Agnostic Application Development Middleware for VANETs

Rui Miguel Correia Costa

Dissertation submitted to obtain the Master Degree in

Communications Networks Engineering

May 2013

Acknowledgements

I would like to thank the Academy... ok, now seriously. I would like to start by mentioning the incredible support that my advisor, Teresa Vazão gave me, not only during this thesis but since my freshman year in the University with valuable advices and guidance that ended up helping me achieving success on my academic journey.

Also there is a need to mention the important role that some professors and mentors posed during my studies, not only during university. One should recognize the courage and strength of will that they had by striving in every class to inspire students to be different, enthusiastic and have the excitement and *greed* to make a positive difference.

To all my friends, that during my life and specially my university years supported me not only with words but with funny, unforgettable and legendary moments that one would never forget. Due to the page limit imposed to this thesis and also to save some trees, and therefore preserving planet Earth, it is impossible to name and state here all the friends i keep in my heart. I am privileged to have met interesting people in so many different scenarios such as the academic life, judo, surf and skim, music and so many others situations that my life have made me experience so far. Also, i need to thank David Dias, Fábio Domingos, André Filipe Gonçalves, Ricardo Câncio Silva and Ana Rita Guilherme for dedicating time and patience to help me in the field-tests. To my friend Dario Marques, whose life was taken way to early.

Last but not least, to my family. To my parents for the endless support and comprehension throughout my life, always being there for me and always making every thing to support my decisions. To my uncles, aunts and cousins and to the ones that are not with us anymore, you all were mentors to me and helped me being who i am today.

Let's make this count, "Every accomplishment starts with the decision to try".

Lisbon, May 2013
Rui Costa

For my parents.
Everything I have and will accomplish in life is because of you.

Abstract

The development of vehicular communication technologies and networks allowed the fostering of Intelligent Transportation Systems (ITS). Such systems are heavily based in the creation of applications and services that will be designed to serve multiple purposes, either for delivering more pleasant driving experience to users and also for enhancing the safety and security of road users. This thesis proposes a platform for facilitating the agnostic development of vehicular-environment oriented applications by offering a common API for both safety and comfort-oriented applications. Our solution features application and service's flow control in order to give priority to most relevant applications in every moment, supporting scenarios where the need of safety application's priority is critical. A protocol for epidemically disseminating vehicle safety status is proposed in order to determine the current execution context. Results from field-tests performed in a controlled environment, shown a reduction in non-relevant traffic volume in emergency situations while enhancing the safety application's performance for critical situations.

Keywords

- Vehicular Ad-hoc Networks
- Intelligent Transportation Systems
- Vehicular Applications Development
- Agnostic Vehicular Applications
- API

Resumo

O desenvolvimento de tecnologias de comunicação para redes veiculares permitiu o desenvolvimento de Sistemas Inteligentes de Transporte (ITS). Tais sistemas são fortemente baseados na criação e uso de aplicações e serviços com diversos objetivos, tais como a criação de uma experiência de condução mais agradável e o aumento da segurança dos utilizadores da estrada, sejam eles condutores, passageiros ou peões. Esta tese propõe uma plataforma para facilitar o desenvolvimento agnóstico de aplicações para ambiente veiculares através da criação de uma API generalizada para aplicações tanto de segurança como de conforto. A nossa solução apresenta um controlo do fluxo das aplicações e serviços de forma a garantir a prioridade daqueles que sejam mais relevantes em qualquer momento, tendo em conta possíveis cenários em que as aplicações de segurança tenham um papel preponderante. É proposto um protocolo para disseminar de forma epidémica o estado de cada veículo que permite determinar o contexto no qual o veículo está no momento. Os resultados obtidos num cenário real, em ambiente controlado, mostram a redução significativa de tráfego não relevante em situações de emergência, garantindo ainda a redução do tempo de resposta dos serviços e aplicações de segurança, mostrando melhoramentos no desempenho de tais aplicações em situações críticas.

Palavras Chave

- Redes Ad-hoc Veiculares
- Sistemas Inteligentes de Transporte
- Desenvolvimento de Applications para Ambientes Veiculares
- Desenvolvimento Agnóstico de Aplicações
- API

Contents

List of Figures

List of Tables

Abbreviations

UMTS Universal Mobile Telecommunications System

WLAN Wireless Local Area Network

XML Extensible Markup Language

VANET Vehicular Ad-hoc Network

ITS Intelligent Transportation System

COMeSafety Communications for eSafety

NoW Network On Wheels

ETSI European Telecommunications Standards Institute

ISO International Organization for Standards

C2C-CC Car-to-Car Communications Consortium

VMS Variable Message Signs

RSU Road side unit

VANET Vehicular Ad-hoc network

V2V Vehicle-to-Vehicle

V2I Vehicle-to-Infrastructure

GSM Global System for Mobile Communications

GPRS General packet radio service

WiMAX Worldwide Interoperability for Microwave Access

LTE 3GPP Long Term Evolution

SeVeCom Secure Vehicular Communication

PReVENT PReVENTive and Active Safety Applications

ISO International Organization for Standardization

ETSI European Telecomunications Standards Institute

IETF Internet Engineering Task Force

simTD Safe Intelligent Mobility

MMWAVE Millimeter-wave

DSRC Dedicated Short-Range Communications

WAVE Wireless Access in the Vehicular Environment

API Application Programming Interface

ADM Application Development Middleware

ARM Application Request Manager

ARB Application Registry Bank

SOA Service-Oriented Architecture

EDA Event-Driven Architecture

LDM Local Dynamic Map

HMI Human-Machine Interface

CAN Controller Area Network

GNSS Global Navigation Satellite System

OBD On-Board Diagnostic System

GLS Grid Location Service

RLS Reactive Location Service

HLS Hierarchical Location Service

CBQ Class-based Queueing

ShSD Shortest Safety Distance

ROP Region of Plausibility

VADM Vehicular-Application Development Middleware

ESPD Epidemic Status Plausibility Dissemination

BSM J2735 Basic Safety Message

ASN.1 Abstract Syntax Notation One

DER Distinguished Encoding Rules

API	Application Programming Interface
VADM	Vehicular-Applications Development Middleware
ESPD	Epidemic Status Plausibility Dissemination
MANETs	Mobile Ad-hoc Networks
VANETs	Vehicular Ad-hoc Networks
ITS	Intelligent Transportation Systems
SeVeCom	Secure Vehicular Communication
ETSI	European Telecommunications Standards Institute (ETSI)
IETF	Internet Engineering Task Force
ISO	International Standards Organization
RSU	Roadside Unit
V2V	Vehicle-to-Vehicle
V2I	Vehicle-to-Infrastructure
WAVE	Wireless Access in Vehicular Environments
ROP	Region of Plausibility
BSM	Basic Safety Message
POI	Points of Interest

1 Introduction

Vehicular networks are a new class of networks that have emerged thanks to advances in wireless technologies and high investment from both automotive industry and research entities. Particularly, Vehicular Ad-hoc network (VANET)s are a subset of the general class of mobile ad-hoc networks, with several characteristics, presenting different challenges when developing both services and architectures for that type of network. VANETs are a area of interest of road operations, manufacturers, telecommunication operators and academia, that are investing a lot of resources researching on those networks.

Several wireless technologies, standards and architectures were created to cope with the specific requirements of vehicular networks, enabling the development and deployment of a wide variety of vehicular-specific applications and services[1]. The communication among vehicles and between those same vehicles and a roadside infrastructure have allowed the inception and development of Intelligent Transportation System (ITS).

ITS enable the creation of an enhanced driving experience, by delivering both applications whose goal is to deliver a more pleasant driving experience to users and applications that have the objective of enhancing the overall security and safety of road users. All groups of interest researching and investing in vehicular networks, have as priority the development and deployment of safety applications, due to the contribution that those can have in saving peoples lives and making driving more safe.

However, developing applications and services to be executed in vehicular networks pose several challenges. Applications need to be aware of both the architecture in which they are executing and also take into account the numerous situations that can happen in the various different scenarios where the vehicle can be at a point of time.

There is a large number of vehicular applications and services that can developed for ITS. This work will focus in contributing with a solution that allows a quick development of those applications, absenting the developers from some underlying implementation choices, while bearing in mind the importance and need of applications and services that can contribute to improve the safety involved in transportation for everyone.

1.1 Goals and Contributions

Taking into account all that was mention, the primary goal of this thesis is to design and develop a platform that facilitates the process of application development for vehicular-communication networks. By analyzing the available solutions and also the application's requirements, this thesis aims to develop a API that mitigates most of the hurdles in application development for this scenarios while keeping in mind the overall goal of using vehicular-networks to create safer and more secure roads and transportation environments.

Therefore the contributions of this work are:

1. Development of well-defined API for vehicular-networks oriented applications to be used widely and as open-source by the developers.
2. Creation of a unified integration between applications and the several available communication technologies and systems for vehicular networks.
3. Design and implementation of a architecture and platform that can adapt itself to cope with application's requirements while having in mind the need of granting priority on network access and usage for safety-related applications.
4. Extension of the available standard and widely-accepted solutions for vehicular-oriented application's development.

1.2 Existing Solutions

This thesis will be based on the several work developed by standardization entities and consortia between academia and industry that have addressed some grass-root challenges of vehicular networks and Intelligent Transportation Systems. From the various contributions, one should highlight and take into account the existence of a middleware solution that offers relevant functionalities for application's development, the **Facilities Layer**. This new approach, suggested by a standardized architecture for ITS, aims to facilitate the rapid development, standardized access to information, data and functionalities. Facilities Layer, despite coping with some the peculiar characteristics of vehicular networks and also allowing the rapid development of applications and services for such environments, some issues were yet to be solved:

1. In case of emergency or accident situation, no behavior change is performed, leaving that to be solved by applications.
2. Different accessibility to services for different types of applications

3. Lack of control over applications executing above Facilities Layer
4. No adaptation of underlying network and access layers in order to improve the system's performance in different situations

The thesis will address this issues and proposed a solution for solving them.

1.3 Thesis Structure

This document describes the research and work developed and it is organized as follows:

Chapter 1 presents the motivation, background on vehicular ad-hoc networks and the characteristics that differentiate them from traditional mobile ad-hoc networks. Also presents the proposed solution to solve the open issues in middleware architectures for such kind of networks.

Chapter 2 describes the previous work in the field by defining what is a Intelligent Transportation System, highlighting the most contributive projects a thorough description of reference architecture and technologies widely adopted for such systems.

Chapter 3 describes the system requirements and the architecture of the proposed solution.

Chapter 4 describes the implementation details and the technologies chosen in the proposed solution. To enhance the readability, all the pseudo-code is deliberately detailed in the Appendix section.

Chapter 5 describes the evaluation tests performed and the corresponding results.

Chapter 6 summarizes the work developed and future work.

Appendix section contains several additional information to the thesis writing such as, classes pseudo-code and definitions, detailed analysis tables and partial test-results.

2 Context and Related Work

This section will address the State of the Art technologies and architecture developed specifically for the creation and proliferation of Intelligent Transportation Systems (ITS). We will describe the several contributions made towards the enhancement of such systems and how that work served as a starting point to our work. We will start by stating the projects and initiatives that worked in both VANETs and ITS towards getting to the standardization entities that produces the widely adopted standards for vehicular-communications used nowadays. After that we will detail the types of components that take part on ITS and the communication technologies used to integrate this systems. We will then focus on explaining the new set of functionalities that the adopted ITS standard made available through the creation go the Facilities Layer. At the end of the chapter we will classify applications and services that can be executed in ITS and detail briefly the standards used for inter-application and inter-platform messaging.

2.1 Intelligent Transportation Systems

ITS had its inception and expansion based on the development of several communication technologies and protocols of vehicular networks, in particular of VANETS. As a subset of traditional Mobile Ad-hoc Networks (MANETs), Vehicular Ad-hoc Networks (VANETs) have specific properties that distinguish those from the traditional MANETs[2], for instance the deployment scenario the mobility pattern, properties of the communication nodes as well as some other challenging characteristics as the network scale, network topology variation or even the degree of connectivity [3]. But with the work performed in enhancing VANETs the development and further expansion of ITS architectures, technologies and protocols was finally possible [4].

ITS have as main goal the automation of interaction among vehicles and roadside infrastructure to create new degrees of safety, comfort and transportation efficiency in all kinds of roads. We will now address the several steps made so far towards creating the State of the Art ITS.

2.1.1 Related Projects, Consolidation and Standardisation

A vast number of research projects all over the world have been focusing on inter-vehicle communication systems and related problematics. Those projects, financed and supported by industry, governments and academia are currently developing solutions and services in order to establish new standards to VANET environments. Most of them are result of a joint-venture between auto-makers and mobile or roadway operators in order to enhance their equipments and infrastructures with means to deploy and support ITS.

Some projects have done important contributions to the collaborative elaboration of a standardized European ITS framework:

- **Network On Wheels (NoW)**[5][1] - Solve technical key questions on the communication protocols and data security for car-to-car communications to support active safety applications and infotainment applications.

- **WiSafeCar**[2] and **Safe Intelligent Mobility (simTD)**[3] - Consolidate V2X (either V2I or V2V) functions to supply several types of applications and services.

- **Secure Vehicular Communication (SeVeCom)**[4] - Full definition and implementation of security requirements for vehicular networks.

- **PReVENTive and Active Safety Applications (PReVENT)**[5] - Developing safety applications and technologies to mitigate accidents, through direct control on vehicles action .

In order to grant that ITS communication technologies and systems are disseminated throughout all the road infrastructure, standardization became a mandatory path to unveil. Several initiatives from both academia and industry started developing work in order to conglomerate technologies, applications, protocols and security and management mechanisms into one widely accepted architecture. A European project, named Communications for eSafety (COMeSafety)[6], was responsible for consolidating, gathering and fostering the several contributions from the several initiatives [6]. The Car-to-Car Communications Consortium (C2C-CC)[7] harmonized all the work made by COMeSafety for later proposal to relevant standardization bodies such as European Telecommunications Standards Institute (ETSI) (ETSI), Internet Engineering Task Force (IETF), International Standards Organization (ISO) and IEEE.

A important standard is nowadays considered as a base stone of the development and further expansion of ITS, the ETSI EN 302 666, ITS Standard Communications Architecture [7]. This standard

[1] http://www.network-on-wheels.de/
[2] http://www.wisafecar.com/
[3] http://www.simtd.org/
[4] www.sevecom.org
[5] www.prevent-ip.org/
[6] http://www.comesafety.org/
[7] www.car-to-car.org

made several innovations in several communication layers, also creating new layers that brought new features and functionalities to this communication system. Figure 2.1 depicts the layer representation of the ITS comunications architecture standard.

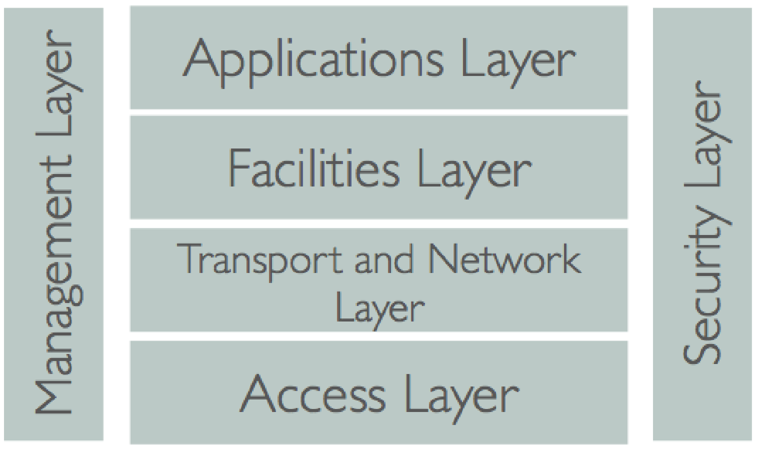


Figure 2.1: Layer representation of the ITS communications architecture standard

2.1.2 ITS Components

Four main components are needed to form a cooperative ITS: the vehicle station, the personal station, the roadside station and the central station.

Vehicle Station

A vehicle, or mobile node, is equipped with wireless communication capabilities for communication with other vehicles or roadside infrastructure. The on-board hardware can be connected to the vehicles built-in network to collect data within the vehicle, using the built-in sensors, and then send that information to the remaining ITS intervinients or process that same collected data to enhance the vehicles environment perception.

The OBUs in order to enhance communication capabilities, can be integrated with the access technologies such as Global System for Mobile Communications (GSM), General packet radio service (GPRS), Universal Mobile Telecommunications System (UMTS), Worldwide Interoperability for Microwave Access (WiMAX), 4G 3GPP Long Term Evolution (LTE), Wireless LAN IEEE 802.11 technologies and Infrared. Also, many vehicular communications specific access technologies such as DSRC/WAVE and CALM have been developed, and will be further analyzed in section 2.1.4.

Personal Station

Typically a mobile consumer device, such as a smart phone, PDA, tablet or laptop that can provide numerous ITS applications or represent the interface to the driver of the embedded vehicle' systems. This device is typically assigned to a person and can also store several profiles or preferences of that person, allowing a ITS system to be portable and independent of the vehicle in which the user is traveling. The Personal Station can communicate with the AU in a wired or wireless configuration, being that configuration dependent of the technologies present on such device. It can be integrated or connected, e.g. via Bluetooth, to a vehicle. It can also be considered as part of the vehicle equipment, and therefore generate or receive additional information data. Furthermore, the vehicle may directly use the communication capabilities of a connected personal device and may also make use of its output devices to present information within a vehicle to the driver. Also a personal station could be used as an independent component of an ITS, communicating with roadside infrastructure or with legacy systems, through cellular networks.

Road Side Unit

The Roadside Unit (RSU) or Roadside Station, is a fixed installation along the road, responsible for the communication with the Vehicle Station or other RSUs. This roadside equipment may be linked via a roadside gateway to road sensors or traffic control units, as traffic lights and Variable Message Signs (VMS) enabling the realization of in-vehicle applications that are aware of the road conditions as well as dynamically adapting and updating the road information based on the information transmitted by the vehicles. Optionally, a border router can connect the ITS Roadside Station to a backbone network, which can be an ITS Roadside infrastructure network or the Internet Domain. A RSU can also make use of existing or upcoming wireless infrastructures such as cellular networks, with UMTS, WiMAX and LTE [8].

Central Station

The Central Station may supply some value-added applications and services that may be accessed and used by the Roadside Unit (RSU)'s via a roadside gateway. This way some processing capabilities could be delegated only to the central station that also can store and process information from all the RSUs connected by the gateway.

2.1.3 Network architecture

Different communication scenarios can be envisaged for ITS and different technologies can be used to support the interactions between the components.

Communication scenarios

In a vehicular network several communication scenarios may be considered [9][10]. Vehicles can make use and communicate with wireless hot-spots along the road, either operated by Internet service providers or by an integrated operator. Legacy and existing cellular communication systems, can also be used. Vehicles can even communicate with other vehicles directly without a communication infrastructure, and therefore cooperatively forward information on behalf of each other. Hence, one may conclude two different communication scenarios in vehicular networks:

- **Vehicle-to-Vehicle (V2V):** Vehicles communicate among them, forming an ad-hoc network
- **Vehicle-to-Infrastructure (V2I):** Vehicles communicate directly with the roadside infrastructure to exchange information.

It is even possible to make an hybrid approach, where are explored both V2V and V2I communication scenarios.

A hybrid vehicular communication system tends to be the most adopted solution since it extends the range of V2I systems. Vehicles can communicate with the Roadside Stations, even when they are not within the wireless range, by using other vehicles as mobile routers, extending the transmission range of the vehicles enabling new services and applications to be developed and installed. However, a hybrid solution presents one disadvantage, since the network connectivity may not be guaranteed in scenarios with low vehicle density[3][11].

2.1.4 ITS Communication Technologies

In order to supply communication capabilities to vehicles and RSUs several technologies can be used. Several general-purpose and traditional communications technologies can be used, such as Cellular Networks as GSM, GPRS, UMTS and LTE. Also, new approaches on using technologies as Millimeter-wave (MMWAVE)[12] and [13] WiMAX have been done.

The most common approach is to use the traditional Wireless Local Area Network (WLAN) technologies such as the IEEE 802.11 a/b/g/n. These technologies are designed for high data rates and reliability and, with some modifications, they could be used in V2V direct communication but not for time-critical applications since it does not yield good results for frequent handover between different communicating nodes [14]. Also this technologies are based in Carrier Sense Multiple Access (CSMA), therefore not providing good support for parallel medium access.

Vehicular-communication oriented technologies have been a interesting research topic for the past years. Several technologies have been developed to cope with the peculiarities of vehicular-networks, in particular VANETs. Dedicated Short-Range Communications (DSRC) standards were developed to allow high-speed communications between vehicles and the roadside, or amoung vehicles, for ITS. DSRC systems are able to work in a peer-to-peer fashion (vehicle-to-vehicle) without any help from masters (e.g. RSUs). Some DSRC systems, commonly used for toll collection purposes, exist, operating in the 915 MHz unlicensed band with less than 30 meters range only provide a 0.5 Mbps data rate, not enough for ITS applications.

Therefore, IEEE developed and approved an amendment to IEEE 802.11 standard for DSRC systems, the IEEE 802.11p, with improvements of the IEEE 802.11a PHY and MAC layer and some additional changes to cope with the characteristics of a vehicular networking environment.

Most of vehicular communications are based on the frequent exchange of short status messages also known as beacons that carry information about the vehicle, such as its position, velocity and acceleration. In IEEE 802.11p, beacons are broadcast periodically by each vehicle. Communication range is normally in the order of several hundred meters and, therefore, beaconing provides awareness about the vehicles within the vicinity. One of the important characteristics of the IEEE 802.11p standard is the separate channel division for applications and services that have different objectives, such as safety or user comfort.

Using the IEEE 802.11p and also the IEEE 1609.x standards, a system architecture for the next-generation DSRC technology, which provides high-speed V2V and V2I data transmission was developed, the Wireless Access in Vehicular Environments (WAVE)[15].

Appendix 1 details the several characteristics of the available communication technologies, both traditional and vehicular-oriented.

Most of the wireless access technologies, such as DSRC and WAVE are characterized by their decentralized topology while cellular networks, for instance, UMTS or LTE, imply a centralized topology. Some frameworks, based on the LTE, have been developed [16][17], showing that LTE has better performances than decentralized solutions and also showing that as the number of vehicles increases, more efficient the framework becomes. The authors have also shown that in terms of overhead, bandwidth usage and packet loss LTE remains more efficient. Moreover, the velocity does not impact on the efficiency, whereas decentralized efficiency decreases when the vehicles velocity increases.

The authors in [18], have done a performance evaluation of the 802.11p/WAVE communication standard, concluding that the technology can not ensure time-critical message dissemination. Some previous work, on how to solve this problem has been proposed by [19], that suggest the use of a relevance-based, altruistic communication scheme that optimizes the application benefit and the bandwidth usage.

A survey on the performance performance of the WAVE and 802.11p standards for vehicular communications have be done in [20]. The authors studied both the overall capacity and delay performance of vehicular networks utilizing those standards. The results showed that the traffic prioritization schemes for the used standards work well, and even in the presence of multi-channel operation implemented by the IEEE 1609.4 the delay of control messages of highest priority remains on the order of tens of milliseconds. Only when the total offered traffic within the radio range approached 1000 packets per second the delay values became excessive.

The authors suggested that, despite WAVE and 802.11p appear to perform well for vehicular networks, some work should be done in higher layers to guarantee the overall network load remains under control. This statement gave us a first direction towards solving the problems that we propose to address in this thesis.

2.1.5 ITS Facilities Layer

In order to provide functionality for rapid application development and also to ensure that each application running on the same station is using the same data and information a middleware was created, the facilities layer. Facilities layer will provide standardized access do information, data and common functionalities. The facilities layer is integrated between the ITS applications layer and the network and transport layer.

To avoid having different data retrieved to different applications, the facilities layer offers consolidated, up-to-date, and consistent information, for example, for position, time, speed, and acceleration.

This layer will be responsible for offering an abstraction between the developed applications and vehicle's brand and model. Facilities layer can obtain vehicle's information through a Controller Area Network (CAN) bus or through the processing and analysis of information retrieved from networking and transport layer.

The information is encapsulated and stored in the facilities and can be accessed by applications in a standardized way. In order to provide consolidated information about the environment of an ITS station, a **Local Dynamic Map (LDM)** is used. The Application Development Middleware (ADM) provides data models to represent both **dynamic information** , e.g. the geometry of an intersection or landmarks for referencing, and **static information** ,e.g. the status of a traffic light or the position of a vehicle, and according to standards is mandatory for all vehicle stations. LDM can be constructed based on messages broadcasted regularly from every vehicle, containing information about position, speed, direction, etc. also known as *beacons* [6].

In order to assure LDM correct functionality, position messages need to be time stamped with a global clock, which could be provided through a Global Navigation Satellite System (GNSS). If there are time discrepancies between the vehicle's position messages due to different local clocks, some erroneous behavior can happen, affecting applications that depend on accurate position in order to function correctly. Such as been described in more detail by the authors in chapter 2 of [21].

The system is developed continuously and can have different configurations as well as different applications being possible to update or install applications into the station. That process must be performed without producing an unstable system or causing any risks for the driver, by affecting running applications.

Adding to the already mentioned characteristics of the facilities layer, the ETSI standard [7] responsible for the proposed standardization architecture - depicted in figure 2.2 - refers the general application, services and communications functionalities that the facilities layer should offer:

- **Application Facilities** - LDM suport, generic Human-Machine Interface (HMI) support , support for data presentation, support ITS application's maintenance,Service-Oriented Architecture (SOA) application support and position and time support.
- **Information Facilities** - Relevance checking, location referencing and time stamping of data, station capabilities management and combining, fusing and provisioning of updated data from different sources.
- **Communication Facilities** - Addressing support, common message management for data exchange between ITS applications, repetitive transmission of messages support and DSRC legacy applications support.

By having a facilities layer, the ITS provides transparent APIs containing frequently used information and functionality while it accelerates the development of applications by also providing the desired information quality, independent of different station types.

