reached the best results regarding the agreed prices. But, it requires more resources to reach an agreement for the negotiation process, expressed in the highest number of the exchanged messages and the requested time to conclude the negotiation. The option for the best negotiation strategy may be dependent of the requirements, but the capability to conclude the negotiation faster can be in favor of the CNP strategy, since the difference in terms of price paid by the drivers is very reduced and it doesn't have such a huge difference in the prices as the English Auction.

# Chapter 6

# Prototype Implementation

This chapter aims to present a prototype created based on the base architecture presented in Chapter 3.1 (not using the security proposed methods) and the CNP protocol chosen in the Chapter 5 focusing on the communication between some interface. The figure 6.1 represents the relation with this proposed prototype with the rest of the work.

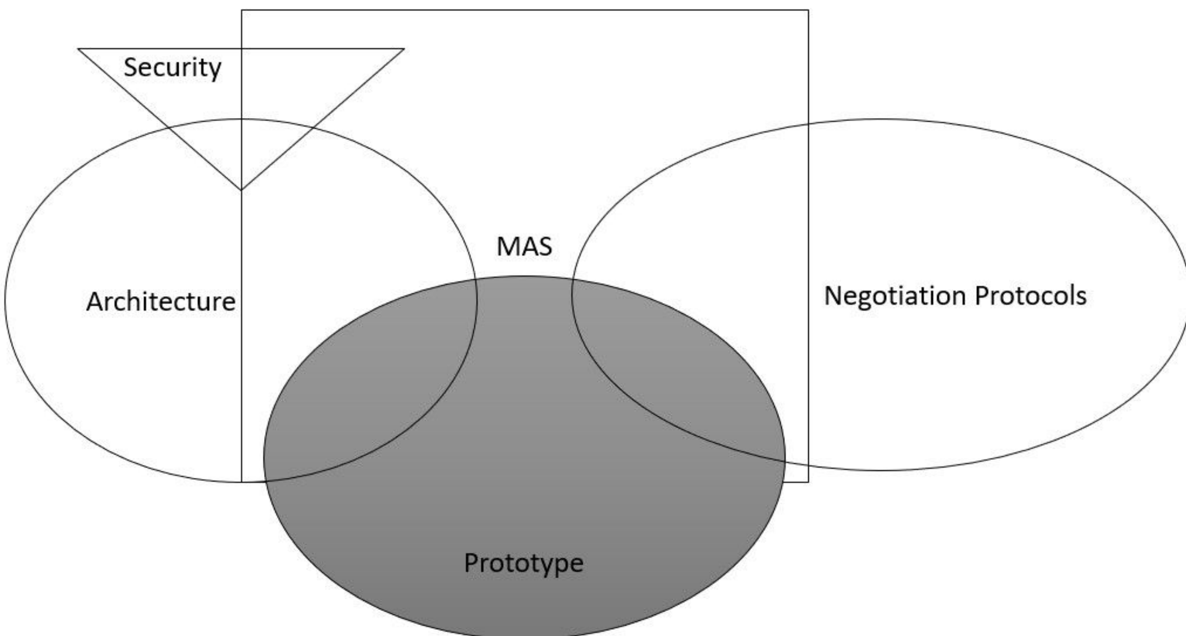


Figure 6.1: Chapter related to the work overview.

First of all, this Chapter will give an overview of the prototype. After, it will be

explained the communication with the interface. Three communications were chosen to be created, focusing on the core of the system, the negotiations itself. The first one refers to a AgentSpot being deployed in system (section 6.2). The second is the reservation of a AgentSpot by some Driver (section 6.3). And the last informs the reservations of a Driver (section 6.4). Following those examples, it is possible to create the remaining communications.

## 6.1 Base Architecture

A prototype of a Smart Parking for cars was created using Java with JADE. Not only that, but two more libraries were used: the JSON for standardizing the data and the MQTT to make the communication between the agents and the interface [12]. Those libraries are explained below. Furthermore, to simulate a mobile application, the extension for the browser Google Chrome MQTTBox Version 0.2.3 was used [34]. That is explained in the same section as the MQTT. After that, an overview of the structure is shown. Then, the main class is explained, to understand the working flow of the system as a whole.

Furthermore, all the agents in the system (AgentSpot and AgentDriver) have 2 standard names, one for the communication between them (the real name of them) that is "carDriverAgentX" for the AgentDrivers or "carSpotAgentX" for the AgentSpots, where the 'X' represents a number in both cases, its id. That id is how the agents are refereed all over the system. It is transparent the agents' real name for an outsider, all of them are referred by its id. Remember that a AgentSpot and a AgentDriver can have the same id. Because any agent communicates with the same type, all the messages have both ids, the AgentSpot and the AgentDriver.

The prototype assumes an interface capable of managing the Drivers and its AgentDriver. That is, there is not a verification on the "user_id". On the other hand, because of this assumption, the prototype accepts any Integer for the "user_id", but not two equals at the same time.

### 6.1.1 JSON

All the communication between entities uses JSON. A lightweight data-interchange format [35]. In other words, is a data format that makes easy to human read and the machine interpret. The library was the json-simple version 1.1 [36].

The messages follow the structure bellow where the keys represent the name of the information that wanted to be sent and the value represents the information itself. There can be N information on a single message.

```
{
    ``Name'' : String,
    ``Age'' : Integer,
    ...
}
```

The communications always happen between agents from different classes. Almost all messages have the variables bellow, with little changes depending on the protocol. Anyway, all the communication protocol will be explained:

- msg_id: The id of the message represents the action that an entity is trying to perform. With it, it's possible to keep track of the system flow.
- user_id: The id of the AgentDriver that is communicating, no matter if it is the receiver or the sender. This information might not be used when the AgentSpot requests to enter the system.
- spot_id: The id of the AgentSpot that is communicating, no matter if it is the receiver or the sender.
- name: This parameter is similar to the spot_id and the user_id, but, instead of the id given by a human. This id represents the name of the sender agent given by the system. In other words, is the address of the sender agent in the system.

More than those, the JSON messages might have more information in a single message. In fact, the previous items are just identifiers. The rest of the items is the core of the messages and represents the main information.

### MQTTBOX

MQTTBox is an extension for Google Chrome and other platforms that allows the user to be a publisher or a subscriber on a MQTT. To connect the application to the desired MQTT, the specifications shown in Figure 6.2 were made. The changes made in the default configuration were the MQTT Client Name, Host to inform the connection, Protocol to inform the method used to exchange the messages and, most importantly, the Quality of Service (QoS).

| MQTT Client Name | MQTT Client Id | Append timestamp to MQTT client id? | Broker is MQTT v3.1.1 compliant? |
| --- | --- | --- | --- |
| brunoChrome | fb2f6d07-6ff2-4f97-9db8-c6717b87a83 | Yes | Yes |
| **Protocol** | **Host** | **Clean Session?** | **Auto connect on app launch?** |
| mqtt / tcp | 193.136.195.56:1883 | Yes | Yes |
| **Username** | **Password** | **Reschedule Pings?** | **Queue outgoing QoS zero messages?** |
| Username | Password | Yes | Yes |
| **Reconnect Period (milliseconds)** | **Connect Timeout (milliseconds)** | **KeepAlive (seconds)** | |
| 1000 | 30000 | 10 | |
| **Will - Topic** | **Will - QoS** | **Will - Retain** | **Will - Payload** |
| Will - Topic | 2 - Exactly Once | No | |

Figure 6.2: Configurations to connect to some MQTT in MQTTBox.

The last attribute (QoS) represents the certainty of the message being delivered and is divided into three possible levels. The first and the fastest only sends one message. The second uses a two-steps handshake. The third and the slowest uses a four-steps handshake. Those levels are better-explained bellow.

- 0 (at most one): The message is only sent, no matter what happens. It doesn't have verification of delivery.
- 1 (at least one): The message is sent and verification is made to ensure that. In this case, multiple messages can be received.

- 2 (exactly one): The message is sent and verification is made to ensure that. But, there is guaranteed that only one message is received, not more. That's because the sender sends a confirmation of the confirmation. Only then the message is available for the entity.

