Universidade de Aveiro
2014

Departamento de Eletrónica, Telecomunicações e Informática

Paulo Jorge Carvalho dos Santos

VANESS: DNS em redes veiculares para suporte a utilizadores itinerantes

VANESS: DNS for nomadic users in vehicular networks

Universidade de Aveiro
2014

Departamento de Eletrónica, Telecomunicações e Informática

Paulo Jorge Carvalho dos Santos

VANESS: DNS em redes veiculares para suporte a utilizadores itinerantes

VANESS: DNS for nomadic users in vehicular networks

Tese 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 do Professor Doutor José Maria Fernandes, Professor auxiliar do Departamento de Eletrónica, Telecomunicações e Informática da Universidade de Aveiro e co-orientação do Doutor André Cardote, Engenheiro de sistemas na Veniam Lda.

O júri

Presidente

Professor Doutor Rui Luís Andrade Aguiar
Professor Associado C/ Agregação do Departamento de Eletrónica, Telecomunicações e Informática da Universidade de Aveiro

Vogais

Professor Doutor Pedro Miguel Alves Brandão
Professor Auxiliar da Faculdade de Ciências da Universidade do Porto
(Arguente Principal)

Professor Doutor José Maria Amaral Fernandes
Professor Auxiliar do Departamento de Eletrónica, Telecomunicações e Informática da Universidade de Aveiro
(Orientador)

Agradecimentos

Quero agradecer a todos os professores por contribuírem para o meu percurso académico. A todos os colegas pelos bons momentos. A toda a minha família à qual não pude dar a devida atenção durante estes anos e em especial aos meus pais por me terem apoiado de forma incansável desde sempre.

Agradeço à Veniam Works por ter cedido o equipamento necessário para a realização deste trabalho bem como aos seus colaboradores pela sua disponibilidade.

Agradeço aos professores José Maria Fernandes, à professora Susana Sargento e ao Doutor André Cardote por terem coordenado e motivado o desenvolvimento deste trabalho sempre com um bom ambiente e grande disponibilidade.

Palavras-chave

Redes veiculares, VANET, comunicação utilizador-a-utilizador, REINVENT, gateway, serviço de nomeação, Android.

Resumo

As redes veiculares, também conhecidas por VANETs, são redes ad-hoc formadas por veículos e road-site units. Hoje em dia estas redes têm atraído bastante interesse tanto por parte de investigadores como da indústria automóvel. Com o aparecimento de sistemas operativos especificamente dedicados a automóveis e carros de condução autónoma, o uso de aplicações em veículos e a integração com dipositivos móveis está-se a tornar uma parte cada vez maior nas VANETs. Apesar dos grandes avanços tecnológicos realizados nesta área, ainda existe uma grande discrepância entre os serviços de camada de comunicação disponibilizados pelas VANETs e os serviços ao nível do utilizador, nomeadamente aqueles acessíveis a partir de aplicações móveis noutras redes e tecnologias. Os utilizadores e programadores estão habituados a interações utilizador-utilizador ou utilizador-empresa sem preocupações explícitas sobre a camada de transporte em causa. Isto não é possível em redes VANETs, uma vez que cada pessoa pode usar vários veículos. No entanto, uma mensagem para ser enviada para um utilizador específico através destas redes precisa do indentificador do veículo onde tal utilizador está, ou seja, existe uma falta de serviços que mapeiem cada utilizador individual a nós na VANET (identificação do veículo).

O trabalho desta dissertação propõe o sistema VANESS, um serviço de nomes (*naming service)* disponibilizado como um recurso para suporte a comunicação utilizador-a-utilizador num cenário heterogéneo englobando o típico ISP e VANETs focadas em dispositivos móveis. O sistema proposto é capaz de mapear um utilizador a um veículo tanto localmente (i.e. não existe ligação à internet), online (i.e. o sistema não está numa rede veicular mas tem acesso direto à internet) e usando um gateway (i.e. o sistema está numa rede veicular onde algum nó tem acesso à internet e irá servir como gateway).

VANESS foi integralmente implementado em Android OS, onde os resultados dos testes mostram que é um sistema viável, e parcialmente em iOS mostrando a sua capacidade para multi-plataformas.

Keywords

Vehicular network, VANET, user-to-user communication, REINVENT, gateway, naming services, Android.

Abstract

Vehicular networks, also known as VANETs, are an ad-hoc network formed by vehicles and road-side units. Nowadays they have been attracting big interest both from researchers as from the automotive industry. With the upcoming of automotive specific operating systems and self-driving cars, the use of applications on vehicles and the integration with common mobile devices is becoming a big part of VANETs. Although many advances have been made on this field, there is still a big discrepancy between the communication layer services provided by VANETs and the user level services, namely those accessible through mobile applications on other networks and technologies. Users and developers are accustomed to user-to-user or user-to-business communication without explicit concerns related with the available communication transport layer. Such is not possible in VANETs since people may use more than one vehicle. However, to send a message to a specific user in these networks, there is a need to know the ID of the vehicle where the user is, meaning that there is a lack of services that map each individual user to VANETs endpoint (vehicle identification).

This dissertation work proposes VANESS, a naming service as a resource to support user-to-user communication within a heterogeneous scenario comprising typical ISP scenario and VANETs focused on mobile devices. The proposed system is able to map the user to an end point either locally (i.e. there is not internet connection at all), online (i.e. system is not in a vehicular network but has direct internet connection) and using a gateway (i.e. the system is in a vehicular network where some of the nodes have internet access and will act as a gateway).

VANESS was fully implemented on android OS with results proving his viability, and partially on iOS showing its multi-platform capabilities.

I. Contents

I. Contents i
II. List of figures iii
1 Introduction 1
1.1 The problem 1
1.2 Our solution 2
1.3 Dissertation Outline 2
2 State of the Art 5
2.1 Vehicular Networks and Applications 5
2.2 Mobile device use on VANETs 8
2.3 Naming Services 10
2.4 Conclusions 11
3 VANESS: DNS for nomadic users in vehicular networks 13
3.1 REINVENT architecture 13
3.2 VANESS architecture 14
3.3 VNS: VANET Naming Service 16
3.3.1 VNS Protocol 18
3.3.2 VNS messages 20
3.3.3 Server Synchronization 21
3.3.4 Cache Management 22
3.4 VnAuth | Vehicular Network Authentication 22
3.4.1 Why VnAuth? 22
3.4.2 VnAuth Protocol 23
3.5 Summary 23
4 VANESS Implementation 25
4.1 Relevant concepts 25
4.1.1 RabbitMQ: The Messaging Broker 25
4.1.2 Android’s Broadcast Receivers 26
4.1.3 Android’s Account Manager 26
4.1.4 Android’s Content providers and Observers 27

4.2 VANESS implementation architecture 28
4.3 VNS Extensions to REINVENT 29
4.3.1 VNS Content Provider Interface 30
4.3.2 The REINVENT developer interface 35
4.3.3 Gateway: VANETs and the world connected 37
4.3.4 VnAuth Developer Interface 39
4.4 Summary 41
5 VANESS: applications and tests 43
5.1 VNRide 43
5.2 iOS Consumer App 46
5.3 VNBrowser: The gateway's proof of concept 46
5.4 System Tests 47
5.4.1 Simplified tests 48
5.4.2 Real setup Tests 50
5.4.3 Test conclusions 54
5.5 Summary 55
6 Conclusions 57
6.1 Future Work 57
7 Bibliography 59
8 Appendices 63
8.1 VANESS deployment 63
8.2 Parameter list for each message type 64

II. List of figures

Figure 1 - VANET typical architecture [17]: cars exchange messages between them through the OBU interface. To communicate with internet resources the VANET uses RSUs. ITS services can be either available using the internet or the directly through the VANET 5

Figure 2 - One of the OBUs used. It provides Wi-Fi connection so common devices can connect to it and WAVE which is used to communicate with other OBUs/RSUs 7

Figure 3 - current architecture of VANET-Mobile interaction: VANET is formed by vehicles, on-board units and road-ride units. These units may have internet connection, serving as gateway to the internet. Each mobile phone connects via Wi-Fi to one on-board unit which relays messages between the VANET and the phone and/or the internet... 8

Figure 4 - The progress of a message on the system, between phones: communication between each phone and the OBU uses rabbitMQ and between OBUs (via VANET) uses wave protocol. 14

Figure 5 - VANESS architecture: The phone has modules to manage accounts, communications and the naming service; The OBU has proxy modules for communication between the phone and the vehicular network (REINVENT) and the phone or vehicular network to the internet (gateway)... 15

Figure 6 – The mobile applications use alias/usernames to refer to message receivers/senders. It is VANESS' task to ensure that the messages are received by the correct persons, regardless of their vehicle's ID... 16

Figure 7 – Interaction between the system's modules when user "Paulo" sends a text message to "Tiago" and he responds back 17

Figure 8 – Basic representation of the connections of the systems' parts: REINVENT uses VNS to ask for the username translation; VnAuth reports when any user logs in to VNS; NetworkChangeListener warns VNS each time mobile phone connects to a different OBU (different vehicle ID). In turn, VNS uses REINVENT to send messages through the vehicular network and uses VNSAccess to make server calls. 18

Figure 9 - The progress of DNS messages thought VANET with gateway: When the phone sends a message it is initially sent to the local OBU, using REINVENT which in turn uses RabbitMQ. The local OBU will broadcast that message to the vehicular network. If it arrives to a Gateway it convert it to HTTP and calls the server. 19

Figure 10 - DNS_TRANSLATE (blue) broadcasted and DNS (orange) response from the addressee. Notice that only the user present on the request message responds. All the nodes on the network may update their cache with the response. 20

Figure 11 – Some typical RabbitMQ supported models of operation, as illustrated in [40]. Note to mention that RabittMQ provides programming interfaces for development languages. 25

Figure 12 - Accounts section on Settings screen in an Android OS 27

Figure 13 - VANESS architecture: The phone has modules to manage accounts, communications and the naming service; The OBU has proxy modules for communication

between the phone and the vehicular network (REINVENT) and the phone or vehicular network to the internet (gateway) ... 28

Figure 14 - OBU/Rabbit Address - Manual Mode: This configuration screen comes with the installation of the VNS module. Whenever the user defines the address in this field, it is always used; ... 32

Figure 15 - OBU/Rabbit Address - Dynamic mode: If the OBU address field is left empty, the system will use the gateway's address by default. The image on the right shows a manual configuration of the network, however it equally works using DHCP ... 33

Figure 16 - VNS Server API, translate action ... 38

Figure 17 - VNS Server API, register action ... 39

Figure 18 - Account Manager life cycle ... 40

Figure 19 - VnAuth login button integrated in VnRide on the left; the authentication activity on the right. This activity appears if it is the first time logging-in on that device ... 43

Figure 20 - Chat screen from VnRide, talking with JoaoSilva ... 44

Figure 21 - Android's account settings for VnAuth: This screen is accessed via Setting application from the Android system and allows to define multiple settings for use of the network and account management ... 45

Figure 22 - VnRide screens. In order: Search for a ride (A); Sharing my ride (B); My timeline/notifications (C); ... 45

Figure 23 - VNS consumer app for iOS: this screen shows the emulator running the consumer application for VNS. Below the button we see a list of associations user-vehicle, returned from the server. This list contains the most recent ID of all the OBUs each user connected to. ... 46

Figure 24 - Initial screen from VNBrowser and a request/response example to Google ... 47

Figure 25 – First demo application: Home screen (in a tablet) with the user emulating his position on a car with the ID 'veh_0'. ... 48

Figure 26 - On the left, the example of a message being sent; On the right, emulating the position of the vehicle ... 49

Figure 27 - City simulator, simulating the movement of passengers on taxis and buses ... 49

Figure 28 - Our test scenario: Two OBUs with VANET + Wi-Fi capabilities, one public web server with VNS services running and two mobile devices. The phone was connected to one OBU and the tablet to the other one. The OBUs are connected to the PC to debug proposes and, in some scenarios, to get internet access through the PC's Ethernet connection. ... 50

Figure 29 – setup for one test example: two devices, with a different user and connected to different vehicles. A tablet (on the left) and a phone (on the right) ... 51

Figure 30 – Demo of two users chating on VANET ... 51

Figure 31 - Log messages from one of the OBUs. VNS server was online; A message was sent to a inexistent user (ghost) resulting in a "No translation found". ... 52

Figure 32 - Log messages from one of the OBUs. VNS server was online; A message was sent to an existent user (pjcs) resulting in a response from the server with the vehicle ID = 12. ... 52

Figure 33 - Debug messages from one OBU when two users exchanged messages. First a message was sent from "pjcs" to "paulo" saying "test"; then a response saying "test2" 52

Figure 34 - Debug messages for the Gateway action, when the local OBU has internet connection 53

Figure 35 - Debug messages for the Gateway action, when OBU receives a request from the VANET... 53

1 Introduction

Automotive industry is a big part of society all over the world, with about 1 billion vehicles on the road [1] especially concentrated in cities. With this boom in automotive industry, there came a lot of problems like accidents, which are the eighth leading cause of death globally, and the leading cause of death for young people [2], or traffic congestion generating waste of our time and fuel. Besides that many mainstream services are not available on traffic scenarios due to the loss or quality declension of internet on road/high speed situations. Many advances, namely VANETs, have been made to minimize this and other problems.

Vehicular ad hoc networks (VANETs) are mobile networks where nodes are vehicles or road site units. They communicate with other nodes within the range, which in turn re-send those messages to the destination, increasing reachability.

Nowadays, vehicular networks have been attracting big interest both from researchers and from the automotive industry. Some common use cases are close-formation platoons [3], earlier and safe responses [4] and identification of hazards [5]. Commonly used devices like smartphones may also be part of the network indirectly, by connecting to the vehicle which will work as a proxy between the phone’s applications and the vehicular network.

This year an enormous breakthrough in the use of mobile apps on cars happened when Apple, Google and Microsoft decided to bring their mobile operating systems to the car. Apple launched CarPlay [6] which connects to an iOS device. Apple says "CarPlay takes the things you want to do with your iPhone while driving and puts them right on your car's built-in display" [6]. Google introduced the Android Auto [7], as part of Open Automotive Alliance [8], to make the use of mobile applications safer while driving, but it also aims to provide a fully functional OS on the car which will work even without being connected to any smartphone. The same goes for Microsoft whom also has Windows Embedded Automotive [9]. Google announced a self-driving car [10] still in prototype. The upcoming of these vehicles makes more evident the need of infotainment application on the road.

1.1 The problem

Although many advances have been made on this field, there is still a big discrepancy between the communication layer services provided by VANETs [5, 11, 12, 13, 14] and the user level services, namely those accessible through mobile applications. Users are accustomed to user-to-user or user-to-business communication without explicit concerns related with the available communication transport layer. Such is not possible in VANETs since people may use more than one vehicle throughout time. There is a lack of services that map each individual user to VANETs endpoint (vehicle identification) in real time. However, to send a message to a specific

user in the VANET, there is a need to find the identification (ID) of the vehicle where the user is, in order to send the message.

1.2 Our solution

We propose VANESS, a DNS for nomadic users in vehicular networks, a naming service that can be used as a resource to support user-to-user and user-to-business communication within a heterogeneous scenario comprising typical ISP (Internet Service Provider) scenario and VANETs focused on mobile devices.

VANESS is able to map the user to an end point either locally (i.e. there is not internet connection at all), online (i.e. system is not in a VANET but has direct internet connection) and using a gateway (i.e. the system is in a VANET where some of the nodes have internet access and will act as a gateway). VANESS provides authentication and communication modules that allow applications to keep track of users in the VANET or outside, providing support to logical user-to-user communication.

This system was fully developed and tested, extending a pre-existent system for communication between mobile applications on vehicular networks – REINVENT [15]. Using the final system, VANESS, any developer can build mobile applications with communication between users through the vehicular networks, without knowing à priori which is the vehicle where they are connected.

VANESS was tested in simplified and controlled scenarios, as well as real test scenarios using two on-board units. After these tests, we concluded that, although it does not provide a delivery guarantee, VANESS is a viable and scalable prototype, which simplifies the development of user-level applications.

1.3 Dissertation Outline

This dissertation is divided in seven Chapters. In this Chapter we presented, in a generic way, the context, the problem, and how we aim to solve it.

In Chapter 2, an initial overview on the current state of vehicular networks technology is presented, focused on what they do, how they do it and why they can be useful. We then present the state of the art describing work which address similar problems as ours, and in some cases influenced our own solution. In the process, insights about the use of mobile devices (e.g. smartphones) on vehicular networks are presented and existing naming service solutions are described.

In Chapter 3, VANESS, this dissertation main contribution, is presented. After describing VANESS main concepts and rationale, a description on main details on each of the modules' features and architecture is presented.

Chapter 4 presents the implementation details and discusses them, providing code examples to better illustrate and to ease the development of any future work based on VANESS. The implementation and deployment details of VANESS' main architectural components are presented – Server, mobile and on-board unit.

In Chapter 5, applications used as proofs of concepts of VANESS usage are described enhancing the VANESS features. In this section the tests performed on VANESS are described and results are discussed.

Chapter 6 presents the final conclusions on VANESS and discusses areas for future work.

2 State of the Art

In this chapter we will describe the state of the art of vehicular networks (VANETs) together with some of the common VANETs uses cases. Next we focus on systems where mobile devices are used on traditional VANET scenarios. Given the focus of this work, we also provide an overview on naming services and main concepts that will support some of the VANESS' design options.

2.1 Vehicular Networks and Applications

A network is said to be ad-hoc if the collections of nodes that form it are dynamic, without any existent infrastructure, changing its topology dependent of devices in range. On these networks devices are free to join and all nodes on the network have equal status.

Vehicular networks, also known as VANETs, are a good example of ad-hoc network formed by vehicles and road-side units. Nowadays they've been attracting big interest both from researchers as from the automotive industry.