2.1.6 ITS Management and Security Layers

The standardized ITS communication architecture present two additional layers to extend the and facilitate applications and service's management and security. The ITS Management layer offers functionalities such as:

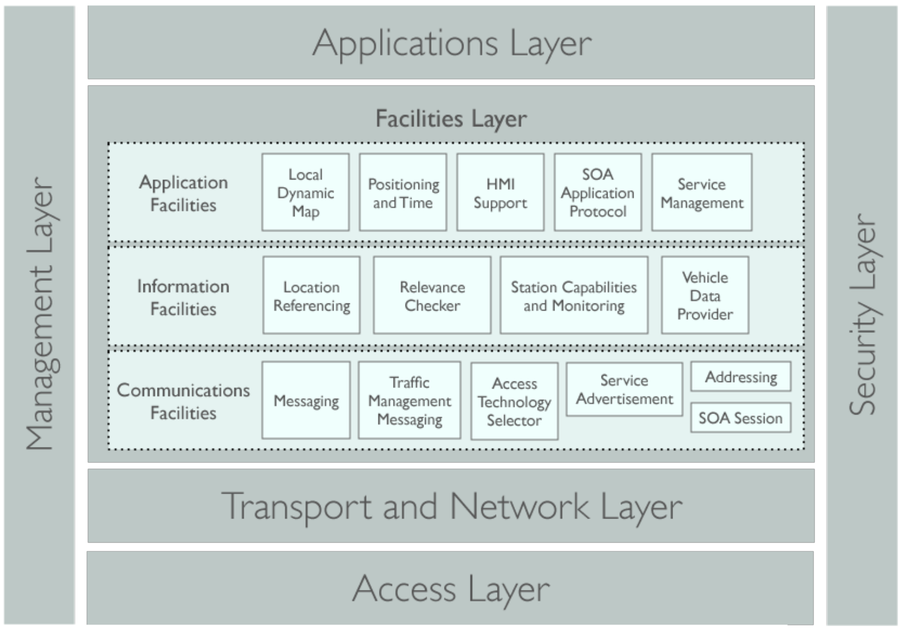


Figure 2.2: Facilities functionalities of the ITS reference architecture

- Management of policies and profile settings of the several layers of the standard.
- Management access to the communication technologies based on criteria such as application's policies and requirements.
- Manage software updates, data logging and system diagnosis.
- Interact with the security layer for security and privacy functions.

The standard addresses as one of the main challenges of the management layer the need of combination of requirements of both safety applications and other application types.

The ITS Security layer is responsible for coordinating the use of secure generic message format in order to grant a security degree on all the layers of the ITS communication architecture. Also the Security Layer is responsible for any other interactions and management tasks related to setting up, key generation and configuration of the overall security system of the platform.

2.1.7 ITS Applications and Services

Within ITS, several applications and services can be developed. Traditional applications may be used, but generally vehicular oriented applications tend to be the option taken by developers working with ITS applications and services, therefore, one will only consider the classification and characterization of vehicular-oriented applications and services.

Applications can be classified in several ways, and many authors have done different approaches on that subject. While in [10] the authors divide applications in safety applications and comfort applications, [22] divides them in safety and user-related applications. Both of the approaches then subdivide applications in several categories taking into account the objective and priority of each application, as well as the technical requirements like, for instance the need of Internet connectivity, presence of OBUs and others.

As far as vehicular applications are concerned, several addressing schemes are considered in wireless ad-hoc networks, and applications are directly correlated with this addressing schemes [13]. Some approaches have been done using topological and attribute based schemes [2], but most of the developed work uses the following schemes:

- **Fixed Addressing:** IP-Based location where each node has a fixed address assigned to it as soon as it joins the network. This is the most common addressing scheme used in the Internet, being mobile IP an exception.
- **Geographical Addressing:** Location-Based addressing, where each node is characterized by its geographical position, changing its address while it moves. Also, the direction of the movement, road identifiers, type of vehicles and other characteristics may be used in this addressing scheme.

Some authors also classify and divide applications in Safety-Related in Transportation Safety and Transportation Efficiency. On this work services and applications will be classified as **Safety-Related Applications and Services** and **User Comfort Applications and Services** that will now be described. A summary and discussion on the application requirements and characteristics will also be presented.

Safety-Related Applications and Services

Safety applications are intended to prevent accidents by warning the drivers about traffic or road conditions. The driver can be assisted, while driving, to prevent or avoid accidents, and many studies shown that a large percentage of those could be avoided if the driver is provided with some kind of warning, half a second before the collision. Safety applications and services can be divided into **information**, **assistance** and **warning** categories [23]:

- **Information:** Applications that provide some kind of information to the driver, such as, road conditions and weather information, traffic lights scheduling and speed limits. Also **traffic coordination and monitoring** can be achieved, were the traffic can be analyzed and vehicles can be rerouted in order to prevent roads congestion.
- **Assistance:** Applications that provide navigation assistance information to the driver. Some examples are **navigation and local information**, that present information about local points of interest and travel information, **cooperative collision avoidance**, that allows vehicles to interact with each others and cooperatively avoid accidents, for instance, when changing lane, by detecting nearby vehicles.
- **Warning:** Applications that provide collisions notification or support for emergency management purposes. The notifications can either be used for traffic efficiency, where drivers are warned to avoid the accident, suggesting alternative routes, or for emergency management,

where emergency procedures, such as, calling an ambulance or road assistance, can be triggered.

User-Comfort Applications and Services

The main objective of comfort applications is to give user a more pleasant travel. Nowadays, people are used to alway be communicating, either with people or accessing to services available in the Internet. As stated in [23], all kinds of applications that may run on top of TCP/IP stack may be applied in this category. User-Comfort applications and services can be divided into **information**, **entertainment** and **business** categories:

- **Information:** Applications that provide information about points of interest such as restaurants, museums and other local information. This type of application demands some type of data archiving and information gathering. Information can also be gathered from vehicles in order to collect data for a number of applications, such as transportation planning, safety analysis or research.
- **Entertainment:** Mainly Internet-based applications such as audio and video streaming, web browsing and E-mail access. Voice and Instant Messaging applications may also be developed, allowing users to communicate either with other vehicles or with other users on the Web, making use of their smartphones or vehicle-embedded interfaces.
- **Business:** Applications that provide support for e-commerce and commercial information like advertises or electronic payments and pricing. Also, companies can make use of vehicular communications in order to track their commercial vehicles and improve the efficiency of freight terminal processes and drayage operation, as well as the exchange of inspection data with regulating agencies and also improve fleet security.

Summary and Discussion

In order to deploy the aforementioned applications and services in a VANETs, several requirements have to be fulfilled. Providing safety applications are the major concern when taking into account the network architecture, since they consist mainly of time-critical applications. Also providing delay sensitive services such as multimedia data transfer or instant messaging, must be taken into account. Therefore a list of requirements, arises and the following will be compared with safety and user-comfort applications in table 2.1

Table 2.1: Safety and User-Comfort Applications Requirements

Application Type	Interaction Type	Time Critical	Addressing Scheme	QoS	Bootstrapping
Safety	V2V , V2I	Yes	Location-Based	Yes	Dependent
User-Comfort	V2V, V2I , X 2 Legacy	No	IP-Based	Optional	Not Dependent

All ITS Applications and Services are heavily dependent on what communication scheme was adopted, the chosen addressing mode, system penetration and real-time requirements. Most of the projects mentioned in 2.1.1 are developing their own applications and services while taking into the account the network architecture decisions and the requirements of the given applications. Some research is being done in specific aspects of vehicular applications and services, as in, [24, 25], where the authors propose solutions to predict drivers intents and destinations fetching the possibility of creating traffic efficiency enhancer services. In [26] the authors proposed a publish/subscriber communication infrastructure for efficient information dissemination along RSUs. Wireless technologies options have been related with specific applications in the work developed by *Dat et.al.* in [12]. Also, a survey and solution proposal on streaming media distribution along VANETs have been done in [27].

Security concerns within vehicular applications arise, since several new attacks can be done, and the usage of those attacks can have significant impact on peoples lives. One type of attack has been presented in [28], where the author presents why traditional security mechanisms do not work in vehicular networks. In [9] it is proposed a framework designed for vehicular environments that cope with some security issues of this specific network type. In general, most of the security designs increase the amount of requirements of each application. Despite the fact that this work will not address security, having the requirements and developments of such subject in mind is important to define the proposed architecture.

2.1.8 Standards for Vehicular-Network Application's Messaging

On the previous sections we described the several intervinients of a vehicular network, the standards and technologies involved. All this technologies and developments aim to serve applications and services that offer a number of functionalities to the end-users of this systems either for their comfort or their safety.

There is no defined structure for the applications in this environments due to the notion that there is a sub-layer that offer services to the applications, that then can use one or more service or form a hierarchy with other applications. In order to guarantee a certain degree of compatibility with both the below-layer functionalities, inter-vehicle communications and also other applications, the need for defining a inter-application messaging standard arose.

The SAE J2735 (SAE J2735 2009) [29] standard defines the messages format and structure to be used in DSRC. A set of *Message Types* is defined and its called the *Message Set*. Each *Message Type* is defined by:

1. **Data Elements -** The most basic structure type, existing approximately 150 standard Data Elements.

2. **Data Frame -** Constituted by one or more Data Elements or other Data Frames, existed approximately 70 standardized Data Frames.

The Message Sets, Data Elements and Data Frames are defined in the formal language *Abstract Syntax Notation One (ASN.1)* [30]. The over-the-air translation into bits and bytes is done by using the *Distinguished Encoding Rules (DER)*, that encodes each data item intro a *Tag*, *Length* and *Value* structure.

Table 2.2: SAE J2735 standard message sets and carried information

Message Set	Purpose	Information Carried
A La Carte Message	Generic message with flexible content	General data chosen by applications
Basic Safety Message	Carry vehicle-state information to support V2V safety-related applications	Coordinates, Current Speed and Transmission, Brake System Status, Vehicle Size
Common Safety Request	Request specific state information from other vehicles	Same as *Basic Safety Message*
Emergency Vehicle Alert Message	Alert drivers about the proximity of an emergency vehicle	Emergency Type, Vehicle Mass and Type
Intersection Collision Avoidance	Vehicle location information relative to specific intersection	Intersection ID, Lane Number, Event Flags
Map Data	RSU informs about geographic description of an intersection.	Roadway and intersection data, Road and Curve segments information
Traveler Information	RSU informs about advisory and road sign types of information to passing vehicles	Lane width, Common Directionality, Speed Limit, Generic Sign

Several types of *Message Sets* are available, suiting different purposes. Table 2.2 depicts some of the available message sets described in the J2735 standard and also the most relevant information transported in each one of them.

2.2 Chapter Summary

Several design decisions are to be made when defining a ITS vehicular network architecture. Although some standardization is available, it is not specified the technologies and equipment to be used, leaving that aspect to be decided by the projects teams. Design decisions have to be done always considering the predefined requirements for the ITS that is being developed, the application requirements and degree of market penetration needed. Most of the aforementioned technologies and applications have been developed having in mind the VANETs characteristics, however the integration and deployment in a real scenario has been shown a challenge to both operators, developers and academia.

Most of the tests and results presented previously are developed for a big scale networks with high degree of ITS penetration. Therefore it is necessary the development of a solution that allows a incremental deployment of ITS applications and technologies. This implies that both the framework and applications are the most agnostic possible from the underlying implementation.

3 Architecture

The following chapter explains the architecture design developed to cope with the objectives of this work. The main goal is to facilitate the development of vehicular applications by creating an abstraction layer to be used by the developers which provides a set of services to be used. That abstraction layer also controls the applications used, being able to adapt application's performance and functioning according to the context in which applications are being used.

3.1 Requirements

The objective for this work is to facilitate the fast development of applications to be deployed in vehicular environments. Also, the work presented in [31] stated a set of desirable architectural features for vehicular-application systems. The goals include:

- **Future-proof -** The system must be both backward compatible as well as future-proof regarding newly divided or upgraded applications
- **Flexibility -** The system must be flexible in order to cope with the heterogeneous marketplace of vehicles running different subsets of applications
- **Extensible -** The system must offer extendibility for non-standardized applications and data elements (e.g. proprietary to one or more manufacturers)
- **Unified Interface -** The system must offer policy and self-policing between various applications within a single vehicle to be managed in a single entity. Also support authentication and other security primitives managed across all applications by the same single entity.

- **Layered Architecture -** The system must be developed by following a layered architecture that abstracts message sending and all lower-layer details from the application development and design
- **Low bandwidth usage -** The system must be designed in order to use the less bandwidth possible
- **Information Rate -** The system must support different data rates from different applications
- **Recognize vehicle capabilities -** The system must cope with different amounts of information that different vehicles may supply
- **Enable product differentiation -** The system must support applications and service's differentiation from different vendors and developers.

Bearing this characteristics in mind, the authors based most of the decisions in order to cope with the above while creating a dynamic, yet agnostic platform for vehicular-applications development. We also defined a set of basic-step requirements, that guided the work presented in this thesis:

1. Low cost platform;
2. Application Programming Interface (API) to facilitate the development of vehicular applications with common functionalities;
3. context aware configurable platform.

To cope with a low-cost and modular platform, the authors developed a general architecture for both roadside units and vehicles, that can be used with of-the-self embedded systems and communication technologies. Also, all the developed work makes use of open source technologies and code.

The developed API, extends and uses the ITS communication standard, through combination of a set of functionalities of the Facilities Layer. Also, in order to grant compatibility with other solutions for vehicular environments, the proposed solution makes use of the SAE J2735 messaging standard[29]. This standard defines message's format and structure to be used in the IEEE 802.11p wireless vehicular-communication standard.

The developed architecture has the possibility to adapt itself to the scenario in which the vehicle is at the moment, and therefore adapt the applications and service's flow to give priority to the most relevant in such situations. The developed platform has an internal state that mirrors the above mentioned scenarios, representing the vehicle status at a given time. We considered three (3) possible states:

- **Normal -** Vehicle is traveling normally;
- **Emergency -** Vehicle declared an emergency, for instance, caused by a flat tyre, engine damage, emergency driving, etc.;
- **Accident -** Vehicle detected an accident or eminent accident;

Platform status mirrors both the vehicle situation as well as a local region situation, since one vehicle if it is in a Accident or Emergency situation ends up affecting all the circulating vehicles around. Therefore, status can be altered by the VADM through some based on data analysis, by request from nearby vehicles or by human interaction.

Having all the above in mind, our driving-thought along the way was that the final architecture must contemplate the easy development of applications, allowing programmers to select easily what kind of information they need in a common and easy to use API, abstracting them from the underlying technologies and specifications.

3.2 Proposed Solution Description

The authors propose in this work an architecture that makes use of the expected functionalities of the ITS Facilities Layer in order to facilitate the rapid development of applications to be used in vehicular-networks environments. A new interaction mechanism between modules of the Facilities Layer and applications is proposed in order to get efficiency improvements.

A new technique for application's run-time relevance checking is also proposed. The platform has the possibility to adapt itself and the deployed applications to the current execution context, taking into account the possible external events that may or may not affect applications relevance in such situations.

The proposed solution blends functionalities of the Management Layer within Facilities Layer modules in order to create a self-adaptable yet modular platform, that analyses the execution environment and makes changes in order to be more efficient and offer adaptable capabilities to applications. This work assumes the existence of a support platform that addresses and offers functionalities of the Network, Transport and Physical Layer, and also connectivity to sensing capabilities of the node, whose scope stands out of the present work. The functionalities required and used from a underlying support platform are further explained in section 3.2.1.

Taking into account the desired properties of the Facilities Layer for a standard ITS platform, as stated in [7] and already mentioned in section 2.1.5 some functionalities were implemented while others, whose scope stood out of the goal of this work, were considered supplementary services to be offered by a support platform. Table 3.1 depicts the functionalities addressed by our architecture and the functionalities that are assumed to be part of a support platform.

3.2.1 Support platform integration and functionalities

By following a layered paradigm of a standardized communication development, the goal of this work assumes only developments made at the level of the Facilities and Application Layers. Therefore, a support platform, that addresses functionalities of the remaining layers is needed. Such functionalities are now enlisted and briefly described:

- **Routing -** Implementation of vehicular networks-oriented routing protocols to forward information among all the ITS components. Routing protocols can be of several types, such as Topology

Table 3.1: ITS Facilities Layer features addressed in the proposed solution and offered by the support platform.

Property	Support Platform	Our Work
Addressing support	X	-
Position and time support	X	-
Location referencing and time stamping of data	X	-
Station capabilities management	X	-
Common message management for data exchange between ITS applications	X	-
Repetitive transmission of messages support	X	-
LDM suport	X	-
Support for maintenance of ITS applications	-	X
SOA application support	-	X
Relevance checking	-	X
Combining and fusing updated data from different sources	-	X
Station data provision	-	X
DSRC legacy applications support	-	X
Generic HMI support	-	X
Support for data presentation	-	X

based, Position based, Cluster based, Geocast based and Broadcast based. No specific type of routing is required on the proposed solution.

- **Communication Technologies -** Access and management of communications technologies, either traditional or vehicular network-oriented.
- **Location -** Implementation of location protocols or mechanisms to determine and share vehicle's position. Such mechanisms can be based on information sharing between the participants of the network or based in GPS device, whose management and access must also be available. Examples of location services, based in geographic routing protocols, are the Grid Location Service (GLS) [32], Reactive Location Service (RLS) [33] and the Hierarchical Location Service (HLS) [34].
- **In-vehicle sensing -** Access to vehicle sensors and information in order to get more accurate information about its status, speed, and detect possible malfunctions. In order to get that information from the vehicle a On-Board Diagnostic System (OBD) interface can be used.
- **Beaconing -** Beaconing system to broadcast or geocast several vehicle's information such as the unique ID, number of propagation hops, position and other management data.
- **Management Information -** Module that aggregates information from vehicle's sensors and data as well as information retrieved from the beacons received from the surrounding intervinients of the vehicular network.

Although not all functionalities are used and required for the proposed work feasibility, the availability of the above can enhance and extend this solution efficiency and performance.

3.3 System Architecture Description

Taking into account the already defined requirements, the authors developed an architecture that aims to be modular to facilitate both implementation and update or the several internal modules, opening the possibility to extend its functionalities if needed. Also, the proposed architecture is designed to be used in both vehicle stations and RSUs.

The **Vehicular-Application Development Middleware (VADM)**, is the core module of the proposed solution. It gathers and supplies, from the underlying modules, parsed data to applications, taking into account applications characteristics and requirements.

VADM has the following functionalities:

1. Coordination of all the interactions with the underlying layers and architectures.
2. Coordination of application's flow.
3. Coordination of the inner-modules flow and functioning.
4. Getting information about the surrounding execution scenario from the underlying modules of the support platform.
5. If supported, Vehicular-Applications Development Middleware (VADM) may also act directly on that support platform affecting and adjusting its behavior, e.g. adjust beaconing frequency, packet size, etc.

VADM coordination behavior takes into account the context in which the platform is being executed and also external events, for instance, accidents, speed increase, emergency situation, vehicle state, among others. The way the proposed platform interacts with both applications and support platform, as well an overview of the overall functioning of our architecture will now be briefly presented.

3.3.1 Architecture Interactions and Description

Applications interaction with the **VADM** is based on a Publish-Subscribe messaging pattern [1].

Applications interact with the platform, in a decoupled fashion, by sending information through API primitive, and retrieving all the information, from the own platform or from other vehicles or nodes using the Publish/Subscribe messaging pattern. Applications can subscribe to services installed in the platform or deploy, run and subscribe their specific services. Those services can be developed by the composition of other services or simple updates to the existing ones. Applications are default listeners of a service that publishes all the information received from other nodes of the network using the same application. Therefore it is assured the communication of the same application in different nodes, while keeping the decoupled event-driven paradigm of the indirection between applications and the platform.

[1] A Publish-Subscribe pattern is a model where some entities, called senders, can publish information that will only be viewed by entities, called subscribers, that have chosen to receive that specific information. Information is gathered in topics, from which the subscribers choose what ones to subscribe

Application's flow and adaptation is coordinated by the VADM that, by taking into account the current environment status, will make changes and give priority to both services and applications whose execution should be triggered or shutdown in such situations.

There are two levels of priorities within the platform, one for the **applications** and other for the **services**. Priorities are defined based in both applications and services classes and types, according to their functionality. Such classification and priority analysis is thoroughly detailed in section 3.6.

VADM retrieves the available parsed information from the support platform. Depending on the support platform and also on the node type, that information can be for instance the current Road ID, Direction of Movement, Vehicle Type, Current Location and Speed and Neighbors information. Note that some information is exclusive of vehicles, while other may be part of a RSU node.

Besides taking into account the aforementioned information in order to infer possible external events and execution environment, VADM can make use of sensors that might be available. Those sensors can be directly connected to the platform, or embedded on the Vehicle/RSU and accessed by CAN bus or any OBD interface (if node is a vehicle).

VADM will also interact with the support platform in order to access the several communication means available, specifying also the messaging type to be used (Broadcast, Geocast or Multicast) as well as other possible characteristics, such as number of hops and radius of message propagation. Alternative communication methods can be used as for instance cellular networks, LTE or WIMAX. By having all this communication options, VADM is able to choose which ones to use in the several different situations.

By having only two abstract dependencies of the underlying platform, the **Communications Pipe** and **Informations Pipe**, the proposed architecture can be widely used in any platforms as long as those offer access to communication and parsed information facilities.

The proposed solution, besides delivering an easy to use application development environment for vehicular applications, acts as a coordinator entity for scheduling and adapting applications-flow within vehicular environments, depending on the context in which the platform is being executed.

This thesis proposed architecture is inspired in several distributed computing concepts applied to a single architecture in order to cope with requirements such as the creation of a modular system whose parts can be decoupled and redesigned without compromising the rest of the parts of the same architecture. Therefore, the proposed solution mirrors some resemblance to concepts as SOA and Event-Driven Architecture (EDA), already mentioned by the authors in [35], as some of the architecture building blocks for vehicular networks architectures.

Figure 3.1 depicts a diagram of the proposed solution and its integration in the standardized ITS platform.

3.4 Vehicular-Application Development Middleware

The VADM, is the core module of the proposed solution. Therein lies the main abstraction layer of the solution that offers a set of API primitives that can be used by applications regardless of their

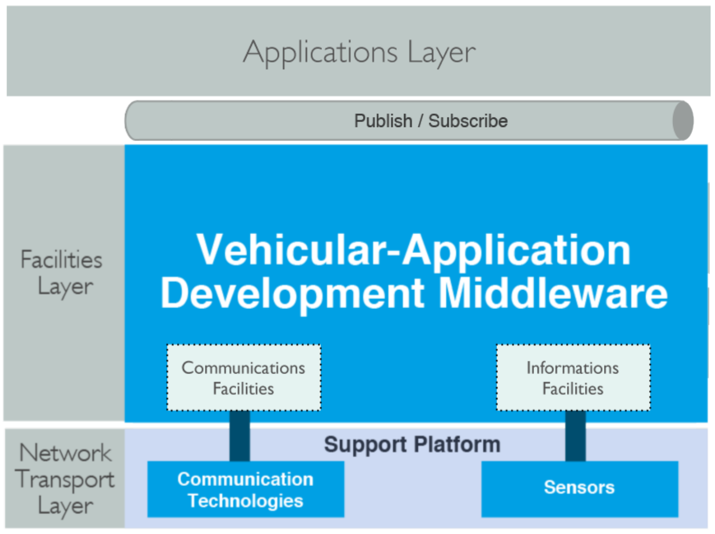


Figure 3.1: Proposed solution architecture diagram and integration within the standardized ITS platform

types or requirements:

1. **Register Application -** Register the application within the platform in order to use its functionalities.
2. **Retrieve Node Information -** Get information regarding the current node characteristics, such as node type (Vehicle/RSU), license plate, current IP, platform version.
3. **Request Service List -** Get list of available services currently active in the platform.
4. **Subscribe to Platform Services -** Subscribe to the services available at the platform that will be needed by the application.
5. **Deploy and Run Application-Service -** Deploy and run a service specific of the application. This service can be a new implementation or a composition of other services.
6. **Publish Information -** Publish information and data to the platform that will send it through the available communication resources using TCP or UDP transport protocols with the available communication technologies.
7. **Unregister Application -** Unregister application from the platform, remove all application deployed services and unsubscribe to all subscribed platform services.

A set of decoupled modules was developed in order to offer several functionalities within the VADM platform to applications. Modules can be updated separately in order to enhance their functionality, while keeping the overall functionality of the VADM. Bellow follows a brief description of each module:

- **Application Request Module (ARM)-** Management of all the underlying modules by orchestrating interactions among them and also acting as a coordinator of the overall inner-VADM architecture. All communication is made through the ARM that contains a scheduler to define the relative priority of each message being sent based on the application type and the current state of the platform. Case an application does not have priority to access the platform, the scheduler blocks the request. Several policies can be configured, e.g. discard or queuing of the request.

- **Service -** Several system services were identified as common requirements for applications that execute in vehicular environments and therefore are made available as default. Services such as retrieving the current coordinates, speed or any other available vehicle information (*GetNode-Information*), as well as getting the current platform status (*Normal*, *Emergency* or *Accident*) (*GetPlatformStatus*) or getting the current type of node (RSU or vehicle) (*GetNodeType*). Also there is a service that provides the current neighbors list and retrieves information of those, such as the address or last known position (*GetNeighboursInfo*). Some system support services, that can only be accessed by the platform and its protocols, were also developed. With this already available services it is possible to grant basic functionalities for applications, that then can compose several services or create and install their owns in order to cope with the application's needs.

- **Application Registry Bank (ARB)-** Storage of registered application's information.

- **System Analyzer -** Alter the current execution state of the VADM platform (*Normal*, *Emergency*, *Accident*). VADM platform state can be altered in three (3) different fashions: through analytical methods[36], that determine the need of altering the state based on data analysis gathered from several sources such as in-vehicle sensors, navigation data, modules malfunctioning or any other relevant information. Also the state of the platform can be altered through user-input within a human-machine interface. Besides the aforementioned methods for changing the platform state, state can be also altered by proximity to vehicles in accident situation using the protocol described in 3.5.3, the Epidemic Status Plausibility Dissemination (ESPD).

In order to maximize compatibility to whatever support platform or architecture supplies the communications technologies and sensors, while keeping the platform overall functionality, only two modules have access to the underlying facilities support platform:

- **Call Manager -** Storage of parsed and updated information retrieved from several sources such as sensors, navigation and data analysis.

