Fernando Jorge Varandas Anão Rosado

Licenciado em Ciências da Engenharia Electrotécnica e de Computadores

A Model-driven Architecture for Multi-protocol OBD Emulator.

Dissertação para obtenção do Grau de
Mestre em Engenharia Electrotécnica e de Computadores

Orientador:	Prof. Doutor Pedro Miguel Maló, Professor Auxiliar, FCT-UNL
Co-orientador:	Mestre Edgar Miguel Felício Oliveira da Silva, Investigador, UNINOVA-CTS

Júri:

Presidente:	Doutor Tiago Oliveira Machado de Figueiredo Cardoso - FCT/UNL
Arguente:	Doutor Rui Manuel Leitão Santos Tavares - FCT/UNL
Vogal:	Doutor Pedro Miguel Negrão Maló - FCT/UNL

FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

Setembro, 2016

A Model-driven Architecture for Multi-protocol OBD Emulator.

Aos meus avós, pais
e irmão

ACKNOWLEDGEMENTS

Para começar, gostaria de expressar a minha gratidão ao meu orientador Pedro Maló e co-orientador Edgar Silva pela oportunidade de desenvolver esta dissertação sob a sua supervisão. Todo o tempo, apoio e recursos fornecidos foram essenciais para o desenvolvimento do trabalho apresentado, mas também para a partilha de conhecimentos valiosos.

Aos meus colegas da FCT-UNL e Politechnika Poznańska, pelo espírito de entreajuda e cooperação nas mais diversas situações e pelo convivio proporcionado. Entre eles gostaria de destacar o António Furtado, Cláudio Penedo, Filipe Melo, André Brígida, Miguel Alves, Hugo Frazão, Franscisco Vieira, Nuno de Castro, Mauro Marques, Rúben Anjos, Pedro Sousa, Karim Jalal.

Gostaria também de prestar uma palavra de apreço a toda a minha equipa de natação, CIRL, pelo apoio, perseverança e garra que instruem todos os dias, servindo como escola para as situações do dia a dia. Entre eles não posso deixar de destacar o meu treinador, Alexandre Serrasqueiro, como um exemplo de incansável motivação, dedicação e atitude. Um especial agradecimento à família Pires pelo apoio e preocupação ao longo destes anos.

Aos meus amigos de sempre, também um grande obrigado por acreditarem em mim e estarem sempre do meu lado, tal como todas as outras pessoas que contribuiram para chegar até aqui. Entre eles gostaria de destacar o Manuel Dordio, João Carvalho, Mariana Linhan, Cláudia Marques, Inês Ribeiro, Susana Escária, João Chora, Daniel Lopes (um especial obrigado pela revisão da minha dissertação), João Fialho, João Mósca, Patrícia Magro, Vanessa Magro e fiel Zazu.

Um agradecimento muito especial ao Fernando Brito e Silva por ser um irmão no verdadeiro sentido da palavra e à Ana Teresa Maiorgas por todo o apoio incondicional, entreajuda e momentos passados ao longo deste ultimo ano. Obrigado Ana.

Para terminar, a toda a minha familia, em especial ao meu Pai e Mãe, Fernando Rosado e Cecília Varandas, ao meu irmão Miguel Rosado e Avós Armindo, António, Isabel e Catarina, obrigado por tudo. Por todo o carinho, motivação e apoio dado ao longo da minha vida. Muito Obrigado.

ABSTRACT

The Internet of Things (IoT) might be the next revolutionary technology to mark a generation. It could have a particularly strong influence on the automotive industry, changing people's perception of what a vehicle can do. By connecting several things in a car, IoT empowers it to sense and communicate. Furthermore, this technology clearly opens the way to emerging applications such as automated driving, Vehicle-to-Vehicle and Vehicle-to-Infrastructure communication.

Vehicle's information about its environment and surroundings is crucial to the development of existing and emerging applications. It is already possible to communicate directly (on-site) with vehicles through a built-in On Board Diagnostics (OBD), making it possible to obtain crucial information about the state of the vehicle in real environments. However, there is zero tolerance for error when developing new applications for vehicles that are, a priori, extremely costly and that must also safeguard human lives. Therefore, there is an increasing need for OBD emulators which can allow the development of new applications.

This Thesis proposes a model-driven architecture for multi-protocol OBD emulator, encouraging the development of new emerging OBD systems in a safety environment, to promote the creation of applications to interact or use vehicles' data. In this sense, the addressed specifications are: Less expensive comparing with today's solutions; Compatible with different OBD protocols communication; Open Source Hardware and Software suitable for Do-It-Yourself (DIY) development.

Keywords: Internet of Things, On Board Diagnostics, Interoperability, Emulation, Open Source.

RESUMO

A Internet das Coisas poderá ser a próxima revolução tecnológica a marcar uma geração. O IoT poderá ter uma influência particularmente forte na indústria automóvel, mudando a percepção das pessoas á cerca do que um veículo pode fazer. Ao ligar várias coisas em um carro, o IoT habilita-o a sentir e comunicar. Além disso, esta tecnologia abre claramente caminho para aplicações em desenvolvimento, como por exemplo: condução autónoma, comunicação veículo-veículo e comunicação veículo-infra-estrutura.

A informação de um veículo acerca do seu ambiente circundante é crucial para o desenvolvimento de aplicações existentes e emergentes. Já é possível comunicar directamente com veículos através de um On Board Diagnostics (OBD), possibilitando assim obter informações importantes relativas ao estado do veículo, em ambientes reais. Contudo, não existe tolerância para erros no que diz respeito a novas aplicações utilizadas em veículos, visto que são a priori dispendiosos e devem salvaguardar vidas humanas. Por conseguinte, existe uma necessidade crescente de emuladores OBD que permitam o desenvolvimento de novas aplicações.

Esta tese propõe uma arquitectura de modelos para emuladores OBD multi-protocolares. Encoraja o desenvolvimento de sistemas OBD emergentes em um ambiente de segurança, de modo a promover a concepção de aplicações para interagir ou usar dados de veículos. Neste sentido, as especificações que foram abordadas são: menos dispendioso quando comparado com as soluções actuais; Compatível com protocolos de comunicação OBD; Hardware e Software em código aberto adequado para desenvolvimento Faça você mesmo (Do-it-Yourself (DIY)).

Palavras-chave: Internet das Coisas, On Board Diagnostics, Interoperabilidade, Emulação, Código Aberto.

CONTENTS

List of Figures

List of Tables

GLOSSARY

ABS Antilock Braking System4.

ACK Acknowledges.

CAN Controller Area Network.

CEM Communication Enabler Module.

CIM Computation Independent Model.

CPH Communication Protocols Handler.

CPU Central Processing Unit.

CRC Cyclic Redundancy Code.

DF Data Format.

DLC Data Link Connector.

DTC Diagnostic Trouble Codes.

ECM Engine Control Module.

ECU Electronic Control Unit.

EOF End Of Frame.

ESB Enterprise Service Bus.

FTP File Transfer Protocol.

GSM Global System for Mobile Communications.

GSN Global Sensor Networks.

HTTP Hypertext Transfer Protocol.

ICT Information and Communications Technology.

IDE Integrated Development Environment.

IFS Interframe Space.

IoT Internet of Things.

IPsec Internet Protocol Security.

IPv6 Internet Protocol version 6.

ISG Intermediate Sensor Gateway.

ISO International Organization for Standardization.

JAAS Java Authentication and Authorization Service.

JBI Java Business Integration.

JDBC Java Database Connectivity.

JMS Java Message Service.

JSON JavaScript Object Notation.

LSM Linked Stream Middleware.

M2M Machine-to-Machine.

MDA Model Driven Architecture.

MDD Model Driven Development.

MDE Model Driven Engineering.

NMR Normalized Message Router.

OBD On-Board Diagnostics.

ODE Orchestration Director Engine.

OSGi Open Services Gateway initiative.

PCM Powertrain Control Module.

PID Parameter ID.

PIM Platform Independent Model.

PSM Platform Specific Model.

RDF Resource Description Format.

REST Representational State Transfer.

RTR Remote Transmission Request.

SaaS Software as a service.

SAE Society of Automotive Engineers.

SMTP Simple Mail Transfer Protocol.

SOA Service Oriented Architecture.

SOAP Simple Object Access Protocol.

SOF Start Of Frame.

SSN Semantic Sensor Networks.

TCM Transmission Control Module.

TCP Transmission Control Protocol.

UDG Universal Device Gateway.

UDP User Datagram Protocol.

WG Wireless Gateway.

WSS Web Services Security.

XML Extensible Markup Language.

XMPP Extensible Messaging and Presence Protocol.

XSLT Extensible Stylesheet Language Transformations.

CHAPTER 1

INTRODUCTION

After the World Wide Web and universal mobile accessibility, the Internet of Things (IoT) might represent the biggest technological revolution of our lifetime. The IoT concept refers to the incorporation of several heterogeneous end systems, making them able to communicate among themselves and with users. In this sense, becoming an integral part of the Internet. These interoperable systems find indeed applications in many different domains, such as home and industrial automation, medical aids, intelligent energy management, automotive, traffic management, and many others (Singh et al. 2014).

1.1 Motivation Scenario: OBD - New Emerging Applications

For instance, let us consider the road vehicles used nowadays. Motor vehicles are an essential resource in developed countries; however, car accidents are among the top ten causes of death according to the *World Health Organization*[1]. A monitoring system which identifies and reports the status of various vehicle's subsystems could help prevent road traffic accidents and reduce the response time of emergency services.

As an example, the environment where a car is being monitored and sends all its data to a cloud service. This scenario can be implemented using a microcontroller board connected to the On-Board Diagnostics (OBD), which gives access to live data streams of the vehicle subsystems. This data is processed on a microcontroller and sent, when requested through, for example, an integrated GSM module. A cloud service receives this data and provides it to any smartphone or computer, delivering in real time the diagnosis of the car. The main advantage of this implementation is the capacity to access the car information, with live

[1] Available at `http://www.who.int/mediacentre/factsheets/fs310/en/` - last update May 2014.

stream data, which can prevent accidents and transmit a distress signal in case of theft. Another important advantage is to reduce the response time of rescue services. In other words, if the microcontroller detects an accident through the OBD data (for example, if the airbag is deployed), an emergency response system will be triggered. This system will send the vehicle localization via a web service to one operator or computer system that will start the emergency procedure. Additionally, an emergency trigger can send a data stream to the police and to roadside assistance, enforcing a rescue procedure.

Another possible application is the prevention and identification of vehicle breakdown. The system can alert the user, through his/her smartphone, and suggest some simple steps to easily solve the problem or take some prevention measures. If the system considers the solution to the car's breakdown to be complicated for a common person, it can automatically request roadside assistance providing localization and vehicle diagnostic. The central idea behind this application is to help drivers prevent and solve simple problems related to their vehicle, which can become much worse when ignored. From a financial perspective, it can help them to save money on auto repairs by skipping the mechanic altogether.

A different but interesting scenario would be a social study about driver behavior. It would be helpful to have a system which shows how drivers behave per city and even per country. Studying anonymous data from the OBD of each vehicle could be valuable to identify car accidents causes and heavy traffic. Social studies could identify behavior patterns and contribute to efficiently adapt the road traffic regulations. For instance, if there is a considerable number of accidents on a specific street, the OBD data of each vehicle involved could help to identify the causes. Which lead in the adoption of new measures to decrease accidents at that place. In this specific example, the system could identify a need for traffic lights, roundabouts or speed bumps.

Another interesting solution could be to allow car brands to access the OBD data in the cloud services. A statistical study of the OBD data in each vehicle could help car brands to solve massive mechanical failures and improve their own automobiles according to customer needs. Note that all this OBD data should be anonymous and analyzed statistically to ensure client privacy.

The exponential use of smartphones can bring a new perspective for OBD solutions. As mentioned above, it can alert users in the case of theft as well as automatically alert police and immobilize the vehicle remotely. In terms of quality of life and productivity, when a vehicle enters a parking lot it could be notified with the localization of the nearest free parking space.
These scenarios justify the increased interest in the OBD and their possible applications. The motivation scenario described above is merely representative of what this master's

thesis can help propel and not the work developed itself.

1.2 Problem Characterization

The problem characterization is the starting point of all research work. Only with this characterization it will be possible to have a base start for the background research, in order to acknowledge all work already developed.

Nowadays solutions does not full fills the needs of the development community. An example of it are the Multiprotocol ECU Simulator, Freematics OBD-II Emulator and ECUsim 5100 Professional which will be analyze in chapter 6.

The general obstacle to overcome in this systems, is to accomplish a technological solution which allows intercommunication between several data formats. This sets some key focal points, which needs to be scrutinized.

The first and most obvious one, lies on the future usefulness. It needs to be capable to evolve and be flexible enough to integrate a future solution (a new different protocol), which cannot yet be developed.

Another goal of this characterization relays on system simplicity for adaptive services. The solution development needs to presupposes accessible integration for external services with minimal transformations. It also demands minimal knowledge to understand how to perform the referred integrations.

Finally, the system needs to be light in terms of computational power. The approach has to be executable in devices with limited resources, in order to be used without high needs of Central Processing Unit (CPU), storage and energy.

1.3 Work Methodology

The work methodology applied in this master's thesis is based upon the basic principles of the Scientific Method (Carey 2011), (Schafersman 1997). The methodology used is illustrated in Figure 1.1, and is composed of the following seven steps:

1. Problem Characterization
2. Background Research

3. Formulate Hypothesis
4. Setup an Experiment
5. Test Hypothesis through an Experimentation
6. Hypothesis Validation
7. Publish Results

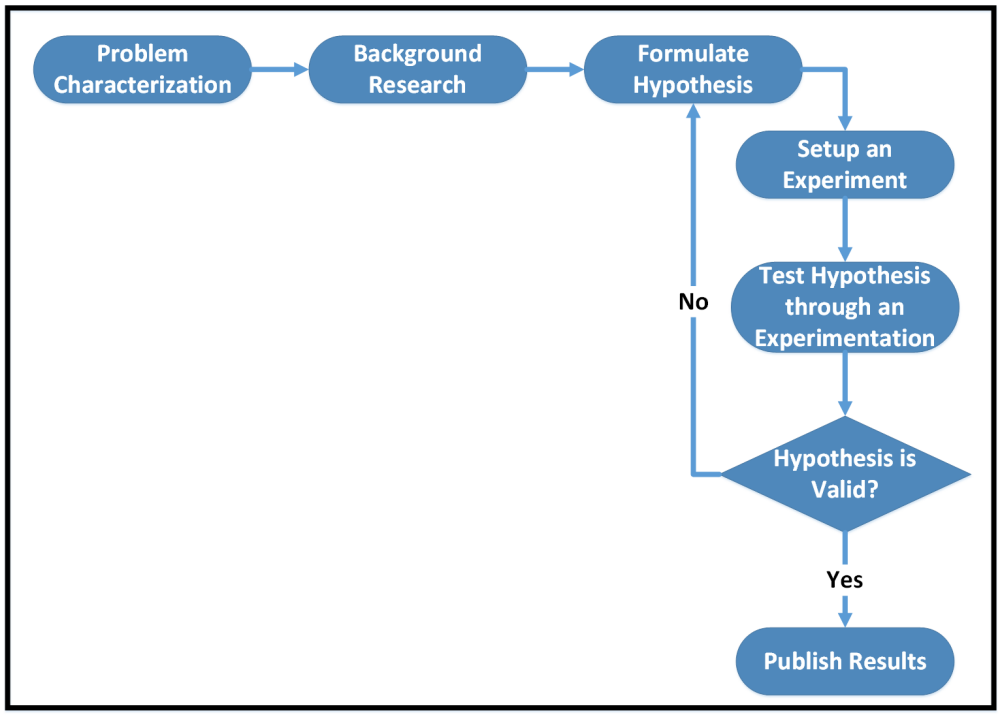

Figure 1.1: Overview of the Work Methodology used in this thesis.

1. **Problem Characterization**

   The focal point in this step is the problem definition and its characteristics, which lead to a research question that will be the basis of the research work. It is already defined in 1.2.

2. **Background Research**

   This step features the study of prior work related to information about the problem formulated in the first step. This study is based on a characterization, analysis and identification of a question. This approach provides the necessary tools for a starting point of the work itself. Awareness of what has already been accomplished facilitates the identification and justification of the progress that this work aims to accomplish.

3. **Formulate Hypothesis**

   The hypothesis is focused on the background research and specifies a theoretical approach to solve the defined problem, which results in the setup of an experiment. In this master's dissertation the hypothesis consists in an architecture which supports several tests for OBD systems.

4. **Setup an Experiment**

   In this step the proof-of-concept is implemented to validate the hypothesis. This phase includes detailed planning and execution of the experimental approach to the technological architecture.

5. **Test Hypothesis through an Experimentation**

   This step defines a series of tests to which the implementation will be submitted, according to the characteristics of the problem and formulated hypothesis. The tests must be executed in a controlled environment to ensure their quality. The data collected in each test must be interpreted and analysed in order to validate the previous hypothesis. If possible, qualitative and quantitative data analysis should be applied to the results.

6. **Hypothesis Validation**

   After an extensive analysis of the results in the previous step, and taking into account the characteristics identified in the problem, the hypothesis must be validated. Results can enlarge hypothesis's value or jeopardize all assumptions made in the beginning of the research. In case of failure, the original approach must be improve, and step 3 must be restructured.

7. **Publish Results**

   This final step consists in publications of the successful results provided by the research work, as well as respective recommendations for further research. In order to make a contribution to the scientific community, the publication must be suitable for the type of research conducted. Scientific papers should be written to present intermediate results and a dissertation must be focused on the hypothesis. This master's dissertation is the publication invoked in this step.

1.4 Dissertation Outline

This dissertation is divided into six chapters where the first is this introduction. This chapter presents a motivation scenario based on the new emerging applications with OBD systems and a description of the approach used to do this work, which was based upon the principles of the Scientific Method. The key points of the problem characterization are

also described, serving as a based start for the State-of-the-Art.