VANETs are a subset of Mobile Ad Hoc Networks – MANET – and form a big important part of Intelligent Transportation Systems – ITS – which aim to build a truly integrated transport system, making road transport cleaner, more environmentally friendly, more efficient and safer. [16]

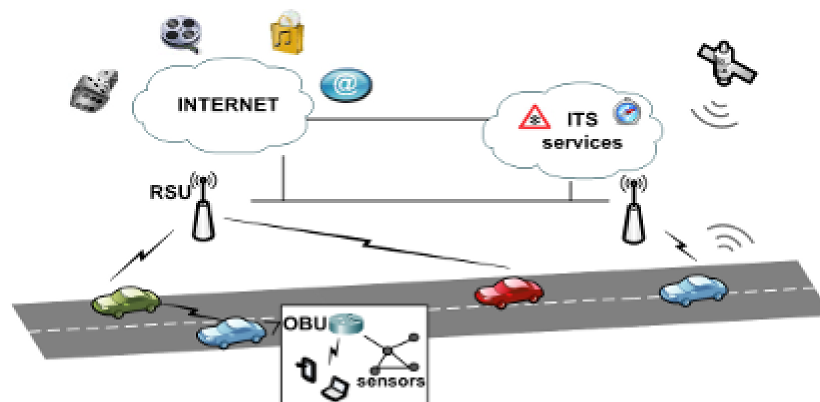


Figure 1 - VANET typical architecture [17]: cars exchange messages between them through the OBU interface. To communicate with internet resources the VANET uses RSUs. ITS services can be either available using the internet or the directly through the VANET

As shown in Figure 1 this kind of network is more than just a set of cars connected. It can bring access to our common files, entertainment/infotainment features when connected to the internet, connect to our personal devices – e.g. mobile phones, laptops – and with a set of sensors, between others.

There are plenty of use cases for this kinds of networks, as we'll see later in this chapter, but they also have distinctive characteristics [18]:

- **Need support for high number of nodes -** In the future is expected that most cars will be able to join vehicular networks. Due to its ad-hoc nature and the big communication range VANETs have to be very scalable, behaving well both with few nodes and with a big network.
- **High mobility and frequent topology changes -** Since every node is moving independently, the topology of the network is constantly changing. The available time to exchange a message can be rather small, for instance when vehicles on the network are moving at high speed on opposite directions.
- **High quality requirements**. Since some applications deal with traffic safety, even with life-safety, this networks need to have strict requirements about delivery delays or reachability
- **Privacy -** Communications to the network about the driver/user might disclosure private information. As for any public network, security of communications should be a constant focus on vehicular networks.

Applications for these networks are usually classified in [11]:

- **Active road safety applications -**Safety applications process one or more data sources from one or more vehicles (like his position, speed and heading) aiming to predict collisions and assist the driver to avoid them, warning them when violating road signals and about traffic conditions/hazards.
- **Traffic efficiency and management applications -** Focuses on assist and improve traffic flow usually by analyzing the state of the remaining vehicles nearby.
- **Infotainment applications -** Entertainment applications based on a media or social components that can be obtained locally or from WWW when possible.

The typical VANET – Vehicular Ad-hoc Network- topology involves a set of on-board units (OBU) and road-site units (RSU). OBUs - Figure 2- are placed on vehicles enabling them to communicate with other vehicles/OBUs in an ad-hoc formation using IEEE 802.11p standard (WAVE) [19] and integrating other devices like smartphones through common Wi-Fi technology. RSUs are placed in fixed points near the road with the point to increase networks reachability - doing the broadcast of received messages - and may also serve as outside gateways - Figure 1.

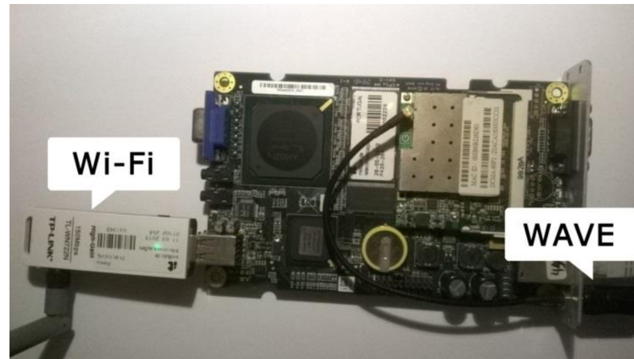


Figure 2 - One of the OBUs used. It provides Wi-Fi connection so common devices can connect to it and WAVE which is used to communicate with other OBUs/RSUs

These on-board units are plug-and-play devices and compatible with any vehicle, transforming them to a VANET node. In a near future, it is expected [20] that most cars will be capable of participating in these networks

A strong sign of that is the US United States Department of Transportation pushing for using V2V (vehicle-to-vehicle communication) technology in cars and light trucks [20]. They expect this change to reduce accidents and consumptions.

Also, working with the Michigan Department of Transportation, University of Michigan researchers have designed a unique test facility for evaluating the capabilities of connected and automated vehicles and systems – Michigan Mobility Transformation Facility [21]. It has multiple road and road-side types, a variety of signs, traffic control devices, light conditions and event buildings. It simulates the complexities of an urban environment where work on connected and automated mobility systems can be securely tested.

Together with several fleet companies, our group has built and deployed a vehicular mesh network with more than 600 connected vehicles [22] [23] [24], currently in operation in Porto, Portugal. This platform serves as testbed and proof-of-concept for several groups' to test their vehicular networking solutions.

Figure 3 shows the architecture of our system. Each vehicle has an on-board unit to communicate with other vehicles or road side units. This OBU may have internet connection, serving as a gateway to the internet: it is then possible to connect mobile phones to on-board units using Wi-Fi connection provided by the OBU.

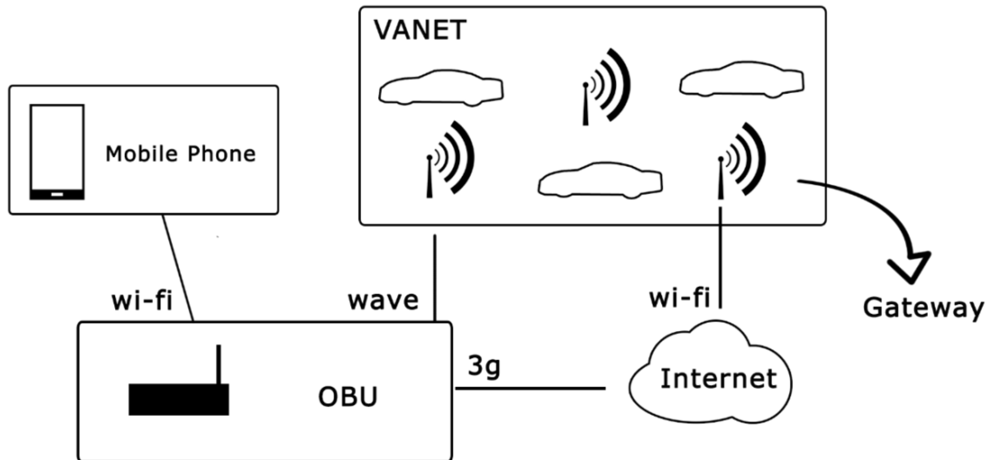


Figure 3 - current architecture of VANET-Mobile interaction: VANET is formed by vehicles, on-board units and road-ride units. These units may have internet connection, serving as gateway to the internet. Each mobile phone connects via Wi-Fi to one on-board unit which relays messages between the VANET and the phone and/or the internet.

2.2 Mobile device use on VANETs

Currently, the use of mobile devices while driving is considered a bad and irresponsible act, distracting the driver and causing traffic crashes [25, 26]. With the upcoming of automotive specific operating systems, applications designed especially for driving scenarios and self-driving cars the use of applications on vehicles and the integration with common mobile devices is trying to solve this problem.

This year an enormous breakthrough in the use of mobile apps on cars happened when Apple, Google and Microsoft decided to bring their mobile operating systems to the car. Apple launched CarPlay [6] which connects to an iOS device. Apple says "CarPlay takes the things you want to do with your iPhone while driving and puts them right on your car's built-in display" [6]. Google introduced the Android Auto [7], as part of Open Automotive Alliance [8], to make the use of mobile applications safer while driving, but it also aims to provide a fully functional OS on the car which will work even without being connected to any smartphone. The same goes for Microsoft whom also has Windows Embedded Automotive [9]. Google announced a self-driving car [10] still in prototype.

The use of infotainment application on cars is already a reality [27], and due to the big diversity of devices existing with possibility to connect to our vehicle, a standard was created called MOST - Media Oriented Systems Transport [28] This standard is supported by most car manufacturers worldwide allowing application to transmit real-time audio, video, data and control information between any devices attached, creating a network with all the connected devices, additionally allowing each device to get information about the services provided by other devices on that network. It also provides a framework to ease the development of multimedia

sharing/management applications. In MOST scenarios a network is one or more device connected to a single car. It does not offer solutions to connected cars nor share information between them.

Magneti Marelli in cooperation with Politecnico di Torino developed [29] an in-vehicle open infotainment system based on Android OS. Their motivation was that Tier-one manufactures (companies who are direct suppliers to carmakers) need to deal with different requirements from multiple carmakers, sometimes even each car model has a different version of the Infotainment system. They also point that users are used to breathtaking features on smartphones and they expect similar in their vehicles. They propose a generic open platform which could be used by all/any carmaker while allowing third-party applications. The vehicles functions may be used by this application but only by trusted applications. They hope that this system will allow products to grow and adapt to the user preferences while reducing costs.

Spelta et al. [30] created a system that connects a smartphone to a motorcycle using Bluetooth, designed to increase safety. This system creates a new interface between driver and motorcycle mainly via speech recognition. In their implementation the driver could: ask for the state of the vehicle, e.g. speed or temperature; spell certain commands understood by the system, e.g. turn off lights; be notified on his helmet when something irregular happened, e.g. motor problems or low gas. This system has also access to online servers, so it’s possible to access data services - The information about the vehicle can be periodically sent to a server which processes it in multiple ways, then returning useful information - and to perform data analysis. - It can be used as an infotainment system, retrieving, for instance, information about the near points-of-interest.

Hernandez et al. [31] created an automotive embedded system which allows communication from the mobile device to the car and to the VANET in order to use ITS. They prototyped an OBU that connected to the OBD, so it can collect data related to the vehicle. Devices connect to the OBU using wireless communications to send or receive information from the VANET and the vehicle. As proof of concept they developed two services: one for economizing driving and other for traffic reports. The system contains a touchscreen as the interface between the driver and the on-board unit but it is mentioned the use of common mobile devices as the interface. However, details on how they do it (or aim to do it) are not mentioned on the paper [31].

Diewald et al. [12] introduced *DriveAssist*, an application which shows traffic information originated from the vehicle network and traffic centrals on the smartphone. It uses vehicle-to-X communication (i.e communication vehicle-to-vehicle as well as vehicle-to-infrastructure). Besides showing information it also triggers visual warnings for certain incidents. They consider mobile devices like smartphones an alternative to ITS consumers, especially in mid-sized or compact cars, which have no or limited digital screens.

Instant Mobility [32] has developed functional and non-functional requirements for vehicular applications, grouped in three main scenarios: personal travel companion, smart city logistics and

transport infrastructure as a service. First it defines this requirements generally for all scenarios and then specifics to each scenario. Requirements defined should serve as specifications for future applications. It considers requirements regarding Reliability, performance, security, private, etc.

Filipe Oliveira et al. studied, created and tested a solution named REINVENT [15] for the integration of mobile applications and vehicular networks. Their system allows developers to create applications which communicate between them using vehicular networks without dealing with the transport and networks layers. Using this system, a text message is sent to a specific vehicle passing the text message and the vehicle identifier. The system running both on mobile phones and on on-board units takes care of composing message, sending them and processing them on the other nodes.

Currently the integration of mobile devices on road scenarios is a big area of interest. In this section we presented part of the many works on the field. Overall, researchers have worked to connect devices to vehicles [28, 30], bring mobile applications to road scenario [29] and to allow the integration of mobile devices with vehicular networks [12, 15, 31]. However, in all of these solutions the communication is done vehicle-to-vehicle, meaning that application with the need for communications between users are limited to scenarios with internet access. The solution for this problem is the subject of this dissertation.

2.3 Naming Services

Although many advances have been made on vehicular networks field, there is still a big discrepancy between the communication layer services provided by it and the user level services namely those accessible through mobile applications on other networks and technologies. Users and developers are accustomed to user-to-user or user-to-business communication without explicit concerns related with communication transport layer available. Common technologies and protocols like naming services, can be useful to solve this cases.

DNS (Domain Name System) [34, 35] is a common solution in other domains. DNS mainly consists of a system that translates from logical names/alias into transport layer addresses. However in VANETs context, naming services need to adapt to the dynamic nature of the ad-hoc networks and its moving nodes (i.e. vehicles) make the data invalid very quickly, faster than traditional DNS were designed for.

K. Mershad and H. Artail "designed a system that enables vehicles in a VANET to search for mobile cloud servers that are moving nearby and discover their services and resources" [13]. Because some vehicles have more resources than others, for instance network access, storage or specific files, they can become clouds servers sharing those resources on demand with others on the same network. In their solution they used RSUs as the intermediary for the registration, dissemination and query of services. When registering to be a server the vehicle should send a

message to the RSU with parameters about the service itself, QoS, bandwidth, service costs, between others. This system works at network level, creating associations between vehicles and his shared resources. Migrating this system to a user discovery service would mean that RSUs would have to deal with many user entries in crowded areas, possibly would have implement a way to detect when that user becomes unreachable.

A. Lakas et al. proposed [14] a service discovery scheme for VANETs where a query is a broadcast message where the solicited service is specified. A routing protocol is used to re-broadcast it until it reaches a vehicle who knows a node providing that service or the service provider itself. The response is forwarded back following the reverse path. This requires each node to store in a routing table, which brings some complexity to OBUs. In a user context, this solution is similar to the previous, moving the complexity to the OBU but this one guarantees that, if the requested user (the one we are looking for) is reachable, then a response will be received.

MulticastDNS [36] is a zero-configuration network protocol to translate hostnames to IP addresses in a non-centralized manner. In this protocol when a query to a hostname is multicasted the machine with that hostname responds with his IP address to the network, allowing other nodes to update their cache. MulticastDNS itself runs over Ethernet networks using the same interface and packet formats than the usual WWW DNS, it's not prepared to vehicular networks. If transposed to user discovery on vehicular networks context this system would behave well both in resource use, since it only stores cached values as for results viability, since it always responds when the requested user is reachable.

2.4 Conclusions

In this chapter we have addressed several subjects: we presented VANETs and described solutions that aim at improving communication between a) vehicles; b) between vehicles and the infrastructure; and also c) between vehicles and personal mobile devices.

We also addressed the naming services used for translation [35, 34, 36] and discovery services [13, 14] both related with vehicular networks or with other technologies, some of them use a centralized approach, others are distributed.

From our review, it was clear that research using mobile devices integrated on vehicular networks has been increasing, but these use mobile devices only to consume data from OBU and as a human interaction tool for the system itself. The exception is REINVENT which allows mobile applications to send messages between them; however messages are exchanged between vehicles obligating final applications to know the ID of the destination vehicle.

From our state of the art survey, and to the best of our knowledge, there is no work that aims to abstract VANETs communication resources and topology to mobile application i.e. allowing users on different vehicles to send messages between them, without knowing à priori any network details nor in which vehicle the other user is in. Providing such solution is the focus of our work.

3 VANESS: DNS for nomadic users in vehicular networks

In this chapter we will describe VANESS – VANET DNS – a naming service to support user-to-user communication in heterogeneous scenario comprising both VANET and traditional web. VANESS' objective is to ease the use of vehicular networks' communication resources on mobile applications working as a naming service resource transparently to developers, making the exchange of messages between users immediate to any software running on a mobile device. It will handle the translations/mapping between the user alias/username and the transport level needed information e.g. vehicle IDs. VANESS was conceived initially to be used in a VANET scenario having mobile device integration in mind, where users from different vehicles want to interact through mobile applications. VANESS concept is flexible and can be implemented in other mobile devices other than smartphones, namely wearable technology, smartwatch, Google Glass, etc.

We opted to integrate our system with previous work from the team, called REINVENT [15] – a mobile to VANET access solution that allows decoupling the logical message exchange in mobile application from transport/communication details. Using REINVENT messages the destination is identified by the ID of the on-board unit, at the transport layer. However is common for mobile application to handle users instead of vehicles, having the need for communication user-to-user. We extended REINVENT to make this possible and called the overall system VANESS.

3.1 REINVENT architecture

REINVENT [15] is a system to decouple the vehicular network specificities for mobile communications, allowing applications to send and receive messages to/from the vehicular network. REINVENT allows developers to create applications which communicate between them using vehicular networks without dealing with the transport and network layers, namely in V2V communications.

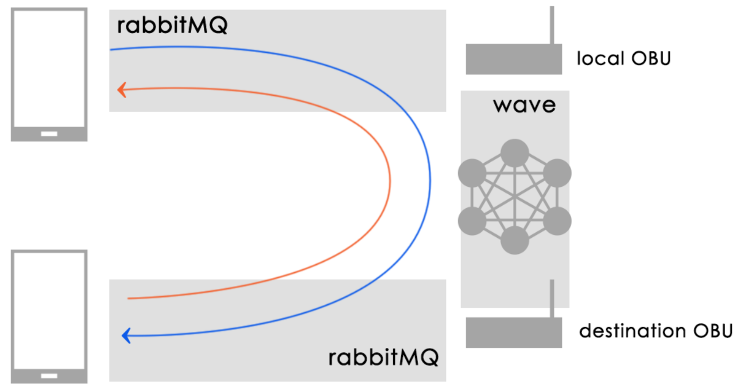


Figure 4 - The progress of a message on the system, between phones: communication between each phone and the OBU uses rabbitMQ and between OBUs (via VANET) uses wave protocol.

REINVENT has two components, one running on the mobile phone and one at the on-board unit (OBU). – Figure 2.