- **Communication Manager -** Establish connection with all the available communication facilities to send and receive information and act as scheduler between several of the communication technologies. In case several communication technologies are available, Communication Manager can use its scheduler to decide upon sending or retrieving information from one or several communication technologies based on several possible policies. One example of a policy, is in case of accident situation, information can be send through both Wi-Fi and cellular technologies.

Figure 3.2 depicts the components of the VADM and the following sections will thoroughly detail each module architecture and functionality.

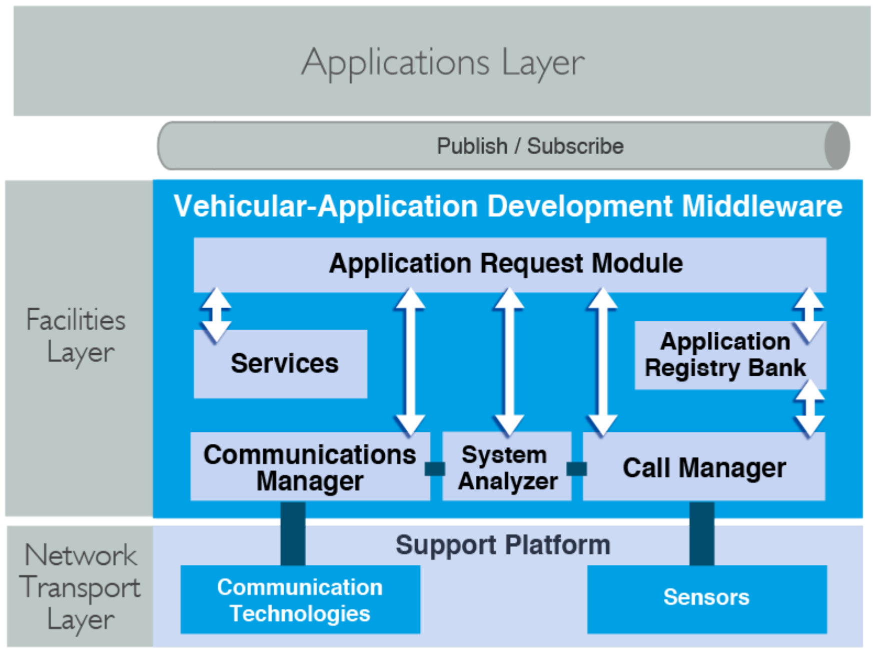


Figure 3.2: Vehicular-Application Development Middleware module components

3.4.1 Application Request Manager

The Application Request Manager (ARM) is the module responsible for coordinating all the remaining modules of the VADM. Every time the platform is launched, the ARM will start and setup all the VADM modules and impose default configurations, loaded from a configuration file. Such configurations include for instance:

- **Services Publishing Frequency -** Frequency in which services publish information to the Publish-Subscribe system to be later retrieved by applications that subscribed to those services.
- **CallManager Retrieving Frequency -** Frequency of getting parsed data and information from the Information Facilities functions of the support platform.

or the frequency in which services publish information to the Publish-Subscribe system.

In case of a VADM status change, ARM will impose some adjustments in the remaining modules functionality. ARM will be responsible for tweaking each module in order to adjust the overall platform performance to the current execution situation, ARM acting as a coordinator inside the VADM.

ARM module is also responsible for applying the policies and priorities to all data sent by applications. A scheduler is triggered every time a application wants to send data in order to determine if such information is relevant according to the current platform state and other applications also sending information, that might have higher priorities.

3.4.2 Call Manager

Call Manager module is responsible for gathering all the information about the context in which the platform is being executed. It retrieves information from the node where the platform is installed, from the current and past neighbors or participants of the network and all the possible information that can be obtained either by data collection and sharing and by sensors reading.

Since the presence of vehicle or RSU sensors is not mandatory, the Call Manager has the possibility of disabling and enabling the functionality of sensors reading. The module manages itself in order to perceive which sources of information are available and then parse that data to be put available to services and applications through a ARM request.

Call Manager will gather data with a frequency specified and controlled by the ARM, and loaded from platform's configuration file.

Since the Call Manager is an isolated module inside of the ARM, if one needs to change the underlying support platform or sensors technology, only the Call Manager module needs to be changed, leaving the rest of the features and functionalities intact, not rising the need of any change within the VADM nor in the remaining modules.

Data retrieved by Call Manager module can be accessed only by ARM and System Analyzer. The later will lookup into the retrieved data and information to infer system decisions, for example, a status change. More details on how System Analyzer uses that information are described in 3.5.

3.4.3 Communication Manager

Communication Manager is the module responsible for creating and managing all the connections with the communication technologies supplied by the underlying support platform. This module is responsible for all the sending of data as well as management of all the data received. In order to guarantee the full decoupling of applications and remaining modules of the proposed solution, when data is received on any of the communication technologies in use, the communication manager, publishes that information to a default class of the Publish-Subscribe pattern, listened by default in all the applications.

When applications what to send data, VADM publish API primitive is invoked and the module retransmits that data to the Communication Manager through the ARM. Applications may define the messaging scheme to use (*Broadcast*, *Geocast* and *Unicast*) and the communications manager will send the message accordingly, through the support platform, using the available communications technologies.

Information generated by safety-related applications are encoded with the J2375 standard, and despite the fact that Communication Manager module receives the encoded messages and only needs to relay them to the proper communication technologies of the support platform, a J2375 decoder is also available in order to interpret the received safety messages and deliver those to the proper destinations within the VADM platform.

User-comfort application's messages are sent through traditional Internet Protocols, with no encoding.

Case several communication technologies are available to be used by the platform, communication manager can opt to send information through one or more of those technologies.

Communication Manager is responsible prioritizing messages to be sent, because depending on the VADM platform status those priorities might change. Those priorities will be used by the Communications Facilities of the underlying support platform to help in message re-ordering, proper communication channel choose and other decisions that can be done based on a message's priority.

Also, Communication Manager has a scheduler responsible for applying the policies and actions, dependent of certain situations and platform status, to applications. Those policies are further described in section 3.7.1.

The Communication Manager, by allowing the installation of different implementations to connect and use several communication technologies, allows a full abstraction and independence of the platform in which the proposed solution is being used, while at the same time assures the possibility of using several resources to expand the guarantees of delivery and diversity of information exchange.

3.4.4 Application Registry Bank

All the registered application's information is stored in a common registry, the Application Registry Bank (ARB). Every time a application registers in the platform, a entry is created in the ARB so that it can be queried by any module or function that demands it. Each application has its own information contained in a configuration file that stores the following data:

- **Name -** Unique ID for the application;
- **Type and Class -** Types and classes of applications according to the analysis already described in section 2.1.7;
- **Version -** Application version in order to determine the implemented functionalities and also the application's compatibilities;
- **Functions -** Functions to be used and loaded case the platform is being executed on a Vehicle station or RSU;

The ARB, besides supplying the functions to register and unregister applications, also offers a series of functions to retrieve the above mentioned information by any other applications or services.

3.4.5 Services

One of the main and most important functionalities of the platform are the Services. Services will guarantee a large percentage of the abstraction to applications development as well as play an important role in the platform's overall performance. Therefore some important characteristics of the services are:

- Services are available to all the applications of the platform.

- Applications can implement and deploy their own services onto the platform, in order to enhance the existing or create new ones, either from scratch of by composition of the available services.
- All the information produced by services is delivered to applications by the Publish-Subscribe system.
- Services can only access information retrieved from the support platform through ARM interactions;
- Services have versions in order to determine the implemented functionalities and also the overall compatibility;
- Services have priorities;

All services are executed at a given frequency, determined by the VADM platform configuration file. By fixing the publish frequency of the services, it is possible to guarantee the platform stability and control of resources usage. Also it is possible to control the publishing frequency, increasing or decreasing it, and therefore adapt the resource's usage by the services to cope with safety situations or system overload. As mentioned, that frequency is controlled by the ARM module.

Services are divided in 2 queues, one for user-contort services and other for safety-related services. At the given publishing frequency, safety-related service's queue is executed first followed by the user-contort service's queue. By dividing in 2 different queues, service's execution can be controlled based on the service's functionality.

As in applications, services also have meta-information, stored in a given configuration file, stating the following:

- **Name -** Unique ID for the service;
- **Version -** Documented version in order to determine the implemented functionalities;
- **Type and Class -** Service type and class according to the classification adopted in section 2.1.7;
- **Service Attributes -** Expected input and output value types so that application know what to supply and expect from the services;

Applications can retrieve service's information, through a VADM API primitive, and then decide either to use the available services, install new ones or implement improvements to the ones available in the platform. Applications may decide upon updating or not the available services by basing in the specified versions, whose features will be stated and defined in the available documentation. Also, by versioning services it is possible to determine its compatibility with other services, applications or support platform.

The proposed solution remains configurable and customizable due to the possibility of installing or upgrading services by the applications. For abstraction and security purposes application service's implementation are not persistently installed in the platform. Instead, when needed, the application service's implementation is sent and deployed in the Service's module and executed, publishing the results at the next publishing period. Also, this allows that applications only execute services when in need. The publishing frequency of the application service's is limited by the already mentioned VADM Services Publishing Frequency, that may change based on several factors.

The services available and persistently installed in the platform can be extended or reimplemented if needed.

3.5 System Analyzer

The System Analyzer module is responsible for determining both platform and vehicle status in order to decide upon the need of taking actions based on the later. This module can change the VADM status in 3 different fashions:

1. **Analytical -** Determines the need of altering the state based on the analysis of the data gathered from several sources such as, in-vehicle sensors, GPS data, modules malfunctioning and support platform information.
2. **User Input -** Change the status based on the user-input through a human-machine interface, by declaring one of several possible situations.
3. **Epidemic Status Dissemination -** The status change is driven by information received from the surrounding nodes alerting to the fact that one or more vehicle's status is altered from the normal situation.

As previously mentioned, VADM's status can be *Normal*, *Emergency* or *Accident*. The following sections will detail, the analytical and user-input methodologies for status altering, as well as the protocol followed for determining the plausibility of a epidemic status change driven by network-received information.

3.5.1 Analytical Heuristics

System Analyzer module fetches the information retrieved by the Call Manager from the support platform and runs several analytical heuristics in order to determine the need of a VADM platform status change. Several criteria and heuristics can be used, based on the amount of information supplied, in order to determine the current context and situation of the vehicle.

For this work, the authors propose two analytical heuristics to be used:

1. **GPS Data Analysis -** Take into account information gathered by the GPS device and infer based on details such as, position and position variation, current and past speed and altitude.
2. **Vehicle Sensors -** Gather information from the in-vehicle sensors that give details such as, current speed, current engine load, airbag sensors, car-systems malfunction, among others.

All of the information about is retrieved from CallManager through the ARM module. The heuristics used to determine the status alteration can be further improved and can subject of several research work. The enhancement of the heuristics used, based on several sources of information

3.5.2 User-Input

In order to allow the driver or any occupant of the vehicle to declare the status alteration based on the context a interface was created with the following options:

- **Request Assistance -** User can request for assistance for both mechanical or medical situations. System Analyzer declares status as Emergency
- **Declare Emergency Situation -** Choose an emergency situation such as Emergency Driving or severe medical assistance. System Analyzer declares status as Emergency
- **Declare Accident -** Declare accident and trigger emergency services call. System Analyzer declares status as Accident

This option lacks the control of the truthfulness of the user declaration. Analytical options should be the base of all the status alteration on the proposed platform. In order to enhance the reliability of the user-input method, every time a emergency or accident status is declared, some comparison with the information retrieved by both vehicle's sensors and network can be performed.

3.5.3 Epidemic Status Dissemination

In order to disseminate the status of a vehicle to a region where that status affects the surrounding intervinients of the network, a Epidemic Status Plausibility Dissemination Protocol was defined, henceforth described as ESPD.

First, we defined a metric called the Shortest Safety Distance (ShSD) that defines the minimum security distance between 1 moving vehicle and 1 stopped vehicle, in order for the moving vehicle to stop before crashing without suffering more than 1G deceleration. ShSD takes into account both the minimum distance for a vehicle to break and also the driver's reaction time. ShSD is determined by the following formula:

$$ShSD = (2.t_{react}.v) + (\frac{1}{2}\frac{v^2}{deceleration})$$

$$t_{react} = 2.5s = \ average\ driver's\ reaction\ time\ in\ road\ traveling\ situation[37]$$

$$v = \ vehicle's\ instant\ speed\ in\ m/s$$

$$deceleration = 9.8m/s^2 = \ gravitational\ constant\ of\ Earth$$

For example, a vehicle traveling at 80 km/h would take 136,1449 meters to stop between driver noticing the vehicle, applying brakes and traveling vehicle stops completely. Table 3.2 depicts some ShSD values, but note that this formula assumes some simplifications, not taking into account vehicle's weight and friction applied to the road.

To calculate the distance between 2 vehicles GPS coordinates, we used the *haversine* formula:

Table 3.2: Average ShSD values for typical vehicle+s traveling velocity

Velocity (km/h)	Velocity (m/s)	ShSD (m)
30	8,333	45,2078
50	13,889	76,286
80	22,2	136,1449
120	33,333	223,353

$a = \sin^2(\Delta\varphi/2) + \cos(\varphi 1).cos(\varphi 2).\sin^2(\Delta\lambda/2)$

$c = 2.atan2(\sqrt{a}, \sqrt{1-a})$

$Distance = R.c$

$\varphi = latitude$; $\lambda = longitude$; $R = 6,371km,\ Earth's\ radius$

Figure 3.3 depicts an illustration on how ShSD can vary with the vehicle's speed and how the plausibility range can affect other circulating roads.

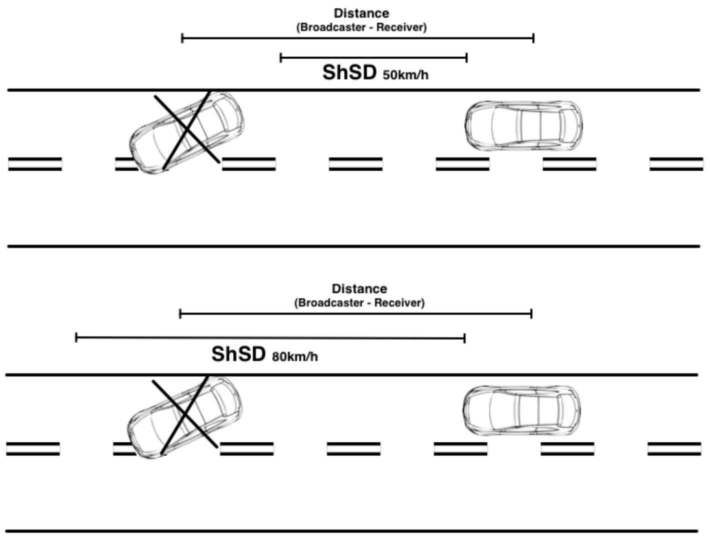


Figure 3.3: Shortest Safety Distance range scenario illustration for vehicle traveling at 50 km/h and 80 km/h

With this metrics we can define a region where the alteration of one vehicle's status also imposes the alteration of the neighbor's status. That region is described henceforth as Region of Plausibility (ROP), and poses that all vehicles inside ROP must activate their scheduling mechanisms giving priority only to relevant applications to be used in the current scenario of the region. Figure 3.4 illustrates a highway scenario for the ROP, representing vehicle's status inside and outside ROP in a situation where one vehicle had an accident and changed its status to **Accident**. Every time a vehicle gets outside the ROP, changes its status back to **Normal**. Note that vehicles traveling on the opposite direction of the highway, if inside the ROP also must alter their status to the broadcaster status in order to only use safety applications and therefore minimize network congestion with non-priority traffic.

The ESPD is split in two different procedures:

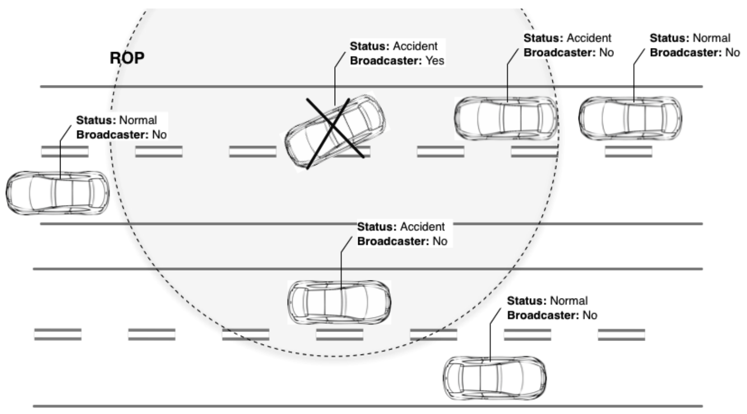


Figure 3.4: Highway scenario illustration for Region of Plausibility with a vehicle broadcasting ”Accident” status

1. **ESPD Receive -** ESPD receives and parses information contained in the received Basic Safety Message (BSM) at the Communications Manager module and determines plausibility of the status alteration.

2. **ESPD Advertise -** Whenever a status change is triggered, the ESPD Advertise service beacons a BSM through the Communications Manager module, issuing the need of a status change to the vehicles nearby.

Note that, despite the literature states that BSM should be transmitted at 10Hz, due to the different purpose of the BSM used in the ESPD, the later is only transmitted when a status change is detected.

Figure 3.5 shows the ESPD protocol for both receive and advertise procedures.

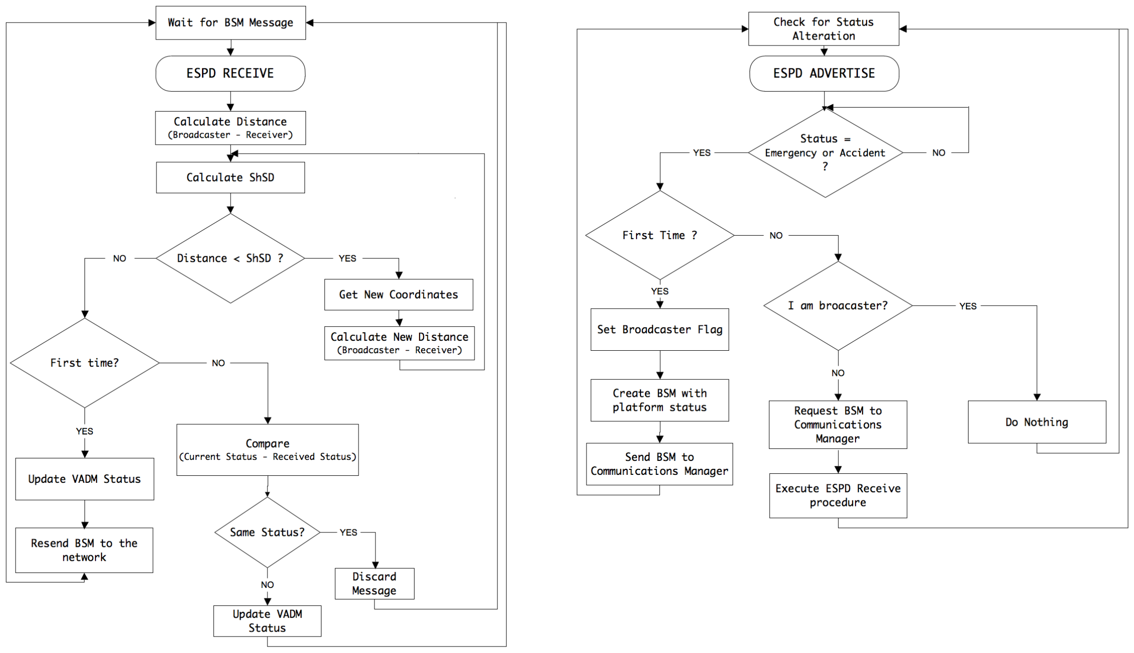

Wait for BSM Message
ESPD RECEIVE
Calculate Distance (Broadcaster - Receiver)
Calculate ShSD
Distance < ShSD ?
NO
YES
Get New Coordinates
Calculate New Distance (Broadcaster - Receiver)
First time?
NO
YES
Update VADM Status
Resend BSM to the network
Compare (Current Status - Received Status)
Same Status?
YES
NO
Discard Message
Update VADM Status
Check for Status Alteration
ESPD ADVERTISE
Status = Emergency or Accident ?
YES
NO
First Time ?
NO
YES
Set Broadcaster Flag
Create BSM with platform status
Send BSM to Communications Manager
I am broacaster?
YES
NO
Request BSM to Communications Manager
Execute ESPD Receive procedure
Do Nothing

Figure 3.5: ESPD protocol, for handling and advertising network status alteration messages.

1. **ESPD Receive -** Distance between the broadcaster and current vehicle is calculated in order to compare it with the ShSD, also computed at that moment. If the distance between both vehicles is smaller than the ShSD means that the circulating vehicle is within collision range with the broadcaster, therefore inside the ROP. If inside the ROP, the procedure checks if it is the first time the message is received. In case it is the first time this message is received, the platform changes its status to the status of the broadcaster, and resends the BSM to the network without altering it. If the system message was already received, it compares the status in order to comprehend if there was a change on the ROP status and updates if needed. In order to determine if a vehicle is getting inside or further the ROP, the procedure computes the distance comparison with ShSD several times.
2. **ESPD Advertise -** Checks the new platform status. If is either Accident or Emergency, tests if the procedure was already executed. If it was not executed before, creates a BSM with the platform and vehicle information and sends to Communications Manager module. In case it is not the first time it is executed, the protocol checks if the status alteration was caused in the local platform, and if it was not, requests the last BSM message received from the Communications Manager Module to determine if there is an update on the ROP status, and executes the ESPD Advertise procedure.

The ESPD helps determining a region where the VADM platform status alteration poses a valuable role in making the network available only for safety-related applications, minimizing the bandwidth usage by applications whose objective does not relate with guaranteeing safety of both vehicles and their drivers. The protocol is designed to adopt future changes in the criteria for establishing the region boundaries for status alteration, posing some flexibility for future implementations and improvements of the platform.

3.6 Applications and Services Priorities

Applications can be diverse and pose different requirements from both the platform and the network based on its functionalities. The right to detain a certain share of access to those resources is measured by the application objective, and in the case of vehicular-networks, if the application is directly related with safety goals or only to deliver a more pleasant driving experience to the user.

Therefore, and in order to classify which applications should detain the right to use the functionalities provided by the platform in several situations, priorities were established.

As stated in [7], the architecture of communications in ITS standards should be designed to support multiple classes of ITS applications. Applications may require more or less stringer requirements depending on how much the later rely on the communication services with respect of reliability, security, latency and other performance parameters.

Also, for applications used in emergency situations, for example, during a car crash, several mechanisms for the Access Layer to send time-critical traffic were proposed. Those proposals can be divided upon two different strategies:

Table 3.3: Applications priority analysis for different environment situations

	Class	Normal	Emergency	Accident
Safety-Related	**Information**	Medium	Medium	Low
	Assistance	Medium	High	High
	Warning	Medium	High	High
User-Comfort	**Information**	Medium	Medium	Low
	Entertainment	Medium	Low	Low
	Business	Medium	Low	Low

- Change intelligently the number of vehicles transmitting the emergency messages and the rate at which they are transmitting, [38]
- Change the transmission range of the emergency messages as the number of affected vehicles increase, [39]

Despite the fact that prioritization of transmission is being handled on the Access layer of the underlying support platform, some prioritization scheme is addressed in the proposed architecture in order to cope with the several possible execution environments that pose several different situations and that affect how applications behave and what applications and services are more relevant in such situations.

In order to define application's priority, an analysis was performed, for each possible environment status, *Normal*, *Emergency* and *Accident*s, based on the following guidelines:

- Relevance for drivers safety
- Communications services reliability
- Impact on platforms performance
- Time-Critical

Such analysis is presented on table 3.3, where three levels (Low, Medium and High), based on the above mentioned guidelines, were used to determine applications priority for all the considered execution scenarios.

From the performed analysis, one may conclude that application's priority depends on both type and situation. We will now detail on the next section the policies adopted for both applications and services based upon their priorities and class.

3.7 Rescheduling and Discard Policy

Different policies are applied to services and applications since those use different functionalities of both the support platform and network. Several different policies are used, based on the VADM status:

- When the status is **Normal**, all application types have the same priority having equal access to resources;
- In case of **Emergency** status, all *safety-related* applications have higher priority but with more relevance for *Assistance* and *Warning* types. Overall *user-comfort* applications have less priority, exception made for the *Information* type, whose functions of delivering information about local hospital or automobile workshop might be relevant.
- If the **Accident** status is declared, all *user-comfort* applications have lower priority, meaning no access to platform resources, while *safety-related* applications, in particular of the *Warning* and *Assistance* type have higher priority for accessing and using platform resources due to its relevance in drivers safety.

Based on this, all the application priority schemes embedded in the platform will take into account the application's class and type.

A similar analysis can be performed for services, but with the particularity that the priority schemes for services can be used both in the platform and within applications, for instance to choose what services to subscribe.

In sum, several different procedures are done in different modules depending on the platform status and application/services priority:

- **Status:** Normal
  - All applications and services execute normally
  - No priority check is performed
  - All applications and service have similar access permission to resources
- **Status:** Emergency or Accident
  - Every time an application wants do send information, the ARM decides wether or not the message should be sent, discarded or stored for later retrieval.
  - Every time an application wants to send a message, depending on application priority and messaging scheme, several communication technologies can be used. That decision is made on the ARM.
  - The ARM may shutdown all the non-relevant services and applications based on their priority (determined through the class and type of the later), hardware load or any other relevant criteria.

– When CommManager receives data, determines the application or service that originated the data and decides wether or not that message should proceed to the applications, discarded or stored for later retrieval.

Taking the above into account, the following sections with detail the policies followed for both applications and services, depending on the platform status.

3.7.1 Application's Policies

Applications are subjected to classification and conditioning based on the VADM status and their class and type, as defined in the previous section. Whenever the VADM status is set to *Normal*, all applications have equal right on the access to the available resources. If the status is *Emergency* or *Accident* policies are applied. Figure 3.6 depicts the policies applied to all application's messages in *Emergency* and *Accident* situations.