In addition to this chapter, the remainder of this dissertation is divided into five other chapters, each one with the following characteristics:

State-of-the-Art: Contains an overview about the previous related work concerning this thesis field. It is based on existing technologies, documents and studies which can help to better understand an subsequently problem, leading to a main Research Question.

Multiprotocol Architecture for OBD Emulator: The third chapter, relays on an exhaustive description of the architecture from its basic concept to its detailed logical specifications.

Experimental Setup: The fourth chapter, describes the supporting technologies that were used in order to perform the implementation, clarifying the OBD concept and defining all its requirements.

Implementation and Validation: This chapter contemplates a detailed view of the implemented solution which was based on the proposed architecture. The experimental results are presented and discussed.

Conclusions and Future Work: This document ends with the highlights of final thoughts and conclusions, describing the general impact about the developed work. Finally, it points the direction for future research works regarding the obtained results.

CHAPTER 2

STATE-OF-THE-ART

In order to start with a solid background, an extensive research was made to identify the current similar technologies and approaches related to this master's thesis. Various solutions were identified but only five gets main focus with the finality to prepare a research question.

2.1 Review

This investigation keeps its major focus on integration and management of heterogeneous devices in an interoperable environment. Distributed and independent system with a heterogeneous integration and high scalability are eliminator criterion for the searched architectures.

The architectures studied are presented in alphabetic order:

- An IoT Gateway Centric Architecture to Provide Novel M2M Services: This research proposes an innovative IoT architecture that allows real time interaction between mobile clients and smart sensors/actuators via a wireless gateway (Datta et al. 2014).

- Apache ServiceMix: It is an open source Enterprise Service Bus (ESB) that combines modularity with the performance of a Service Oriented Architecture (SOA). This architecture integrates a service bus that allows to decouple all applications together and reduce dependencies. Connectors enables the exchange of information using different communication protocols and messages are used to exchange data between applications (Apache S. F. 2016).

- OpenIoT - Open Source cloud solution for the Internet of Things: It is an open source middleware which communicates with sensor clouds in an independently and interoperable way. This project handles and manage cloud environments composed by sensors, actuators and smart devices, offering efficient utility-based IoT services. Its provide cloud-based and utility-based sensing services, via an adaptive middleware framework (Soldatos et al. 2015).
- Universal Device Gateway: Develops and provides customized Information and Communications Technology (ICT) solutions and services enabling among others the integration and management of a highly scalable and heterogeneous IoT. It is also a cross-domain integrator in compressing, energy efficiency, comfort, security and M2M automation, leading the way to a highly scalable network, from home, office building, up to smart cities and beyond (UDG Alliance 2016).

2.2 Individual Review

The next sections will present a detailed analysis about the previous studies.

2.2.1 An IoT Gateway Centric Architecture to Provide Novel M2M Services

This project introduces an IoT gateway centric architecture in order to provide innovative and improved, Machine-to-Machine (M2M) services. The backbone of the proposed IoT architecture, shown in Figure 2.1, is performed by a Wireless Gateway (WG). The architecture consists in three layers: sensing layers containing M2M devices and endpoints; a gateway API layer; and an application layer. The gateway contains a central database to store resource configurations of devices, endpoints and sensor metadata. Such implementation of the mobile client and wireless gateway is highly applicable in personal health monitoring. The functionalities and APIs are broadly associated with two interfaces, north and south. The north interface communicates with mobiles clients and assists in discovery phase. The south interface manages and interacts with M2M devices and stores their configuration in a local database. The novel services offered by this architecture are:

- Dynamic discovery of M2M devices and endpoints by clients.
- Managing connection with non-smart things connected over modbus.
- Associate metadata to sensor and actuator measurements using Sensor Markup Language (SenML) representation.
- Extending the current capabilities of SenML to support actuator control from mobile clients.

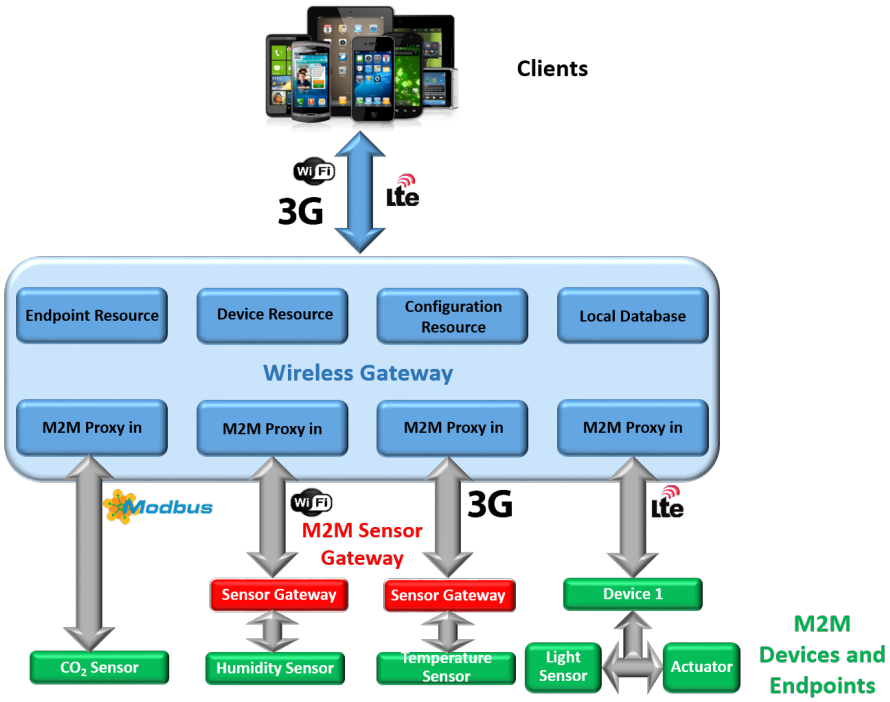

Figure 2.1: The proposed IoT Gateway Centric Architecture, based on (Datta et al. 2014).

Analysis

This approach proposes a solution, via wireless gateway, that allows device and service discovery; a standard protocol for endpoints integration; and mobile client control. The M2M devices and endpoints must register themselves in the gateway in order to be available to the mobile clients. This is made by a discovery phase which establish a connection to WG, allowing add or remove of M2M devices and endpoints, at real time. The API for the discovery phase is implemented in the gateway which is a novel solution provided by the architecture.

After the discovery phase, users can select, from a sensors list, the desired measurements to be received which are represented using SenML (Jennings et al. 2013). In order to contemplate the limited capabilities of M2M smart devices, SenML is a perfect solution because of is lightweight, which provides an easily encode sensor measurements into different media types. Other advantage of SenML is its implementation by using JavaScript Object Notation (JSON) (ECMA-404 2013), enabling a server to efficiently parse huge numbers of sensor metadata at a very short time.

Another important feature in this architecture is the capability to manage non-smart things communication. This devices are incapable of generating SenML metadata. With the purpose of solving this problem, such devices are connected using a serial communication protocol Modbus (Pefhany 2000) to the WG, which converts the raw measurements

into SenML metadata. This approach also addresses actuator control from mobile clients. For that, SenML had to extend its capabilities in favor of address actuator control problems.

The APIs are based on the Representational State Transfer (REST), also known as RESTful web services (Fielding and Taylor 2000). The mobile clients are equipped with an application that receives measurements from sensors and can control actuators.

The M2M device receives requests from clients through Hypertext Transfer Protocol (HTTP). Afterwards there is a protocol that translates the HTTP data into M2M device specific command, make it understandable for the endpoints. In case of non-smart devices, the instructions received are converted into machine executable format by suitable mapping to a local database entry. The M2M proxy-in invokes an API which creates the desired SenML compliant metadata.This process can be assist by the Intermediate Sensor Gateway (ISG), helping in the creation of metadata and forwarding it to WG.

2.2.2 Apache ServiceMix

Apache ServiceMix is an open source implementation of an ESB in Java programming language. The main features of this architecture, shown in Figure 2.2, is to exchange messages from one component to another, as well as, the ability to route messages. It differentiates from other systems mainly because it reuses the communication channels, optimizing, in this way, the classical principles of connectivity and data exchange. ServiceMix fully supports the Open Services Gateway initiative (OSGi) (OSGi Alliance 2016) framework, which add an important feature to the architecture, modularity. That means that it can handle classloading and application lifecycle differently between components.

This architecture is present in Tecsisa, a company specialized in the development of solutions for the energy sector focusing on Service Oriented Architecture and cloud computing technologies (Tecsisa 2016). Not less important, GASwerk is an open source project which provides production ready solutions based on others open source components and it also uses ServiceMix (GASwerk 2016).

In order to standardize the ESB, Apache ServiceMix is builded on the Java Business Integration (JBI), which is defined by the JBI 1.0 Specification as *"an architecture that allows the construction of integration systems from plug-in components, that interoperate through the method of mediated message exchange."* (Ten-Hove and Walker 2005).

The imperative features which compose the ServiceMix are:

- ActiveMQ, to provide remoting, clustering, reliability and distributed failover.

- JBI specification including: Normalized Message Router (NMR) which is a bus that shuttles messages between the endpoints deployed on the ESB and JBI Management MBeans that expose management functions.
- Supports many communication protocols: File Transfer Protocol (FTP), HTTP, Java Message Service (JMS), Simple Mail Transfer Protocol (SMTP), Simple Object Access Protocol (SOAP), Transmission Control Protocol (TCP), Extensible Messaging and Presence Protocol (XMPP).
- Supports many engines: Apache Camel, Apache CXF, Apache Orchestration Director Engine (ODE), Scripting, Saxon XQuery, Extensible Stylesheet Language Transformations (XSLT).
- Support for Security: Java Authentication and Authorization Service (JAAS), Web Services Security (WSS).
- Web Container/App Server Integration: Geronimo, JBoss, Jetty, Tomcat, Weblogic, Websphere.

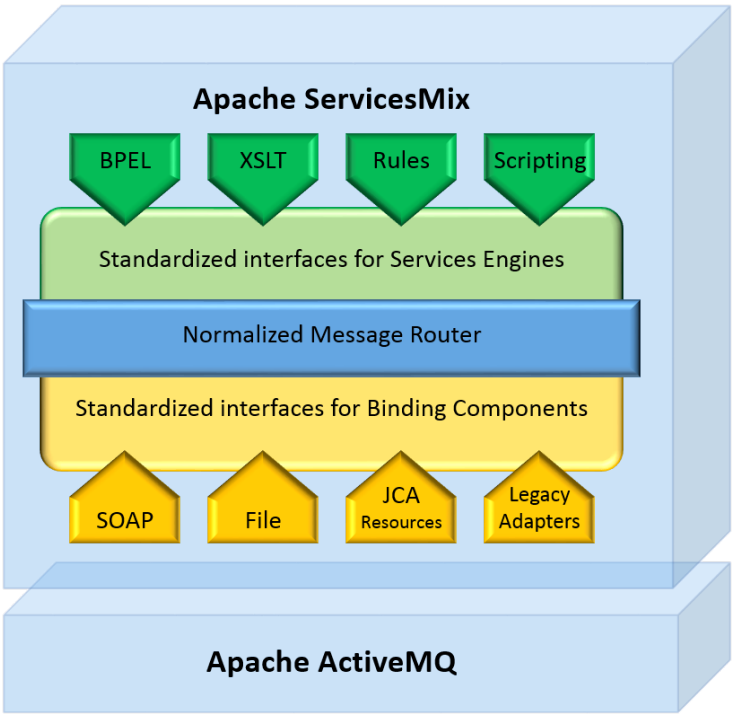

Figure 2.2: Apache ServiceMix Architecture, based on (Snyder 2008).

Analysis

For a further analysis of Apache ServiceMix, it is imperative to clarify the definition of ESB, which is defined by the Gartner Group as: *"a new architecture that exploits Web services, messaging middleware, intelligent routing, and transformation. ESBs act as a lightweight,*

ubiquitous integration backbone through which software services and application components flow." (Thomas 2007).

This architecture implements ESB through JBI, which describes all components with Java terminology and interfaces. This enables the integration of services in an independent way, allowing the plug and play concept. The JBI standard allows the manage of data translation and routing.

The messages exchanged in this architecture are embedded with the interoperability concept in order to connect external systems to the bus. This requires a standard translation of messages, called Normalized Message, which contains the actual data, along with various properties and destination endpoint. Normalized Message, typically contains Extensible Markup Language (XML) data, but is not restricted to it, being also possible to attach binary information to the message. The NMR has the responsibility to transport the message to its final destination. When the Normalized Message leaves the NMR, it's converted to what is understood by the target system.

2.2.3 OpenIoT - Open Source cloud solution for the Internet of Things

OpenIoT is similar to an extension of cloud computing implementations, which is focused on the Internet of Things based resources and capabilities. OpenIoT includes and interrelates three important IoT technological areas: Middleware for sensors and sensor networks; Ontologies, semantic models and annotations for representing internet-connected objects; and Cloud/Utility computing. This architecture, illustrated in Figure 2.3, is constituted by seven main elements that belong to three different logical planes:

1. **Utility/Application Plane**

    - Request Definition: Enables on-the-fly specification of service requests to the OpenIoT platform by providing a Web 2.0 interface. It comprises a set of services for specifying and formulating requests, while also submitting them to the Scheduler.
    - Configuration and Monitoring: It enables visual management and configurations of functionalities over sensors and services that are deployed within the OpenIoT platform.
    - Request Presentation: It is in charge of visualization for service outputs. In order to visualize these services, it communicates directly with the Service Delivery & Utility Manager to retrieve the relevant data.

2. **Virtualized Plane**

   - Scheduler: It processes all requests for services from the Request Definition and ensures its proper access to the resources required. This component undertakes the following tasks: it discovers sensors and associated data streams that can contribute to a given service; it also manages a service and activates the resources involved in its provision.
   - Cloud Data Storage: It enables the storage of data streams stemming from the sensor middleware, thereby acting as a cloud database. The cloud infrastructure stores also the metadata required for the operation of the OpenIoT platform.
   - Service Delivery & Utility Manager: On the one hand, it combines the data streams as indicated by service workflows within the OpenIoT system in order to deliver the requested service either to the Request Presentation or a third-party application. On the other hand, this component acts as a service metering facility, which keeps track of utility metrics for each individual service.

3. **Physical Plane**

   - Sensor Middleware: It acts as a hub between the OpenIoT platform and the physical world. The Sensor Middleware is deployed on the basis of one or more distributed instances, which may belong to different administrative entities. The prototype implementation of the OpenIoT platform uses the Global Sensor Networks (GSN) sensor middleware that has been extended and now called X-GSN.

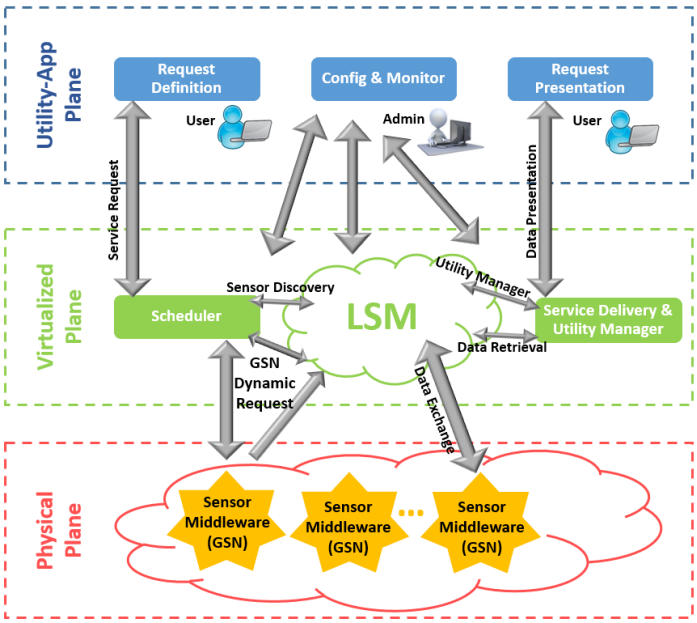

Figure 2.3: OpenIoT Architecture Overview, based on (Soldatos et al. 2015).

Analysis

This architecture provides an open source IoT platform enabling the interoperability of IoT services in the cloud. The central feature of OpenIoT is Semantic Sensor Networks (SSN) ontology (Compton et al. 2012), which provides a common standards-based model for representing physical and virtual sensors. The implementation of the ontology and its integration in the OpenIoT architecture are realized through the Linked Stream Middleware (LSM) (Le-Phuoc et al. 2011). LSM translates the data from virtual sensors into Linked Data stored using Resource Description Format (RDF).

In order to manages the registration, data acquisition, deployment of sensors and interconnected objects over LSM, this approach provides the X-GSN. The GSN is a flexible middleware layer which abstracts from the underlying, heterogeneous sensor network technologies. It supports a fast and simple deployment of new platforms, facilitates efficient distributed query processing and combination of sensor data. This provides support for sensor mobility, and enables the dynamic adaption of the system configuration during runtime with minimal effort. The core concept in X-GSN is the virtual sensor. This enables sensor heterogeneity between abstract and concrete entities (GSN Team 2014).

The data acquisition for each virtual sensor is provide by wrappers that collect data through serial port communication, User Datagram Protocol (UDP) connections, HTTP requests, Java Database Connectivity (JDBC) queries, and more. In order to receive data from various data sources, X-GSN implements wrappers and allows users to develop custom ones.

Other advantage of this architecture is its visual tools that enable the development and deployment of IoT applications with almost zero programming.

2.2.4 Universal Device Gateway

Universal Device Gateway (UDG) is a open source, highly scalable, portable, multiprotocol, control and monitoring system for the IoT. It communicates with all sorts of devices, protocols and standards, enabling multiprotocol and cross-domain interoperability. The integration of several devices and protocols generates a flexible interoperable system.

This approach simplifies the harmonization of heterogeneous system with Machine-to-Machine communication. It brings flexibility and easiest evolution of existing deployments by integrating devices through unique Internet Protocol version 6 (IPv6) addresses. Its benefits from high skill capability with approximately 3.4×10^{38} public addresses. That is more than enough to provide each smart device deployed with an unique address.

