Universidade de Aveiro
2013

Departamento de Electrónica, Telecomunicações e Informática

Filipe Ferreira de Oliveira

REINVENT: solução para uso de redes veiculares em aplicações móveis

REINVENT: accessing Vehicular Networks in Mobile Application

Universidade de Aveiro
2013

Departamento de Electrónica, Telecomunicações e Informática

Filipe Ferreira de Oliveira

REINVENT: solução para uso de redes veiculares em aplicações móveis

REINVENT: accessing Vehicular Networks in Mobile Application

Dissertação apresentada à Universidade de Aveiro para cumprimento dos requisitos necessários à obtenção do grau de Mestre em Engenharia de Computadores e Telemática, realizada sob a orientação científica da Dra. Susana Sargento e Dr. José Maria Fernandes, Professores Auxiliares do Departamento de Electrónica, Telecomunicações e Informática da Universidade de Aveiro

Dedico este trabalho aos meus Pais e à minha Namorada por me terem apoiado de forma incansável.

O júri

Presidente	**Prof. Dr. José Alberto Gouveia Fonseca** Professor Associado do Departamento de Electrónica, Telecomunicações e Informática da Universidade de Aveiro
Vogais	**Prof. Dra. Ana Cristina Costa Aguiar** Professora Auxiliar Convidada do Departamento de Engenharia Electrotécnica e de Computadores da Faculdade de Engenharia da Universidade do Porto (Arguente Principal)
	Prof. Dra. Susana Isabel Barreto de Miranda Sargento Professora Auxiliar do Departamento de Electrónica, Telecomunicações e Informática da Universidade de Aveiro (Orientadora)
	Prof. Dr. José Maria Amaral Fernandes Professor Auxiliar do Departamento de Electrónica, Telecomunicações e Informática da Universidade de Aveiro (Co-Orientador)

Agradecimentos

Quero agradecer aos meus Pais por me proporcionarem a oportunidade de frequentar um curso superior e por me apoiarem ao longo de todo o meu percurso académico.

Agradeço também à Joana por toda a motivação, carinho, conselhos, compreensão e apoio nas situações difíceis durante todo o meu percurso académico.

Quero agradecer a todos os colaboradores do Instituto de Telecomunicações que me ajudaram durante o meu trabalho e se mostraram sempre disponíveis. Um agradecimento especial para o Luis Coelho e Filipe Neves pela ajuda prestada na integração do meu trabalho com as placas.

Agradeço a todos os meus amigos por estarem presentes, em especial ao Joel Santos e Diogo Vieira que embora não estejam directamente ligados ao desenvolvimento deste trabalho, seguiram de perto o meu percurso académico e foram importantes em muitas outras etapas do percurso. Um agradecimento especial também ao Rui Nunes por se ter mostrado disponível para realizar os testes no terreno deste trabalho.

Por último, mas não menos importante, quero agradecer à Professora Susana Sargento e ao Professor José Maria Fernandes pela orientação, coordenação e dedicação prestada durante todo o trabalho. Agradeço também ao André Cardote pois foi ele que me motivou para a temática deste trabalho e ajudou de forma incansável a todos os níveis.

Palavras-chave

Android, Content Provider, REST, Simulação, Experiências Reais, Integração, Comunicação Veicular, Dispositivos Móveis, Arquitectura, Experiências e Testes, Software.

Resumo

As redes veiculares têm sido alvo de grandes avanços tecnológicos, e a comunicação entre veículos é hoje uma realidade que tem despertado o interesse tanto ao nível da investigação como de alguns dos principais fabricantes de automóveis com o intuito de criar um conjunto de serviços para melhorar a experiência dos utilizadores deste tipo de redes. Por outro lado, os dispositivos móveis como smartphones, tablets ou PDA's também são uma área emergente no mundo das tecnologias devido ao enorme aumento de capacidade computacional que sofreram nos últimos anos. Embora as redes veiculares tenham sido alvo de grandes avanços tecnológicos continuam a encontrar obstáculos para a sua afirmação devido à indisponibilidade de dispositivos nos veículos que permitam usufruir das suas potencialidades. Esta falta de dispositivos pode ser ultrapassada aliando o mundo dos dispositivos móveis com as redes veiculares. Utilizando o potencial das redes veiculares e a capacidade computacional dos novos dispositivos móveis pode-se explorar um cenário de criação de serviços e aplicações de segurança, controlo e eficiência de tráfego e entretenimento.

O presente trabalho propõe-se a estudar, criar e testar uma solução para a integração das duas áreas tecnológicas referidas anteriormente. Neste documento é descrita uma arquitectura de alto nível que permite a integração de aplicações móveis com as redes veiculares, abstraindo as camadas de transporte e de rede com um módulo de *software* que fornece os métodos necessários para as aplicações usufruírem dos serviços das redes veiculares. O resultado final deste trabalho é uma arquitectura de *software* para integração em aplicações Android que permite utilizar a rede veicular para comunicação entre as aplicações. Ao longo deste documento é descrito todo o processo de implementação desta arquitectura, e posteriormente é apresentada a implementação de aplicações exemplo para experimentação da arquitectura e avaliação do seu desempenho.

No âmbito da Dissertação foram criados cenários para realização de testes de desempenho das aplicações em ambientes reais e simulados. Estes testes serviram para identificar a viabilidade da utilização do REINVENT em dispositivos com diferentes características de hardware, e também para identificar potenciais pontos de atraso na estrutura da arquitectura criada.

Os resultados obtidos permitiram constatar que a utilização desta arquitectura não induz qualquer tipo de interferência nem atraso no normal funcionamento das aplicações, e que o REINVENT pode ser utilizado na criação de novas aplicações móveis no âmbito das redes veiculares.

Keywords

Android, Content Provider, REST, Simulation, Real Experimentation, Vehicular Communication, Mobile Devices, Architecture, Experiments and Tests, Integration, Software.

Abstract

Vehicular networks have been the subject of major technological progress, and the communication between vehicles is a reality that has been the subject of interest both in terms of research and of some of the major car manufacturers in order to create a set of services to enhance the user experience of such networks. On the other hand, mobile devices such as smartphones, tablets and PDA's are also an emerging technology in the world due to the enormous increase of computing power they got in recent years. Although vehicular networks have been the subject of great technological advances, they continue to encounter obstacles to their raising due to unavailability of devices in vehicles that allow the use of its potential. This lack of devices can be overcome by combining the world of mobile devices with vehicular networks. Using the potential of vehicular networks and computational capabilities of new mobile devices, a set of scenarios can be explored in order to create services and applications for security, control and efficiency of traffic and entertainment.

This work proposes to study, create and test a solution for the integration of the two technology areas mentioned above, applications and vehicular networks. In this Dissertation we describe a high-level architecture that allows the integration of mobile applications with vehicular networks by abstracting transport and network layers with a software architecture that provides the methods needed for the applications to take advantage of vehicular networks services. The end result of this work is a software architecture for integration into Android applications that allows the use of vehicular network for relaying communication between applications. Throughout this document, the whole process of the architecture implementation is described as well as two example applications for proof of concept, testing purpose and performance evaluation.

In order to test the performance of the REINVENT module in the applications, two test scenario environments were created, a simulated environment, integrating a VANET simulation framework with mobile devices, and a real environment using an on board unit for vehicle communication purposes. These tests served to identify the feasibility of using REINVENT in devices with different hardware characteristics, and also to identify potential sources of delay in the structure of the architecture created.

The results revealed that the use of this module does not induce any interference or delay on the normal operation of applications, and REINVENT can be used in creating new mobile applications in the context of vehicular networks.

I. Contents

I. Contents ii
II. List of Figures v
III. List of Tables vii
1 Introduction 1
1.1 Objective 2
1.2 Contribution 3
1.3 Thesis Outline 4
2 State of the Art 5
2.1 Vehicular Networks and Applications 5
2.2 Simulation of Vehicular Networks 16
2.3 Conclusions 24
3 REINVENT: Conceptual Architecture 27
3.1 REST as an architectural style 27
3.2 Message-Oriented Middleware 29
3.3 The Architecture 31
3.4 Conclusions 33
4 REINVENT: an Android based proof of concept 34
4.1 Android Platform 34
4.2 Android Content Providers as REST application interface 36
4.3 REINVENT Android Architecture Overview 41
4.4 RabbitMQ as a messaging provider 42
4.5 REINVENT services 44
4.6 Developing Applications using REINVENT 45
4.7 Implementation Details 50
4.8 Conclusions 56
5 REINVENT: Creating the testing scenarios 58
5.1 Simulated Scenario 58
5.2 Real Scenario 70
5.3 Conclusions 75
6 Experimental Tests & Results 76

6.1 REINVENT Performance on Different Devices 76
6.2 Simulated Environment 78
6.3 Real World Environment 83
6.4 Conclusion 91
7 Conclusions and Future Work 93
7.1 Future Works 95
8 Bibliography 96
Annex 101
Rest API Description 101

II. List of Figures

Figure 1-1 – Conceptualization of the problem. 2
Figure 2-1 - Intelligent Transport System [6] 6
Figure 2-2 - Vehicular Network communications types [8] 6
Figure 2-3 - Comparison of the different VANET simulation framework features[26] 24
Figure 3-1 Synchronous and Asynchronous Models [48] 29
Figure 3-2 – Conceptual Architecture of REINVENT 32
Figure 4-1 – Android Stack Overview [55] 35
Figure 4-2 – Android Content Providers Overview 36
Figure 4-3 - Google I/O REST Architecture using Content Providers[59] 39
Figure 4-4 – REINVENT Android Architecture 41
Figure 4-5- REINVENT Architecture 42
Figure 4-6 – RabbitMQ Architecture Overview 43
Figure 4-7 Welcome Activity and Contact List Activity 47
Figure 4-8 Add Contact Activity and Message Activity 48
Figure 4-9 MapView Activity and DetailView Activity 50
Figure 4-10 – NetworkProvider core class 51
Figure 4-11 - Commands Hashtable Diagram 53
Figure 4-12 - Message Container class diagram 55
Figure 4-13 - UserDescriptor class diagram 56
Figure 5-1 – Simulation Environment Architecture 59
Figure 5-2 – Simulation Scenario Overview 60
Figure 5-3 – Testing Application 63
Figure 5-4 - On Board Unit 71
Figure 5-5 - Toyota Corola and Ford Fiesta 73
Figure 5-6 - Real Scenario Architecture 74
Figure 6-1 - Message Sending Delay by Device 77
Figure 6-2 - Message Receiving Delay by Device 78
Figure 6-3 Simulated vehicle placement 79
Figure 6-4 Measuring delay inside the OBU application 79
Figure 6-5 Simulator Message Processing Time 80
Figure 6-6 Simulation Overall Schema 81

Figure 6-7 Simulation Receive and Sending Average Time 82
Figure 6-8 Point-to-Point delay with distance placement 83
Figure 6-9 Receive and Sending Average Delay with distance 83
Figure 6-10- Round Trip Time Schema 84
Figure 6-11 – Message Round Trip Time in the OBU 85
Figure 6-12 Round Trip Delay with VANET schema.. 86
Figure 6-13 RTT with distance schema.. 86
Figure 6-14 Round Trip Delay with Distance 87
Figure 6-15 Speed Variation Schema.. 88
Figure 6-16 Round Trip Delay - 20km/h.. 88
Figure 6-17- Round Trip Delay - 50km/h 89
Figure 6-18 - Round Trip Delay - 80km/h 89
Figure 6-19 - Round Trip Delay with Message per Second variation................................ 91

III. List of Tables

Table 1 – Traffic Simulators Overview 17
Table 2 – Network Simulator Overview 20
Table 3 - REST Interface overview 44
Table 4 - Android Devices Features Overview 72
Table 5 - API Methods Overview 102

1 Introduction

Vehicles are a fundamental part of our society as they are one of the most important sources of human mobility. Along the years they have been subject of several improvements and innovations at several levels. They are already in a state where they can be considered, in the future, part of the Internet, when they will become capable of inter-vehicle communication and of communication with any other network devices.

These new capabilities acquired by vehicles through the years can be seen as a great opportunity to offer the passengers with new services while driving, such as safety driving applications, Vehicle-to-Vehicle communication and also infotainment applications. One of the most limiting factors of the growing of these services offer is the lack of technology available in the vehicles capable of user interaction, since most of the devices available are on board computers created and developed to support native applications, and consequently are not open to the creation of new applications or services. Even if the European Telecommunications Standards Institute (ETSI) [1] already defined a basic set of applications including their guidelines and use cases, just as definitions, mostly because of the lack of technology available in the vehicles. Some of the examples already defined by ETSI are road hazard warning, traffic information and recommended itinerary, point of interest notification or fleet management. Some car brands are already working in this area and offer some of these services, but we are still far away from being a common feature among the vehicles.

On the other hand, in the recent years we have seen a market explosion of mobile devices [2] such as smartphones, Personal Digital Assistances (PDA) or tablets, where these became absolutely pervasive in many domains due to their computing power relation to size, and they are a great solution to fill the lack of technology available in the vehicles, offering a whole new level of technology to improve services offered in the vehicle networks [3], [4], [1].

However, there is not an integrated solution that allows mobile applications to use transport layer resources, which is clear from a TCP/IP Model stack perspective, where applications cannot have a high level interface to:

- access the communication resources on the transport layer, namely handling protocol attributes such as Channel numbers or the Provider Service Identifier (PSID) of the service [5]
- map the logic naming to real Domain Name System (DNS) of the network topology (i.e. keeping logical names regardless of the actual network addresses)

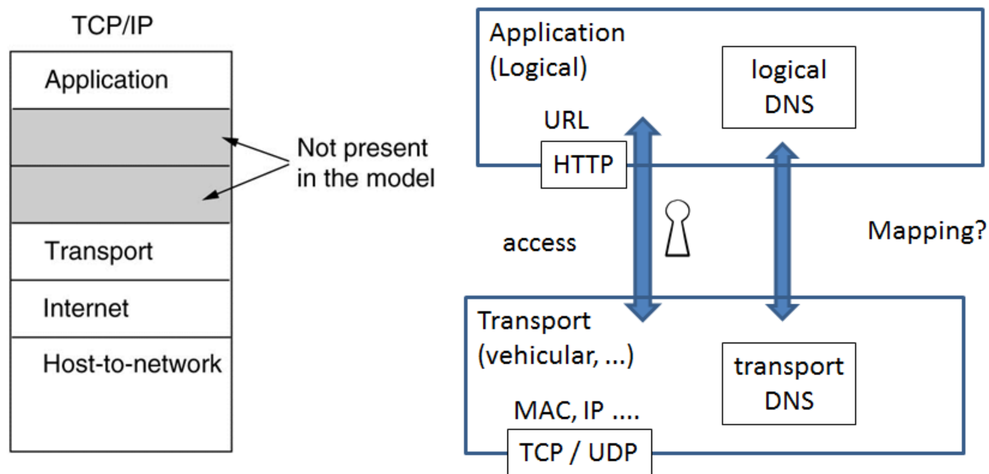


Figure 1-1 – Conceptualization of the problem.

Ideally, such a high level interface at application layer would allow an application to perform transparent data exchange at application level, abstracting from transport layer details. However, to the best of our knowledge, such solution does not exist.

1.1 Objective

The main objective of this work is to fill the gap between the vehicular networks and mobile applications by proposing a conceptual architecture that will allow mobile applications to use vehicular networks without having to understand or implement any network layer specifications. It is also our aim to implement an Android based proof of concept that will support two illustrative applications: a chat application called VNChat that allows the users to exchange simple text messages in a vehicular network context; iThere application that works like a social network for vehicular networks. The users of iThere will be broadcasting their GPS location while also tracking the locations of the vehicles around, and showing them in a map so the users can know who is around them.

Both these applications were built using the module created for exchanging messages and accessing the user's identifications.

1.2 Contribution

As the result of the work performed on this Dissertation, a high level architecture (REINVENT) was idealized and implemented in order to fill the gap between the application and transport layers when creating mobile applications in the vehicular networks context. REINVENT allows any Android developer to create any application layer communication without having to implement any transport layer specifications of vehicular network. Two proof of concept applications were implemented and deployed using REINVENT, and tested in both a simulated and real VANET scenario.

The main contributions of this Dissertation are:

- A Representational State Transfer (REST) based conceptual architectural component named REINVENT that allows a transparent access of the transport layer by mobile application, using messaging to interact with the transport layer - in fact this architecture can also be applied not only to abstract access to VANET (focus of this work), but also to other "dynamic" / reconfigurable transport solutions
- REINVENT architecture implementation for Android mobile applications tested in a simulated environment using the VSimRTI as the simulation framework.
- REINVENT architecture implementation for Android mobile applications tested in a real setup deployed in a set of cars by integrating our solution with the On Board Unit (OBU) developed in the Instituto de Telecomunicações of the Universidade de Aveiro.
- A set of performance tests were made in order to understand if the usage of our architecture implies any decrease of performance of the applications.

1.3 Dissertation Outline

This work is divided in seven fundamental Chapters. The first Chapter introduces the problematic we propose to solve with this work, as well as the contributions of this work.

The Chapter 2 will provide insight on the fundamental aspects of the vehicular networks. We will focus on the applications area and we will go through the vehicular network application classes. We also present related work in the application area for vehicular networks in conjunction with mobile platforms such as Android. The second part of this chapter is an overview of the VANET simulation tools. We will describe and compare the most used Traffic and Network simulators in the vehicular network area. Finally, we will also provide information on the most used Frameworks for simulation of VANET, by presenting features of each solution as well as describing its strengths and weaknesses.

The Chapter 3 presents the conceptual solution of the problem. We will present a generic architecture based on REST.

The Chapter 4 describes the proof of concept of the architecture idealized in the previous chapter. Using the Android platform as the technologic platform, we will provide insight into the architectural details of the architecture, like what solutions were used to fill the conceptual architecture features described in Chapter 3. We will also go through the implementation details of the architecture describing the main classes and the API's. This chapter ends with the description of the creation of mobile applications implementing our created module.

The Chapter 5 the testing scenario creation and definition is described. We created both real and simulated scenarios in order to test our solution.

The Chapter 6 will describe the experimental tests and results obtained by testing the module features using applications. The main focus of these testes will be the performance of our module in different environments and devices.

Lastly, the Chapter 7 will summarize the work done throughout this Dissertation along with the concepts obtained, and an overview on future features that can be implemented and improved in the created framework.

2 State of the Art

In this chapter we will introduce the concepts related to the work developed along this Dissertation, as well as an overview on the current state of works around this Dissertation area of interest. The topics that will be introduced in the following sections will follow the path taken on the development of the work. The chapter begins with the section 2.1, where we introduce applications being developed in the vehicular networks area, giving a special look into the works that combine vehicular network applications with Android. In the following section, 2.2, some solutions for vehicular networks simulation will be presented along the simulators used by them.

2.1 Vehicular Networks and Applications

Vehicular Ad-Hoc Networks (VANET) is a particular class of the Mobile Ad Hoc Networks (MANET) and is characterized by a set of vehicles that can communicate with each other or with external and nearby fixed equipment's using a wireless antenna. In the last few years, we witnessed a large increase of research in this area. There are a lot of factors that have led to this increase of interest, first of all the evolution and wide adaptation of the 802.11 technologies. The other major factor is the evolution of the automobile manufactures, since most of the vehicles nowadays are already equipped with global position system (GPS) which led to an increase of interest of research in the VANET area by the industry.

Figure 2-1 - Intelligent Transport System [6]

VANETs were created initially with the great objective of decreasing one of the leading cause of mortality which is car accidents [7]. Travelling by car offers way more risks than riding a plane or a train. But it is not only for security that VANETs were created; they have a great potential to offer infotainment and commodity services like checking email or reading news while driving. VANETs can also play an important role on preventing traffic jams by organizing and providing the optimum routes just by analyzing the state of traffic based on the network itself (Figure 2-1).

The deployment of the VANET was made to ensure two types of communication, vehicle-to-vehicle and vehicle-to-infrastructure communication represented in Figure 2-2.

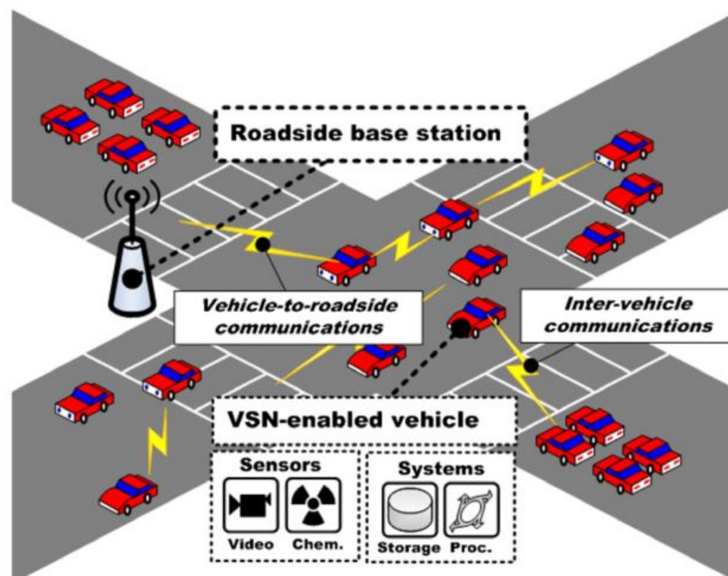


Figure 2-2 - Vehicular Network communications types [8]

The deployment of the first type of communication is made by using an On Board Unit (OBU), a piece of hardware and software placed in every vehicle that will be responsible for handling, sending and receiving all the information to and from the network. The second way of deployment is integrating the network with hotspots placed along the roads that are called RoadSide Units (RSU) [9].

VANETs are a very special part of the mobile network family as they offer some very unique and special features distinguishing them from all the other networks. Some of the features that make VANET so unique are [10]:

- Predictable mobility – Unlike the other mobile ad hoc networks, vehicles tend to have predictable paths since they are limited to the roadways, while in the other mobile networks, the nodes can basically take a different path every time. Using information gathered from location based technologies like GPS, 3G, plus the information of the vehicle like speed, it is relatively easy to predict the future positions of the vehicle.
- Unlimited transmission power – Since we are talking about the vehicle being the node of the network, we can consider that the vehicle can provide continuous power for computing and communication to the devices.

But the creation of such high potential network also faces a lot of challenges that can turn to be problems, so they must be looked in and be taken into account [9]:

- Large scale of network – It is almost impossible to predict the size of a VANET. Since it is an adaptable network, there is no maximum number of nodes allowed at given time. If we think of a scenario of a big metropolitan area in rush hour, it is impossible to predict the amount of nodes that will constitute the network.
- The mobility and partition – Since we are talking about a network made of vehicles, we need to assume that the network will need to operate in very dynamic and extreme configurations. If we think of a city in a rush hour where the vehicles would have a reasonably low relative speed to each other and the amount of nodes would be huge, in the highway the scenario is exactly the opposite as the vehicles travel with great relative speed to each other and the density of nodes can be very low at a given point.

- Dynamic topology leads to frequent loose of connection – The very dynamic topology of a VANET, due to the constant vehicle movement, leads to the isolation of some nodes at a given point which means that a particular node will lose its connection to the network. The usage of backup RSUs represents a possible solution to this potential problem.

2.1.1 Vehicular Network Applications

As referred previously, the great purpose of creation and research around the vehicular networks is the road safety, but this can only be achieved by creating actual applications taking into account real use cases. The development of applications also creates something more noticeable to stimulate the market around the VANET. In this section we will explain how applications are classified and which use cases have been defined for the creation of real applications.

Applications around VANET have been commonly classified into three categories [1][11][12][13]:

- Active road safety applications
- Traffic Efficiency and management applications
- Infotainment applications

Active road safety applications represent probably the main goal of the VANET that is to reduce the chance of traffic accidents by permanently exchanging information between the vehicles. It will provide the drivers with information of eventual hazardous events that happened in the road, and give them a chance of a preemptive action about it. This information can simply be the vehicle speed, position or distance to a given intersection. Some examples of some possible applications defined in this class as well as the use cases associated with them are given below:

- Collision avoidance in intersections and sudden break – We often watch vehicles reducing their speed abruptly for various reasons like a red traffic light or a slow moving car in a highway. If the vehicles involved are equipped with vehicle-to-vehicle communication devices, these types of collisions can be avoided by sending a warning message to the cars in the area, giving them more time to react and be

prepared for the situation. Even if the collision if eminent, the applications can be of the most useful by inflating the airbags sooner or by tighten the seat belts [11].

- Road Hazard Warning – There are several road hazards that can result in a car accident that can be avoided if they are correctly spread through the vehicles around. An emergency vehicle running in an emergency mission like an ambulance or a police car are examples of hazards that can cause an eventual accident, and can be easily avoided broadcasting an emergency message to the vehicles around so they can leave a free lane. If a vehicle needs to make an emergency stop in an unexpected place, sometimes the visual warning (the signaling triangle) is not enough to highlight the situation, and once again, a message indicating the blocked road could help to avoid an accident. A sudden traffic jam can also be considered as a road hazard, especially if the jam happens after a turn over. Spreading a message indicating the place of the traffic jam would help preventing drivers running into the jam at high speed and eventually cause an accident. Road work, signal violation, wrong way driving are also other use cases in the same topic that could be avoided with these types of applications [12].

Traffic Efficiency and management applications focuses on improving the traffic flow, coordination and assistance by providing updated information about the current state of the traffic in a given area. This information can be in the form of maps, indications or messages with space and time relevance to the vehicle requesting them. These applications can result on the reduction of congestions, accidents and even in the travel times. Vehicles play several roles in these applications; they can work as sensors by spreading their speed, relay as they will be used to spread messages, and as destination if they are the ones using the information. Some examples of some possible applications defined in this class as well as the use cases associated with them are given below:

- Speed management – These types of applications aim to help the driver to manage the speed of his vehicle, providing a smoother driving and also avoid unnecessary stopping. Providing the state of the traffic light and the optimal speed to pass the green light without having to stop would be a possible scenario of application in this area [11].

- Co-operative navigation – These are the types of applications that will increase the traffic efficiency by providing means so the vehicles can cooperate with each other. A platooning management application fits this scenario by forming groups of vehicles that have the same destination place, and by forming thigh columns of vehicles following each other would result in the increase of capacity of the roads. It is very usual in big cities to exist those bottlenecks of traffic at rush hour in the main roads. An application that would gather information of most critical points and return to the driver a recommended itinerary based on the state of the several alternatives would be another example that would fit this topic [11][14][12].