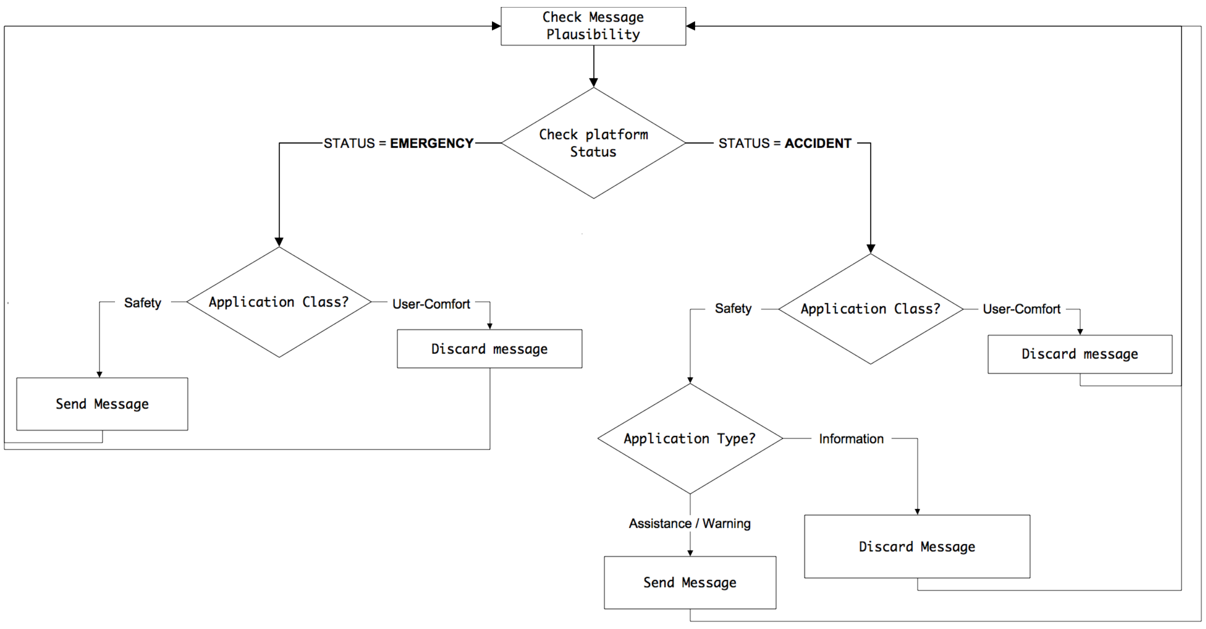


Figure 3.6: Policies applied to application's messages for both *Emergency* and *Accident* situations by the Communication Manager and ARM schedulers

Case the VADM status is **Emergency**, all messages originated from user-comfort applications are discarded. If the message is generated by a safety application of any time, it has normal access to platform resources. If the VADM status is **Accident**, user-comfort applications are discarded. Then, safety-related messages are evaluated based on its type (Information, Assistance or Warning), and all Information-type messages are put into a buffer, while Assistance and Warning-type have normal access to the platform resources. If no Assistance or Warning messages are to be sent and the resource is available, then the Information messages have access to the underlying support platform communication facilities.

Application policies are applied in the ARM and Communication Manager modules whenever a application is trying to send messages to the network, or a message is received from the network, by a functionality named *Message Plausibility Scheduling*.

Different policies can be developed and applied in the platform in order to give more just access to all kinds of applications in all situations. For instance, instead of discarding application's messages, buffers with timeout or buffers with reorganization based on the application priority can be used. Also, as other policy that can be further taken into account is applications being shutdown by the system manager module, whenever they are consuming to much computational resources.

3.7.2 Services's Policies

Services policies are applied also based on the class and type of those. However, more simple approach is taken regarding the policies for systems whenever the platform is in a Emergency or Accident situation.

Case the platform status is defined as *Emergency* or *Accident*, only the queue with safety-related services is executed, in order to spare the available resources. Figure 3.7 depicts the policies applied to service's at the platform publish frequency.

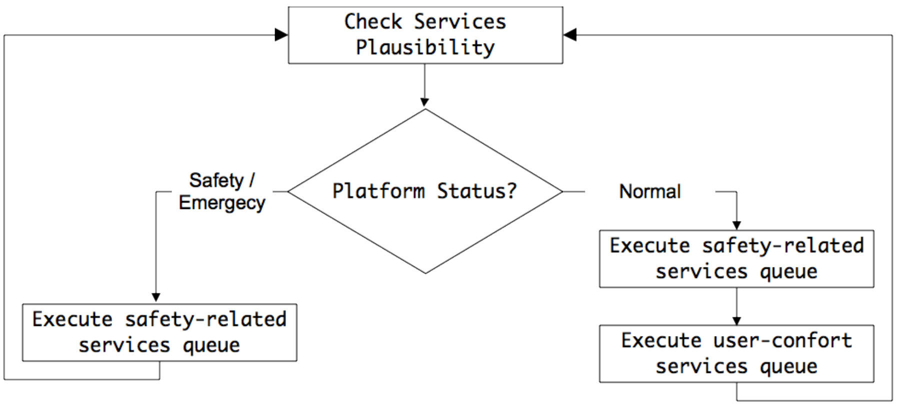

Check Services Plausibility
Platform Status?
Safety / Emergecy
Normal
Execute safety-related services queue
Execute safety-related services queue
Execute user-confort services queue

Figure 3.7: Policies applied to services for both Normal or Emergency and Accident situations, executed at the publishing frequency

Regarding the application' services, no action is needed, because when in *Emergency* or *Accident* situation, only safety-related applications are executed and therefore only the services to serve those purposes are installed and executed in the platform.

Service policies are applied by the Services modules that control all the running and deployed services.

As in applications, several other policies can be applied to services in the various situations in which the platform can be executing, posing one possible improvement point for future enhancements of the proposed solution.

3.8 Applications

Applications can interact with the developed architecture in order to exchange information, both with other vehicles and participants of the network and also with the several services and functionalities offered by the platform.

Applications to be used with the VADM interact with the platform in two different ways:

1. **API functions -** Direct usage of a set of API functions made available by the platform
2. **Publish-Subscribe -** Subscribing to one or more services made available by the platform

Figure 3.8 depicts an illustration on how an application interacts with the platform.

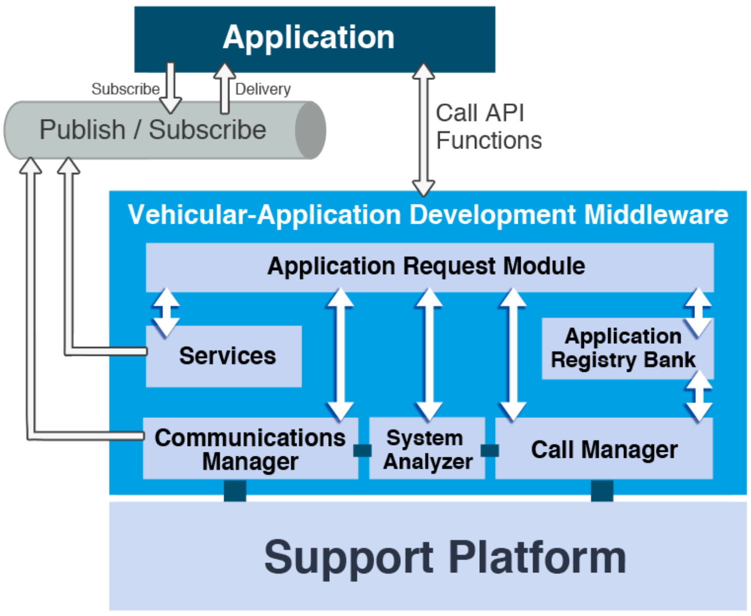


Figure 3.8: Interaction between applications and the proposed platform

Applications should create functions and functionalities for all the several node types in which they might be installed, e.g. Vehicle and RSUs. Also, applications may develop services to be deployed within the platform in order to extend its functionality while using the available resources of the later. There is a set of system services that are subscribed by default:

- **InterNodeCommunication -** Obtain all the messages sent by the same application, installed in other vehicles. Communication Manager module, publishes this information based on Application unique ID.
- **SystemMessages -** Receive informative messages, generated by the **System Analyzer Module** regarding system status or system changes.

Note that the remaining available services in the platform can be subscribed optionality by applications.

3.8.1 Application's Lifecycle and functioning

Figure 3.9 depicts application's lifecycle and process since the early startup and initialization up until the terminating process.

Applications start by loading their configuration and default values from the application configuration file and then register themselves in the VADM platform. After that a request for the current node information is performed in order to define the application behavior, functions and functionalities

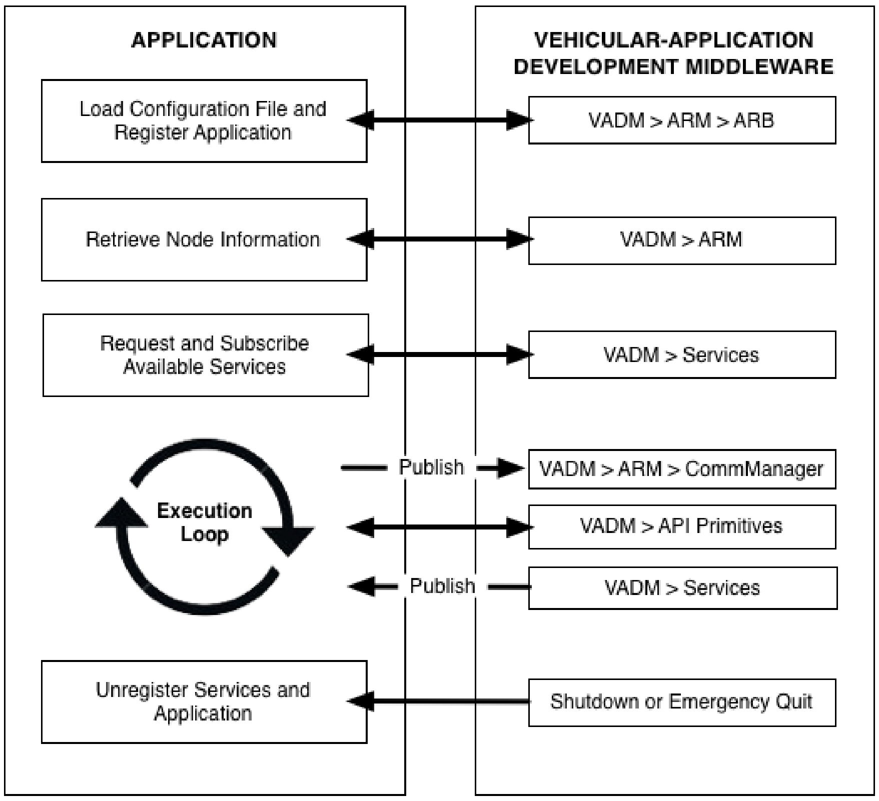

APPLICATION
VEHICULAR-APPLICATION DEVELOPMENT MIDDLEWARE
Load Configuration File and Register Application
VADM > ARM > ARB
Retrieve Node Information
VADM > ARM
Request and Subscribe Available Services
VADM > Services
Execution Loop
Publish
VADM > ARM > CommManager
VADM > API Primitives
Publish
VADM > Services
Unregister Services and Application
Shutdown or Emergency Quit

Figure 3.9: VADM application's lifecycle and interaction process with the developed platform.

to execute, based on node type. A request for the available and active services on the platform is performed in order to determine the need to subscribing to some of those services or deploy some application services later in the execution.

After the setup process the application starts its normal functioning and interacts with the platform using the VADM API primitives and obtains information from the platform through the subscribed services with the publish-subscribe mechanism. If some service need to be deployed or updated in the platform, that is performed during the execution loop.

In case of Emergency situation or platform shutdown, VADM triggers a shutdown function in the application that will start by unregister its serves and application from the platform and exit seamlessly.

3.8.2 Application's Messaging

Safety applications messaging is J2375 compliant. Therefore all the safety messages created by those applications are sent to the platform encoded with that standard. Also, if the VADM platform wants to deliver a message to a specific application, the service responsible for publishing that information, does it without altering the J2375 message received from the Communications Manager module. By following this standard, our platform supports both product innovation and differentiation, while maintaining interoperability among other vehicular-applications developed.

User-comfort applications use the traditional Internet Protocols, made available by the platform, like TCP and UDP, with no encoding.

More protocols can be made available as long the underlying support platform made those available.

3.9 Chapter Summary

In this chapter we have presented the overall description of the proposed architecture in order to cope with the imposed requirements. Based on the later, we have presented the design options of the architecture and its modules and out it interacts with the support platform as well as the integration with the applications that will make use of our solution.

Taking into account the desirable architectural features already stated in section 3.1 and defined by the authors in [31], the following list explains how those features were addressed:

- **Future-proof -** By providing versioning systems for both applications, services and platforms it is possible to guarantee the retro-compatibility between old and newly defined applications.
- **Flexibility -** By having only 2 points of contact with the support platform, through Communications Manager and Call Manager, the developed platform can cope with different underlying implementations, offering a considerable amount of compatibility.
- **Extensible -** By offering no constrains regarding application's requirements and data sets, the proposed platform supports multiple application's implementations.
- **Unified Interface -** Since all the applications interact with both vehicle and the network throughout the VADM, centralized systems for management and security are guaranteed.
- **Layered Architecture -** By following the standardized ITS platform requirements and design orientations, the architecture is developed in a layered paradigm, while guaranteeing compatibility with other research work that follows the same standardized ITS platform.
- **Low bandwidth usage -** By managing what applications and services are executing and have access to communication resources, the proposed architecture can guarantee that only applications whose functioning is valuable for the situation in which the platform is being executing, are running. Therefore, the bandwidth is only occupied with plausible and relevant traffic. Also, in case of multichannel or diverse communication technologies are available, the platform also support scheduling to the later.
- **Information Rate -** By allowing the applications to set their information data-rate, the proposed architecture lets applications choose the frequency in which information should be sent. In case of emergency situations the data-rates may change depending on the application's priority.
- **Recognize vehicle capabilities -** By changing part of the Call Manager implementation it is possible to adapt the proposed architecture to the vehicle-available information.
- **Enable product differentiation -** By supplying a uniform set of API functions and use J2375 compliant messaging the proposed architecture lets application developers work with the same

basis while opening the possibility to create differentiating functionalities in their applications, granting a certain degree of interoperability.

In chapter 4 we will describe the implementation details and decisions made in order to create a prototype for the architecture proposed in this thesis.

4 Implementation

This chapter will address the main decisions adopted regarding the implementation of the present work platform. A description and brief analysis on the available solutions for programming language and technologies is presented. In section 4.1 the authors present the analysis and decisions regarding the programming languages and technologies adopted. The following sections will describe the implementation options of the proposed solution modules, already presented in the previous chapter. Section 4.7 presents the conclusions obtained from the implementation process of the present work.

4.1 Adopted Technologies

In order to develop the proposed work, several options regarding the technologies were evaluated and opted. Such evaluation was made taking into account the requirements posed in the problem definition and other requirements such as performance, extendibility and readability of code, demand for extensive documentation and easiness of functionalities extendibility by others. The following sections will illustrate the decisions made regarding the programming languages and other implementation options adopted.

4.1.1 Programming Languages

A set of programming languages were available to implement the solution proposed in this work. Those languages had as requirements the availability of free compilers/cross compilers, provide easy to read/write code and supply easy access to shell scripting and system commands. Also, a object-oriented language was required in order to facilitate code reuse as well as extensibility and modularity. Having that in mind, both **Java** and **Python** posed as possible solutions.

Python was adopted as chosen programming language because it offered the following advantages when compared with Java:

- **Readable Code -** Since the proposed solution aims to be developed in a open source fashion, the ability of having short-sintaxed and easy-to-read code posed an enormous advantage because it facilitates further and future improvements on the platform code without having to follow deeply intense amount of documentation. Such readability is deepened with the availability of simple to use Callbacks and Factory methods functionalities.
- **Compact code -** By using a object-oriented programming language, classes definition is used often. In Java each top-level class is traditionaly defined in its own file, while Python allows multiple classes to be defined in a single file. Therefore, the amount of files coded is substantially decreased permuting for instance using only one file per module while containing the full features of those.
- **Performance -** Despite the fact that Python can be slower than Java in some situations such as object allocation, interpreter speed and native methods usage, Python offers better results in I/O handling, standard output and interpreter initialization that may be of relevance when used in embedded systems for the functionalities required in this platform.

Both languages are subjective to discussion in several latitudes since there are several different implementations of both Python (CPython, IronPython, Jython, and PyPy) and Java (HotSpot VM, OpenJRE, Mac OS X Java VM, etc.) that pose different performance results. Since vehicles nowadays start offering powerful embedded processors due to the expansion of several in-vehicle services, hardware will not create performance limitations, therefore posing Python as a reasonable option to be used in the presented work.

4.1.2 Implementation Options

All the configuration files were developed in Extensible Markup Language (XML) due to the high readability delivered for being a very descriptive language. Despite the fact that other languages, for instance JSON, offer less overhead than XML, such characteristic would only affect solution's performance if configuration files where exchanged between modules, for messaging for instance, not being the case in the proposed work.

For inter-module communication an adaptation of JSON scripting language was used, due to the less overhead when comparing to XML.

In order to create the Publish-Subscribe mechanism PyPubSub 3.1.2 [1] was used. This packaged was chosen for being developed in a community-driven fashion and also due to its simple implementation and usage.

[1] https://pypi.python.org/pypi/PyPubSub

Table 4.1: Implemented functionalities for the support platform

Functionality	Not Addressed	Addressed
Routing Protocol	X	-
Communication Technologies		
802.11p	X	-
WLAN	-	X
Wimax	X	-
Cellular Networks	X	-
Location		
GPS	-	X
In-Vehicle Sensing		
OBD	-	X
Beaconing	-	X
Management Information	-	X

4.2 Support Platform

As stated in section 3.2.1, our work assumes the existence of a support platform that addresses functionalities of the underlying Transport, Network and Access Layer. Also, by following the standardized ITS platform, functionalities regarding the management and security should also be addressed by such support platform.

Since no platform was available for testing, the authors developed a prototype for a support platform with some functionalities. Table 4.1 states the expected functionalities of a support platform and the functionalities implemented in the prototype developed:

No routing protocols were implemented due to simplicity of the test scenarios to be performed in the evaluation of the proposed solution. When adding more complexity, meaning more vehicles and intervinients, to the test scenarios, the need of routing protocols arises. Also, since no geographic routing protocols were used, location services as HLS, RLS and GLS are not addressed as well.

Regarding the Communication Technologies, no vehicular networks oriented technologies were available to perform the tests. In particular equipments that use the IEEE 802.11p standard, specifically designed to address problems of vehicular communications, would pose a important role in the test scenarios. Therefore, an adaptation to simulate some of the improvements on that standard was performed by using off-the-shelf wireless LAN technologies.

The standard IEEE 802.11p in Europe uses the 5.9 GHz band with 10 MHz channels, as approved by ETSI in 2008. There are 7 channels, starting at the 5,850 GHz, where 4 of them are allocated for both Safety and User-Comfort traffic, 1 control channel and 2 channels dedicated for special applications, Critical Safety of Life and High Power Public Safety [40].

In order to simulate the specific channel allocation in the 802.11p standard, the support platform is able to support 3 (two) 802.11n interfaces, allocated in different channels. 802.11n standard uses the 5GHz band, a maximum of 150 Mbit/s data rate and has 250 meters outdoor range. Channels 100 (5,500 Ghz) and 108 (5,540GHz) will be used in order to minimize the interference between those channels. One channel will be used to transmit all the data generated from both User-Comfort and Safety applications, while other will be reserved for safety control information only. Figure 4.1 illustrates the channel allocation for the IEEE 802.11p in Europe, and the adopted channels to simulate its functionality in this thesis implementation.

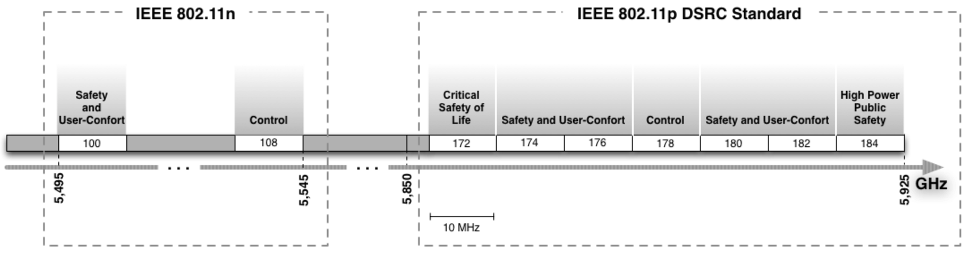


Figure 4.1: European Channel allocation for the IEEE 802.11p standard side-by-side with the adaptation implemented

The multiplexing of the messages to the control and standard channel is done by gathering information from the J2375 compliant messages. Messages will include an overhead that defines which technology should be used. That choice is done by the VADM platform, that is aware of application's types and requirements.

The implementation of the above mentioned adaptation is done in the Interface Manager module, further explained in section 4.2.2.

To supply location information to the vehicles a GPS receiver was used. A OBD interface was used in order to retrieve information from the in-vehicle sensors. Details about the implementation of this sensing functionalities is further described in section 4.2.1

Beacons make use of both GPS and sensors data to enrich the information sent to other vehicles. Beacon generation as well as the information sent in each beacon is detailed in section 4.2.2.

Table Manager, gathers information from all the above mentioned sensors as well as parsed information from the beacons received from the other intervinients, neighbors of the vehicle within the network.

Figure 4.2 depicts the Support Platform developed and all the modules that will be described in detail on the following sections.

4.2.1 Acquisition Module

The Acquisition module is responsible for getting information from both in-vehicle sensors and GPS receiver, parse that information and make it available to be accessed by other modules from the upper layers, and it is divided in two parts, the **Setup phase** and the **Listen phase**.

During the Setup phase, the Acquisition module creates 2 threads. The first one, responsible for obtaining GPS data from the serial port and parse values for latitude, longitude, speed in km/h, number of satellites and altitude into sea level. The second, responsible for gathering information from the in-vehicle sensors through a OBD interface. Information to be retrieved from the OBD may vary since the amount of sensors in each vehicle depends on the manufacturers. However most vehicles have a number of standard sensors embedded to retrieve information such as vehicle instant speed, engine RPM and fuel level.

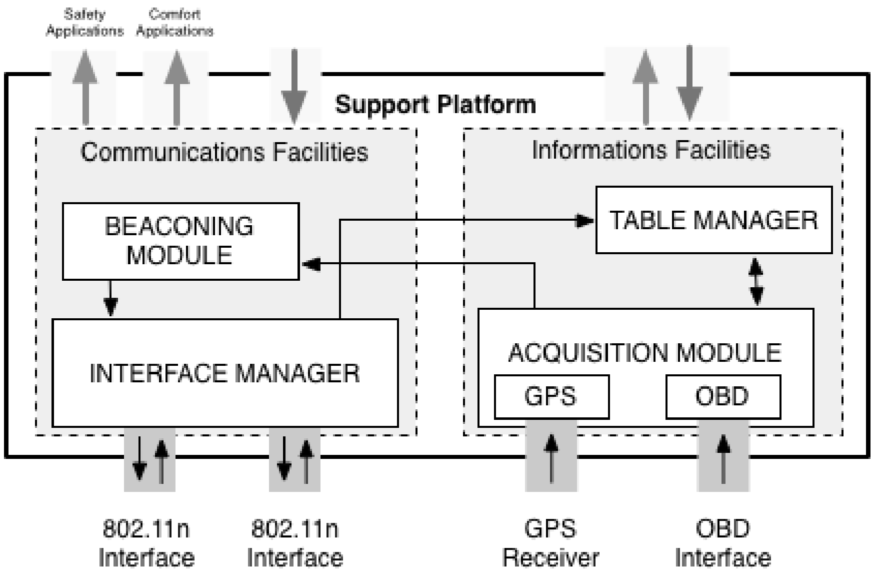


Figure 4.2: Support platform modules diagram

Each thread stores the values above mentioned and keep updating them at a given frequency, that can be adjusted by the Acquisition module, but never higher than the acquisition frequency supported by the equipments.

Handlers are created for each thread with functions to access asynchronously to the in-memory values kept by the later.

On the Listen phase, a bidirectional socket is created in order to give access to the information gathered to other modules from the layers above. A set of different requests is made available in order for those modules to choose which information they need. Requests, can be done by sending the following 3 byte length messages:

- **GPS -** Location information gathered from the GPS receiver
- **OBD -** Information retrieved from the In-Vehicle Sensors through the OBD interface
- **NEI -** Neighbors information gathered from received beacons and stored in the Table Manager
- **ALL -** All the above information.

GPS and OBD requests will trigger a thread access through their specific handlers while a NEI request will trigger a request to the Table Manager to get information regarding the current neighbors of the vehicle.

Both requests and replies are done using JSON messaging and are depicted in Listings 4.1 and 4.2. With requests it is possible to define what values are being requested, by changing the values in the JSON message.

Listing 4.1: Acquisition module JSON request messages format

```
{REQ: {
        values: <GPS,OBD,NEI,ALL>
}}
```

Listing 4.2: Acquisition module JSON reply messages format

```
{GPS: {
        latitude: <latitude>,
        longitude: <longitude>,
        GPSspeed: <speed in km/h>
        altitude: <altitude in meters>
}}

{OBD: {
        speed: <speed in m/s>,
        RPM: <Engine Rotations per Minute>,
        fuel: <Fuel Level>
}}

{NEI: {
        id:<License plate number>,
        ip:<IP Address>,
        information:{
                speed:<speed in m/s>,
                },
        position: {
                latitude: <latitude>,
                longitude: <longitude>,
                GPSspeed: <speed in km/h>
                altitude: <altitude in meters>
                },
        ts:<Timestamp>
}}
```

Acquisition module pseudo-code is detailed Appendix 2.

4.2.2 Interface Manager

The Interface Manager module is responsible for managing the available communication technologies in the support platform since several, and maybe diverse, communication technologies can be connected to it.

To connect with the VADM platform, the Interface Manager offers syncronous inter-process socket connections. Therefore 3 connections are established:

1. Receive all information sent by the Communication Manager that is going to be sent to the network
2. Deliver all the inbound safety-related traffic to the Communication Manager module
3. Deliver all the inbound comfort-application's traffic to the Communication Manager module

The reason why 2 different sockets are established to deliver inbound traffic to the Communication Manager is so that additional multiplexing and scheduling can be done by the VADM platform. For the traffic that is to be sent to the network only one socket is created because, messages are already stamped with priorities and scheduled in the VADM platform. No prioritization of messages being sent to the network is done, but stamping is kept to support future inclusion of communication technologies as IEEE 802.11p.