The UDG architecture, shown in Figure 2.4, enables an integration with the global internet and cloud, including Software as a service (SaaS), web application and smart things information Services. Other distinctive characteristic of UDG is its portability and heterogeneity which can be easily deployed in various hardware, in order to distribute intelligence across the whole network of smart things. This results in the absence of bottlenecks and an unique point of failure.

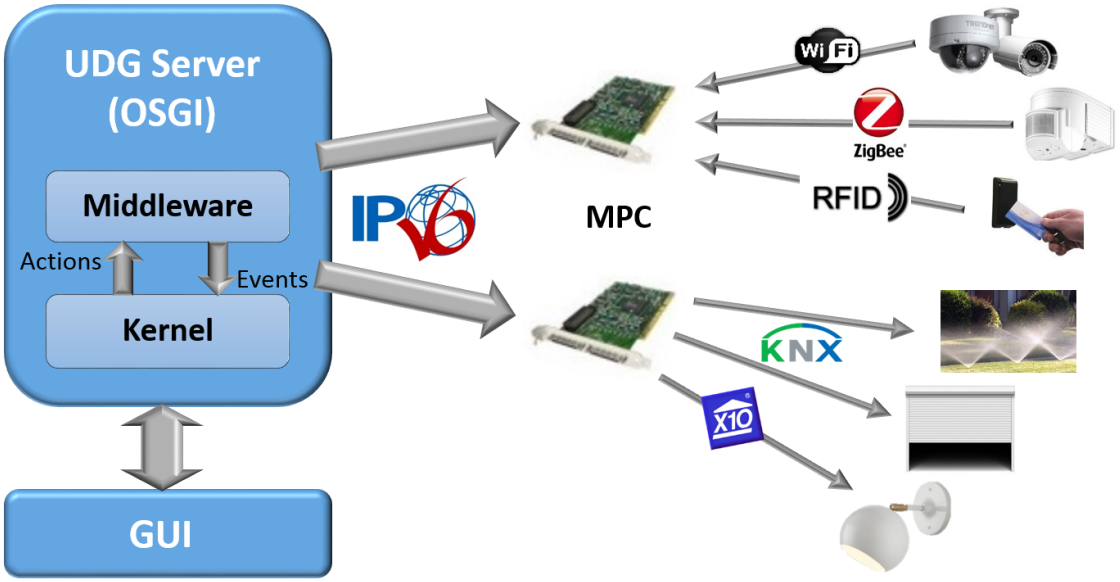

Figure 2.4: Universal Device Gateway architecture, based on (Bocchi et al. 2012).

Analysis

One of the most differentiating features of this approach is its IPv6 based technology. This enables high scalability and powerful mechanisms such as auto-configuration, multicast, anycast and mobility. On the safety level, Internet Protocol Security (IPsec), provides strong security and authentication mechanisms.

This architecture is compliant with IPv6 and M2M, providing interoperability with over 40 communication protocols. In order to allows heterogeneity, UDG permits an unique IPv6 and M2M proxy, enabling bi-directional interactions between non-IP systems and IPv6/M2M standards. Each devices gets its own and unique IPv6/M2M address.

Another important characteristic of this approach is its portability, enabling various hardware, from Raspberry-Pi to routers and servers. This results in the absence of bottlenecks and unique point of failure.

The cloud-based platform to monitor, control and manage sensors/actuators, coupled with on-line services, it is also an innovative feature which allows: secured remote access,

remote monitoring, alerts management, third parties SaaS and web services integration.

From the architectures studies the one which concretely grants a future scalable interoperability between systems it is the UDG because of its IPv6 oriented service based. However the IPv6 adoption among Google users is 10% at the moment that this thesis was written (IPv6 Google -). Therefore, it is a solution to take in high count for the future but has some limitations at the moment.

2.2.5 Synthesis

A first look at all four researches it is evident that Wireless Gateway (WG) and Universal Device Gateway (UDG) approaches focuses more on the Machine to Machine (M2M) devices. On the other hand, ServiceMix and OpenIoT are more comprehensive, being even able to manage virtual and pyhsical sensors.

The discovery device phase problem is one of the most discussed in all four architectures presents in this chapter. Excluding UDG, which seems to work with servers based on OSGi, all others three systems has its own discovery device system. In this specific case the WG architecture brings a novel discovery system implemented on the gateway, allowing a dynamical discover at real time. The OpenIoT approach, also stands out with its Scheduler system where X-GNS records all devices.

Another recurrent problem covered in this studies is the data translation. All four approaches deals with this problem on a proper way. However two of them differentiate from the others: ServiceMix with Normalized Message, which converts data to XML, and OpenIoT with X-GNS which introduces the wrappers concept.

Multiprotocol communication is addressed in all architectures, except WG which seems to only use HTTP. ServiceMix, introduces its NMR and OpenIoT with IPv6 protocol enables an heterogeneous communication systems.

Besides WG approach, which seems to relegate that to future work, all the others architectures have they security systems based on authentication and time expiration session. The API role on OpenIoT offers a different approach being composed by visual tools with almost zero programming. RESTful web services and SaaS are respectively addressed in WG and UDG architecture as API services.

Concerning modern architectures and flexibility, WG and OpenIoT are one step ahead of UDG and ServiceMix. On the other hand, WG, UDG and OpenIoT offers features suitable to be executed in devices with limited resources. The architectures which use a model-based development guide are OpenIoT and ServiceMix.

Each research study in this chapter has some special characteristic which differentiate them from the other. For the WG is the extended SenML allowing control actuator and in ServiceMix the central point is the reuse concept through ESB. In the case of OpenIoT the virtual sensors are a major value and for the UDG is its IPv6 based architecture. The most relevant conclusions of this analysis are summarised in Table 2.1.

Table 2.1: Overview of the State of the Art Research

	Main Feature	Modern and flexible architecture	Run in devices with limited resources	Model-based	Main technical approaches
WG	HTTP requests and Modbus	Yes	Yes	No	JSON
ServiceMix	Reuse Concept (ESB)	No	No	Yes	MDE
OpenIoT	Allows Virtual Sensors	Yes	Yes	No	Wrappers
UDG	IPv6 based	No	Yes	No	Software as a Service

2.3 Main technical approaches

In this section the main technical approaches will be scrutinized in detail.

2.3.1 JSON

The JavaScript Object Notation (JSON) is an open-standard text format that facilitates structured data interchange between all programming languages and is the most common data format used for asynchronous browser/server communication. It derives from JavaScript code to generate and parse the JSON-format data (Bray 2014).

JSON is agnostic about numbers, what can make interchange between different programming languages difficult. It offers only the representation of numbers that humans use, like a sequence of digits. Due to the fact that all programming languages know how to make sense of digit sequences, even if they disagree on internal representations, interchange can be guaranteed.

The JSON text is a sequence of Unicode code points which provides support for ordered lists of values. Because objects and arrays can nest, trees and other complex data structures can be represented. By accepting JSON's simple convention, complex data structures can be easily interchanged between incompatible programming languages.

This open-standard text format does not support cyclic graphs and is not indicated for applications requiring binary data.

It is expected that other standards will refer to this one, strictly adhering to the JSON text format, while imposing restrictions on various encoding details. Such standards may require specific behaviours since JSON itself specifies no behaviour (ECMA-404 2013).

2.3.2 Wrappers

The GSN can receive data from various data sources and this is done using wrappers, shown in Figure 2.5. They are used to encapsulate the data received from the data source into the standard GSN data model. This process is called StreamElement, which is an object representing a row of a SQL table.

Each wrapper is a Java class that extends the AbstractWrapper parent class. Usually a wrapper initializes a specialized third-party library in its constructor and also provides a method which is called each time the library receives data from the monitored device. This method will extract the interesting data, optionally parse it, and create one or more StreamElements with one or more columns.

The received data has been mapped to a SQL data structure with fields that have a name and a type. GSN is then able to filter this using SQL syntax (GSN Team 2014).

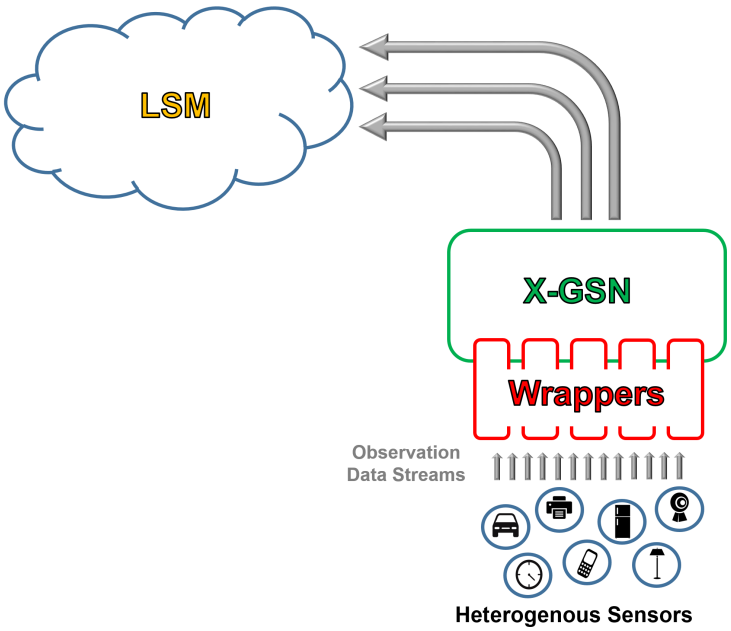

Figure 2.5: An example of a wrapper functionality, based on (GSN Team 2014).

2.3.3 Model Driven Engineering

The basic concept of Model Driven Engineering (MDE), also known as Model Driven Development (MDD) is to address everything as a model, which is characterized by the OMG with a: *"set of information about a system that is within scope, the integrity rules that apply to that system, and the meaning of terms used."* (OMG 2014). MDE is an open and integrative approach that embraces a set of associated concepts, body of knowledge, tools, required skills, and possibilities. It simplifies complex systems in several activities and tasks that comprise an information system life cycle. In this way it is possible to maximize compatibility between systems and promoting communication between developers, increasing productivity.

An enterprise is constituted by business processes, services, information, resources, etc. Each of this core features has an important role describing a model. A single model can be described as a set of expressions composed by information about a system. A model is exposed in a model notation or language and can be visual or textual. A language description contains an abstract syntax, one or more concrete syntax descriptions, mappings between abstract and concrete syntax, and a description of the semantics. The abstract syntax of a language is often defined using a metamodel. In this case a metamodel can be

defined as a modeling language (Schmidt 2006).

MDE supports models at different levels of abstraction, from high-level business models, focusing on goals, to complex scenario models, for business execution. Subsequently, quality is an important aspect of MDE, which can be guaranteed with: model validation, model checking, and model-based testing (OMG 2014).

Normaly, MDE is often confused with Model Driven Architecture (MDA). However according to Favre (Favre 2004) MDA can be seen as a specific incarnation of the MDE approach. Comparing MDE and MDA it can be concluded that the last one is more focuses on technical software heterogeneity and how to specify a software in independent way within a platform (Kent 2002).

The MDE approach can be define in three different abstractions levels of information:

- Computation Independent Model (CIM), is actually called, by OMG, a Business or Domain model and it specifies the requirements for the system and environment where it will operate. All of that defines the function of a system without showing constructional details and plays an important role doing a bridge between business, design and IT experts. In order to provide the business needs, CIM, only describes business concepts whereas a PIM may define a high-level systems architecture.

- Platform Independent Model (PIM), is the formal specification of a system from the platform independent viewpoint. In order to be suitable for several different platforms of similar type, it specifies a system without implementation details.

- Platform Specific Model (PSM), can be defined as a more detailed version of PIM. This platform associates the specifications in PIM with technical details and implementations that are available in a system. That influences how such system uses a particular type of platform, which includes middleware, operating systems and programming languages.

The input and output of each model must be defined as metamodels and as well classified according to the metamodelling level they belong, which brings optimized interoperability benefits (Kent 2002; OMG 2014).

Another important concept of MDA it is executable model transformations, which are use to increase automation in program development. The high levels models are constantly transformed into lower levels until it is possible to execute any code generation or model interpretation. There are two types of transformations over in MDA transformation definition: Horizontal Transformations and Vertical Transformations.

The first one, does not affect the abstraction level leading to enable collaborative activities and interoperability benefits. However the language associated between systems needs to have its specifications well define. On the other hand, Vertical Transformations necessarily implies a change on the abstraction level of the resulting model, affecting its specifications. Vertical transformation tools provide a mechanism to perform model annotations and means to customize the transformation rules according to the user need. They also provide a pre-defined PSMs which influences, along with the code generator, the amount of generated code (OMG 2014).

2.3.4 SaaS - Software as a Service

In the software as a service model, the application, or service, is deployed from a centralized data center across a network providing access and use on a recurring fee basis. Users "rent," "subscribe to," “are assigned”, or "are granted access to" the applications from a central provider. Business models vary according to the level to which the software is streamlined, to lower price and increase efficiency, or value-added through customization to further improve digitized business processes.

The core value of software as a service is providing access to, and management of, a commercially available application. The potential benefits of the model are significant for both the vendor and the customer.

2.4 Remarks

Multiprotocol is one of the features more approached in this study and the majority of systems fulfill the necessary specifications of IoT architectures. WG not specifies concretely the protocols communication allowed except for non smart devices, where it uses Modbus.

All approaches uses Web Services or OSGI, which grants that all kinds of users can access the network, by different operating systems and using different devices. Nonetheless WG seems to be still in a early phase of development and do not fully guarantee access rights for multiple users and security & privacy.

In all researches has a mature development for the discovery device feature but OpenIoT platform only allows the creation of sensor types called gsn. This is a major issue and limitation as sensors like soil moisture, weather, etc. cannot be discovered (Soldatos et al. 2015).

All the systems referred earlier are not able to fully solve the characteristics of a multiprotocol OBD emulator architecture. Different functionalities of each system were extracted

from all of them, in order to identify a problem, respective hypothesis and creating a unique solution capable of fulfilling all the requirements exposed.

2.5 Research Question

In order to create successful On-Board Diagnostics applications, it is inevitable ensure an exhaustive test system which improves user interaction among different environments. However, experiments with vehicles are not economically viable because of possible manipulation in car sensors and actuators. Errors are unavoidable during tests and manipulating sensitive aspects in vehicle could jeopardize the security of users.

In favour of overcoming this issue there are some commercial solutions which simulate OBD behavior to controllable car data inputs. Upon analyzing OBD simulators in the current market there is a common point between them: the price. In general, OBD simulators are expensive and do not offer all main protocols in their basic kit. For instance, the ECUsim 2000 OBD-II ECU Simulator, from OBD Solutions enterprise, offers many interesting characteristics but with only one communication protocol it costs approximately 178€(OBD Solutions -a).

On the other hand, non-commercial solutions where users can manipulate and customize every aspect of their applications are in the line of an unexplored market. For the author of this master's thesis and until the date that it was written there is no solid academic or independent work which allows the development of projects in a low-cost non-commercial OBD simulator. There a gap on the IoT community for a specific project which addresses this field with open source technology.

These facts leads to the following research question, which supports this master's thesis:

How a multiprotocol architecture can assist on the developement of new and interoperable OBD systems?

This problem presents a set of characteristics that needs to be addressed:

- Modernity and Flexibility: The architecture must be flexible and capable to evolve in order to accommodate more protocols that may appear in the future. In this sense, there is a need for modern, flexible and plug-and-play approaches.

- Model-based: The system needs to concern a model-based development and domain specific languages in order to be suitable for users with the referred knowledge and not the technological characteristics itself.

- Light format: There is a need for simple architectures with small time and spatial footprints that can be executed in small devices. The output services must come in a standardised frame which can grant interoperability in a light format allowing access to different kinds of devices.

CHAPTER 3

MULTIPROTOCOL ARCHITECTURE FOR OBD EMULATOR

This chapter proposes a possible solution to the Research Question introduced previously. The suggested architecture is exhaustively described from its basic concept to its logical specifications. In order to have a clearly overview of this approach it is important to clarify the difference between emulator and simulator. In the author perspective, emulation concept is linked to the duplication of inner workings in a system. On the other hand a simulation tries to duplicate the behaviour of a system. In other words, emulators replicate the original usage of a software or hardware and simulators are an environment of test models with the purpose of analysis and study.

3.1 Hypothesis

Based on the background observation summarized on chapter 2 and in order to satisfy the demands of the research question, it is believed that the following hypothesis could be the answer to the mentioned problematic.

If a Model-based approach is followed, other systems can import these formal descriptions, then new and interoperable OBD systems can be developed.

The next section describes the adopted approaches and expected outcomes of hypothesis implementation, testing and validation.

3.2 Concept

The idea behind OBD Emulator is the development and testing of OBD applications. This system provides external control in a standard OBD port which simulates the output signals of a vehicle's Electronic Control Unit (ECU). The major advantage for developers is to avoid real connection to an actual vehicle with the purpose of testing and validation of new solutions. This concept also allows common persons to access vital information in a vehicle without expert assistance.

The general idea is to design a method which enables communication between two different systems. In this particular case, one system symbolizes a car and other system represents a diagnostic tool which acquires car data. It is crucial to have an integration platform in order to guarantee interoperability for physical interfaces and data formats. The Figure 3.1 shows the concept designed for the OBD Emulator.

