

POLITECNICO DI TORINO

Master of Science in Computer Engineering

Master Thesis

V2X Connectivity and User Experience on Android

Design and Development of Applications Concerning C-ITS



Advisor

prof. Gianpiero Cabodi

Author

Fabiola POLIDORO

Tutor

eng. Fabio Toso

Magneti Marelli S.p.A.

APRIL 2018

Summary

Context

This Master Thesis has been written in collaboration with Magneti Marelli S.p.A., at Venaria Reale plant, under the supervision of engineer Fabio Tosetto, Project Leader of “Technology Innovation – Innovation & Connectivity” department.

The research team is focused on the field of next generation Intelligent Transport Systems, called Cooperative-ITS.

C-ITS are systems that allow vehicles to connect to each other, to other road users and to the road infrastructure through wireless technology, in order to allow data exchange for improving road safety and optimizing traffic flows to reduce environmental pollution.

For what concerns cars, the Cooperative Intelligent Transport System is implemented through a technology called V2X, Vehicle-to-Everything, that is a vehicular communication system through which vehicles sense the environment and send information to any entity that may affect or be affected by their presence.

The team has currently developed a prototype of V2X communication which is being tested on company cars both on track and on road.

Goals

This paper is going to illustrate the design of two Android applications addressing likewise different topics concerning V2X.

The first one concerns a specific type of V2X communication, called Vehicle-to-Pedestrian or V2P, which is the one occurring between a vehicle and a non-motorized road user such as, indeed, a pedestrian. The application implements the pedestrian-part of the communication and brings the V2X connectivity on the smartphone of the user, thanks to a special plug-in hardware designed by Magneti Marelli.

The hardware, along with the application, are one of the innovations that the Company has brought to the Consumer Electronic Show 2018 in Las Vegas, USA.

The second topic is related to the V2X system mounted on a car: the testing phase is currently uncomfortable because the events appear only on the dashboard in front of the driver and the logs reading require a laptop to be connected to the system. The goal of the second application is to implement a replica of the V2X system mounted on a vehicle so that all the passengers can see what is happening on the dashboard and easily read the logs.

This paper, after a brief introduction to Vehicle-to-Everything communication, is going to present the design of the two applications describing the reasons of the selected implementations and comparing them with alternative ones.

Results

Both the applications developed for this thesis are based on a client-server architecture and carry out the client role.

They have been written in Java, following Agile approaches such as product backlogs and exploratory testing and the rules of Material Design for what concerns the Graphical User Interface. Indeed, particular attention has been given to that in order to obtain usable solutions, that means the user can interact with the applications in an effective, efficient and satisfactory way.

All the images used by the applications have been especially drawn and rendered respectively with Adobe Illustrator and Blender in order to be compliant with the Material Design guidelines.

Both applications have been written in a modular way for allowing further changes and expansions.

The V2P application is made of a background service that connects to the V2X infrastructure and retrieves the real-time position of a vehicle. If the distance between that and the user is below a certain value, a warning notification is sent to the user. The Graphical User Interface allows the user to know the current state of the service and manage the settings of the application.

The V2X mirroring application (also called V2X HMI application) allows the user to connect to the V2X board of the car and see, on a mobile device, the V2X events currently displayed on the car's dashboard. Beyond that, the application can also show the logs produced by the V2X Connectivity board, allowing the user to filter them by priority.

Acknowledgements

I would like to thank my tutor Eng. Fabio Toso and the whole Innovation & Connectivity team for providing me support and advices through the development process and the drafting of this thesis.

Contents

Context	2
Goals	2
Results	3
1 Introduction	12
1.1 The Company	13
1.1.1 Innovation & Connectivity	13
1.2 C-ITS and V2X	14
1.2.1 V2X Standards and Legislation	15
1.3 Goals of this thesis	16
1.4 Tools	16
1.5 Development methodology: Agile	18
1.6 Graphical User Interface	19
1.6.1 Usability	19
1.6.2 Material Design	19
I V2P application	21
2 Vehicle-to-Pedestrian	23
3 Architecture	25
3.1 V2P Architecture	25
3.2 Customized V2P Architecture	26
3.2.1 Skyloc™ devices	26
4 V2P App Requirements	29
4.1 V2P Functional Requirements	30
4.2 V2P Non-Functional Requirements	31
4.3 Use Cases	31