Infotainment and other applications are those that do not fit on the previous categories i.e. road safety and traffic management and efficiency applications. Most of the applications in this class are related to the vehicle passengers as they should be mostly used to provide entertainment or information on a regular basis. The 'other' should include all the applications that are not directly related to the vehicular network, but still play an important role to provide a better driving experience like increasing the fuel economy or providing diagnostic information about any anomaly in the car to be accessed more easily by the technicians. Some examples of possible applications and their use cases are described below:

- Internet Access in a Vehicle – This is a use case to be used as an application to provide internet access to all the passengers, to allow them to use all types of IP based services from inside the vehicle. This would be possible by using multi-hop route to a RSU that would act as an Internet gateway. This is probably the most important application in this class, as it would allow the VANETs to be part of the internet [11], [12].
- Point of interest notification – If we think of a scenario of traveling in a foreign country, it would be of the major interest of the driver to have access to information about the points of interest of the area, like fuel gas station in case of being running low on gas, tourist attraction to be visited or even local business for shopping. These types of applications present a huge market potential as it would be a way of

advertising all kinds of businesses or points of interest that otherwise could not be caught by the attention of the driver [12].

- Remote diagnostics – This is a major help in case of any vehicle breakdown, as a service garage could start the vehicle diagnosis from the garage without having to go to the breakdown local, and even have access to the history of the vehicle in terms of previous interventions, even if they were made in another place as the application would give access not only to the current vehicle readings but also to the records kept along the time [11].

2.1.2 Mobile platforms and Vehicular Networks

There are few works where the Vehicular Networks and mobile operating systems based applications meet. Moreover, none of them refer to communication between two direct applications using the vehicular networks. Therefore, we will describe the applications developed in the vehicular network areas, so that we can understand how close we are from having some of the use cases explained in the previous section.

Hernandez et al. [15] have prototyped and tested an in-vehicle embedded system to allow communication from any user gadgets like smartphones, PDAs or tablets to the vehicle system, or even with road infrastructures in order to get access to Intelligent Transportation Systems (ITS). This work results on the creation of an OBU prototype, as well as two services, eco-driving and traffic reports, to work as proof of concept of the architecture prototyped. They enhance the fact that most of the services referenced by the related work are using a top-down approach, so the OBUs should be designed after the services are created, so they fill the requirements of the services. This is not a good approach, since the changes in hardware are usually way more expensive than the software ones. The OBU was prototyped for supporting several new and future services.

Al-Ani et al. [16] built an integrated system that provides several infotainment services to the user like playing music, location based services, traffic and road information, as well as many other services available from third parties. They created their system based on Android OS with the main goal of creating a standard architecture for the in-vehicle infotainment industry.

Cheng et al. [17] designed an Android based mobile device platform that integrated with the network management functions and MOST (Media Oriented System Transport)

technologies, which is a standard for multimedia and infotainment network in the automotive industry [18]. The objective is to provide heterogeneous network management functionalities to guarantee communication quality. They propose an algorithm to support seamless handover via effective and rapid resource between the networks. The system was designed to provide two different functions. The first one, Roaming, where the system would be searching for the wireless network with the best resources available, which is very helpful if we are moving between heterogeneous and homogeneous networks. The second, Sharing, the system would make available all the resources for the users connected to all the different interfaces. In the future, they want to redesign the algorithm in order to enhance the quality of service during the network roaming and sharing functions, as well as to improve the resource management.

Spelta et al. [19] created a smartphone based system for motorcycles. The system was designed to increase the safety level of a motorcycle, and it is composed by a CAN Bus, an electronic unit that works as CAN-to-Bluetooth gateway, a smartphone and a helmet equipped with a Bluetooth device. The idea of the system is to create a vehicle-to-driver and vehicle-to-environment communication mechanism based on the smartphone core. The vehicle is equipped with the CAN-to-Bluetooth converter that is interfaced with the smartphone, which acts as a gateway with the helmet and also with an external webserver. They created this architecture mostly as a base for developing new services and applications in the motorcycle industry, as it turns out to be needed to have a relatively low development effort by the manufacturers, as the CAN Bus should be a plug-n-play system. To create a proof of concept and a test purpose, they implemented the system in a real motorcycle and they used two smartphones with different operating systems running the same software adapted to each OS. The system handled five types of operations:

- Driver requesting information from the vehicle – Using a speech synthesizer, the vehicle would respond to different requests like the water temperature or the current speed.
- Notification Alert – The system can recognize irregular values on the vehicle and then the driver would receive a notification on the audio system of the helmet.

- Driver sends commands – Driver could spell a couple of commands handled by the system that would process them into the vehicle like turning off lights.
- Data analysis – The vehicle can submit data to the web server to be analyzed and diagnosed. An example of this operation can be the report of the current miles to be submitted and then a warning could be sent to the driver informing the miles required for the next revision.
- Driver-to-Web communication – The interaction here is made between the driver and the smartphone, where the driver asks for information, possibly about a point of interest, and the smartphone will fetch from the web and give it to the driver in form of voice synthesizer.

Diewald et al. [20] created an Android based driver assistance system called DriveAssist. The system consisted of two components, a vehicle integrated V2X communication unit (Commonly known as OBU) supporting ITS G5 or IEEE 802.11p, and one or more mobile devices such as smartphones or tablets. The system main goal is to gather information from different data sources, including other V2X communications, giving the driver an overview of the surrounding traffic on a map view. The system also runs a background system to trigger certain warning messages being spread on the network. The system was developed and tested on a seven inch Samsung Galaxy Tab running Android Gingerbread 2.3.7. The application main menu has four available options, the first one, Stop Services, is used to manage all the background running services. The second one, Show Map, leads the user to a map view where the user can see a representation of his vehicle, as well as all the traffic information signals received and placed around the map. The third option is Traffic Info, where the information placed on the previous option will now be displayed in a sorted and filtered table. Finally, the last option is used to configure the application preferences like the radius of interest to receive information or the theme. The objective of this work was to create an audio-visual system for V2X data and information, and thereby a vehicle on-board driver assistance that could be available at a competitive cost.

Su et al. [21] found an alternative solution for vehicle-to-vehicle and vehicle-to-infrastructure communication protocols. Due to the Google announcement of Android supporting Wi-Fi direct connection, they came up with the idea of using this new feature of

Android 4.0 to replace the commonly used WAVE/DSRC protocols. Unlike the older versions or any other mobile OS, the Wi-Fi direct connection does not require any access point for the communication. The Wi-Fi direct technology, defined in the android.net.wifip2pclass, created an infrastructure network instead of a distributed network (ad-hoc) by switching the device to a portable access point. The use of this technology brings some drawbacks like being limited to devices running Android 4.0 natively and the operating system requires the user to accept all the connections manually due to security reasons. Due to the restrictions described above, they revise their system to use the Android 2.2 API to control de Wi-Fi AP/Client instead of the 4.0 direct connections. With this revision they reached a much bigger population, and the system did not require the manual acceptation from the users, but it also came with a drawback: it still took too much time to set up a connection with the access point device, around 1.7 seconds for tablets and 3 seconds for smartphones. In order to obtain the best performance of the android devices they evaluated the system under an ad-hoc network with rooted android devices. With the rooted devices, it is possible to send and receive any UDP broadcast packets with a pre-setup ad-hoc environment within the communication range. They concluded from their work that while their communication range is a little low, around 50 to 60 meters, the round trip delay time of the system for one hop of 30ms can provide an alternative to the 802.11p used by the OBU for vehicle communication.

Yun et al. [22] developed components, based on the work done in [23], to get eco-driving and safety-driving information using information fetched from the vehicle (mileage, speed, sensor data) that would be analyzed and presented to the user through an android application. The objective of this work is to provide the drivers with information so they can analyze their driving style and behavior. They developed an API that can be reused by vehicle-it companies to develop further mobile service applications more easily and quickly. The system is divided in two components; the first one is related to the eco-driving, was implemented in C++ and gets and analyzes information from the vehicle about the fuel consumption and efficiency and CO_2 emission rates. The second component of this system was implemented in Java and analyses the Safety-driving information by getting information about quick acceleration, quick start, over speeding, long term over speeding, sudden acceleration, quick deceleration, quick breaking and quick stop. All this information is fetched from the vehicle On Board Diagnosis module through a Bluetooth

communication. They concluded from the work that the information provided by the system can be important for the drivers to analyze and readapt their driving styles and behaviors.

Campolo et al. [24] created SMaRTCaR, a smartphone based platform that communicates with low-cost dedicated hardware to interact with vehicle sensors and their surroundings. The data retrieved from these sensors is classified according the type, and opportunistically transmitted via the most convenient wireless interface to an external monitoring center. The SMaRTCaR system can be devided in two different blocks; the first one concerns the data collection and pre-processing. They used an Arduino platform, USB capable, which received inputs from several sensors inside the vehicle as well as from the surrounding sensors. It is also responsible for merging and storing temporarily the received data. The second block of the system concerns the data visualization and transmission and is represented by the Android smartphone. The smartphone application receives data from the Arduino board and shows them to the user, as well as packing it to be sent in an appropriate time which is every time the smartphone is connected to a Wi-Fi access point. The smartphone also gathers information like GPS location and time in order to tag the data received from the Arduino. The communication between the two blocks is made through the Accessory Development Kit, a standard created by Google to provide communication between Android devices and external hardware devices. The SMaRTCaR system enables the collection of wide and modular set measurements from the sensors inside the vehicle as well as from environmental sensors. It uses a plug and play approach so any android can connect to the system, and finally has the virtue of cheapness as the hardware required to interface with the smartphones is not high (around 100€).

In this section we presented some works in the vehicular network context using mobile platforms. Works like Diewald et al. [20] are designed simply to support the requirements of the system and can be hardly used to implement further features, while other works like Spelta et al. [19], Hernandez et al. [15] and Al-Ani et al. [16] created an architecture from the ground in order to work as a base for creating further services and features. Other works like Campolo et al. [24] and Yun et al.[23] developed systems in the Android platform that receive information from the vehicle sensors to supply the vehicle driver with useful information about the vehicle condition and surroundings, but they still

do not share and spread that information through the vehicle network. But what is important to point out is that none of the works presents an abstraction of the network and transport layers, so if further applications, services or new features had to be implemented, a strong knowledge of the vehicular network context would be needed. Most of these works focus on the connection of the vehicle network environment to the OBU that will have the network information available to external or incorporated devices, but what if we wanted those devices to be able to use the network resources in their applications and not only to consume the information gathered by the OBU services? This is the subject of this Dissertation.

2.2 Simulation of Vehicular Networks

The possible dimension of a real vehicular network scenario is big enough to become a problem when it comes to develop, test and integrate with real systems. If we think that each node of the network is a vehicle, it is unfeasible to create those kinds of scenarios with real vehicles in the real world. The costs of that real implementation would be too high and would be only accessible to a little niche of people. Even some large real scale development scenarios that currently exist [25] have only the capacity to test a little portion of what the real scenarios would be. On the other hand, the vehicular networks environment is very complex, and there are a lot of models that need to be considered for the network and also the proprieties of the vehicles and their drivers.

Due to these conditions, simulation has become one of the main approaches in the vehicular networks research. However, there is still no standard simulator for vehicle communications [26]. Since most of the times it is not realistic to test a system in real VANET scenarios, testing them in simulated scenarios is commonly the most cost-effective option. In order to create a realistic VANET simulation scenario, there are three fundamental requirements:

- Traffic simulator in order to simulate vehicle behavior like physical vehicle movement and interactions or to create a road network.
- Network simulator in order to handle the wireless transmission among the vehicles and between the vehicles and RSUs.

- A runtime environment for the applications that are to be implemented in each vehicle that are responsible for further interaction with external elements in the vehicles, possibly a smartphone or a tablet [27].

There is still no standard simulator for VANET simulation [26] as there is no "Vehicle Network Simulator". The actual solution for simulating VANET is coupling independent Network and Traffic simulator, but once again there are a lot of different simulators of each type that will be described in the following section, in order to understand the features of each simulator.

2.2.1 Traffic Simulation Overview

The simulators described in this section are capable of creating mobility traces that can be used as an input for network simulators. The trace files generated by these simulators are in a format supported by most network simulators, which makes their interaction easier. These traffic simulators are not capable of simulating communication protocols, and therefore, the exchanged messages between them cannot impact the mobility model [26].

Table 1 shows a brief comparison on some features of the traffic simulators that will be described in this section.

Name	*OS Support*	*GUI*	*Open-Source*	*Controllable by external apps*	*Integration with external maps*
CORSIM	Windows	No	Yes	No	No
VISSIM	Windows	No	No	No	No
SUMO	Windows Linux	Yes	Yes	Yes	Yes*
VanetMobiSim	Windows Linux	Yes	Yes	No	Yes
MOVE	Windows Linux	Yes	Yes	No	Yes

Table 1 – Traffic Simulators Overview

*It can transform OSM maps into SUMO model using an external tool

We will now make an overview on the most commonly used traffic simulators in VANET simulation;

- **CORidor SIMulator (CORSIM)** – is a vehicular mobility simulator that was developed by the US Federal Highway Administration and is currently maintained and supported by the University of Florida. It consists on the integration of two microscopic models to represent all the traffic environments. NETSIM represents and is responsible for traffic in urban scenarios, while FERSIM represents the traffic in highways or freeways. CORSIM requires Microsoft Windows operating systems to run, and therefore, it can be an issue to integrate with network simulations as most of the simulators are usually Linux based [28].
- **VISSIM** – is a very powerful framework developed by PTV Planung Transport Verkehr AG in Karlsruhe, Germany. The framework is based on a visual block diagram language that allows the user to create and define the scenarios. This simulator is one of the few where the traffic model is a car-following model that also considers psychological characteristics of the drivers. It also includes pedestrian mobility model which is very interesting for simulation of urban scenarios. Like CORSIM, it is only available for Microsoft Windows operating systems [29].
- **Simulation of Urban MObility (SUMO)** - is an open-source microscopic simulator developed by members of the Institute of Transportation Systems at the German Aerospace Center. It can be used in most operating systems, and has a great community around the project that have developed a lot of interesting extensions for the simulator like generation of GPS traces in real time or communication with external applications for controlling the flow of the simulation in real time. It has one of the most simplistic driver models which translate not only in a high speed simulation time, but also in a low detail mobility model (that can be seen as good or bad depending on the purpose of the simulation). Due to his high portability [1], SUMO has

become the most used traffic simulator for vehicle communications [30], [31].

- **VanetMobiSim** – is a freely distributed, open-source vehicular mobility generator based on the CanuMobiSim[32] architecture. It is developed specially for vehicular network simulation in Java and can generate mobility traces for a lot of different network simulators like ns-2[33], QualNet[34] and GloMoSim. It supports several road topology definition types like extracting from TIGER or GDF maps, or simply user-defined maps by listing the vertices of the graph and their connection edges. The movement models can define random trips, where the vehicle will travel along random roads or an origin-destination route. The mobility model also includes intersection management, lane changing and, unlike most traffic simulators, simulations of road incidents like accidents [35].
- **MObility model generator for VEhicular networks (MOVE)** – it is built on top of SUMO and can also produce traces that are ready to be directly used by several network simulators like ns-2, OPNET and Qualnet. MOVE allows users to quickly generate VANET mobility models by interfacing with real map databases, like TIGER and Google Earth. This is all accomplished by using a Graphical User Interface (GUI) created to improve and automate the script generation. MOVE allows the creation of a user-generated map and it also proposes some pre-defined topologies (grid, spider, random networks) [36].

2.2.2 Network Simulation Overview

In this section we will describe the more popular network simulators in the VANET simulation communities that commonly use them to simulate many different network scenarios. Our focus will be on describing the features related to the vehicular communications that are included in each of them. We will also discuss their performance presented by Weingärtner et al. [37].

Table 2 shows a brief comparison on some features of the network simulators that will be described in this section.

Name	*OS Support*	*Open-Source*	*GUI*	*IVC Comunication protocols*
Ns-2	Linux	Yes	No	Yes
Ns-3	Linux	Yes	No	Yes
QualNet	Windows & Linux	No	Yes	No
OMNet++	Windows & Linux	Yes	Yes	Yes*
JiST / SWANS	Windows & Linux	Yes	No	Yes*

Table 2 – Network Simulator Overview

*Using unofficial modules

We will now describe in more detail every simulator referenced in the previously shown table in order to understand each simulator features, strengths and weaknesses.

- **Network Simulator 2 (ns-2)** [33] – is the most used network simulation tool in academic network research [31]. The project that began is supported by the US DARPA and it is an open-source framework developed in C++ and TCL. The framework became a standard due to the large community. The official ns-2 release contains two important models for VANET simulation, the Manhattan model that represents a grid model where the vehicles are allowed to move along the grid of horizontal and vertical streets. The other model is the freeway model which is a much simpler model where the vehicles are restricted to his lane and cannot take any turns. Concerning the higher communication layers, a lot of contributions have been made by the community in order to create an accurate model of the IEEE 802.11p, the standard protocol in VANET communication. The biggest disadvantage of using the ns-2 in vehicular network simulation is the performance when it comes to scenarios with more than a few hundreds of nodes. The project has become inactive since 2010.

- **Network Simulator 3 (ns-3)** [38] – is the evolution of the previous simulator ns-2. The simulator has an optimized kernel and the elimination of C++/TCL interactions reduced much of his complexity as all the modules are now implemented purely in C++, although some simulation parts can be optionally implemented in Python. On the opposite of ns-2, it allows now the simulation of large scale scenarios composed by a few thousands of nodes. However, even if the ns-3 is an evolution of ns-2, most of the models developed to ns-2 were ported to ns-3.
- **QualNET** [34] – is the commercial version of GloMoSim Project, that stopped in 2000, and therefore a freeware software for planning and testing tool that simulates the behavior of real communication networks. This framework is capable of simulating scenarios with a few thousands of nodes without any performance loss, and it provides a very powerful Graphical User Interface (GUI) used in every simulation scenario detail like terrain, network connections or mobility patterns. It also has tools for analysis and packet tracing of the simulations. There is no currently VANET mobility models implemented in Qualnet, and therefore, it is not used in VANET simulation but it has a great potential due to the performance with a large number of nodes.
- **OMNet++** [39] - in contrast to all the other simulators, OMNet++ is not a network simulator by definition, but a discrete event-based simulator framework used for general purpose simulations. OMNet++ only provides the framework necessary for developing network modules, as the modules necessary for VANET simulation must be implemented independently of the simulator. At this moment there is only one framework developed specifically targeted to the VANET simulation that is VEINS [40], [41], an open source Inter-Vehicular Communication (IVC) simulation framework composed of an event-based network simulator and a road traffic micro simulation model.
- **JiST/SWANS** [42] – JiST, like OMNet++ is a general purpose discrete event simulation engine written in Java at the Cornell University. As it was designed for general purpose simulations, the Scalable Wireless Ad-hoc Network Simulator (SWANS) has been developed specially for MANET simulation. SWANS is a simulator built atop of JiST platform. It was created because the existing network simulation tools at the time were not handling the current research needs. The

SWANS architecture is composed of several software modules that can be used to form a complete wireless network. Its capabilities are similar to most network simulators but it was specially designed for handling larger networks and at the same time achieves high performances. The simulator was also created, unlike all the others, to handle standard java network applications running over the simulated network.

2.2.3 VANET Simulation Frameworks

In this section we will describe some dedicated frameworks created to couple the network and traffic simulators in order to support the simulation of vehicular networks. The main concern of these frameworks is supporting the interaction of both network and traffic simulators while maintaining the temporal consistency (i.e. synchrony) of the overall VANET simulation, while supporting the implementation and testing of applications on each simulated node.

- **Traffic and Network Simulation Environment (TraNS)** [43], [44] – was developed by the École Polytechnique Fédérale de Lausanne and it is a GUI tool that integrates traffic and network simulators in order to generate realistic simulation of VANETs. TraNS is the combination of the SUMO and ns-2 simulators. TraNS has two modes of operation. In the first one the mobility traces generated by the SUMO are used as input directly in the ns-2 and it is used for a network centric simulation. The second mode is more of an application-centric mode and is used for testing applications that impact the mobility traces. This second mode requires the synchronization of the two simulators, as the messages exchanged between the nodes of the network will have a direct impact in the vehicles being simulated on SUMO using a specific module, created by the SUMO community, called TraCI. The project has been abandoned since 2008 and cannot be used anymore, since it does not support the most recent versions of SUMO; however, the project has his merits as it was one of the first projects to implement the concept of synchronizing both traffic and network simulation in order to test applications.

- **iTetris** [45], [46] – this framework was developed by the iTetris Project Consortium funded by the European Comission. It integrates both wireless communication and traffic simulation platforms in a very easily configurative environment. They use SUMO and ns-3 as their traffic and network simulators, respectively, and they synchronize them using a central control block named iTetris Control System (iCS). Although it can be considered as a high performance simulation platform as they developed their own ns-3 modules in order to optimize the network simulation core, this framework does not support the developing and testing of applications that would interact with the mobility model.
- **Vehicles in Network Simulation (VEINS)** [40], [41] – as previously referred, this framework is an extension of a OMNet++ module and was developed in Computer Networks and Communication Systems, University of Erlangen. VEINS is the result of coupling SUMO with the INET module of OMNet through a TCP connection, as SUMO works as an extension of the network simulator allowing the simulation to run synchronously. This is a very important feature as it allows the network simulator to interact directly with the vehicles being simulated by sending messages that will influence their path or speed. The communication protocol between the OMNet module and SUMO is once again TraCI, which allows a bidirectional coupling of both traffic and network simulation and consequently makes VEINS a great platform for simulation applications in vehicular environments.
- **V2X Simulation Runtime Infrastructure** – created by Daimler Center for Automotive Information Technology Innovations (DCAITI) and inspired by the IEEE Standard for Modeling and Simulation High Level Architecture (HLA). VSimRTI is a very generic framework that focuses on the synchronization of both network and traffic simulators. The framework offers the integration of different simulators and also the possibility of changing the simulator to be used without having to change the bottom line platform, which makes a great platform for testing the same scenario using different simulators with few effort and work. But what really makes this platform shine to our eyes is that it can emulate the environment of V2X applications in the vehicles, which makes VSimRTI the perfect platform for testing real applications.

After describing these platforms we can see that all of them have their strengths and weaknesses and it is important to analyze them very well before picking up a framework for our work. As the focus of our work will mostly be at the application level, the most important aspects we were looking for in these frameworks, is the integration of application simulation for vehicles and also the possibility of connecting with applications outside of the simulation environment

Stanica et al. [26] made a study for comparing some of these frameworks and identified six attributes that they considered to be the most important when comparing a VANET simulation framework, Routing Protocol, MAC Protocol, Safety Application, Information Dissemination, Traffic Management and Internet Access. From these six attributes, there are two very crucial for our work, Safety Application and Internet Access, and although we did take the others in consideration, we wanted a platform with a strong interaction between network and traffic simulators with a very precise mobility model that would allow us to test real applications in the vehicles.

Simulator	Routing protocol	MAC protocol	Safety application	Information dissemination	Traffic management	Internet access
TraNS	+	+	–	– –	– –	+
iTetris	+	+	+	+ +	+ +	+
Veins	+	+ +	–	– –	– –	+
VSimRTI	+	+	+ +	+	+	+ +

Figure 2-3 - Comparison of the different VANET simulation framework features[26]

The picture above represents the result of the study made by Stanica et al. [26], and we can see from the analyzed frameworks that VSimRTI is the one fitting the most of our interests, and therefore, it was the framework chosen to be used in our work to be described in the next chapters of this Dissertation.

2.3 Conclusions

In this chapter we presented an overview on Vehicular Networks basics and on VANET related application with special focus on those integrating mobile platforms. From our review it was possible to conclude that there is still almost no work that integrates mobile devices with the vehicular network resources. Most services and access to the

network still depend on the On Board Systems that will simply present data to be consumed by the mobile devices, but not expose the communication services in a standard way e.g. through a high level architecture abstracting the VANET specificities under an API accessible at mobile application level. Such architecture would need to abstract the transport and network layers so the applications would be communicating at an application level and without having to implement VANET specifications.

At the same time, even if there is still no standard VANET simulator, there are a lot of frameworks that couple both Network and Traffic simulators with the bonus of having an application runtime platform in order to simulate real applications in VANET scenarios.

In the next chapter, we will present the solution developed in order to solve the problem presented in the introduction section.

3 REINVENT: Conceptual Architecture

In this chapter we will introduce REINVENT, an architecture to provide an abstraction layer between VANET Transport layer and the mobile applications. REINVENT is supported on a REST architecture style and relies on messaging for data transfer.

In the first section we will describe REST, and afterwards we give an overview on Message-Oriented Middleware. Finally, the last section will give an overview of our conceptual architecture and where the concepts explained previously will be fitting.

3.1 REST as an architectural style

Representational State Transfer (REST) [47] is an architectural style for designing networked applications. It relies on stateless and client-server communication protocol and it is commonly used over the HTTP protocol. The idea behind REST is to use simple HTTP calls instead of using complex protocols like RPC or SOAP. REST based applications use HTTP requests in order to read, delete and post data, thus, REST supports all four CRUD (create, read, update, delete) operations.

According to Fielding et al. [47], the REST architecture style is composed by six constraints. In order for an architecture to be deemed a REST architecture, it must satisfy all the constraints defined:

- The first major constraint is the client-server constraint. Based on a computing principle called separation of concerns, it requires the existence of one or several clients that do requests that will be received by a server that consequently can produce a response to the client. This constraint induces a set of properties to the architecture like portability, resolvability and scalability.
- Stateless communication constraint is related to every request done by the clients that must contain all the information needed in order to be interpreted. This constraint increases the visibility, reliability and scalability, but also induces a decrease of performance because it requires the messages to be larger.
- Layered system constraint defines hierarchy layers for components limiting the communications to their direct neighbors. Basically, a client cannot know if he is

connected to the end server or simply to an intermediary. This constraint increases the scalability by using intermediary components that can act as caches or load balancers. It can induce to reduce the performance as the number of intermediaries increase between the client and the server.

- Caching constraint is related to the capability of the messages to be labeled as cacheable in order for the clients and intermediaries to be able to cache and re-issue messages, increasing performance and efficiency.
- Uniform Interface constraint means that all the component interfaces must be the most generic possible simplifying and decoupling the architecture in order for each component of the architecture to evolve independently. This constraint increases the visibility and scalability of the system, but comes with the down side of reducing the efficiency due to the generic data types.
- Finally, the last constraint and also the only optional one, Code on Demand constraint, allows a client to execute and download code from the server. The server can extend the clients functionalities. This constraint increases the scalability of the system as it delegates work to the clients, but it has the tradeoff of reducing visibility created by the client code, as it could be hard for other components to interpret.

These are the basic rules in order to create a REST style architecture, but it is also important to understand that there is no standard created for REST, as these constraints are just guidelines to create an architectural system for network applications.