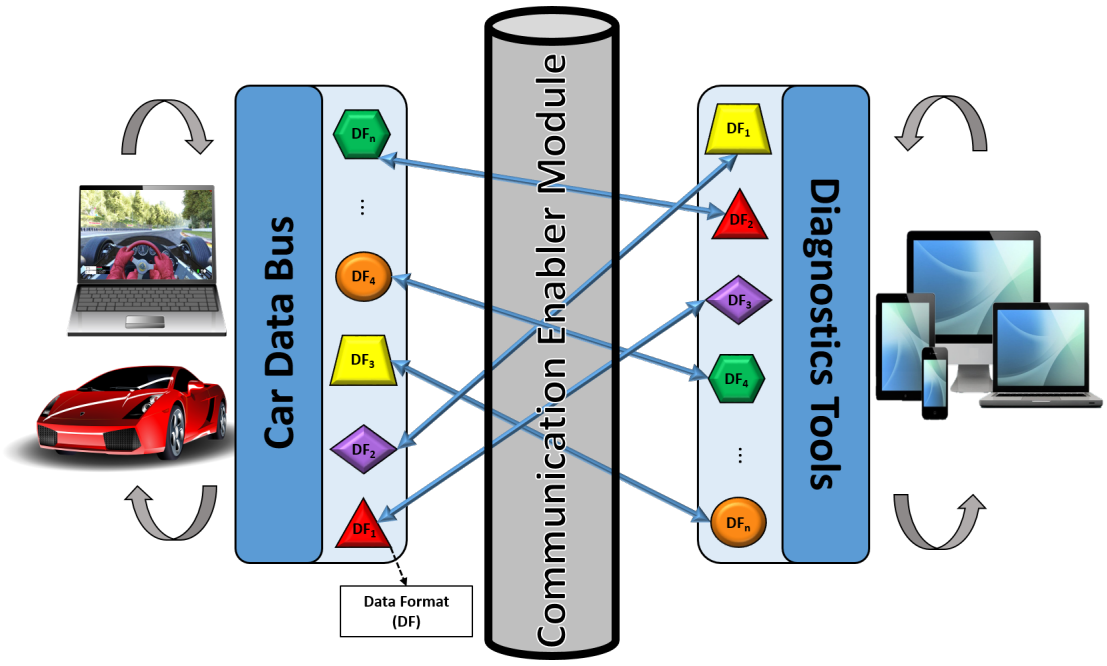

Figure 3.1: OBD Emulator Concept.

This concept relies in sending requests, trough a smart device, over the avaiable Diagnostics Tools. These mechanism translates the previous requests to a specific Data Format (DF). Afterwards, forwards it through the Communication Enabler Module (CEM) to other DF, on the car driving simulator side.

The CEM operates as a bridge between devices and car driving simulators. It is responsible for the interoperability among differents DFs and allows communication in an heterogeneous environment.

Requests/replies are received respectively on the Car Data Bus and Diagnostics Tools, through DFs. These two modules manages requests and respective replies, coding/decoding the data acquired from car driving simulators or devices (e.g: computers, smart phones), in the available DFs.

3.3 Architecture Overview

At this point it is possible to address the essential module of an OBD emulator architecture, which intends to test the presented hypothesis. It is reasonable to conclude that the above approach is centralized in CEM, therefore it will be scrutinized in next section. The architecture overview in CEM, uses three different logical modules as illustrated in Figure 3.2. A brief description of each individual modules is presented next.

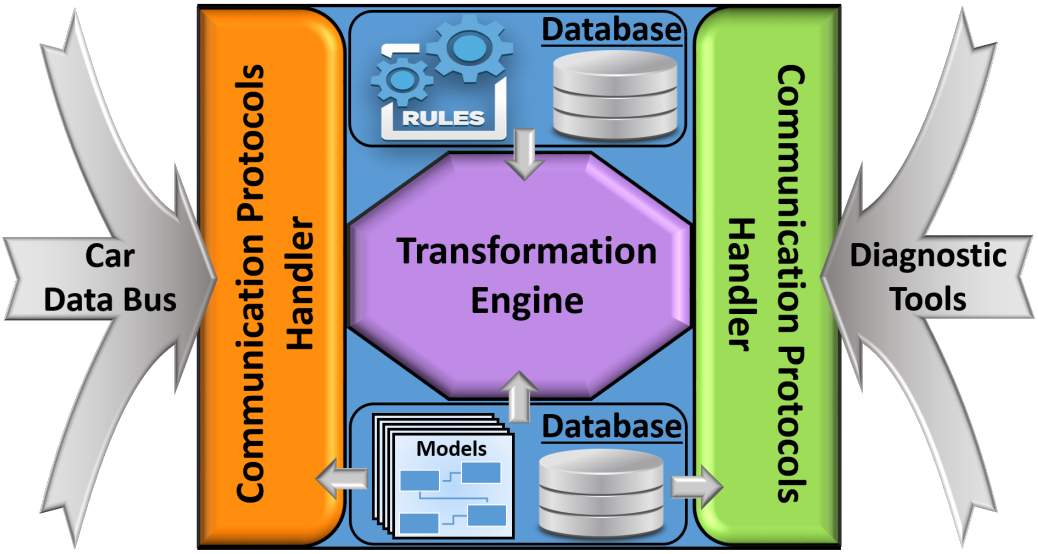

Figure 3.2: Communication Enabler Module.

Communication Protocols Handler (CPH), works as an interface for all communication modules, providing necessary tools for establish connection between incoming and out coming data and the Transformation Engine Module.

Transformation Engine is a central communication bridge in the architecture, receives requests/replies and forwards them from one to other CPH module. It transforms Data Formats according to actual requirements, specifications enabling interoperability between each others.

Database is a data collection composed by models and rules. It contains all DFs models used, as well as, rules definitions established for the proper architecture operation.

3.4 Logical Architecture Specification

In the next section is presented a brief description of functionalities and characteristics of each logical module.

3.4.1 Communication Protocols Handler

The logical module Communication Protocols Handler (CPH) is responsible for providing system connection with the outside systems data bus, using available data formats to communicate. Data formats can be added or removed independently, not compromising the system. The CPH is composed by three logical sub-modules: Hardware Layer, Car Data Bus Interface or Diagnostic Tools Bus Interface and Data Protocols.

The module composition, showed in Figure 3.3, has input information from the vehicle by a bus called Car Data Bus. There is also other CPH module with a similar function but with diagnostic requests data as input. In this way, it is assured bidirectional communication between these two modules.

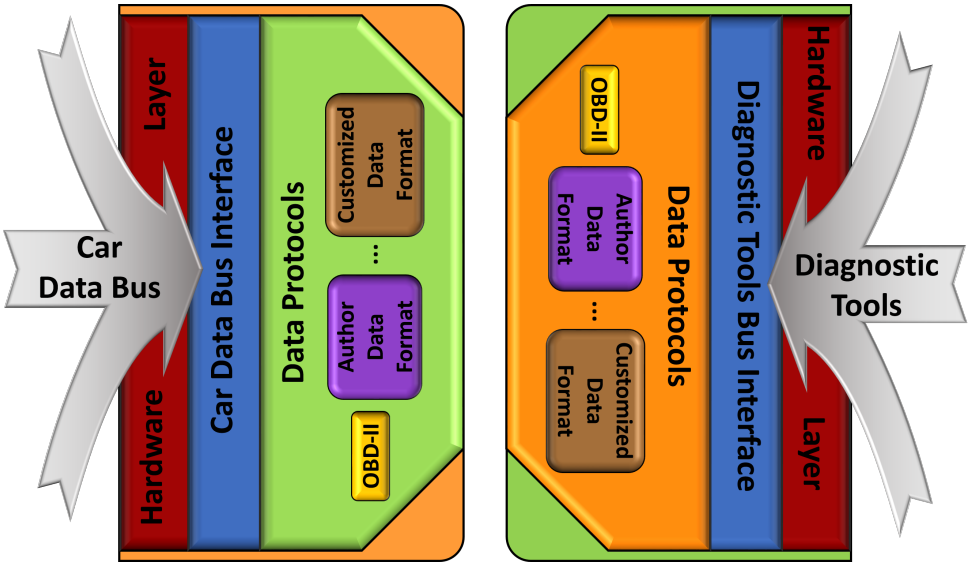

Figure 3.3: Communication Protocols Handler modules.

The Hardware Layer assures support to other sub-modules with all the physical features necessary for communication. It is constitute by communications protocols (e.g.: tcp/ip, CAN) and their respective physical interfaces (e.g.: ethernet, CAN interface).

The Car Data Bus Interface and Diagnostic Tools Bus Interface are a connection bridge between outside data and the Transformation Engine.

Finally, the sub-module Data Protocols has the ability to interpret data and arrange them according to the required model specifications (e.g.: OBD-II). This sub-module allows to identify incoming data formats, standard or new ones made by developers, which is an important feature of this architecture, since it is capable to retrieve new data models from the storage module and therefore identify new/different incoming Data Protocols. The author of this thesis customized his own data format, described in chapter 5.

3.4.2 Transformation Engine

The Transformation Engine enables communication between different data formats, transforming them in order to be understandable to the receiver DF.

This module receives data from the CPH, in one DF from its list, and transform it (accordingly to specified rules) to a target DF in order to be successfully received on the other CPH. The target CPH can contain or not the same source data format. Communication will always be assured by the Transformation Engine module, even between different data formats.

Modulation is the key feature in this transformation and it is supported by rules and models. The module composition is showed in Figure 3.4.

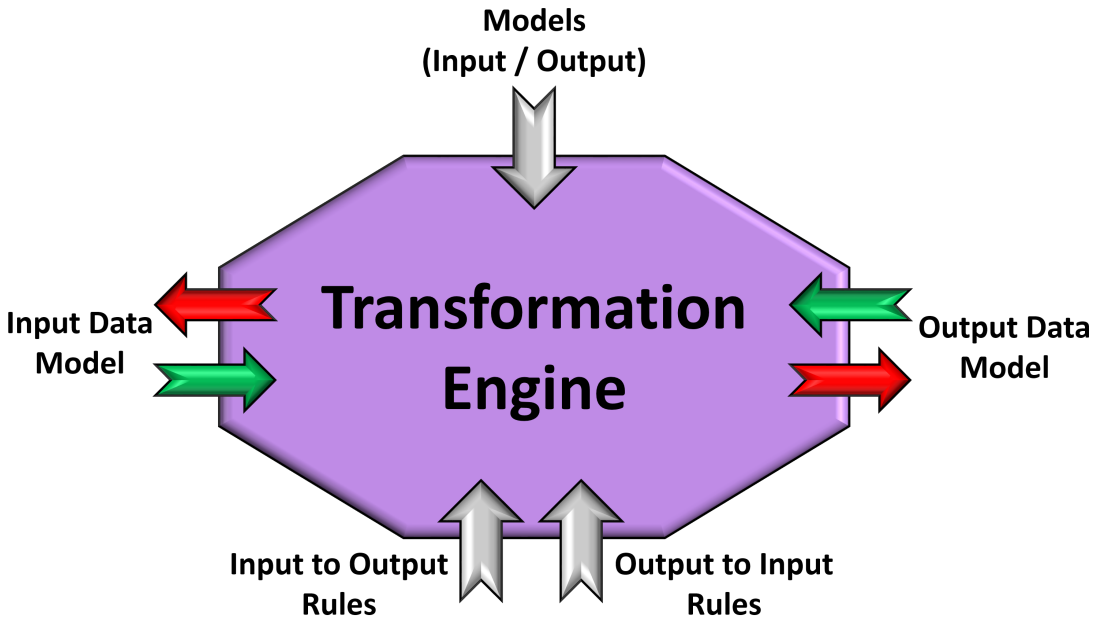

Figure 3.4: Transformation Engine module.

In a general overview this modulation consists on the use of meta-models mapping,

so data can be transform from one format to another. To achieve data conversion, two levels of abstraction must be defined to use the Model Driven Architecture (MDA). The Platform Independent Model (PIM) and Platform Specific Model (PSM). These meta-models concepts are already described in subsection 2.3.3 - Model Driven Engineering.

The explained concept is represented in the next Figure 3.5.

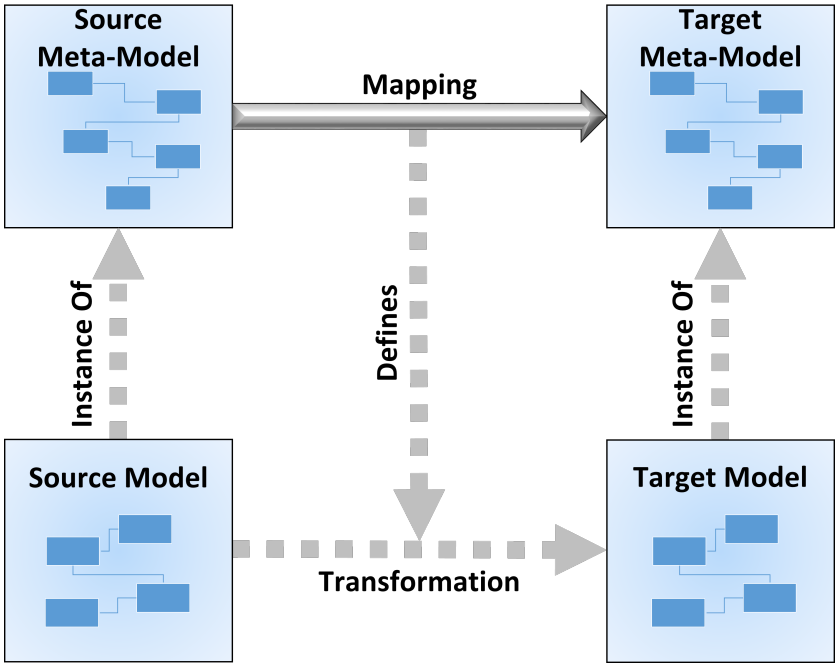

Figure 3.5: A transformation example between two Data Formats.

Succinctly, the mapping between a source meta-model and a destination meta-model generates rules. These rules are applied in the transformation of a source model in order to obtain a target model.

3.4.3 Database

The Database is a repository used to store models that describe data formats and also rules needed to transform a source data model to an target data model. Acting as a repository the DataBase also allows new models to be added, as well as, the corresponding rules necessary for transformation from one source model can be held for different target models.

3.5 Architecture Composition

All together the logical modules described in the previous section with their interfaces provide a new approach to represent the Multiprotocol OBD Emulator Architecture, as showed in Figure 3.6

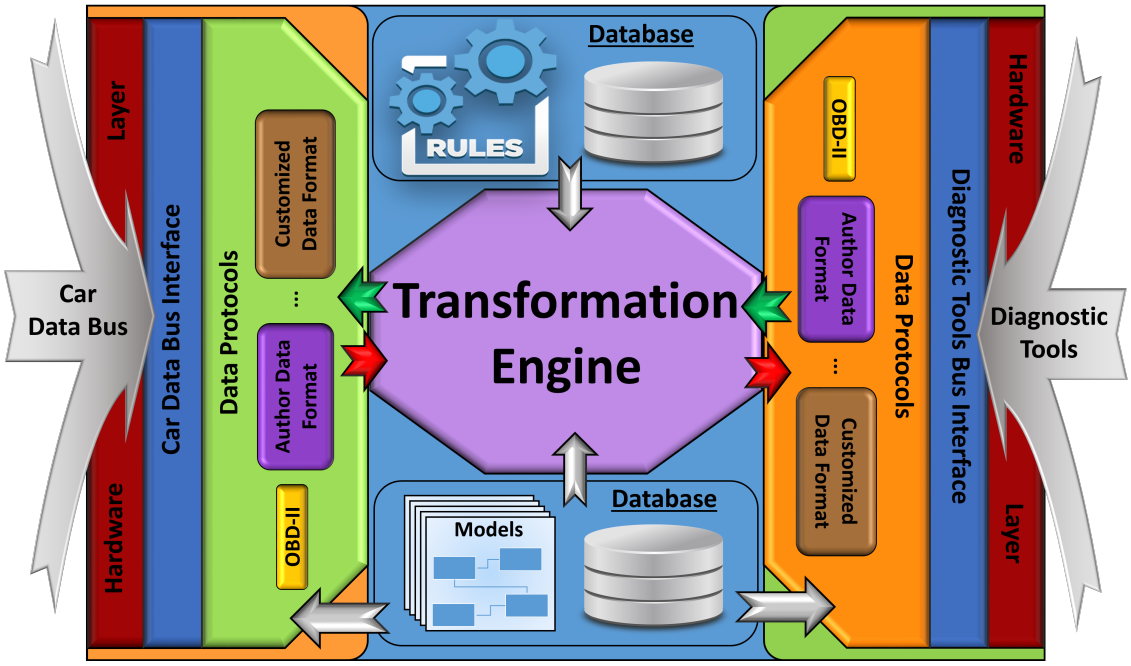

Figure 3.6: Architecture Composition.

CHAPTER 4

EXPERIMENTAL SETUP

This chapter describes supporting technologies that were used in order to perform the implementation. It clarifies the On-Board Diagnostics (OBD) concept and defines all implementation requirements. Finally, an overview of all experimental setup is described.

4.1 On-Board Diagnostics

On-Board Diagnostics (OBD) is an electronic diagnostic system which has the capacity to receive information transmitted from car sensors. This sensors, controlled by Electronic Control Unit (ECU), are designed to ensure the security of several car subsystems. The ECU controls all the fundamental systems of the car, such as, automatic transmission, drive system, brake system, and steering system. The sensors parameters analysed are used by the ECU, ensuring the safety and efficient operation of the vehicle. Usually it takes more time to diagnose a problem in a vehicle than to rectify it and through this systems it is possible to do an efficient diagnose, saving time (Zaldivar et al. 2011).

The OBD standards were developed to detect car engine issues that can provoke gas emission levels beyond acceptable limits. The first OBD generation, known as OBD-I, did not establish a specific emission level for vehicles and only a few diagnostic parameters were defined. However the major disadvantage was the non-standardization, what brought a different OBD system to each car brand.

The second generation of OBD, OBD-II, brought a new standardize connector used for diagnostic, known as Data Link Connector (DLC). The list of vehicle parameters to monitor, the electrical signalling protocols, and the message format were others standardize features. It also brought an enhanced definition of communications protocols, regulated

by the Society of Automotive Engineers (SAE) and International Organization for Standardization (ISO), who standardized five different protocols: SAE J1850 PWM, SAE J1850 VPW, ISO 9141-2, ISO 14230 KWP2000, and ISO 15765 CAN. These protocols have significant differences between them in terms of the electrical pin assignments. Since 2008, the majority of vehicles are equipped with the ISO 15765-4 CAN protocol or just Controller Area Network (CAN), due to a greater flexibility, immunity to noise and speed.

Diagnostic Trouble Codes (DTC) were strongly regulated, allowing technicians to easily determine the reason of a vehicle malfunctioning, using generic scanners. The proposed format, assigns alphanumeric codes to different causes of failure, although extensions to the standard are allowed to support manufacturer-specific failures.