The phone component is a centralized point of access for any applications to interact with. It receives the parameters needed to compose a message, creates it and sends it for the module running on the local OBU (as depicted on Figure 4). This is done via a message middleware named RabbitMQ which allows the exchange of messages between different independent devices based on message queues. RabbitMQ basic architecture consists of a server which stores and handles the queues and one or more clients which send and listen for messages on the server's queues.

In REINVENTs architecture, the RabbitMQ server runs on the OBU and the client on the phone. The OBU component works like a proxy between the phone and the VANET: It listens for new messages sent by the phone and sends them to the vehicular network. It also listens for incoming messages on the vehicular network and sends them to the mobile phone (using RabbitMQ's queues).

3.2 VANESS architecture

VANESS is meant to act as a naming service to support user-to-user communication within a heterogeneous scenario comprising typical ISP scenario and VANETs focused on mobile devices.

Its architecture reflects the heterogeneous nature of the communication infrastructure since our system is able to map the user to an end point either locally (i.e. there's not internet connection at all), online (i.e. system is not in a vehicular network but has direct internet connection) and using a gateway (i.e. the system is in a vehicular network where some of the nodes has internet access and will act as a gateway).

VANESS deploys over the typical ITS architecture and has three main components:

- on the mobile phone to whom applications will interact with for communication and authentication purposes.
- on the on-board unit allowing messages to be exchanged between connected devices and the VANET and between multiple vehicles; and
- on the web, where a centralized approach is used both for data management and allow two distinct vehicular networks to communicate when each has an internet access.

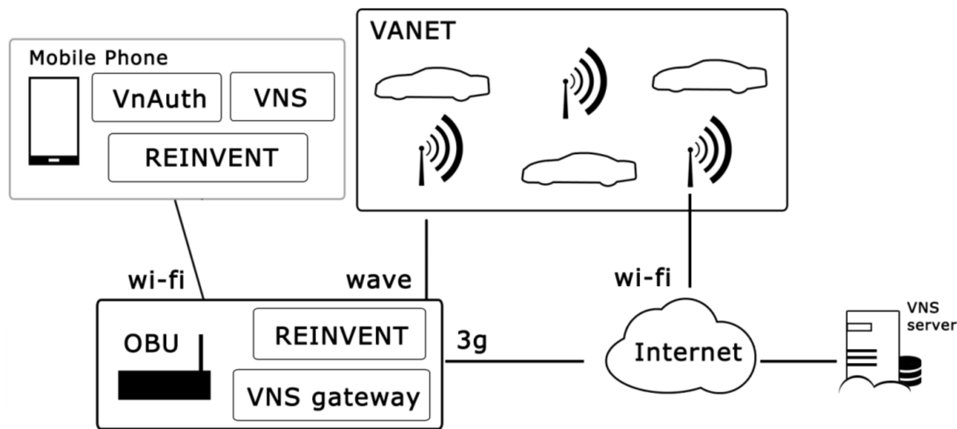


Figure 5 - VANESS architecture: The phone has modules to manage accounts, communications and the naming service; The OBU has proxy modules for communication between the phone and the vehicular network (REINVENT) and the phone or vehicular network to the internet (gateway)

As a system, VANESS extends REINVENTs features by adding two modules: **VNS** (VANET Naming Service) and **VnAuth** (VANET Authentication) as seen on Figure 5. In VANESS, REINVENT is used to 1) assessing OBU data, like his ID and the connectivity status (direct access to internet via 3G, VANET connection, etc) and 2) use WAVE resources to send and receive messages. Now, when any phone's application asks REINVENT to send a message to a certain user, REINVENT requests VNS to translate the receiver username/alias into a valid destination address (vehicle ID). If the application identifies the message addressee as a vehicle, VNS step is skipped, since no translation is required, otherwise – addressee is a username/alias – VNS will find the identification of the vehicle that will make the message arrive to that person.

VNS module keeps track of the users' alias and associated transport layer addresses – vehicle ID. When sending a message, VNS will return the current address (i.e. last known) of the receiver and is prepared to cope with any identifier which can be represented as text, for example standard IP, Google's registration ID [37] or VANET vehicle IDs. VNS module has two main components: at the mobile and on the server. It also makes use of a gateway so the web server be aware of changes happening on VANETs. The server is located on the internet and is only accessed internally by the VNS's mobile component. This server aims to have global associations

between users and vehicles as updated as possible (when none of the vehicular network nodes have internet access some server entries can become outdated).

VnAuth module provides a sign-up and sign-in process integrated in the mobile OS that provides a common identity throughout all the apps on the phone. All the VANETs authentication process will be performed automatically and made available as mobile phone resource, through VnAuth**.** Naturally any sign-in process will be communicated to VNS to associate the current user to his current vehicle's identifier.

3.3 VNS: VANET Naming Service

Users want to interact with other users, not with cars. And they do that mainly with their smartphones. A hurdle to the common vehicle-to-vehicle communication is the act of users exchanging vehicles, being either personal cars, bus, taxis, etc. This means that REINVENT does not know to which vehicle the message must be sent when the destination is a specific user. To surpass these handicaps, VANET Naming Service (VNS) was created as a new module on the system that is able to know what vehicle to which a message should be sent, so that it arrives to his destination.

VNS is internal to VANESS and transparent to any user or developer who is creating applications to run on vehicular networks. From the application/developer point of view (Figure 6), the interaction is completely done with REINVENT, which in turn will interact with VNS, as depicted in Figure 67.

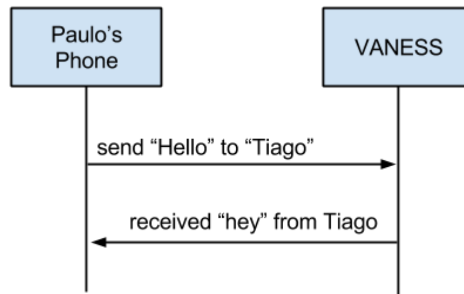


Figure 6 – The mobile applications use alias/usernames to refer to message receivers/senders. It is VANESS' task to ensure that the messages are received by the correct persons, regardless of their vehicle's ID

This apparent simplicity hides, intentionally, the complexity existent on the communication process (e.g. translation and validation process) that is done internally involving several different interactions and modules (depicted in Figure 7).

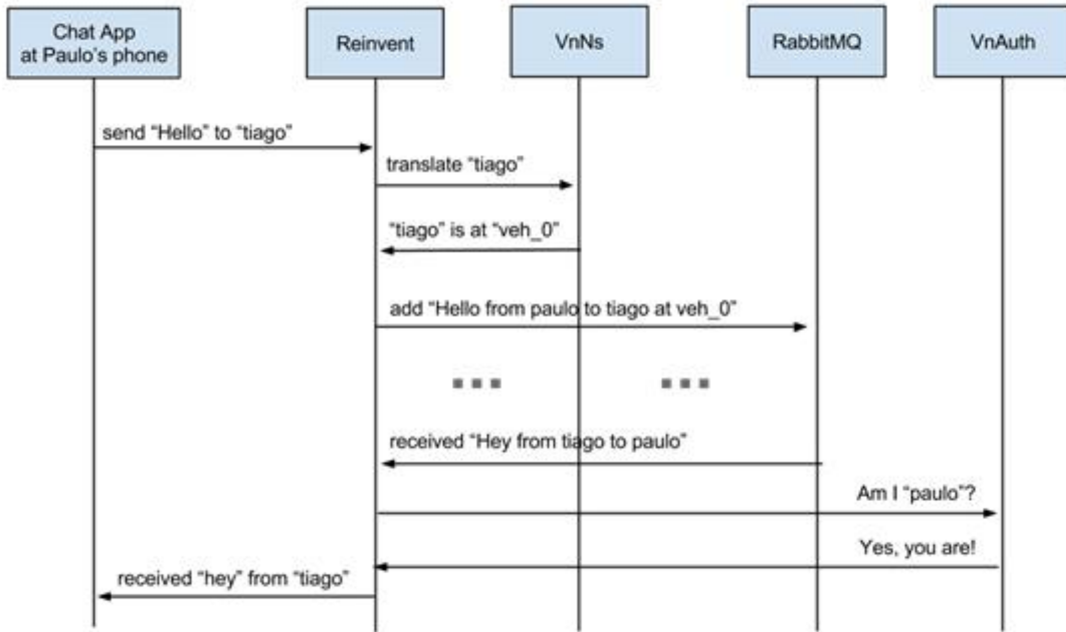


Figure 7 – Interaction between the system's modules when user "Paulo" sends a text message to "Tiago" and he responds back

As seen on Figure 7, REINVENT is the communication endpoint. For each message to a specific user, it sends a request for VNS module with the destination username/alias and receives, as response, the identification of the OBU/vehicle to contact. Then, REINVENT forms the message with a specific, predefined format and sends it to the OBU which forwards it through the vehicular network to reach the destination (e.g. vehicle) where it is pushed to the receiving end application. Although applications can keep the user-vehicle association in cache, we opted to make each individual endpoint the only trustworthy entities (besides the server) responsible for announcing/providing for their own user-vehicle association. If that was not the case, there was the danger of flooding the network with responses from several nodes to a single request on the network – this simple option is more scalable in situation where the number of nodes rises to hundreds or even thousands on the same VANET.

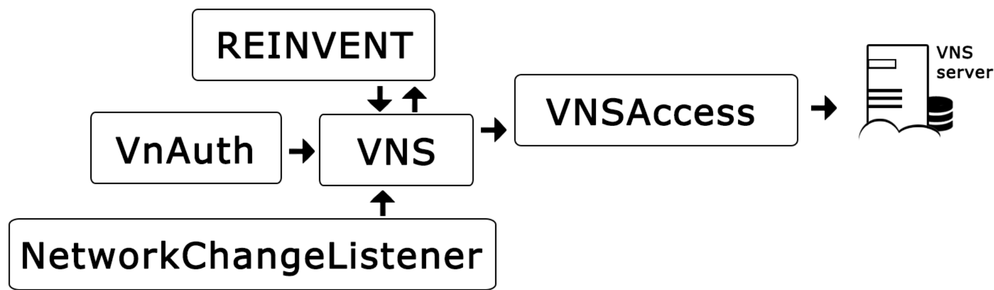


Figure 8 – Basic representation of the connections of the systems' parts: REINVENT uses VNS to ask for the username translation; VnAuth reports when any user logs in to VNS; NetworkChangeListener warns VNS each time mobile phone connects to a different OBU (different vehicle ID). In turn, VNS uses REINVENT to send messages through the vehicular network and uses VNSAccess to make server calls.

On the mobile phone, VNS provides a REST interface accessed in a locally centralized (on the phone) way so it is a common and shared resource to all apps of that device.

On the web, VNS provides a server as the globally centralized resource of VNS. Since multiple ad-hoc networks exists over geographic space, this server aims to maintain a broad record of registers independently of those networks specificities, also allowing communication between different VANETs if each one of them has at least one gateway access. The VNS server is located on the internet and accessed internally (by VNS module) via REST web services. The server keeps track of the global associations between users and vehicles. To keep this information updated in VANESS, VNS module on phones sends the available data to the server whenever possible, where it will be merged with the existing one.

VNSAccess is the abstraction layer we created to interact to VNS server. It is used when the mobile phone has direct internet connection providing the abstraction from the any webservice API details, allowing the services to be called without API implementation details, like any usual java method.

Figure 8 represents the connections of the systems' parts. REINVENT uses VNS when a message is to be sent for a certain username, asking for the translation, while VNS uses REINVENT to send messages to the network (broadcast messages or to specific vehicles). VnAuth reports to VNS when the logged user changes and NetworkChangeListener when the mobile phone connects to a different OBU (different vehicle). Finally, VNSAccess is the access layer from the mobile phone to the VNS server.

3.3.1 VNS Protocol

Each time any application sends a message to a certain user, REINVENT queries VNS for a translation i.e. VNS sends a *DNS_TRANSLATE* message for the vehicular network to be broadcasted throughout the network and received by mobile phones on other nodes. Each phone, when receiving a *DNS_TRANSLATE* from the network, assesses the username to be translated

comparatively with the logged username, obtained from VnAuth. If they are the same VNS responds with the identification of the current connected OBU using a DNS message to the VANET - Figure 10. This message has a *timestamp* field used to distinguish outdated entries form new ones. Since, when the user is responding himself this OBU ID is always the most recent, is used the current timestamp in this situations.

Each time a user logs-in or the device connects to a different OBU, VNS module is notified, creating a new association between the user and the vehicle. This way VNS has the information as updated as possible, bearing in mind that the "disconnecting from OBU" is an action we can't detect. In situations where some user disconnects from an OBU and doesn't have any connection (neither VANET nor internet) the system may have him (i.e. the user) associated with a vehicle where he is not at. This is a real issue but, considering that in this situation he is unreachable we did not consider it a severe one because this message wouldn't be delivered anyway. In the next time he connects, a new association is made, becoming reachable again.

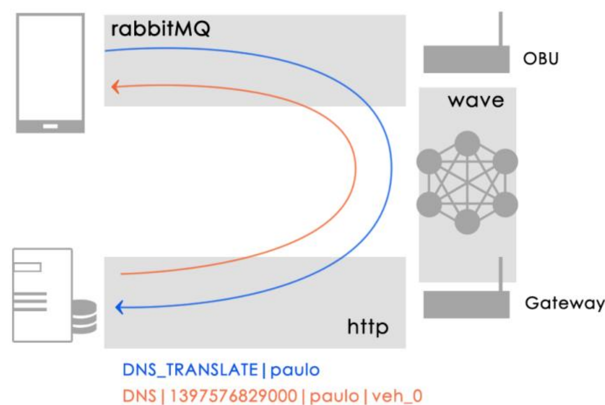


Figure 9 - The progress of DNS messages thought VANET with gateway: When the phone sends a message it is initially sent to the local OBU, using REINVENT which in turn uses RabbitMQ. The local OBU will broadcast that message to the vehicular network. If it arrives to a Gateway it convert it to HTTP and calls the server.

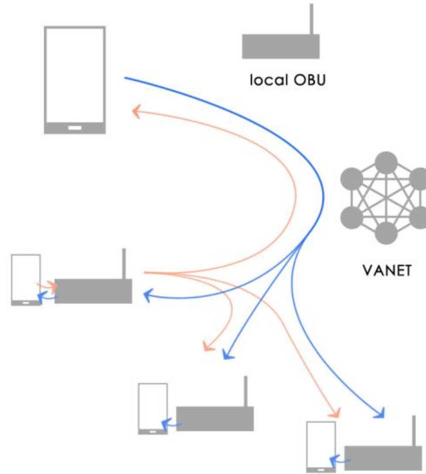


Figure 10 - DNS_TRANSLATE (blue) broadcasted and DNS (orange) response from the addressee. Notice that only the user present on the request message responds. All the nodes on the network may update their cache with the response.

VNS will communicate the changes to the server in two ways: 1) when the mobile phone is directly connected to the internet it registers the user-vehicle association using the available *webservices*; or 2) when the phone is connected to the OBU it sends a *DNS* message to the VANET (either as a response to *DNS_TRANSLATE* or because VnAuth or a given application explicitly sends it). Either way, when it arrives to a gateway, it is converted to a *webservice* call to the VNS server – Figure 9. Notice that the same message type used for responses is also used to report explicitly changes detected by the phone, namely: 1) the logged user has changed; or 2) phone has connected to a different access point (OBU or internet)

3.3.2 VNS messages

In the design of the VNS message an effort was placed in ensuring that only essential fields were included on each message: the VNS request has the message type *DNS_TRANSLATE* and contains only the username to whom we want to know his vehicle identification; the VNS response has the message type *DNS* and contains the timestamp, so the client be able to solve conflicts if it receives more than one answer.

Request:	DNS_TRANSLATE|*<username>*
Response:	DNS|*<timestamp>*|*<username>*|*<vehicleID>*

Table 1 - VNS message formats

To make possible the communication from inside the vehicular network to the server we created a gateway module which converts *DNS_TRANLATE* messages to HTTP requests to our server's API and the response back to *DNS* messages.

A few changes had to be made on REINVENT to make the new system work. We extended REINVENT to support VNS message types and listen for changes on the correspondent queues. When a *DNS_TRANSLATE* message is received, instead of notifying other applications, like it does with other message types, it responds directly when appropriated (as defined by the protocol). When other applications ask REINVENT to send a new message, if they specify the destination vehicle, then REINVENT behaves as before, but now the application can also specify the destination user. We adapted it to rely on VNS to make the translation from username to destination vehicle.

Each time someone logs-in at any application using VnAuth, it registers the new association between user and the current OBU using VNS. Whenever the device connects to a different OBU, the module NetworkChangeListener detects it and registers the logged user with the new OBU ID. NetworkChangeListener is a listener for Wi-Fi state change. Each time a change occurs, it uses the phone's connectivity manager together with the stored information about his previous state to assess if this user should be re-registered. If yes, VNS is called.

3.3.3 Server Synchronization

As mentioned before, the VNS module on the mobile phone communicates login and OBU changes to: 1) other nodes on the VANET via DNS messages; 2) directly to the server when internet is directly accessible; 3) indirectly to the server, when a gateway receives a DNS message.

When a VNS request (*DNS_TRANSLATE* message) is sent in a network with gateway access it asks for a translation on the server and broadcasts the answer (*DNS* message) on the network. Due to the frequent topology changes it's possible to get an outdated response from the server – if, when the login/connection happened, there was not gateway access. To help the server having the most updated information possible, each time a user receives a VNS response for its own name but with wrong information it sends another VNS response with the correct information. This response will hopefully arrive to a gateway and consequently the server, updating the association with the right information.

3.3.4 Cache Management

REINVENT's message exchange does not include an acknowledge response, so it is difficult to establish if messages are being correctly delivered or not and consequently, difficult to establish if cached associations between users and vehicles are outdated.