As the system must have the guarantee of the connection and the agents must receive just one of the expected message, the level of QoS chosen was 2 (exactly one), even this process been the slowest.

### 6.1.2 Main Class

The class "Main2" creates two containers (JADE), one for each type of agent (AgentSpot and AgentDriver). More than that, it connects to the MQTT to receive a request for deployment.

The codes below are one of the most important parts of this class. It shows what happens when a request to create an agent arrives. When a Driver wants to make a reservation (msg_id = 1) and when a AdmSpot wants to deploy a AgentSpot into the system. In the first case, an AgentDriver must be called, but in this prototype, it's created to respond to the request, after that, it's deleted. On the other hand, the AgentSpots are created and stays in the system until the system is closed.

To deploy a AgentSpot into the system a message with the information of it must be delivered in the broker to the "Mains2" object deploy it. The code below represents the creation of the AgentSpot. The information about the new AgentSpot is cloned to introduce a new value, the "spot_id". This attribute is previously generated in the class following an order. After, this information is passed to the object that will be created by the "arg".

---

```
private void createCarSpot(int spot_id) {
    // concatenating attributes
    JSONObject attributes = (JSONObject) jsonInMQTT.clone();
```

```
    attributes.put("spot_id", spot_id);

    // create and save
    AgentController aux;
    Object arg[] = new Object[1];
    arg[0] = attributes;
    try {
        aux = carSpotController.createNewAgent("carSpotAgent" + spot_id,
            "agent.CarSpot", arg);
        aux.start();
        carSpots.add(aux);
    } catch (StaleProxyException e) {
        e.printStackTrace();
    }
}
```

Listing 6.1: Code: Creation of a AgentSpot

The code below shows the creation of the AgentDriver. The “user_id” variable represents the name for the system of that new AgentDriver. Furthermore, the information of the new agent is in the “jsonInMQTT” variable, received by this object by a Driver application for example represented by the MQTTBox in this work. Moreover, the size of the parking, that is, the number of available spots are also sending to its creation (“carSpot.size();”).

All the needed information goes to the new object by “arg”. After that, the agent is created in the car container and started in another thread or the system (“carDriver.start();”).

Notice, different from the AgentSpot the agent id is sent by the requested. That is, it is assumed some controller of the Driver is made outside of it.

```
private void createCarDriver() {
```

```
    String user_id = "carDriverAgent" + jsonInMQTT.get("user_id");

    // criar e guardar
    AgentController carDriver;
    Object arg[] = new Object[2];
    arg[0] = jsonInMQTT;
    arg[1] = carSpots.size();
    try {
        carDriver = carDriverController.createNewAgent(user_id,
            "agent.CarDriver", arg);
        carDriver.start();
    } catch (Exception e) {
        System.out.println(user_id + " could not be created, there is already
            this name in the system");
    }
}
```

Listing 6.2: Code: Creation of a AgentDriver

The distance of the agents and the map of the parking was made using an array. All the AgentSpot has a spot_id from 0 to N representing the index of that AgentSpot in the parking.

## 6.2 Deployment of AgentSpot

To deploy an AgentSpot in the system a protocol is followed. Bellow is the sequence of MQTT messages and what parameters each one must have to deploy it correctly. Notice, the "msg_id" is also part of the JSON inside the MQTT message. The AdmSpot is assumed to have an application to interact with the system. To simulate that, MQTTBox is used, but could be a mobile application for instance.

When the AgentSpot is created, it responds using MQTT with the "msg_id : 8".

Table 6.1: msg_id : “7”. This message is sent by the AdmSpot, and have the following parameters inside a JSON.

| Key | Value | Description |
|---|---|---|
| new_location | Boolean | It informs if it’s a new parking in the system. |
| location_id | Integer | It can represent different parkings in the system. If is a new location, this attribute is not important. |
| latitude | Double | Latitude. |
| longitude | Double | Longitude. |
| parking_type | String | It represents the vehicle the spot can park. |
| profile | String | In this work it was used “conservative”, “moderated” or “aggressive”. |
| average_price | Double | It represents the start price of the AgentSpot, but it is actualized with time. If it participates of 10 negotiations in a row without getting a client the price decreases. On the other hand, each time it wins the price increases. |

Table 6.2: msg_id : “8”. This message is sent by the new AgentSpot, and has the following parameters inside a JSON.

| Key | Value | Description |
|---|---|---|
| spot_id | Integer | All the AgentSpot have an unique spot_id. |
| location_id | Integer | Is a new location this value will be generated. Otherwise it is the same as the received. |
| answer | Boolean | It will inform if the AgentSpot was created or not. |
| reason | String | If the AgentSpot couldn’t be created, it will inform the reason. |

After that, it starts to receive requests for reservations. This process can be seen in the appendix F.

## 6.3 Make Reservation

The following protocol is made to a Driver makes a reservation. Bellow is the sequence of MQTT messages and what parameters each one must have to deploy it correctly. Notice, the “msg_id” is also part of the JSON inside the MQTT message. The Driver is assumed to have an application to interact with the system. To simulate that, MQTTBox is used, but could be a mobile application for instance.

More than those interactions with the front-end, interactions between the agents are made to perform the negotiation. For that, another protocol is made only regarding the interactions between the AgentDriver and the AgentSpots to reach a consensus in a CNP negotiation. They both can be seen in Figure 6.3.

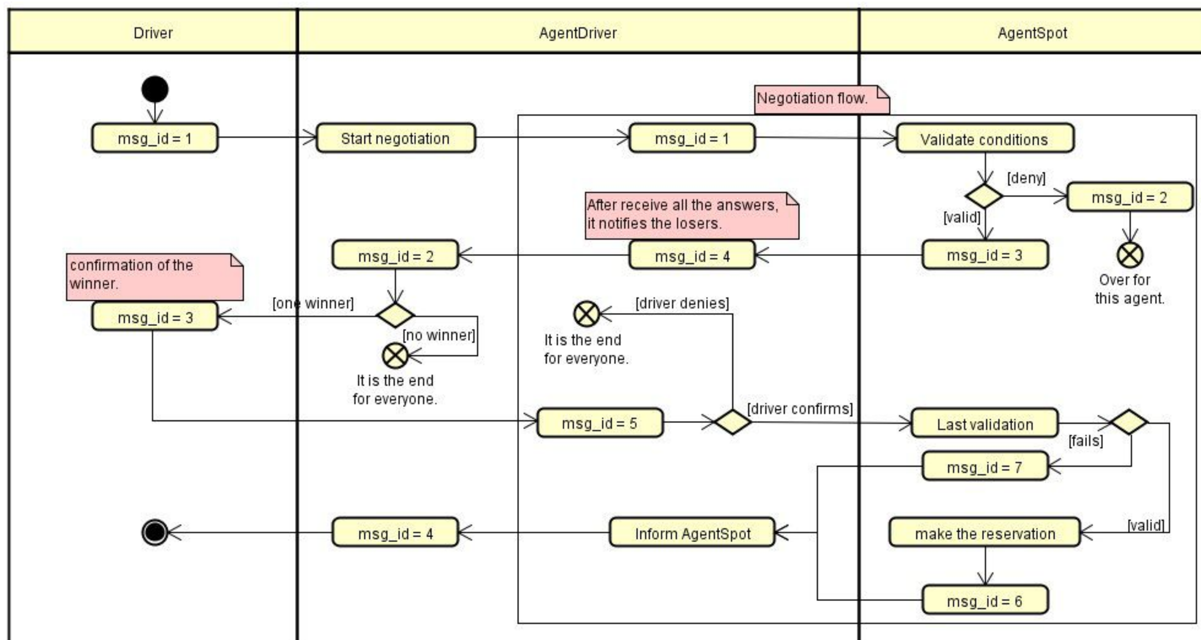


Figure 6.3: CNP and MQTT protocols interactions.