The adoption of different message priorities was an important improvement in the OBD-II data format, in order to make sure that critical information is processed first. The frame formats allow up to 7 data bytes and include a checksum field to detect transmission errors (Hasan et al. 2011).

The OBD communication is stablish via OBD-II connector, represented in Figure 4.1, which allows the transmission and reception of codes. The Parameter ID (PID), are the serial of codes used to request data from vehicles. SAE standard J-1939 defines many PIDs, but manufacturers also customized they own specific PIDs to their vehicles. After sending the PID request, the scan tool sends it to the vehicle's through the pre-defined OBD protocol (CAN, PWM, VPW, KWP or OBD Standards). A device on the car's bus recognizes the respective PID and reports with a reply value of it to the bus. Finally the scan tool reads the response, and displays it in an appropriate device.

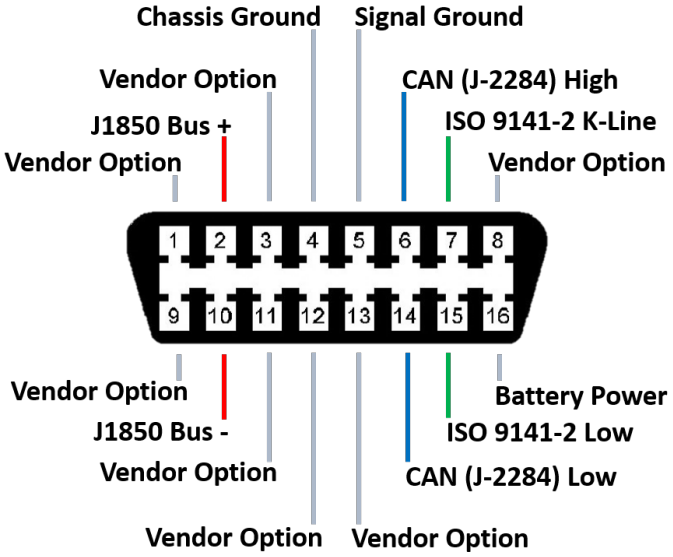

Figure 4.1: OBD-II connector and respective pinout.

Each PID is characterized by its mode, code, number of returned bytes, minimum value, maximum value, units and correspondent formula. On the SAE standard, J-1939, the PIDs codes are subdivided per ten different Modes (in hexadecimal):

- 01 - Show current data;
- 02 - Show freeze frame data;
- 03 - Show stored Diagnostic Trouble Codes;
- 04 - Clear Diagnostic Trouble Codes and stored values;
- 05 - Test results, oxygen sensor monitoring (except CAN);
- 06 - Test results, other component/system monitoring (Test results, oxygen sensor monitoring for CAN only);
- 07 - Show pending Diagnostic Trouble Codes (detected during current or last driving cycle);
- 08 - Control operation of on-board component/system;
- 09 - Request vehicle information;
- 0A - Permanent Diagnostic Trouble Codes.

The communication is provided by the referred five protocols, however there are common points in each data frame sent. All data requests might contain the number of additional data bytes, the Mode and PID Code. On the other hand, the response data must have all referred bytes and a value for the specified parameter. Afterwards, the final value is calculate by its correspondent formula.

For instance, lets consider a CAN communication for current data of engine coolant temperature, where there is a Request data frame and a respective Response, represented in the example of Figure 4.2. The Request data frame is composed by: the value 2 in position 0 which represents two additional data bytes, Mode 01 in position 1 and the PID code 05 for engine coolant temperature, in position 2. The remain 5 positions are not used and in CAN protocol may be 55 in hexadecimal.

The Response data frame (see Figure 4.2) is composed with a similar structure of the previous one. However, it contains 3 additional data bytes, the Mode in byte 1 is converted to hexadecimal (41h = 1d) and the additional third byte contains the value for engine coolant temperature, 130 in this case. The next 3 bytes are optional and only used if necessary. Therefore, applying the correspondent formula for the PID 05, the result will be 90 degrees for engine coolant temperature (130 - 40 = 90).

The next Figure 4.2, presents a request and respective response data frame for current data of engine coolant temperature. The example represents CAN data frames, scrutinized in section 5.1.

Figure 4.2: Request/Response CAN data frames.

4.2 Supporting Technologies

Before execute the planned implementation it is necessary to define its requirements. This section describes the technologies used in order to perform experiments, from software side to hardware one.

4.2.1 Hardware

In the next subsection it will be define all hardware technologies used during the implementation.

4.2.1.1 OBD Scanner

OBD scanner is an electronic tool used to interface with vehicles control modules. The OBD scan tool is connected to the vehicle's Data Link Connector and reads out Diagnostic Trouble Codes. It allows to transmit/receive data between the five OBD communication protocols and the vehicle. However, this data can be accessed, by the user, via serial or preferably, wireless. With the purpose of chose an adequate OBD Scanner it was predefined a few characteristics: inexpensive, multi-platform for the main OBD-II protocols and smart devices. These characteristics justified the choice by ELM327 Interface Bluetooth version, shown in Figure 4.3, with an approximately price of 3.58€[1].

[1] Available at `https://goo.gl/QBLv1q` - Accessed 21/08/2016.

Figure 4.3: The ELM327 Interface Bluetooth version.

4.2.1.2 Arduino

An arduino has the responsibility to emulate all Electronic Control Unit (ECU). This microcontroller-based kit allows to build digital systems and interactive objects that can sense the environment and control physical devices. It was chosen for this implementation due to its cheap price, simple and clear programming environment, open source and extensible software/hardware feature. The Arduino board elected was the Arduino Due, represented in Figure 4.4. Is the most suitable platform for the proposed problem due to its superior processing capacity in relation to others Arduino boards. Another important feature for this thesis, is that it has the CAN communication protocol semi implemented. Having implemented the digital part makes itself possible connect to the OBD scanner tool (Arduino -).

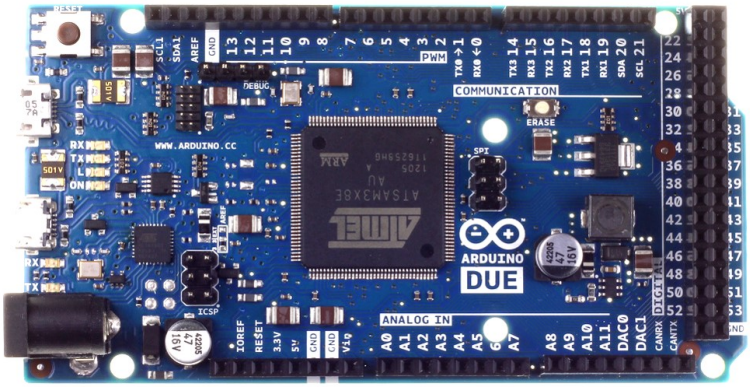

Figure 4.4: Front side of Arduino Due.

4.2.1.3 CAN Transceiver

A CAN Transceiver allows communication between controller devices with CAN modules in multi processor systems. It is used for data exchange through a transmission line with 120Ω termination resistors at both ends of the bus (Barrenscheen 2002). Since the chosen development board was the Arduino Due, a physical connection and analog part had to be considered, in order to allow CAN communication. The need to connect an Arduino Due with a OBD scanner led to the need to manufacture a Can Transceiver. It was chosen a MCP2551 since it has high-speed operation and high-noise immunity. The next Figure 4.3 represents the circuit schematic versus the real implementation of it.

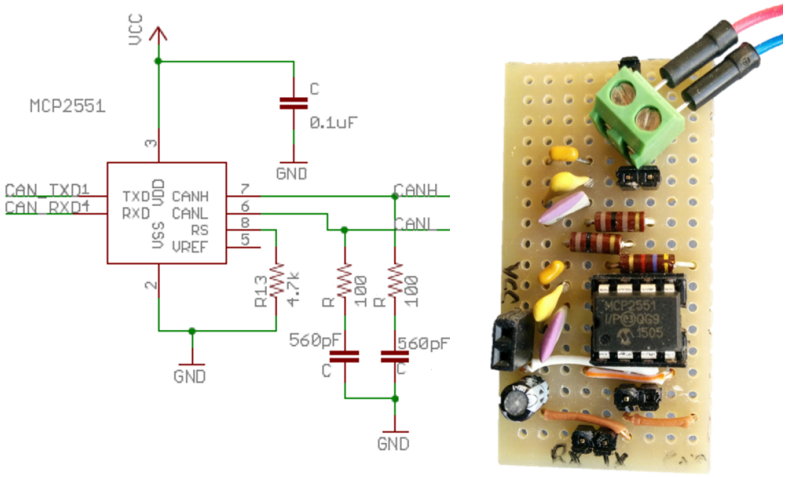

Figure 4.5: MCP2551 circuit schematic versus MCP2551 real implementation.

4.2.2 Software

All software features used in the implementation are described in this subsection.

4.2.2.1 Torque Lite (OBD2 & Car)

The Torque Lite App is an OBD2 performance and diagnostic tool for smart devices. It allows to communicate with a OBD scanner tool, via wireless communication(e.g.: bluetooth), in order to access real time data of vehicles engine as well as view and clear trouble codes. It allows to customize the interface in order to only show the selected data. The Torque Lite App was chosen mainly due to is high performance, simplicity and free of cost. It can be easily downloaded to any smart device which runs Android operating system[2].

[2]Available at `https://play.google.com/store/apps/details?id=org.prowl.torquefree&hl` - Accessed 21/08/2016.

4.2.2.2 Arduino IDE

Arduino Integrated Development Environment (IDE) provides an open-source and easy-to-use programming tool, for writing code and uploading it to arduino's board (Arduino -). It was chosen in line with the Arduino Due choice.

4.2.2.3 Project Cars

In order to simulate real car data it was chosen the Project Cars video game. Project Cars is a motorsport racing simulator video game which intends to represent a realistic driving simulation. It was released in May 2015 and offers a C++ interface to export different real-time information about the simulator. This interface allows access to structured data, using the Shared Memory concept which offers an efficient mean of passing data between programs.

There was found two other options to simulate real car data, like the TORCS - The Open Racing Car Simulator and the Euro Truck Simulator. The first one was discarded because of its arcade and underdeveloped system, which do not fulfilled the implementation needs. On the other hand Euro Truck Simulator, offered a simplistic shared tool to collect the data comparing with Project Cars. Furthermore, the Truck simulation environment do not reflect the common vehicle.

4.2.2.4 Microsoft Visual Studio

Microsoft Visual Studio is an IDE from Microsoft and it is used to develop computer programs for Microsoft Windows, as well as web sites, web applications and web services. It can produce both native code and managed code. The Visual Studio Enterprise version was chosen has a developer software for the communication enabler between Project Cars and Arduino Due. The main reason for this choice was a free licence offered due to a protocol between Microsoft and Faculdade de Ciências e Tecnologias.

4.3 Experimental Setup Overview

The final implementation, shown in Figure 4.6, was the combined result of hardware and software components, describe above. Succinctly, a smart device with a Torque Lite App request vehicle data information. The OBD Scanner receives the request and sends it to the CAN Transceiver, which converts the analog signal to a digital one for Arduino Due. The referred microcontroller sends it to a computer through serial COM, where the communication enabler, developed with Visual Studio, replies with data from Project Cars. The replies goes through in the reverse direction described, until be successfully received by the smart device.

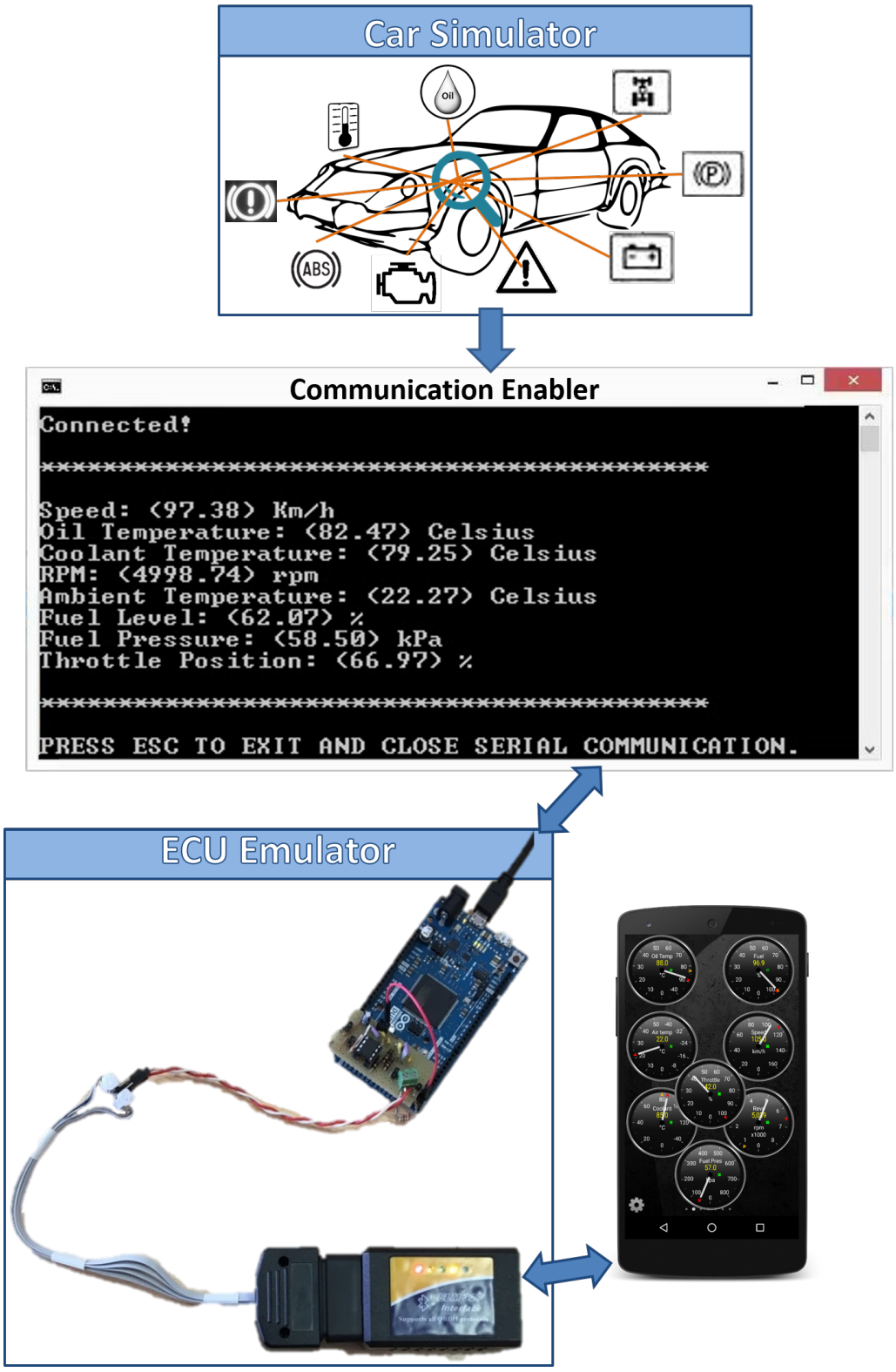

Figure 4.6: Experimental Setup Overview.

CHAPTER 5

IMPLEMENTATION AND VALIDATION

This chapter contemplates a detailed view of the implemented solution which is based on the proposed architecture, described in Chapter 3. It starts to present a description of Controller Area Network protocol and scrutinise three different implementations with its corresponding validation. Finally, it is shown today's actual solutions for multi-interfaces On-Board Diagnostics and it is compared with the final implementation of this thesis.

5.1 Controller Area Network

Controller Area Network (CAN) bus is one of five protocols used in the OBD-II vehicle diagnostics standard and since 2008, cars in USA must have it as one of the protocols. It is a perfect solution for applications which requires a large number of short messages with high reliability in rugged operating environments. Due to its message based, it is especially well suited for situations, where data is needed by more than one system and data consistency is mandatory (Corrigan 2008).

CAN provides an inexpensive network solution, where communication between multiple devices is needed. An advantage to this is that vehicle's Electronic Control Units (ECU) can have a significantly reduction of wiring. Each device can decide if a message is relevant or not, which allows modifications to CAN networks with minimal impact. Every message has a priority, enabling networks to meet deterministic timing constraints. The CAN specification includes a Cyclic Redundancy Code (CRC) to perform error checking on each frame. If too many errors are detected, individual nodes can stop transmitting or disconnect itself from the network completely (National Instruments 2014). The next Figure 5.1 represents the CAN Bus influence for wiring reduction, in a networks.

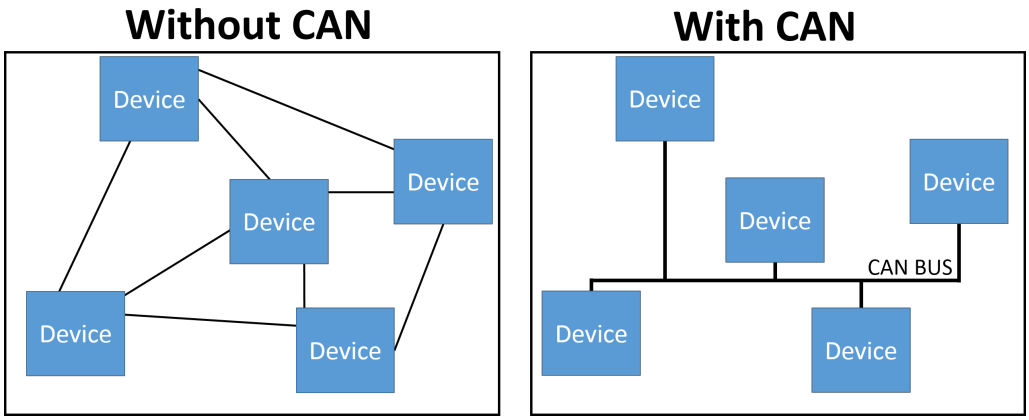

Figure 5.1: Network with and without CAN, based on (National Instruments 2014).