In VNS we opted not to rely on cached association user-vehicle by default, so when sending a message, VNS always asks for the real address of a given alias/username. However, since different applications have different needs, VNS can be configured to use cached addresses for existing alias in cache. Doing this will make the process of sending a message faster and lighter since VNS messages are less frequent. Cache management has three modes of operation, which can be passed as a parameter (optional) to REINVENT when sending each message:

- Cache OFF (default): each message sent VNS process happens. A request is sent and it waits for a response. It takes time but is more reliable.
- Cache ALWAYS: if there is a user-vehicle association in cache, it is always used. VNS process only occurs if no association exits in cache yet. It's the lighter but also the less reliable, since it assumes a user is in a fixed vehicle.
- Cache AGED: it is a midterm of the above. It uses the cached value if the age of the local entry is recent – less than a pre-defined timeout – otherwise, it re-asks for a translation.

3.4 VnAuth | Vehicular Network Authentication

For VANESS to work properly, it needs to access the username and vehicle identifier each time the user logs in or changes vehicle. VnAuth module was inspired on Social Sign-in model and was created to be a device-centralized login and signup method which also handles the communication with other modules of the system.

3.4.1 Why VnAuth?

With the advent of social sign-in people became used to login/register one time per device and have their account synchronized throughout the device. This turns to be a key factor for application that have to be used on driving scenarios, since the user has to keep his attention on the road. Making the user enters his credentials while driving will be dangerous, so a "touch to log-in" button or automatic log-in is a key feature.

Social sign in is closely related with the single sign-on concept. Single sign-on (SSO) is a concept related to authentication which makes possible control access in multiple independent systems. With this a user logs in once in one machine and gains access in all other (pre-defined)

machines. This is becoming a concept common inside enterprises because it reduces time and fatigue spent on login and password recovering.

Social sign-in concept is very similar and consists on Social applications allowing third party applications to login/register they users with their account management system. Many web and mobile applications already allow users to use login from one or more known social networks usually by adding a simple buttons for each one. When the user clicks a button the authentication process for that service is used, usually if the user has already log in in that device he is automatically validated, otherwise the enters his credentials.

Following a similar approach, VnAuth brings flexibility to authentication on vehicular networks. It provides a device-centralized login and signup method which can be used by any phone's application that want to communicate through the vehicular network. With a global signing process as VnAuth developers don't need to worry about these details.

A user control like this allows the apps to be developed without any knowledge about the specific implementation on user management nor the naming system. Besides that, any change on the process it's guaranteed to be unnoticed to all the already implemented apps, for instance: At this (initial) stage the user only has to choose his username. If now we decided to request also a password, only the VNAuth module would need to be updated internally, all the apps remain exactly the same.

3.4.2 VnAuth Protocol

The interaction of this module with the rest of the VANESS system is fairly simple. Each time the user clicks the log-in button whatever the application is, if the login process is successful, it makes a call to VNS register method. The registry is then completely handled by VNS.

It also interacts with the system's account manager, so it makes that information sharable to all local applications.

3.5 Summary

This chapter presented the system's concept. VANESS was created due to the limitation of communication on vehicular networks where a message is addressed to a vehicle, whereas most mobile applications require user-to-user communication. We aimed to give developers an ease to use interface for message exchange between users while keeping support for vehicle to vehicle exchanges.

VANESS is an extension of REINVENT [15], an abstraction of the communication process on vehicular networks. It is composed by two main modules: VnAuth, mainly responsible for the association of the user to the device, and VNS, mainly responsible for handling requests, responses and cache management.

Besides the ad-hoc approach, VANESS has also a centralized component – VNS Server – which aims to have information on multiple independent VANETs. For that, a gateway was created which converts messages from vehicular networks to webservices access on the server.

4 VANESS Implementation

In this chapter we describe the VANESS implementation details, namely its final architecture and individual module details. In the process we introduce some relevant concepts (RabbitMQ message broker, BroadcastReceivers and ContentProviders) that are cornerstones on both VANESS and REINVENT.

In the end we present some code samples on how to use/access the VANESS and REINVENT services and resources.

4.1 Relevant concepts

VANESS implementation relies on several technical solutions that are described in this section with enough detail to support VANESS implementation decisions.

4.1.1 RabbitMQ: The Messaging Broker

RabbitMQ is open source message-oriented middleware. It mediates communication amongst applications minimizing the mutual awareness that applications should have of each other in order to be able to exchange messages [38] and supports multiple patterns e.g. push notifications, publish/subscribe, and work queues [39].

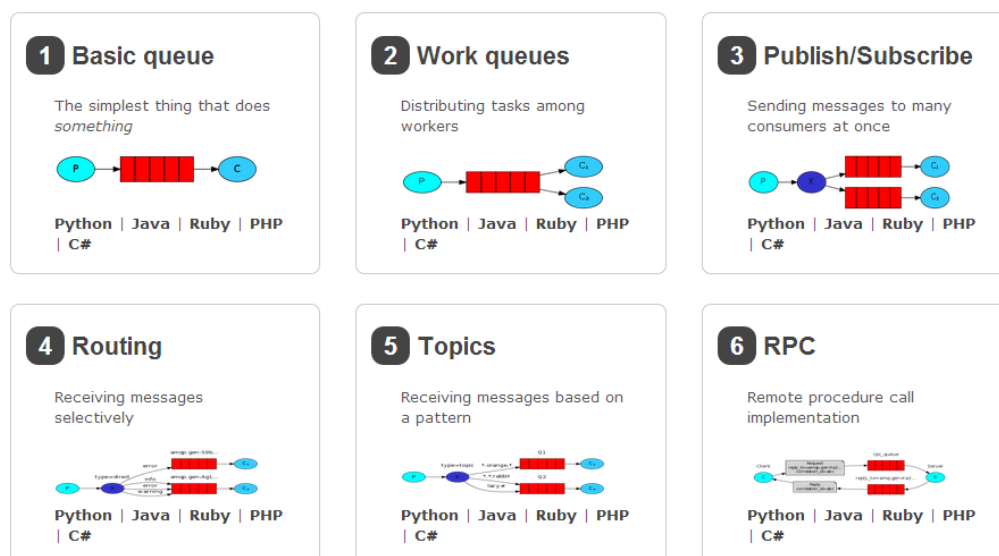


Figure 11 – Some typical RabbitMQ supported models of operation, as illustrated in [40]. Note to mention that RabittMQ provides programming interfaces for development languages.

RabbitMQ uses MQ Telemetry Transport (MQTT). It is designed for connections with remote locations where a "small code footprint" is required and/or network bandwidth is limited. It also supports many development languages which gives VANESS some extra ubiquitousity.

Figure 11 depicts some typical RabbitMQ supported models of operation. The "**work queues**" model sends messages to a queue that will be delivered to one, and only one, of the consumers chosen via round-robin dispatching by default. Usually this pattern is used to avoid doing resource-intensive tasks immediately, so messages that represent tasks are added to the queue and then some worker thread is responsible to process it.

The "**publish/subscribe**" model on the other hand delivers all the messages to all the listeners/subscribers. In this model the producer never sends any messages directly to a queue, instead sends it to an intermediary (named *exchanges*) which in turn adds those messages to one or more queues, depending on his configurations.

The "**routing**" and "**topic"** model are similar to publish/subscribe but the *exchange* can filter messages based on a *binding key* (or multiple keys, in case of the "topic" model) and choose to which(s) queue(s) to add it to.

This was a high level presentation of some of the supported models – the ones related to this thesis - the official site [39] is very descriptive about this and others patters and features supported by RabbitMQ.

4.1.2 Android's Broadcast Receivers

Android offers a standard component named Broadcast Receivers [41] which gives applications the ability to register for certain events, system wide. These events can be made available by the operating system itself e.g. battery level threshold reached, new application installed; or can be provided publicly by applications.

Broadcast Receivers can be configured statically in the manifest file, or dynamically using *registerReceiver(…)* method programmatically. When using the static way the receiver will be called whenever the event occurs, independently if the application is open or not. This is an advantage of this component over Content Observers mentioned early.

Each application can opt to also broadcast its own events, identified by a text-based name, locally – only the own application receives notifications – or globally – all applications can register and register and receive notifications.

4.1.3 Android's Account Manager

The Android Account Manager [42] centralizes accounts from multiple applications installed on the device and allows each one to authenticate with third party applications without knowing

user's credentials. It implements the OAuth authentication protocol but, unlike on the web, the device needs the third party application installed, since is that application the only with information on how to perform the authentication process itself – usually involves a call to an online server.

Some examples of applications using custom account on android are shown on Figure 12.

Besides providing this centralized and shared approach using this android's component also simplifies some details e.g. handling token's expiration time and password change on other client.

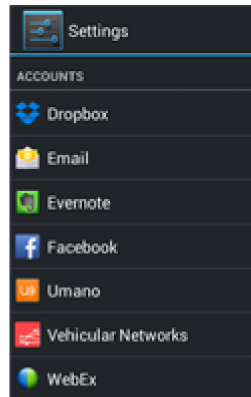


Figure 12 - Accounts section on Settings screen in an Android OS

Udi Cohen [43], leading developer at Any.do [44], explained the main building blocks on his blog [45], as follows:

AccountManager – Manages all the accounts on the device, either showing the Sign-In/Sign-up screen or providing authentication tokens to applications for each authentication method. The AccountManager knows which *AccountAuthenticator* is responsible for each method and calls the necessary action from it.

AccountAuthenticator - Handles a specific account type. The AccountAuthenticator is responsible for deciding which activity to show the user for entering his credentials and where to find the authentication token previously returned by the server.

AccountAuthenticatorActivity – This is a screen responsible to show the UI for the sign-in/create account action. The activity is in charge of the sign-in or account creation process against the server and return the correspondent authentication token to the authenticator.

4.1.4 Android's Content providers and Observers

REST is an architectural style for client-server stateless communications [15]. Due to its stateless constraint, requests should have all the needed information for the process to complete.

In REST, information is represented by URIs (Uniform Resources Identifiers) – called Resources – hiding internal implementation details.

Content providers [46] are an Android solution to provide a simple REST interface to abstract/share information between different applications. Content Providers are managed by the Android OS which, besides presenting them as Android resources, it also manages the concurrent accesses to them.

The REST like interface of Content Providers can be programmed using functions that implement the basic REST interface: insert(), update(), delete(), and query() methods. Although not mandatory, Content Providers are usually used as data sources, namely as abstract database front-ends. Content Providers also allow applications to register Content Observers, which triggers registered callbacks when change occurs on Content Providers (e.g. data changes, internal event).

4.2 VANESS implementation architecture

As previously referred, our system depends on an existent layer that provides abstraction over transport details for mobile applications, called REINVENT [15]. The main idea of REINVENT is to provide a RESTful interface for mobile applications that allow them to use VANETs communication resources transparently i.e. without handling VANET specific transport layer configurations.

Instead of creating another solution, we opted to extend REINVENT with new modules, leaving the interface almost unchanged and compatible with the original version on REINVENT. To the overall system we called it VANESS – VANET + DNS.

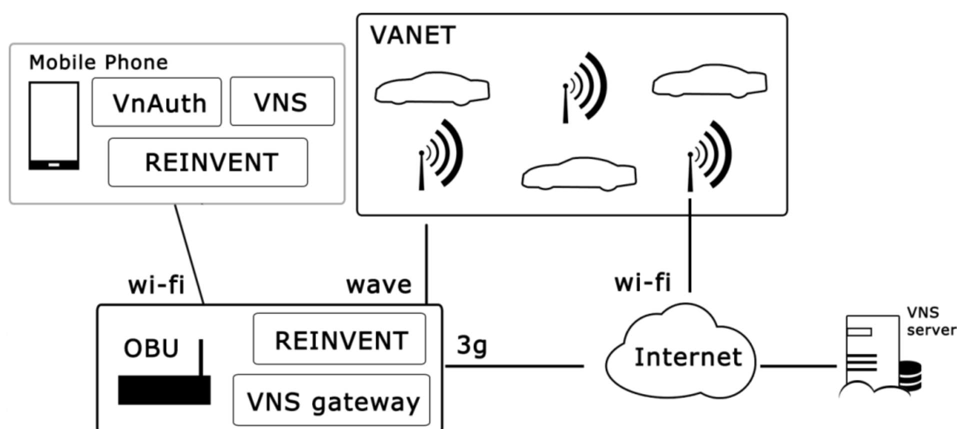


Figure 13 - VANESS architecture: The phone has modules to manage accounts, communications and the naming service; The OBU has proxy modules for communication between the phone and the vehicular network (REINVENT) and the phone or vehicular network to the internet (gateway)

As seen on Figure 13 the VnAuth and VNS modules were added to the mobile phone, alongside with REINVENT. On the OBU exists a gateway for converting VNS messages from VANET to HTTP and vice-versa. There is also a webserver on the internet with REST webservices interface.

One of the key actions of REINVENT is the exchange of messages between the phone and the OBU. This is done using *RabbitMQ*, a messaging broker.

Both REINVENT and VNS were implemented as Content Providers + Observers. VnAuth is implemented on top of Android's Account Manager; and *NetworkChangeListener* is a Broadcast Receiver, both of these are default android components. Each of these components is described in this section.

4.3 VNS Extensions to REINVENT

REINVENT relies on message passing using RabbitMQ to exchange information from the mobile device to the OBUs, where the VANET resources are.

REINVENT provides a centralized REST interface via a Content Provider in his android implementation, used both to send messages and to subscribe to a certain message type, so that the application is notified when a new messages arrives. Messages are exchanged as simple text with a specific format per type, guaranteeing interoperability between different systems and platforms.

Messages are broadcasted through the VANET using WAVE. Wireless Access in Vehicular Environments (WAVE) is a IEEE standard - IEEE 802.11p [19] – which is improved to high speed data exchange, since a connection between two vehicles or a vehicle and a road side unit may be available for only a short amount of time.

When a message is received on the destination OBU, REINVENT is listening for all incoming WAVE messages. This client assesses the message type and adds it to the corresponding queue according to it. The REINVENT module on the mobile phone will be listening for incomes on this queues and will notify and deliver them to any application who has registered for those message types.

However, REINVENT original implementation only implemented one message type that could work as text content namely for transmitting chat text messages. In VANESS we needed to extend VNS to handle new types of messages – this meant adding new queues, new message container data types, new URIs for each entry point on the API and replicating the listeners on the phone and also on the OBU. Originally, message queues were divided in two - *CarToPhone*: Messages received from the network to be delivered to applications; and *PhoneToCar:* Messages sent from applications to the OBU and should be sent over the network. Now, queues are

sectioned by their type so clients could register independently, receiving only notifications to messages of interest and each type has his *CarToPhone* and *PhoneToCar* queue.

For example, the VNS messages: Requests are queued on *PhoneToCarDNSREQUESTMSG* on the requester side and then passed throughout the networking being queued on *CarToPhoneDNSREQUESTMSG* on other nodes. Devices on this nodes will then be notified of this message and will process them, responding or not according to the protocol.

Responses are queued on *PhoneToCarDNSRESPONSEMSG* on the responder side and on *CarToPhoneDNSRESPONSEMSG* on other nodes on arrival. Again, devices will be notified and process this messages according to the protocol.

Since VNS module has to process messages even when the phone has no VANET based application open – direct request to the OBU. For example some messages types are used to get information about the OBU itself, like the ID number and the connection state and the OBU responds directly to these messages instead of being sent through the network as implemented in REINVENT. This involved a re-engineering of REINVENT's message listener, implemented as as a simple thread, to an Android service in the background for handling the message on the background, so the system don't kill it [47].

4.3.1 VNS Content Provider Interface

All the VNS's data is provided by a content provider as detailed on the next table:

Authority: `content://pt.ua.vnns.vnnsprovider`	
query	

`/local`
`- OR -`
`VNSConstants.`*`VNS_URI_LOCAL`*
	Used to retrieve information about the currently connected OBU. e.g. his IP address and ID.

`/translate`
`- OR -`
`VNSConstants.`*`VNS_URI_TRANSLATE`*
	Used to retrieve the identifier of the vehicle to which a given user is associated in that moment

insert	

`/register`
`- OR -`
`VNSConstants.`*`VNS_URI_REGISTER`*
	Indicates that a user is now reachable on a certain vehicle

Table 2 - Interface of the VNS provider

VNSConstants data type has all necessary strings to interact with this module, including the URIs, the parameters names and the returned column names. This interface is only used

internally; there is no need to final applications to use this directly, unless they want to use a custom login system, instead of VnAuth.

```
Cursor cursor = getContext().getContentResolver().query(
                         VNSConstants.VNS_URI_TRANSLATE, /* resource URI */
                         new String[]{},
                         receiverName,                   /* username/alias */
                         new String[]{},
                         "");

if(cursor != null && cursor.getCount()>0){
      cursor.moveToFirst();
      int columnIndex = cursor.getColumnIndex(
                               VNSConstants.COLS_NAME.ADDRESS.s() );
      receiver = cursor.getString(columnIndex);
}
```

Code Block 1 - VNS translate request made to the phone's VNS module

Code Block 1 exemplifies how to ask VNS to do the translation from username – *variable reveiverName* – to the end point's ID – *receiver*. It's worth noticing that any call to content providers is blocking, which means that Code Block 1 might take several seconds to finish, since it sends a request until the arrival of a response or, if no response received, a *timeout*.

On Code Block 2 we get the IP address of the OBU the phone is connected to – variable *address* - and his ID - *obuid*. The *obuid* is defined by the OBU and returned to the phone on the STATUS response message. The IP address is obtained from the phone and has two options: 1) manually defined on the VNS configuration activity – Figure 14 – 2) Automatic, where the networks' gateway address is used - Figure 15 - VNS chooses the manually option if there's any address defined, if however that field is empty the automatic approach is used.