3.1.1 Resources and Resources Identifies

The representation of information in REST is called Resource. Any kind of information that can be named can be a resource: an image, a table, a service, or even a collection of resources. A resource can be interpreted as the conceptual mapping of entities. Each resource is referenced with a global identifier that can be a URI for example. In order to manipulate these resources, all the components of the system communicate via the standard defined REST interface and exchange representations of the resources. The representations are basically a sequence of bytes representing the actual document or photo, plus the meta-data for describing the bytes for the receiving entity to interpret the

data. In order for an application to interact with a resource, it needs to know three aspects: the identifier of the resource, the action to be made on the resource, and finally, to understand the format of the information of the resource (the representation) if it returns any information [47].

3.1.2 REST as a solution

Relating this concept to our work, we will use the REST architectural style to provide an abstraction between our applications (the REST client) and the module (the server). Conceptually, the applications will send a request to a given resource on the module either for relaying a message, or to perform a simple request for information on the network, or the list of entities in the module naming service.

3.2 Message-Oriented Middleware

According to Curry [48], Message-Oriented Middleware (MOM) is a software or hardware designed to provide communication methods between heterogeneous software entities. MOM can be defined as any middleware infrastructure capable of providing messaging services. A software client implementing a MOM should be able to send and receive messages from other software clients using the same MOM, even if they are implemented in different technologies. Although MOM requires the clients to connect to one or more servers that will act as intermediaries, the model used in this middleware is a peer-to-peer relation between two clients.

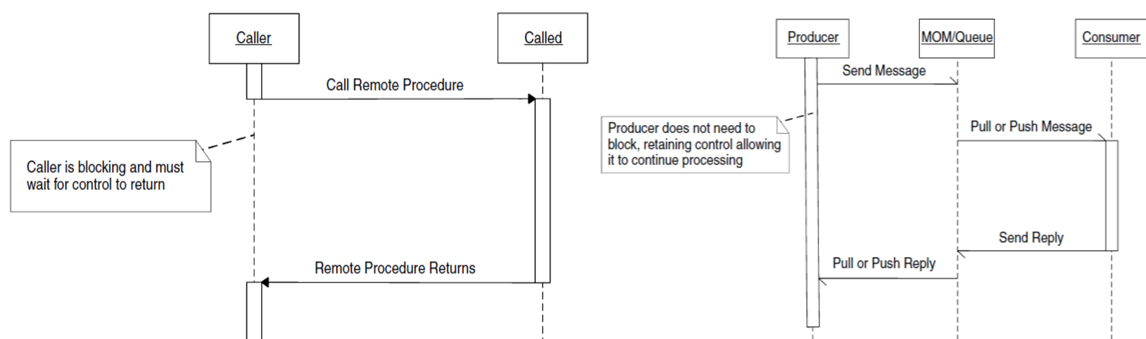


Figure 3-1 Synchronous and Asynchronous Models [48]

There are two interaction models commonly used in MOM, synchronous and asynchronous. In a synchronous model, when a method is called, it requires the caller to block and wait for the method execution to end. This kind of systems does not have

independence of processing control as they require other entities to give back control of the process.

The asynchronous model allows the method caller to remain in control of the process while the method is being executed. The caller entity can continue his processing regardless the state of the method execution. This model requires an intermediary to handle the requests; this intermediary is usually represented by a message queue. Although asynchronous models require more complex implementations, they allow all the clients to keep their processing independence. They can continue their processing regardless of the other participants.

3.2.1 Message Queues

Message Queues are a fundamental part of a MOM system. MOM clients should be able to send and receive messages from a queue that provides store ability. Queues are a crucial point of asynchronous MOM systems. Using message queues, clients can send messages to a destination without having to block or wait for the message to be handled and delivered, as it will be stored in the queue to be fetched by the destination client. The standard queue system found in most of Message Queues is the First-In First-Out (FIFO) queue; the first message stored in the queue will be the first message to be retrieved from the queue retaining the arrival order.

There are two messaging models supported by most of the message queues:

- **Point-To-Point** – The Point-to-point (PTP) messaging model ensures that a message is delivered only once to a single consumer providing a straightforward asynchronous exchange of messages between entities. When a message is consumed, it is removed from the head of the queue. Although this type of message model only supports one consumer (since the message is removed from the queue after being consumed), there is no restriction about the number of producers that can connect and publish in the queue.
- **Publish/Subscribe** – The Publisher/Subscribe model is more powerful and complex than PTP. It allows one-to-many and many-to-many communication relations so one message can be sent to potentially an infinite number of receivers. The clients publish messages to a channel or topic that will be managed by the MOM system. These channels and topics can be subscribed by clients that want to

receive messages from those topics. The MOM system will be responsible to route the messages related to the subscribed channels to the receiving clients.

3.2.2 Standards

There is no standard for Message-Oriented Middleware. There are a lot of different implementations, each with its own API and tools.

The Advanced Message Queue Protocol (AMQP) [49] is an application layer protocol for MOM. It defines the protocol and formats to be used in both client and server. AMQP was created and defined to offer flexible routing and includes the most common message models like PTP and Publish/Subscribe and request-response. There are AMQP APIs developed for several languages like Java, C, C++, C#, PHP, Python or Ruby which makes AMQP a very versatile MOM implementation.

The Simple Text Oriented Messaging Protocol (STOMP) [50], formerly TTMP, is a text based protocol designed for MOM. It provides an interoperable wire format allowing STOMP clients to communicate with any message broker that implements the protocol. STOMP is a frame based protocol, with frames modeled on HTTP. A frame consists of a command, a set of optional headers and an optional body. STOMP is text based but also allows for the transmission of binary messages. There are a different number of STOMP implementations in different languages like Java, Python, Ruby, C++, Perl or OCaml.

The Java Messaging Service (JMS) [51][48] is a Java MOM API. It allows applications based on all the Java platforms to create, send and receive messages. JMS provide the means for communication between components of a distributed system in a reliable and asynchronous way. JMS supports both PTP and Publisher/Subscribe messaging models. This service has the downside of only being supported by Java based systems.

3.3 The Architecture

The main concept of REINVENT is to abstract the transport resources behind a REST interface with a module that will enclose the support of a messaging system already coupled with the transport layer specifications. This abstraction will prevent the application layer from having to deal with those specifications like PSID's or channel numbers. The module incorporates the messaging service that will encapsulate the application messages

in the specific transport layer message protocol. It will also have a logical naming service translator that translates names into addresses, much like programs like Skype do: people are represented by names or alias, but in the end they are translated to addresses.

We can observe in Figure 3-2 the concept of our solution: we have a representation of a logic application interfacing our module through a REST interface that will be working as a proxy to the transport domain

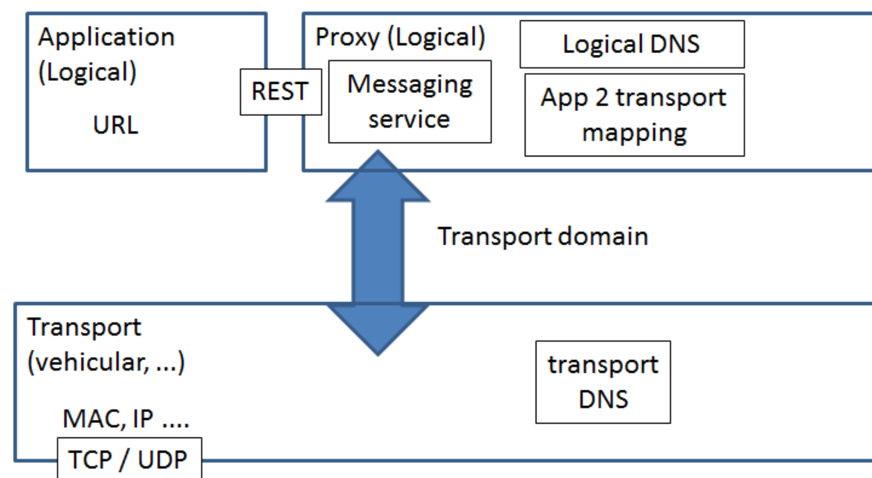


Figure 3-2 – Conceptual Architecture of REINVENT

The objective of the messaging service is to maintain the connection between the applications and the lower transport layers, by translating all the incoming messages from those layers, and also receiving the messages from the requests made through the REST interface. Then, if needed, it prepares the data to be sent through the transport layer. The messaging service should not only be responsible for keeping a connection state, but also to notify the applications if the connection is not available and make sure that no messages are lost in the process of reconnection providing reliability to a certain point.

The logical name service is a module that will work like a persistence unit, keeping all the network related data from the users known to the applications implementing this module. The applications will only need to work with real names or alias, since the logical name service will keep track of the network identifier of that entity. These identifiers can be in several formats concerning the network being used on the lower layers: it can be an address or a vehicle identifier.

3.4 Conclusions

In this chapter we explained the concepts of a REST-based architecture and Message-Oriented, that supports the REINVENT architecture concept that we describe afterwards in more detail.

It is important to note that the architecture described is abstract design, as we idealized the REINVENT architecture as a system to be implemented in any high level language that integrates with layers of low-level communication. Although the architecture has an abstract character, we refer specifically to solutions like REST and Message-Oriented Middleware, since they are not solutions restricted to any technology or language.

In the next chapter, we will present the proof of concept implementation of REINVENT in Android mobile operating system.

4 REINVENT: an Android based proof of concept

In this chapter we will describe the implementation process of REINVENT in Android operating system. REINVENT uses Android's Content Provider to implement the REST abstract interface of the conceptual architecture. A description of the REST API is provided for mobile applications to access the vehicular network resources together with some illustrative examples. As MOM provider, RabbitMQ [52] was the selected solution for the messaging service in REINVENT.

As a proof of concept, we present also two applications using the REINVENT module to access the VANET resources: VNChat and iThere. The first application, VNChat, is a message exchanging application that allows the users to exchange messages using the VANET network. The second application, iThere, is a type of social network for vehicular networks. The chapter ends with the implementation details: we describe the main classes and structures that compose REINVENT.

4.1 Android Platform

Android Incorporation was created in Palo Alto, California in 2003 by Andy Rubyn, Rich Miner, Nick Sears and Chris White and bought later by Google in 2005. In 2007, the project was presented to the public by the Open Handset Alliance, which was composed by a set of companies, HTC, Samsung, Qualcomm, Texas Instruments and Google. The first public phone running the Android Operating System was released in 2008, and it was the beginning of a market dominance operating system as today Android is the most used mobile platform in the market [53], [54].

Figure 4-1 – Android Stack Overview [55]

Android has a very robust software stack composed by a Linux Kernel, a middleware layer and a virtual machine called Dalvik Virtual Machine in order to create an environment to run Java applications as well as some native applications like the messaging system or the browser (Figure 4-1) [55]. It is also an open source operating system with a very large community of companies and developers.

The Android platform also provides a software developing kit (SDK) that includes all the API libraries and developer tools necessary to create and test applications for the Android OS, and it is compatible with all the major Operating Systems like Windows, OS X and Linux. Unlike most mobile platforms, due to being a very open platform, it provides the tools for creating very good looking applications as well as also taking advantage of accessing all the hardware components directly like the GPS or Bluetooth. The backhand of being such a wide open platform with so many companies involved is that there are a very large number of devices with different specs in the market, which can cause a lot of compatibilities when creating new applications, as they should be compatible with not only different screen resolutions, but also with different hardware specifications like processors and internal memory [56], [57].

In the next subsections we will describe Android Applications Layer concepts that were the base of the concept created and developed in this Dissertation. We will describe the Content Providers as well as the different approaches where they are used in applications contexts.

4.2 Android Content Providers as REST application interface

Content providers are a well-known concept of the Android OS commonly used for managing and accessing data sources (Figure 4-2). They are used to encapsulate data in order to provide it to applications through a simple interface, which makes them the obvious solution when it comes to share any kind of data between applications [57], [58]. The most commonly usage for the Content Providers is to share a database, for example the phone contacts, between several applications.

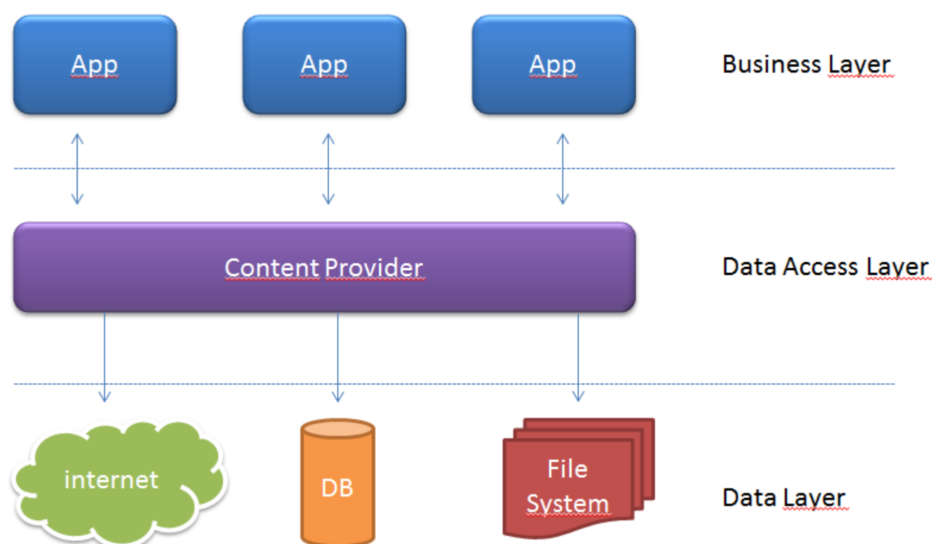


Figure 4-2 – Android Content Providers Overview

The content provider API allows applications to query the OS for any kind of data using Uniform Resource Identifier (URI), very similar to the way web sites request information to the Internet. The application querying a certain URI does not know the source or who will be providing the information, it simply presents the URI to the OS that will be responsible to start the application responsible for providing the information related to the given URI. This is a great feature in terms of performance, as it is the Operating

System that is responsible for managing all the existing providers as well as managing the access of each application to a given provider.

The content provider API also provides methods for creating, reading, updating and deleting the shared content following the model used by REST architecture, which means the applications use an URI-oriented requests model. These requesting methods translate into direct methods that must be implemented when creating a new content provider [58]:

- **Query** – Used to request specific data and returns it to the caller.
- **Insert** – Used to insert new data into the specified URI.
- **Update** – Updated existing data in the provider
- **Delete** – Deletes existing data in the provider
- **getType** – Returns the MIME type of data in the content provider
- **onCreate** – Called by the application main thread to initialize the provider. It is not recommended to perform long operations in this method.

On the application side, a ContentResolver object is used in order to access and process the Content Provider retrieved data. It also handles all the inter-process communication needed to connect the application to the content provider. It is also important to refer that applications must request specific permissions for accessing the desired provider in the Android Manifest file.

4.2.1 Content URI

The Content URI is the identifier of the data in the provider. It is composed by the identification of the provider that is called Authority followed by path referring to a table where the data is translated to by the content provider. A provider is usually composed by a single authority that is used to identify the provider by the Android OS. It is a common use, in order to avoid conflicts, to use an Internet Domain ownership (in a reverse way) to identity the base of the content provider followed by the specific name of the provider. After defining the authority, it is necessary to define the paths available to the provider in order to access the data structure represented by the provider. These paths follow the same path as accessing a certain section of a website, like the following example taken from the Android Developer website in the Content Provider description section:

- content://com.example.app.provider/table1: A table called table1.
- content://com.example.app.provider/table2/dataset1: A table called dataset1.

- content://com.example.app.provider/table2/dataset2: A table called dataset2.
- content://com.example.app.provider/table3: A table called table3.

It is also possible to request specific values of a given table using the follow URI,

- content://com.example.app.provider/table3/1 for the row identified by 1 in the table 3

Or even request all the values of a given path using the following wildcard character,

- content://com.example.app.provider/table2/* that would return dataset 1 and 2 of table2.

4.2.2 Why this architecture?

It is very important to understand that, due to the limited processing capabilities of the smartphone devices (maybe not so limited nowadays, but still needs to be taken into consideration), the thread handling the interface should be released of any long operations as it needs to be free for handling all the real interface events. These long operations can be Network Requests, as they can last for an uncertain amount of time or simply long processing of data like accessing a database. To minimize the execution of any long operations on the user interface layer, it is necessary to create a very structured and multi-layered architecture for handling these operations like Dobjanschi [59] proposed in the 2010 Google I/O session, where they proposed a solution for REST Client application using Content Providers as the platform for managing the data that would needed to be written or read from the persistence layer. The problem of most REST Client implementations is that they are subject to the operating system shutdown the process, due to the long operations made in the network and consequently, the data is not persistently stored resulting in faulty data or even the loss of some data.

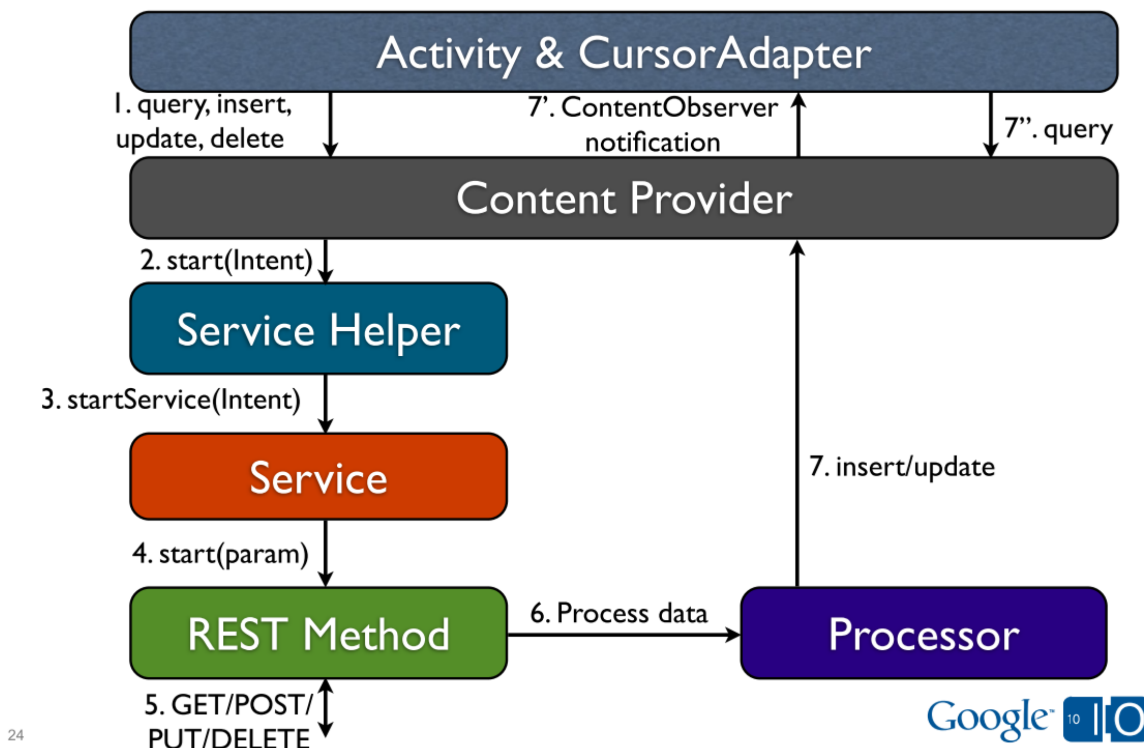


Figure 4-3 - Google I/O REST Architecture using Content Providers[59]

The solution proposed is presented in Figure 4-3 and it can be divided into four different sections that will be explained in order to understand how this REST Client is used and what are the advantages of using these types of architectures when creating applications.

Rest Method

This is the entity that is responsible for the connection with the external resource. It runs a working thread that prepares the HTTP URL and HTTP Request Bodies, and executes/processes the HTTP transaction/responses. It should be also responsible, or simply delegate the processing to the Processor if it is too complex, for selecting the requested content type from responses like JSON or XML.

Processor

The processor is responsible for handling the data returned from the REST Method execution, which involves selecting the important data types from the HTTP Response packages. It is the process responsibility to perform all the persistent layer actions required for the received data on the content provider. As it was explained before, Content

Providers support all the CRUD (Create, Read, Update and Delete) operations which are the methods that the processor calls upon the provider.

Service

This section is composed by two entities, the Service Helper and Service. This block is responsible for exposing a simple asynchronous API to be used by the Content Provider. When the content provider needs to send/get new information from a REST Method, it starts an intention for executing a specific method and it is the job of the service section to check if there is already a request for the same method; if not, it prepares the method call with the specific parameters and starts the method. It is a one way process as the result from the execution of the method is delegated to the Processor, as was explained previously.

Content Provider

The content provider plays a major role in this architecture as it is the middleware connection of the activity to the network data as well as working as a persistence unit. The user interface can make several requests that should be handled by the content provider. In case of the requested information is already in the persistence layer, it notifies the activity layer; if the information is not present or needs an update, it starts an intention for requesting/sending new information that will be handled by the Service layer.

Activity

This is the Interface layer and, according to the Android design lines, this layer should not handle any long running operations nor any network operations so, the activity layer only performs requests methods to the content provider and listens for notifications of the provider on specific content via ContentObserver [60], a class that receives call backs for changes on specific content on a content provider.

The project presented by Dobjanschi [59] is a very detailed example of a good, and different, usage of the Android content providers in order to create a very structured architecture for handling a REST interface. This project has some resemblances with our architecture as they delegate all the network specifications and operations to a content provider releasing the application from all the long task like network calls of database reads and writes.

4.3 REINVENT Android Architecture Overview

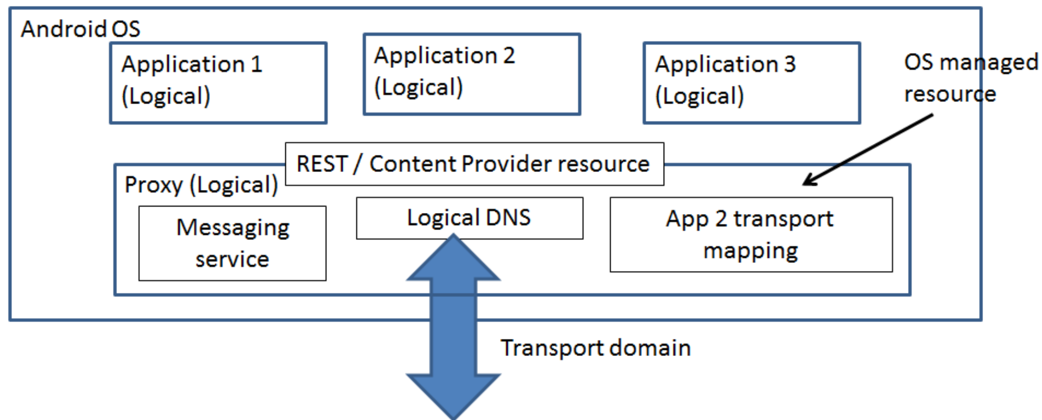


Figure 4-4 – REINVENT Android Architecture

In REINVENT, Content Providers are used to abstract both network and transport layers under a REST unified interface. REINVENT content providers will supply applications with a fully structured and well defined service API to interact with these lower layers. Using the URI concept we can provide global identification for either the resources, like accessing the Naming service information, or to the transport services for sending/receive messages. Besides providing an interface to the applications, content providers are also a very good solution in terms of performance, as it is a resource handled at the operating system level and consequently delegates to it all the concurrency control increasing the performance of such operations. Due to the fact of being a OS level resources makes it also a sharable resource among the applications which means that REINVENT will be available to all applications and most importantly, information like the naming service will be available to all applications to use and modify as well as all the services that can be used by all the applications in a concurrent way.

The logical naming service will be implemented as an internal SQLite Database keeping all the information of the entities known by the application. That information will be used to map the entities names used by the applications real network identifiers when it comes to build a message to be sent to the network.

The messaging service will be implemented using an Advanced Message Queuing Protocol framework called RabbitMQ that will be responsible for handling the incoming messages from and to the applications.

Figure 4-5 shows an overview of our concept applied to the Android OS context, where several applications existing in the operating system, will be provided to our REINVENT module interfacing their communication with the transport domain. In order to represent the REST architecture explained in the previous section we rely on an operating system managed and public resource called Content Providers that can be seen as REST interface implementation on the Android System.

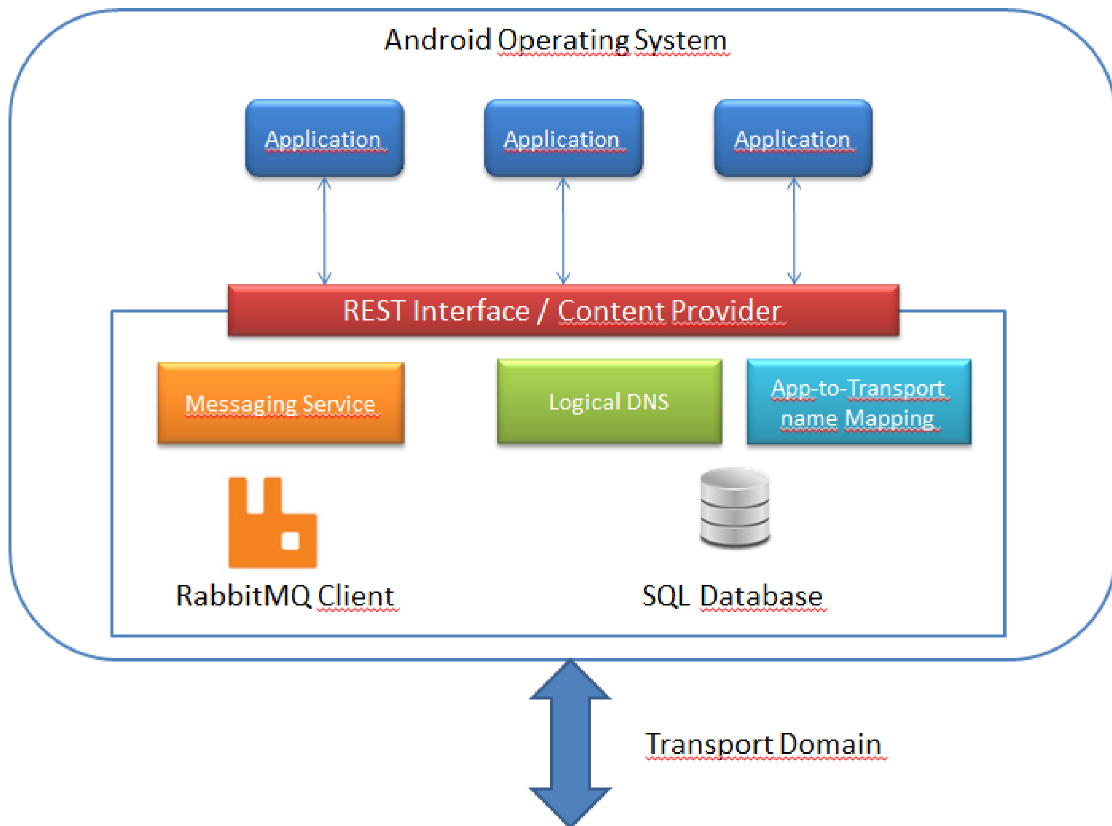


Figure 4-5- REINVENT Architecture

4.4 RabbitMQ as a messaging provider

In REINVENT, we used RabbitMQ framework to support the messaging service needed in our module.