In our implementation, the basic transport protocol used for messaging between the nodes was UDP to follow the decoupled mind set adopted in the rest of the architecture and also spare the network of the traffic overhead that TCP protocol imposes.

Interface Manager module is divided in two (2) classes:

1. **Broadcast Class -** Responsible for receiving the information sent by the Communication Manager, and choose the adequate communication technologies based on the overhead attached by the module. The overhead states both the expected communication technology/channel to be used (comfort or safety) and also the relative priority of the messages. Messages prioritization is not adopted in our work for simplification purposes.

2. **Listeners Class -** Two (2) different listeners are created, one for receiving all the inbound traffic from the safety-related channel and other for the inbound traffic from the comfort channel.

Interface Manager is also responsible for managing and storing information about the current network topology regarding the neighbor nodes. For that two (2) sub modules were created, the Beaconing Module and the Table Manager.

Beaconing Module

Beaconing Module is responsible for gathering information from the connected sensors in order to compose simple, yet relevant beacons to be sent to the network at a given frequency. The adopted beaconing frequency of our solution is of 10 Hz.

Beacons contain the following information:

1. **Beacon ID -** Random number generated every time a beacon is created.

2. **Vehicle Unique ID -** Unique vehicle identifier based on its license plate.

3. **Number of Hops -** Number of hops that the beacon should be routed.

4. **Vehicle IP -** Source vehicle IP address.

5. **Vehicle Location -** GPS coordinates of the source vehicle.

In our solution we used 0 as the number of hops to spare network resources and due to the simplicity of the network topology that is going to be used. More information could be added to beacons in order to enrich future features of both the support platform or VADM platform.

Beacon module uses the Broadcast class of the Interface Manager module, sending the beacons through the channel reserved for safety applications.

Table Manager

Whenever a beacon is received in by the safety-related Interface Manager module listener it is sent to the Table Manager module. Table Manager module is responsible for storing and updating beacons in a way that the table mirrors the current network topology on the vicinity of the vehicle.

Beacons are kept in the table using the Vehicle Unique ID as identifier and store all the information received in the beacon, already defined. Every time a beacon with the same Vehicle Unique ID is received, Table Manager updates its entry.

Each entry of the table as also a timeout value that corresponds to the time period where that entry is still valid. If a beacon is not received that updates the information regarding one vehicle in the vicity until the timeout is reached, Table Manager will delete its entry assuming that the vehicle is no longer in the vicinity. The standard timeout used in our implementation is of 1 second.

Table Manager module makes available the following functions, depicted in Listing 4.3, that can be used by other modules of the support platform for accessing its data.

Listing 4.3: Functions made available by the Table Manager module

```
getCurrentNeighbors()
insertBeacon(beacon)
getNeighbor(vehicleID)
```

The *getCurrentNeighbors()* returns the list of current vehicles in the vicity with their information, retrieved from the received beacons. Also a function to insert a new beacon to the Table Manager but no function to remove since Table Manager will take care of removing entries whose timeout has been reached. Also it is possible to retrieve information from a specific neighbor through its unique vehicle ID.

4.3 SAE J2735 DSRC Standard

In order to implement the coding and decoding of messages to the J2735 standard ASN.1 Python library was used, the *Pyasn1*[2].

The library offers the definition of the standard data structures and serialization protocols. No translation from the J2735 was available to any platform (C nor Python), therefore the transcription from the standard was done.

Listings 4.4 and 4.5 depicts the implementation of the J2735 Basic Safety Message to be used in the Pyasn1 and equivalent ASN.1 notation.

Listing 4.4: J2735 BSM ASN.1 Notation

```
BasicSafetyMessage ::=  SEQUENCE {
   msgID       DSRCmsgID,            -- App ID value, 1 byte
   msgCnt      MsgCount,             -- 1 byte
   id          TemporaryID,          -- 4 bytes
   secMark     DSecond,              -- 2 bytes
   lat         Latitude,             -- 4 bytes
   long        Longitude,            -- 4 bytes
   elev        Elevation,            -- 2 bytes
   accuracy    PositionalAccuracy,   -- 4 bytes
   speed       TransmissionAndSpeed, -- 2 bytes
   heading     Heading,              -- 2 bytes
   angle       SteeringWheelAngle,   -- 1 bytes
   accelSet    AccelerationSet4Way,  -- 7 bytes
   brakes      BrakeSystemStatus,    -- 2 bytes
   size        VehicleSize,          -- 3 bytes
   }
```

[2]http://sourceforge.net/projects/pyasn1/

Listing 4.5: J2735 BSM Pyasn1 implementation

```
class BasicSafetyMessage(univ.Sequence):
        componentType = namedtype.NamedTypes(
                namedtype.NamedType('msgID',DSRCmsgID()),
                namedtype.NamedType('msgCnt',MsgCount()),
                namedtype.NamedType('id',TemporaryID()),
                namedtype.NamedType('secMark',DSecond()),
                namedtype.NamedType('lat',Latitude()),
                namedtype.NamedType('long',Longitude()),
                namedtype.NamedType('elev',Elevation()),
                namedtype.NamedType('accuracy',PositionalAccuracy()),
                namedtype.NamedType('speed',TransmissionAndSpeed()),
                namedtype.NamedType('heading',Heading()),
                namedtype.NamedType('angle',SteeringWheelAngle()),
                namedtype.NamedType('accelSet',AccelerationSet4Way()),
                namedtype.NamedType('brakes',BrakeSystemStatus()),
                namedtype.NamedType('size',VehicleSize())
                )
```

Other Message Sets, such as the *A La Carte Message* and the *Traveler Information* were also developed. More Message Sets can be developed in further extensions of the J2735 implementation for the Pyasn1.

Besides implementing the Message Sets, both the Data Elements and Data Frames needed were also developed. The most used Data Elements are the Integer, the Octet String, Enumerated and Sequence. Data Frames are mere compositions of those elements that can later be combined in Sequences or with other Data Frames.

Listings 4.6 and 4.7 illustrate the Pyasn1 implementation of 3 Data Elements, part of the BSM, the *DSecond*, the *TemporaryID* and *TransmissionState* together with the ASN.1 notation of the later.

Listing 4.6: J2735ASN.1 Notation for the DSecond, TemporaryID and TransmissionState Data Elements

```
DSecond ::= INTEGER (0..65535)

TemporaryID ::= OCTET STRING (SIZE(4))

TransmissionState ::= ENUMERATED {
   neutral         (0), -- Neutral, speed relative to the vehicle alignment
   park            (1), -- Park, speed relative the to vehicle alignment
   forwardGears    (2), -- Forward gears, speed relative the to vehicle alignment
   reverseGears    (3), -- Reverse gears, speed relative the to vehicle alignment
   unavailable     (7), -- not-equipped or unavailable value,
   }
```

Listing 4.7: J2735ASN.1 Notation for the DSecond, TemporaryID and TransmissionState Data Elements

```
class DSecond(univ.Integer):
        subtypeSpec = univ.Integer.subtypeSpec + constraint.ValueRangeConstraint(0,65535)


class TemporaryID(univ.OctetString):
        subtypeSpec = constraint.ValueSizeConstraint(4,4)

class TransmissionState(univ.Enumerated):
        namedValues = namedval.NamedValues(
                ('neutral',0),
                ('park',1),
                ('forwardGears',2),
                ('reverseGears',3),
                ('unavailable',7)
                )
        subtypeSpec = univ.Enumerated.subtypeSpec + constraint.SingleValueConstraint
            (0,1,2,3,4,5,6,7)
```

For inter-module and inter-application messaging the authors opted for encoding all the messages that are developed for the VADM platform with a modified version of ALC message set. By adopting this message standard it is possible to standardize procedures between modules and applications and open ways for future security and message integrity mechanisms. Listing 4.8 depicts the ASN.1 notation of the modified version of the ALC message adopted in our work.

Listing 4.8: J2735 modified ALC ASN.1 Notation

```
ALaCarte ::=  SEQUENCE {
   msgID       DSRCmsgID,            -- App ID value, 1 byte
   source      IPAddress,            -- 4 bytes
   destination IPAddress,            -- 4 bytes
   port            AppPort                              -- 2 bytes
   appData     VADMApplicationData,  -- 1000 bytes
   }

VADMApplicationData ::= OCTET STRING (MAXSIZE(1..1000))
```

The remaining implementation of both *Message Sets* and *Data Elements/Frames* can be found in Appendix 1. In order to transform application's messages to J2735 compliant messages encoding and decoding is needed, therefore Encoder and Decoder classes were developed and are used in several parts of the platform. The following sections will detail both the Encoder and Decoder implementation for the aforementioned developed Message Sets.

4.3.1 J2735 Encoder and Decoder

In order to use the J2735 standard message sets adopted in our work we developed functions to encode and decode such messages as well as some other utility functions.

As mentioned in the previous section, we developed some the J2735 standard messages with Python based on the library Pyasn.1 that are now available on open source for further reuse by developers that want to follow the J2735 message sets and use Python as base coding language[3].

Listing 4.9 depicts the encoding functions made available to create the adopted J2735 messages of our work

Listing 4.9: Pyasn.1 J2735 encoding functions

```
createBSM(status, nmealatitude,nmealongitude, altitude, speed, vehicleWidth, vehicleLength)
createALaCarte(appID, sourceIP, destinationIP, destPort ,content)
```

Regarding the decoding of J2735 message, additional functions were created in order to offer the possibility to decode the full message and also the possibility of getting specific fields of the message for fast access. The available main decoding functions are depicted in listing 4.10.

Listing 4.10: Pyasn.1 J2735 decoding functions

```
decodeBSM(encodedJ2735)
decodeALC(encodedALC)
getMessageType(encodedJ2735)
```

The decoding functions return a Python dictionary with the values of the encoded J2735 message and the *getMessageType* returns the type of J2735 message. Future extensions of the J2735 message-sets based in Pyasn.1 can be implemented the future and the above mentioned functions can be later extended, offering the VADM platform the possibility to keep compliant with the standard.

[3]The repository with the Pyasn.1 implementation is available at https://github.com/tallis/SAE-J2735-Python

4.4 Vehicular-Application Development Middleware

VADM supplies an API for applications to use all the functionalities developed in the proposed solution. Having in mind that the developed applications must be decoupled and abstracted from the surrounding environment and platform implementation details, the functions stated in 4.11 were made available. For simplification purposes and also to guarantee future features like data integrity and security in the inner modules of the VADM platform, all messages sent by applications are encoded, by the applications, with a modified version of the J2735 ALC already explained in section 4.3.

Listing 4.11: Vehicular-Application Development Middleware API

```
registerApplication(applicationName, XMLPath)
getNodeType()
requestServicesList()
subscribeService(serviceName)
deployandRunApplicationService(serviceName,serviceImplementation)
publishInformation(message_{J2735ALC},protocol)
unregisterApplication(applicationID)
```

1. Register application within the platform by supplying application's name and path to the configuration XML file to be then transferred and stored by the ARB module.

2. Get information regarding the type of node where the platform is currently being used. Nodes can be Vehicles or RSUs.

3. Retrieves information about the available and active services within the platform

4. Subscribe to a given service by the service name (unique identifier). Information about what services have applications subscribes is stored in the ARB.

5. Transfers a service implementation to the Service module in order for it to be deployed and executed.

6. Sends information through the platform to the network. The message destination is defined within the encoded message and will later be interpreted by the Communications Manager module. The transport protocol to be used, e.g. UDP or TCP, can be defined.

7. Unregister application from the platform, removing all the deployed services and remaining information.

VADM poses an abstraction to all the modules contained inside itself configuring a *black-box* paradigm towards the applications. Implementation of all the modules contained inside the VADM will be detailed on the following sections.

4.4.1 Application Request Manager

ARM module plays a coordinator role inside the VADM. To coordinate all the interactions, ARM is responsible for instantiating all the remaining modules by creating threads for the Call Manager, Communication Manager and Services modules and accessing them through the respective handlers.

Each handler supplies a API to access either their functions or stored values so that ARM multiplex those functions and information while being able to answer to requests from other modules of the platform.

Listings 4.12, 4.13 and 4.14 present the handlers API for Call Manager, Communication Manager and Services modules respectively.

Listing 4.12: Call Manager handler functions

```
changeUpdateFrequency(newFrequency)
getCurrentCoordinates()
getCurrentSpeed()
getVehiclesInformation()
getCurrentNeighbors()
```

1. Change the update frequency of informations to be retrieved from the underlying support platform.
2. Get current coordinates by giving Latitude, Longitude and altitude.
3. Get current vehicle's speed in meters per second.
4. Retrieve vehicle's information obtained from in-vehicle sensors and other static information supplied to the platform.
5. Get list of current neighbors with information regarding their license plates (unique ID), IP address, position and other relevant information.

Listing 4.13: Communication Manager handler functions

```
sendMessage(message$_{J2735ALC}$)
sendMessageControlBroadcast(message$_{J2735BSM}$)
sendMessageComfort(message, protocol)
```

1. Sends message through the communication technologies for the network. Received the relayed information sent by the VADM originated from the applications.
2. Sends safety message through the control channel[4]. Can be used by inner services of functionalities of the VADM that need to broadcast J2735 messages to the network, e.g. ESPD protocol.
3. Sends message through the traditional communication technologies using the usual transport protocols as UDP or TCP.

Note that all every time these functionalities are used a *Message Plausibility Scheduling* function is executed in order to determine if the messages should proceed to the Communications Manager module or have the policies applied. The *Message Plausibility Scheduling* will be further explained in section 4.4.7

[4]Channel reserved exclusively for safety-related applications and services

Listing 4.14: Services module handler functions

```
changeFrequency(newFrequency)
deployandRunApplicationService(serviceName,serviceImplementation)
addService(serviceName)
removeService(serviceName)
getServiceList()
```

1. Change services publish frequency.
2. Run service defined by the application by passing its implementation as an argument.
3. Add service to the list of active services to execute. The new service must be already defined in the services module.
4. Remove service from the list of active services being executed.
5. Get list of current active services.

ARM module, besides accessing and managing the remaining modules of the VADM also is aware of the current status of the platform. In case of status change, ARM will adjust all the other modules behavior according to the current status of the platform. That behavior is controlled by changing the frequency values, adding or removing services or any other adaptations that can be implemented later.

When started, the ARM loads the platform configuration file where several startup values and informations are stored, such as:

1. **Name -** Descriptive name of the platform
2. **Type -** Node type [Vehicle, RSU]
3. **Publish Frequency -** Frequency in which services should publish their information
4. **Sensing Frequency -** Frequency in which the Call Manager should retrieve information from the underlying support platform
5. **ID -** Unique ID for the vehicle based on its license plate
6. **Version -** Platform implementation version, can be used to have different configurations based on a versioning system for the solution
7. **Description -** Additional comments regarding the platform implementation or configuration
8. **Default Services -** Services to be loaded and active during platform startup

Additional fields can be added to the configuration file in order to supply more information to platform implementation or default values to be loaded for other devices or by the support platform in use.

Note that the policy adopted in our implementation is of discarding the message case it is not plausible. If some other policies as storing or queuing the messages want to be used, they should be implemented here in the ARM module.

4.4.2 Platform Status

The VADM platform status was a value that needed to be accessible from all the several modules of our platform while allowing more that 1 process to change its value. To do so, a Python memory-mapped file was used. A memory-mapped file allows processes to access data on the filesystem through the operative system virtual memory, without using the common I/O functions that tend to be slow. With this approach, we managed to have a quick access to the platform status, while giving the ability of changing it by several other processes, therefore making that change visible to the rest of the VADM platform.

The memory-mapped file was also used to store other variables from some modules and procedures of the VADM platform for faster access, e.g. Node Type, ESPD protocol variables and flags, etc.

Functions to access for both read and write procedures the memory-mapped file were built and made available to all the inner modules of the VADM platform.

4.4.3 Call Manager

Call Manager module is instantiated as a thread by the ARM module during the setup period, that keeps a handler to access asynchronously to the values retrieved by the module and kept in memory. Call Manager establishes a bidirectional socket connection with the underlying support platform and sends request messages depending on the type of information that wants to obtain, as detailed in section 4.2.1.

When the Call Manager thread is started by the ARM module, and after initializing the thread variables, the Call Manager starts requesting information from the support platform Acquisition Module at the frequency stated by the VADM platform configuration file. Bear in mind that the frequency can be changed by the ARM in order to adapt the VADM platform functioning in emergency situations.

Requests are sent and received using well defined JSON messages. Every time a request is performed, a connection to the bidirectional socket between the VADM platform and the Acquisition module of the support platform is used. These requests make a synchronous calls to the support platform.

Call Manager has therefore, updated information regarding the vehicle sensors information stored in its thread variables, ready to be quickly accessed by the ARM handler functions to the Call Manager module.

Appendix 3 depicts Call Manager module pseudo-code and Listing 4.15 shows the function made available by the Call Manager module for requesting the several types of information needed and available.

Listing 4.15: Call Manager module request data function

```
requestToPlatform(request)
```

4.4.4 Communication Manager

Communication Manager module is instantiated as a thread by the ARM module during its setup period, that keeps a handler to access asynchronously to its send message functionality. Communication Manager abstracts the remaining modules and applications from the communication details and is also responsible for prioritizing applications messaging and calculate plausibility of all the received messages, using the *Message Plausibility Scheduling* already explained in 4.4.7.

Communication Manager is divided in two different threads:

1. **CommManagerSender -** Responsible for sending the information to the support platform which as the direct relation with the ARM handler.
2. **CommManagerListener -** Responsible for receiving all the inbound traffic from the support platform.

CommManagerSender

Every time a message is to be sent by the Communications Manager, a priority check is performed. All messages receive a overhead based on the adopted priority for such kind of applications. The priority check uses the application ID and current VADM platform status, that is going to check the Application's Priority Table, depicted in Table 4.2, also stored in the Communication Manager module, what is the relative priority of the message. The priority check function returns a overhead, with the channel that should be used (Safety or Comfort) and priority class, that should be attached to the message before it is sent to the support platform InterfaceManager module.

The overhead tags the message with the priority so that the supporting platform communications facilities can determine the adequate means to use, regarding both communication technologies and channel access priority. A classification similar to the one used in Class-based Queueing (CBQ) scheduling algorithm has been adopted.

Note that in our implementation, the interaction with the support platform is made using bidirectional sockets. Case a new support platform is used, the send functions should be updated to comply with the new interaction mechanisms.

Listing 4.16 depicts the relevant Communications Manager module functions to send messages to the network while Appendix 4 details the pseudo-code of the CommManagerSender thread class.

Listing 4.16: Communication Manager send functions

```
sendMessage(message$_{J2735ALC}$):
sendMessageControlBroadcast(message$_{J2735BSM}$)
sendMessageComfort(message, protocol)
```

Three (3) send functions are available:

1. Send message encoded with J2735 ALC. The message will be sent to the communication technologies and with the priority dependent upon the application type and class that generated it.

Table 4.2: Application's Priority Table according to application type and class depending upon the current VADM status

		VADM Status					
		Normal		**Emergency**		**Accident**	
		Channel	Class	Channel	Class	Channel	Class
Safety-Related	**Information**	Safety	B	Safety	B	Safety	C
	Assistance	Safety	B	Safety	A	Safety	B
	Warning	Safety	B	Safety	A	Safety	A
User-Comfort	**Information**	Comfort	B	*Discard*		*Discard*	
	Entertainment	Comfort	B	*Discard*		*Discard*	
	Business	Comfort	B	*Discard*		*Discard*	

2. Send message encoded with J2735 BSM through the available control channel. This function is used to communicate safety messages within network intervinients and therefore are not subject of prioritization.

3. Send message through the traditional communication technologies available. A specific transport protocol to be used, can be defined.

Application's Priority Table

Since different channels can be assigned for user-comfort and safety-related applications, the Application Priority Table defines which should in used in all the considered situations, according to the application type and class. Case several different communication technologies are available, this table can be upgraded with a new entry that determines what technology should be used in the several scenarios. The priority class, follows the classification adopted in this document already explained in section 3.6.

All user-comfort applications never reach this point in a *Emergency* or *Accident* situation due to already being discarded in the ARM module. Case the discard policy is changed, future upgrades on this table should be made to comply.

By querying this table it is possible to determine the overhead that should be stamped in the message to be sent to the support platform so that the adequate communication means and priorities are applied.

Case a new priority scheme or classification is adopted, only the Application's Priority Table needs to be updated.

Table 4.2 depicts the relative priority-class overhead added to applications based on the application's type and class and also the current VADM platform status.

CommManagerReceiver

Whenever a message is received on the Communication Manager, sent by the support platform, its type must be checked. Case the received information is a J2735 BSM message, a the ESPDReceive function of the ESPD protocol is executed to check if the current node should change its status to comply with the status of the surrounding vehicles. The ESPDReceive function is further explained in section 4.5.3.

Case the received message is a J2735 ALC, a plausibility check is performed in order to determine if the message should be sent to the application or of should be discarded, sparing the computational resources of the VADM platform. A message is plausible if the application that generated it is relevant to the current scenario situation where the node is executing. This plausibility check function is the same function used for message scheduling in the ARM module, already explained in section 4.4.1. If the message is plausible, the same is sent to the destination applications by using the Publish Subscribe mechanism [5].

Appendix 5 details CommManagerListener thread class pseudo-code.

4.4.5 Application Registry Bank

Application Registry Bank keeps a copy of all the registered application's information running and active currently on the platform. Every time a application is started, it must register in the platform through the API made available by the VADM already described in section 4.4. VADM will then use ARB application register function to copy application's information, stored in the its configuration file, to the in-memory application data bank. One application is only active and able to run in the platform when registered successfully in the ARB. Also, when a application subscribes to a system service, such subscription is also registered and stored. Besides the register function, ARM makes available to all the remaining modules several other functions detailed in listing 4.17.

Listing 4.17: Application Registry Bank functions

```
registerApplication(applicationID, SerializedApplicationInfo)
unregisterApplication(applicationID)
subscribeService(serviceName,applicationID)
getApplicationName(applicationID)
getApplicationVersion(applicationID)
getApplicationType(applicationID)
getApplicationClass(applicationID)
getApplicationDefaultFunctions(applicationID, nodeType)
getApplicationImplementedServices(applicationID)
```

1. 2. Register and Unregister applications functions to be used by the VADM.

3. Register service subscription by one application. Is also used to register services that are deployed by the applications in the platform.

4. 5. Get application name and version.

6. 7. Get application Class and Type.

8. Get name of application's default functions to be executed depending on node type (Vehicle or RSU)

9. Get service's names implemented and deployed by the application on the platform.

[5]All applications are default listeners of a publish service that publishes information generated by the same application received from other participants of the network

4.4.6 Services

Services module is instantiated by the ARM module during the setup period. ARM maintains a handler to asynchronously access Service's functionalities, as mentioned in section 4.4.1. Services module instantiates a set of services, that at a given frequency, compute and publish the results on a Publish-Subscriber fashion. Applications that have subscribed to specific services will then receive the results from each service. This way, a full decoupling between applications, platform and services is achieved, granting a level of abstraction among them.

Services can be already installed and configured on the platform, named **System Services**, or implemented and installed by applications, named **Application's Services**.

A set of system services is available by default in the platform but only the services mentioned in the platform configuration file, already described in section 4.4.1, will be automatically activated when the platform starts its execution.

For abstraction and security purposes, services developed by applications, have their implementation serialized and sent to the Services module that executes that service at the same frequency of the system services. With that, application's services are executed only once, or as many times a application needs, sparing platform computational resources.

Services are divided and executed in 2 different queues, one for safety-related services and the other for comfort related services. Case the VADM platform status is set to normal both of the qeues are executed. If the platform status is either *Emergency* or *Accident*, only the services that are relevant to be used in such situations are executed. This mechanism applies to both System Services and Application's Services.

Services type and class as well as other possible configurations are stored in a XML file and loaded when the VADM platform is initialized.

Listing 4.18 show the Services module available functions while the full module pseudo-code is available in Appendix 6.

Listing 4.18: Services Module pseudo-code Thread class pseudo code

```
runApplicationService(serviceName, serviceImplementation)
addService(serviceName)
removeService(serviceName)
```

An evaluation was performed in order to determine what were the basic services that most applications needed. By choosing a key set of services, one was able to unveil the possibility of service reuse by several applications. Since the mechanism for getting information from services to applications is based in Publish-Subscribe messaging, the fact that more applications subscribe to the same service does not impose challenges or more computational resources usage.

Therefore, a set of four (4) system services was developed and installed in the platform for granting basic functionality of the later:

1. **GetNodeType -** Returns the current node type (RSU or Vehicle) as well as some other additional information about the node that can be loaded from the configuration file.

2. **GetPlatformStatus -** Returns current VADM platform status.

3. **GetNodeInformation -** Get current vehicle information regarding Speed, Fuel, RPM and GPS position.

4. **GetNeighborInfo -** Return list with current neighbors information regarding ID, IP Address, speed and GPS position.

All the developed services are classified as safety-related due to its functionalities being mostly needed for safety-oriented applications.

More services can be implemented and installed in the platform as long as the API's for communicating with the remaining modules are used. Besides the System Services developed by default, there is an additional service running the ESPD Advertise function of the ESPD protocol. Such service is further explained in section 4.5.3.

4.4.7 Message Plausibility Scheduling

In order to evaluate if a message is plausible or not to be used in the VADM platform, either for sending or receiving, a Message Plausibility Scheduling function was defined. This function takes into account the policies already defined and described in 3.7.1.

Every time a message is to be sent or is received, this plausibility check mechanism is instantiated, in both ARM and Communications Manager modules respectively. This plausibility check function, checks the platform priorities and policies defined and returns if that message should be sent or discarded based on the application ID and current VADM platform status.

A message that is classified as not plausible if the current VADM status is either *Emergency* or *Accident*, posing a situation where resources should be spared. If a message is plausible it may proceed its normal execution.

The reader should have in mind that the adopted policy in this work was to discard the message case the same is not plausible. Other implementations such as the storage, queuing or prioritization of messages can be applied in this function, making it easy to renew the policies adopted in the VADM platform.

4.5 System Analyzer

System Analyzer module contains several different methodologies for determining the current execution scenario situation and altering the VADM platform status to cope with that scenario. Three (3) different approaches are used to change the VADM status and its implementation will now be detailed.