On Figure 15 we defined IP and gateway manually since DHCP was disabled; otherwise this information would also be attributed dynamically when connecting to the OBU.

```
Cursor query = context.getContentResolver().query(
        VNSConstants.VNS_URI_LOCAL, new String[] {}, "", new String[] {}, ""
        );

if (query != null && query.getColumnCount() > 0) {
        query.moveToFirst();
        String address = query.getString(
                query.getColumnIndex(VNSConstants.COLS_NAME.ADDRESS.s())
                );

        String obuid = query.getString(
                query.getColumnIndex(VNSConstants.COLS_NAME.VEHICLE_ID.s())
                );

}
```

Code Block 2 - VNS get OBU info

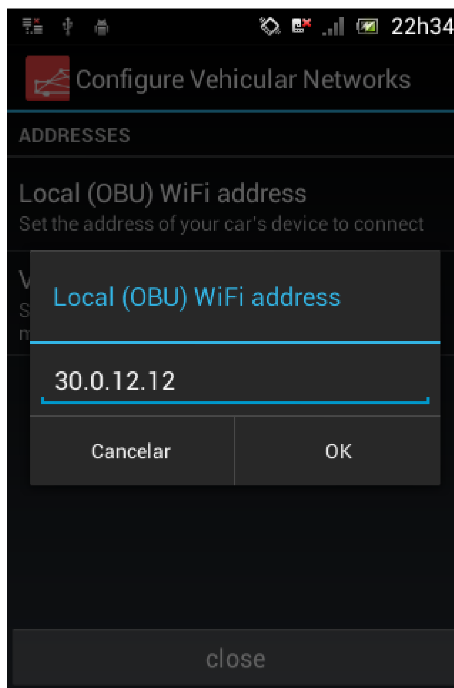
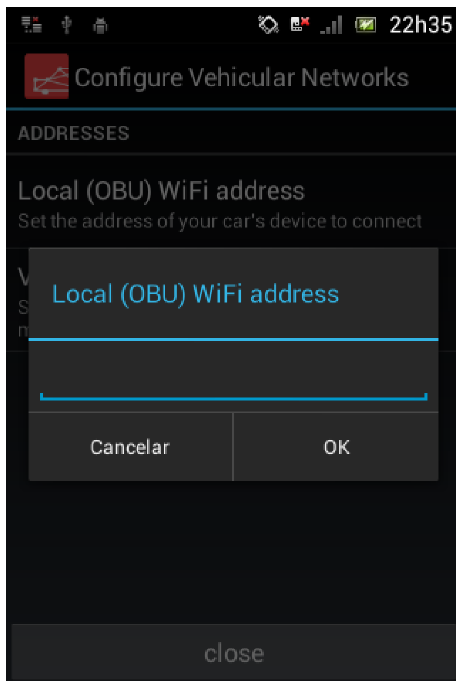


Figure 14 - OBU/Rabbit Address - Manual Mode: This configuration screen comes with the installation of the VNS module. Whenever the user defines the address in this field, it is always used;



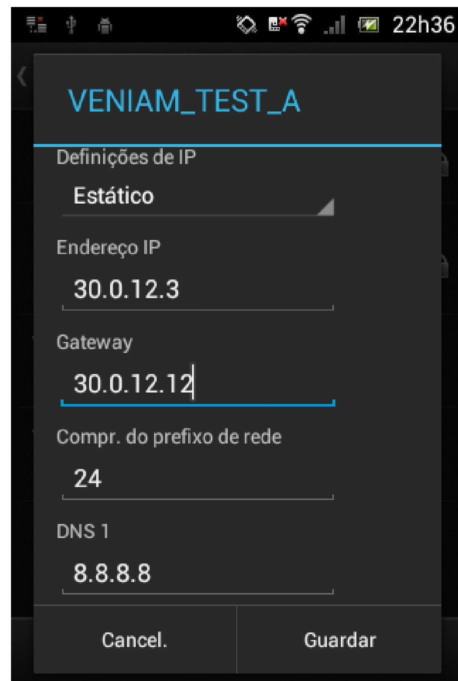


Figure 15 - OBU/Rabbit Address - Dynamic mode: If the OBU address field is left empty, the system will use the gateway's address by default. The image on the right shows a manual configuration of the network, however it equally works using DHCP

These calls are made through Android's Content Resolver [48] which is a global instance that accepts requests from clients and resolves them by directing them to the appropriated Content Provider, depending on the URI's authority. On Code Block 1 and Code Block 2 URIs are respectively `VNSConstants.VNS_URI_TRANSLATE` and `VNSConstants.VNS_URI_LOCAL`.

s( ) is a simple method to convert the *enum* constant to a String, which is accepted by *getColumnIndex.*

The previous examples made a query to VNS and then, in case some result was returned, it obtained the returned values. To avoid completely implementation details we recommend the use of *getColumnIndex* with the respective column name from *VNSContants*, so that changes in the module won't affect the interface.

Code Block 3 is used to register a new association between a user – variable *username* – and a vehicle – *obuid.* As this call is adding new information to the system, is used the *insert* method with the values to add – username and vehicle id are mandatory.

```
ContentValues cv = new ContentValues();
cv.put(VNSConstants.COLS_NAME.USERNAME.s(), username);
cv.put(VNSConstants.COLS_NAME.VEHICLE_ID.s(), obuid);

getContentResolver().insert(VNSConstants.VNS_URI_REGISTER, cv);
```

Code Block 3 - Registering a new association

The server stores a basic list of associations user-to-addresses and the timestamp when that association was made, so that when in doubt a client can find what is the most recent value. With this we can deduce the last vehicle each user is connected to, find if they lastly connected on a ISP scenario, reproduce the user's steps through time or even find statistic information about the number of users in some public transportations vehicles.

The interaction with the server is done via a REST based API, created using PHP on top of Tonic framework [49] and Zebra_Database [50]. Tonic allowed us to create flexible and extensible REST webservices with little configuration using annotations while Zebra_Database helped improving security, performance and ease of debug to database connections.

To access the server from the phone, we developed two levels of abstraction level. Firstly, *RestJsonCall* creates a generic abstraction for any REST webservices that use JSON messages. On his constructor is passed the URL of the WS. In case it has some parameters they should have a name preceded by colon, as Code Block 4 exemplifies. In this example `:name` will be replaced by `paulo` and `:addr` by `veh_1001`, meaning it will request the URL `"http://localhost/paulo/veh_1001"`;

```
final String URL_REGISTER = "http://localhost/:name/:addr";

RestJsonCall ws = new RestJsonCall(URL_REGISTER);

ws.addParameter("name", "paulo");
ws.addParameter("addr", "veh_1001");

String response = ws.call(METHOD.POST);
```

Code Block 4 - example of calling a webservice using post and parameters on the URL

The second abstraction layer mentioned is *VNAccess* which allows VNS module to access the webservices like if they were common methods, abstracting the addresses, parameters and methods of each service. So, each time VNS has to access the service he does a call as simple as Code Block 5. Both *VNAccess* and *RestJsonCall* are located on the auxiliary library.

```
String response = VNSAccess.register("paulo", "veh_1001");
```

Code Block 5 – Using the abstractions created to access VNS Server in a simple way

4.3.2 The REINVENT developer interface

Here lay some examples on how to use REINVENT to send messages for vehicular networks. To send a message is used the insert method – Code Block 6, with the parameters in a *ContentValues* object. If the message is to be sent to a user his username should be added with the key *Users.NAME*. If, otherwise, the message is to a specific vehicle instead *Users.SENDER_ID* should be added.

```
ContentValues cv = new ContentValues();
cv.put("type","txt");
cv.put("sender", AccountGeneral.getAuthenticatedAccount(context));
cv.put("_id", id);
cv.put("name", destinationUsername);
cv.put("message", msg);
cv.put(VNSConstants.COLS_NAME.CACHE_METHOD.s(), VNSConstants.CACHE_METHOD.AGED.s());

context.getContentResolver().insert(Uri.parse(Users.CONTENT_URI+"/"+id+"/send"),cv);
```

Code Block 6 - sending a message to a user

To be notified when a message from a certain type is received, an application registers a *ContentObserver* using a URI present in *StaticUtils* data type. There is a distinct URI for each message type so the application be notified only for types they are interested in. In Code Block 7 we see the example of a chat, where it listens for text messages only when the application is open – registers the observer when the chat windows is opening (*onResume*) and unregisters it when it is closing (onPause).

```
@Override
protected void onResume() {
	super.onResume();

	ContentResolver cres = getContentResolver();
	cres.registerContentObserver(
		Uri.parse(StaticUtils.URIForNewTXTMsg),
		true,
		newTxtObserver);
}

protected void onPause() {
	super.onPause();
	getContentResolver().unregisterContentObserver(newTxtObserver);
}

private ContentObserver newTxtObserver = new ContentObserver(null){
    @Override
    public void onChange(boolean self) {
       Log.e(TAG, "New Message Notification");
       getMessageFromReinvent();
    }
};
```

Code Block 7 - registering an observer for new text messages

When the application is notified that a new message was received, it has to go get it from REINVENT – Code Block 8. For that it uses the *call* method from the content resolver, with the right URI. This URI are provided at *StaticUtils* as well. Response messages are modeled in a *MessageContainer* per type from where message fields can be obtained.

```
protected void getMessageFromReinvent() {
	Bundle bdl = getContentResolver().call(
	StaticUtils. URIProvider, StaticUtils.CallGetTXTMessageMethod,"txt",null);
	bdl.setClassLoader(MessageContainer.class.getClassLoader());

	final MessageContainer message = bdl.getParcelable(StaticUtils.
GetTXTMessagesGetParceable);

	//do something with the message
}
```

Code Block 8 - obtaining a message after a notification

Finally, since the system uses a database to cache values, the application that use REINVENT need permissions for that process – Code Block 9.

```
<!-- REINVENT -->
<uses-permission
android:name="android.permission.WRITE_EXTERNAL_STORAGE" />
<uses-permission
android:name="com.example.vanetprovider.networkprovider.READ_DATABASE" />
<uses-permission
android:name="com.example.vanetprovider.networkprovider.WRITE_DATABASE" />
```

Code Block 9 - permissions mandatory to use REINVENT

4.3.3 Gateway: VANETs and the world connected

We created a message interceptor on REINVENT component that runs on the OBU. This interceptor was made very generic based on the plugin pattern. Each plugin is simply a python file with a specific structure indicating what message type he processes and the code to do it, placed on a specific folder with a specific name prefix. After processing the message, it returns a yes/no value indicating if the message should be discarded or can continue his lifecycle – i.e. be sent to VANET or mobile phone. This way it may be used for other actions other than serving as a gateway; for instance, it is also used to answer STATUS messages which are used to get information about the OBU (e.g. OBU ID and connection status).

When the client starts, it looks for files with the prefix “plugin_” on the same folder as it is running and loads them. Each plugin must have, at least, the methods:

- init()
    - Called at the moment the plugin is first loaded
- messageType()
    - Should return a String with the message type this plugin wants to intercept.
- shouldInterceptFromPhone()
    - If this method returns *true* the plugin will intercept messages sent by the phone to the vehicular network.
- shouldInterceptFromVanet()
    - If this method returns *true* the plugin will intercept messages received from the vehicular network to be sent to the phone.
- processMessage(message, receivedFrom)
    - This method will be called each time a matching message – message type and origin – is received on the OBU. Parameters are, respectively, the message received and the source (phone or VANET). This method should return a negative value if the message should be discarded, or a non-negative value if it should continue his usual route.

Supported on the interceptor, we implemented a Gateway from VANET to the internet installed in OBUs/RSU's with a simple rationale. The Gateway may be useful to send VANET WAVE messages out of the vehicular network to a web server –in case of VNS, contacting the centralized server to keep it more complete/updated.

To create the gateway for VNS we added two plugins. One processes VNS requests (*DNS_TRANSLATE* messages), converts them to a HTTP requests for querying the webservice, waits for the response, parses the JSON response and sends a *DNS* message with the information from the server's response to the VANET. The second plugin processes VNS responses (*DNS* messages) from the VANET, converting them to HTTP requests to the webservice. The point of this is to make the server as updated as possible, with data of the multiple vehicular networks. The capacity to intercept messages and analyze, process or block them before they are sent brings a lot of flexibility to the system. Also adding new modules without the need to change any existing code means it is scalable and is easily updated.

4.3.3.1 VNS ***Server WS Interface***

This server provides the required services to VNS: register and query user-vehicle associations. It interfaces with the phone's module through REST *webservices* with JSON messages. They are used directly from VNS phone's component when the phone has internet connection or indirectly through some gateway on vehicular networks.

Public methods are (notice these are suffixes, the server address should be added before it):

A) Ask for the translation for a given username. This returns the last known identification registered for this user.

```
REQUEST

        GET          /username
```

```
RESPONSE

        [{"ip":"vehicleID"}]

              - or -

        [ ]                          ,if that username is not registered
```

Figure 16 - VNS Server API, translate action

B) Register a new association user-vehicle

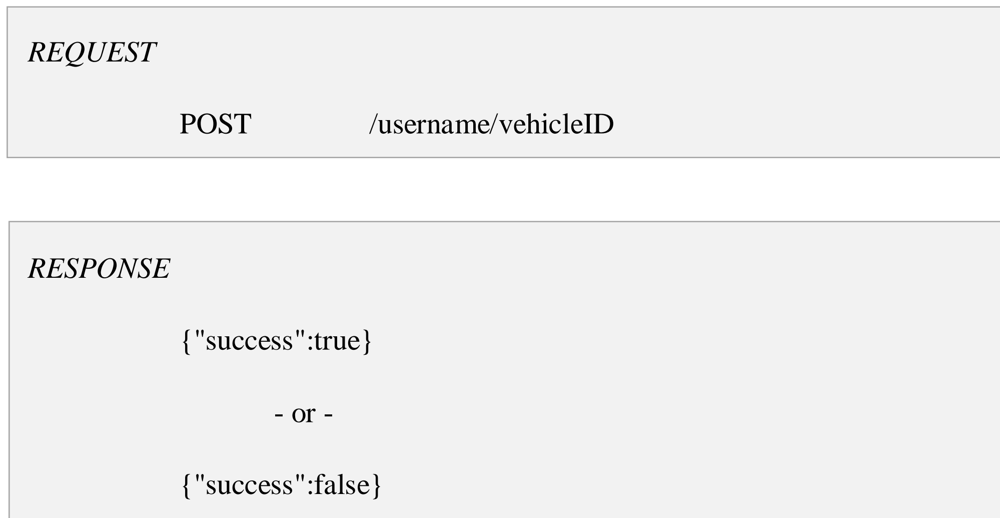


Figure 17 - VNS Server API, register action

4.3.4 VnAuth Developer Interface

In VANESS's android implementation, VnAuth was created using the standard Android authenticator [51] which interacts with VNS for registering the association user-vehicle when he first logs-in.

This authenticator has the lifecycle as shown on Figure 18.

Each time an application logs-in using VnAuth, the AccountManager is queried for the accounts existent on the device, for a specific account type. If one account exists, the authentication token is returned to the application, the login process is complete and the application can use that token to identify the user on future interactions (e.g. with the server). On other hand, if the user has not logged yet, there is no account saved on the phone, so the authenticator activity from VnAuth will be launched asking the user for credentials and registering them with the account manager, making this account available for one-click sign-in on future calls. If more than one account for a certain type is saved on the phone, then the user will be prompted to choose one to log in. The remaining process is the same.

Since at the moment we do not require password on VnAuth, we simplified, by using the username as the token, using it as a user identifier both on the device and server side. The authentication process was also simplified by assuming it was always valid (obviously this may bring some security issues, but this will be performed in future work).

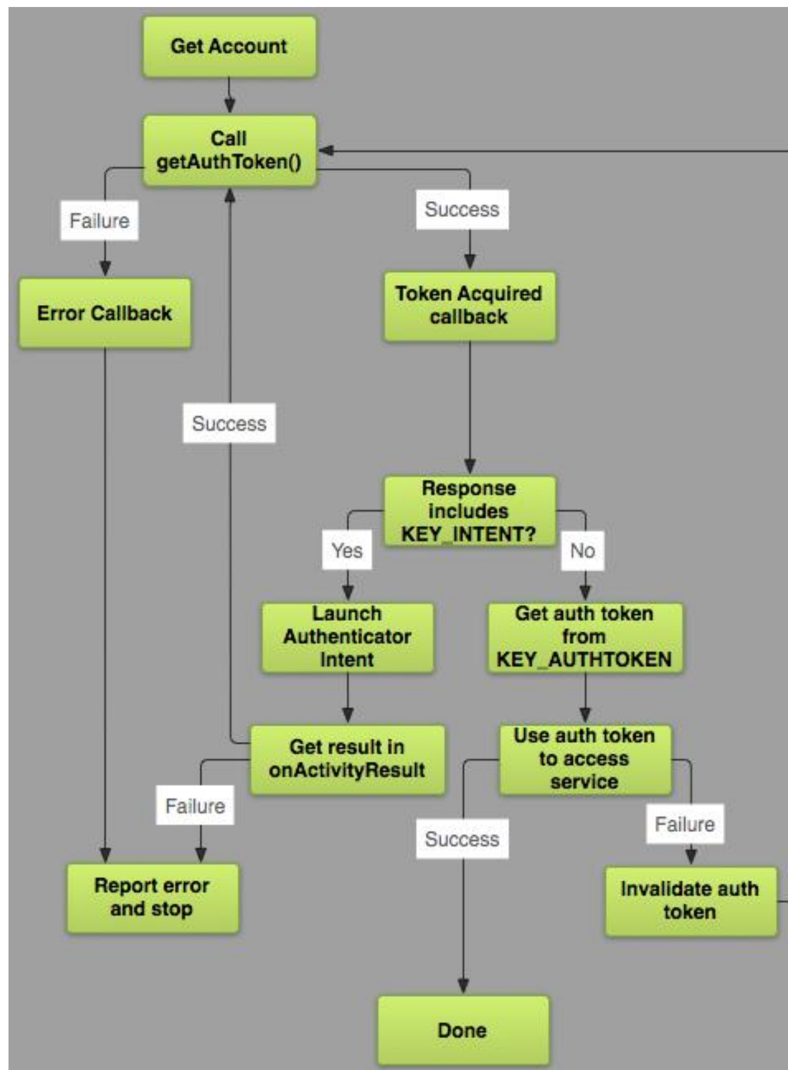


Figure 18 - Account Manager life cycle

4.3.4.1 *VnAuth developer interface*

We made it very easy for any application to add a “One-Click Login on VANET” Button: There is a class on *mAuxiliarLib* called *VnLoginButton*: adding it to the UI is done as any other UI component on Android, for instance on the XML file - Code Block 10.

```
<pt.ua.vnauth.VnLoginButton
        android:id="@+id/loginButton"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="center_horizontal"
        android:layout_margin="15dp" />
```

Code Block 10 - Adding login button to the Android’s UI

With this peace of code, all the account management and VNS calls are already happening when the user clicks the button. The application has then the option to register callbacks to the successful or failed login as shown on Code Block 11 and get the logged user info, as shown on Code Block 12.

```
VnLoginButton loginButton = (VnLoginButton) findViewById(R.id.loginButton);
loginButton.setCallback(new Callback() {
	@Override
	public void loginFailed(Exception theProblem) {
		theProblem.printStackTrace();
		//Login Failed: do something
	}

	@Override
	public void loginConfirmed(String token) {
		//Login Successful: do something
	}
});
```

Code Block 11 - Register callbacks to the login finished

```
AccountGeneral.getAuthenticatedAccount(context)
```

Code Block 12 - obtaining the token for the logged account

Android restricts the use of accounts, so to use VnAuth it is mandatory to add the permissions seen on Code Block 13 on the Manifest File.

```
<!-- VNAuth -->
<uses-permission android:name="android.permission.USE_CREDENTIALS" />
<uses-permission android:name="android.permission.GET_ACCOUNTS" />
<uses-permission android:name="android.permission.AUTHENTICATE_ACCOUNTS" />
<uses-permission android:name="android.permission.MANAGE_ACCOUNTS" />
```

Code Block 13 - permission to use VnAuth

4.4 Summary

This chapter aimed to be a guide to developers who

1) will use VANESS on their application: this chapter showed the public interface to be used;

2) aim to extend VANESS' features: this chapter presented internal details of the system required to understand our implementation.