CAN bus uses two dedicated wires for communication, called CAN high and CAN low. When the CAN bus is in idle mode, both lines carry 2.5 Volts. When data bits are being transmitted, the CAN high line goes to 3.75 Volts and the CAN low drops to 1.25 Volts, thereby generating a 2.5 Volts differential between the lines, as demonstrate in Figure 5.2. Due to its voltage differential, the CAN bus is not sensitive to inductive spikes, electrical fields or other noise.

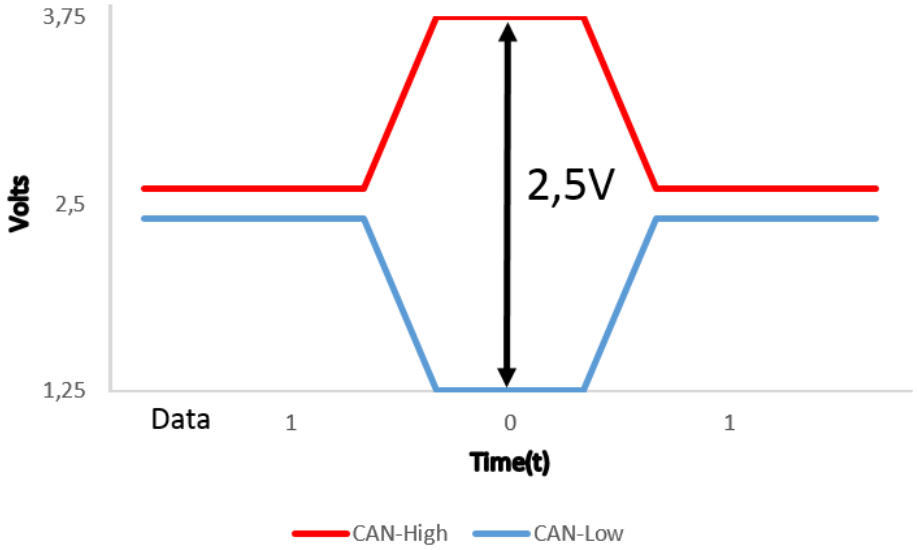

Figure 5.2: Can communication electrical levels view, based on (Axiomatic 2006).

CAN devices send data across the CAN network in packets called frames. There is two different frames, the Standard CAN with a 11-bit identifier and Extended CAN with 29-bit identifier. Both of them provides a transmission rate from 125kbps to 1 Mbps. The base frame format, in Figure 5.3, is composed by:

- SOF: The Start Of Frame (SOF) bit marks the start of a message, and is used to synchronize the nodes on a bus after being idle.

- Identifier: The Standard CAN 11-bit identifier establishes the priority of the message. The lower the binary value, the higher its priority.
- RTR: The Remote Transmission Request (RTR) bit is dominant when information is required from another node. All nodes receive the request, but the identifier determines the destination node. In this way, all data being used in a system is uniform.
- IDE: The Identifier Extension bit is set to zero meaning that there will be 11 identifier bits to be transmitted.
- r0: It is a reserved bit for possible use by future standard amendment.
- DLC: The 4-bit Data Length Code contains the number of data bytes being transmitted.
- Data: Up to 64 bits of application data may be transmitted.
- CRC: The 16-bit CRC contains the checksum. It is composed by the number of bits transmitted which precedes application data for error detection.
- ACK: Every node receiving an accurate message, overwrites this recessive bit in the original message with a dominate bit, indicating an error-free message has been sent. Should a receiving node detect an error and leave this bit recessive, it discards the message and the sending node repeats the message after rearbitration. In this way, each node Acknowledges (ACK) the integrity of its data. ACK is 2 bits, one is the acknowledgment bit and the second is a delimiter.
- EOF: The End Of Frame (EOF) can be up to 7-bit and marks the end of a CAN frame disabling bit-stuffing, indicating a stuffing error when dominant.
- IFS: The Interframe Space (IFS) it also can be up to 7-bits and contains the time required by the controller to move a correctly received frame to its proper position in a message buffer area. Combination between EOF and IFS needs to be of into 10-bits size.

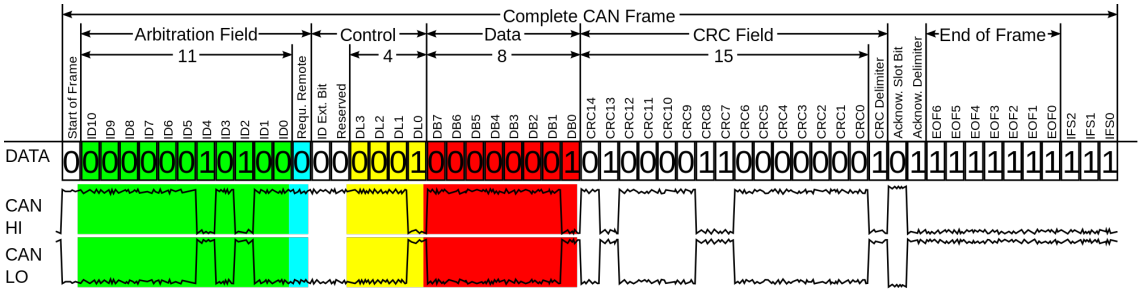

Figure 5.3: CAN-Frame in base format with electrical levels (Can Bus - Wikipedia).

The Extended CAN messages differ from Standard CAN frame in the following positions (Corrigan 2008):

- SRR : The 1-bit Substitute Remote Request replaces the RTR Bit that would be sent in a Standard CAN message.
- IDE: The 1-bit Identifier Extension is set to high to indicate that there will be 29 identifier bits.
- Control Field: It is compose by 6-bits, where the first two are reserved, r0 and r1. The remaining bits have the same meaning than in Standard CAN frame.

5.2 Testing and Validation

The first target in the implementation process was to connect, via CAN, two Arduinos. One to work as transmitter and the other one as receiver. Subsequently, the second implementation stage was establish a communication between the OBD Scanner module with one Arduino. Arduino which was emulating the vehicle's ECU with random data. Finally, the third and last stage was accomplish with a bridge between Car Simulation module and the previous approach. This was achieved via a Communication Enabler module, which replaces the random data generation for a car simulator module. Furthermore, to simplify the final implementation description it is fundamental to retain some basic ideas. The communication overview, shown in Figure 5.4, is a based start for all subsequent explanation.

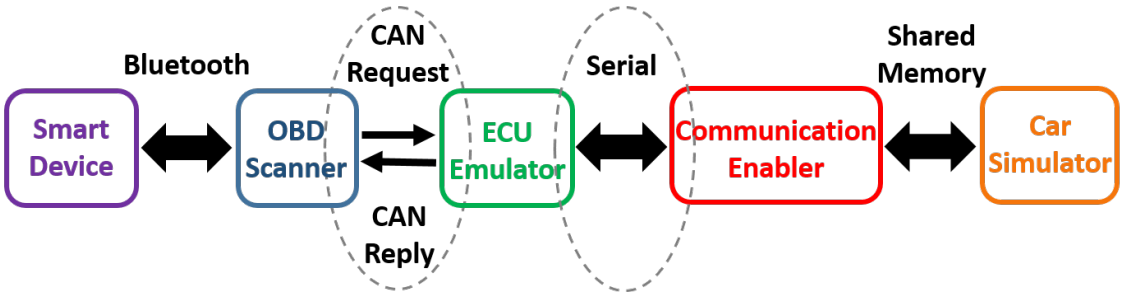

Figure 5.4: Communication Overview.

As it is possible to see in Figure 5.4, a Smart Device communicates via bluetooth to the OBD Scanner module. The OBD module fowards requests/replies between the Smart Device and the ECU Emulator. It sends a CAN Request to ECU Emulator and this one replies in the same format. The connection between the ECU and Communication Enabler module is made by Serial. The dashed line in the Figure 5.4 represents two communication types (CAN and Serial) implemented by the author. The Bluetooth communication comes already with the smart device and OBD tool, as long as, the Shared Memory is offered by

the car simulator.

In this particular case, the serial frames where customized in order to fulfill the implementation needs. In this sense, the Request Frame presented in Figure 5.5 is composed by one byte with the Mode, other one with PID Value and three bytes with 255 each for message control. This frame is sent from the ECU Emulator to the Communication Enabler module.

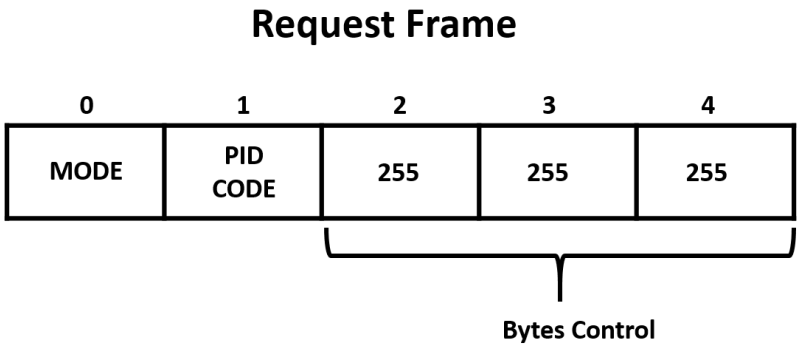

Figure 5.5: Request Frame.

The Response Frame contains one byte for the Mode, another one for PID Code and four bytes of PID Value. The frame's, last 3 bytes are reserved for message control once again. This frame is received by the ECU Emulator and comes from the Communication Enabler module.

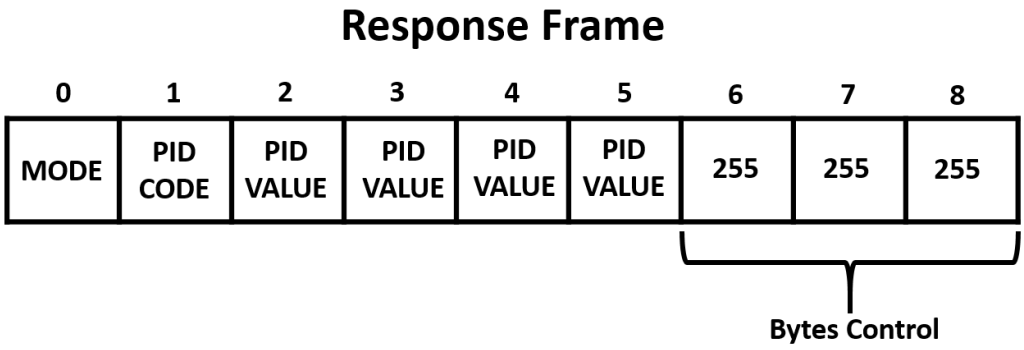

Figure 5.6: Response Frame.

To finalize the communication overview the last two modules, Communication Enabler and Car Simulator, are connected via a Shared Memory.

5.2.1 Data Protocols Meta-Models

In this section, the Data Protocols Meta-Models using during the implementation are presented. First, the Meta-Model for Data Protocol used by the OBD-II was built based on the data exchange presented in (OBD-II PIDs - Wikipedia). The OBD-II data exchange occurs between the OBD module and the Arduino. Next is presented the Meta-Model used by the author, which describes the way that messages are exchange between the Arduino and the Simulator.

5.2.1.1 OBD-II Meta-Model

The OBD-II Meta-Model is composed by a Mode, where the type of data is defined. A PID code, which is a parameter identification code supported by the referred Mode. The Value is the concrete data associated to the correspondent PID code. Finally there is the Addictional byte size and Addictional bytes only used if necessary.

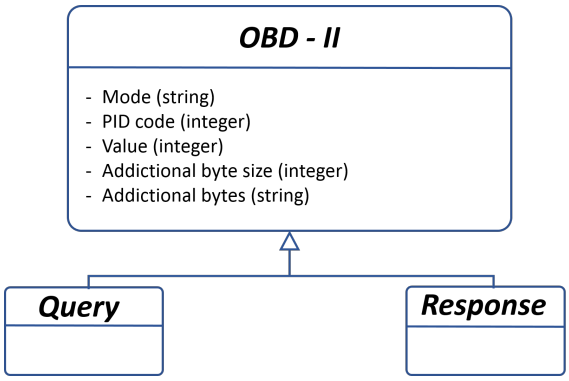

Figure 5.7: OBD-II Model, based on (OBD-II PIDs - Wikipedia).

5.2.1.2 Author Data Protocols

The Author Data Protocols is composed by the Mode, PID and PID value which has the same specifications of the previous description. However there is a field for Byte control in order to guarantee quality of service.

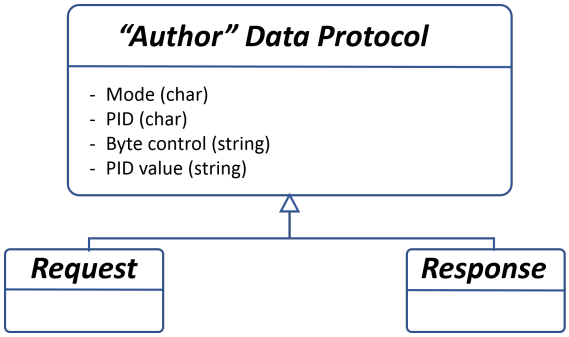

Figure 5.8: "Author" Data Protocol Model, based on (OBD-II PIDs - Wikipedia).

5.2.1.3 Rules/Mapping

The next table 5.1, presents the Meta-Model mapping between OBD-II protocols and the Author data protocols. The mapping for Mode and PIDcode of the element Query are directly mapped with the Mode and PID for Response element in the author protocol. The R.PIDvalue can be composed by the combination of Q.value and Q.AdditionalByte. The Q.AdditionalByte is used in order to contemplate at least two decimal numbers in the value data exchanged.

Table 5.1: Meta-Models Mapping, based on (OBD-II PIDs - Wikipedia).

OBD-II Meta-Model	**Author Data Protocols**
Query:Q	Request:R
Q.Mode Q.PIDcode Q.Value Q.Additionalbyte Q.AdditionalbyteSize	R.Mode R.PID R.PIDvalue
Response:R1	Response:R2
R1.Mode R1.PID R1.Value R1.Additionalbytes R1.AdditionalbyteSize	R2.Mode R2.PID R2.PIDvalue

5.2.2 Arduino to Arduino CAN communication

For the Arduino to Arduino implementation there are two similar routines. The transmitter routine, showed in Figure 5.9-a, starts at the moment a new CAN message is available to be sent. After it, message control assures all bytes are received correctly. If is not, the routine return back and waits for a new message. A correct message is compliant with the CAN frame standard. Afterwards, the CAN Message is sent to the Receiver and the process returns to the beginning, waiting for a new message.

The receiver routine, represented in Figure 5.9-b, starts when a CAN Message is received and it is posteriorly submitted to the same message bytes control. However, in case of a positive response from bytes control, the message is processed, decomposing the main bytes from its frame. Finally, the CAN message is printed to a simple user interface and the process returns back waiting for a new CAN Message.

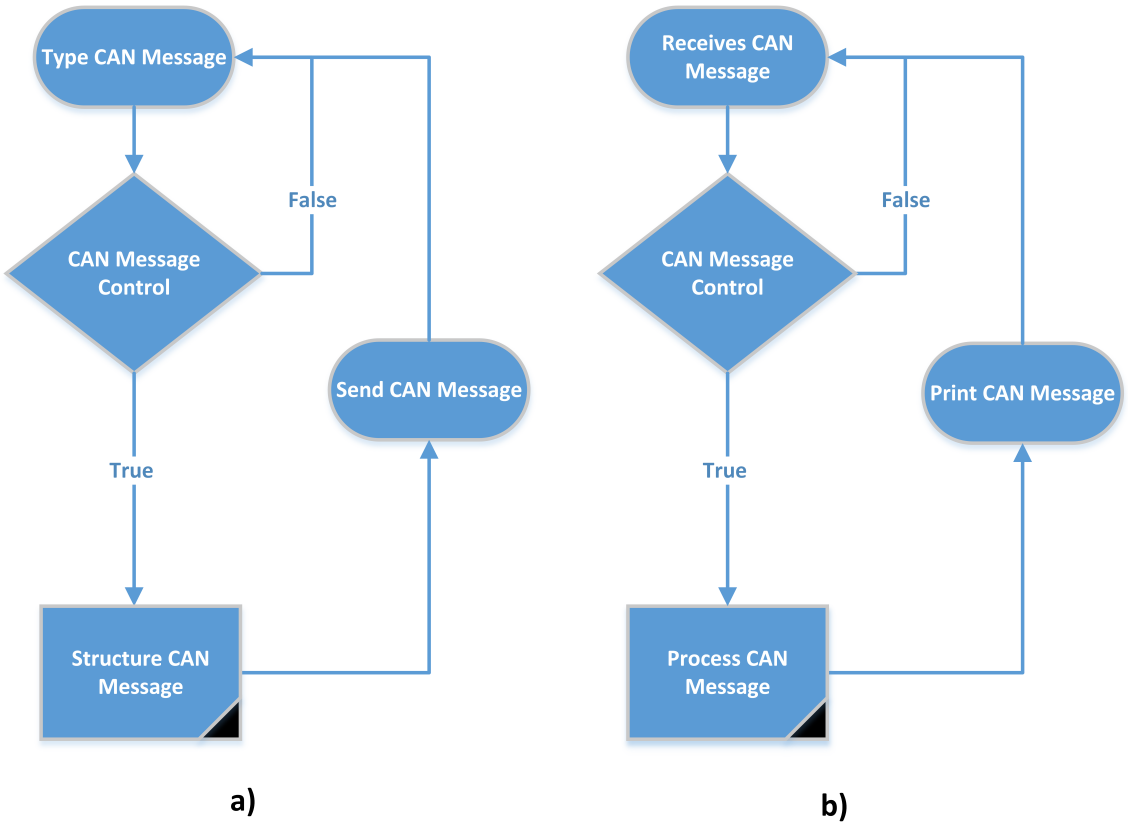

Figure 5.9: Arduino to Arduino flowchart (a - Transmitter; b - Receiver).

The Figure 5.10 shows the Arduino to Arduino implementation connected through a CAN Bus, with one CAN transceiver in each end.

Figure 5.10: Arduino to Arduino implementation.