As the CNP was already explained before (Chapter 4), it will not be the focus. The

Table 6.3: msg_id : “1”. This message is sent by the Driver, and have the following parameters inside a JSON.

| Key | Value | Description |
|---|---|---|
| user_id | Integer | It represents the Driver and its AgentDriver. |
| spot_id | Integer | It represents the id of the wanted AgentSpot. |
| driver_type | String | driver_type |
| location_id | Integer | The parking it wants to park. At this point in the prototype it is not relevant. This part is ignored. |
| distance_range | Integer | It will inform the range around the wanted place that the Driver accepts. In other words, the search range of a AgentSpot. |
| latitude | Double | When the system is completed, it can be the GPS parameter. |
| longitude | Double | When the system is completed, it can be the GPS parameter. |
| start_year | Integer | Start year |
| start_month | Integer | Start month |
| start_day | Integer | Start day |
| start_hour | Integer | Start hour |
| start_minute | Integer | Start minute |
| end_year | Integer | End year |
| end_month | Integer | End month |
| end_day | Integer | End day |
| end_hour | Integer | End hour |
| end_minute | Integer | End minute |
| maximum_price | Integer | The minimum accepted in the system is 1 and the maximum is 15. |
| price_weigh | Integer | The importance of this parameter for the Driver in comparation with the others in percentage (0-100). |
| distance_weigh | Integer | The importance of this parameter for the Driver in comparation with the others in percentage (0-100). |

Table 6.4: msg_id : “2”. This message is sent by the AgentDriver to the Driver, and have the following parameters inside a JSON.

| Key | Value | Description |
|---|---|---|
| user_id | Integer | User ID |
| spot_id | Integer | If the negotiation fails for some reason it returns “-1”. For instance, all the AgentSpots in range of the search do not have this time available for a reservation. Otherwise it return the spot_id of the winner. |
| location_id | Integer | Location ID |
| price | Double | Price |
| negotiation_type | String | Negotiation Type |
| reason | String | If the negotiation fails, the reason is returned. Otherwise, is empty. |

Table 6.5: msg_id : “3”. This message is sent by the Driver, and have the following parameters inside a JSON.

| Key | Value | Description |
|---|---|---|
| user_id | Integer | User ID |
| spot_id | Integer | Spot ID |
| location_id | Integer | Location ID |
| answer | Boolean | The Driver may cancel the negotiation before it is completed. The Driver may not like the proposal. |
| driver_type | String | Negotiation Type |

Table 6.6: msg_id : “4”. This message is sent by the AgentDriver to the Driver, and have the following parameters inside a JSON.

| Key | Value | Description |
|---|---|---|
| user_id | Integer | User ID |
| spot_id | Integer | Spot ID |
| answer | Boolean | The AgentSpot can, for instance, make a reservation for another Driver. |

Diagram present in the Figure 6.3 is divided in three main entities, the Driver, the AgentDriver and the AgentSpot. The communication between the first two is made using the protocol described above. The communication between the agents refers to the CNP and is highlighted by the square.

Going further, all starts with the Driver sending a reservation request. Notice, the creation of the AgentDriver is implicit since it was explained in the previous session. The negotiation is performed by the AgentDriver and the AgentSpots that have the conditions for it. When a winner is selected or not, this information is sent to the Driver confirm the reservation ("msg_id = 2" inside the AgentDriver). If there is no winner after this information is delivered the agent is excluded.

Otherwise, the Driver can check the result and approve it or not. After AgentDriver send this information to the winner AgentSpot if the Driver denies it, the AgentDriver is excluded and the negotiation is over. On the other hand, if the Driver confirms it, the AgentSpot checks again its agenda. If it is all good, it makes the reservation. No matter what, it informs the Driver about it ("msg_id = 7" if not or "msg_id = 7" if it's all good). As for the AgentDriver it informs the Driver about the result and is deleted from the system finishing the process. This process can be seen in the appendix G.

## 6.4 Request Reservations

To the Driver request the reservation of a Driver in a parking a protocol is followed. Bellow is the sequence of MQTT messages and what parameters each one must have to deploy it correctly. Notice, the "msg_id" is also part of the JSON inside the MQTT message. The Driver is assumed to have an application to interact with the system. To simulate that, MQTTBox is used, but could be a mobile application for instance.

Table 6.7: msg_id : “5”. This message is sent by the AgentDriver to the AgentSpots, and have the following parameters inside a JSON.

| Key | Value | Description |
|---|---|---|
| user_id | Integer | User ID |
| location_id | Integer | Location ID |
| driver_type | String | Driver type. |

Table 6.8: msg_id : “6”. This message is sent by the AgentSpots to the AgentDriver, and have the following parameters inside a JSON.

| Key | Value | Description |
|---|---|---|
| user_id | Integer | User ID. |
| spot_id | Integer | Spot ID. |
| location_id | Integer | Location ID. |
| reservations | JSON[ ] | This JSON array contains the start date and the end date: {dateStart : “Date”, dateEnd : “Date”}. Each AgentSpot will delivery a separate message with its reservations. The format of the date is: “dd-MM-yyyy HH:mm”. |

## 6.5 Discussion

This chapter exemplified how to start an implementation focused on the communication between the MAS and the mobile application. Depending on the needs of further communications is possible to create other protocols following the examples as a base.

To do that, the messages need to have an ID and be delivered in an order to control the flow. Moreover, the *spot_id*, the *location_id*, *user_id*, *driver_type*, *spot_type* might be used to define the communication participants. The rest of the data depends on the propose of communication.

# Chapter 7

# Conclusions and Future Work

## 7.1 Conclusions

This work intended to study negotiation protocols for the Smart Parking System and proposed a suitable approach for the Smart Parking problem. As a result, we had shown the CNP is a good option to be used in general cases because it gets the agreement fast compared to the other protocols and needs to exchange the fewer amount of messages between agents compared with the other negotiation protocols. Even so, it has a price around the average.

But, other negotiation protocols can be used for different goals. For instance, the Faratin Auction can be used to get the highest prices and the English Auction to get the lowest prices. Moreover, the Faratin Auction can be used to get the further distance to the desired spot while the English Auction can be used to stress the system since it has a higher number of messages exchanged and it takes longer than the others.

The architecture described in UML Diagrams can be used to develop a Smart Parking System based on MASs. It shows the base of the system, and also, some suggestions to improve the system like the option to create a promotion and the security module focused on the message exchange.

Furthermore, the JADE framework was able to handle the system and organize the

agents very well in containers. That could be useful when the security module is implemented to divide the agents into regions. Despite that, the framework is not necessary, it is possible to follow the UML Diagrams alone to implement the same system.

Not only that, the system can be adapted using the described parameters and diagrams to create other scenarios, for instance, a 24 hours parking. The base of the system is the same. Just like the scenarios, the module itself can be used in other cases that need to have security in the communication between agents, that is, can be used in any MAS.

The work has its focus on the car parking case, but it can be used to other vehicles or even the use of multiple vehicles like a harbor where it is needed to have several types of ships, like cargo ships and cruise ships.

In the end, we expose protocol examples of the communication between the MAS and the user interfaces using MQTT. Using that as a base, create the other communications may be easier.

Finally, the work had shown how to create a Smart Parking System using CNP, from the creation of the agents to the communications involving the MAS part. Moreover, it is possible to update and upgrade the system to other scenarios and cases.

## 7.2 Future Works

The next works will be devoted to testing the base and the security-focused architecture to analyze the performance in both cases. Not over, the possibility to implement some design patterns in the architectures should be analyzed. Going further in the architecture, a more focus on the AI could be very useful, making the system more autonomous and competitive.

Subsequently, testing the negotiation strategies for the Smart Parking problem using the holonic principles, e.g., considering holarchies of Drivers or Spots, which may influence the performance of the negotiation process. Not only that, but more protocols should be studied. Also, the analysis of a 24 hours parking scenario should be considered, discarding the initial results and running the systems for a longer period.

Following this, a test in real scenarios should be done. Moreover, an interface for the AdmSpot focusing on the management of multiple AgentSpot and for the Driver focused on the management of owned vehicles and reservations should be done.