4.5.1 Analytical Heuristics

A vehicle can determine its current state based on interpretation of the information received by sensors or any other vehicles that depict a abnormal state of circulation of the same vehicle. The criteria to be used is varied and can be subject to further research work in order to evaluate automatically and with a high degree of accuracy the current vehicle state. In our work we propose a simple approach for determining if the vehicle is in a *Emergency* or *Accident* situation.

Some assumptions were made when defying the environment detection heuristics:

1. There is no differentiation between a *Emergency* or *Accident* status.
2. The frequency of heuristics execution is 1 Hz.
3. Heuristics make use of data made available by the Services module.
4. Heuristics assume that the information supplied by the Services module is fairly up-to-date and reliable.

Having the above in mind, the heuristics used for changing the vehicle status were:

1. Current vehicle speed is between 0 and 1.38 m/s (0 and 5 km/h) and the average speed of the last 3 seconds was of 8.3 m/s (30 km/h)
2. Current vehicle speed is between 0 and 1.38 m/s (0 and 5 km/h) and RPM's higher than 3000.
3. GPS traveled distance of the last 3 seconds is shorter than the average speed of the last 3 seconds.

We do not take into account some factors that can invalidate the heuristics above defined, for instance, sensors malfunctioning, sensors unavailability, or VADM platform calculation errors.

This heuristics take a simple approach for determining that a vehicle is in a emergency or accident situation by evaluating its speed variation in short periods of time. More heuristics could be defined by being based in the retrieval of in-vehicle sensors that indicate system malfunction or parsing of information from the surrounding vehicles, e.g. by determining the average speed of the vicinity group of vehicles and comparing with own speed.

4.5.2 User-Input

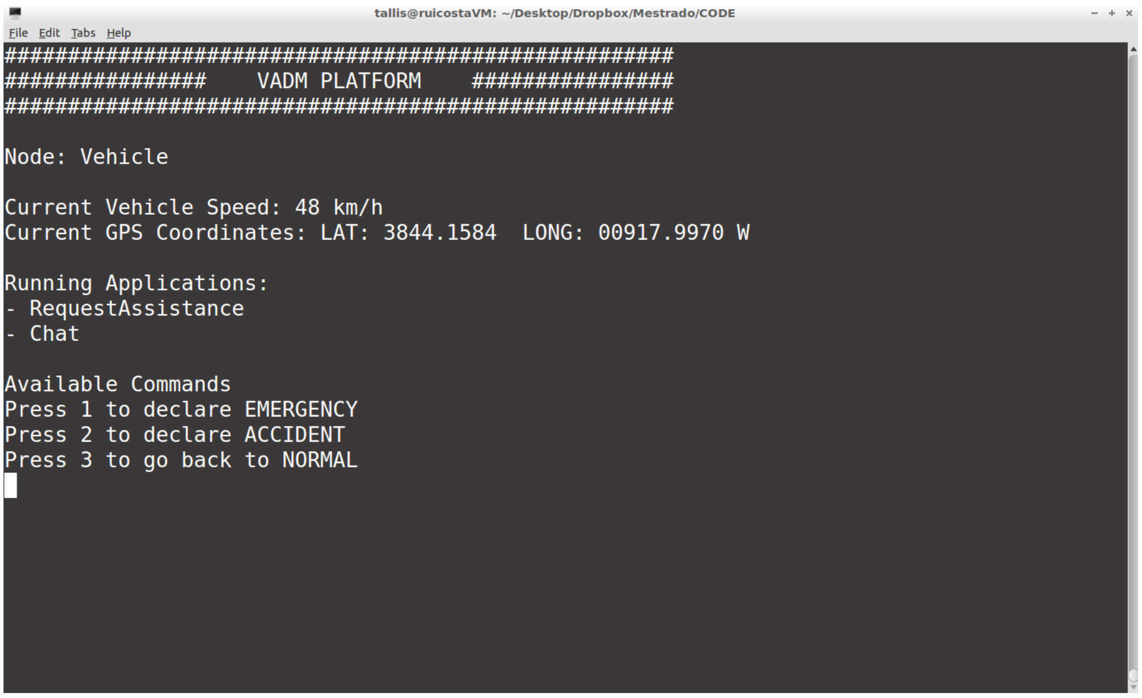
A simple user interface was designed in order to give feedback to the user and mainly to give the ability of manually setting the current vehicle situation to *Emergency* or *Accident*.

Our approach aimed to be merely representative and to help on the debug and testing, therefore lacking the validation of the commands typed. The user-interface developed is launched with the VADM platform, instantiated by ARM, and its implementation is able to access the ARM module functionalities as any other module of our platform. Therefore, the information displayed in the user-interface is gathered from the Services running in the platform, respecting the overall architecture of our solution.

Figure 4.3 shows a print screen of the user-interface made available to the manual change of the vehicle status.

4.5.3 Epidemic Status Dissemination

The ESPD protocol makes use of the J2375 standard **BSM**, for inter-node messaging, making use of fields such as *MsgCount*, *Latitude*, *Longitude* and *TransmissionAndSpeed*. In order to spread



```
#####################################################
################    VADM PLATFORM    ################
#####################################################

Node: Vehicle

Current Vehicle Speed: 48 km/h
Current GPS Coordinates: LAT: 3844.1584  LONG: 00917.9970 W

Running Applications:
- RequestAssistance
- Chat

Available Commands
Press 1 to declare EMERGENCY
Press 2 to declare ACCIDENT
Press 3 to go back to NORMAL
```

Figure 4.3: VADM platform user interface for changing manually the vehicle status and get vehicle information

the platform status the *TemporaryID* (4bytes), is used. Table 4.3 depicts the several possible platform status and their representation in the *TemporaryID* field.

Table 4.3: Platform status 4-byte representation

Status	4-byte representation
Normal	norm
Emergency	emer
Accident	acci

As mentioned, the ESPD protocol is divided in two (2) parts:

1. **ESPD Advertise -** Detect VADM platform status change and advertise the need of a platform change to the network. Executed as a service in the Services module.

2. **ESPD Receive -** Parsing of all the requests for a VADM status change received from the network and calculate the plausibility of changing node vehicle to comply with the received request for alteration. Executed every time the Communications Manager module receives a message from the network.

Algoritms 4.1 and 4.2 depict the pseudo-code for the ESPD Advertise service and ESPD Receive procedure respectively.

Algorithm 4.1 ESPD Advertise service pseudo-code

```
if currentStatus = ”Emergency” or ”Accident” then
    if firstTime = ”Yes” then
        get currentCoordinates
        create J2735 BSM message with (currentCoordinates, currentStatus)
        send message through control channel
        firstTime = ”No”
        broadcaster = ”me”
    else
        if broadcaster = ”me” then
            return
        else
            get last J2735 BSM message received
            repeat ESPDReceive algorithm two times
        end
    end
else
    do nothing
end
```

Algorithm 4.2 ESPD Received service pseudo-code

```
Data: receivedMessage J2735 BSM
get currentCoordinates
senderCoordinates ← get coordinates ← receivedMessage J2735 BSM
receivedStatus ← get status ←receivedMessageJ2735 BSM
calculate Distance (currentCoordinates, senderCoordinates)
get currentSpeed
calculate SHSD (speed)
if distance ¡ SHSD then
    if firstTime = ”Yes” then
        set currentStatus = receivedStatus
        send receivedMessageJ2735 BSM through control channel
        firstTime = ”No”
    else
        if currentStatus = receivedStatus then
            do nothing
        else
            set currentStatus = receivedStatus
        end
    end
else
    newCoordinates ← get currentCoordinates
    calculate Distance (newCoordinates, currentCoordinates)
    repeat from SHSD calculation
end
```

Each status alteration for *Emergency* or *Accident* has a timeout that is renewed every time a J2735 BSM message from the ESPD protocol is received. If the timeout elapses, the status is set back to *Normal* since it is assumed that the emergency or accident situation is not valid anymore, or the vehicle is currently out of the ROP.

4.6 Applications

Applications load all the setup information from a XML configuration file with the following details:

1. **Name -** Application unique ID
2. **Type -** Application type [User-comfort, Safety]
3. **Class -** Application's class based on its functionality
4. **Version -** Current version of the application
5. **Description -** Nominal description of the application's functionality
6. **Vehicle-functions -** Application's functions to be loaded if the later is running in a Vehicle
7. **RSU-functions -** Application's functions to be loaded if the later is running in a Road-Side Unit
8. **Platform Services -** Platform services to be subscribed case the same are available and active
9. **Application Services -** Services implemented by the application to be installed and executed within the platform

Applications, must follow some specific structure, in order to function and interact with the platform. Two classes must be available and implemented within a application:

- **Main Application Class -** State the implementations of all the functions to be executed both as Vehicle and RSU and designated at **Vehicle-functions** and **RSU-functions** fields of the configuration file.
- **Listener Class -** State the implementation of the dispatchers that will handle the information received by the subscribed services. Handlers for **SystemMessages** and **InterNodeCommunication** system services are implemented by default in order to get information sent by the platform for intern management and messages sent from the same application running in another node of the network. Inter-application messaging is not supported.

In order to demonstrate how applications can easily be developed using the VADM platform by using the supplied API and structure two (2) application examples, one for a safety application and other for a comfort-related application, will now be depicted:

4.6.1 Application Example: RequestAssistance

RequestAssistance application requests for emergency vehicles dispatch to the location of an accident by sending the vehicle position to the closest RSU. The RSU will then send a remote request to the Emergency Vehicle central that will reply with the emergency vehicle's ETA information that will then be sent to the vehicle that made the request in the first place.

RequestAssistance XML configuration file is depicted in listing 4.19 and it lists the services and functions than should be executed wether the node is a RSU or a vehicle.

Listing 4.19: RequestAssistance Application XML configuration file

```
<vnetplat>
        <application>
                <name>RequestAssistance</name>
                <type>Safety</type>
                <class>Assistance</class>
                <version>1.0</version>
                <description>Detects the change in the platform status and request for
                    assistance from a centralized service</description>
                <vehicle-functions>requestAssistanceVehicle;</vehicle-functions>
                            <rsu-functions>receiveAndRequestAssistance;</rsu-functions>
                <Vehicle-platformServices>GetPlatformStatus;GetNodeInformation</Vehicle-
                    platformServices>
                <RSU-platformServices>GetNodeInformation;</RSU-platformServices>
                <applicationServices>;</applicationServices>
        </application>
</vnetplat>
```

The several parts of the application will now be depicted and further explained:

Listing 4.20: RequestAssistance Main Application Class pseudo-code

```
class Application:
        def receiveAndRequestAssistance():
                startEVserverConnection()
                waitForRequests()

        def requestAssistanceVehicle():
                while(true):
                        if(currentStatus == "Emergency" or "Accident"):
                                message$_{J2735ALC}$.setDestination(broadcast)
                                message$_{J2735ALC}$.setMessage(RequestAssistance,
                                    currentCoordinates)
                                while(ACK=="Null"):
                                        publishInformation(message$_{J2735ALC}$)
                                        sleep(1)
```

The Main Application Class, where the functions to be executed when the application starts, depending on the node type, are defined. In this application, we only define 2 functions, one to be executed when the node is a RSU (*receiveAndRequestAssistance()*) and another one to be executed when its a vehicle (*requestAssistanceVehicle()*).

receiveAndRequestAssistance() will start the connection with the Emergency Vehicle's Central server and then wait for requests for assistance from vehicles.

requestAssistanceVehicle() will check the current VADM status for changes and if the status is changed to a *Emergency or Accident* status it triggers a broadcast request with the objective of reaching the nearby RSU. The application will keep on broadcasting at 1 Hz the request until receiving a ACK from a nearby RSU.

Listing 4.21: RequestAssistance Listener Class pseudo-code

```
class Listener:
        def InterNodeCommunication(receivedMessage):
                if(GetNodeType()=="RSU"):
                        sourceIP = receivedMessage$_{J2735ALC}$.getSource()
                        sourceCoordinates = receivedMessage$_{J2735ALC}$.getCoordinates()
                        ETA = requestETAtoServer(sourceCoordinates)
                        response$_{J2735ALC}$.setMessage(ETA)
                        response$_{J2735ALC}$.setDestination(sourceIP)
                        publishInformation(response$_{J2735BSM}$)

                if(GetNodeType()=="Vehicle"):
                        if(broadcaster=="me"):
                                if(receivedMessage$_{J2735ALC}$.getSource()=="RSU"):
                                        if(ACK == "Not␣Received" and receivedMessage$_{J2735ALC}$
                                            .getMessage() == "ACK"):
                                                ACK = Received
                                        elif(ACK == "Received" and receivedMessage$_{J2735ALC}$.
                                            getMessage() == "ETA"):
                                                displayETA()
                                else:
                                        checkTimeout()
                        else():
                                publishInformation(receivedMessage$_{J2735BSM}$)

        def GetPlatformStatus(serviceMessage):
                storeVADMstatus(serviceMessage)

        def GetNodeInformation(serviceMessage):
                storeCurrentCoordinates(serviceMessage.getCoordinates())
```

The Listener Class depicts the implementation of the subscribed services listeners. Every time a application receives a message from a subscribed service it executes the listener implementation. For the *RequestAssistance* application, three (3) services were subscribed, meaning that 3 listeners were defined as well.

InterNodeCommunication, listener of all the messages received from the same application running in other nodes of the network, will start by checking the current node type. If the node is a RSU, it means that a request for assistance was sent by a vehicle and therefore it sends the request to the Emergency Vehicles central server that will reply with the ETA to the vehicle location. Then the RSU will send back the ETA message to the vehicle that started the *RequestAssistance* application.

GetPlatformStatus and *GetNodeInformation* will receive and store information published by the servers regarding the current platform status and GPS coordinates.

Listing 4.22: RequestAssistance Listener Class pseudo-code

```
def main():
        registerApplication(Name,XMLPath)
        NodeType = getNodeType()
        availableServices = requestServicesList()
        subscribeNeededServices(XMLpath, availableServices)
        Listener.start()
        Application.run(NodeType)
```

The main class of the application starts by registering the application in the VADM platform. Then requests the node type so that the application know which functions to execute and which services to subscribe. After that, it subscribes the needed services and starts both the Main Application Class and the Listener Class.

By using the API primitives made available by the VADM platform, was very simple to develop a application that automatically requests Emergency Vehicles to a accident location that can be installed in both vehicles and RSUs without any need of adaptation or any long development process.

4.7 Chapter Summary

In this section we presented the implementation details and options of the development process of the VADM platform. We started by developing a support platform with basic features to supply a additional functionalities to our work that was outside of the scope of this thesis. Then the several design decisions and implementation options were detailed.

In all author's choices, taking into account that a easy-to-use, configurable and agnostic platform should be the output of this work. We highlight the following features of our solution:

1. Several of the characteristics and behaviors of the platform and applications are loaded from XML files, delivering a degree of configuration to the whole solution
2. By only having two (2) points of contact with the support platform it is fairly easy to change the underlying support technologies in use.
3. The behavior of the several internal modules of the VADM platform can be adapted to cope with the platform needs or as a response to a system optimization therefore delivering an adaptable platform.
4. A set of well define API primitives is offered to applications so that they can make use of the VADM platform functionalities and be developed having an agnostic approach on both the support communication technologies and protocols and the current context of execution.
5. Modules were developed in a decoupled approach, having well defined functions to communicate within themselves. With this, is it possible to easily update a specific module implementation without prejudicing the remaining of the VADM platform functionality.
6. A standard for vehicular-oriented application messaging was used, opening the path for future compatibility with other systems and solutions.

By using the VADM platform and the described interaction mechanisms, a simple application-structure can defined, facilitating the easy-development of applications to be used in vehicular-network environments while being totally agnostic of the scenario in which is being executed.

On the Chapter 5 we will evaluate our solution and present results.

5 Evaluation

In order to evaluate the developed solution, several tests were performed. All tests were performed in a real scenario with a controlled environment with the objective to assess both the platform performance, functionality and viability. The following sections detail the several steps and decisions towards evaluating the proposed solution, starting by the definition of the developed applications and used equipment and software in section 5.1 and 5.3. Test objectives and goals are defined in section 5.2. After that the test scenarios are explained in section 5.4 followed by the results presentation and discussion in section 5.5.

5.1 Developed Applications

A set of applications were created in order to be used in the controlled scenario in a situation where a vehicle has an accident and triggers the usage of a application that will request for Emergency Vehicles dispatch to its location.

Five (5) applications were developed, each one with different behaviors and network requirements:

1. **RequestAssistance -** Application that every time a vehicle set its status to *Accident*[1], triggers a broadcast request to the nearby RSU in order to have emergency vehicles sent to its location.
2. **VehicleStream -** A vehicle streams video in a broadcast fashion to the RSU nearby. RSU answers with an ACK for every video packet received. Video is streamed at 400 kbps.
3. **Chat -** Provides bidirectional text communication with the surrounding vehicles through the RSU.

[1] Note that the status of a vehicle can be changed in 3 different fashions as mentioned in section 3.5. In this test, the method User-Input was used.

4. **Points of Interest (POI) -** The RSU broadcasts information regarding the nearby POI (Restaurants, Hotels and Gas Station locations) and other relevant local information, such as the maximum speed limit on that road. POI messages are broadcasted at 1 Hz. When on a vehicle compares the current speed with the received information regarding the maximum speed allowed on that road and displays warning if exceeding such velocity. This application creates one service and uses another default service, depending on the node type:

   - **RSU -** Creates a service to pinpoint the distance to the nearby hotels and restaurants through a remote database call.
   - **Vehicle -** Uses system service to get current vehicle speed.

5. **RSUStream -** RSU streams video to all the surrounding vehicles in a broadcast fashion at 300 kbps. Vehicle answers with a ACK for every video packet received.

The aforementioned applications cover situations in which the vehicle is the main source of both high and medium traffic volume generation and also situations where the RSU, not the station in need of using the safety application, is the most relevant traffic volume generator of the network. Table 5.1 depicts the application type and class, following the classification adopted in this document, already mentioned in section 2.1.7.

Table 5.1: Developed Application's classification

	Application Type	**Application Class**
RequestAssistance	Safety	Assistance
VehicleStream	Comfort	Entertainment
Chat	Comfort	Entertainment
POI	Comfort	Information
RSUStream	Comfort	Entertainment

Table 5.2 shows the default services available on the platform that are used by the developed applications. By analyzing this table it is possible to understand that although applications have different purposes, they end up requiring fairly the same services when being developed. When the needed services are not available they can create their own services, or compose existing ones.

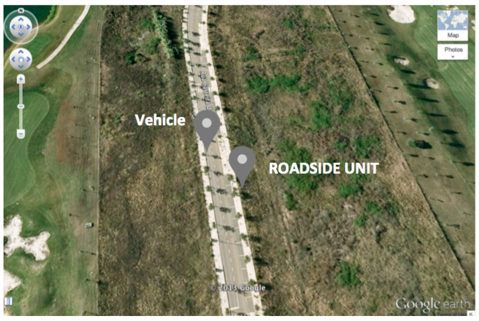
5.2 Tests Description

Several tests were performed in order to evaluate several aspects of the proposed platform such as:

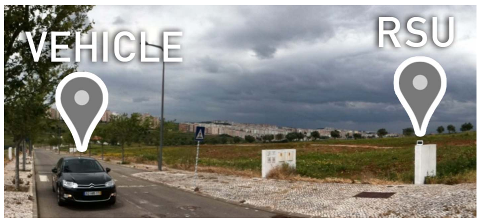
- Functionality and feasibility of the proposed platform for application development in vehicular environments.
- Application's performance impact of applying the several functionalities of the development platform.

Table 5.2: Platform services used by the developed applications

	GetNodeInformation	GetPlatformStatus	GetNodeType	GetNeighboursInfo	Creates Own Service
RequestAssistance	Yes	Yes	No	No	No
VehicleStream	No	Yes	No	No	No
Chat	No	Yes	Yes	No	No
POI	Yes	No	Yes	No	Yes
RSUStream	Yes	No	Yes	No	No



(a) Test-bed map location



(b) Test-bed environment

Figure 5.1: Test-bed location and overall position of the communication nodes

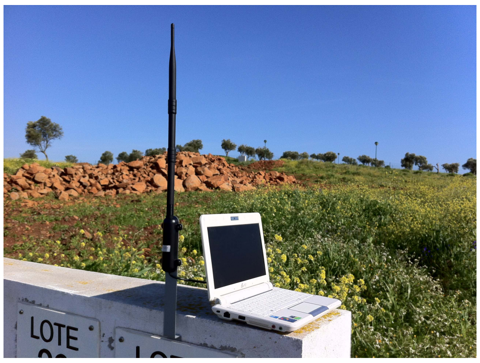
- Proposed platform performance

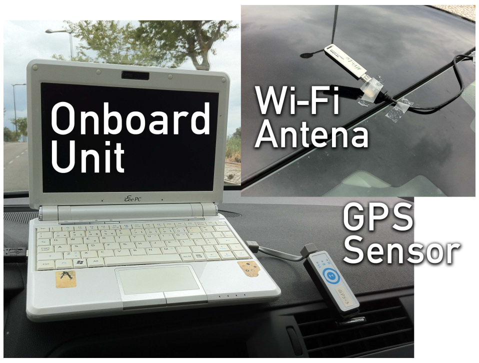
A controlled scenario was then prepared in order to evaluate and test the proposed solution focusing on achieving the aforementioned test objectives. The first scenario consisted of a simple setup of a Vehicle Station and a RSU placed at a distance of 20 meters of each other, in a road with low frequency of passing vehicles. During all the performed tests, a average frequency of 5 vehicles per minute was registered. The tests were performed during 5 week days, on the period of 1:00 PM to 7:00 PM, in the region of Taguspark, Oeiras, Portugal. Figure 5.1 depicts the location and placement of both stations in the controlled scenario test-bed.

5.3 Used Software and Equipment

Both node types, vehicle and roadside-unit were equipped with a ASUS eeePC equipment with Intel Atom running at 1600 MHz and 2 GB of RAM running a Lubuntu distribution with Linux kernel 3.5.5. This distribution was chosen due to its low system requirements facilitating the usage of all the developed work in systems with limited capacities. For communication purposes the vehicle was equipped with a SMCWUSBS-N3 802.11b/g/n wireless pen, while for the RSU was used a high-gain PowerLink Ultra High Power, both running IEEE 802.11n and with chipset chipset rt3070.

As sensors, the vehicle was equipped with:


(a) RSU node equipment



(b) Vehicle node equipment

Figure 5.2: Equipment used in both RSU and Vehicle nodes

- **Holux M-1200E GPS USB Dongle** with sensitivity up to -150 dBm and a L1 1575.42 MHz receiver, to determine vehicles position.
- **Soliport ELM 327 OBD2 diagnostic scanner** accessed by Bluetooth, to retrieve in-vehicle sensor information such as current speed and RPM, Fuel system status, Intake air temperature and others.

The developed platform was installed in both vehicle and RSU stations and configured using the XML configuration file of each node-type, as described in section 4.4.1. Listing 5.1 depicts one example of a platform configuration file.

Listing 5.1: Platform XML configuration file example

```
<vnetplat>
        <platform>
                <name>Application Development Platform 2013 - Vehicle 1</name>
                <type>Vehicle</type>
                <publishFrequency>1</publishFrequency>
                <sensingFrequency>1</sensingFrequency>
                <ID>53-20-QG</ID>
                <IP>192.168.0.1</IP>
                <version>1.0</version>
                <description>Platform Description</description>
                <defaultServices>
                        GetNeighboursInfo;
                        GetNodeType;
                        GetPlatformStatus;
                        GetNodeInformation;
                </defaultServices>
        </platform>
</vnetplat>
```

In order to gather information regarding the generated and received traffic volume as well as to interpret application's behavior in the several tests, TCPDump packet analyzer was used.

Figure 5.2 depicts the vehicle and RSU nodes equipment, as well as the equipment positioning for the performed tests.

5.4 Test Scenarios

The developed application sets were first tested in a scenario with one vehicle and one RSU. The controlled scenario situation consisted of a 30 seconds experience where at second 15 the vehicle has an accident and triggers the *RequestAssistance* application.

In order to speed the testing procedures, spare additional costs with gas refueling and avoid possible material damage in simulating an accident, we forced GPS and speed values to the Call Manager module, that assumed the data as received from the sensors provided by the support platform. The dummy data forced into the platform was retrieved from a one-time experience where the objective was only to retrieve the information gathered by the sensors. On the first 15 seconds of the test the vehicle is traveling at an average of 50 km/h and at the 15th second is suffers a sudden stop, staying still until the end of the test.

The developed applications were combined and executed in the controlled scenario 10 times. To evaluate the overall performance of the proposed solution, 2 different metrics were taken into account:

1. Generated and received traffic volume by each node.
2. Response-time of the *RequestAssistance* application.

The traffic volume was used to determine the average generated and received traffic on the network for the several applications on the several tested scenarios. With this it is possible to have a clear idea of the application's behavior and the impact that has on the RequestAssistance application performance.

ResquestAssistance response-time was used in order to determine how relative priority scheduling can affect safety-related services performance.

Applications were combined in such way, so that it would be possible to evaluate the situations where the node having the accident was the node producing the majority of the network traffic as well as the opposite situation, where another node was generating the vast majority of traffic. Therefore, the below mentioned application sets were developed and tested:

- Vehicle as generator of the majority of network traffic:
  - *RequestAssistance* + *VehicleStream*
  - *RequestAssistance* + *Chat*
  - *RequestAssistance* + *VehicleStream* + *Chat*
- RSU as generator of the majority of network traffic:
  - *RequestAssistance* + *POI*
  - *RequestAssistance* + *RSUStream*
  - *RequestAssistance* + *POI* + *RSUStream*

Also, the tests were grouped in such a way that it was possible to evaluate situations where there was a considerable amount of generated and received traffic value and situations where the generated and received traffic volume had lower values.

Three (3) different test-scenarios were considered for the tests in order to evaluate different parts of the architecture. Table 5.3 depicts the variations of the several test-scenarios.

Table 5.3: Test-scenarios architecture configuration variations

Scenario	ARM scheduling	CommManager scheduling	ESPD protocol
A	No	No	inactive
B	No	Yes	active
C	Yes	Yes	active

The first scenario aims to test the situation where scheduling is not used, simulating a situation where the fact of a vehicle being in a emergency or accident situation does not imply application prioritization schemes to work. Scenario B, by activating the scheduling of the CommManager, that aims to apply the priority rules to all received information based on the current state and application type, will let us comprehend how such scheduling for the received data affects the overall behavior of applications and services. Finally, scenario C represents the case where the full functionality of application flow control, by using all scheduling mechanisms, of the proposed solution is working. In the last 2 scenarios, the ESPD protocol is active, meaning that all vehicles or RSUs within the ROP will change their state to comply with the region status, normal situation, vehicle with emergency or vehicle(s) involved in a accident.