RabbitMQ is open source message broker software that implements the AMQP standard (Figure 4-6). The server is written in Erlang and built on the Open Telecom Platform framework for clustering and failover. It was recently acquired by VMware, who now supports and develops further versions of RabbitMQ. The project includes client libraries for Java, .NET and Erlang, although there are a lot of clients available from other sources.

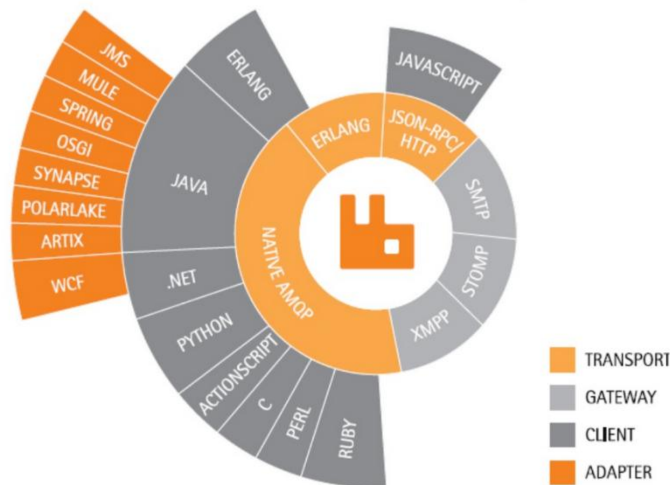


Figure 4-6 – RabbitMQ Architecture Overview

Why RabbitMQ? Since there are a lot of solutions for messaging, the decision to go with the RabbitMQ was made mainly because it is a very lightweight solution, and we were aiming to get it to run in a relatively low computing capacity device such as the mobile devices or even in an on-board vehicle device. Since RabbitMQ implements AMQP, which is not a language strict protocol, it automatically makes our architecture open to interact with systems using different languages than Java (which is the language used for developing Android applications). The queue system of the RabbitMQ is a very important feature to take in consideration, as it will play a major role in REINVENT and it will allow the applications created that implement our architecture to retrieve messages received while not connected to the network, as they will be kept by the RabbitMQ server in a proper queue, so no messages will be lost while the RabbitMQ server is up (which should be always).

4.5 REINVENT services

REINVENT provides a set of services to the applications. We tried to create a simple and generic interface that could be used by every application and not to be strongly related to any particular application domain. Being implemented using Content providers and relying on the application to maintain application specific status, these services can be used by several applications simultaneously – concurrency managed at Android OS level.

In a glance, REINVENT API offers a complete set of methods for applications presented in Table 3 with a brief description (full details in annex section):

- The CRUD (create, read, update, delete) operations over the naming service.
- The necessary methods for the applications to use the messaging service.

Service	*Description*
GetAllUsers	Returns all users from the naming service
GetUser	Returns a specific user from the naming service
InsertUser	Inserts an entry in the naming service
DeleteUser	Deletes an entry in the naming service
UpdateUser	Updates the information of an entry in the naming service
NewMessage	This method does not return any value, it is used to listen for new incoming messages
SendMessage	This method is used to send a message using an entry from the naming service as the destination.
Close	This method is used to stop the listening thread of the provider
GetNewMessages	Returns any new messages from a given type.

Table 3 - REST Interface overview

The naming services entries are mapped into URI that can be used for each of those operations, so, if we want to update an already existing entry in the naming service we would do as follows:

- Create a Content Resolver object
- Create a Content Value where the user would place the information of the updated entry

- Call the *insert* method over the Content Resolver using the following URI
  - *content://AUTHORITY/users/# ,*where the # would be the id of the entry to be updated.

If applications want to receive any new messages, the API has a dedicated URI for sending notifications when a new message is received and processed in REINVENT module, so applications can simply observe that URI, and when they receive the notification of a new message, they can request new messages of a given type if they exist. We choose to do a generic new message URI as we wanted the module to be as generic as possible, and not to be dependent on the messages types leaving the parsing of the messages to the application. Finally, the API provides a method for sending messages using a very similar method to the example presented previously for updating an entry of the naming service. If the message was to be directed to a specific entry, the URI to be used would be similar to the previous one by just adding */send* to the end which would be as follows:

- For sending a message to a specific user, *content://AUTHORITY/users/#/send*
- For sending a message in broadcast, *content://AUTHORITY/users/send*

4.6 Developing Applications using REINVENT

After developing the REINVENT module, it was necessary to apply it to real applications so we could test it in several scenarios. In this section we will describe the two created Android Applications implementing our REINVENT module.

The first application, VNChat, is a messaging application where the user can send messages through the network to other user. The second one, iThere, is a VANET Social Network, where the application users will receive visual information of the friends driving around the user.

These applications were a very important part of the work process, not only to prove that REINVENT module works, but also to test the usage of the module by more than a single application and the way the module handles the concurrency of applications. Both applications can be considered very simple. Their main objective is to cover all the possible use cases of REINVENT and also, like explained before, to test the concurrency of several applications to the module.

4.6.1 VNChat

This application can be inserted in the Infotainment and Other Applications domain (according to the State of the Art taxonomy), and it was created aiming to be used not by the driver itself, but by the car passengers as it requires a strong interaction with the Android device.

The application is composed by four activities that will be explained and described in order to understand the layout of the application and all its functionalities.

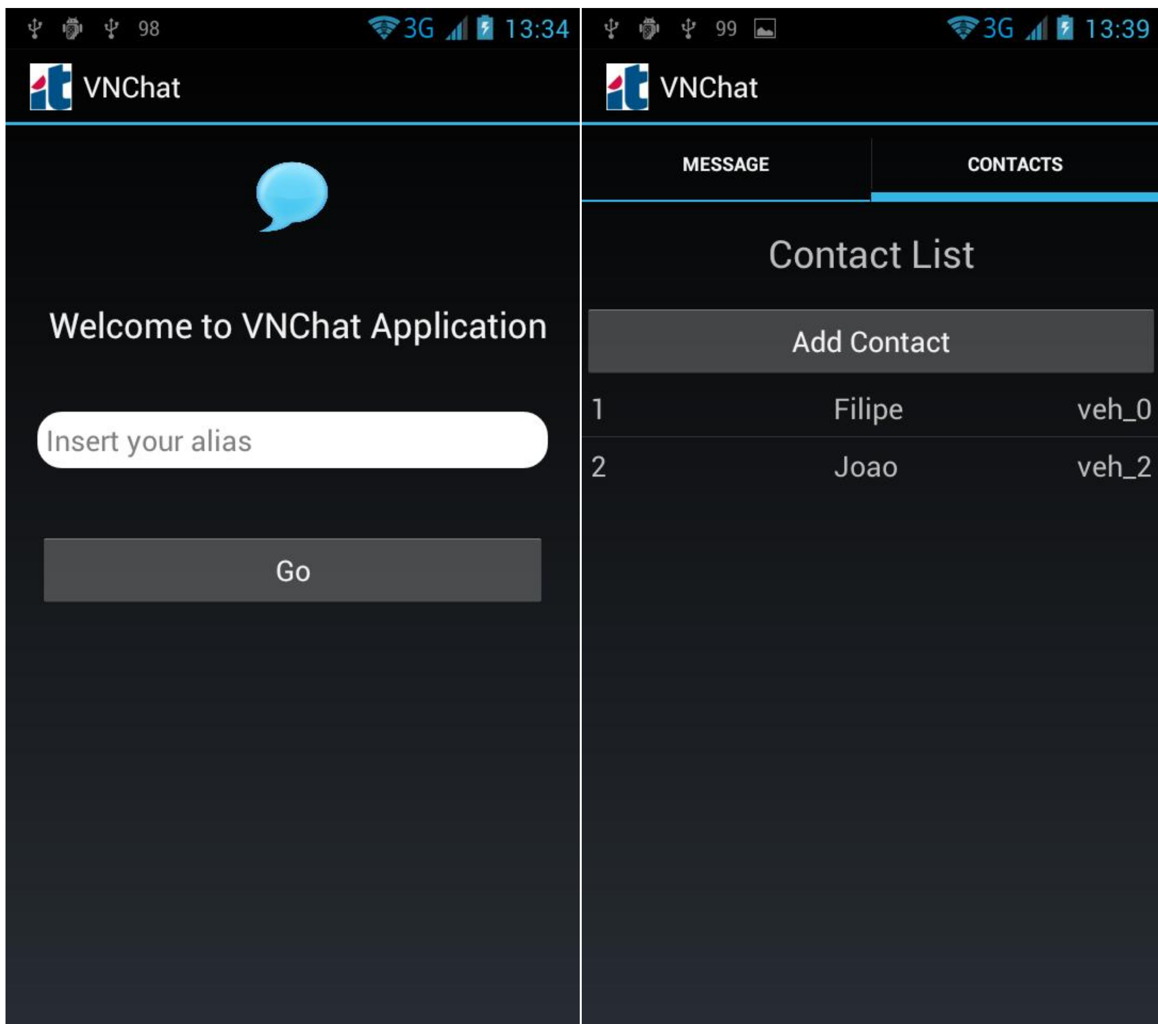
The first activity is the Welcome Activity that is the starting activity of the application; it can be seen as a simple login screen as the only interaction with the user is an edit text field that must be filled with the desired identification for the user of the application. That identification will be used to identify the user messages, and it will work as your “nickname” and it will be associated with the vehicle identification which is handled by REINVENT module. This identification will be sent to REINVENT as it will be used to identify all further messages sent to the network, even if they are from other applications using this module. After the user fills the text field with the desired identification and proceeds to the application itself, a Tabbed screen shows up with two tabs, one for messaging and one for contacts.

The Contact Activity is basically a contact list of the naming service entities, where the user can consult the list of entities presented in the naming service with their name identification and their vehicle identification. From this activity, the user can choose to add a new contact that takes the Add Contact Activity that has a simple form for the user to fill with the Name and Alias of the user to be added. These activities also interact with the REINVENT module, as they perform insert and get operations in order to insert a new contact to the naming service and return the list of all contacts available respectively.

The final activity, and probably the most relevant activity of this application is the Messaging Activity where the user has the list of the contacts available in the naming service. The activity has a text field where the user can type messages that will be sent through the REINVENT to the selected contact. The activity also has a list view where all the received messages will be shown. This activity is the only one that uses the transport layer services of the REINVENT module for sending messages, and also to listen for the notification of new messages and consequently get those messages from the module. The activity also requires the information from naming service in order to select the destination

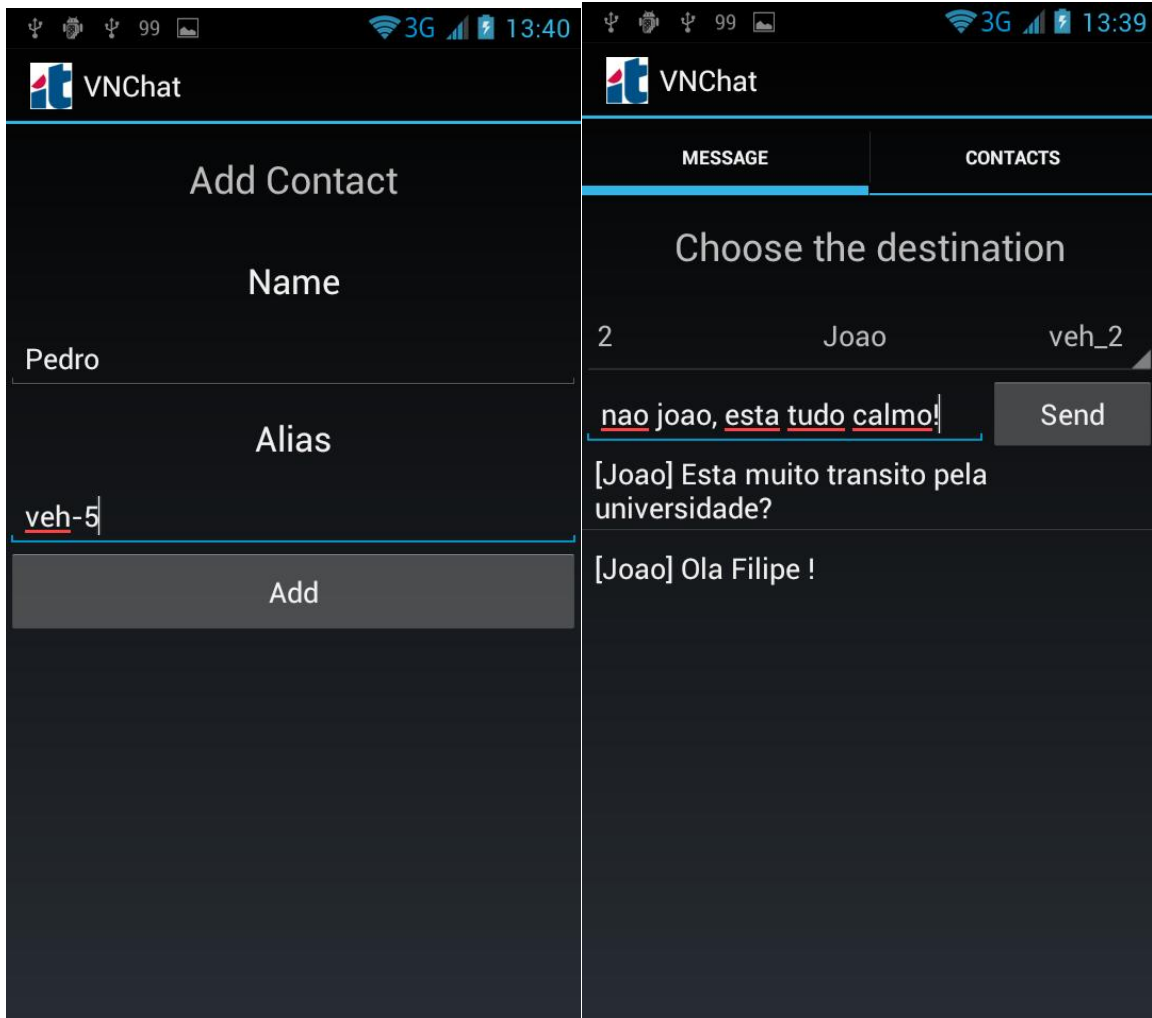
contact to send a message. The messages received and sent by this application come with the *TXT* type identification, so the application can only fetch the messages from the module that is concerning this application. This activity itself is probably the most important to look in this application, as it covers most of the services supplied by the REINVENT API.

The Figure 4-7 and Figure 4-8 present all the activities of the application. These screenshots were taken in a real conversation using two Android devices. The main use case of this application, Instant Messaging, is covered in the ITS basic set of applications definition [1], [12] as it should be a base application to be developed for vehicular networks



(A) (B)

Figure 4-7 Welcome Activity and Contact List Activity



(A) (B)

Figure 4-8 Add Contact Activity and Message Activity

4.6.2 iThere

The second application developed was the iThere Application that can be inserted into the location based category. It was designed to be used by both the driver and the passenger, as it requires almost no interaction with the device because the application plays a purely informative role.

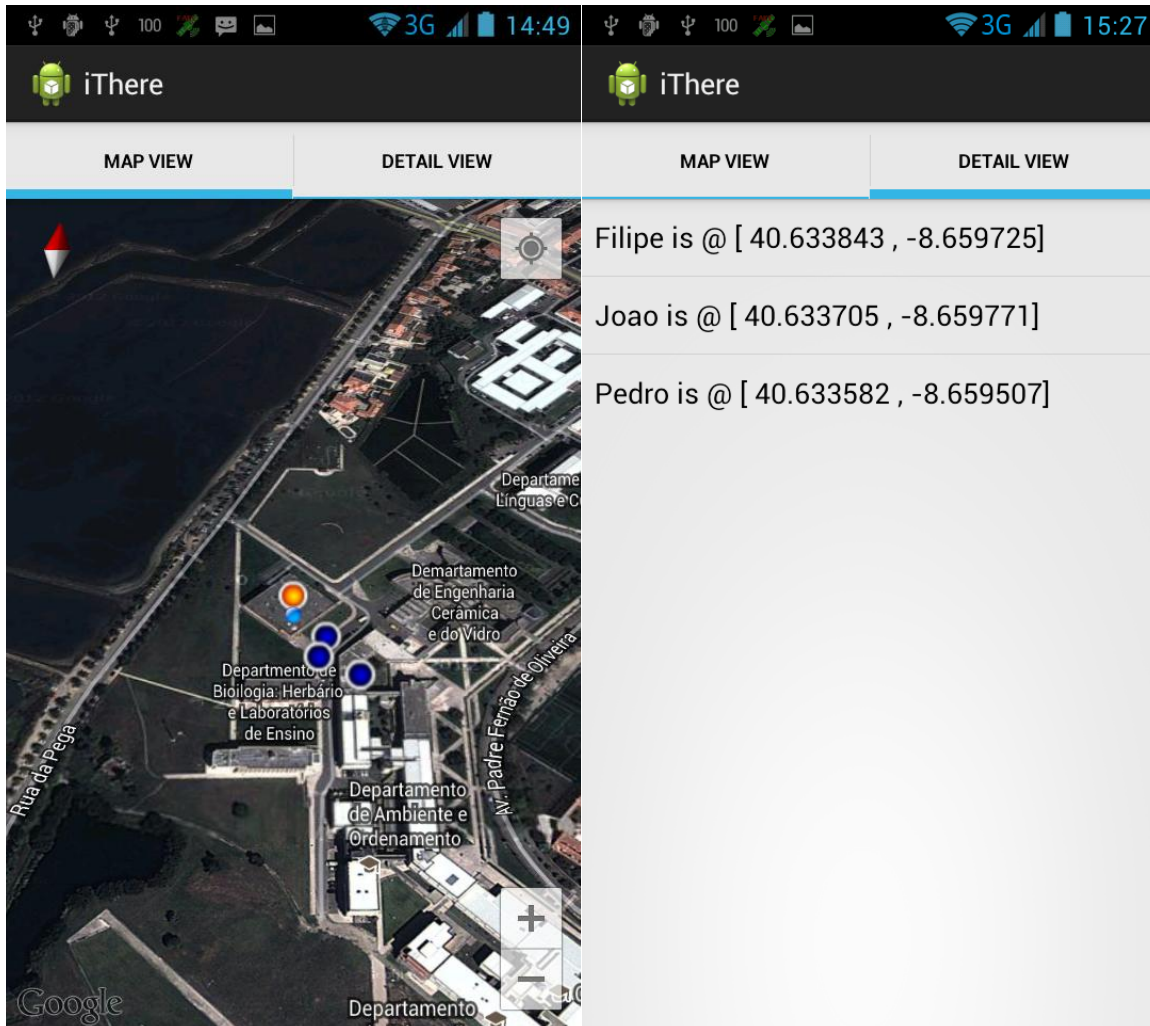
The objective of the application is to inform the user of the friends using the same application that are in the near location of the user. It works almost as a social network for vehicle networks. The application works by broadcasting the user current location every time it changes in a defined time space with a specific message type, so all the users of the

application can pick it up and then synchronize and update their own applications with the latest location.

The application is constituted by two activities (Figure 4-9). The first one is the Map Activity, where a Google map is shown and centers on the user current location that is marked by an orange dot, which can be clicked for detailed information of the local. The application will continue to follow the user location and keep the map centered on it, and will also show blue markets that represent the location of the friends around. They will only be shown in a one kilometer radius and like the orange marker, can be clicked in order to get more detailed information of the friend and his location.

The second activity is the Detail Activity, which is the representation of map information translated to a list containing the location of the nearby friends. This list contains the name, vehicle identification, latitude and longitude, and it could be very useful to have an overview of who is actually around you since the map view only shows the markers it would require clicking on each one to see the detailed information which can be consulted in this activity.

In order to keep both activities equally updated without having redundant information, we developed a singleton structure for managing the friends. The application will initially get all the naming service entries and create one Friend, a structure created similar to the one being used by the naming service with the location information added, for each naming service entry that will be added to the singleton structure created that can be then accessed and updated in both views keeping the information in a single structure instead of passing the information to each activity every time the user decides to switch between them. This structure has all the required methods for setting and getting attributes of each entry of the structure.



(A) (B)

Figure 4-9 MapView Activity and DetailView Activity

4.7 Implementation Details

In the following sections we will provide insight about the implementation details of REINVENT. We will present and describe the main structures that compose the architecture.

4.7.1 Main Classes and Structures

REINVENT is composed by four classes. The first class implements the content provider method which is also the core class of this architecture. The other three classes are helping classes, one to deal with database connection, one for messaging structuring and finally one describing User related fields.

4.7.2 Network Provider

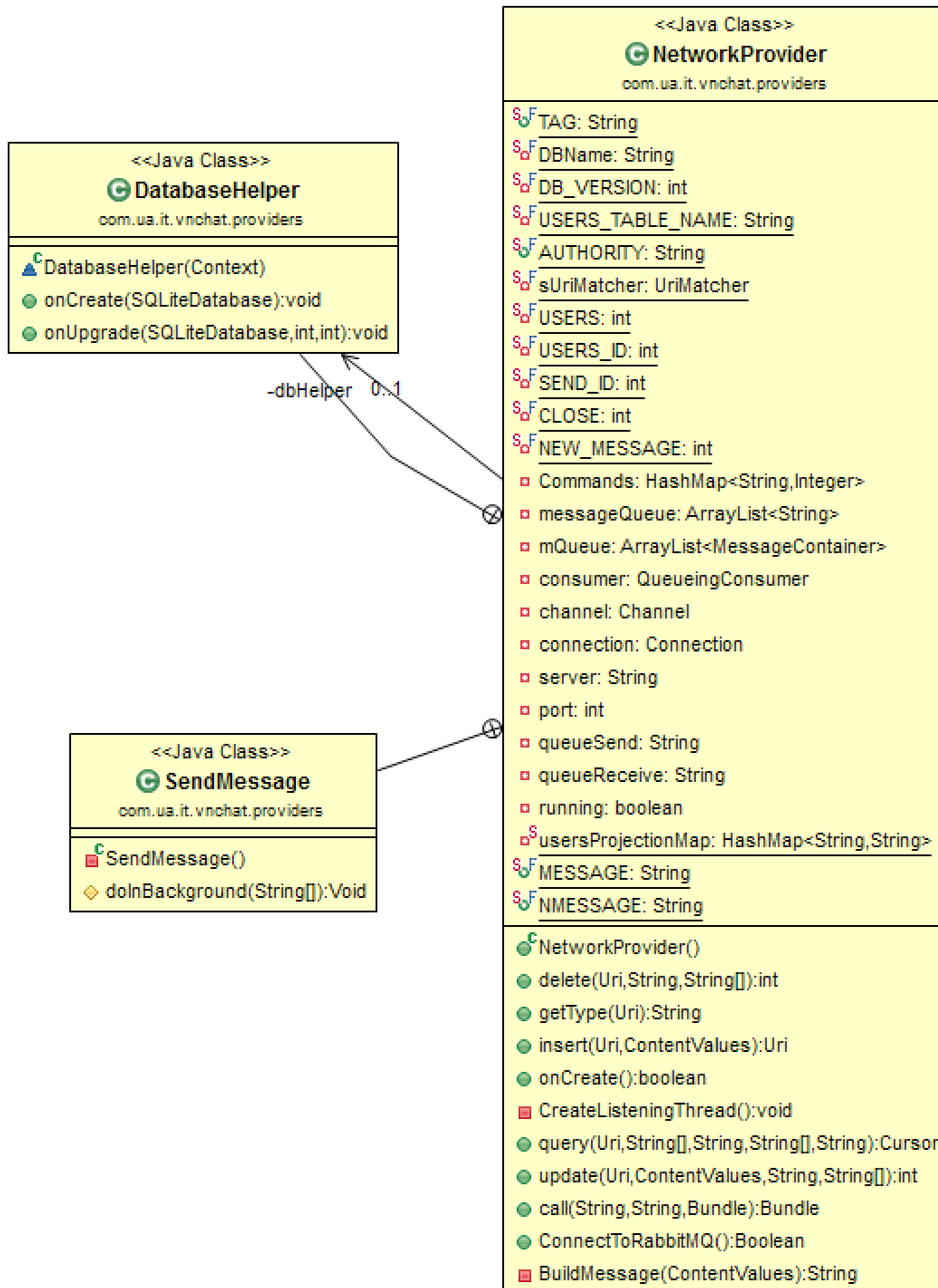


Figure 4-10 – NetworkProvider core class

The Figure 4-10 represents the class diagram for the core class of the module implemented. The Network Provider Class extends the Android abstract class Content

Provider, which requires the implementation of some methods that will be explained, and what is their purpose in our architecture since we are applying the Content Providers context in a slightly different way of the common usage that is a support for persisting data between several applications. In our architecture, the main focus of the content provider is to provide a set of methods and services through a REST interface in order to abstract the transport layer. It also works as a common persistence unit, as the naming service will be implemented as a database that will be available to all applications implementing this module.

We will now explain the main methods of this class, as well as some of the most important attributes shown in the previous diagram:

- OnCreate() – This method is called to initialize a content provider. In our architecture it is responsible for creating a listening thread that will be listening for new messages from our messaging system. It is also here that we open the connection to our naming service database in order to fetch all the users to a List so they can be easily accessible.
- Insert() – The objective of this method by default in the Android Developers SDK [58] is to insert new data to the provider. In our architecture, it keeps that functionality as applications might want to add new entities to the naming service. But in our architecture, this method presents two additional functions; it is used to send a message to an entity specified in the URI (that will be explained in the next section when we describe the available methods by the REST interface and how to access them). The process of sending a message is made by instantiating a new SendMessage class that is a normal java class that extends the Android abstract class AsyncTask. An AsyncTask is commonly used in Android development when we want to perform a network operation, or simply a long operation that must run outside the User Interface main thread, in order to prevent the decrease of user experience by locking the interface while waiting for a result. Finally, this method is also used to terminate the listening thread when called with the proper URI. It could be useful to stop the thread if, at a certain point, we do not desire to receive any more messages or simply to terminate the thread along the closure of the application.