## 7.3 Published works

During the execution of this work, the following papers were presented and published:

- Bruno Rafael Alves, André Pinz Borges, Gleifer Vaz Alves and Paulo Leitão, Security in multi agent systems, WPCCG, pp. 17-23, 2019 [37].
- Bruno Rafael Alves, André Pinz Borges, Gleifer Vaz Alves and Paulo Leitão, Experimentation of Negotiation Protocols for Consensus Problems in Smart Parking Systems, HoloMAS2019, pp. 189-202, 2019 [38].

# Bibliography


[1] U. Nation, *68% of the world population projected to live in urban areas by 2050, says un.* [Online]. Available: `https://www.un.org/development/desa/en/news/population/2018-revision-of-world-urbanization-prospects.html` (visited on 08/21/2019).

[2] P. Neirottia, A. D. Marcob, A. C. Caglianoc, G. Manganod, and F. Scorranoe, "Current trends in smart city initiatives: Some stylised facts", *Cities*, vol. 38, pp. 25–36, 2014.

[3] L. Anthopoulos and P. Fitsilis, "Digital cities: Towards connected citizens and governance", in *Politics, Democracy and E-Government*, 2010, pp. 275–291.

[4] "Motorists spend four days a year looking for a parking space". [Online]. Available: `https://www.telegraph.co.uk/news/2017/02/01/motorists-spend-four-days-year-looking-parking-space/` (visited on 09/24/2019).

[5] A. S. Tanenbaum and M. V. Steen, *Distributed Systems Principles and Paradigms.* Createspace Independent Publishing Platform, 2016.

[6] M. Wooldridge and N. R. Jennings, "Intelligent agents: Theory and practice", *The knowledge engineering review*, vol. 10, no. 2, pp. 115–152, 1995.

[7] S. J. Russell and P. Norvig, *Artificial intelligence: a modern approach.* Malaysia; Pearson Education Limited, 2016.

[8] R. Olfati-Saber, J. A. Fax, and R. M. Murray, "Consensus and cooperation in networked multi-agent systems", *IEEE*, vol. 95, no. 1, pp. 215–233, 2007.

[9] C. Di Napoli, D. Di Nocera, and S. Rossi, “Agent negotiation for different needs in smart parking allocation”, in *International Conference on Practical Applications of Agents and Multi-Agent Systems*, Springer, 2014, pp. 98–109.

[10] H. Wang, “A reservation-based smart parking system”, PhD thesis, University of Nebraska - Lincoln, 2011.

[11] M. Beer, M. D’inverno, M. Luck, N. Jennings, C. Preist, and M. Schroeder, “Negotiation in multi-agent systems”, *The Knowledge Engineering Review*, vol. 14, no. 3, pp. 285–289, 1999.

[12] U. Hunkeler, H. L. Truong, and A. Stanford-Clark, “Mqtt-s—a publish/subscribe protocol for wireless sensor networks”, in *2008 3rd International Conference on Communication Systems Software and Middleware and Workshops (COMSWARE’08)*, IEEE, 2008, pp. 791–798.

[13] F. Bellifemine, A. Poggi, and G. Rimassa, “Jade a fipa compliant agent framework”, in *Proceedings of the Practical Applications of Intelligent Agents (PAAM’99)*, 1999, pp. 97–108.

[14] F. Bellifemine, G. Caire, and D. Greenwood, *Developing multi-agent systems with JADE*, ser. Wiley Series in Agent Technology. John Wiley & Sons, 2007, ISBN: 9780470058404.

[15] *Java*. [Online]. Available: `https://www.java.com/en/` (visited on 08/21/2019).

[16] *Foundation for Intelligent Physical Agents(FIPA)*. [Online]. Available: `http://www.fipa.org/index.html` (visited on 08/20/2019).

[17] *Institute of electrical and electronics engineers*. [Online]. Available: `https://www.ieee.org/` (visited on 08/20/2019).

[18] M. Wooldridge, *An Introduction to MultiAgent Systems*. John Wiley & Sons, 2002, ISBN: 047149691X.

[19] H. Young, *Negotiation Analysis*. University of Michigan Press, 1991, ISBN: 9780472081578.

[20] R. G. Smith, "The contract net protocol: High-level communication and control in a distributed problem solver", *Transactions on Computers*, vol. C-29, no. 12, pp. 1104–1113, 1980.

[21] K. Omote and A. Miyaji, "A practical english auction with one-time registration", in *Information Security and Privacy*, ser. Lecture Notes in Computer Science, vol. 2119, 2001, pp. 221–234.

[22] T. E. Rockoff and M. Groves, "Design of an internet-based system for remote dutch auctions", *Internet Research*, vol. 5, pp. 10–16, 1995.

[23] P. Faratin, C. Sierra, and N. R. Jennings, "Negotiation decision functions for autonomous agents", *Robotics and Autonomous Systems*, vol. 24, no. 3-4, pp. 159–182, 1998.

[24] *Eclipse foundation*. [Online]. Available: `https://www.eclipse.org` (visited on 08/21/2019).

[25] *Jade architecture image*. [Online]. Available: `https://jade.tilab.com/documentation/tutorials-guides/jade-administration-tutorial/architecture-overview/` (visited on 08/21/2019).

[26] O. M. Group, *Omg unified modeling language (omg uml)*, `https://www.omg.org/spec/UML/`, version 2.5.1, Dec. 2017.

[27] M. Shand and J. Vuillemin, "Fast implementations of rsa cryptography", in *Proceedings of IEEE 11th Symposium on Computer Arithmetic*, IEEE, 1993, pp. 252–259.

[28] J. Katz, A. J. Menezes, P. C. Van Oorschot, and S. A. Vanstone, *Handbook of applied cryptography*. CRC press, 1996.

[29] J. Daemen and V. Rijmen, *The design of Rijndael: AES-the advanced encryption standard*. Springer Science & Business Media, 2013.

[30] X. Zhang and K. K. Parhi, “High-speed vlsi architectures for the aes algorithm”, *IEEE transactions on very large scale integration (VLSI) systems*, vol. 12, no. 9, pp. 957–967, 2004.

[31] A. Freier, P. Karlton, and P. Kocher, “The secure sockets layer (ssl) protocol version 3.0”, 2011.

[32] I. Seilonen, “Distributed and collaborative production management systems in discrete part manufacturing: A review of research and technology”, *VTT Research Notes, Espoo*, 1997.

[33] *Brazilian spot size*. [Online]. Available: `http://www.brasilpark.com.br/tecnico/vaga` (visited on 09/26/2019).

[34] *Mqttbox*. [Online]. Available: `http://workswithweb.com/mqttbox.html` (visited on 08/21/2019).

[35] *Introducing json*. [Online]. Available: `https://www.json.org` (visited on 08/21/2019).

[36] *A simple java toolkit for json*. [Online]. Available: `https://github.com/fangyidong/json-simple` (visited on 08/21/2019).

[37] B. R. Alves, G. V. Alves, A. P. Borges, and P. Leitão, “Security in multi agent systems”, *Workshop de Pesquisa em Computação dos Campos Gerais (WPCCG)*, vol. 3, pp. 17–23, 2019.

[38] B. R. Alves, A. P. Borges, G. V. Alves, and P. Leitão, “Experimentation of negotiation protocols for consensus problems in smart parking systems”, *HoloMAS2019*, 2019.