Note that, for this evaluation the scheduling decision for not priority traffic adopted was full discard without any caching of content.

5.5 Test Results Presentation and Discussion

We will now present the results obtained from the tests performed in our controlled scenario. Despite the fact that we used a simple architecture the results allowed us to draw valuable conclusions regarding the options we took on the development of our architecture.

5.5.1 Traffic Volume

Figures 5.3 and 5.4 depict the average traffic volume for the several test-scenarios with the different application-sets. The first one, details the applications where the vehicle was the main generator of traffic in the network while the second the situation where another node, the RSU, was the main traffic-generator of the network. Bear in mind that **Test Scenario A** represents the situation were no scheduling, messaging plausibility check or application priority rules are applied, **Test Scenario B** the situation where scheduling is applied only on the receiving of messages and the VADM status propagation protocol, ESPD is active and finally **Test Scenario C**, where the full functionality of the VADM platform is active.

The tests depicted in figure 5.3 represent the situation where the vehicle is the main source of traffic generation in the network and we will now detail each one of the plots:

- **5.3 (a) -** Since no scheduling is active no actions are made in application's behavior when a accident happens and safety-applications are functioning. One may see that the generated and received traffic for the remaining comfort-related applications was not altered.

- **5.3 (b) -** In this scenario the scheduler is only active to inbound traffic. Meaning that all the traffic was still generated to the network but the VADM platform discards the incoming traffic to spare computational resources. One may notice that traffic generated by comfort-related applications was still making is way to the network. Note that, this behavior applies to the remaining nodes that, by receiving the notification to change their status, also discard the receiving traffic.

- **5.3 (c) -** This plot represents the situation where all the scheduling mechanisms and ESPD protocol are active. As one may see, both the generation and receiving of traffic if stopped in the moment a accident is detected and such notification is spread to all nodes, therefore giving priority only to safety-related applications.

This results shown the situation where the vehicle was the node generating the most amount of traffic to the network. The larger percentage traffic generation was made by the Video stream application and despite the fact that the results of the received traffic for that application - remind that every time a RSU receives a Video Stream packet, responds with a ACK - are close to the generated traffic visually, the Y axis of the plot is on a logarithmic scale, therefore not depicting visually the percentage difference between generated and received traffic.

Figure 5.4 shows the situation where the main traffic generator node on the network was not the vehicle that had the accident. This situation tries to illustrate the effect of needing to share and warn the nodes in the vicinity that one vehicle is in a *Accident* or *Emergency* situation, and that all safety-related applications should have priority in both resources usage and network occupation. Note that the applications in use in this test scenario are different from the ones used in the previously described test scenario, since we needed to assess applications that had the RSU as the main traffic generator. We will now analyze in detail the results of each plot:

- **5.4 (a) -** This plot shows that, as the plot depicted in 5.3 (a), no scheduling is activated when the accident happens, fairly at second # 15.

- **5.4 (b) -** In this scenario, it is possible to see the effect of spreading the VADM status alteration need through the network, using the ESPD protocol. One may see that, after the vehicle have the accident, the ESPD message reached the RSU and it stopped all running comfort applications. However one may notice that in some situations the responses of the RequestAssistance application arrived only after the receiving of comfort-related applications information stopped. Also, please note that the traffic generation on the RSUStream application stopped only because it represents the ACKs that are sent for each packet received.

- **5.4 (c) -** This last plot, shows that when all the scheduling mechanisms are functioning, the both receiving and sending of applications are stopped, but in this situation, and comparing with the previous, one may notice the difference between the time that the vehicle stopped accepting the receiving of information and the time where there was a reduction on the received traffic. This is due to the fact that since there is a large amount of traffic being received, the communication technologies buffers take time to deliver all the buffered information. Nevertheless, it is possible to see that the number of cases where the RequestAssistance application received the final response was lower than the previous scenario. This is because, since the scheduling on the sending of information is applied, on the RSU side, after the status is altered, only the information from safety-related applications will be sent to the network.

Table 5.4 discloses and summarizes the traffic volume reduction results obtained by using the scheduling mechanisms and the full platform functionality. Note that the node used as reference is the vehicle triggering the safety application. Refer to Appendix 2 for the average traffic volume for all the combined application's sets used in the controlled scenario.

Table 5.4: Traffic volume reduction of the several application's sets by using VADM application level scheduling

Applications	**Traffic Volume Difference**			
	Scenario B		**Scenario C**	
	Generated	**Received**	**Generated**	**Received**
RequestAssistance + VehicleStream	-0.111 %	-55.956 %	-65.473 %	-56.784 %
RequestAssistance + Chat	-14.463 %	-48.804 %	-70.925%	-53.575 %
RequestAssistance + VehicleStream + Chat	-1.508%	-55.825 %	-64.213 %	-55.971 %
RequestAssistance + POI	-8.207 %	-49.119 %	-16.644 %	-61.818 %
RequestAssistance + RSUStream	-58.480 %	-4.671 %	-58.222 %	-46.460 %
RequestAssistance + RSUStream + POI	-56.407 %	-8.228 %	-55.449 %	-42.932 %

The results regarding the traffic volume generated and received by the applications in the several situations shown that:

1. By using the scheduling based on the VADM platform status, it is possible to reduce the non-relevant traffic in the network for safety situations.

2. When scheduling is only applied on the receiving data, at the CommManager, there is still non-relevant application data being transmitted to the network. Therefore, the wireless channel is still being flooded with non-relevant information that, by comparing with the results presented previously, prejudice the safety-application response-time for situations where the node using that application is not the main network-traffic generator.

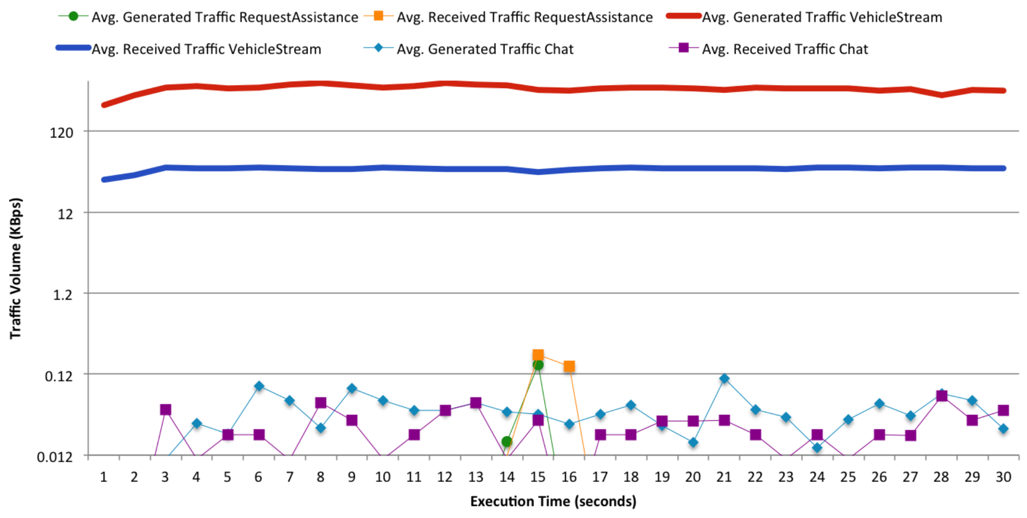
3. The resulting reduction in the generated traffic is mostly due to the discard of information produced by comfort applications that in a *Accident* or *Emergency* scenario, have less priority in access the platform and the available means.

5.5.2 RequestAssistance Response-Time

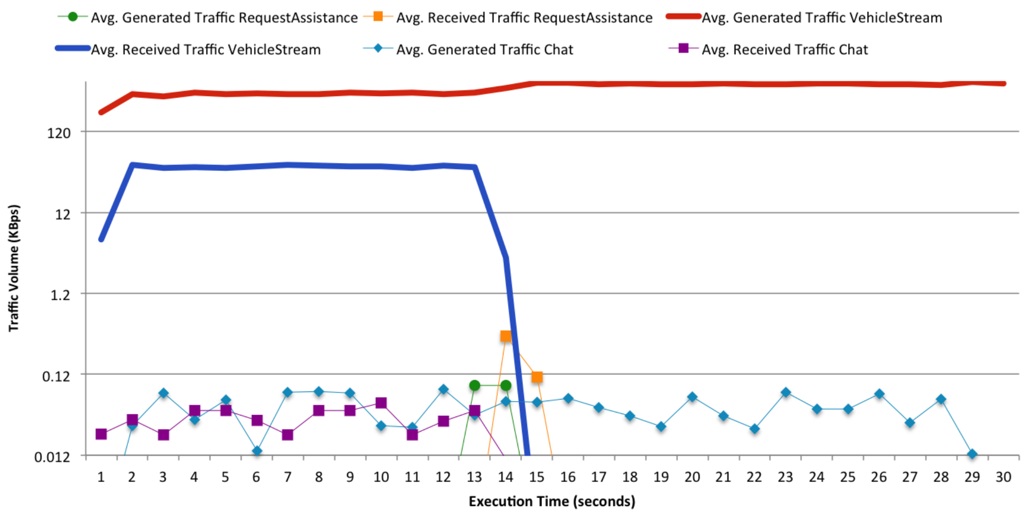
Figure 5.5 depicts the response-time of the *RequestAssistance* when in combination with several other applications, as described previously in section 5.4.

By analyzing the results regarding the response-time of the *RequestAssistance* application in the various situations, one can conclude that:

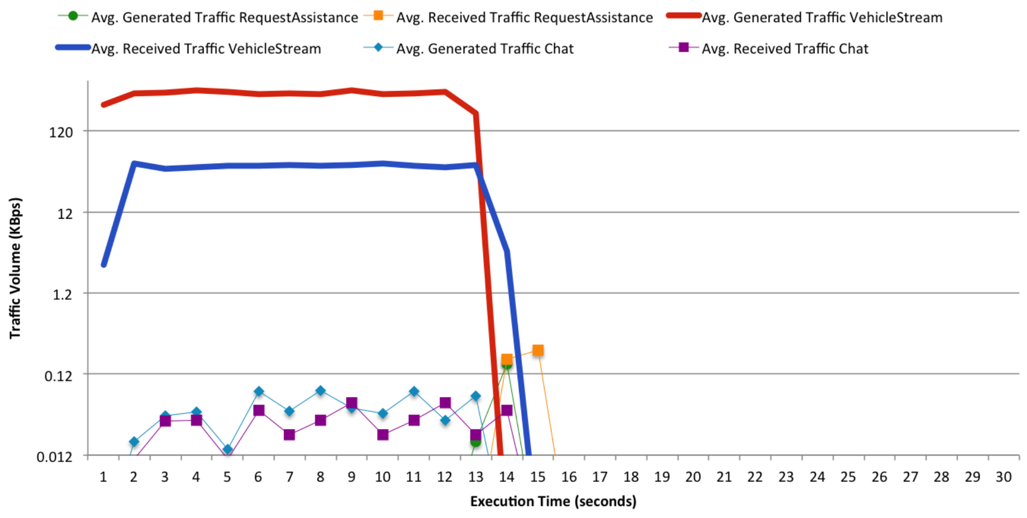
1. When no scheduling mechanisms are used in safety situations, the safety-application response-time is overly higher for all the application sets combinations.



(a) Traffic volume for Test Scenario A

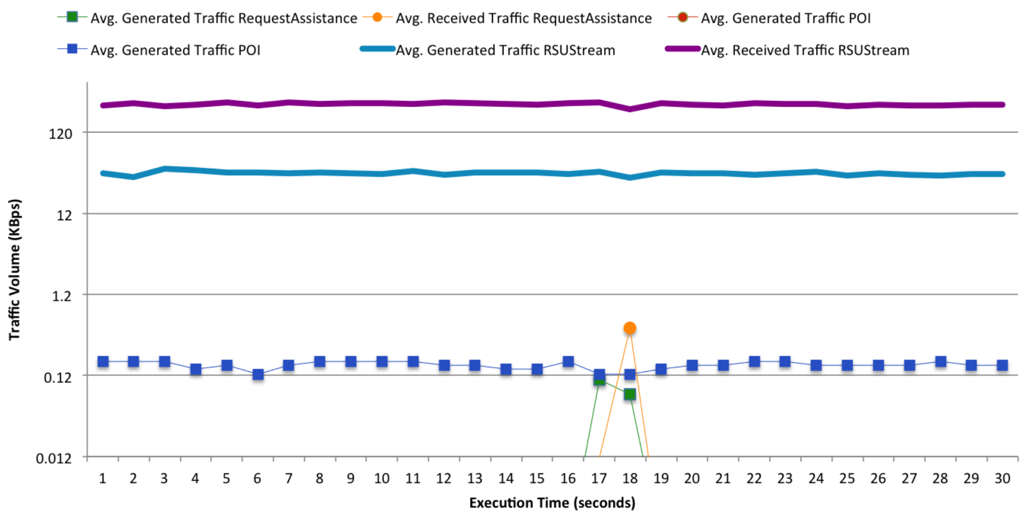


(b) Traffic volume for Test Scenario B

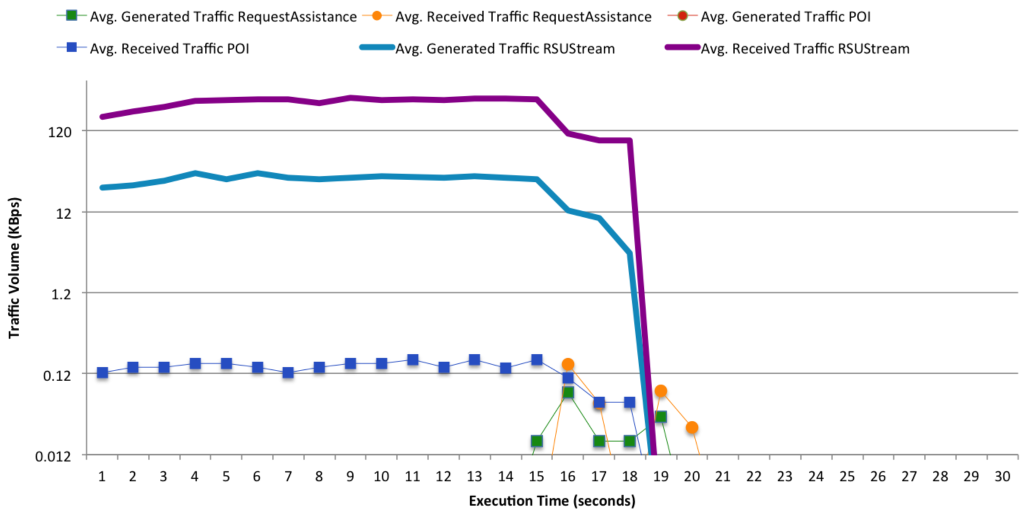


(c) Traffic volume for Test Scenario C

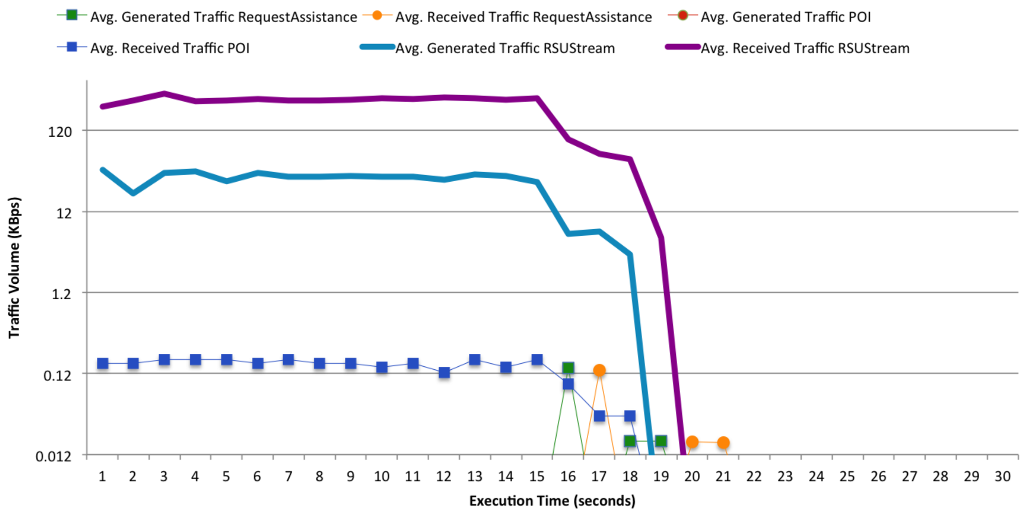
Figure 5.3: Average traffic volume generated and received for tests using applications *RequestAssistance*, *VehicleStream* and *Chat*



(a) Traffic volume for Test Scenario A



(b) Traffic volume for Test Scenario B



(c) Traffic volume for Test Scenario C

Figure 5.4: Average traffic volume generated and received for tests using applications *RequestAssistance*, *RSUStream* and *POI*

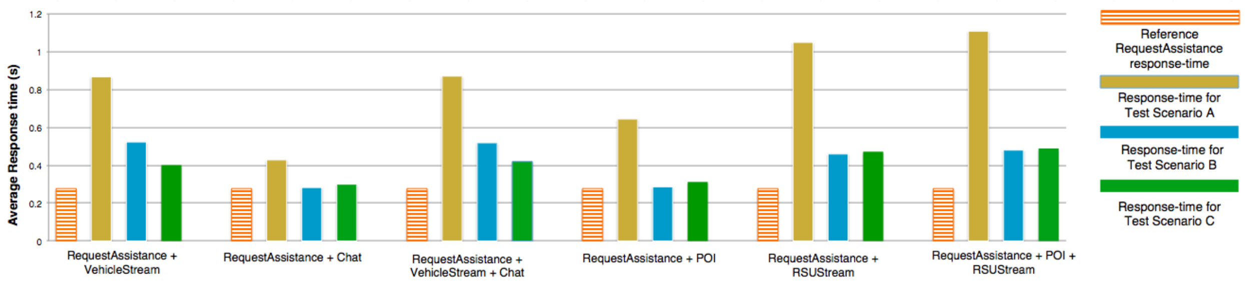


Figure 5.5: RequestAssistance average response-time comparison with several application's sets

2. When the CommManager scheduling mechanism is used, meaning that all received data that is not relevant for the current node status is discarded, the response-time of the safety-application is reduced.

3. When the full feature of the solution is functioning, meaning that all the scheduling mechanisms and VADM status sharing protocol are active, the safety-application response-time is only enhanced when the node in need of using such safety service, is also the node generating the most considerable amount of traffic in the network.

4. When all scheduling mechanisms are activated, for the situations where the node using the safety-application is not the main generator of network-traffic, the response-time slightly increases due to wireless channel being occupied with applications with traffic volume generation, and therefore, the information to change the platform state and with that, start the scheduling procedures, takes more time reaching the other node.

5. Also, it is possible to conclude that applications that have low requirements regarding the bandwidth, have a response-time similar to the response-time used as benchmark due to its low traffic volume generation.

One should bear in mind that the results here presented, are obtained by using as a scheduling policy that discards without caching or storing all the non-relevant content. Other policies may be applied, but the results show that when no non-relevant information is sent to the network, the efficiency of the safety-applications, or other relevant services running, is improved.

5.6 Chapter Summary

On this chapter we described the developed test-bed in order to evaluate both the functionality, performance and improvements achieved with the proposed solution. Despite the simple network topology, several extrapolations for more complex situations can be undertaken from the presented results due to the extreme-case application sets used.

The main focus of the evaluation tests performed was pointing in the direction of evaluating the performance of safety-applications in safety-related situations such as emergency or accident scenarios. Such performance was evaluated with the combination of several other types of applications that could be in use by the several participants of the vehicular network, mostly applications that are not relevant in safety-related scenarios.

The results shown that, by using our solution, the performance of safety-applications is increased[2] up into 55.91% for safety-related situations, by using a priority-based mechanisms that takes into account the relevance of one or more applications for the current road or traveling scenarios.

Even by only using two (2) nodes in the network, we created a complex and varied situation of interaction between vehicles creating extreme cases of network and platform occupation with the video streaming applications. With this, we believe that the results here presented are valid for scenarios with more nodes in the network.

Comfort-applications were not developed having in mind the current execution state, meaning that they are fully agnostic of the scenario and situation in where they are executing. All the application management and prioritization is performed by the our platform.

Regarding the tests, some hurdles were felt while performing the field-test experiments, being worth of mention:

1. Despite the fact that all the execution and logging is automated, performing tests still needs human interaction to launch all the scripts in time and reset parameters on the several iterations of the tests. At least 2 individuals were needed to perform the tests, one in the vehicle and one in the RSU. Both individuals were also responsible for taking notes on external events that happened while during the tests (e.g. note the number and type of passing vehicles).

2. By having persons running the tests, those were subject to human errors that were only minimized when already prepared individuals were helping on the tests

3. Laptops battery-life lasted an average 4 hours on the vehicle node and roughly 2 and a half hours on the RSU due to the high-power consumption antenna in use. Despite the fact that 1 extra battery was available, tests took longer in the process of changing the battery of the RSU node and going back to charge all batteries when all were drained out of energy.

4. Some tests were delayed due to unpredictable rainy weather conditions.

Also this experiment was based in using adapted application-level implementations of a support platform and traditional off-the-shelf equipment. Despite the fact that the developed support platform supported the usage of 2 interfaces to simulate different channels for comfort and safety-related applications as available in the IEEE 802.11p standard, for logistic reasons we were only able to test using one interface in each node, being the multiplexing of applications done by sending information to different transport layer port ranges. However, since the available bandwidth was of 150 Mbit/s, it was never fully occupied, still making the results here presented relevant.

Of course, since Vehicular-network oriented communication technologies such as IEEE802.11p supply different channels for both comfort and safety-applications, results should show some improvements when in use of several applications. Also 802.11p supplies some low-level priority mechanisms

[2]Safety-applications performance improvement is measured by taking into account the response-time reduction.

for applications. Nevertheless, such supporting technologies do not bear in mind the current execution situation, meaning that all the scheduling mechanisms and architecture of the proposed solution remain relevant.

6 Conclusions and Future Work

In this thesis, our goal was to facilitate the development of applications to be used in vehicular communication scenarios. Intelligent Transportation Systems (ITS) had its inception and expansion based on the development of several communication technologies and protocols of vehicular networks. With such, the number of possible services and applications that could be developed for such systems is numerous. Despite the fact that standardization for a application development platform is available, no well-defined implementation or support architecture structure is available. With that, the application development process becomes dependent of the developers and manufacturers design options, creating barriers to the easy and widespread application development. Another problem found is that applications are heavily dependent on the scenario in which they are executing and some end up being more plausible than others depending upon the context in which the vehicle is currently.

Having the above in mind, we started our work by evaluating the available standardization initiatives and projects as well as the vehicular-oriented technologies available. Also, we focused on understanding the several applications and services that can be used in vehicular-communication scenarios, evaluating their requirements and common characteristics.

After the primary evaluation of the State of the Art, we started defining our architecture, that by coping with the available standardization work done, extended its functionalities in order to create platform to facilitate the easy development of applications. Therefore the main contributions of this work were:

1. A API for an agnostic application-development process that decouple applications implementation from both the underlying platform and also the context in which the vehicle is currently, to be used freely by developers.

2. Development of a modular and decoupled architecture to facilitate the integration between ap-

plications and vehicular-communication technologies and systems.

3. Development of a platform that can organically adapt itself and pose adaptations in the network towards giving priority to applications and services whose goal is the overall improvement of road users safety.

4. Extension and reuse of the functionalities offered by the Facilities Layer of the standardized ITS reference architecture.

The authors detailed thoroughly the several aspects of both architecture design and implementation process of the proposed platform, that presents a solution to be used in both vehicles and RSUs for fostering the vehicular-application development process. Since we based the full development process in the available standards and in open-source solution, creating a open-source solution as well, we created a modular solution that is mainly important because of the concepts and basic structure, opening now the possibility of future enhancements of each module. The VADM architecture suffered several changes during the development process of this thesis, and several of the decisions presented in this document are result of extensive tries towards achieving the solution that coped with the most of the problems we were addressing. One of the important changes that made its way until the final architecture was the need to creation of a protocol that spread the need of platform status adaptation in a epidemic fashion.

After the design and development process, we evaluated our solution in a controlled scenario where it was possible to see that, by using our platform to coordinate application's execution having as priority safety applications, the performance of the later was generally improved. The scenario aimed to simulate a situation where a vehicle has an accident and there is a sudden need of giving priority to safety-related applications, in particular an application that is going to request for Emergency Vehicles. We developed a set of applications with different requirements and evaluated the impact of each one of them. Results shown that by using the several scheduling mechanisms and also the developed ESPD protocol for sharing of the platform status within a region of plausibility, the performance of the safety-related application was enhanced largely. This posed a positive turn on the development process of this thesis since the evaluation performed, confirmed the validity and plausibility of the design options we have made when developing the architecture.

Due to several logistic hurdles found while doing the tests and evaluating the platform, one was not possible to test features of the platform such as the adaptation of modules behavior, in particular the Call Manager getting of data frequency and Services publishing of information frequency. Nevertheless, the results of the platform, without any changes on the modules behavior was still very positive and any change and improvement in modules behavior will most likely enhance and never prejudice the results achieved.

With our architecture we also created the possibility of future extensions and enhancements of the several parts that can help developing new approaches towards facilitating the application development process while always giving priority to applications and services that can help enhancing the overall security and safety of road users.

As future work, we would like to extend the controlled scenarios tests to a more complex network topology until reaching the stage of taking the tests to a real-scenario in several communicating nodes and high degree of vehicle's behavior unpredictability. Moreover, we would like to test the platform with using vehicular-communication oriented technologies such as the IEEE 802.11p that offers additional features that can enhance and improve the results achieved by using our solution. After such extensive testings, one should start addressing issues such as security, privacy and scalability that, when developed and implemented, can taylor the possibility for our work to reach an extensive and general use by both application developers, manufacturers and road users.

Bibliography

[1] A. Amanna, “Overview of IntelliDrive / Vehicle Infrastructure Integration (VII),” no. Vii, 2009.