- Query() – This method, like recommended by the API, is used to return data to the caller of the method. In our architecture it is used to get information from the naming service concerning a specific entity, or simply to return all the entities in the naming service.
- Delete() – Like the previously method, this method follows the recommendations of the API, as it is used to delete an entry on the naming service.
- Update() – Like the two previously methods, Update is responsible for updating an existing record in the naming service database.
- Call() – Although this method is presented on the content provider class, it is not mandatory to implement by a content provider, but in our architecture it is absolutely fundamental. This method is responsible to handle provider specific methods that can be called from applications. All the handled methods are statically defined in a HashMap that maps commands into values to be handled individually (Figure 4-11).

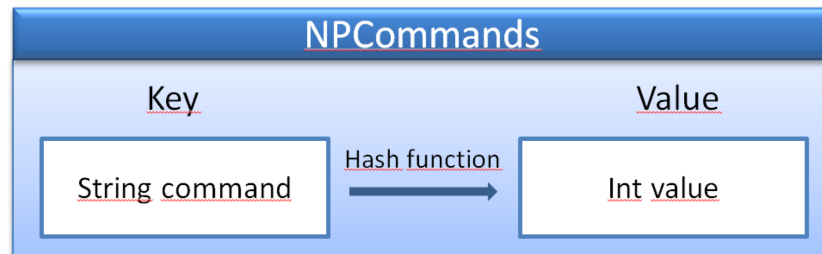


Figure 4-11 - Commands Hashtable Diagram

In our REINVENT implementation, we defined only one command to be handled in this method, but in order to present new services or methods to the applications, they could be handled here. The command available is the GetMessages, which is used by applications to get all the messages of a certain type available in the provider. The type of the message is passed as an argument of the Call method, and the return value is returned as a Bundle containing an ArrayList of MessageContainer, a class to be explained later, messages of the specified type. Applications can require new messages at any time, but to improve the performance of the system, we implemented a notification system that can be listened by the application, and be notified every time a new message is received in

the provider, avoiding the usage of polling for new messages. Since this architecture was made to be very generic, it does not support the notifications of specific types of messages so it will notify every listener about a new message independently of the type desired of each application.

- BuildMessage() – This is an auxiliary private method to compose a String message in a format to be correctly understood by other applications implementing and using our module. The generic format used by our module for message exchanging is [TYPE] [SENDER] [DESTINATION] [BODY]. The objective of creating this message format was to continue with our goal to create a very generic architecture, independently of the application, using this module they can always define their own types of messages to be consumed, and even if they want they can encapsulate new headers in the BODY field of the message.
- ConnectToRabbitMQ() – This is a method for creating a connection to the RabbitMQ server. It also creates a queue for receiving messages because even if the listening thread is not active, the messages should not be lost at any point.

4.7.3 Message Container

This class is an auxiliary class used by both the module and the applications in order to encapsulate the messages passed between the applications and the module.

Figure 4-12 represents the Message Container class that is a descriptor for a message packet. The class has four attributes that represent the four basic and mandatory fields of a message; a type, which is an enumeration of three character string representing the type of the message. We have created two basic types of messages: the first one is *TXT* that is to be used for plain text messages and the second one *GPS* that represents messages containing GPS coordinates. These were the types of messages created in order to cover a basic set of applications. The next two attributes are strings representing the ID's of the message sender and destination, respectively. The final attribute is the body of the message bearing the content of the message.

The message container methods are composed by two constructors, a default one creating an empty message, and one that used the four attributes as arguments, a set of Getters and Setters for all attributes. There are two more methods implemented as the class implements the Parcelable interface. We need to implement this interface because the messages exchanged between the applications and the module will be made through a

Bundle, which is a special container for exchanging information between entities in Android. The parcelable interface methods help the serialization of data content objects into a parcel to be sent in the bundle. The *writeToParcel()* method writes the four attributes of the message into a parcel, while the *createFromParcel()* method does exactly the opposite, fetches the four attributes into a MessageContainer object from a given parcel.

This class was created in order to facilitate the creation and handling of messages on the application side, since the messages received by the provider come in plain text, and if they were directly sent to the applications, they would need some kind of parsing mechanism that could result in several errors or misinterpretation of data, so we decided to do that parsing and processing in the NetworkProvider and providing a well-structured message container to the applications.

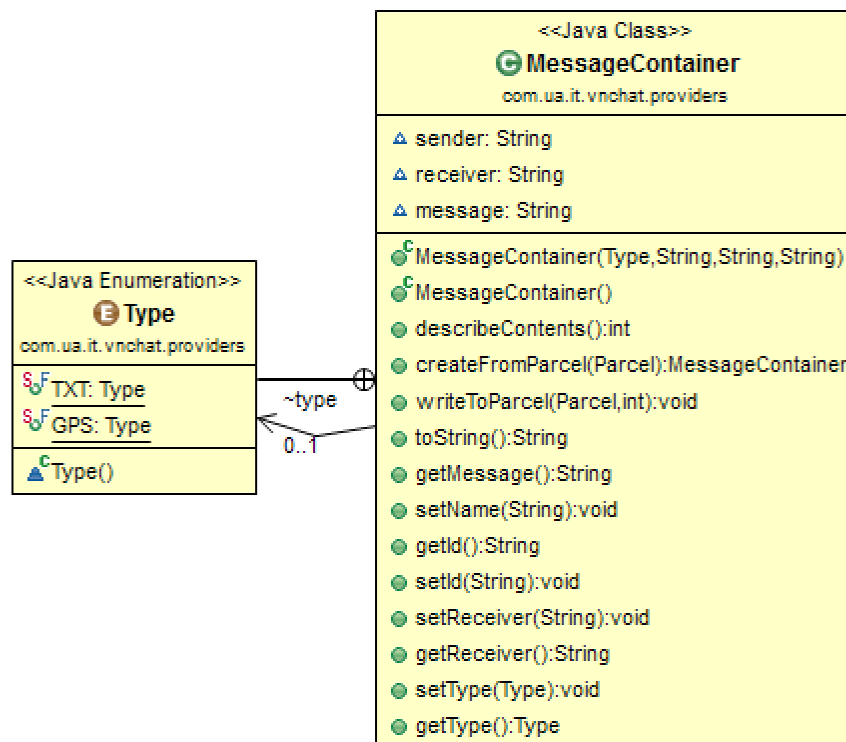


Figure 4-12 - Message Container class diagram

4.7.4 User Descriptor

The final class that composes REINVENT is the UserDescriptor (Figure 4-13), which is a class that defines some required constant values for our NetworkProvider.

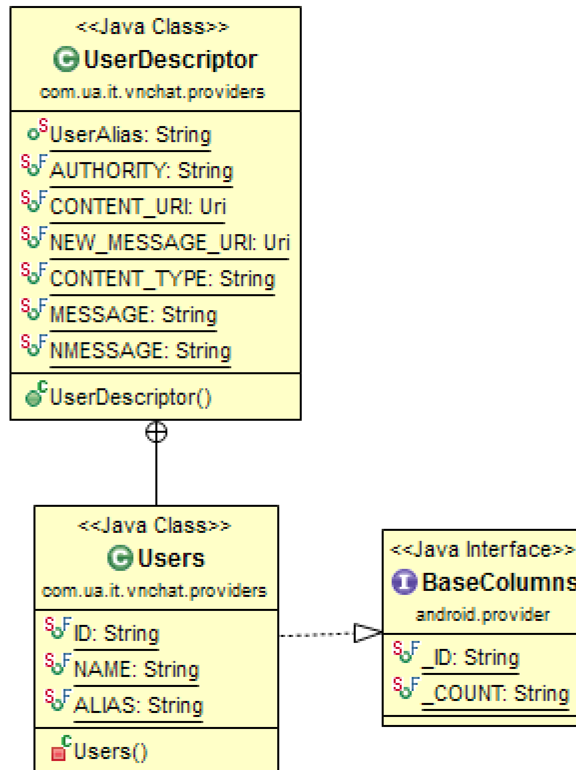


Figure 4-13 - UserDescriptor class diagram

This class contains the definition of an entry in the naming service that is composed by and ID, a Name and an Alias. It implements the BaseColumns interface in order to be used by the SQL database that will be used to store the naming service entries. The other attributes of this class are static references for the Authority of the provider as well as the URI address available on the provider.

4.8 Conclusions

In this chapter we presented the implementation of REINVENT in Android Operation system. We introduced Android content provider as the solution for implementing our REST interface, and the RabbitMQ framework as the solution for our messaging service. In the following section, we describe the REINVENT services provided to the applications. These services include all methods necessary for the applications to communicate through a VANET.

The chapter ends with the implementation details of REINVENT. We started by describing two applications: VNChat, a message exchanging application and iThere, a

location based application to find nearby users. Finally, we described the main classes and structures that compose REINVENT.

At this point we have Android based REINVENT module totally implemented along two applications ready to be used, but we still do not have any scenarios to test and experiment. In the next chapter, we will describe the creation, configuration and setup of both real and simulated scenarios

.

5 REINVENT: Creating the testing scenarios

In this chapter we describe the process of creating the testing scenarios for REINVENT. We defined two different types of scenarios that will be presented: a simulated, using a VANET simulation framework, and a real one that uses On Board Units developed in the Instituto de Telecomunicações de Aveiro.

Specific details on each of the scenarios will be presented and discussed namely addressing configuration and execution issues.

5.1 Simulated Scenario

The simulation scenario, prior to execution implied three steps:

- Select and deploy the simulation framework that integrates both network and traffic simulation
- Define a scenario to simulate and create it in the framework.
- Integrate the REINVENT and mobile application within the simulated environment, while maintaining temporal consistency i.e. map deterministically the simulation time with the mobile application time.

5.1.1 Simulation Architecture

The main challenge of creating VANET simulation architecture was the integration of mobile devices in the simulation, since the framework used, as with all existing, was not created or supported integrated connections to external devices.

We selected the VSimRTI framework as our starting point, as it presented the best features for simulating applications and connecting with external applications. The VSimRTI is the only framework that has an integrated application management unit that allows the deployment of applications in each simulated entity, which is a great advantage to all other frameworks, as the main goal of our simulation scenario is to run and test applications in the vehicular network context.

In our scenario VSimRTI is coupled to various mobile devices using RabbitMQ framework that, through periodic messages to simulator, keeps the mobile devices with most actual information as possible giving the simulation an idea of synchronization with

the devices. Every vehicle-to-device connection is made by creating two queues to allow bidirectional communication without interference.

As the applications are completely independent of the simulation, they are ready to run in a real scenario, and at the same time capable of being part of the simulation. The simulation will not be dependent on any indication of action from the mobile devices, thereby allowing the interaction between the device and the simulation to be done asynchronously. Like a real world scenario, users can perform actions at random times, but the simulation keeps running synchronously and deterministic as supposed.

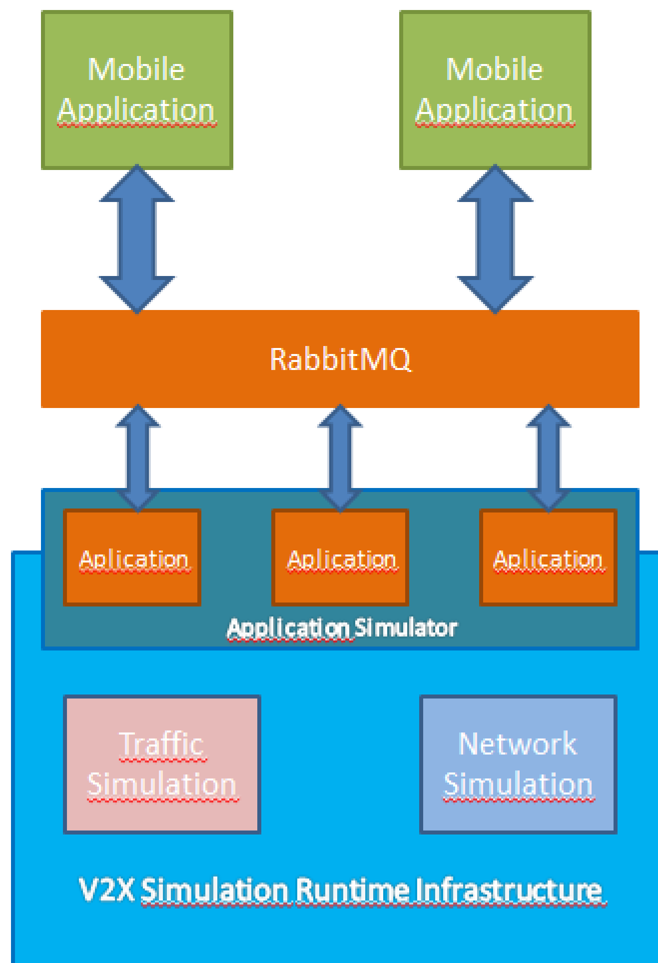


Figure 5-1 – Simulation Environment Architecture

The setup of the simulation is composed by the following components (Figure 5-1):

- Traffic Simulator – SUMO
- Network Simulator – JiST/SWANS
- Application Simulator – VSimRTI Native

- Communication between simulation and devices – RabbitMQ
- Mobile Devices – Android Virtual Devices

SUMO is the traffic simulator used to simulate the behavior of each vehicle in the road. As described previously, SUMO offers a runtime interface to control and influence vehicle behavior at runtime of simulation, and it is also used to get information [30]. Hence, the corresponding ambassador converts messages and commands into the TraCI format to be read by SUMO [61]. The traffic generated by SUMO is used as an input for our network simulator that is JiST/SWANS. VSimRTI team developed an extension to JiST/SWANS to allow the synchronization of the internal scheduler with the received messages or commands using a socket interface. The applications running in the vehicles are simulated by the application simulator, which is an in-house multi-layered application container with its own application and facility layer and interfaces the network layer. It serves as a framework for users to create custom V2X applications [62]. As it is native to VSimRTi, it does not need to be installed.

Finally, we have the mobile devices that are simulated as Android Virtual Devices created using the Android Developing Tools [63].

5.1.2 Simulation scenario

In our simulation we used a segment of the Portuguese national road 109, which will be referred from now on as N109, with one lane on each side, where three vehicles capable of V2X Communication are travelling at a constant speed and distance. Two of these vehicles are equipped with an Android mobile device.

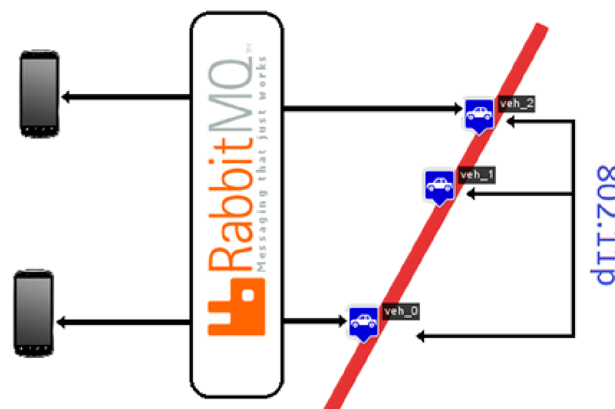


Figure 5-2 – Simulation Scenario Overview

The three vehicles will travel along the road, and during that route, the Android devices will be able communicate through the network provided by the simulation. The simulation follows basic workflow:

- A mobile device sends a message that will be stored in the RabbitMQ
- The correspondent vehicle application will get the message stored in rabbit, transform it in a vehicular network message format and sends it in broadcast.
- The destination car application will receive the message and forward it to the RabbitMQ
- The receiver mobile device will read it and show it on the interface.

This sequence of events can occur as long as the simulation runs, and it does not need to be performed at any special time or targets. Since the messages are sent in broadcast, all vehicles will receive them, but they will discard any messages that are not addressed to them.

5.1.3 Connecting Applications to the Simulation

The key factor for connection the simulation to external applications was the RabbitMQ. We used an instance of the RabbitMQ server installed locally that will work as a messaging interface from the simulation scenario to the mobile applications that will connect to the simulation. The simulation will be sending periodic messages (plus the applications specific ones) with the state of the simulation, so the mobile applications have the feel of being part of the simulation, even if they are totally independent and do not need to interact with the event scheduler of the simulation.

The setup we used to connect our simulation to the mobile devices was to create two message queues in the RabbitMQ server for each device-vehicle pair, in order to obtain bidirectional communication. As described in REINVENT architecture, there is a thread running in parallel to the main application that will be listening to the receiver queue for any new messages published by the mobile device. On the other hand, any messages that the vehicle wants to forward to the device will be published on the other queue. We created our own Java clients based on the libraries available. The first one publishes messages, which will create the desired queue if it does not exist, and only

connects to it when it needs to publish a message. The second client is the consumer which will be permanently connected to the listening queue and waiting for new messages. The clients are fairly the same in the Android and car applications, and only differ at the time of handling the message. In the Android client, the message handling is delegated to a handler that will process it and update the user interface if it is necessary; on the other hand, the car application simply saves the messages in the internal inbox so they can be handled in the next timercall cycle.

5.1.4 Android Test Application

In order to test the described setup, we created a simple Android application. The application consists on a simple text message exchange application between the vehicles. The application has only one activity that shows the status of the simulation, the current simulation time and all the received messages. There is also a text field to write the messages to be sent followed by the sending button. The connect button was created to force a connection to the RabbitMQ in case it was lost or it simply could not connect at the time.

We created simulation specific message types that will be sent by the vehicle application in order to inform the Android applications about the current state of the simulation. The following messages should be handled by the mobile applications in order to obtain the simulation information:

- SIMSTART – this message signals the application that a simulation has started.
- SIMOVER - This messages marks the end of a simulation
- TIME – These messages are sent periodically and represent an update from the simulation with the current simulation time. The application can use them to update the simulation timer.

We can see in Figure 5-3 the result of two running applications in the same simulated scenario, where two messages were sent between them, and we can also see that

the simulation time is synchronized between them, so it means that the car applications are actually keeping the Android applications up-to-date.

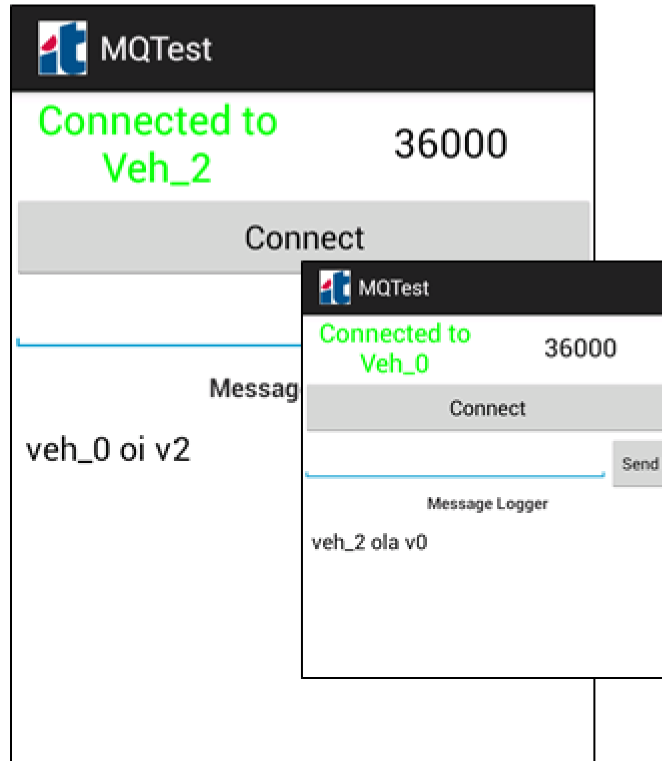


Figure 5-3 – Testing Application

Because this application was merely created for simulation purpose, both the server configuration, the car to be connected and the queues are made by launch parameters, so we can control the flow of simulation by assigning a specific application to a specific vehicle. It is also important to understand that, at this point, we could have used any kind of application to test, as this solution was created to allow any Android application to connect to simulation.

5.1.5 Simulation scenario setup and configuration

Besides specific details concerning our scenarios, the setup and configuration of the overall simulation implies several steps that are described in detail:

- VSimRTI Configuration – setting up the simulation environment
- Creating the map and converting to VSimRTI format – prepare the scenario map to the simulation environment

- Setting up the traffic simulation- configure SUMO
- Setting up the network simulation – configure JiST/SWANS,
- Mapping, Navigation and Application simulators setup
- Defining the vehicle Application

VSimRTI Configuration

VSimRTI is organized as a hierarchy of folders where the configuration files of each simulator will be placed in a folder named after the simulator, for example, the SUMO configurations would be placed under a folder called SUMO, which will contain XML files related to the simulation roads, vehicles and routes.

The framework has also a main configuration file where it should be defined which simulators will be used. It is a simple XML file with the names of the several simulators set to true or false as needed. The minimum setup for a simulation should include all the embedded simulators, such as the application, mapping and navigation, traffic simulator and network simulator. The mapping and navigation simulators were created to help the creation and control of the several entities of the simulation, and to help exchanging messages between the simulators respectively.

It is also in this file where the main attributes of the simulation are defined. These attributes include the name of the simulation, start and end time, the map offset values that come from SUMO, and finally the real geographical coordinates of the map if it suits the simulation.

Creating the map and converting to VSimRTI format

To create the map used in this simulation, we used a tool to convert maps exported from the *OpenStreetMap*. *OpenStreetMap* is a project that creates and distributes geographic data from the whole world, by simply selecting the desired area and exporting it in a couple of different formats such as XML *OpenStreetMap*, which we used in our work or simply as an image format. After we select our N109, we export the map in the OSM format and we will process it with a tool called *osmconvert* to transform the map to the VSimRTI and SUMO formats.

Osmconvert is a tool for importing, converting and exporting OpenStreetMap data to different simulator specific formats, e.g. it converts OpenStreetMap files to a database, which can be read by VSimRTI. This database is the basis for all map-related tasks, which can be performed by VSimRTI (e.g. navigation, route calculation). Based on a VSimRTI database, *osmconvert* can export the data to SUMO input formats, which then can be used in the simulation. To enable dynamic rerouting of vehicles, *osmconvert* generates exports and imports route data from and to SUMO. This way, one can choose whether to use generated routes (all possible routes between a point A and B), use existing routes and import them via *osmconvert*, or build own routes via the route generation tools supplied with the standard SUMO installation.

VSimRTI uses a database for internal storage of map and route data. Therefore, it is necessary to convert *OpenStreetMap* data into a VSimRTI specific database format. Using this database, the map data can be exported to different traffic simulators, such as SUMO. This makes sure that each part of the simulation relies on the same map using the *osmconvert* tool [64].

Setting up the traffic simulation

After transforming the map to the VSimRTI and SUMO format we need to define which routes the vehicles in the simulation will take. These routes can also be defined using the osmconvert tool using the starting and ending point of the desired route, and the tool will return a route with the shortest path between those two points in case it is possible. The routes can also be defined by hand in the specific SUMO routing file, which is also a XML file. In this work, we only defined one route which will travel along all the extension of the road available.

Having a route defined, in that same XML file, we need to define how many and the type of the vehicles which will be participating in the simulation. So we defined three vehicles of the same type, length and maximum speed with a departure difference of 5 simulation seconds. At this point, we have our traffic simulation configured up and it is possible to test it with the SUMO visualization tool, which will result in three vehicles traveling in the road at the same speed and distance from each other until the end of the road.

This was the setup chosen for the traffic simulation, as the main focus of this work was the integration of mobile applications in vehicular network simulation, so we created a simple and very controlled traffic scenario to ensure that the cars would have connectivity at all times. At the same time, the scenario is complex enough to test network messages exchange between more than two entities.

Network simulation setup

The configuration of the JiST/SWANS, as all other simulation components, will be under a folder called swans, with XML file for configurations. This file contains the used model parameters especially for the radio channel, physical layer and the routing protocol on the network layer. The file is divided in two different sections: the first one contains the common parameters, and the second one contains two sections with specific parameters to support different configurations for different entities (e.g. Vehicles and RSU).

In the common section the following parameters should be configured:

- **Dimensions of the simulated area** – It should not be smaller than the ones defined in the SUMO.
- **Configuration of the Random Number Generator settings** (Three types supported, Linear Congruential Generator, Mersenne Twister and BlumBlumShub) – We used linear Congruential Generator
- **Propagation model for path loss and fast fading** – We used Free Space model which does not need any further parameter configuration
- **Common parameters for physical layer** – The default SWANS assume configuration of the IEEE 802.11b, even if the IEEE 802.11p is the reference standard for vehicular communication, so we used a configuration provided by the VSimRTI that has the IEEE 802.11p parameters.
- **Routing models –** If none, which is our case, all V2X message will be sent using single hop broadcasting.

Since we only have vehicles in this simulation, there is only one more section to configure with the physical layer parameters specific of the vehicles, which includes antenna heights, TX power and RX sensitivity. We used once again the configuration provided by the VSimRTI, since it was already used and tested in a scenario for vehicular network simulation [62].

Mapping, Navigation and Application simulators setup

As was referred before, these three simulators are part of the mandatory simulators for running a simulation in VSimRTI, as they take an important role connecting and synchronizing all the simulation together.

The application simulator has very few configurations to do, as most of them are made by the mapping simulator. The application folder structure is divided in three different folders for each of the three different entities supported by VSimRTI, vehicles, RSU and Traffic light. It is under each of these folders that the application JAR file related to each entity will be placed. There is also a XML configuration file that is used to configure the parameters of the messages sent by applications. It is possible to configure the automatic sending of messages with a given time interval or position change value. Since we do not want our vehicles to send any messages by themselves, as we want to control all the messages exchanged between the vehicles, there was no configuration needed in this file.

The mapping simulator is responsible for creating instances of simulated entities and starts the applications on them. It is responsible for handling every type of entity, vehicle, RSU or traffic light. VSimRTI offers the possibility to assign different applications to different vehicles or groups of vehicles, and it is in the mapping simulator that those configurations and attributions are made. It is responsibility of the mapping simulator to define and configure the following:

- Simulated traffic entity in detail such has vehicles, RSU or traffic lights.
- The ratio between certain traffic entities with different attributes upon creating.
- The attributes of a specific traffic entity.

The configuration is made under the mapping folder in a XML configuration file which is divided in two parts, attribute definition and mapping configuration. In the first part of the configuration file all basic entities and compositions can be defined, each basic entity configuration is made by an assigned configuration file. Applications and vehicles will have their own configuration files, applications file will have the mapping of JAR files into elements to be assigned later to a vehicle, while the vehicle will have the properties of

vehicles if we want to configure different types of vehicles. The vehicles properties will just be mapped into the SUMO configuration files created previously. The second part of the main configuration file is used to define how many vehicles of a specific type should drive in the simulation. It is possible to choose between deterministic mapping, where the sequence of the mapped vehicles will always be the same in every simulation, and stochastic mapping resulting on a random order of the mapped vehicles. The different mapping sections found in the file defines what percentage of each entity will be created for the simulation.