As the VANESS system is an extension of REINVENT, the interface was not changed, for compatibility reasons. However, when sending a message, instead of always sending the destination vehicle identification as one of the parameters, now it can be either the destination vehicle or the destination username/alias.

The VNS module is called internally by REINVENT each time it has to send a message, or when any change is detected, either network change (connected to other OBU) or user change. Both REINVENT and VNS are implemented in Android as Content Providers.

The user management is done by the VnAuth module, which handles the sign up and login process in a device-centralized way. It is implemented using Android's built in account manager and allows for a “one click login”.

The gateway runs on OBUs and it is completely transparent for the app developers. It was created using the plugin analogy. Each plugin is a code file with a specific interface, where it defines the message type it processes, the code that processes it and decides if that message should continue his journey or must be discarded.

In the gateway plugin for VNS, it takes the parameters from VANET's VNS messages and does a request to the webservices. Then, it converts the response to a VANET's VNS response.

5 VANESS: applications and tests

We developed two applications in order to test VANESS' feasibility and ease of integration for application developers for different case scenarios, and to make clear the usefulness of the system. With these applications we tested the system's internal communication (between modules), the concurrent use by multiple applications on the same device, and finally its integration, synchronization and interoperability of the system with VANET and the internet. These applications are VNRide, a carpooling application directed to pre-planned journeys and VNBrowser, a very basic web page viewer to be used on vehicular networks.

5.1 VNRide

In **VNRide** people publish how many available seats on a future car trip they have available. If someone else needs a ride for that same area, they apply for it. With this we could test the VNS mechanism – both locally and with internet access - as well as the authentication system.

As soon as the application starts we see the login button in Figure 19, provided by the VnAuth module. If the user already has logged-in on that device, it will login with a single click; otherwise a generic prompt dialog will be shown, as seen on the right image. This dialog is not application-specific, it belongs to VnAuth and it is transversal throughout the device.

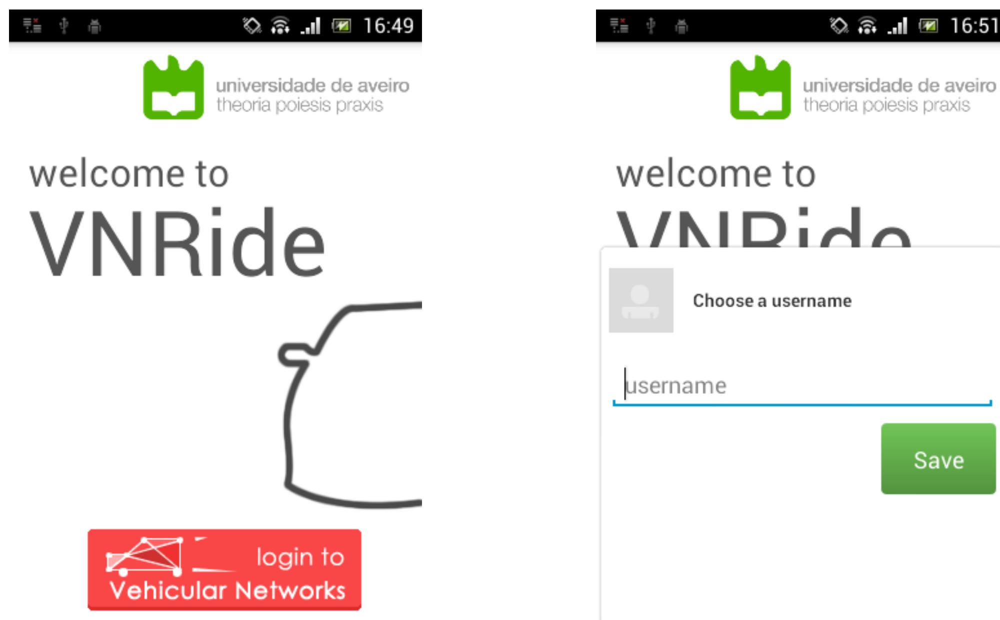


Figure 19 - VnAuth login button integrated in VnRide on the left; the authentication activity on the right. This activity appears if it is the first time logging-in on that device

When two users want to ride together, they are able to chat with each other – Figure 20. The chat messages are exchanged through the vehicular network even though the application itself

does not have information on where or in what vehicle the other user is. The chat relies on VNS to perform alias to real network addresses translation.

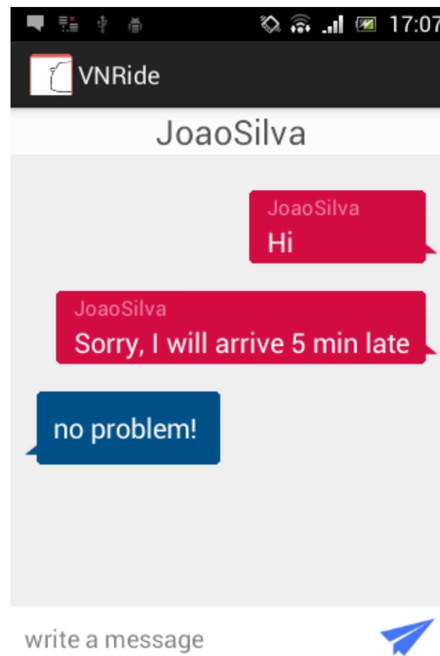


Figure 20 - Chat screen from VnRide, talking with JoaoSilva

Users acconts may be managed centrally on the Android's settings page - Figure 21. Beyond adding and deleting accounts to be used on vehicular networks, we also included some aditional configurations that let the user chooses if it allows our system to send data to be sent over wi-fi and/or VANET. These configurations are made per device, so it is not possible to have distinct settings for each user.

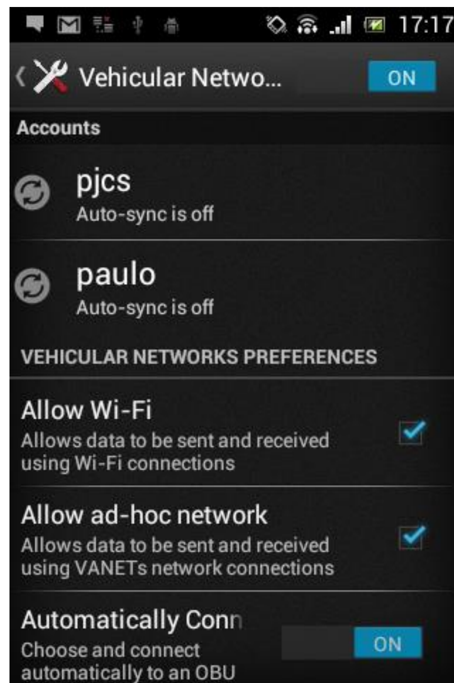


Figure 21 - Android's account settings for VnAuth: This screen is accessed via Setting application from the Android system and allows to define multiple settings for use of the network and account management

Figure 22 shows the remaining screens from VNRide's application. These interact with the services from the VNS server to search (A) and add new (B) rides from/to the system. Each ride is identified by the start and end destination, a realization date and the number of available seats which decreases when someone applies for a ride (by clicking on a search result).

Each user has a dedicated notification space (C) with warnings about their next actions, either to pick someone or to be picked by a driver.

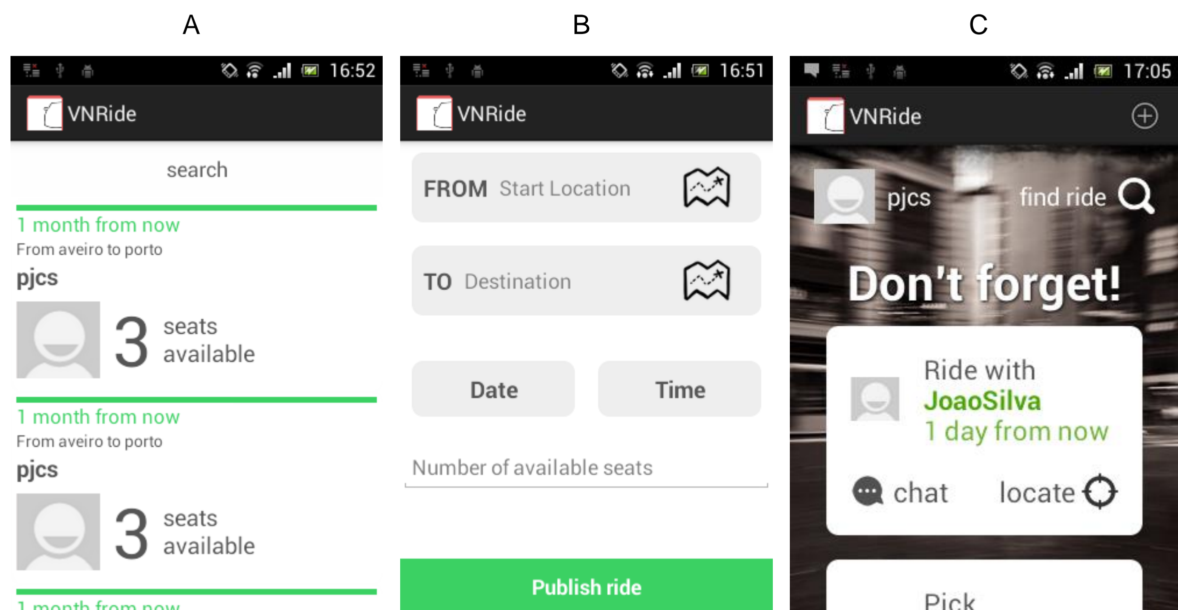


Figure 22 - VnRide screens. Search for a ride (A); Sharing my ride (B); My timeline/notifications (C);

5.2 iOS Consumer App

This simple application developed for iOS was created to show the ubiquitous nature of the system, proving that, although we implemented it on the Android system, it can be easily adapted to other platforms. On this proof of concept, we write any username and it gives us the identification of that person's vehicle. If no username is chosen, it returns all the users and their vehicles (as seen on Figure 23). At the moment this application does not have any VANET communication component yet; it is a consumer of VNS data. Future work may be done on this to implement the complete features for iOS or any other platform on which RabbitMQ is supported.

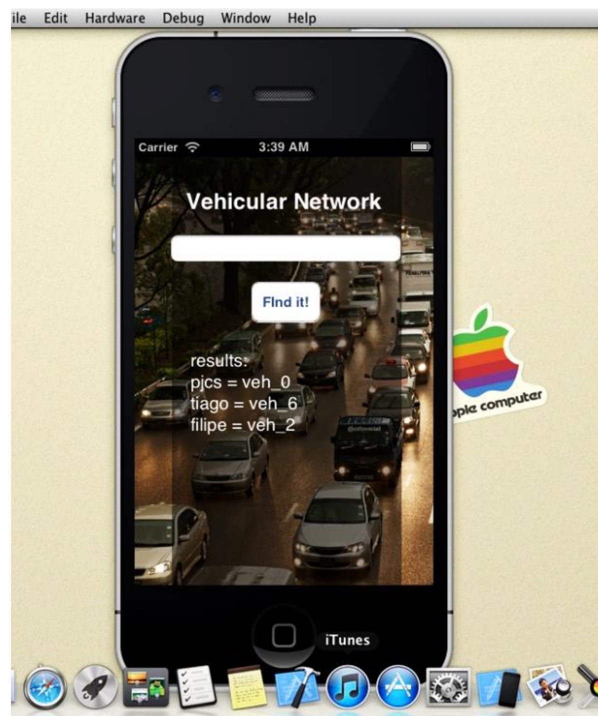


Figure 23 - VNS consumer app for iOS: this screen shows the emulator running the consumer application for VNS. Below the button we see a list of associations user-vehicle, returned from the server. This list contains the most recent ID of all the OBUs each user connected to.

5.3 VNBrowser: The gateway's proof of concept

This application sends a request for a page from the internet using a VANET message, waits for a response (HTML code) and shows it as a website – Figure 24. Since it is common for applications to access online contents (e.g. web-services), our system provides a way to convert VANET messages to usual HTTP requests and the response back to the client. These are a generic version of the previously mentioned "VNS gateway". With VNBrowser we could assess the robustness and extensibility of the gateway – the component that allows interoperability between in and out vehicular networks which is important for VNS since it has an outside server.

The aim of this application is to test and prove the gateway implementation, but it is important noticing that accessing websites using these VANET messages is unappropriated, since in the vast majority of websites the full response with all the HTML code is not scalable for big networks, due to the message's size.

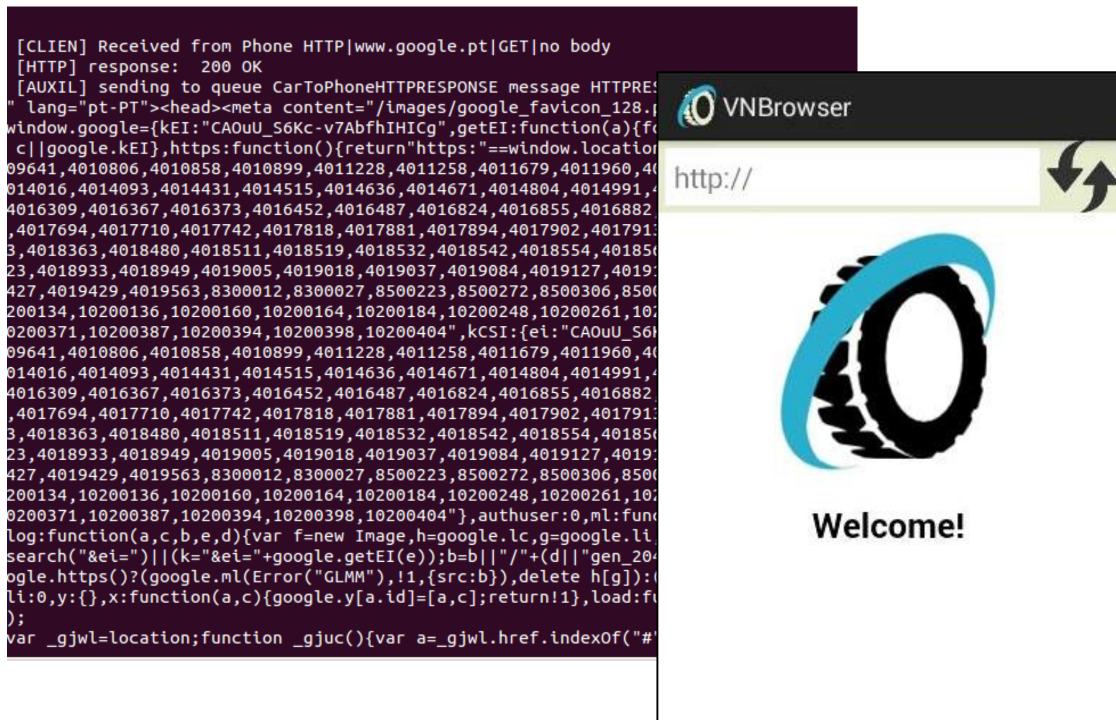


Figure 24 - Initial screen from VNBrowser and a request/response example to Google

In this section three mobile applications were introduced as proof of concept for VANESS: VNBrowser, as the proof of concept for the gateway; the iOS consumer to show the multi-platform potential of the concept; and VNRide as the main proof of concept which makes use of the complete life cycle of VANESS and also serves as a real use case where our system is useful.

5.4 System Tests

To finish with a robust concept and a working prototype, VANESS had to pass through multiple tests. These were used to find the system's weaknesses and then as validation processes. We started with simple tests directed to parts of the system which required few resources, and in a later phase we made complete tests using a real setup scenario, enabling a full lifecycle validation process.

5.4.1 Simplified tests

Our work evolved over a vehicular network system tested in real world scenarios, with communication capabilities V2X (i.e. vehicle-to-vehicle and vehicle-to-infrastructure) and using REINVENT to provide the ability to exchange messages between devices.

As we are working on a tested messaging system, there is no need to make road tests, since those questions lay on REINVENT layer. Furthermore, physically travel between multiple vehicles would be cumbersome and very inefficient, so we firstly created applications to emulate the connection to OBUs (entering a vehicle action), moving geographically and sending text messages. Later we tested the system with two real OBU as seen on Figure 1.