[2] A. Dahiya and R. K. Chauhan, “A Comparative study of MANET and VANET Environment,” *DBMS*, vol. 2, no. 7, pp. 87–92, 2010.

[3] M. Kafsi, P. Papadimitratos, O. Dousse, T. Alpcan, and J.-p. Hubaux, “VANET Connectivity Analysis,” *IEEE Workshop on Automotive Networking and Applications*, 2008.

[4] J. Jakubiak and Y. Koucheryavy, “State of the Art and Research Challenges for VANETs,” *5th IEEE Consumer Communications and Networking Conference*, pp. 912–916, 2008.

[5] A. Festag, G. Noecker, M. Strassberger, A. Lübke, and B. Bochow, “NoW – Network on Wheels: Project Objectives, Technology and Achievements,” *International Workshop on Intelligent Transportation (WIT)*, no. March, pp. 211–216, 2008.

[6] T. Kosch, I. Kulp, M. Bechler, M. Strassberger, B. Weyl, and B. M. W. Group, “Communication Architecture for Cooperative Systems in Europe,” *IEEE Communications Magazine*, no. May, 2009.

[7] ETSI EN 302 665, “Intelligent Transport Systems (ITS); Communications Architecture,” pp. 1–44, 2010.

[8] P. Papadimitratos, A. La Fortelle, K. Evenssen, R. Brignolo, and S. Cosenza, “Vehicular communication systems: Enabling technologies, applications, and future outlook on intelligent transportation,” *IEEE Communications Magazine*, vol. 47, no. 11, pp. 84–95, Nov. 2009.

[9] Y. Qian and N. Moayeri, “Design of Secure and Application-Oriented VANETs,” *VTC Spring 2008 - IEEE Vehicular Technology Conference*, pp. 2794–2799, May 2008.

[10] L. S. MIHAIL and M. KIHL, “Inter-vehicle Communication Systems: A Survey,” *IEEE Communications Surveys & Tutorials*, vol. 21, no. 1, pp. 111–111, Jun. 2008.

[11] M. Jerbi, P. Marlier, and S. M. Senouci, “Experimental Assessment of V2V and I2V Communications,” *IEEE Internatonal Conference on Mobile Adhoc and Sensor Systems*, pp. 1–6, Oct. 2007.

[12] K. Dar, M. Bakhouya, J. Gaber, and M. Wack, “Wireless Communication Technologies for ITS Applications,” *IEEE Communications Magazine*, no. May, pp. 156–162, 2010.

[13] M. Sawahashi, Y. Kishiyama, H. Taoka, M. Tanno, and T. Nakamura, “Broadband radio access: LTE and LTE-advanced,” *2009 International Symposium on Intelligent Signal Processing and Communication Systems (ISPACS)*, no. Ispacs, pp. 224–227, Dec. 2009. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5383862

[14] M. Wellens, B. Westphal, and M. Petri, “Performance Evaluation of IEEE 802 . 11-based WLANs in Vehicular Scenarios,” *Performance Evaluation*, pp. 1167–1171, 2007.

[15] R. A. Uzcátegui and G. Acosta-Marum, “WAVE : A Tutorial,” no. May, pp. 126–133, 2009.

[16] G. Rémy, S.-m. Senouci, F. Jan, and Y. Gourhant, “LTE4V2X : LTE for a Centralized VANET Organization,” *IEEE Globlecom*, pp. 0–5, 2011.

[17] R. Guillaume, S.-m. Senouci, and Y. Gourhant, “LTE4V2X - Impact of High Mobility in Highway Scenarios,” 2011.

[18] S. Eichler, C. Networks, and T. U. München, “Performance Evaluation of the IEEE 802 . 11p WAVE Communication Standard,” pp. 2199–2203, 2007.

[19] T. I. M. O. K. Osch, C. H. J. A. Dler, S. T. E. Ichler, and C. H. S. Chroth, “The Scalability Problem of Vehicular Ad Hoc Networks and How to Solve IT,” no. October, pp. 22–28, 2006.

[20] S. Gr and M. Petri, “Performance Evaluation of IEEE 1609 WAVE and IEEE 802 . 11p for Vehicular Communications,” pp. 344–348, 2010.

[21] H. Hartenstein and K. Labertaux, “VANET - Vehicular Applications and Inter-Networking Technologies,” 2010.

[22] Y. Toor, P. Mühlethaler, A. Laouiti, and A. de La Fortelle, “Vehicle Ad Hoc Networks: Applications and Related Technical Issues,” pp. 74–88, 2008.

[23] R. Karim and C. Science, “VANET : Superior System for Content Distribution in Vehicular Network Applications,” pp. 1–8, 2008.

[24] K. Ioannis and P. Poulicos, “A Map Matching Algorithm for Car Navigation Systems with GPS Input,” pp. 1–4, 2007.

[25] R. Simmons, B. Browning, and V. Sadekar, “Learning to Predict Driver Route and Destination Intent,” *IEEE Intelligent Transportation Systems Conference*, pp. 127–132, 2006.

[26] T. Mishra, D. Garg, and M. M. Gore, “A Publish/Subscribe Communication Infrastructure for VANET Applications,” *IEEE Workshops of International Conference on Advanced Information Networking and Applications*, pp. 442–446, Mar. 2011.

[27] F. Soldo, C. Casetti, C.-f. Chiasserini, P. Torino, and P. Chaparro, “Streaming Media Distribution in VANETs,” *IEEE Globlecom*, pp. 1–6, 2008.

[28] N.-W. Lo and H.-C. Tsai, “Illusion Attack on VANET Applications - A Message Plausibility Problem,” *IEEE Globecom Workshops*, pp. 1–8, Nov. 2007.

[29] D. Kelley, “DSRC Implementation Guide A guide to users of SAE J2735 message sets over DSRC.”

[30] D. Mundy and D. W. Chadwick, “An XML Alternative for Performance and Security: ASN.1,” *It Professional*, vol. 6, no. 1, 2004. [Online]. Available: http://dx.doi.org/10.1109/MITP.2004.1265540

[31] C. L. Robinson, D. Caveney, L. Caminiti, G. Baliga, K. Laberteaux, and P. R. Kumar, “Efficient Message Composition and Coding for Cooperative Vehicular Safety Applications,” pp. 3244–3255, 2007. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=4358481&tag=1

[32] J. Li, J. Jannotti, D. S. J. De Couto, D. R. Karger, and R. Morris, “A scalable location service for geographic ad hoc routing,” *Proceedings of the 6th annual international conference on Mobile computing and networking - MobiCom ’00*, pp. 120–130, 2000. [Online]. Available: http://portal.acm.org/citation.cfm?doid=345910.345931

[33] K. Michael, “A Reactive Location Service for Mobile Ad Hoc Networks,” 2002.

[34] W. Kieß, H. Füß ler, J. Widmer, and M. Mauve, “Hierarchical location service for mobile ad-hoc networks,” *ACM SIGMOBILE Mobile Computing and Communications Review*, vol. 8, no. 4, pp. 47–58, Oct. 2004. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1052871.1052875

[35] T. M. Bohnert, S. A. P. Ag, and M. Schulze, “PRE-DRIVE C2X Definition of PRE-DRIVE C2X / COMeSafety architecture framework,” 2008.

[36] C. Oh, E. Jeong, K. Kang, and Y. Kang, “Hazardous Driving Event Detection and Analysis System in Vehicular Networks (HEAVEN): Methodology and Field Implementation,” vol. TRB 2013 A, pp. 1–18, 2013.

[37] T. J. Triggs and W. G. Harris, “Reaction time of drivers to road stimuli,” no. June 1982.

[38] N. Vaidya, X. Yang, J. Liu, and F. Zhao, “A vehicle-to-vehicle communication protocol for cooperative collision warning,” *The First Annual International Conference on Mobile and Ubiquitous Systems: Networking and Services, 2004. MOBIQUITOUS 2004.*, pp. 114–123. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1331717

[39] M. Torrent-Moreno, P. Santi, and H. Hartenstein, “Fair Sharing of Bandwidth in VANETs,” *Vehicular Ad-hoc Networks*, 2005.

[40] D. Jiang and L. Delgrossi, “IEEE 802 . 11p : Towards an International Standard for Wireless Access in Vehicular Environments,” pp. 2036–2040, 2008.

A

.1 Technologies and Applications Characterizations

Table 1: Wireless Communication Technologies characteristics and features

Access Technology	Communication Mode	Directionality	Latency	Data Rate	Range	Transmission Mode	Mobility	Operating Band
GPRS / UMTS	V2I and V2V^{i}	Both-ways	1.5 - 3.5 sec	80 - 384 kb/s	10 km	Uni / Geo	Yes	0.8 - 1.9 GHz
LTE	V2I and V2V^{i}	Both-ways	5 ms	300 Mb/s	∼ 1 km	Uni / Geo	Yes	800 - 2600 MHz
WiMAX	I2V	Both-ways	∼110 ms	1 - 32 Mb/s	15 km	Uni / Geo	Yes	5.x GHz
DVB/DAB	V2I and V2V^{d}	One-way	10 - 30 sec	∼1 Mb/s	40 km	Broad	Yes	6 - 8 GHz
WLAN	V2I and V2V^{d}	Both-ways	∼46ms	54 - 600 Mb/s	250 m	Uni	Limited	2.4 - 5.2 GHz
MMWAVE	V2I and V2V^{d}	Both-ways	∼150μs	∼1 Gb/s	∼ 10 m	Uni	Limited	60 - 64 GHz
Infrared	V2I and V2V^{d}	Both-ways	Very-low	∼1 Mb/s	∼ 10 m	Uni	No	2.6 GHz
ZigBee	V2V^{d}	Both-ways	∼16 ms	20 - 250 kb/s	∼ 100 m	Uni	Yes	2.4 - 2.5 GHz
Bluetooth	V2I	Both-ways	∼100 ms	1 - 3 Mb/s	∼ 10 m	Uni	Limited	2.4 GHz
DSRC/WAVE	V2I and V2V^{d}	Both-ways	200μs	∼6 Mb/s	∼ 1 km	Uni	Yes	5.8 - 5.9 GHz
CALM M5	V2I and V2V^{d}	Both-ways	200μs	∼6 Mb/s	∼ 1 km	Uni	Yes	5 - 6 GHz

.2 J2735 Pyasn.1 Implementation and ASN.1 defition

Listing 1: J2735 ASN.1 Notation and Pyasn1 implementation

```
# DSRCmsgID ::= ENUMERATED
class DSRCmsgID(univ.Enumerated):
        namedValues = namedval.NamedValues(
                ('reserved',0),
                ('alaCarteMessage',1),
                ('basicSafetyMessage',2),
                ('basicSafetyMessageVerbose',3),
                ('commonSafetyRequest',4),
                ('emergencyVehicleAlert',5),
                ('intersectionCollisionAlert',6),
                ('mapData',7),
                ('nmeaCorrections',8),
                ('probeDataManagement',9),
                ('probeVehicleData',10),
                ('roadSideAlert',11),
                ('rtcmCorrections',12),
                ('signalPhaseAndTimingMessage',13),
                ('signalRequestMessage',14),
                ('signalStatusMessage',15),
                ('travelerInformation',16)
                )
        subtypeSpec = univ.Enumerated.subtypeSpec + constraint.SingleValueConstraint
            (0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16)

# MsgCount ::= INTEGER (0..127)
class MsgCount(univ.Integer):
        subtypeSpec = univ.Integer.subtypeSpec + constraint.ValueRangeConstraint(0,127)

# TemporaryID ::= OCTET STRING (SIZE(4))
class TemporaryID(univ.OctetString):
        subtypeSpec = constraint.ValueSizeConstraint(4,4)

# DSecond ::= INTEGER (0..65535)
class DSecond(univ.Integer):
        subtypeSpec = univ.Integer.subtypeSpec + constraint.ValueRangeConstraint(0,65535)

# Latitude ::= INTEGER (-900000000..900000001)
class Latitude(univ.Integer):
        subtypeSpec = univ.Integer.subtypeSpec + constraint.ValueRangeConstraint
            (-900000000,900000001)

# Longitude ::= INTEGER (-1800000000..1800000001)
class Longitude(univ.Integer):
        subtypeSpec = univ.Integer.subtypeSpec + constraint.ValueRangeConstraint
            (-1800000000,1800000001)

# Elevation ::= OCTET STRING (SIZE(2))
#   -- 1 decimeter LSB (10 cm)
#   -- Encode elevations from 0 to 6143.9 meters
#   -- above the reference ellipsoid as 0x0000 to 0xEFFF.
#   -- Encode elevations from -409.5 to -0.1 meters,
#   -- i.e. below the reference ellipsoid, as 0xF001 to 0xFFFF
#   -- unknown as 0xF000
class Elevation(univ.OctetString):
        subtypeSpec = constraint.ValueSizeConstraint(2,2)

# PositionalAccuracy ::= OCTET STRING (SIZE(4))
#   -- And the bytes defined as folllows

#   -- Byte 1: semi-major accuracy at one standard dev
#   -- range 0-12.7 meter, LSB = .05m
#   -- 0xFE=254=any value equal or greater than 12.70 meter
#   -- 0xFF=255=unavailable semi-major value

#   -- Byte 2: semi-minor accuracy at one standard dev
#   -- range 0-12.7 meter, LSB = .05m
#   -- 0xFE=254=any value equal or greater than 12.70 meter
#   -- 0xFF=255=unavailable semi-minor value
#   -- Bytes 3-4: orientation of semi-major axis
#   -- relative to true north (0˜359.9945078786 degrees)
#   -- LSB units of 360/65535 deg  = 0.0054932479
#   -- a value of 0x0000 =0 shall be 0 degrees
#   -- a value of 0x0001 =1 shall be 0.0054932479degrees
#   -- a value of 0xFFFE =65534 shall be 359.9945078786 deg
#   -- a value of 0xFFFF =65535 shall be used for orientation unavailable
#   -- (In NMEA GPGST)
class PositionalAccuracy(univ.OctetString):
        subtypeSpec = constraint.ValueSizeConstraint(4,4)

# TransmissionState ::= ENUMERATED {
#    neutral         (0), -- Neutral, speed relative to the vehicle alignment
#    park            (1), -- Park, speed relative the to vehicle alignment
#    forwardGears    (2), -- Forward gears, speed relative the to vehicle alignment
#    reverseGears    (3), -- Reverse gears, speed relative the to vehicle alignment
#    reserved1       (4),
#    reserved2       (5),
#    reserved3       (6),
#    unavailable     (7), -- not-equipped or unavailable value,
#    }
class TransmissionState(univ.Enumerated):
        namedValues = namedval.NamedValues(
                ('neutral',0),
                ('park',1),
                ('forwardGears',2),
                ('reverseGears',3),
                ('reserved1',4),
                ('reserved2',5),
                ('reserved3',6),
                ('unavailable',7)
                )
        subtypeSpec = univ.Enumerated.subtypeSpec + constraint.SingleValueConstraint
            (0,1,2,3,4,5,6,7)

# Speed ::= INTEGER (0..8191) -- Units of 0.02 m/s
#           -- The value 8191 indicates that
#           -- speed is unavailable
class Speed(univ.Integer):
        subtypeSpec = univ.Integer.subtypeSpec + constraint.ValueRangeConstraint(0,8191)

# TransmissionAndSpeed ::= OCTET STRING (SIZE(2))
# ADAPTATION :: USED AS A SEQUENCE
#     -- Bits 14˜16 to be made up of the data element
#     -- DE_TransmissionState
#     -- Bits 1˜13 to be made up of the data element
#     -- DE_Speed
class TransmissionAndSpeed(univ.Sequence):
        componentType = namedtype.NamedTypes(
                namedtype.NamedType('state',TransmissionState()),
                namedtype.NamedType('speed',Speed())
                )

# Heading ::= INTEGER (0..28800)
#    -- LSB of 0.0125 degrees
#    -- A range of 0 to 359.9875 degrees
class Heading(univ.Integer):
        subtypeSpec = univ.Integer.subtypeSpec + constraint.ValueRangeConstraint(0,28800)

# SteeringWheelAngle ::= OCTET STRING (SIZE(1))
#     -- LSB units of 1.5 degrees.
#     -- a range of -189 to +189 degrees
#     -- 0x01 = 00 = +1.5 deg
#     -- 0x81 = -126 = -189 deg and beyond
#     -- 0x7E = +126 = +189 deg and beyond
#     -- 0x7F = +127 to be used for unavailable
class SteeringWheelAngle(univ.OctetString):
        subtypeSpec = constraint.ValueSizeConstraint(1,1)

# AccelerationSet4Way ::= OCTET STRING (SIZE(7))
#    -- composed of the following:
#    -- SEQUENCE {
#    --    long Acceleration,          -x- Along the Vehicle Longitudinal axis
#    --    lat  Acceleration,          -x- Along the Vehicle Lateral axis
#    --    vert VerticalAcceleration,  -x- Along the Vehicle Vertical axis
#    --    yaw  YawRa
#    --    }
class AccelerationSet4Way(univ.OctetString):
        # TODO Data Frame to implement
        subtypeSpec = constraint.ValueSizeConstraint(7,7)

# BrakeSystemStatus ::= OCTET STRING (SIZE(2))
   # -- Encoded with the packed content of:
   # -- SEQUENCE {
   # --   wheelBrakes        BrakeAppliedStatus,
   # --                      -x- 4 bits
   # --   wheelBrakesUnavailable  BOOL
   # --                      -x- 1 bit (1=true)
   # --   spareBit
   # --                      -x- 1 bit, set to zero
   # --   traction           TractionControlState,
   # --                      -x- 2 bits
   # --   abs                AntiLockBrakeStatus,
   # --                      -x- 2 bits
   # --   scs                StabilityControlStatus,
   # --                      -x- 2 bits
   # --   brakeBoost         BrakeBoostApplied,
   # --                      -x- 2 bits
   # --   auxBrakes          AuxiliaryBrakeStatus,
   # --                      -x- 2 bits
   # --   }
class BrakeSystemStatus(univ.OctetString):
        # TODO
        subtypeSpec = constraint.ValueSizeConstraint(2,2)\

# VehicleWidth ::= INTEGER (0..1023) -- LSB units are 1 cm
class VehicleWidth(univ.Integer):
        subtypeSpec = univ.Integer.subtypeSpec + constraint.ValueRangeConstraint(0,1023)

# VehicleLength ::= INTEGER (0..16383) -- LSB units are 1 cm
class VehicleLength(univ.Integer):
        subtypeSpec = univ.Integer.subtypeSpec + constraint.ValueRangeConstraint(0,16383)

# VehicleSize ::=  SEQUENCE {
#    width     VehicleWidth,
#    length    VehicleLength
#  }  -- 3 bytes in length
class VehicleSize(univ.Sequence):
        componentType = namedtype.NamedTypes(
                namedtype.NamedType('width',VehicleWidth()),
                namedtype.NamedType('length',VehicleLength()),
                )

#####
## SAE J2735 DSRC standard message sets
#####
class BasicSafetyMessage(univ.Sequence):
        componentType = namedtype.NamedTypes(
                namedtype.NamedType('msgID',DSRCmsgID()),
                namedtype.NamedType('msgCnt',MsgCount()),
                namedtype.NamedType('id',TemporaryID()),
                namedtype.NamedType('secMark',DSecond()),
                namedtype.NamedType('lat',Latitude()),
                namedtype.NamedType('long',Longitude()),
                namedtype.NamedType('elev',Elevation()),
                namedtype.NamedType('accuracy',PositionalAccuracy()),
                namedtype.NamedType('speed',TransmissionAndSpeed()),
                namedtype.NamedType('heading',Heading()),
                namedtype.NamedType('angle',SteeringWheelAngle()),
                namedtype.NamedType('accelSet',AccelerationSet4Way()),
                namedtype.NamedType('brakes',BrakeSystemStatus()),
                namedtype.NamedType('size',VehicleSize())
                )
```

.3 Implementation Pseudo-code

Listing 2: Acquisition Module Pseudocode

```
class GPSAcquisition():
        def __init__():
                currentLatitude = null
                currentLongitude= null
                currentSpeedGPS = null
                currentAltitude = null


        def run(port):
                checkSerialPort(port)
                while(true):
                        NMEA = port.readlines(None)
                        currentLatitude  = parseNMEA_Latitude(NMEA)
                        currentLongitude  = parseNMEA_Longitude(NMEA)
                        currentSpeedGPS  = parseNMEA_Speed(NMEA)
                        currentAltitude  = parseNMEA_Altitude(NMEA)

        def getCoordinates(self):
                return currentLatitude, currentLongitude, currentSpeedGPS, currentAltitude

class OBDAcquisition():
        def __init__():
                OBDHandler = null
                currentSpeed = null
                currentRPM = null
                currentFuel = null

        def run(port):
                OBDHandler = setupBluetoothOBD(port)

        def getOOBDinfo(self):
                currentSpeed = OBDHandler.getandParse('speed')
                currentRPM = OBDHandler.getandParse('rpm')
                currentFuel = OBDHandler.getandParse('fuel')
                return currentSpeed, currentRPM, currentFuel

/*Main Execution*/
GPSAcquisition.start()
OBDAcquisition.start()
connection = startListening()
while(true):
        JSONrequest = connection.acceptAndReceive()
        if(JSONrequest == "GPS"):
                connection.send(GPSAcquisition.getCoordinates())
        elif(JSONrequest == "OBD"):
                connection.send(OBDAcquisition.getOBDinfo())
        elif(JSONrequest == "NEI"):
                connection.send(TableManager.getCurrentNeighbours())
        elif(JSONrequest == "ALL"):
                connection.send(GPSAcquisition.getCoordinates(),
                        OBDAcquisition.getOBDinfo(),
                        TableManager.requestNeighbours()
                        )
```

Listing 3: Call Manager thread class pseudo code

```
class callManager(Thread):
        def __init__():
                requestFrequency =      <Frequency value loaded from platform configuration
                    file>
                currentCoordinates = null
                currentSpeed = null
                currentRPM = null
                currentFuelLevel = null
                currentNeighborList = null
                self.start()

        def run():
                while(True):
                        currentCoordinates = requestToPlatform("GPS")
                        currentSpeed = requestToPlatform("OBD").speed
                        currentRPM = requestToPlatform("OBD").RPM
                        currentFuelLevel = requestToPlatform("OBD").fuel
                        currentNeighborList = requestToPlatform("NEI")
                        sleep(requestFrequency)

        def requestToPlatform(request):
                connectToPlatform(port)
                connection.send(request)
                listen(myport)
                reply = listen.acceptReceive()
                return reply
```

Listing 4: Communication Manager Sender thread class pseudo-code

```
class commManagerSender(Thread):
        def __init__():
                safetyChannelListener.start(port)
                comfortChannelListener.start(port)
                InterfaceManager.start(port)

        def sendMessage(message$_{J2735ALC}$):
                stampedMessage$_{J2735ALC}$ = runPriorityCheck(message$_{J2735ALC}$.getApplicationID)
                InterfaceManager.send(stampedMessage$_{J2735ALC}$)

        def sendMessageControlBroadcast(message$_{J2735BSM}$):
                stampedMessage$_{J2735BSM}$ = stampWithHighPriority(message$_{J2735BSM}$ )
                InterfaceManager.send(stampedMessage$_{J2735BSM}$)

        def sendMessageComfort(message, protocol):
                stampedMessage$_{J2735ALC}$ = runPriorityCheck(message$_{J2735ALC}$.getApplicationID)
                InterfaceManager.send(stampedMessage$_{J2735ALC}$)
```

Listing 5: Communication Manager Listener thread class pseudo-code

```
class commManagerListener(Thread):
        connection = startListening()
        while(true):
                receivedMessage = connection.acceptAndReceive()
                if(receivedMessage.type()=="J2735␣BSM"):
                        ESPDReceive(receivedMessage)
                elif(receivedMessage.type()=="J2735␣ALC"):
                        checkPlausibility(receivedMessage)
                        if(plausibleMessage):
                                publishToApplication(message)
                        else:
                                discardMessage()
```

Listing 6: Services Module pseudo-code Thread class pseudo code

```
class Services(Thread):
        def __init__():
                /* Setup Variables */
                publishFrequency = Frequency value loaded from  platform configuration file
                SafetyServicesToExecuteList = [] //List with default services to be
                    activated and executed, loaded from platform configuration file
                ComfortServicesToExecuteList = [] //List with default services to be
                    activated and executed, loaded from platform configuration file
                self.start()

        def run():
                while(True):
                        if(currentVADMstatus == "Normal"):
                                for service in SafetyServicesToExecuteList:
                                        runServiceImplementation(service)
                                for service in ComfortServicesToExecuteList:
                                        runServiceImplementation(service)
                                sleep(publishFrequency)
                        else:
                                for service in SafetyServicesToExecuteList:
                                        runServiceImplementation(service)

        def runApplicationService(serviceName, serviceImplementation):
                runServiceImplementation(serviceName, serviceImplementation)

        def addService(serviceName):
                if(checkServiceType(serviceName)=="Safety"):
                        SafetyServicesToExecuteList.append(serviceName)
                elif(checkServiceType(serviceName)=="Comfort"):
                        ComfortServicesToExecuteList.append(serviceName)

        def removeService(serviceName):
                if(checkServiceType(serviceName)=="Safety"):
                        SafetyServicesToExecuteList.remove(serviceName)
                elif(checkServiceType(serviceName)=="Comfort"):
                        ComfortServicesToExecuteList.remove(serviceName)
```

.4 Evaluation and Tests Partial Results

Table 2: Traffic volume generated and received by the several applications reduction by using VADM application level scheduling

Applications	**Traffic Volume (KBps)**					
	Scenario A		**Scenario B**		**Scenario C**	
	Generated	**Received**	**Generated**	**Received**	**Generated**	**Received**
RequestAssistance + VehicleStream	202.488	21.099	202.263	9.293	69.913	9.118
RequestAssistance + Chat	0.032	0.022	0.027	0.011	0.009	0.010
RequestAssistance + VehicleStream + Chat	134.847	13.698	132.813	6.051	48.257	6.031
RequestAssistance + POI	0.004	0.107	0.00324	0.054	62.582	0.041
RequestAssistance + RSUStream	19.257	136.900	7.995	130.506	8.045	73.296
RequestAssistance + RSUStream + POI	12.470	88.720	5.436	81.420	5.555	50.631