# Appendix A

# Original Project Proposal

**Proposta de tema para**
**Dissertação/Estágio/Projeto - Trabalho de Conclusão de Curso**

Orientador da Instituição onde se realiza o trabalho:

| Paulo Leitão + José Eduardo Fernandes | pleitao@ipb.pt + jef@ipb.pt |
|---|---|

Instituição do orientador:

| Instituto Politécnico de Bragança - IPB | Escola Superior de Tecnologia e Gestão - ESTiG |
|---|---|

Co-orientador da Instituição parceira:

| Gleifer Vaz Alves + André Pinz Borges | gleifer@utfpr.edu.br + apborges( |
|---|---|

Instituição do co-orientador:

| Universidade Tecnológica Federal do Parana - U | Campus de Ponta Grossa |
|---|---|

Curso ou cursos da Instituição do orientador onde se propõe que o trabalho seja realizado:

| Mestrado em Sistemas de Informação |
|---|

Título do trabalho:

| Estudo e implementação de protocolos para problemas de consenso em sistemas de bizantinos |
|---|

Palavras chave:

| Sistemas multi-agente; problemas de consenso; protocolos de cooperação |
|---|

Objetivos:

O projeto SmartParking visa o desenvolvimento de um sistema ciber-fisico, baseado em sistemas multiagente, para o aparcamento inteligente em parques de estacionamento de carros e de bicicletas. Este tipo de sistemas é constituído por inúmeros dispositivos que possuem inteligência e autonomia, e na qual o comportamento global emerge da interação entre os dispositivos individuais. Neste sentido, pretende-se estudar, implementar e comparar protocolos de negociação aplicados ao caso de estudo.

Descrição adicional:

Os sistemas multiagentes, oriundos da inteligência artificial distribuída, são constituídos por um conjunto de agentes inteligentes e autónomos, representando os componentes físicos e lógicos de um sistema, que cooperam de forma a atingir os seus objetivos. A conceção dos protocolos de cooperação, seja em termos de colaboração, negociação ou outro, assume crucial importância na eficácia do sistema de software resultante, sendo particularmente dependente do aumento de escala em termos de utilizadores e nós ligados.
Assim, pretende-se que, de entre um conjunto de alternativas, se estude, analise e compare possíveis abordagens para solucionar problemas de consenso em sistemas de bizantinos. Com base no conhecimento adquirido, deverá ser proposta uma estratégia de negociação que melhor se adapte aos requisitos do sistema de aparcamento inteligente, e desenvolvido um protótipo que implemente e explore a estratégia definida.

Metodologia/Plano de trabalhos:

Tarefa 1 - Familiarização do sistema multi-agente de aparcamento inteligente e definição dos requisitos em termos de protocolos de negociação
Tarefa 2 - Estudo de protocolos de negociação existentes na literatura
Tarefa 3 - Implementação de diversos protocolos de negociação
Tarefa 4 - Teste e comparação dos protocolos de negociação implementados
Tarefa 5 - Definição da estratégia de negociação e implementação em protótipo
Tarefa 6 - Testes e validação do protótipo
Tarefa 7 - Escrita da dissertação e defesa final do trabalho

Recursos necessários:

# Appendix B

# Base Use Case Diagram

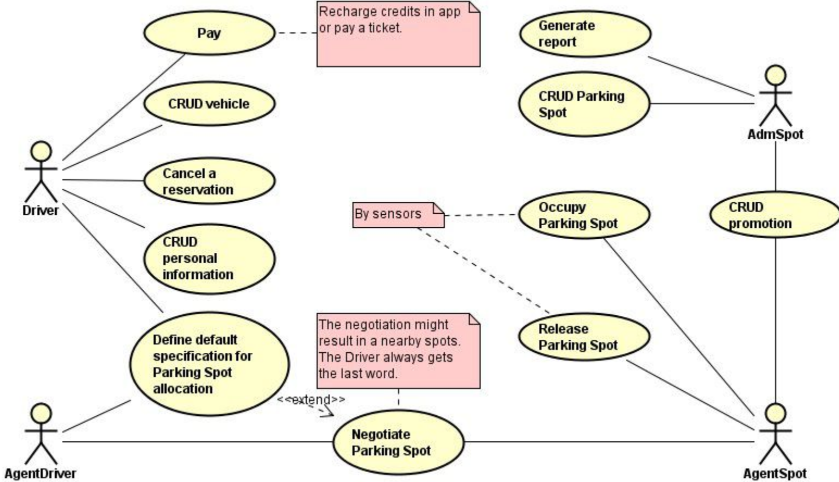


Figure B.1: Use case diagram.

# Appendix C

# Base Class Diagram

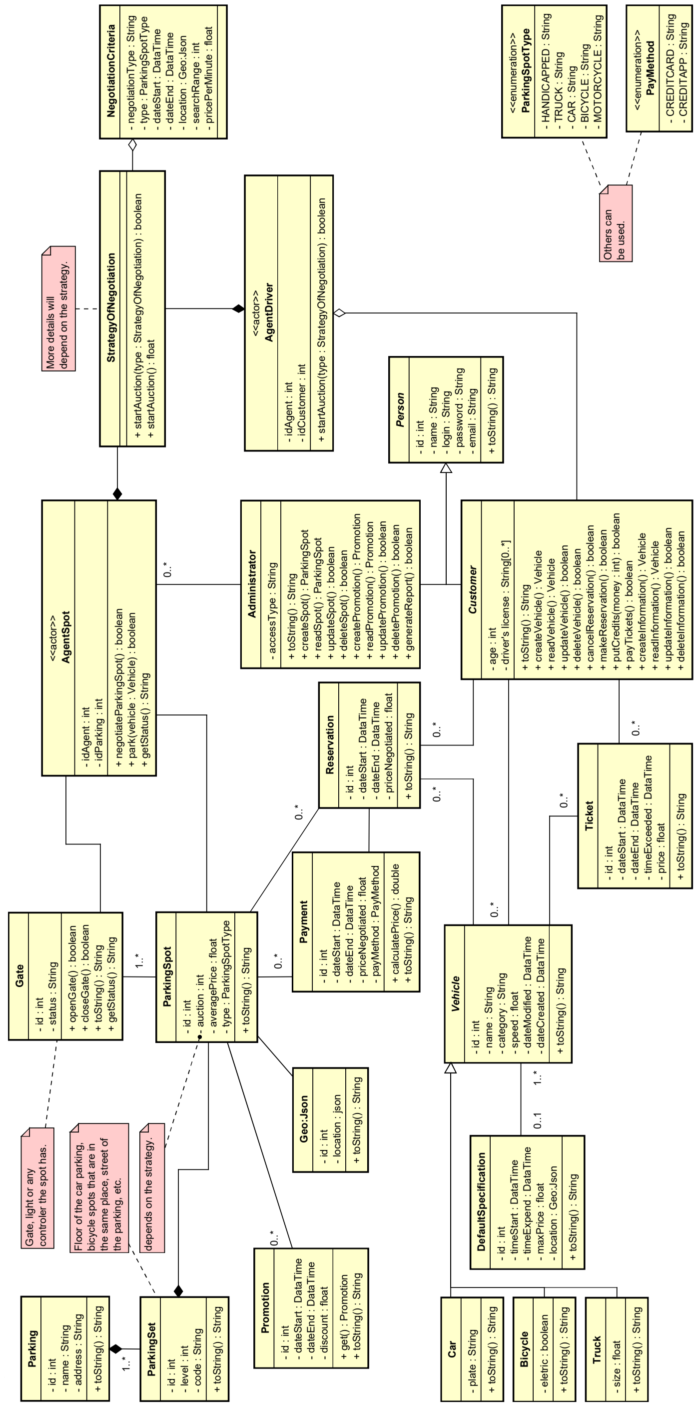


Figure C.1: Class diagram.

# Appendix D

# Base Activity Diagrams

## D.1 CRUD

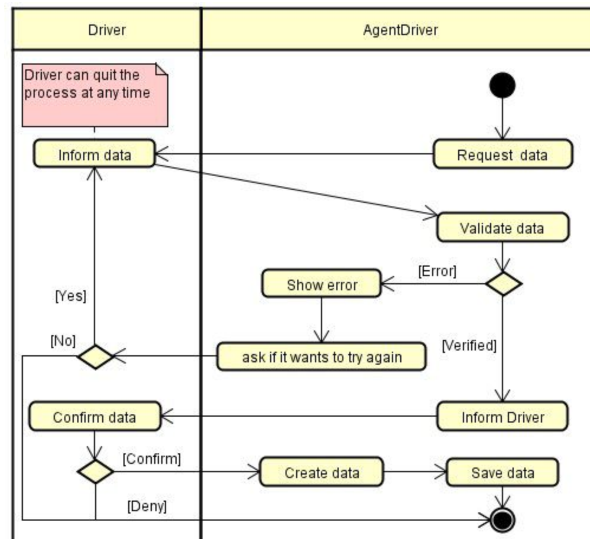


Figure D.1: Create.