In our simulation we used the deterministic mode because we want to have total control of the simulation status, so we do not have to care about the random situations that could be created. In this particular case, it would not be a big problem to use stochastic mode because our three vehicles have exactly the same attributes, but in terms of the application logic, they would be different as the mobile devices will be connected to specific vehicles.

Vehicle Application in VSimRTI

The Vehicle application role in VSimRTI simulation is to be used as relay between the simulated transport platform for messages sent and received by mobile devices. Its design is simple, since it will only make processing of incoming messages and forward either from the network to the mobile device or vice-versa. This implies converting contents and composing a message in the format supported by the network carrier or understood by the mobile device application.

To keep the simulation running smoothly without interfering with the internal scheduler of the VSimRTI, we needed to find a solution for the messages sent from the mobile devices to be inserted in the simulation structure in a synchronous way. The solution found was to create a thread running in parallel to the vehicle application that is listening for new messages from the mobile devices. Once it receives a new message, it will be stored in the applications internal message queue, like an inbox. Once the *timercall* method is called, which was configured to be called once every second, it will have two jobs, the first one is to report to the mobile device the actual time of the simulation so it keeps updated on what timing the simulation is running, giving a notion of synchronization to the simulation that actually does not exist. The second job is to get all messages in the

internal message queue and transform them into a supported format to be sent in the vehicular network. This was a very important step to integrate the mobile devices in the simulation without interfering with the internal scheduler of the simulation. In the worst case, the messages will be delivered to the vehicle with an additional second.

To forward a message from the network to the mobile device, the OBU application only needs to invoke a method every time it gets a message (a callback). After receiving a message, the OBU application will check if the mobile device is the destination of the message and if it succeeds, the message is forwarded to the device, otherwise the message is discarded.

Defining Vehicle Application in VSimRTI

The creation of applications for VSimRTI is done in Java language using an interface provided by VSimRTI that must be implemented by all applications called Application. This interface is responsible for providing the necessary methods to the simulator if a certain event happens. All methods of this interface should be implemented even if they are left empty.

Applications may be specific to the three types of entities supported by VSimRTI, vehicles, traffic lights or RSU. For each of them, there is a specific type of reference which must be instantiated so as to obtain the required methods for the control of entities.

The Application interface implements the following methods:

• **Void initialize** (ComponentFactory factory, Logger log) - This is the method that should be booting the entity using the input parameter of the factory method to remove the references to control the entity that runs the application:

- Timer - Reference to the simulation time, is updated automatically by the simulator and serves to maintain the state of the simulation time
- CommunicationModule - reference to the communication module, used for sending messages to the network
- VehicleStateManagement / RSUStateManagement - This module is responsible for storing the state variables of the respective entities and should only be used for the same query. Some examples of available attributes are vehicle identifier or current vehicle speed.

- VehicleControl / TrafficLightControl - This module unlike the previous point serves to interact with the entities and change their status. Some examples of possible state change are changing lanes for a vehicle or even slowing. For Traffic Light it is possible to force a specific state of the state machine (Forcing a red light).

• **Void receiveMessage** (TypedV2XMessage msg) - This method is called whenever the entity in question receives a message from the network and it is responsible for handling the message.

• **Void timerCall** (long time) - This method is called by the simulator if registered by the application to be called with a periodicity that can also be defined. If there is a task that should be performed with a given periodicity, it is this method that should be assigned to take care of it. The input parameter of the method represents the simulation time at which the method was called.

• void dispose () - This method is called before the application is terminated and removed by the simulator. It may be important to record statistical values of the simulation.

The developing of applications can be made any IDE, and must be compiled with version JavaSE-1.7 and the following libraries must be added to the application Buildpath:

• vsimrti / bin / ambassadors / lib / application-messages-xxx.jar

• vsimrti / bin / Fed / application / application-federate-xxx.jar

• vsimrti /bin / lib /slf4j-api.x.x.x.jar

• vsimrti / bin / lib / vsimrti-collections.x.x.x.jar

5.2 Real Scenario

The objective of the real scenario is to test our application within a realistic scenario involving existing OBU deployed on vehicles. The OBU supports the 802.11p for sending messages in the vehicular network specific protocols, and the 802.11b which is use to connect to Android devices.

In the scenario, OBUs will provide proper environment for receiving messages from the mobile applications, and then compose proper vehicular networks messages that will be sent to other OBUs through the VANET. It will also be responsible for receiving

incoming network messages and strip them from the network specific attributes, and forward the message payload to the applications.

5.2.1 Equipment

The OBUs used were developed in the Instituto de Telecomunicações of Universidade de Aveiro and created for vehicle communication purposes. This OBU is already being used in a large scale testbed, being already used by over 400 taxis in Porto city, and it is already a very reliably equipment for V2V communication as it has been subject of several improvements since its creation.

The following list presents the description of some of the main features of this OBU:

- Wi-Fi Module using the IEEE 802.11a/b/g.
- Wi-Fi Module using the IEEE 802.11p with frequency ranges between 5.86 and 5.92 GHz[65].
- Antennas for supporting the previously described modules.
- GPS GlobalTop (MediaTek MT3329).
- Minimalist Linux OS.

The Figure 5-4 shows an overview of the OBU with its several available connectors. On one side, it has access to two connectors to connect the two antennas (for the g and p standards), an Ethernet port, a connector for power supply (battery or lighter), and a serial port (RS-232), which is used to read the values coming from the GPS. On the other side, it has a video output and two USB (Universal Serial Bus).

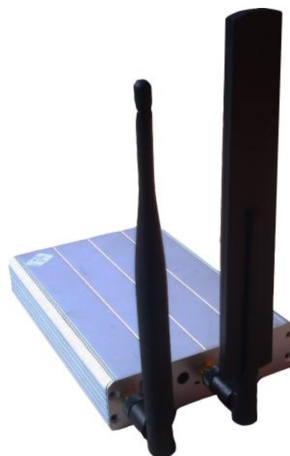


Figure 5-4 - On Board Unit

Several Android devices were used in this scenario comprising four different devices: a TMN Smart A18 (ZTE Grand X), a Galaxy Note 10.1, a HTC One X and a Samsung Galaxy Mini. An overview of the device specifications is described in the following table:

Device	*CPU*	*RAM*	*OS Version*
TMN A18[66]	Dual-core 1 GHz	1GB	Android OS, v4.0 (Ice Cream Sandwich)
HTC One X[67]	Quad-core 1.5 GHz	1GB	Android OS, v4.0 (Ice Cream Sandwich)
Galaxy Mini[68]	600 MHz ARMv6	384 MB	Android OS, v2.2 (Froyo)
Galaxy Note[69]	Quad-core 1.4 GHz	2GB	Android OS, v4.1 (Jelly Bean)

Table 4 - Android Devices Features Overview

We decided to use devices with different specifications so we can understand if the REINVENT module has any impact on the devices performance. These three devices cover all the spectrum of Android devices, with the Galaxy Mini being probably the lowest end of the Android devices, the TMN A18 being the term between, as it is not state of the art of Android technology but is far from being an outdated device, and finally the HTC One X that is among the best current Android devices. We also used a tablet in order to understand the behavior of REINVENT in a slightly different environment, although we did not develop any particular application oriented for tablets.

To carry out the experiments on the road, we always used at least two vehicles: vehicles used for the scenarios in this document have been a Toyota Corola (5 Doors) and a Ford Fiesta (5 Doors). We took into account the use of vehicles of similar size, decreasing the sources of error due to different vehicle structures (Figure 5-5).

Figure 5-5 - Toyota Corola and Ford Fiesta

5.2.2 Scenario architecture

The deployed architecture (Figure 5-6) is composed by the following components:

- Android devices – TMN A18, HTC One X, Samsung Galaxy Mini
- On Board Unit with 802.11p and 802.11a/b interface running the following services:
    - RabbitMQ Server
    - Message Handling Client

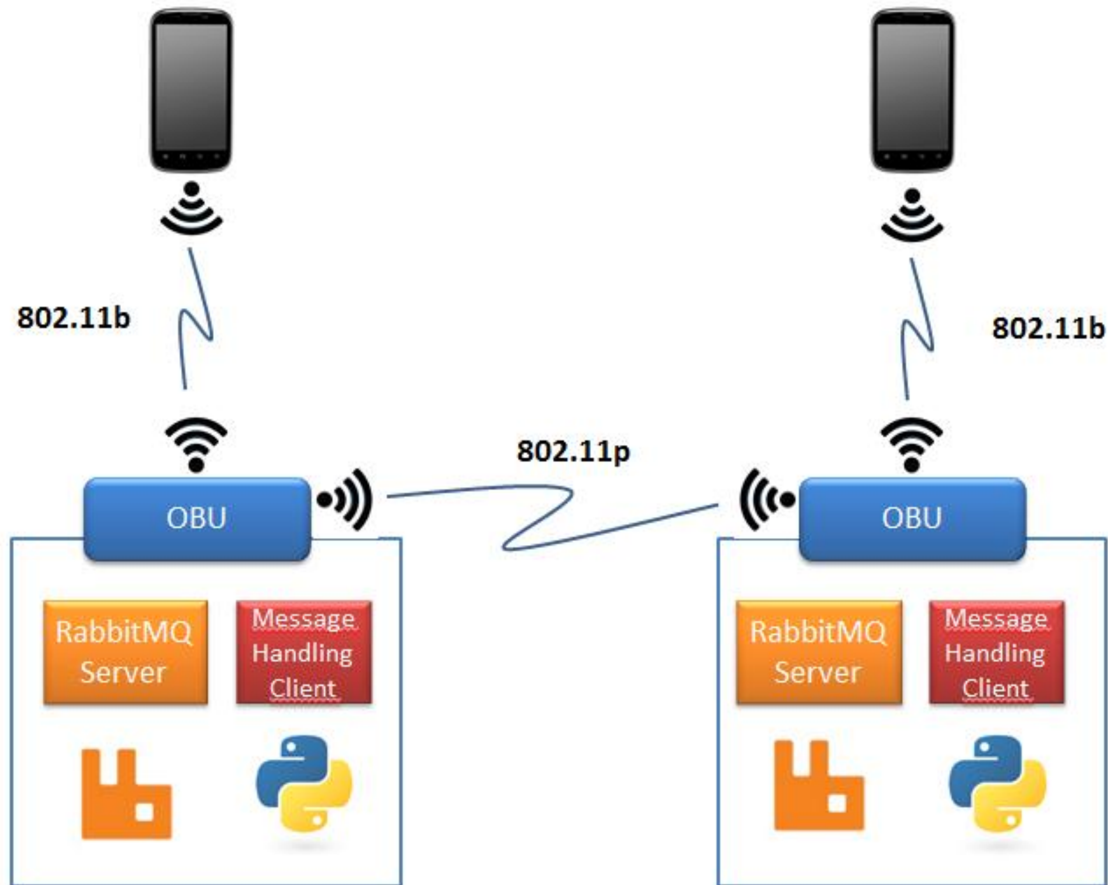


Figure 5-6 - Real Scenario Architecture

In this architecture, the mobile devices connect to the OBU by their Wi-Fi interfaces through a provided network. The OBU runs the RabbitMQ Server and a client that will be handling the messages. These messages can come from both the VANET and the Android applications via the RabbitMQ server.

The android applications can connect to the Rabbit server by simply using the OBU Wi-Fi interface address, as the Rabbit will be coupled to that interface. The client that is running inside the OBU is responsible for listening for the messages received, in order to forward them to the RabbitMQ, so they can be fetched by the applications. It is also responsible for getting the messages sent by the applications to the Rabbit server, and then sent them to the WSMP service, which will be explained next.

The communication between the OBUs is made using the Wave Short Message Protocol (WSMP), a service defined in the IEEE 1609.3 that allows requests from higher layers, such as applications, for sending messages over the air by MAC Address in both unicast and broadcast. For the REINVENT messages, we are using the control channel,

and the messages are sent with the Provider Service Identifier (PSID) 80-01. The client starts a daemon that listens for the incoming Wave Short Messages (WSM) with a given PSID using the following shell command:

uwsmp receiveWSM psid 80-01

Opening a pipe with this command allows our client to read from the Standard Output (stdout) of the daemon where it will write the receive WSM's. After receiving the messages, the client will also parse the body of the message, and then build a message in the proper format to be interpreted by REINVENT.

On the other hand, the client also needs to send WSMs whenever it retrieves a message from RabbitMQ server. This is made using the following command:

uwsmp sendWSM psid 80-01 amount 1 msg [BODY]

5.3 Conclusions

In this chapter we described the process of creating and setting up the scenarios for testing REINVENT.

The first section started by describing the simulation architecture followed by the scenario. The next section described the integration of mobile applications with the simulation. Finally, the chapter ends with scenario configuration and setup information. It is important to note that the main challenge when creating this scenario was the integration of mobile devices with the simulation.

In the second section of this chapter we described the real scenario. We started by describing the equipment used, the Android devices and the OBUs, followed by the scenario architecture where we described not only how the devices connect to the OBU, but also how the OBU handles the messages.

At this point, we have REINVENT totally implemented as well as both simulated and real scenarios.

6 Experimental Tests & Results

In this chapter, we will describe the testing procedure of REINVENT in both simulated and real scenarios. In the evaluation process, we will use two mobile applications (VNChat and iThere) to assess the impact of REINVENT and VANET conditions in their performance, namely in the message exchange. In the simulated scenario, the tests were deployed in a scenario where all the entities are in a single machine, and in the real scenario, the entities are deployed in two cars with VANET configuration (each car with its own OBU), and different mobile devices with different hardware specifications.

In section 6.1, we will analyze the impact of REINVENT in the devices: we will be performing tests in different devices so we can understand if the applications developed using our module have any hardware limitations.

In section 6.2, we will perform several tests in the simulated environment. We will test the performance of the simulated OBU application. We will also measure the delay of sending messages between devices and how parameters like distance affect these delays.

In the section 6.3, we will perform several tests in the real world environment. We will be testing the performance of REINVENT by measuring the delay of the system in several conditions.

6.1 REINVENT Performance on Different Devices

We used four different devices to run these tests, a Samsung Galaxy Mini, a TMN A18 (ZTE Grand X), a HTC One X and a Galaxy Note. The choice of these four devices was made based on the coverage of most of the actual Android device range, being the Galaxy Mini a very low end device with very low specifications (600 MHz CPU and 384 MB RAM), the TMN A18 is the medium range device, as it has good specifications whilst not having the best ones, the HTC One X can be considered as the state of the art of the android devices, as it is one of the best in the market, and finally, the Galaxy Note represents the tablet class of mobile devices.

The first test was focused on delays in message sending and receiving from the mobile. We measured the delay of sending a message from the time the user hits the sending button, to the time the REINVENT sends the message to the RabbitMQ server.

The reception of a message is measured from the time the REINVENT module consumes a message from the RabbitMQ server, to the time it is shown in the user interface. This test is performed to analyze the delay that REINVENT induces in the applications. The scenario used to execute these tests is the real experimental scenario, where each device is connected to an OBU, and we send 40 messages from each device in order to measure the send time. These messages are used to measure the receiving time of the other device.

The Figure 6-1 and Figure 6-2 show the results of both sending and receiving messages in all the four devices. In Figure 6-1 we observe that the results between the mobile phones are similar, being the Galaxy Mini the one presenting the highest delay, while the Galaxy Note has the lowest delay by far from all the four devices. The results of Figure 6-2 follow the same pattern of the Figure 6-1. However, the delay values of the Figure 6-1 and Figure 6-2 are very different. In the Figure 6-1, the delay values are around 160 msec for the three first phones, and 80 msec for the tablet; in the Figure 6-1, the delays of all the devices stay around the 1,5 msec. The large difference of delay values can be explained by the way REINVENT works as for receiving messages: REIVENT creates a connection to the RabbitMQ server and listens for new messages without closing the connection, while every time it sends a message, it creates a new connection to the RabbitMQ server, so the difference on the delay value may be justified with the connection time to the RabbitMQ server for each message.

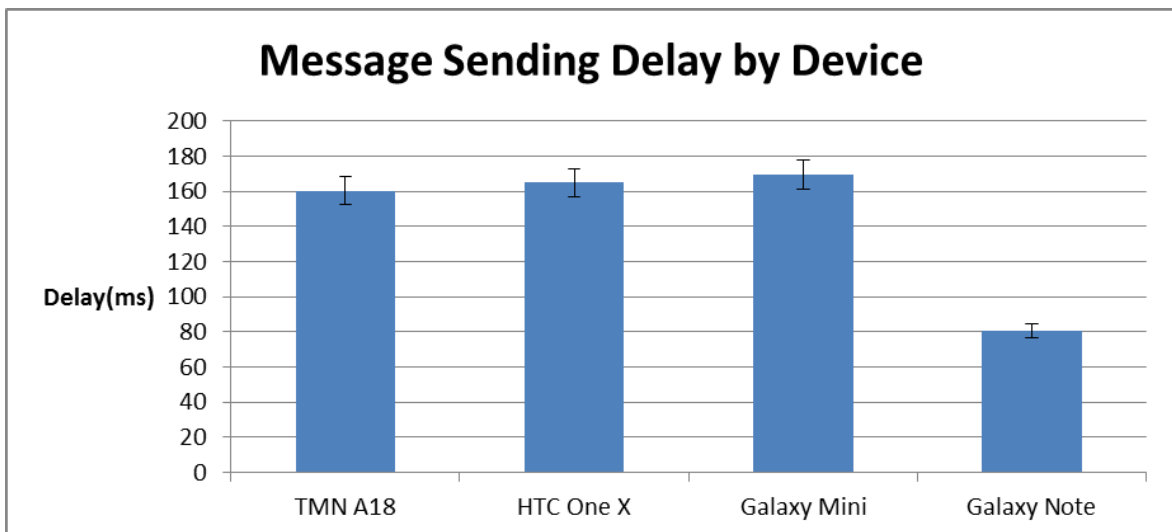


Figure 6-1 - Message Sending Delay by Device

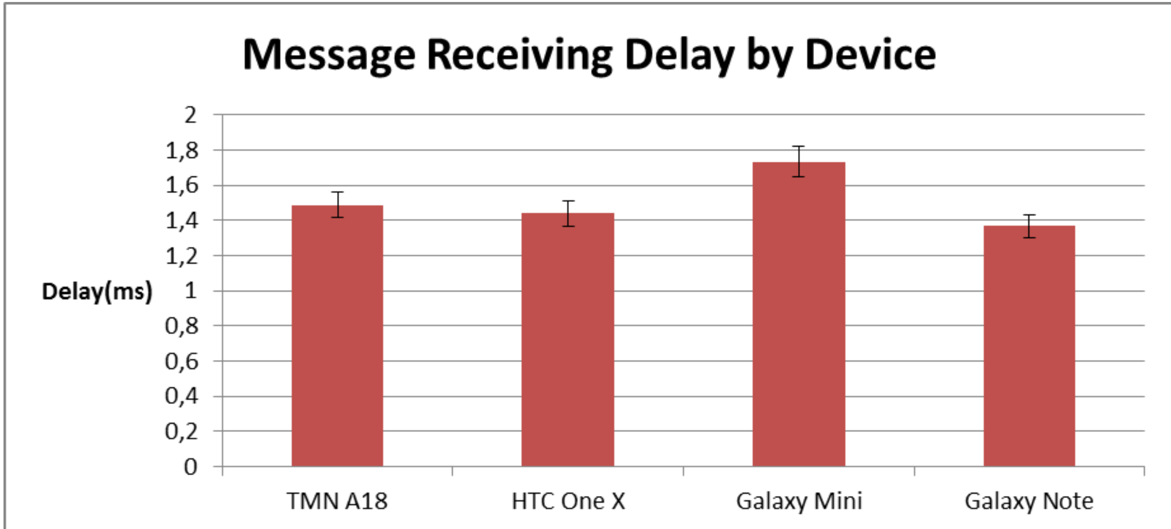


Figure 6-2 - Message Receiving Delay by Device

The results suggest that REINVENT module does not induce any significant delay to the application execution, as the result values can be considered acceptable (almost negligible in the case of the receiving). It is also possible to infer that REINVENT does not depend on any specific hardware requirements, as it behaved similarly in all the devices considered with very different specifications. The slightly variations in the values between the devices are according to the differences in the specifications, where the Galaxy Note presents the best results and the Galaxy Mini presents the highest delays.

6.2 Simulated Environment

The objectives of the tests performed in the simulated scenario were first to assess the feasibility of using a simulated scenario to test mobile applications in vehicular network environtment, and second, if that is possible, to test the performance of REINVENT module in different simulated scenarios. The scenario used was composed by 3 cars, where two of them are connected to real Android devices running the VNChat application. Further details can be found in section 5.1 of Chapter 5 .

6.2.1 OBU Application Performance

In the first test, we wanted to understand the delay of processing a message in the simulated OBU application. We measured the delay inside the application for sending a message received from the application to the network, and the delay for receiving a message from the network to the application. In this scenario, the vehicles were driving at

a constant speed of 50 km/h with 300 meters distance between them. We also used a 40 message sample in order to obtain the results presented in the Figure 6-5

The Figure 6-4 represents the overall scenario where the red arrows represent the measuring points while Figure 6-3 represents the vehicle placement of the simulation. The Green cars are representing the ones running the mobile applications, while the Red car does not have V2V communication capabilities.

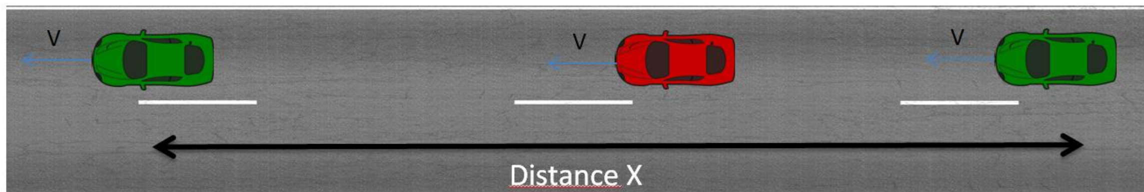


Figure 6-3 Simulated vehicle placement

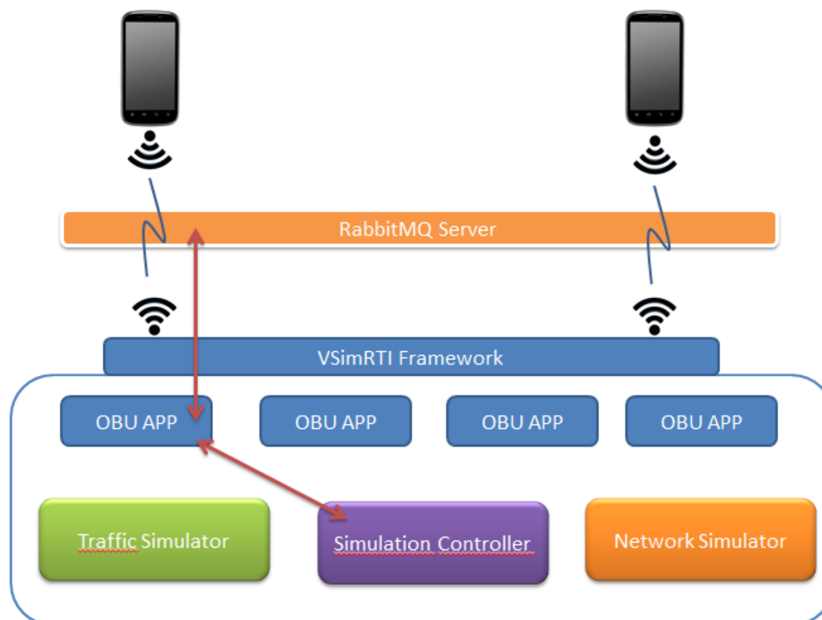


Figure 6-4 Measuring delay inside the OBU application

In Figure 6-5 we can observe the results of the test, where the blue line represents the delay of messages sent from the application to the network, while the red line represents the delay of the messages sent from the network to the application.

The delay from the received messages is fairly constant and has an average of 10 msec. We can conclude that the OBU application itself does not add any major delay to the messages received from the network.

The delay from the sent messages has an average of 140 msec delay, but it contains a high oscillation on its values that can be easily explained. Since the mobile applications are not part of the simulation itself, the messages cannot be handled by the OBU application on the time they are sent from the mobile application, because the OBU application actions are controlled by the simulation scheduler. Like we described in the section 5.1.3, the OBU application runs a message checker in a defined time interval, so if the user sends a message right after the message checking, the message will have an additional delay equal to the time interval of the message checker. On the opposite case, if the user sends a message right before the message checking, the message will be sent to the network with a delay near to 0 msec. In this particular simulation setup, we used a 500 msec time interval for the message checking.

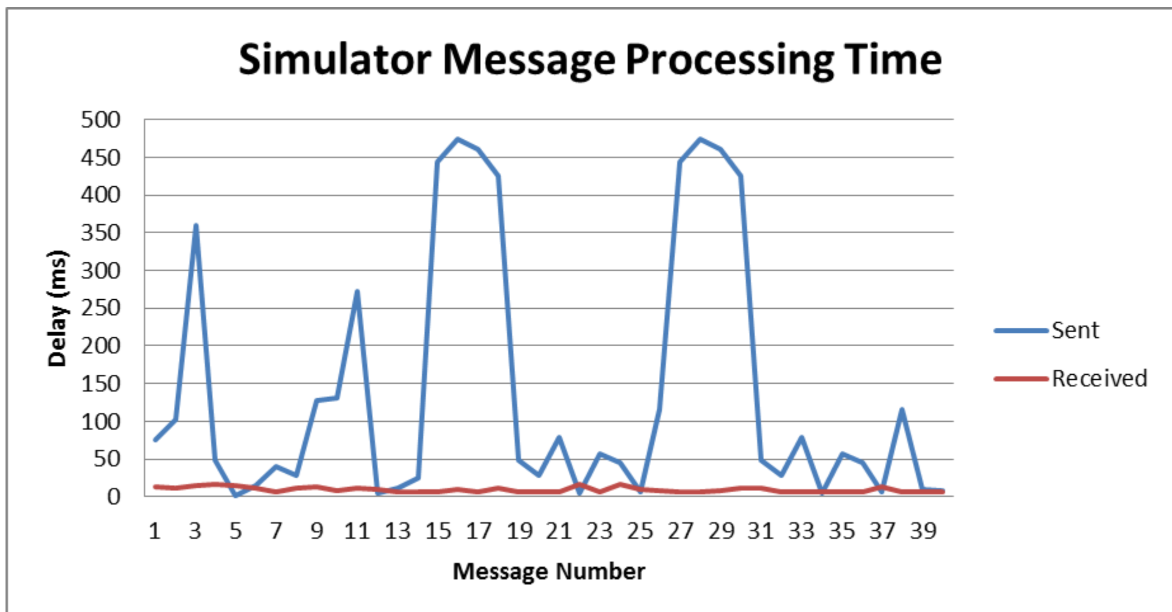


Figure 6-5 Simulator Message Processing Time

6.2.2 Point-to-Point Message Delay

In this scenario we will be analyzing the delay of sending and receiving a message through the overall architecture. The objective of this test is to analyze the overall delay of our architecture when integrated with a simulation framework. This test will also be

proving the overall success of the integration of mobile applications running in real phones with a simulated scenario.