In order to validate this approach the CAN High and Low were connected to an oscilloscope in two different channels. The Figure 5.11 shows the symmetry between the two lines which represents a simple CAN frame message.

The used oscilloscope, performs a reading of 2500 points for each time window. It was calibrated with a 10.0 microsecond/division in the horizontal axis (in a total of 100μs window) and a 2.0 volts/division at the vertical axis.

The sign wave 1 (orange line) illustrates CAN High and the sign wave 2 (blue line) illustrates CAN Low signal.

Figure 5.11: Arduino to Arduino validation.

5.2.3 Connect OBD Scanner Module with Arduino

The OBD Scanner Module to Arduino routine has its starting point when a new CAN Request Message is received. Then the message is decompose, extracting the Mode and PID.

Afterwards it is submitted to a check control for Mode 1 and PID match availability. In this way only known PID for Mode 1 will have a response. If the check control comes positive all random variables generated by the arduino are updated and the reply message structured. The message reply is composed by the Mode 1, the request PID and data value from the correspondent random variable.

Finally the CAN Reply Message is sent to the OBD tool and the process returns to its initial point waiting for a CAN Request Message, as well as if the check control for Mode and PID, fails.

The next Figure 5.12 represents a flowchart diagram for the explained routine.

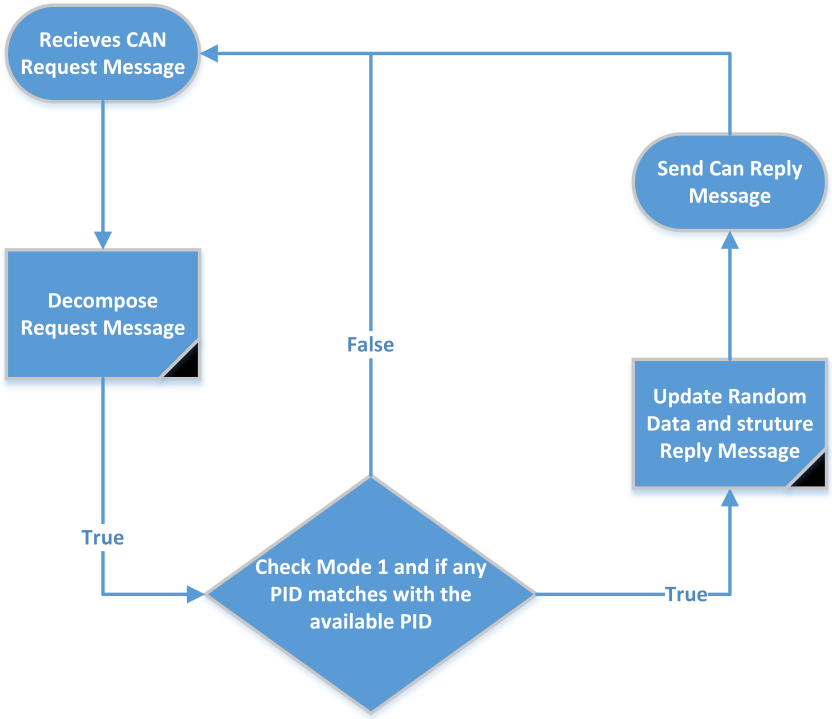

Figure 5.12: OBD Scanner Module to Arduino flowchart.

The correspondent OBD Scanner to Arduino implementation is in the next Figure 5.13. In this case, the transceiver is directly connected to the OBD Scanner tool through a CAN Bus. The smartphone send requests to the OBD module, which reroutes them to the Arduino. This microcontroller replies back thought the inverted way described. Finally, the reply

appears on the smartphone interface.

Figure 5.13: OBD Scanner Module to Arduino implementation.

The validation of this feature was made through a oscilloscope connecting CAN High and Low in two different channels. In the Figure 5.14 it is also obvious the symmetry between the two lines but in this case the CAN frame message is more complex, due to the countless request/reply exchange.

The used oscilloscope, process 2500 points for each time window. It was calibrated with a 100.0 microsecond/division in the horizontal axis (in a total of 1000μs window) and a 2.0 volts/division at the vertical axis. Once again the sign wave 1 (orange line) illustrates CAN High and the sign wave 2 (blue line) illustrates CAN Low signal.

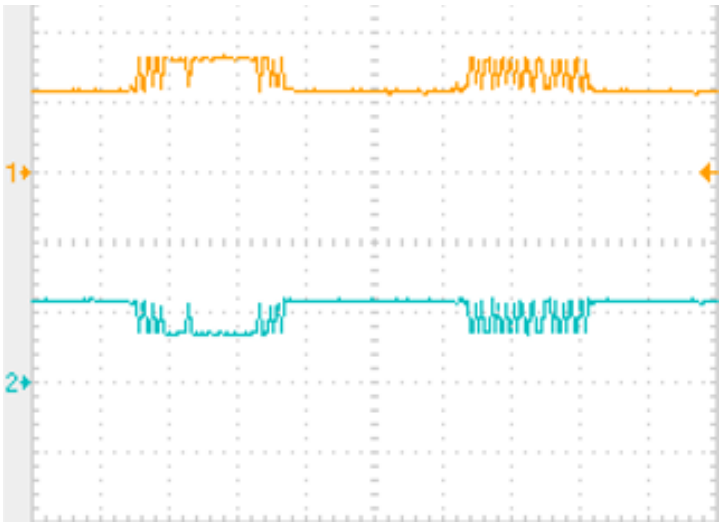

Figure 5.14: OBD Scanner Module to Arduino validation.

5.2.4 Final Demonstrator: Open Source OBD-II Emulator

The OBD-II Emulator is a combination of ECU Emulator and Communication Enabler module. In order to stablish this connection, there is a subroutine inside of the Communication Enabler, called Enabler.

In this case the ECU Emulator flowchart, represented in Figure 5.15, is a derivative of the previous Figure 5.12. However, when it receives a new CAN Request, with the Mode and PID Value, it converts the message to the explained Request Frame. Afterwards, sends it to the Enabler routine, shown in Figure 5.16, by serial.

Other difference relays on the message control, which waits for the Enabler response immediately after the Request Frame is sent. Data validation is also performed in this step and if it fails, it will wait for a new Response Frame from the Enabler.

With a successfully Response Frame received, the routine proceeds to the check control for Mode 1 and PID matching. If from this step results a false response, the routine returns to the begin waiting for a new CAN Request message. On the other hand if the check control returns true, the Response Frame is restructured to a CAN Reply message and sent to the OBD Scanner module. Finally it returns to wait for a new CAN Request.

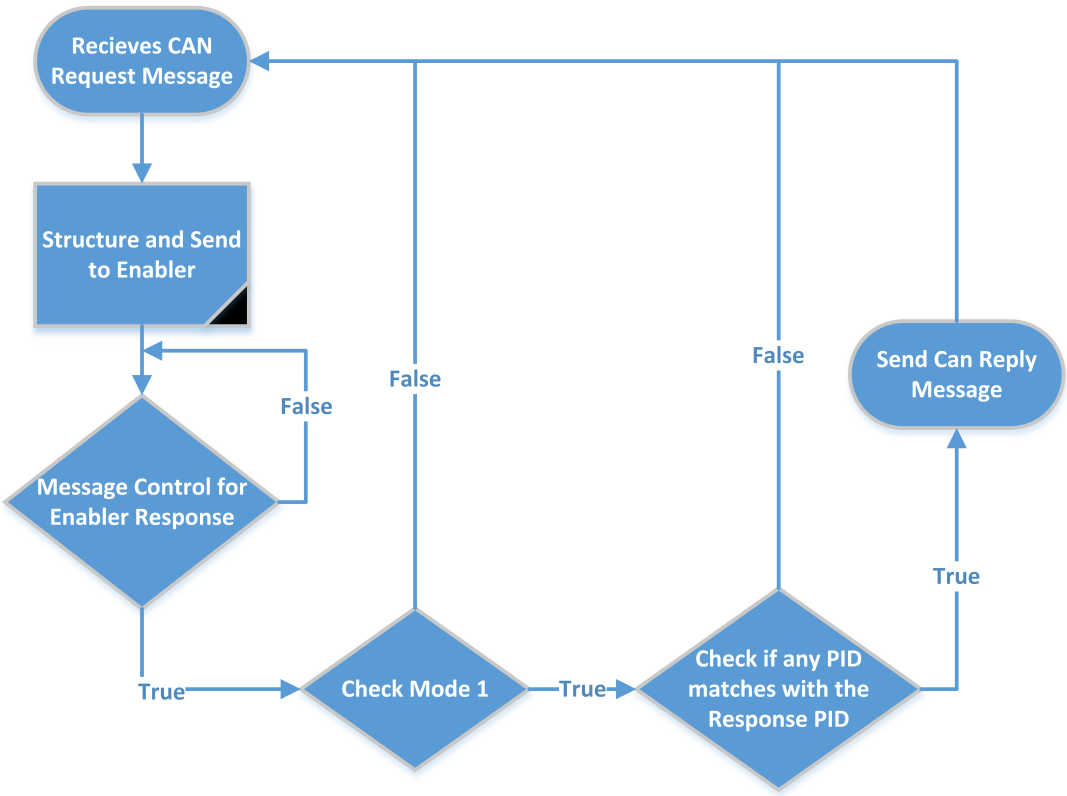

Figure 5.15: ECU Emulator flowchart.

The Enabler, presented in Figure 5.16, starts its routine when a new Request Frame is received from the ECU Emulator.

Afterwards, it is submitted to message control, checking if the message contains mode 1 and if PID matches with the available ones. If it passes all decisions with success, the Response Frame is composed with the corresponding data from the car simulator. Then it is sent to the ECU Emulator.

Subsequently, the main Communication Enabler routine is retaking as well as if any decision fails.

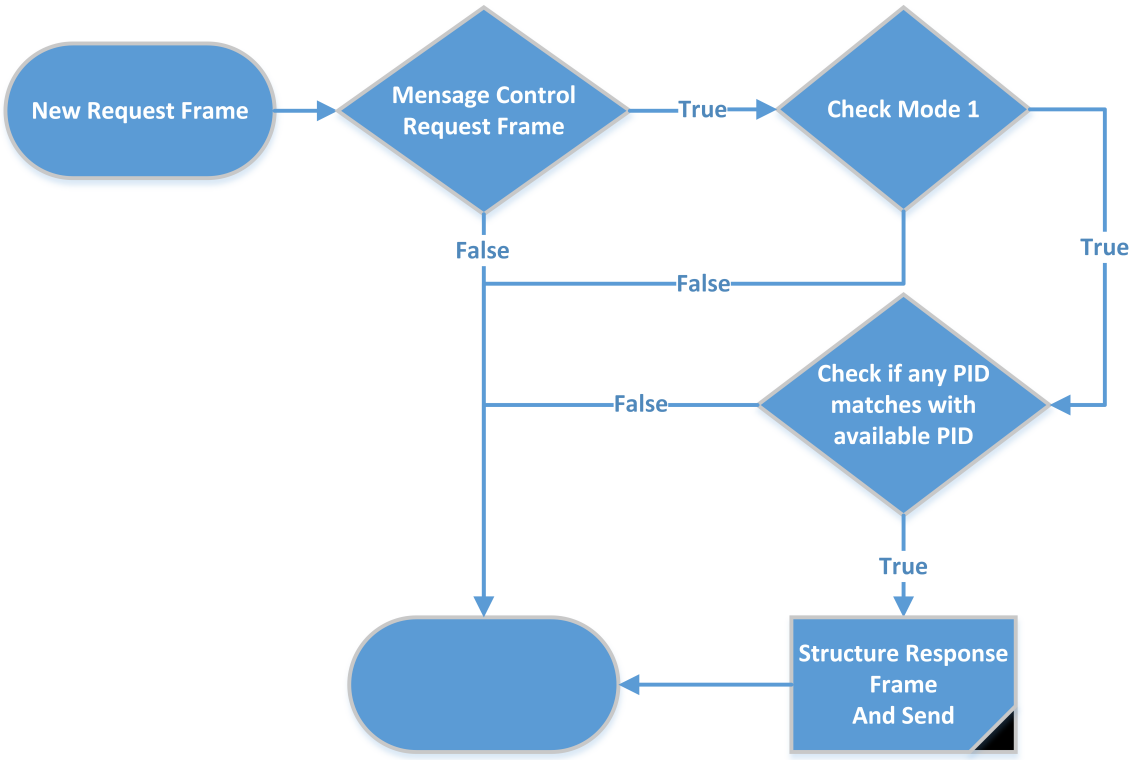

Figure 5.16: Enabler flowchart.

The Communication Enabler routine, shown in Figure 5.17, begins when the car simulator is connected, which in this case is when the shared memory is connected.

Afterwards, it checks if serial communication is stablish. If yes, updates sensors variables from the shared memory and advance for the Enabler subroutine.

Subsequently checks if the exit button was pressed. In case of positive response, the

shared memory and serial communication are closed. If any decision block fails the routine returns back to check if the serial is still connected.

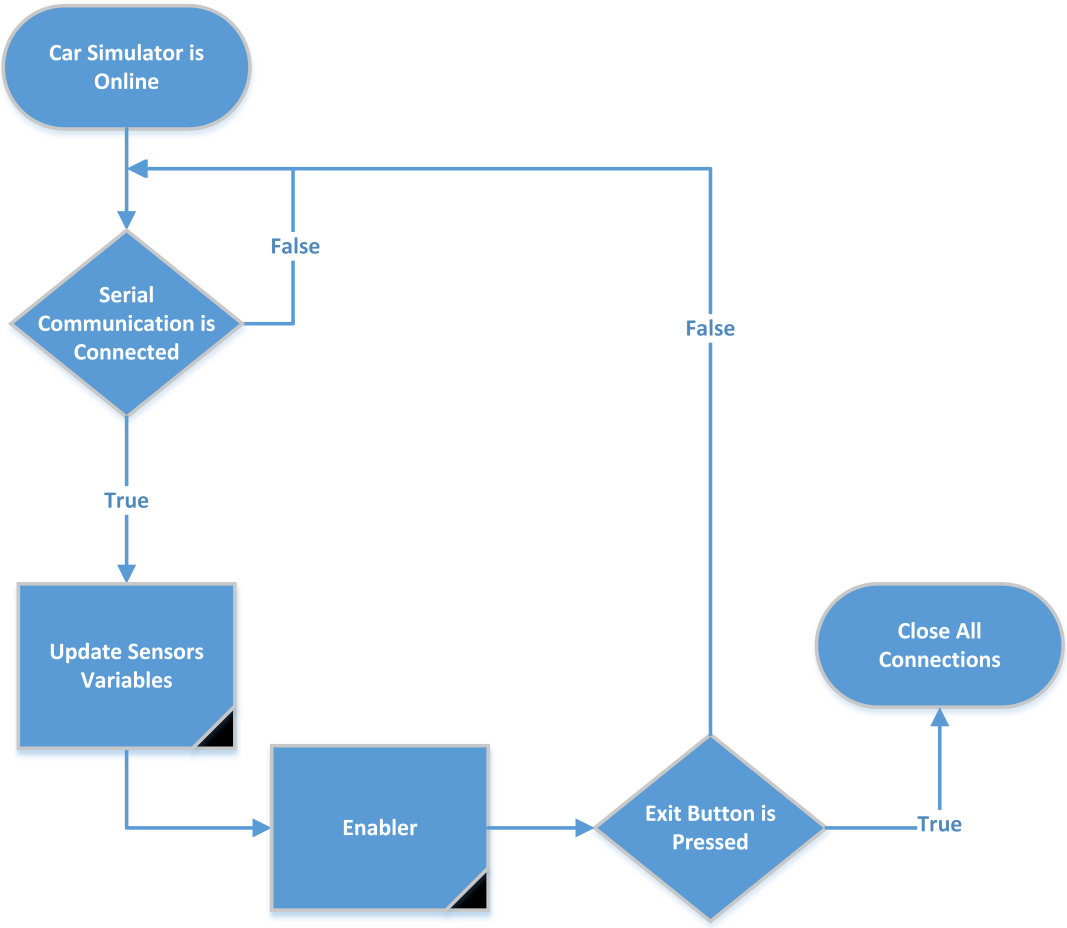

Figure 5.17: Communication Enabler flowchart.

The subsequent final implementation, produced by the author, is represented in Figure 5.18.

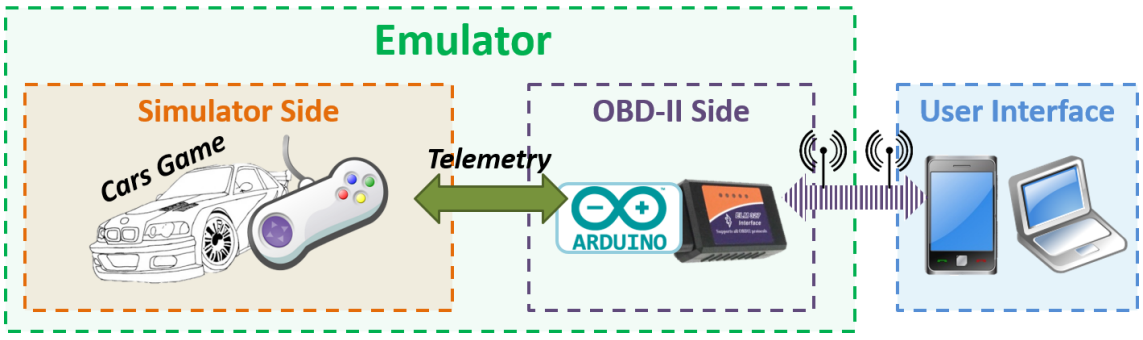

Figure 5.18: Representation of the final implementation.

In order to validate this approach the CAN High and Low were once again connected to an oscilloscope in two different channels. The Figure 5.19 confirms the symmetry between the two lines and described in 5.1. It is possible to see two different CAN frames separated by approximately 250μs.

The used oscilloscope, performs a reading of 2500 points for each time window. It was calibrated with a 100.0 microsecond/division in the horizontal axis (in a total of 1000μs window) and a 2.0 volts/division at the vertical axis.

Again, the sign wave 1, orange line, illustrates CAN High signal and the sign wave 2, blue line, illustrates CAN Low signal.