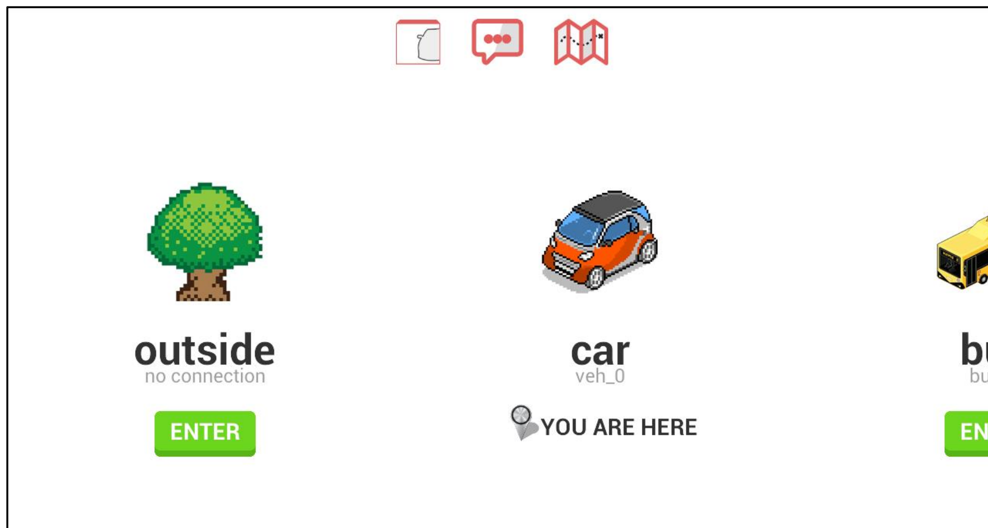


Figure 25 – First demo application: Home screen (in a tablet) with the user emulating his position on a car with the ID 'veh_0'.

Figure 25 shows the initial screen of the first demo application. This application served for initial tests. With it we could have any application running e.g. VnRide, and we would connect this application to the same OBU. It will emulate messages as if we received them via VANET from another OBU. This not only reduced the needed resources, as well as the setup time to test the system. It is useful to be used on presentations or demos to show our system working.

Pressing the ENTER button will emulate the action of connecting to a different on-board unit (i.e. entering a vehicle/place) in different conditions (internet access, VANET only or no connection at all) and/or with different vehicle IDs. It also has an "outside" option that serves to emulate the case of a user disconnecting or loosing connection from any network, which cannot be detected. Clicking on the balloon icon, we are able to send a text message for a specific destination – Figure 26 (A). This allowed us to check if messages were delivered to the right user

even after moving across multiple vehicles and places and if they were rightly filtered on the OBU. The map button opens a map – Figure 26 (B), so we can choose the place we want our emulated vehicle to be.

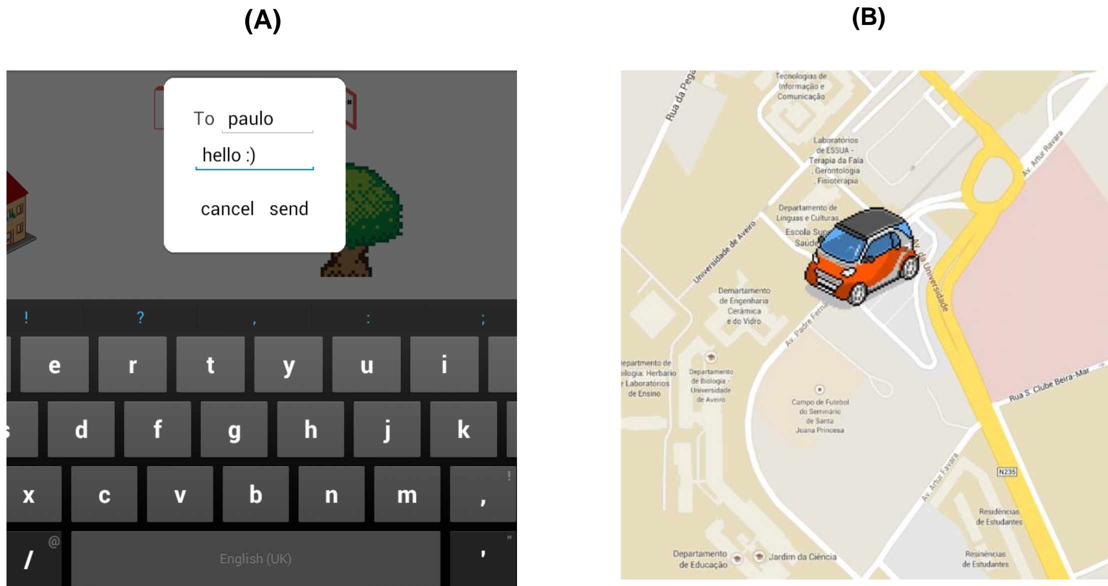


Figure 26 - On the left, the example of a message being sent; On the right, emulating the position of the vehicle

The first icon is VnRide logo, it makes a user apply for the last ride created with VnRide. This allows us to test the full VnRide life cycle, since the time someone shared a ride, someone applied and they talk and followed the other person.

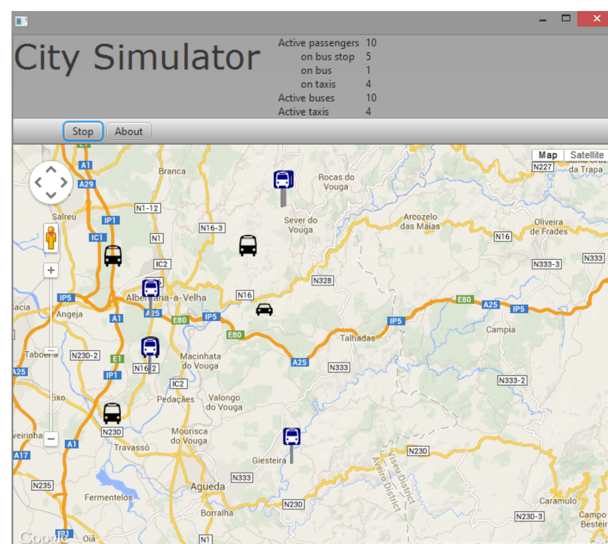


Figure 27 - City simulator, simulating the movement of passengers on taxis and buses

Figure 27 shows a simulator created from scratch to emulate the actions of entering and exiting vehicles (represented as taxis and buses) for multiple passengers, providing callbacks on each event. The number of buses, taxis and passengers are configurable on the start of the program. All the vehicle IDs, usernames, bus locations and journeys are randomly generated. It was configured to access the VNS server on each "enter vehicle action" to emulate real persons using the server. Values from the database can be compared with the logs of this simulator to validate the server's behavior. This is a multi-platform native application where each active element (buses, taxis and passengers) is modeled as an independent thread with inter-thread communication and shared resources (buses and taxis) synchronized in a readers-writer lock scheme [57].

5.4.2 Real setup Tests

To perform real tests we used two similar OBUs with VANET and Wi-Fi capabilities, and one mobile phone connected at each one – Figure 28. In some tests we connected one of the OBUs to the internet shared through a computer's Ethernet connection. This was enough to test all the features and the complete message cycle.

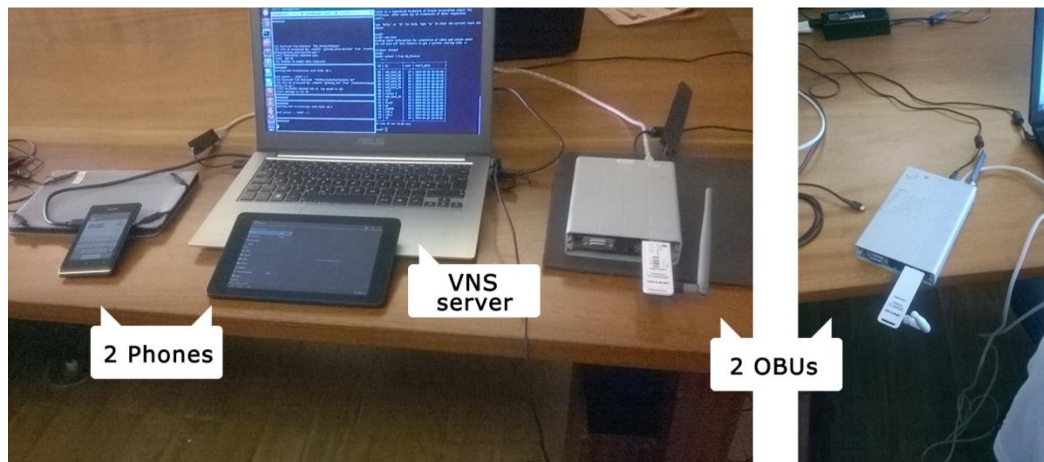


Figure 28 - Our test scenario: Two OBUs with VANET + Wi-Fi capabilities, one public web server with VNS services running and two mobile devices. The phone was connected to one OBU and the tablet to the other one. The OBUs are connected to the PC to debug proposes and, in some scenarios, to get internet access through the PC's Ethernet connection.

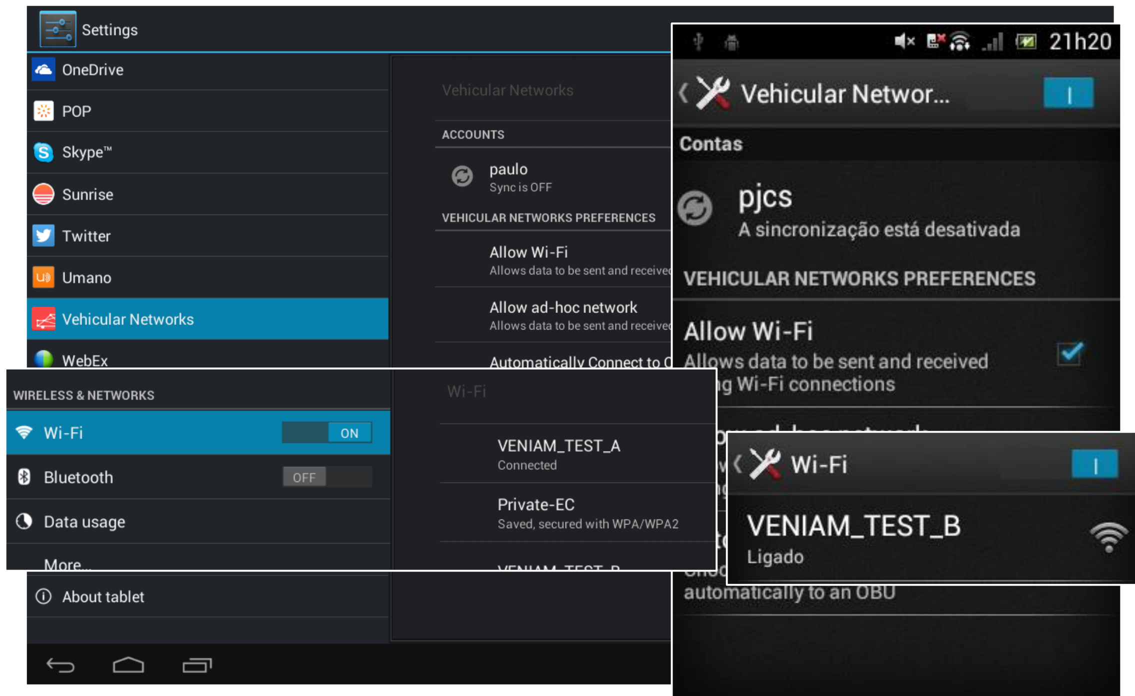


Figure 29 – setup for one test example: two devices, with a different user and connected to different vehicles. A tablet (on the left) and a phone (on the right)

Figure 29 shows one example setup of the tests, where we had two devices, with a different users and connected to different vehicles. In the tablet (on the left) the user is 'paulo' and is connected to the network "VENIAM_TEST_A" while on the phone (image on the right) is the user 'pjcs' connected to "VENIAM_TEST_B" (other OBU). The back images are the Android's settings screen for the "Vehicular Networks" account where we can see the accounts logged on the phone.

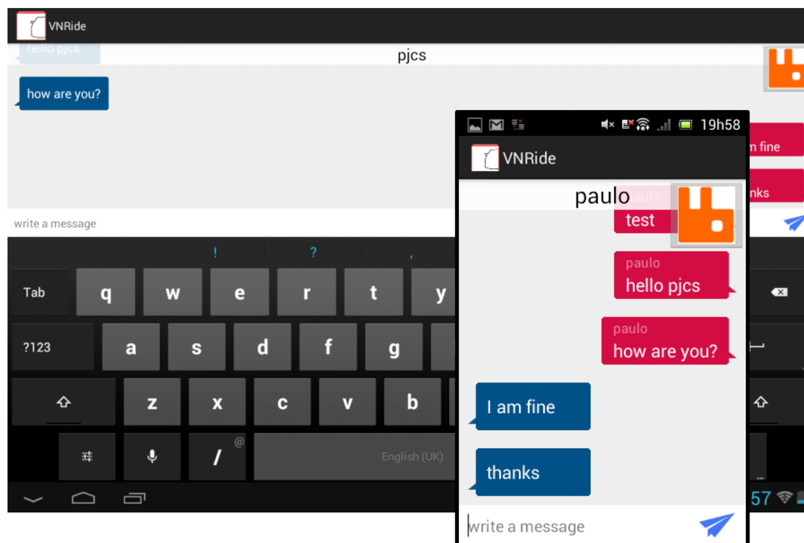


Figure 30 – Demo of two users chating on VANET

Figure 30 shows the chat screens for these same devices after a few messages have been exchanged. Notice that the username on top of the screen is the destination username

```
[CLIEN] Received from Phone DNS_TRANSLATE|ghost
[VNS_T]  200 OK
[VNST!] No translation found!
[CLIEN] sending to vanet DNS_TRANSLATE|ghost
```

Figure 31 - Log messages from one of the OBUs. VNS server was online; A message was sent to a inexistent user (ghost) resulting in a "No translation found".

```
[CLIEN] Received from Phone DNS_TRANSLATE|pjcs
[VNS_T]  200 OK
[AUXIL] sending to queue CarToPhoneDNSRESPONSEMSG message DNS|-1|pjcs|12
[CLIEN] sending to vanet DNS_TRANSLATE|pjcs
[CLIEN] Received from Phone TXTMSG|txt|pjcs|12|hello pjcs
[CLIEN] sending to vanet TXTMSG|txt|pjcs|12|hello pjcs
```

Figure 32 - Log messages from one of the OBUs. VNS server was online; A message was sent to an existent user (pjcs) resulting in a response from the server with the vehicle ID = 12.

Figure 31 and Figure 32 show log messages of the local OBU when sending two different messages as examples. In the first, a message was sent to a user that does not exist. “200 OK” means the response from the server was received but the next message indicates that the server did not know that user. On the other hand, the second message was sent to an existent user, the server returned the correspondent vehicle’s ID, and then the response was sent to the phone – since that was the source of the request. In the text message sent (last two lines) from Figure 32, it is visible the identification of the vehicle embedded on the message (12).

```
[CLIEN] received from vanet: WSM received... Message: 'DNS_TRANSLATE|paulo' | PSID = 80-1

[VNST!] No connection to VNS Server:  [Errno 113] No route to host
[CLIENT] sending to queue CarToPhoneDNSREQUESTMSG message DNS_TRANSLATE|paulo
[CLIEN] Received from Phone DNS|1403960874532|paulo|12
[VNS_R] REGISTERING paulo WITH ID 12
[G!] [Errno 113] No route to host
[CLIEN] sending to vanet DNS|1403960874532|paulo|12
[CLIEN] received from vanet: WSM received... Message: 'TXTMSG|txt|pjcs|12|test' | PSID = 80-1

[TXT  ] Message is for me
[CLIENT] sending to queue CarToPhoneTXT message TXTMSG|txt|pjcs|12|test

[CLIEN] Received from Phone DNS_TRANSLATE|pjcs
[VNST!] No connection to VNS Server:  [Errno 113] No route to host
[CLIEN] sending to vanet DNS_TRANSLATE|pjcs
[CLIEN] received from vanet: WSM received... Message: 'DNS|1403960903055|pjcs|13' | PSID = 80-1

[VNS_R] REGISTERING pjcs WITH ID 13
[G!] [Errno 113] No route to host
[CLIENT] sending to queue CarToPhoneDNSRESPONSEMSG message DNS|1403960903055|pjcs|13
[CLIEN] Received from Phone TXTMSG|txt|paulo|13|test2
[CLIEN] sending to vanet TXTMSG|txt|paulo|13|test2
```

Figure 33 - Debug messages from one OBU when two users exchanged messages. First a message was sent from "pjcs" to "paulo" saying "test"; then a response saying "test2"

Figure 33 contains the debug messages from one OBU when two users exchanged messages, in a scenario where there is no connection to the VNS Server (Errors 113 on the picture). This debug corresponds to a message being sent from "pjcs" to "paulo" saying "test"; then a response saying "test2". This figure corresponds to the OBU where paulo's device is connected to.

First we see a message received from the VANET asking for the translation of the username "paulo". This request is sent to the phone and, since that user was logged on that device, it responded with a VNS response, announcing that user "paulo" is at vehicle with ID 12.

Then, the message itself is received (TXTMSG) with the destination vehicle's ID (12) embedded on it. This happened because the VNS response was sent and processed by the other mobile phone, which only sent the message after receiving the corresponding ID.

The message destination ID is compared with the own OBU ID, and since it matched (debug "this message is for me"), the received message is added a queue to be consumed by the phone.

The same process is repeated to the response from "paulo" to "pjcs".

As mentioned before, a gateway software was created to run on the OBU and do the translation between the VANET messages and internet/web protocols. To test this feature independently, VNBrowser proof of concept was used. Figure 34 and Figure 35 are debug messages form one OBU with internet access, made using VNBrowser to access *www.google.pt*.