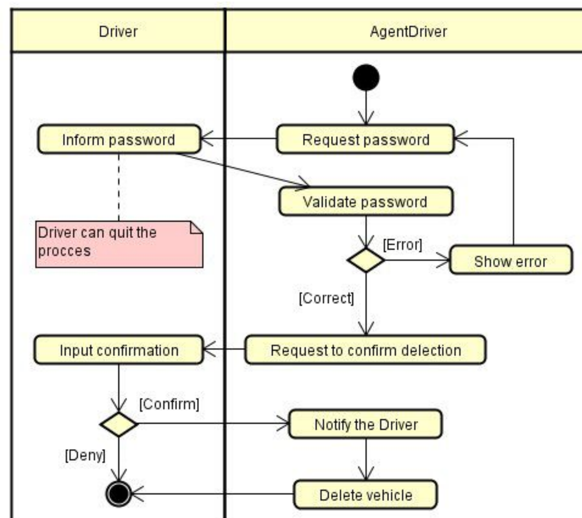


Figure D.2: Delete.

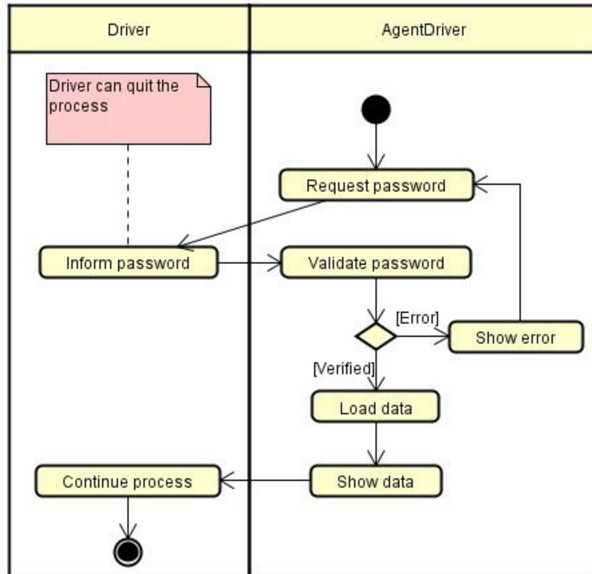


Figure D.3: Read information and parking spot.

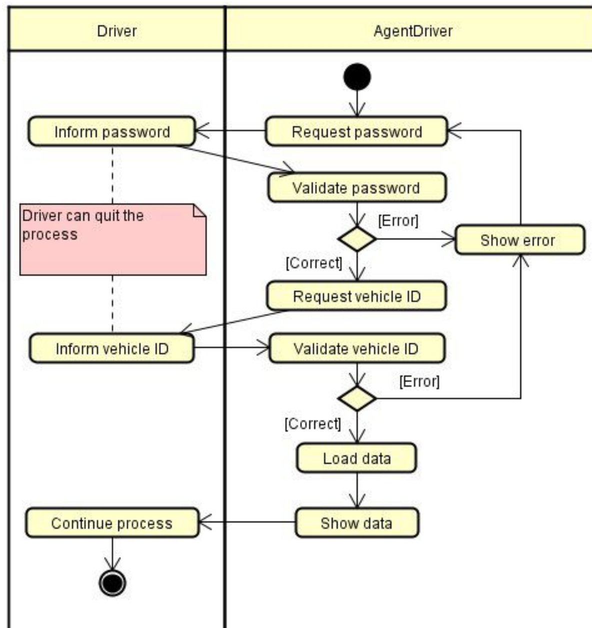


Figure D.4: Read vehicle and promotion.

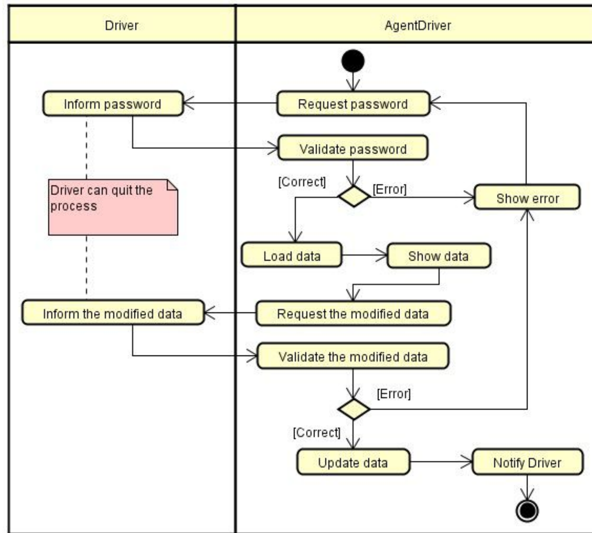


Figure D.5: Update information and parking spot.

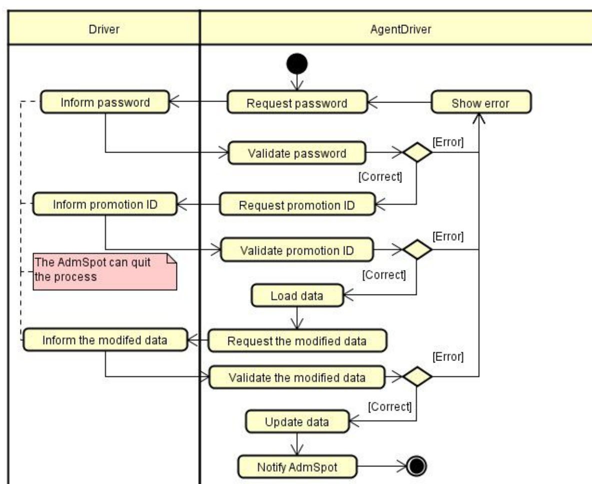


Figure D.6: Update vehicle and promotion.

## D.2 PAY

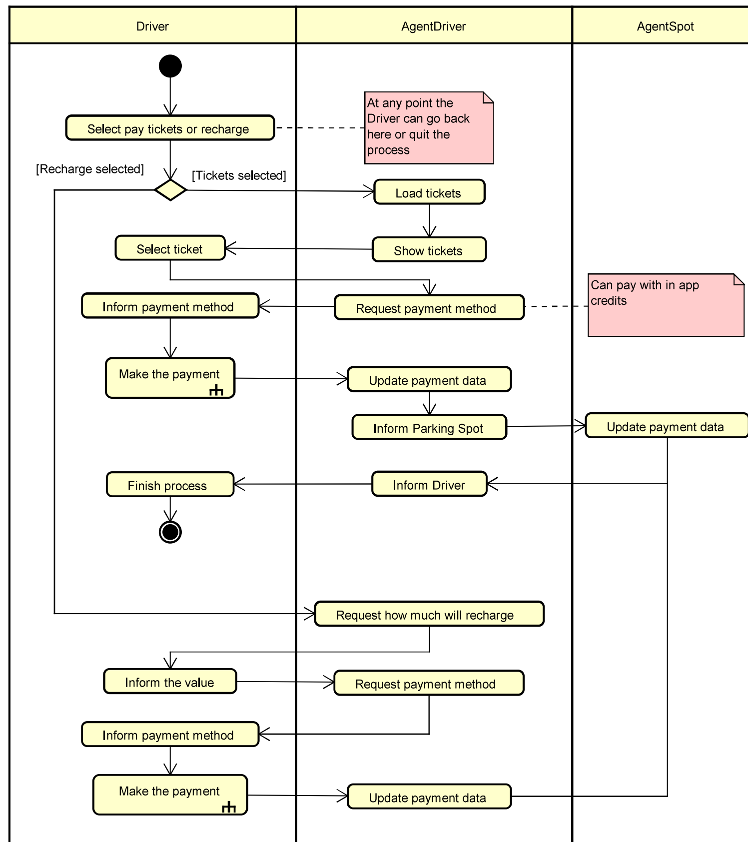


Figure D.7: Pay.