Figure 6-6 represents the overall scenario where the red arrows represent what we will be measuring in this test. We will be measuring the delay from the point a message is sent from one application to the RabbitMQ server then, the respective OBU application will read the message and send it to the network. The destination OBU application will get the message, and then send it to the RabbitMQ server to be read by the destination application where the measurement will end. In this scenario, the vehicles are driving at a constant speed of 50 km/h with 300 meters distance between them. We also use a 40 message sample in order to obtain the results presented in the Figure 6-7.

We can observe in Figure 6-7 the results of this test, where the sending message has an average of 1446 msec, while the receive message has an average of 1302 msec. The results of the two operations should be similar and the slightly difference on the results can be justified by the same reason as presented in section 6.2.1.

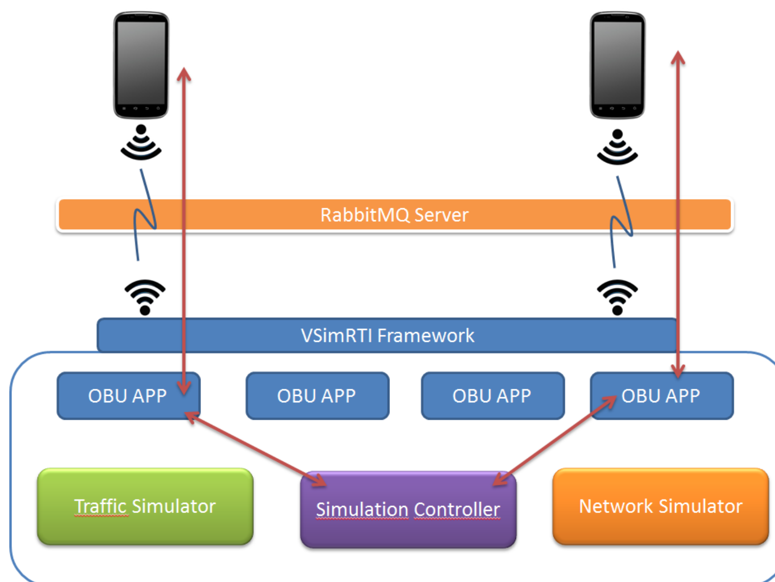


Figure 6-6 Simulation Overall Schema

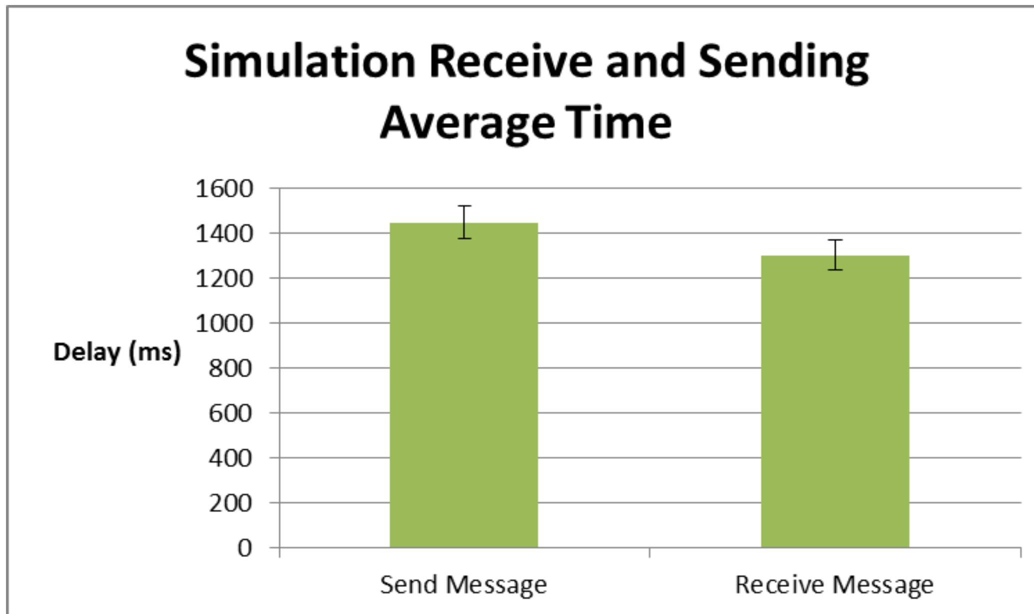


Figure 6-7 Simulation Receive and Sending Average Time

We believe the overall results of this test are according to our expectations. We expected the delays in the simulation scenario to be relatively high as the whole system is running in a single machine, and the flow of the communication between the two applications depends on the simulation processing.

6.2.3 Point-to-Point Message Delay with Distance Variation

This scenario is based on the one described in section 6.2.2, where we were measuring the point-to-point message delay between two applications. In this scenario, we will be measuring the same delay, but we will change the distance between the vehicles so we can understand how it affects the performance of REINVENT.

In this scenario we will be using a vehicle placement similar to Figure 6-8 with a constant speed of 45 km/h. We will be measuring the delay of sending and receiving messages with three different distances. The 50 meters represent a common distance in an urban scenario where vehicles drive relatively close to each other. The 600 meters represent the maximum communication distance in line of sight of the OBU's, and finally the 300 meters represent a midpoint between the two described.

Figure 6-9 presents the results of the test described. We can conclude from these results that the distance is not a relevant factor in REINVENT performance as the delay

variation is very small. For receiving a message, the values are very similar while in the sending we can observer a small difference of the values justified with the random factor induced by the message checker explained in section 6.2.1.

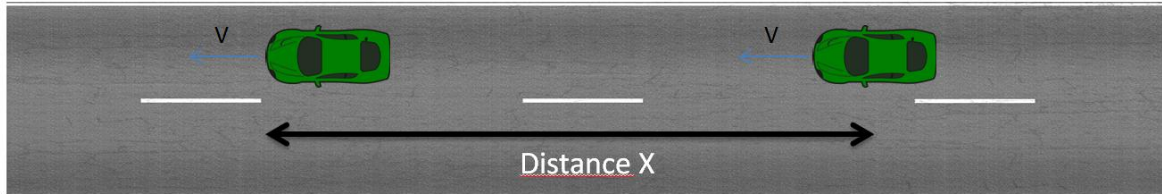


Figure 6-8 Point-to-Point delay with distance placement

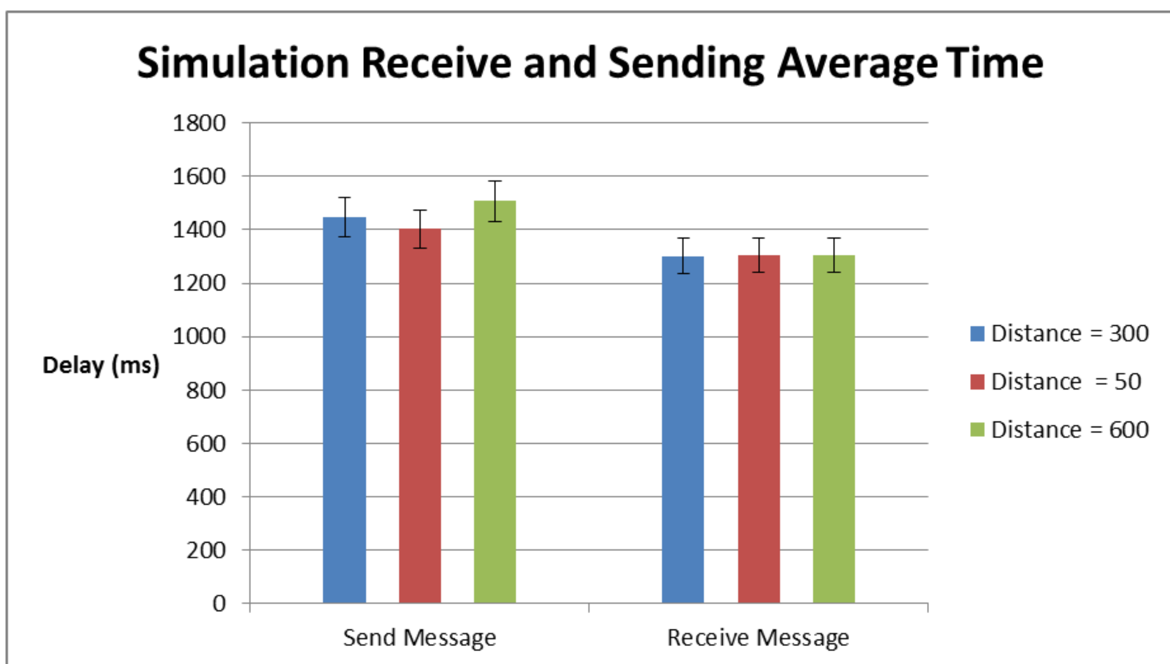


Figure 6-9 Receive and Sending Average Delay with distance

6.3 Real World Environment

After performing the tests in the simulated environment, we need to test our real scenario and analyze its performance. In this subsection we will explain the results obtained in the real world scenario. The scenario was described in chapter 5.2 and it is composed of two OBUs and Android devices running VNChat application. In the first test, we will be measuring the round trip time of a message in a single device in order to understand the performance of the overall architecture without the VANET delay. We will

be sending messages from the application, and the client running in the OBU will be returning the message back to the device, so we can measure the performance of the whole system architecture. In the second section of tests, we want to understand the impact of velocity and distance in the real scenario, so we will present tests with our setup using real vehicles in Aveiro roads.

We decided to measure the delay of the round trip time, since we can do the measure in a single device solving the problem of synchronization of a system clock

6.3.1 Architecture without VANET

In this test we measure the round trip time of a message in a single device, and it is measured in all the four devices with an 80 message sample. This test is very important as it will be showing us the real potential of REINVENT to be used in the developing of applications for vehicular network. It is important for the system to have very low delay, so it can be used in critical applications as safety application that require very little delay times to be considered effective. In this scenario, unlike the normal behavior of the system, the message handling client will be sending back a message once it is read from the rabbit server.

The overall schema of this test is represented on Figure 6-10 where the red arrows represent the path we will be measuring in this test.

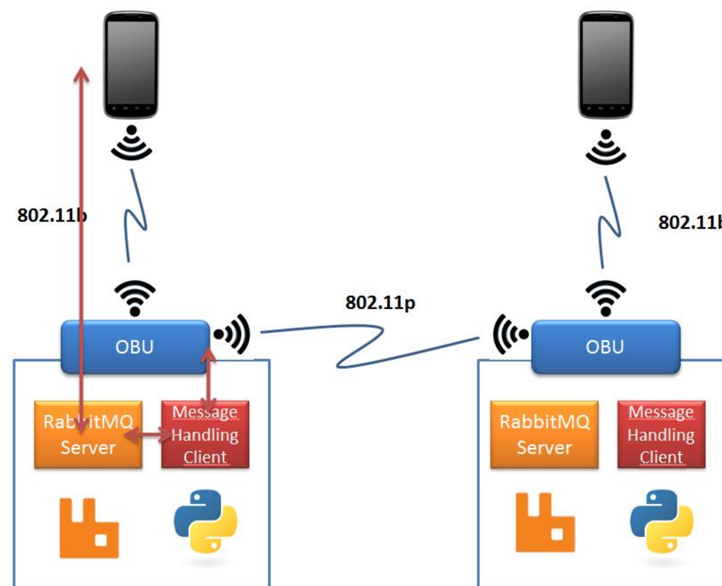


Figure 6-10- Round Trip Time Schema

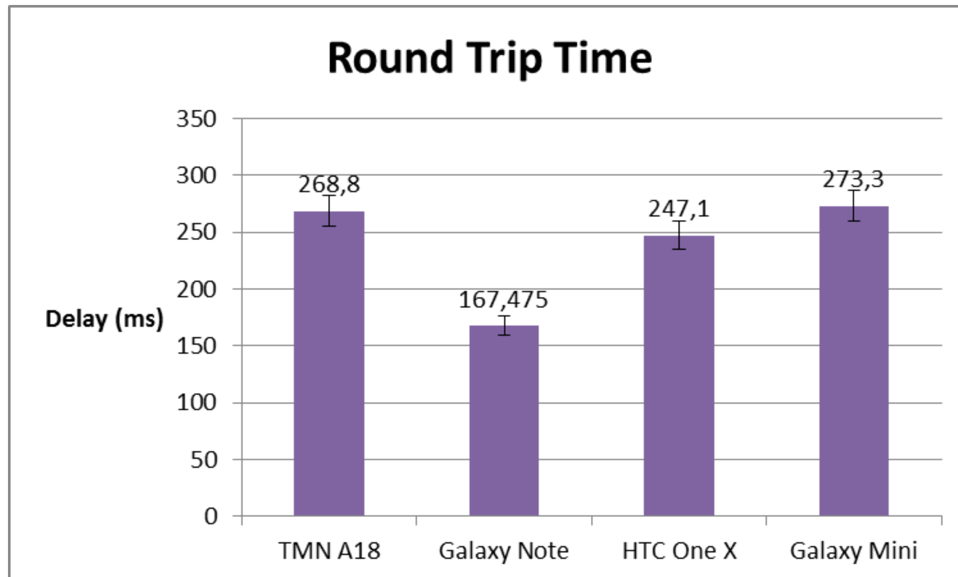


Figure 6-11 – Message Round Trip Time in the OBU

The Figure 6-11 shows the results of the message round trip time. Following the results of section 6.1, the Galaxy Note once again obtains the best results with an average of 167 msec, while Galaxy Mini obtains the largest delays with an average of 273 msec. These results once again can be justified by the hardware specifications as the Galaxy Note has much higher computing power as well as better communication interfaces. We obtained an overall average delay result of 240 msec, which we consider a good result.

6.3.2 Round Trip Delay with Distance Variation

The following tests are performed on the vehicles presented in chapter 5.2 and were made on a real road in Aveiro. In Figure 6-12, the red arrows represent the schema of what will be measured in the following tests.

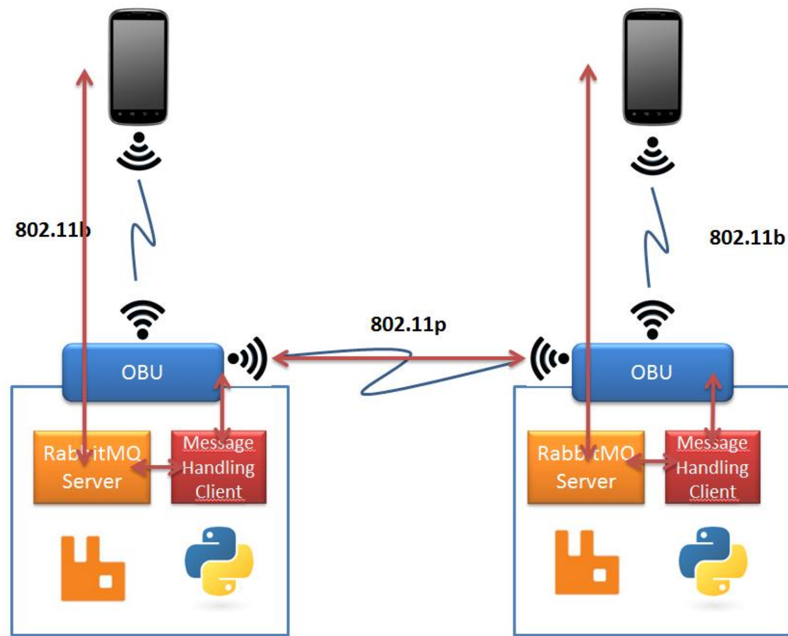


Figure 6-12 Round Trip Delay with VANET schema

The objective of this test is to understand how the distance between the vehicles affect the performance of applications using REINVENT in real scenarios.

In this scenario both vehicles will be stopped and will be placed with different distances like Figure 6-13 shows. We will be measuring the round trip delay with a sample of 40 messages for each distance. We used 4 different distances: 5 meters where the cars represent a situation where they are parked next to each other; the 100 meter distance is a common distance for an urban scenario; the 300 meters represent half of the maximum communication distance, and finally, the 600 meters represent the maximum communication distance of our OBU.

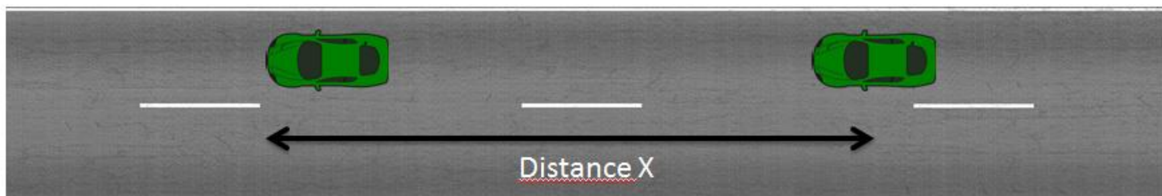


Figure 6-13 RTT with distance schema

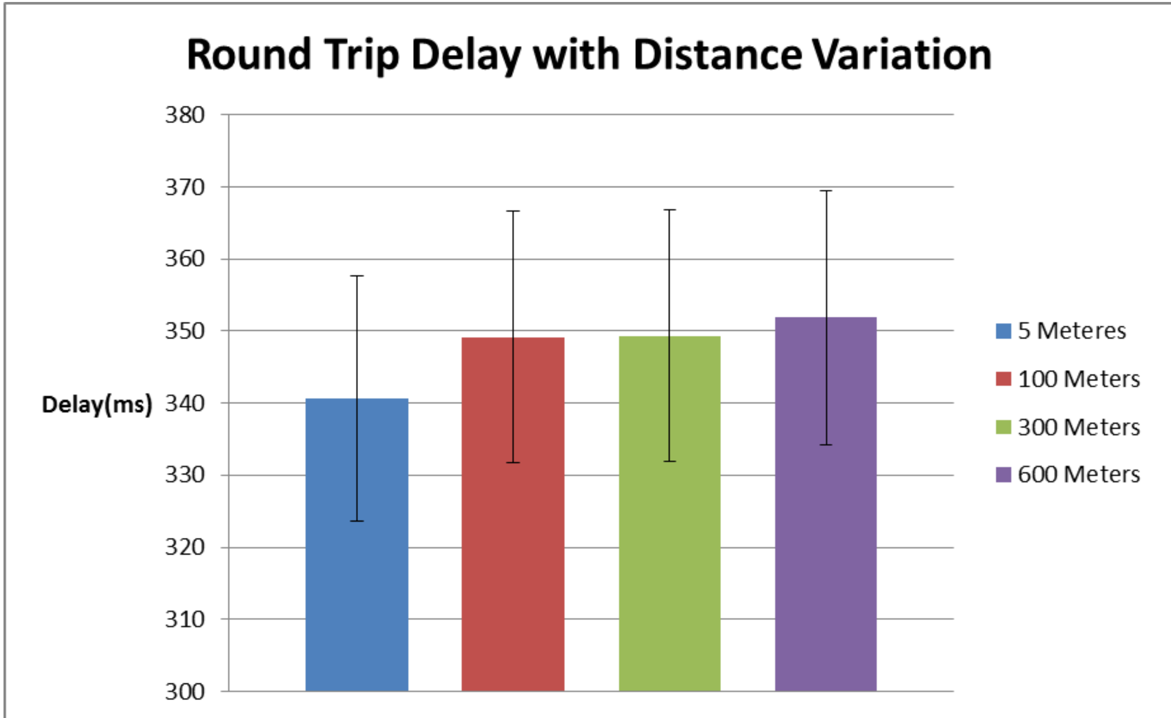


Figure 6-14 Round Trip Delay with Distance

The Figure 6-14 represents the results of this scenario. We can conclude from the obtained results that the distance is not a factor that impacts the performance of our application as the delay obtained between all the distances is very similar. Even at the maximum communication distance, the difference to the 5 meter mark is around 10 msec. The average delay of this test is 347 msec.

6.3.3 Round Trip Delay with Speed Variation

As we have seen in the previous scenario, the distance does not affect the performance of REINVENT. In this section we test the effect of vehicle speed in the performance.

In this scenario, depicted in Figure 6-15 and Figure 6-12, there is a stopped car while the other car will be driving around with different constant speeds: 20km/h, 50km/h and 80km/h. Unlike the other scenarios, we will not have a fixed message sample; we will be sending messages at a similar rate for each speed, so the number of samples will be decreased with the increase of speed.

Figure 6-15 Speed Variation Schema

We can observe from Figure 6-17, Figure 6-18 and Figure 6-18 that the message delays follow the same pattern; they start with high latency of messages, which represents the limit of connectivity where some of the messages were not received and consequently not measured. The latency starts to decrease with the proximity to the other vehicle, and the lower latencies are observed for the smallest distance between the vehicles. We can observe the high latencies in the last messages that represent, once again, the limit of connectivity between the vehicles.

It is important to note that the lowest delays are higher than the average of the delays obtained in all the other scenarios, which makes us conclude that the speed is an important factor and affects the performance of applications using REINVENT. We can also conclude that different speeds result in different delay values: the delays at 20km/h are considerably lower than at 80km/h. At the speed of 20km/h, excluding the limits of connectivity, we do not get any delay values above 600 msec mark, while this does not happen for 50km/h and 80km/h.

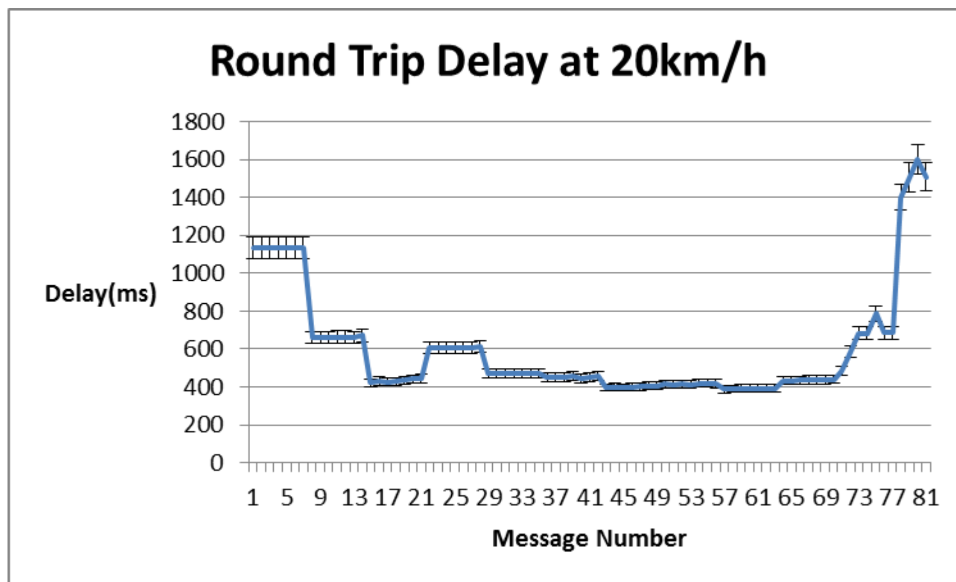


Figure 6-16 Round Trip Delay - 20km/h

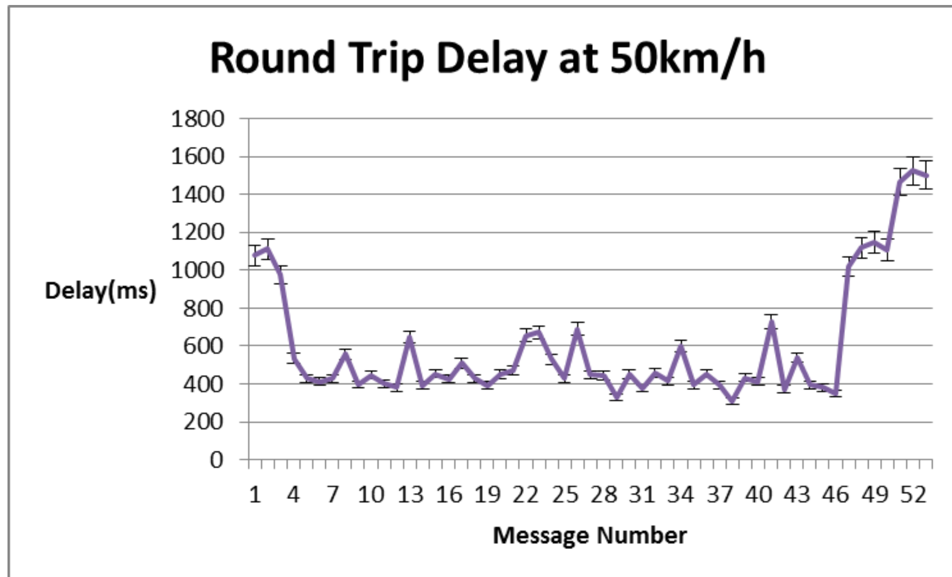


Figure 6-17- Round Trip Delay - 50km/h

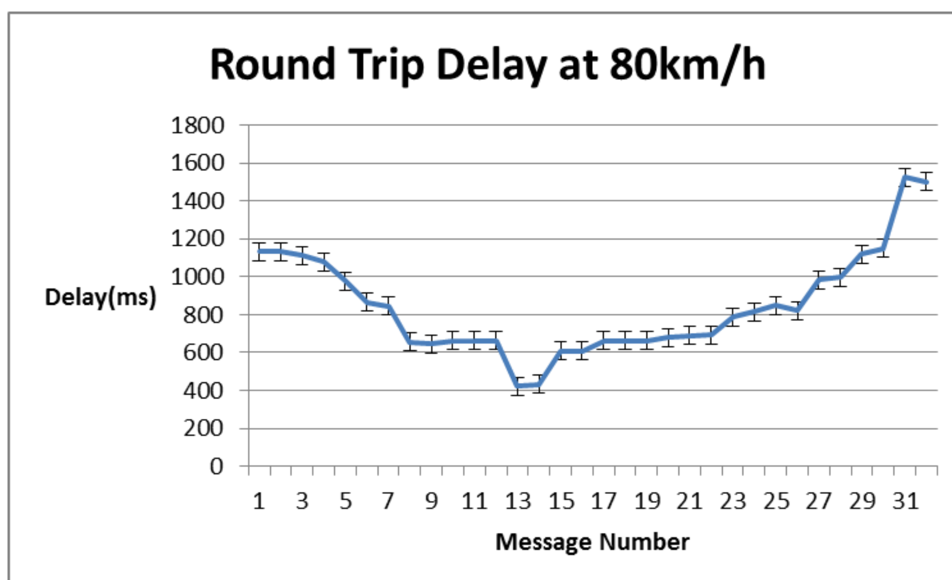


Figure 6-18 - Round Trip Delay - 80km/h

6.3.4 Messages per Second

The objective of this final test is to evaluate the performance of REINVENT with different loads of message traffic. This test was made in the laboratory environment, using the setup presented in the Figure 6-12, because we wanted to evaluate the behavior of REINVENT under heavy message loads, but we did not want our results to be affected by network conditions. With this setup we ensure the results obtained are under the best network conditions, and consequently the results reflect the REINVENT performance.