In Figure 34 the request was made from the locally connected phone, while in Figure 35 the request was made indirectly from a phone in another node on the network. The message was then sent over the VANET until it reached this OBU which responded with the Google's page HTML content. The response is sent to the phone or VANET acordingly to the origin of the request.

```
[CLIEN] Received from Phone HTTP|www.google.pt|GET|no body
[HTTP] response:  200 OK
[AUXIL] sending to queue CarToPhoneHTTPRESPONSE message HTTPRESPONSE|200|<!doctype html><html item
 lang="pt-PT"><head><meta content="/images/google_favicon_128.png" itemprop="image"><title>Google<
indow.google={kEI:"CAOuU_S6Kc-v7AbfhIHICg",getEI:function(a){for(var c;a&&(!a.getAttribute||!(c=a.
```

Figure 34 - Debug messages for the Gateway action, when the local OBU has internet connection

```
root@drivein12 Gateway]#  [CLIEN] received from vanet: WSM received... Message: 'HTTP|www.google.pt|GET|no body'

[HTTP] response:  200 OK
[AUXIL] sending to queue PhoneToCarHTTPRESPONSE message HTTPRESPONSE|200|<!doctype html><html itemscope="" itemty
 lang="pt-PT"><head><meta content="/images/google_favicon_128.png" itemprop="image"><title>Google</title><script>
indow.google={kEI:"owKuU-iLCZDG7AaN6IDADA",getEI:function(a){for(var c;a&&(!a.getAttribute||!(c=a.getAttribute("e
```

Figure 35 - Debug messages for the Gateway action, when OBU receives a request from the VANET

The use cases used to test and guarantee the proper operation of the various components of the system are the following:

A) Chat

Sign-in on both phones with different usernames and using VnRide's chat send messages between them. Since the chat application only knows the username of the destination user, a message being delivered means a correct translation from VNS.

B) Chat with gateway

One of the OBUs and the computer are configured to share internet access between them and the VNS server deployed. Then, the same steps as in A) are performed. Each time we login on a new device connected to VANET a new entry on VNS server's table appeared. This time, even if one of the phone was not connected, the name translation was successful using server's response.

C) Switch vehicles:

Start as use case A). After successfully exchanging some messages the devices switch their position – each one disconnects from the current OBU and connects to the other one. Then messages are again exchanged with success. This proves that VNS works in real time and that it was able to send deliver messages even after some user(s) change position.

D) Switch vehicles with gateway:

Similar to C) but this time one of the OBUs has internet connection and VNS server is active. The act of exchanging vehicles also causes a new entry on the VNS server to appear.

E) Disconnect from VANET and connect to the internet

When a user connects to the internet, but outside the vehicular network the server is updated.

F) Access a website

One OBU on the network has internet access. Using the VNBrowser and the phone connected to an OBU, access a website without redirects. The page will appear on the browser, without images (redirects and images not supported yet).

5.4.3 Test conclusions

After these tests, we concluded that VANESS is a viable and scalable prototype, which simplifies the development of user-level applications, since it removes the endeavor of user discovery for each new mobile application who needs it. Although it is not infallible (i.e does not provide a delivery guarantee), it is scalable due to its reduced number of message exchanges and small message sizes. These tests also revealed questions/limitations that need improvement: 1) REINVENT's OBU component uses round robin delivery to mobile devices connected, meaning that when multiple devices are connected to the same OBU, each message is only

delivered to one of them, possibly the wrong one; 2) REINVENT's OBU component does not detect when a mobile disconnects from the OBU, continuing delivering messages to him. In conjunction with problem 1) this causes part of the messages to be lost.

5.5 Summary

In this chapter we presented the proof of concept, test applications, test setups and the test results.

VNRide, the main proof of concept, is a carpooling application directed to pre-planned journeys. It makes use of all the modules on the system while demonstrating the need a user-to-vehicle translation system on vehicular networks.

More specific applications were also created, for instance VNBrowser, which is a simple browser application that accesses websites in VANET scenarios using a message type which is converted to HTTP access when it finds a gateway.

Due to the complexity of the real test scenario, we firstly created a set of test tools to realize the basic try-outs and essays. After the validation of the main ideas, we moved to real test scenarios using real OBUs, where we tested and improved the proof of concept, finishing with a viable and scalable system.

6 Conclusions

The goal of this Dissertation was to make communication on vehicular networks available to nomadic users (move between different vehicles and different networks), providing user-to-user communication available to third party applications on vehicular networks.

We researched several solutions and, based on acquired knowledge, we idealized our own – VANESS system – **VA**NET D**NS**, a naming service to support user-to-user communication on mobile applications. We opted to adapt REINVENT [15], an abstraction layer which provided mobile applications to exchange messages between vehicles (not users) in a vehicular network context, without dealing with the network specificities. After understanding the initial state of this system and how to adapt it, we implemented a full version of our solution to work with Android OS and a simple application on iOS to prove the his multi-platform nature.

Conceptually VANESS has two modules: 1) VNS (VANET Naming Service) – responsible for keeping the association between users and the endpoint (e.g. vehicles) in real time, inside and outside VANET scenarios. This module provides an interface on the mobile phone side for third party mobile applications to send and receive messages between users, using vehicular network resources, without any other information about the destination user other than his username. It has an OBU component which handles the interaction between the phone, the VANET and the internet. 2) VnAuth – Responsible for the user management on mobile devices and communicating the login action to VNS. Third party applications only need to use a "Login Button" supplied by this module, and automatically it uses a device-centralized authentication process, where the user enters his credentials only once per device, and the login state is shared across all the applications that use VnAuth. After the first login, the button provides a "one click login" especially useful while driving conditions. We tested it successfully with two real on-board units and VNRide (our main proof of concept), a carpooling application directed to pre-planned journeys, shown the need for our system. It is worth noticing that all the application developed for REINVENT also work with our system, since we provide full compatibility between the previous versions of it.

At the end, we concluded that VANESS is a real advantage for the integration of mobile devices on vehicular networks easing the creation of new applications for users on vehicular networks. This architecture might propel the development of new and innovative applications to be used in a near future, with a lot of interesting use cases.

6.1 Future Work

The system that has been proposed here has been implemented and tested, although with some simplifications. Before it can be integrated in a real world scenario, the user authentication needs to be verified in some way to avoid duplicated usernames on the same network or even

better avoid duplicated usernames at all, since it will be confusing if people we want to interact with are constantly changing their usernames and this taken by others.

The ID of the vehicle is, currently, the only information about the addresses of the message that the format of a text message contains. Since - as detailed on section 5.4.3 - REINVENT was only prepared to deal with one phone per OBU, this model works fine. However, when extending REINVENT to work with more than one device per vehicle, it is important to add the username/alias field to the text message type, so the messages will be filtered and only delivered to the right device.

A simple iOS application was developed to show the multi-platform abilities of the system, but to become useful the developed project needs to be extended. It is necessary to convert the code done in all Android's modules to iOS and possibly to other platforms. The OBU and server components are independent of the device platform, so they do not need any change.

Some improvements might be interesting to look at: once our system has a centralized component (VNS server) able to have information about all vehicles independently of the ad-hoc networks he is in (when that network has some gateway), it would be useful the creation of some proxy between those VANETs, enabling messages to be sent to a completely different ad-hoc network in the world, since each had at least one gateway available.

7 Bibliography

[1] International Council on Clean Transportation, “Chapter 2 – Number of Vehicles” in “European Vehicle Market Statistics”, 2013

[2] World Health Organization, “Global status report on road safety”, 2013

[3] Glavaski, S.; Chaves, M.; Day, R.; Nag, P.; Williams, A.; Wei Zhang, "Vehicle networks: achieving regular formation," American Control Conference, 2003. Proceedings of the 2003 , vol.5, no., pp.4095,4100 vol.5, 4-6 June 2003

doi: 10.1109/ACC.2003.1240477

[4] Yeongkwun Kim; Injoo Kim, "Security issues in vehicular networks," Information Networking (ICOIN), 2013 International Conference on , vol., no., pp.468,472, 28-30 Jan. 2013

doi: 10.1109/ICOIN.2013.6496424

[5] Verroios, V.; Vicente, C.R.; Delis, A., "Detecting Hazardous Vehicles and Disseminating Their Behavior in Urban Areas," Mobile Data Management (MDM), 2012 IEEE 13th International Conference on , vol., no., pp.280,281, 23-26 July 2012
doi: 10.1109/MDM.2012.24

[6] Apple, “CarPlay” http://www.apple.com/ios/carplay/, March, 2014

[7] Google, Android Auto, http://www.android.com/auto/, June, 2014

[8] Google, “Open Automotive Alliance”, http://www.openautoalliance.net/, March, 2014

[9] Microsoft, “Windows Embedded Automotive”, http://www.microsoft.com/windowsembedded/en-us/windows-embedded-automotive-7.aspx, March, 2014

[10] The Verge, “Google made a self-driving car, and it doesn't have a steering wheel”, http://www.theverge.com/2014/5/27/5756436/this-is-googles-own-self-driving-car, May 27 2014

[11] Karagiannis, G.; Altintas, O.; Ekici, E.; Heijenk, G.; Jarupan, B.; Lin, K.; Weil, T., "Vehicular Networking: A Survey and Tutorial on Requirements, Architectures, Challenges, Standards and Solutions," Communications Surveys & Tutorials, IEEE , vol.13, no.4, pp.584,616, Fourth Quarter 2011

doi: 10.1109/SURV.2011.061411.00019

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

[13] Mershad, K.; Artail, H., "Finding a STAR in a Vehicular Cloud," Intelligent Transportation Systems Magazine, IEEE , vol.5, no.2, pp.55,68, Summer 2013

doi: 10.1109/MITS.2013.2240041

[14] Lakas, A.; Serhani, Mohamed Adel; Boulmalf, M., "A hybrid cooperative service discovery scheme for mobile services in VANET," Wireless and Mobile Computing, Networking and Communications (WiMob), 2011 IEEE 7th International Conference on , vol., no., pp.25,31, 10-12 Oct. 2011

doi: 10.1109/WiMOB.2011.6085394

[15] F. Oliveira, S. Sargento, J. Fernandes, A. Cardote, “REINVENT: Accessing Vehicular Networks in Mobile Applications”

[16] European Union, Directorate-General for Mobility and Transport, “Intelligent Transport Systems in action”, 2011

[17] University of Waterloo, “VANET/DTN”, http://bbcr.uwaterloo.ca/~slcesped/vanet/

[18] A. Dahiya, R. Chauhan; “A Comparative study of MANET and VANET Environment”, Journal of computing, Vol.2, Issue 7, July 2010

[19] IEEE 802.11p standard, http://standards.ieee.org/findstds/standard/802.11p-2010.html, 2010

[20] CNET, “US to push for mandatory car-to-car wireless communications”, http://www.cnet.com/news/us-to-push-for-mandatory-car-to-car-wireless-communications/, February 2014

[21] University of Michigan – Mobility Transformation Center, “Michigan Mobility Transformation Facility”, http://www.mtc.umich.edu/test-facility, June 2014

[22] Carlos Ameixieira, André Cardote, Filipe Neves, Rui Meireles, Susana Sargento, Luís Coelho, João Afonso, Bruno Areias, Eduardo Mota, Rui Costa, Ricardo Matos, João Barros, “HarborNet: A Real-World Testbed for Vehicular Networks”, aceite IEEE Communications Magazine, ISSN: 0163-6804, Setembro 2014

[23] Carlos Ameixieira, José Matos, Ricardo Moreira, André Cardote, Arnaldo Oliveira, Susana Sargento, An IEEE 802.11p / WAVE Implementation with Synchronous Channel Switching for Seamless Dual-channel Access, IEEE Vehicular Networking Conference (VNC), Amsterdam, The Netherlands, Novembro 2011

[24] Filipe Neves, André Cardote, Ricardo Moreira, Susana Sargento, Real-world Evaluation of IEEE 802.11p for Vehicular Networks, ACM International Workshop on VehiculAr Inter-NETworking (VANET), ACM MOBICOM, Las Vegas, USA, Setembro 2011

[25] U.S Department of Transportation, NHTSA, “The Impact of Hand-Held And Hands-Free Cell Phone Use on Driving Performance and Safety-Critical Event Risk”, 2013, Cap. 4

[26] Word Health Organization, NHTSA, “Mobile Phone Use: A growing Problem of Distraction”, 2011

[27] Accenture, “In-Vehicle Infotainment - A View of the European Marketplace”, http://www.who.int/violence_injury_prevention/publications/road_traffic/distracted_driving_en.pdf, 2010

[28] MOST cooperation, http://www.mostcooperation.com/en/, May 2014

[29] Macario, G.; Torchiano, Marco; Violante, M., "An in-vehicle infotainment software architecture based on google android," Industrial Embedded Systems, 2009. SIES '09. IEEE International Symposium, pp.257,260, 8-10 July 2009

doi: 10.1109/SIES.2009.5196223

[30] 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.

[31] Hernandez, U.; Perallos, A.; Sainz, N.; Angulo, I., "Vehicle on board platform: Communications test and prototyping," Intelligent Vehicles Symposium (IV), 2010 IEEE , vol., no., pp.967,972, 21-24 June 2010

doi: 10.1109/IVS.2010.5548037

[32] "Instant Mobility Use Case Scenarios Functional and Non-Functional Requirements", Multimodality for people and goods in urban areas, FP7 CP 284906, March 2012

[34] P. Mockapetris, *Domain names - Concepts and Facilities,* IETF RFC 1034, http://www.ietf.org/rfc/rfc1034.txt, November 1987

[35] P. Mockapetris, *Domain names - Implementation and Specification,* IETF RFC 1035, http://www.ietf.org/rfc/rfc1035.txt, November 1987

[36] S. Cheshire and S. Cheshire, *Multicast DNS,* IETF RFC 6762, http://tools.ietf.org/html/rfc6762, February 2013

[37] Android, Google, “Implementing GCM Client, Register for GCM”, http://developer.android.com/google/gcm/client.html, 2014

[38] Wikipedia, “RabbitMQ”, http://en.wikipedia.org/wiki/RabbitMQ, May 2014

[39] RabbitMQ, http://www.rabbitmq.com/, October 2013

[40] RabbitMQ, “Get started Guide”, http://www.rabbitmq.com/getstarted.html, October 2013

[41] Android, Google “BroadcastReceive”, http://developer.android.com/reference/android/content/BroadcastReceiver.html, 2013

[42] Android, Google “AccountManager”, http://developer.android.com/reference/android/accounts/AccountManager.html, 2013

[43] LinkedIn, Udi Cohen profile, https://www.linkedin.com/pub/udi-cohen/48/6b4/141, June 2014

[44] Any.do, http://www.any.do/, June 2014

[45] Udinic, "Write your own Android Authenticator", http://udinic.wordpress.com/2013/04/24/write-your-own-android-authenticator/, January 2014

[46] Android, Google "Content Provider", http://developer.android.com/guide/topics/providers/content-providers.html, 2013

[47] Android, "Stopping and Restarting an Activity", http://developer.android.com/training/basics/activity-lifecycle/stopping.html#Stop, December 2013

[48] Android Design Patterns, "Content Resolvers and Content Providers", http://www.androiddesignpatterns.com/2012/06/content-resolvers-and-content-providers.html, January 2014

[49] Tonic, https://github.com/peej/tonic, November 2013

[50] Zebra Database, http://stefangabos.ro/php-libraries/zebra-database/, November 2013

[51] Android, "Android Authenticator", http://developer.android.com/reference/java/net/Authenticator.html, January 2014

[52] Apache, http://www.apache.org/, 2013

[53] MySQL, http://www.mysql.com/, 2013

[54] PHP, http://www.php.net/ ,2013

[55] Github, Tonic, https://github.com/peej/tonic , 2013

[56] Tonic, https://github.com/peej/tonic , 2013

[56] Wikipedia, "Readers–writer lock", http://en.wikipedia.org/wiki/Readers-writer_lock, 2013

8 Appendices

8.1 VANESS deployment

This section reports VANESS' dependencies and software used to deploy the complete system. The VNS server needs a HTTP server, a relational database management system and PHP. In our tests we used Apache 2.4.4 [48] as the web server and MySQL 5.6.12 [49] for the database and PHP version 5.4.16 [50]. To setup the webservices it's required Tonic [51, 52] and add the services implementation files' path on Tonic's dispatcher. For Tonic to work it's important to configure Apache to use *.htaccess* configuration files. Zebra_Database [46] is also required on the server since the VNS interaction with the database is done over it.

As REINVENT and VNS are provided as Content Providers (centralized) and VnAuth as Authenticator (also centralized) so that any app can use them it is mandatory to install them first. The OBU system needs Python and RabbitMQ installed. For the VNS Gateway to work properly it needs Python 2.7.3 or later. RabbitMQ should be started before the REINVENTs' Client, which in turn will automatically import and make calls to the plugins/gateway.

The system's main file on the OBU is *Client.py* which is the REINVENT's OBU component plus the message interceptor (the basis for the gateway). All the files with the prefix *plugin* will be loaded with the client, and each one processes one specific message type:

plugin_dns and *plugin_dnstranslate* handle the conversion from VNS messages from VANET's format to HTTP's request to the server and, inversely, from the HTTP response to VANET's response formats. *plugin_http* is similar but generic, since it allows requests to any server address.

plugin_status intercepts status requests prevenient from the mobile phone. This plugin answers directly with the information about the OBU (e.g. vehicle ID and network availability) and discards them after answering, since their not to be sent to other vehicles.

plugin_txt assess the destination ID, and if the message is to another node the message is discarded i.e. never sent to connected devices/applications.

The remaining files present on the OBU are auxiliar code or scripts used to start (e.g. *startClient* or *startRabbit*) or configure the system (e.g. configBridge – allow internet connection throught Ethernet).

8.2 Parameter list for each message type

This table provides the names of all the parameters that shall be added to the *ContentValues* data type when sending a message, depending on the message type.

Message Type	Parameter Name
TXTMSG	type message sender `Users.ALIAS` or `Users.NAME`
ALERTMAPMSG	AlertType Problem Description Latitude Longitude CarID
ALERTVEHICLEMSG	AlertType Problem Description Latitude Longitude CarID Value
STATUSREQUEST	
CUSTOMMSG	BODY MSGCODE
HTTP	BODY METHOD LINK
DNS_TRANSLATE	USERNAME
DNS	USERNAME TIMESTAMP VEHICLEID