## D.3 DRIVER

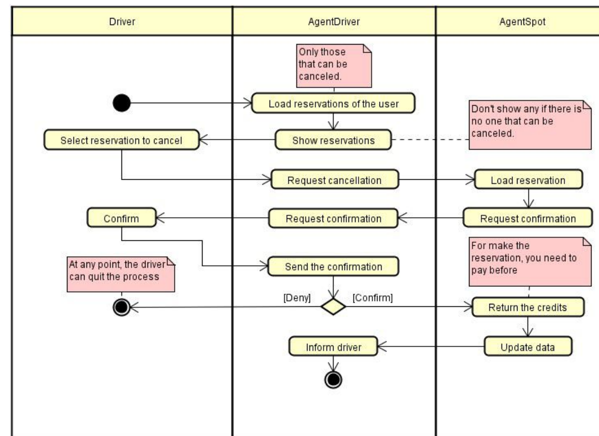


Figure D.8: Cancel a reservation.

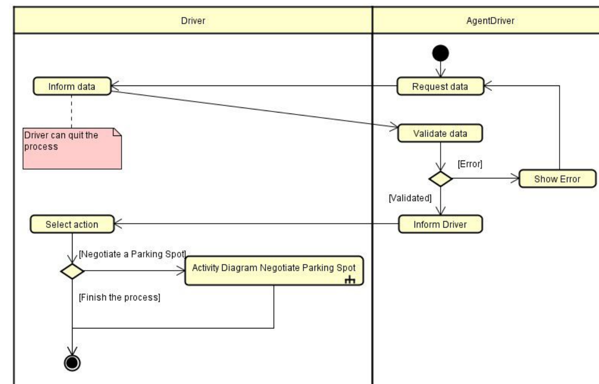


Figure D.9: Define default specification for parking spot allocation.

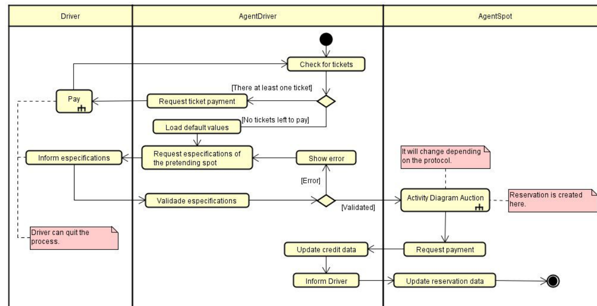


Figure D.10: Negotiate parking spot.

## D.4 PARKING

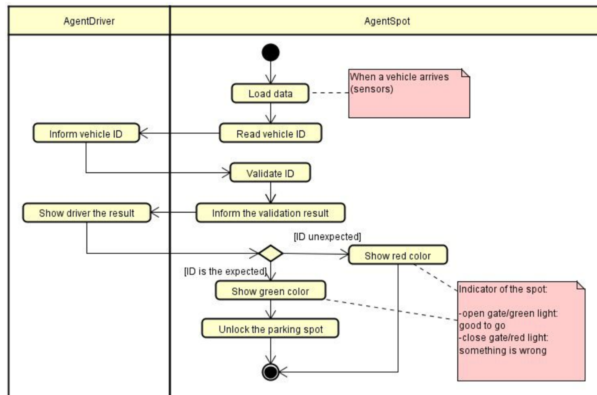


Figure D.11: OCCUPY PARKING SPOT.

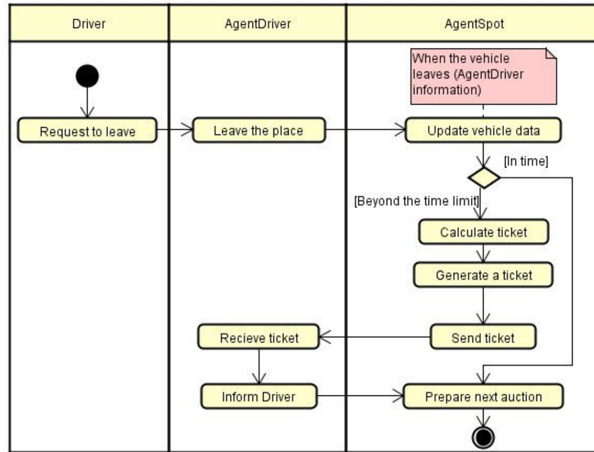


Figure D.12: RELEASE PARKING SPOT.

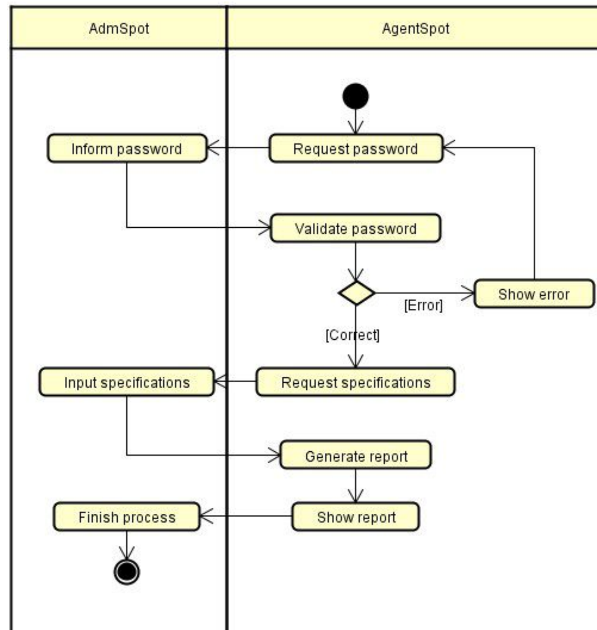


Figure D.13: GENERATE REPORT.

# Appendix E

# Security Class Diagram

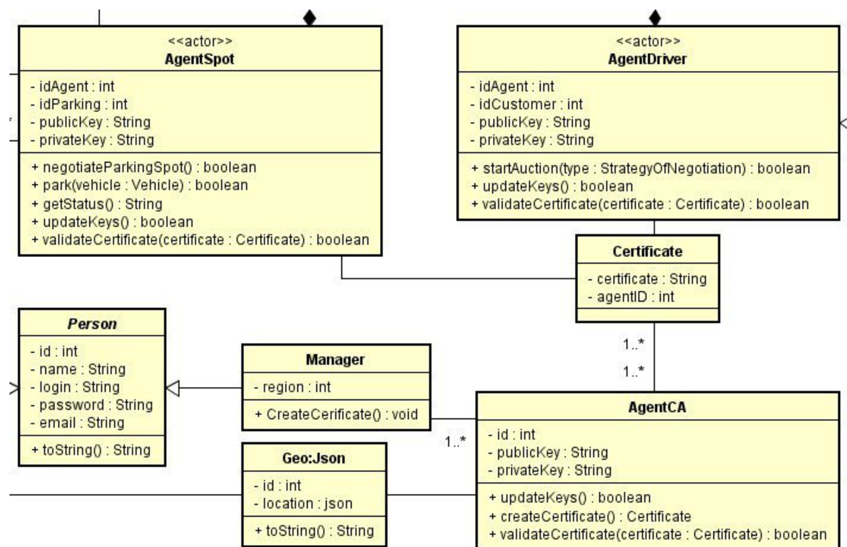


Figure E.1: Class diagram focused in security.