In this test we will be measuring the round trip delay of messages that will be sent at different rates. We will use a 30 message sample with 5 different rates of Messages per Second (MPS).

The results of this test are presented in Figure 6-19. We can observe that REINVENT performance is not affected for the 0.33, 0.5 and 1 MPS rates, as the delays of all the 30 messages are constant. For MPS of 2 and 4 MPS, we can observe a significant increase in the message delay right after the first message. Around the 10th message, the delay is over 5 seconds for both rates and the 30th message arrived approximately with 20 seconds delay.

We can that conclude the messages rate is a very relevant factor for the REINVENT performance especially at high rates. REINVENT supports up to 1MPS rate without any loss of performance, while higher constant MPS rates will increase it substantially. The reason for the increase of delay at higher MPS rates can be explained by the way we implemented REINVENT sending message service. The REINVENT will create a connection for each message, and consequently closes it after the message is sent. During this time, with high MPS rates, another message can be received by REINVENT to be sent, but it has to wait for the previous connection to the RabbitMQ server to be available, which will create a chain of delay. Also, due to the way *AsyncTask* works, the Android OS will only process 5 at the time while others stay in a sleep state, and even those 5 tasks are not processed in parallel but in a queue system.

A possible solution to solve these high delays could be to implement a bundle sending mode for the messaging service. Instead of creating a new task for every message, and consequently a new connection to RabbitMQ server, the task that is still active should check if there are any pending messages and handle them instead of creating a new task to be in the processing queue. The task should only end if no messages arrive in a defined time space.

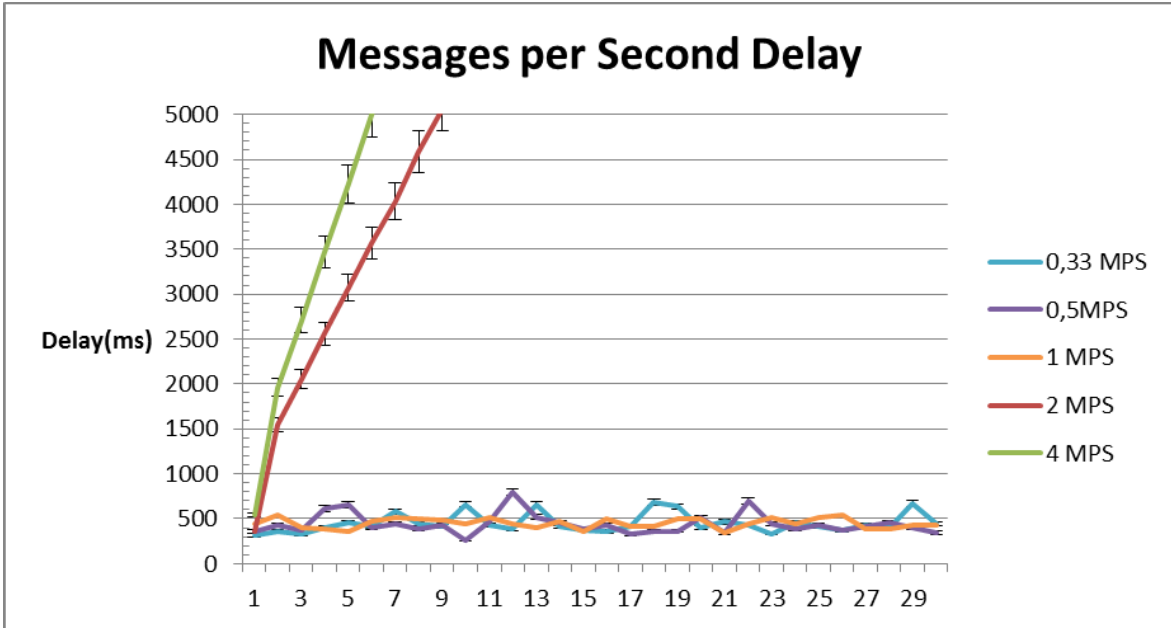


Figure 6-19 - Round Trip Delay with Message per Second variation

6.4 Conclusion

In this chapter we described and explained the results of the experimental tests performed, in order to verify and understand the performance of REINVENT in both simulated and real scenarios.

In the first section we tested the performance of the REINVENT at the device level by testing the basic communication features in different devices. We concluded that the device specification can reduce the delay induced by the module, but it is not a critical factor as we obtained very good delay results in both sending and receiving operations in all the devices.

The 6.2 section is relative to the tests made on the simulated environment. We started by measuring the overall delay of sending and receiving messages from and to the network as well as from and to the application, and we obtained slightly higher delays on the sending operation, that was justified by the message checking method that runs at a fixed time interval. We also measured the delay of sending and receiving messages with different distances between the vehicles, where we concluded that the distance was not a significant factor for REINVENT performance. Finally, we performed tests to measure the delay of operations inside the OBU simulated application, and we concluded that the only significant delay was the one from the message checking method.

The next tests were made on the road in order to understand the performance of REINVENT on real scenarios. We started by measuring the round trip delay of a message with the distance variation on stopped vehicles. We concluded the distance is not a factor that impacts the performance of REINVENT. The vehicles can exchange messages inside the communication limits without any increase of delay. In the next tests, we measured once again the round trip delay of a message, but in this scenario a vehicle was driving at different speeds while the other was stopped. We concluded that the speed of the vehicles affects the overall performance of REINVENT by increasing the delay of the messages, when comparing with the distance scenario, especially at high speeds where we obtained the highest delays. Finally, we evaluated the performance of REINVENT with different message per second rates. We concluded that the REINVENT performance is not affected for MPS rates lower than 1 MPS, while higher constant rates will increase significantly the delay of the messages due to the way we implemented the messaging service.

In order to understand how these delays translate to real world implications, we will need to convert this delay in meters so we can apply to a real road scenario. Using the following formula, ***(RelativeVelocity*1000/3600)*Delay***, we obtain the equivalent delay in a distance base instead of a time base. Considering a delay of 240ms and two vehicles driving in oposing direction each one at 50km/h, they will have a relative velocity to each other of 100km/h, and if we apply it to our formula we obtain a 6,6 meter value, which means that, if a given application sends a message reporting an accident at 350 meters, our architecture induces a maximum of 6,6 meteres error on the 350 meters value. Since the given scenario can be considered one of the worst case in a urban scenario, a 7 meter error can be considered tolerable. If we consider a highway scenario where vehicles can travel up to a relative speed of 240km/h to each other, our system will induce a 16 meter error.

Overall, we can conclude that REINVENT can be considered an efficient solution for developing applications for vehicular networks, since the delays obtained in the tests proved to be tolerable in most cases. We can also conclude that, although the simulated scenario can be used for developing and testing the overall application features in vehicular network scenario, it cannot be used as a reliable source when testing the performance of the applications, since the applications will always be affected by the delay induced by the simulation scheduling.

7 Conclusions and Future Work

The main objective of this Dissertation was to create a solution for integrating mobile applications in vehicular networks environment. We created a software architecture called REINVENT that abstracts both network and transport layers to the application level, by providing a high level interface to applications for communication through the vehicle network. Along the creation of REINVENT, a set of scenarios were created in order to test the viability and the proper functioning of the architecture. We created a simulated scenario based on a VANET simulation framework called VSimRTI that integrates a network and traffic simulator, as well as runtime application manager in order to simulate the OBU application of each car. In order to integrate real mobile devices into the simulation, we used our messaging service from the RabbitMQ for exchanging regular messages between the devices and the simulation in order for the devices to keep the state of the simulation. The other created scenario was a real world scenario by integrating REINVENT to the On Board Unit developed by the Instituto de Telecomunicações da Universidade de Aveiro. The OBU is running both the RabbitMQ server, where the applications will connect in order to send and receive messages, and a client that is responsible to encapsulate the messages to be sent from the applications in the WAVE Short Message Protocol, as well as receiving them and parse the information to be sent to the applications.

For testing and proof of concept purposes, we created two Android applications that implemented our REINVENT module and were integrated with our architecture. The first application is VNChat that is a message exchanging over the vehicle network, where the user can send messages to a known destination registered in the naming service of REINVENT, as well as receiving messages. The second application, iThere, is a location based application that shows the other iThere users around in a map or detailed view. The application sends in broadcast the actual location of the user every time it changes, so other users keep their information updated. Both applications use the REINVENT module to access a shared naming service in order to get the network identification of the users, and also to send messages through the network. In VNChat, the messages are identified with the destination ID, since the messages are sent to a single user, while in iThere they are

sent with broadcast identification on the destination, so that all the users can get the location information sent by the application.

After the implementation of the REINVENT and the applications, it was necessary to do experimental tests in order to understand if the module implied any restrictions in terms of the device specifications, or simply if it induced any delay on the applications behavior. We tested the delay of the basic features of the REINVENT module in devices with very different specifications. We used four devices, covering most of the Android device gammas, from a Galaxy Mini with very low specifications to the Galaxy Note which can be considered state of the art in mobile devices. The results from these tests proved that REINVENT does not have any required device specifications, since the delays between all the four devices were very similar. These results were also used to prove that the delay induced by the REINVENT does not affect the normal behavior of the application, as it can be considered irrelevant. In terms of the Android device, we can conclude that the REINVENT architecture can be used without any setback.

The next experimental tests concerned the performance of REINVENT in both real and simulated scenarios. In the simulated scenario, we measured the performance of the simulated OBU Application, where we concluded that the main source of delay was checking for new messages. The actual delay of processing messages inside the application was around 10 msec. The next tests were made in order to understand the delay of messages sent from an application to another. The results between send and receive messages were similar, around 1400ms; the distance variation proved not to be a relevant factor to the REINVENT performance in the simulation scenario. From the experimental tests and results obtained in this scenario, we can conclude that REINVENT can be used in simulation scenario for developing and testing applications features, but not for performance tests as the delays will always be affected by the simulation processing.

In the realistic scenario, we tested the performance of applications using REINVENT module in real vehicles on the road. We evaluated the impact of factors like distance and speed in REINVENT, where we concluded that distance is not a relevant factor while the speed of the vehicles can increase the delay especially for high speeds. Finally, we measured the performance of REINVENT with different loads of messages per second, where we concluded that REINVENT's performance is only affected for high rates.

From the tests performed in this work, we can conclude that REINVENT is a viable solution for integrating and creating Android applications for the vehicle network. The REINVENT allows any Android programmer to develop new applications in the VANET environment without having any major knowledge of the VANET specifications, since REINVENT provides a high level interface for applications to communicate through the vehicle network with high performance.

7.1 Future Works

There are several topics that can be further explored in the future related work.

On the one hand, the REINVENT module opens a window for implementing countless applications that only have the imagination as a limit. The ETSI Basic set of applications [12] has already several proposed applications that can be implemented as mobile applications using the REINVENT module. We also intend to integrate vehicle information with user information for more general applications.

On the other hand, the REINVENT module itself can be improved in several ways: new features can be added in order to open new scenarios of applications as well as improve the REINVENT performance by trying to minimize the delay of the architecture. We also intend to create a better solution for REINVENT to handle high rates of messages per second.

8 Bibliography

[1] V. Communications, “Vehicular Communications ; Basic Set of Applications ; Part 1 : Functional Requirements,” vol. 1, pp. 1–60, 2010.

[2] C. Forecast, “Cisco Visual Networking Index: Global Mobile data Traffic Forecast Update 2009-2014,” *Cisco Public Information*, pp. 2012–2017, 2010.

[3] M. Chen, J. Chen, and T. Chang, “Android/OSGi-based vehicular network management system,” *Computer Communications*, pp. 1644–1649, 2011.

[4] C. Palazzi, M. Roccetti, and S. Ferretti, “An intervehicular communication architecture for safety and entertainment,” *Intelligent Transportation Systems, IEEE Transactions*, vol. 11, no. 1v, pp. 1–31, 2010.

[5] T. Socolofsky C. Kale, “A TCP/IP Tutorial,” *http://tools.ietf.org/html/rfc1180#page-2*. .

[6] “Intelligent Transport Systems,” *http://www.etsi.org/technologies-clusters/technologies/intelligent-transport*. .

[7] C. Merlin and W. Heinzelman, “A study of safety applications in vehicular networks,” *IEEE International Conference on Mobile Adhoc and Sensor Systems Conference*, pp. 1–8, 2005.

[8] M. Gerla and L. Kleinrock, “Vehicular networks and the future of the mobile internet,” *Computer Networks*, vol. 55, no. 2, pp. 457–469, Feb. 2011.

[9] H. Moustafa, S. Senouci, and M. Jerbi, “Introduction to Vehicular Networks,” *Vehicular Networks: Techniques*, 2009.

[10] M. Nekovee, “Sensor networks on the road: the promises and challenges of vehicular adhoc networks and vehicular grids,” *Proceedings of the Workshop on Ubiquitous Computing and e-Research.*, 2005.

[11] G. Karagiannis and O. Altintas, “Vehicular networking: A survey and tutorial on requirements, architectures, challenges, standards and solutions,” *IEEE Communications Surveys & Tutorials*, vol. 13, no. 4, pp. 1–33, 2011.

[12] T. ETSI, “Intelligent transport systems (ITS); vehicular communications; basic set of applications; definitions,” 2009.

[13] C. 2 C. C. Consortium, “Car 2 car communication consortium manifesto,” *Braunschweig, November*, 2007.

[14] M. Sichitiu and M. Kihl, “Inter-vehicle communication systems: a survey,” *Communications Surveys & Tutorials, IEEE*, pp. 88–105, 2008.

[15] U. Hernandez, A. Perallos, N. Sainz, and I. Angulo, “Vehicle on board platform: Communications test and prototyping,” *2010 IEEE Intelligent Vehicles Symposium*, pp. 967–972, Jun. 2010.

[16] T. Al-ani, “Android In-Vehicle Infotainment System (AIVI),” University of Otago, 2011.

[17] Y. Cheng, W. Kuo, and S. Su, “An Android system design and implementation for Telematics services,” *2010 IEEE International Conference on Intelligent Computing and Intelligent Systems*, pp. 206–210, Oct. 2010.

[18] MOST, “Media Oriented Systems Transport,” *http://www.mostcooperation.com/home/index.html*. .

[19] C. Spelta, V. Manzoni, A. Corti, A. Goggi, and S. M. Savaresi, “Smartphone-Based Vehicle-to-Driver/Environment Interaction System for Motorcycles,” *IEEE Embedded Systems Letters*, vol. 2, no. 2, pp. 39–42, Jun. 2010.

[20] S. Diewald, A. Möller, L. Roalter, and M. Kranz, “DriveAssist-A V2X-Based Driver Assistance System for Android.,” *Mensch & Computer*, pp. 1–8, 2012.

[21] K.-C. Su, H.-M. Wu, W.-L. Chang, and Y.-H. Chou, “Vehicle-to-Vehicle Communication System through Wi-Fi Network Using Android Smartphone,” *2012 International Conference on Connected Vehicles and Expo (ICCVE)*, pp. 191–196, Dec. 2012.

[22] D. Yun and J. Lee, “Development of the eco-driving and safe-driving components using vehicle information,” *International Conference on ICT Convergence (ICTC)*, pp. 561–562, 2012.

[23] D. Yun and J. Lee, “Development of Mobile Common Component for providing vehicle information on mobile device,” *Computer Sciences and Convergence Information Technology (ICCIT)*, pp. 809 – 812, 2011.

[24] C. Campolo and A. Iera, “SMaRTCaR: An integrated smartphone-based platform to support traffic management applications,” *Vehicular Traffic Management for Smart Cities (VTM), 2012 First International Workshop on*, pp. 1 – 6, 2012.

[25] H. Stubing and P. Bechler, M.; Heussner, D.; May, T.; Radusch, I.; Rechner, H.; Vogel, “simTD: a car-to-X system architecture for field operational tests,” *IEEE Communications Magazine*, vol. 48, no. 5, pp. 148– 154.

[26] R. Stanica, E. Chaput, and A.-L. Beylot, “Simulation of vehicular ad-hoc networks: Challenges, review of tools and recommendations,” *Computer Networks*, vol. 55, no. 14, pp. 3179–3188, Oct. 2011.

[27] B. Schünemann, “V2X simulation runtime infrastructure VSimRTI: An assessment tool to design smart traffic management systems,” *Computer Networks*, vol. 55, no. 14, pp. 3189–3198, Oct. 2011.

[28] “Corridor Simulation (CORSIM) – Microscopic Traffic Simulation Model.,” *http://mctrans.ce.ufl.edu/featured/tsis/Version5/corsim.htm*. .

[29] “Verkehr In Stadten SIMulationsmodell (VISSIM),” *http://www.vissim.de*. .

[30] SUMO, “SUMO,” *http://sumo.sourceforge.net/*. .

[31] S. Joerer, C. Sommer, and F. Dressler, "Toward reproducibility and comparability of IVC simulation studies: a literature survey," *IEEE Communications Magazine*, vol. 50, no. 10, pp. 1–7, 2012.

[32] CanuMobiSim, "CANU Mobility Simulation Environmenttle," *http://canu.informatik.uni-stuttgart.de/mobisim/*. .

[33] "The Network Simulator - ns-2," *http://www.isi.edu/nsnam/ns/*. .

[34] "Qualnet," *http://web.scalable-networks.com/content/qualnet*. .

[35] M. Fiore, J. Harri, F. Filali, and C. Bonnet, "Vehicular mobility simulation for VANETs," *Simulation Symposium*, 2007.

[36] F. K. Karnadi, Z. H. Mo, and K. Lan, "Rapid Generation of Realistic Mobility Models for VANET," *2007 IEEE Wireless Communications and Networking Conference*, pp. 2506–2511, 2007.

[37] E. Weingartner, "A performance comparison of recent network simulators," *IEEE International Conference on Communications*, pp. 1 – 5, 2009.

[38] "The Network Simulator - ns-3," *http://www.nsnam.org/*. .

[39] "OMNeT++," *http://www.omnetpp.org/*. .

[40] "VEINS," *http://veins.car2x.org/*. .

[41] R. G. and F. D. Christoph Sommer, "Bidirectionally Coupled Network and Road Traffic Simulation for Improved IVC Analysis," *IEEE Transactions on Mobile Computing*, vol. 10, pp. 3–15.

[42] JiST SWANS, "Java in Simulation Time / Scalable Wireless Ad hoc Network Simulator," *http://jist.ece.cornell.edu/*. .

[43] M. Piorkowski and M. Raya, "TraNS: realistic joint traffic and network simulator for VANETs," *CM SIGMOBILE Mobile Computing and Communications Review*, vol. 12, no. 1, pp. 31–33, 2008.

[44] "TraNS - Traffic and Network Simulation Environment," *Traffic and Network Simulation Environment*. .

[45] V. Kumar, L. Lin, D. Krajzewicz, F. Hrizi, O. Martinez, J. Gozalvez, and R. Bauza, "iTETRIS: Adaptation of ITS Technologies for Large Scale Integrated Simulation," *2010 IEEE 71st Vehicular Technology Conference*, pp. 1–5, 2010.

[46] "iTetris - The Integrated Wireless and Traffic Platform for Real-Time Road Traffic Management Solutions," *http://www.ict-itetris.eu/introduction.htm*. .

[47] R. T. Fielding and R. N. Taylor, "Principled design of the modern Web architecture," *ACM Transactions on Internet Technology*, vol. 2, no. 2, pp. 115–150, May 2002.

[48] E. Curry, "Message-oriented middleware," *Middleware for communications*, 2004.

[49] AMQP, “Advanced Messaging Queue Protocol,” *http://www.amqp.org/product/overview http://www.amqp.org/product/architecture*. .

[50] STOMP, “Simple Text Oriented Messaging Protocol,” *http://stomp.github.io/stomp-specification-1.2.html*. .

[51] JMS, “Java Messaging Service,” *http://www.oracle.com/technetwork/java/index-jsp-142945.html*. .

[52] VMware, “RabbitMQ , Messaging that works,” *http://www.rabbitmq.com/*. .

[53] “Smartphone US Market Share,” *http://www.comscore.com/Insights/Press_Releases/2013/5/comScore_Reports_March_2013_U.S._Smartphone_Subscriber_Market_Share*. .

[54] “Smartphone EU Market Share,” *http://www.comscoredatamine.com/2013/02/samsung-leads-european-smartphone-market-ahead-of-apple/*. .

[55] “Getting Started with Android,” *http://www.arm.com/community/software-enablement/google/solution-center-android/getting-started-with-android-on-arm.php*. .

[56] “Android Developers - Overview,” *http://developer.android.com/about/index.html*. .

[57] M. N. Zigurd Mednieks, Laird Dornin, G. Blake Meike, *Programming Android, 2nd Edition*. O’Reilly Media.

[58] Android Developers, “Android Developers - Content Providers,” *http://developer.android.com/reference/android/content/ContentProvider.html, http://developer.android.com/guide/topics/providers/content-provider-basics.html*. .

[59] V. Dobjanschi, “Developing Android REST client applications,” *Google I/O 2010*, 2010.

[60] “Android Developers - Content Observer,” *http://developer.android.com/reference/android/database/ContentObserver.html*. .

[61] A. Wegener and M. Piórkowski, “TraCI: an interface for coupling road traffic and network simulators,” *Proceedings of the 11th …*, pp. 155–163, 2008.

[62] T. Queck, B. Schünemann, I. Radusch, and C. Meinel, “Realistic Simulation of V2X Communication Scenarios,” *2008 IEEE Asia-Pacific Services Computing Conference*, pp. 1623–1627, Dec. 2008.

[63] Android Developers, “Managing AVD,” *http://developer.android.com/tools/devices/managing-avds.html*. .

[64] F. Kage, “VSimRTI : Vehicle-2-X Simulation Runtime Infrastructure,” 2012.

[65] W. G. of the 802 Committee, “Information technology–Telecommunications and information exchange between systems–Local and metropolitan area networks–Specific requirements,” *IEEE Std*, 2009.

[66] “ZTE Grand X,” *http://www.gsmarena.com/zte_grand_x_in-4962.php*. .

[67] “HTC One X,” *http://www.gsmarena.com/htc_one_x-4320.php*. .

[68] “Samsung Galaxy Mini,” *http://www.gsmarena.com/samsung_galaxy_mini_s5570-3725.php*. .

[69] “Galaxy Note,” *http://www.gsmarena.com/samsung_galaxy_note_lte_10_1_n8020-5151.php*. .

Annex

Rest API Description

Name	*URI*	*Method*	*Description*
GetAllUsers	Content://AUTHORITY/users	Query	Returns all users from the naming service
GetUser	Content://AUTHORITY/users/#	Query	Returns a specific user from the naming service
InsertUser	Content://AUTHORITY/users	Insert	Inserts an entry in the naming service
DeleteUser	Content://AUTHORITY/users/#	Delete	Deletes an entry in the naming service
UpdateUser	Content://AUTHORITY/users/#	Update	Updates the information of an entry in the naming service
NewMessage	Content://AUTHORITY/newmessage	Observable	This method does not return any value, it is used to listen for new incoming messages
SendMessage	Content://AUTHORITY/users/#/send	Insert	This method is used to send a message using an entry from the naming service as the destination.
Close	Content://AUTHORITY/close	Insert	This method is used to stop the listening thread of the provider
GetNewMessages	Content://AUTHORITY/getmessa	Call	Returns any new

	ges	messages from a given type.

Table 5 - API Methods Overview

GetAllUsers

This method is used by creating a ContentResolver, an Android structure for interfacing and interacting with a content provider, and by simply invoking the *query* method with the *content://AUTHORITY/users* URI, the provider returns a cursor that provider random read access to the result set returned by the naming service. This method was created so the applications can have access to all the users registered in the naming service.

GetUser

Similar to the previous method, GetUser is used to access the naming service and return a specific entry. It sould be used by creating a ContentResolver and invoking the *query* method with the *content://AUTHORITY/users/#* URI, where the # character works as wildcard referring a specific id in the naming service. The provider returns a cursor that points to a single entry if it exists, otherwise it returns null.

InsertUser

This method is used to insert an entry in the naming service, and is used by invoking the *insert* method with the *content://AUTHORITY/users/ URI* on the content resolver. The information of the entry being inserted should be passed as an argument on the *insert* method using another android structured used to pass argument information in the content providers, a ContentValues, a structure that works like a HashTable, associating a key to the value. The keys for the entries attributes are described in the UserDescriptor class, *UsersDescriptor.NAME*, for the name of the entry, and *UsersDescriptor.ALIAS* for the alias of the entry.

DeleteUser

The objective of this method is to permanently delete an entry on the naming service, similar to GetUser, it should be used with the same URI, *content://AUTHORITY/users/#* as the argument of the *delete* method invoked on a content resolver.

UpdateUser

This method is very similar to the InsertUser, but the URI used in this method should be , *content://AUTHORITY/users/#* as it will update the information of a already existing entry with the ContentValues passed on the *update* method.

NewMessage

NewMessage cannot be considered as a method and should be seen as a resource. The URI associated with this resource is *content://AUTHORITY/newmessage* and it should be used by the applications with a ContentObserver, a structure that will be listening for notifications on a given URI. The provider sends a notification on this URI every time it receives a new message, so, if the applications want to receive notifications of new messages available on the provider they should create a ContentObserver and listening to this specific URI. The content observer structure must implement the OnChanged method that will be called when the ContentObserver receives the notification from the provider.

SendMessage

This method is used to send a message to a specified naming service entry or simply in broad cast. The URI associated with this resource is c*ontent://AUTHORITY/users/#/send* for sending to a specific entry where the # represents the id of the entry or c*ontent://AUTHORITY/users/send* for sending a message in broadcast. The information to be sent should be passed in a Content Value structure by invoking the insert method on a Content Resolver.

Close

This method is used to stop the listening thread and consequently to stop listening for new messages. It will affect not only the application invoking it, but all the applications that use the content provider. The URI associated is Content://AUTHORITY/close and should be called within the Call method on a Content Resolver with the “close” as the method with no arguments.

GetNewMessage

This method is used for getting new messages from the content provider. The URI associated is c*ontent://AUTHORITY/getmessages* and should be used by invoking the Call method in a Content Resolver, using “getmessages” as the method and the type of the

message desired should be passed as the argument. It returns a bundle containing an ArrayList of MessageCountainers.