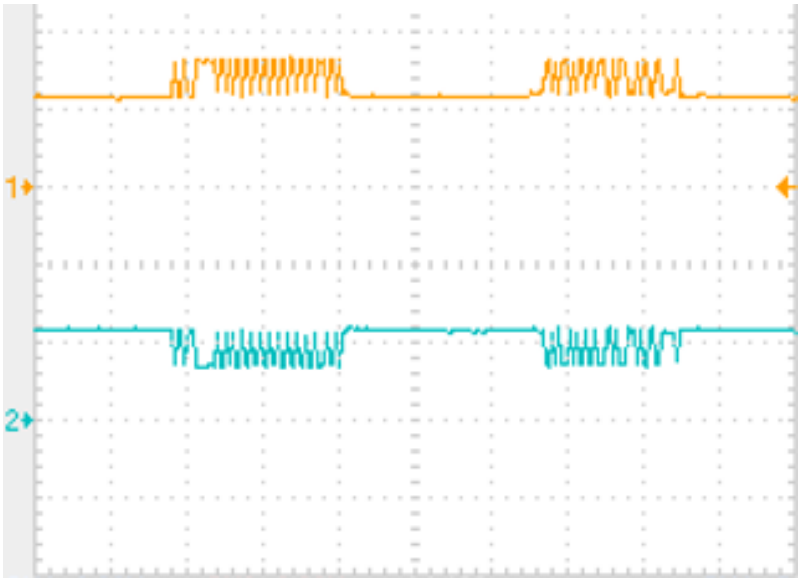

Figure 5.19: Communication Enabler validation.

5.3 Summary: Open Source OBD-II Emulator

The OBD-II Emulator is the solution proposed by the author. It is a low cost and open source solution with CAN, as communication protocol. This emulator is able to simulate all defined PID in Mode 1, therefore it is restricted to the ones supported by the car simulator. It is also supplied with hardware and software capable to provide an easy integration of other communication protocols, Modes and PIDs.

The next Table 5.2 presents a detailed cost list from all components used in order to preform the final implementation of OBD-II Emulator. The components cost is based on eBay prices and smart device applications can be retrieved from Google Play. The Visual Studio code is free of cost due to a protocol between Microsoft and Faculdade de Ciências

e Tecnologias. However, there is many others open source platforms which offers a similar result.

Table 5.2: Open-Source OBD-II Emulator detailed cost.

	Cost [€]:
ELM327 Module with Bluetooth:	3,99
Arduino DUE:	12,50
CAN PCB + components:	3,10
OBD-II Interface:	1,69
Smart device application:	0
Arduino Code:	0
Visual Studio Code:	0
Total:	21,28

The OBD-II Emulator was present in the Open Source Portugal[1] meeting and Expo FCT 2016[2]. The first one is a meeting which aims to bring together national initiatives of various kinds related to open source culture. The second one is an exposition in Faculdade de Ciências e Tecnologia and its main goal is to address the reality in aspects of research, innovation and technology to the population, in particular high school students.

[1] Available at `http://ospt.artica.cc/` - Accessed 12/09/2016.
[2] Available at `http://www.expo.fct.unl.pt/atividades/dee` - Accessed 12/09/2016.

CHAPTER 6

CURRENT SOLUTIONS

In this chapter three current solution will be scrutinize and finally compare with the propose solution. There are in the market some implementations for OBD emulator which is important to take in consideration. This chapter will present a brief description of the three most notable OBD emulator solutions for the author.

6.1 Multiprotocol ECU Simulator - mOByDic4910

This system is made by Ozen Elektronik LTD with a price of 495€. It simulates a single or three different ECU's: Antilock Braking System4 (ABS), Powertrain Control Module (PCM) and Transmission Control Module (TCM). It has a selectable protocol via dip switch and gasoline or diesel PID's. This solution generates DTC for each ECU separately and has a variable PID with eight potentiometers. It has the capacity to simulate the five main protocols: ISO9141-2 , KWP2000 , J1850 PWM , J1850 VPWM and CAN. Finally the system is compatible with the SAE-J1979 regulations and it supports multiframe or multimessages operation (Ozen Elektronik -).

6.2 ECUsim 5100 Professional OBD-II ECU Simulator

The multiprotocol ECUsim 5100 is a small, lightweight simulator that can be used for testing and development of OBD devices and software. It supports all five OBD-II protocols and offers multiple Plug In Modules (PIMs), that allows to run up to three protocols simultaneously, or quickly switch protocols on-the-fly. Only one protocol per PIM can be active at a time and each one supports three virtual ECUs: Engine Control Module (ECM), TCM, and ABS. This product has fixed and user adjustable PIDs: Coolant Temperature, Engine Speed (RPM), Vehicle Speed, Oxygen Sensor Voltage and Mass Airflow (MAF). It is also supports DTCs, freeze frames and others SAE J1979 services. These features are

very useful for production testing and automatic protocol detection. This solution can be acquired for approximately 1360€(OBD Solutions -b). This implementation is an evolved system of the already referred ECUsim 2000 OBD Simulator, in Chapter 2.5 - Research Question.

6.3 Freematics OBD-II Emulator MK2

Freematics OBD-II Emulator MK2 is an OBD-II emulator with KWP2000, ISO9141 and CAN bus simulation. It emulates up to 6 active vehicle diagnostic trouble codes as if a real car has, when encountering a component malfunction or fault. The emulator connects to PC via USB cable and is operated with an open source software. Secondary development is possible with serial connector available for connection of interaction with Arduino or other embedded system with sensors. This OBD emulator has an approximately cost of 224€(Freematics -).

6.4 Comparison

Through a first look to all four solutions it is possible to categorized them in three groups. The two ones with a close source technologies are Multiprotocol ECU Simulator and ECUsim 5100. The one with semi open technology is Freematics, due to its open source software but close hardware. To finalize the author approach, Open Source OBD-II Emulator, is an open hardware and software.

The strong point of Multiprotocol ECU Simulator and ECUsim 5100 are the five implemented communication protocols. In other hand Freematics has 3 in its basic kit and Open Source OBD-II Emulator only have CAN. However, since 2008, vehicles must have CAN as one of the protocols.

Another strong characteristic for Freematics and Open Source OBD-II Emulator is the possibility to add and remove PID's, combined with integration facilities for microcontrollers, such as Arduino. The other two solutions are limeted to some PID's and doesn't support sensors system integration.

To finalized it is evident the differences between all approaches in terms of prince. The author solution, Open Source OBD-II Emulator, has a production cost of 21,28€. This value is ten times cheaper than Freematics, the second one with lowest price. The most relevant conclusions of this analysis are summarised in Table 6.1.

Table 6.1: Comparison.

	Multiprotocol ECU Simulator - mOByDic4910	**ECUsim 5100 Professional OBD-II ECU Simulator**	**Freematics OBD-II Emulator MK2**	**Open Source OBD-II Emulator**
Communication Protocols:	5	5	3	1
PID's:	Limited	Limited	All available	Mode 1
Integration with embedded Sensors Systems:	No	No	Yes	Yes
Open Source Software:	No	No	Yes	Yes
Open Source Hardware:	No	No	No	Yes
Cost [€]:	495	1360	224	21,28

CHAPTER 7

CONCLUSIONS AND FUTURE WORK

The Internet of Things (IoT) experience is growing exponentially nowadays, more specifically because devices and applications are launched frequently. Engineering these systems should take some concerns into account, regarding vehicles embedded of IoT. Developing IoT systems in a car, opens an issue due its price value and sensitivity to safeguard human lives. The need for OBD emulators is an evidence, however this emulators are scarce and costly, discouraged, in this ways, the evolve of new IoT features. In order to promote the development of new applications and devices in vehicles, it is crucial to have open source On-Board Diagnostics (OBD) emulators.

Understanding the challenges in order to increase outcomes led to a research question and hypothesis formulation. In this sense, the developed work proposes an architecture capable to emulate the vehicles environment with multi-protocol support and using a model-based approach.

By setting an experimental setup, in which is used a developed open source OBD emulator, advances have been made in order to stimulate OBD applications. Then, several tests were accomplished with the selected developed system, generating an emulated OBD environment, which was promptly used to present and discuss results.

The OBD emulator developed, offers an open source and inexpensive alternative with integration facilities for hardware and software. After presenting the solution for multiprotocol OBD emulator requirements, results shows that is possible to adopt such methodology for developing new and interoperable OBD systems. It is now opportune to list the future works that can be carried from now on.

An interesting enhancement that could be applied to the solution purposed is the possibility to replace the car simulator by the Freematics open source software, solving the car simulator cost issue. This software is free and can increases control in the simulated values produced by the software. In this way all the software could easily executed in a open source operating system (e.g.: Linux).

Another possible enhancements that could be applied to the architecture, is the possibility of integrating a GPS and GSM modules. This could improve communication interoperability and stimulate the developement of application with location systems. The basic parameters available in a vehicle (e.g.: current speed, revolutions per minute (rpm), the amount of fuel) could be emulate by a physical system in a hardware board.

To finalize, it could be develop a real application capable to connect through multiple communication systems and be able to access data from the internet (e.g.: cloud systems, web-services). The emulator might send data to the OBD module and the cloud, improving its access in case of failure of one of the systems.

BIBLIOGRAPHY

Apache S. F. (2016). *Apache ServiceMix*. The Apache Software Foundation. URL: http://servicemix.apache.org/. Accessed 04/02/2016.

Arduino (-). "Arduino - Introduction". URL: https://www.arduino.cc/en/guide/introduction. Accessed 16/08/2016.

Axiomatic (2006). "What is CAN?" *Axiomatic - Global Electronic Solutions, Aplication Note*. URL: http://www.axiomatic.com/whatiscan.pdf. Accessed 07/09/2016.

Barrenscheen, J. (2002). "On-Board Communication via CAN without Transceiver". *Siemens, Microcontrollers ApNote - AP2921*.

Bocchi, Y., D. Genoud, G. Rizzo, F. Morard, and A. C. Olivieri (2012). *Universal Integration of the Internet of Things through an IPv6-based Service Oriented Architecture enabling heterogeneous components interoperability*. IoT6. URL: http://iot6.eu/. Accessed 03/02/2016.

Bray, T. (2014). *The JavaScript Object Notation (JSON) Data Interchange Format*. RFC 7159. Accessed 03/02/2016.

Can Bus - Wikipedia. "CAN-Bus-frame in base format without stuffbits". URL: https://commons.wikimedia.org/wiki/File:CAN-Bus-frame_in_base_format_without_stuffbits.svg. Accessed 09/09/2016.

Carey, S. S. (2011). *A beginner's guide to scientific method*. Fourth. Wadsworth Pub Co.

Compton, M. et al. (2012). "The SSN Ontology of the W3C Semantic Sensor Network Incubator Group". *Web Semantics: Science, Services and Agents on the World Wide Web* 17.0. ISSN: 1570-8268. URL: http://www.websemanticsjournal.org/index.php/ps/article/view/312. Accessed 16/02/2016.

Corrigan, S. (2008). "Introduction to the Controller Area Network (CAN)". *Texas Instruments - Application Report*. URL: http://www.ti.com/lit/an/sloa101a/sloa101a.pdf. Accessed 07/09/2016.

Datta, S., C. Bonnet, and N. Nikaein (2014). "An IoT gateway centric architecture to provide novel M2M services". *Internet of Things (WF-IoT), 2014 IEEE World Forum on*, pp. 514–519. DOI: 10.1109/WF-IoT.2014.6803221.

ECMA-404 (2013). *The JSON Data Interchange Format*. EMAC International. URL: http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf. Accessed 14/02/2016.

Favre, J. marie (2004). "Towards a Basic Theory to Model Model Driven Engineering". *Third Workshop in Software Model Engineering (WiSME@UML)*. URL: http://www-adele.imag.fr/Les.Publications/intConferences/WISME2004Fav.pdf. Accessed 18/02/2016.

Fielding, R. T. and R. N. Taylor (2000). "Principled Design of the Modern Web Architecture". *Proceedings of the 22Nd International Conference on Software Engineering*. ICSE '00. New York, NY, USA: ACM, pp. 407–416. URL: http://doi.acm.org/10.1145/337180.337228.

Freematics (-). "Freematics OBD-II Emulator MK2". URL: http://freematics.com/store/index.php?route=product/product&product_id=71. Accessed 11/09/2016.

GASwerk (2016). "GASwerk - Geronimo Application Server Assemblies". URL: http://gaswerk.sourceforge.net/. Accessed 28/04/2016.

GSN Team (2014). *Global Sensors Networks*. URL: https://github.com/LSIR/gsn/blob/documentations/book-of-gsn/main.pdf?raw=true. Accessed 10/02/2016.

Hasan, N., A. Arif, U. Pervez, M. Hassam, and S. Husnain (2011). "Micro-controller Based On-Board Diagnostic (OBD) System for Non-OBD Vehicles". *Computer Modelling and Simulation (UKSim), 2011 UkSim 13th International Conference on*, pp. 540–544. DOI: 10.1109/UKSIM.2011.109.

IPv6 Google (-). "World-wide IPv6 deployment as measured by Google". URL: http://www.google.com/intl/en/ipv6/statistics.html#tab=ipv6-adoption&tab=ipv6-adoption. Accessed 19/02/2016.

Jennings, C., Z. Shelby, and J. Arkko (2013). "Media Types for Sensor Markup Language (SENML)". *Internet Engineering Task Force* 53. URL: http://tools.ietf.org/html/draft-jennings-senml-10. Accessed 14/02/2016.

Kent, S. (2002). "Model Driven Engineering". *Proceedings of the Third International Conference on Integrated Formal Methods*. IFM '02. London, UK, UK: Springer-Verlag, pp. 286–298. ISBN: 3-540-43703-7.

Le-Phuoc, D., H. Quoc, J. Parreira, and M. Hauswirth (2011). "The linked sensor middleware - connecting the real world and the semantic web". *Semantic Web Challenge 2011*.

National Instruments (2014). "Controller Area Network (CAN) Overview". *National Instruments Innovations Library*. URL: http://www.ni.com/white-paper/2732/en/#toc2. Accessed 07/09/2016.

OBD-II PIDs - Wikipedia. "On-board diagnostics Parameter IDs". URL: https://en.wikipedia.org/wiki/OBD-II_PIDs. Accessed 30/01/2017.

OBD Solutions (-a). "ECUsim 2000 OBD Simulator". URL: https://www.scantool.net/development-tools/obd-simulators/ecusim-2000/. Accessed 19/04/2016.

— (-b). "ECUsim 5100 Professional OBD-II ECU Simulator". URL: http://www.obdsol.com/solutions/development-tools/obd-simulators/. Accessed 20/07/2016.

OMG (2014). *MDA Guide rev. 2.0*. URL: http://www.omg.org/cgi-bin/doc?ormsc/14-06-01.pdf. Accessed 17/02/2016.

OSGi Alliance (2016). *Open Services Gateway Initiative*. URL: https://www.osgi.org//. Accessed 04/02/2016.

Ozen Elektronik (-). "Multiprotocol ECU Simulator - mOByDic4910". URL: https://www.ozenelektronik.com/multiple-protocol-obd-ecu-simulator-p.html. Accessed 20/07/2016.

Pefhany, S. (2000). *Modbus Protocol*. URL: http://www.interlog.com/~speff/usefulinfo/modbus_protocol.pdf. Accessed 01/02/2016.

Schafersman, S. D. (1997). "An introduction to science: Scientific thinking and the scientific method." URL: http://www.geo.sunysb.edu/esp/files/scientific-method.html. Accessed 08/01/2016.

Schmidt, D. C. (2006). "Model-Driven Engineering". Vol. 39. 2. URL: www.cs.wustl.edu/~schmidt/GEI.pdf. Accessed 18/02/2016.

Singh, D., G. Tripathi, and A. Jara (2014). "A survey of Internet-of-Things: Future vision, architecture, challenges and services". *Internet of Things (WF-IoT), 2014 IEEE World Forum on*, pp. 287–292.

Snyder, B. (2008). "Service Oriented Integration With Apache ServiceMix". URL: http://pt.slideshare.net/bruce.snyder/serviceoriented-integration-with-apache-servicemix. Accessed 04/02/2016.

Soldatos, J., N. Kefalakis, M. Hauswirth, M. Serrano, J.-P. Calbimonte, M. Riahi, K. Aberer, P. P. Jayaraman, A. Zaslavsky, I. P. Žarko, L. Skorin-Kapov, and R. Herzog (2015). "Interoperability and Open-Source Solutions for the Internet of Things: International Workshop, FP7 OpenIoT Project, Held in Conjunction with SoftCOM 2014, Split, Croatia, September 18, 2014, Invited Papers". Ed. by I. Podnar Žarko, K. Pripužić, and M. Serrano. Springer International Publishing. Chap. OpenIoT: Open Source Internet-of-Things in the Cloud, pp. 13–25. ISBN: 978-3-319-16546-2. DOI: 10.1007/978-3-319-16546-2_3.

Tecsisa (2016). "Tecsisa - Tecnologia, Sistemas y Aplicaciones S.L." *Tecsisa*. URL: http://www.tecsisa.com/. Accessed 28/04/2016.

Ten-Hove, R. and P. Walker (2005). *Java Business Integration (JBI) 1.0*. URL: http://download.oracle.com/otndocs/jcp/jbi-1.0-fr-eval-oth-JSpec/. Accessed 14/02/2016.

Thomas, A. (2007). "Enterprise Service Bus: A Definition". *Gartner Group*. URL: http://i.i.cbsi.com/cnwk.1d/html/itp/burton_ESB.pdf. Accessed 14/02/2016.

UDG Alliance (2016). *Universal Device Gateway*. URL: http://www.devicegateway.com/. Accessed 03/02/2016.

Zaldivar, J., C. Calafate, J. Cano, and P. Manzoni (2011). "Providing accident detection in vehicular networks through OBD-II devices and Android-based smartphones". *Local Computer Networks (LCN), 2011 IEEE 36th Conference on*, pp. 813–819. DOI: 10.1109/LCN.2011.6115556.