# Appendix F

# Deployment of AgentSpots demonstration

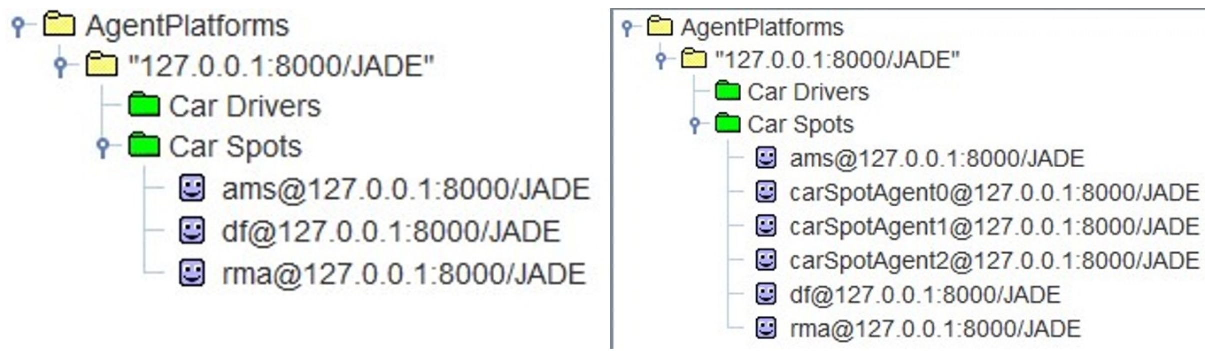


Figure F.1: Initial and final using JADE GUI.

Figure F.2: Message flow. Read from bottom to top.

# Appendix G

# Make Reservation demonstration

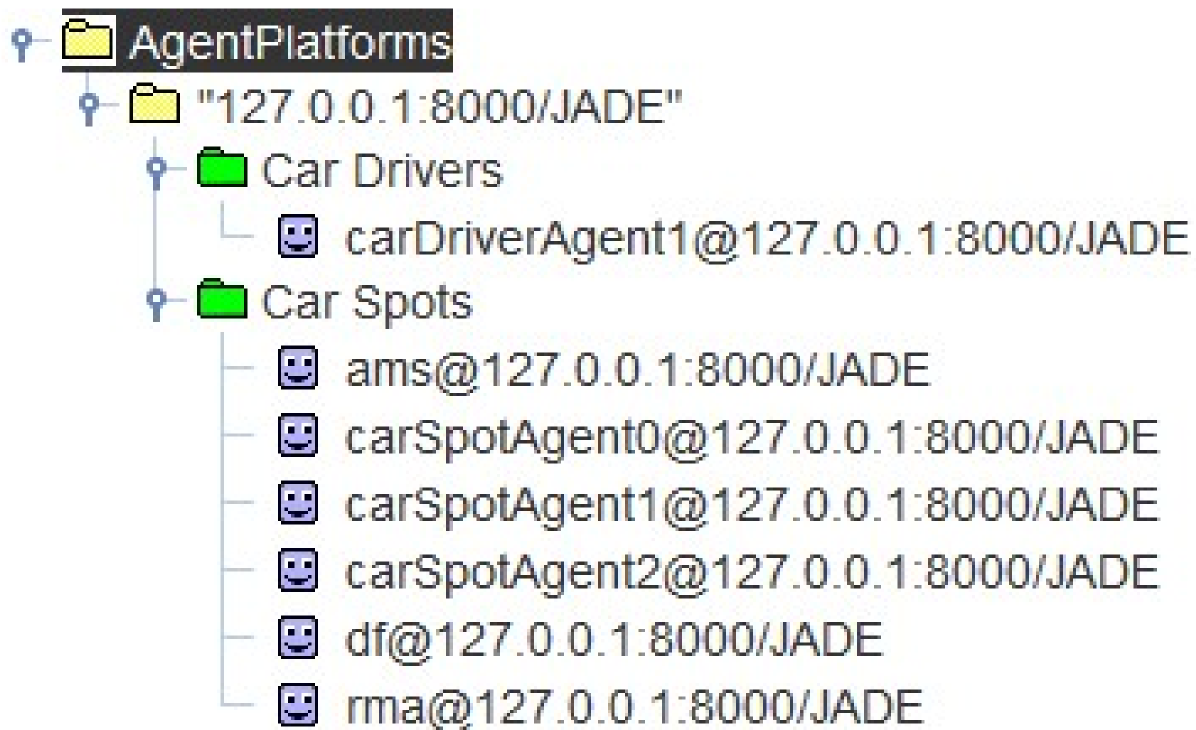


Figure G.1: Middle state using JADE GUI. Notice, the carDriverAgent is in another container.

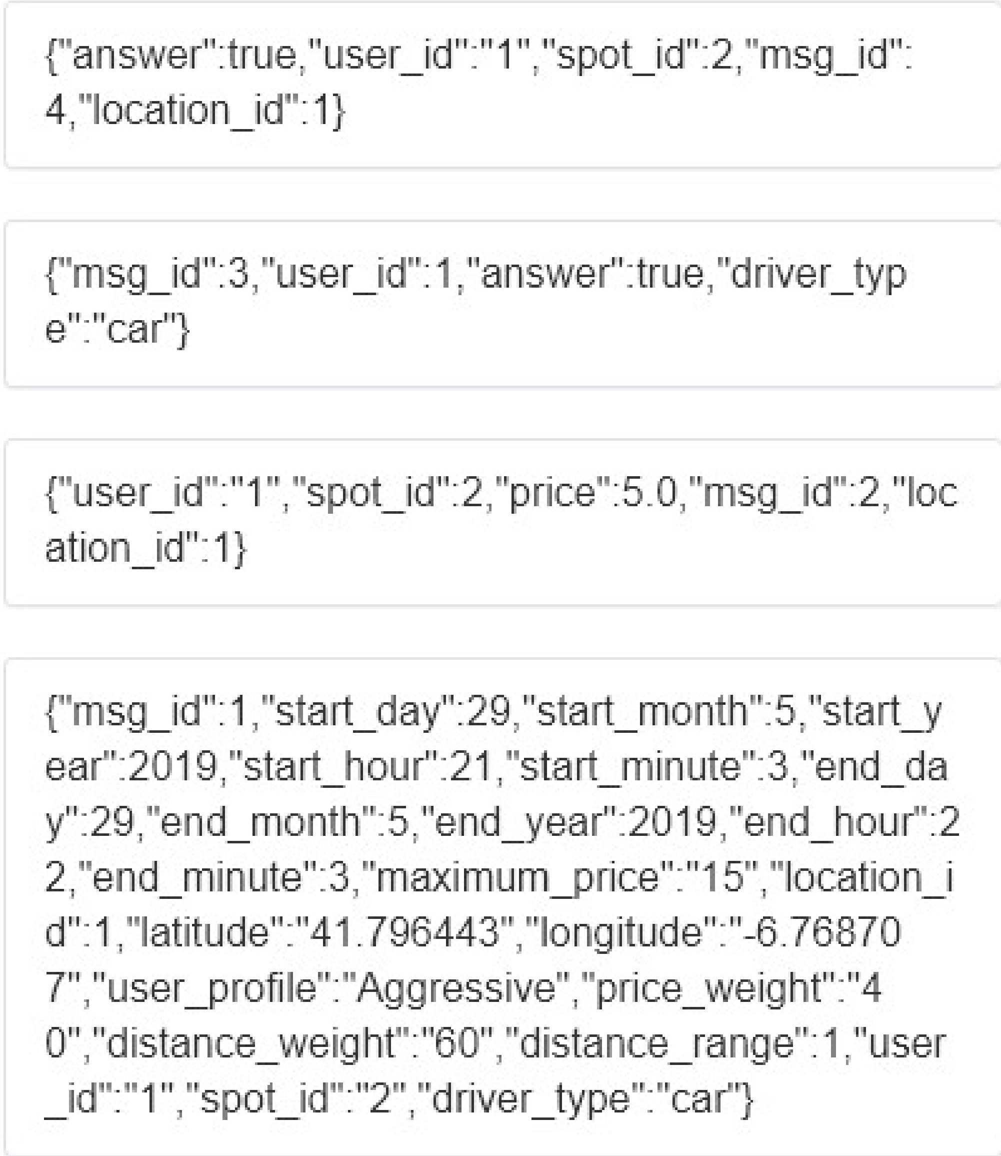


Figure G.2: Message flow. Read from bottom to top.

# Appendix H

# Request Reservations demonstration

{"reservations":[{"dateStart":"29-05-2019 21:03","dateEnd":"29-05-2019 22:03"}],"user_id":"1","spot_id":2,"msg_id":6,"location_id":1}

{"msg_id":5,"user_id":1,"location_id":1,"driver_type":"car"}

Figure H.1: read from bottom to top.