4.3.1	Use case: Connection Handling	32
4.3.2	Use case: Send/Clear Notification	32
5	Overview of the V2P app	35
5.1	<i>Widget</i>	37
5.1.1	ACTION_WIDGET_ENABLED	38
5.1.2	ACTION_APPWIDGET_UPDATE	38
5.1.3	ACTION_WIDGET_DISABLED	38
5.2	ReceiveService	39
5.2.1	ConnectThread	41
5.2.2	ReceiveThread	41
5.3	BluetoothStateReceiver	42
5.4	LocalNotificationManager	42
5.5	The Settings Activity (MainActivity)	44
5.5.1	StatusFragment	44
5.5.2	RecycleSettingsActivity	45
5.5.3	DebugSettingsFragment	45
5.6	The Configuration Activity (FirstRunActivity)	46
5.6.1	WelcomeFragment	47
5.6.2	DeviceSelectionFragment	47
5.6.3	RequestEnableOverlayFragment	48
5.6.4	EndOfConfigFragment	49
5.7	Notifications & Screen Overlay	50
5.7.1	Notifications	50
5.7.2	Overlays	51
6	V2P Algorithm	53
7	Design Choices for the V2P App	57
7.1	Hardware choices	57
7.1.1	Bluetooth	57
7.1.2	The infrastructure	58
7.2	Software choices	58
7.2.1	Android OS support	58
7.2.2	Home Screen Widget vs. auto-starting Activity	59
7.2.3	The debug screen	59
7.2.4	V2P algorithm & Pythagorean theorem	59

8	V2P Testing	61
II	V2X HMI application	63
9	Vehicle-to-Vehicle	65
9.1	Vehicle-to-Vehicle Communication	65
9.1.1	V2X Use Cases	66
9.2	Test methodology	70
9.3	A note about the images for the use cases	70
10	V2V Architecture	71
10.1	Hardware Architecture	71
10.2	Software Architecture	72
10.3	The Instrument Cluster of the Ego vehicle	73
10.3.1	Functioning	75
11	V2X HMI App Requirements	81
11.1	V2X HMI Functional Requirements	82
11.2	V2X HMI Non-Functional Requirements	82
11.3	V2X HMI Use Cases	82
11.3.1	V2X HMI Use Case: Connection Handling	83
11.3.2	V2X HMI Use Case: Display V2X Events	84
11.3.3	V2X HMI Use Case: Display Logs	85
12	Overview of the V2X HMI app	87
12.1	MainActivity	87
12.1.1	CockpitFragment	89
12.1.2	SplashFragment	90
12.1.3	LogcatFragment	93
12.2	SettingsFragment	94
12.3	IOService	95
12.4	WifiStateReceiver	95
12.5	LocalV2XEventsReceiver	96
13	Algorithm	97
14	Design choices for the V2X Mirroring App	99
14.1	Communication protocol: TCP	99

14.2	Recycled Bitmap Images	100
14.3	Image and texts maps	100
15	V2X HMI Application Testing	103
15.1	Graphical User Interface Testing	104
15.2	<i>Logcat</i> Testing	104
III	Results	105
16	Results	107
16.1	Further Improvements	108
16.2	Future Developments	108

List of Figures

1.1	Material Design applied to layouts	20
2.1	V2P use case scenario	23
3.1	The Skyloc TM infrastructure	27
4.1	Context diagram of the V2P system	29
4.2	V2P Use case: Connection Handling	32
4.3	V2P Use case: Send/Clear Notification	33
5.1	Main components of V2P application	36
5.2	V2P <i>widget</i>	37
a	Layout	37
b	Red badge	37
5.3	Service flow chart	39
5.4	Threads of the V2P application	40
5.5	ConnectThread flow chart	41
5.6	ReceiveThread flow chart	41
5.7	V2P <i>Widget</i> (a) with and (b) without badge	43
a	<i>Widget</i> with badge	43
b	<i>Widget</i>	43
5.8	Views of MainActivity	45
a	StatusFragment	45
b	DebugFragment	45
c	DeviceSelectionFragment	45
5.9	FirstRunActivity layout	46
a	StatusFragment	46
b	DebugFragment	46
5.10	WelcomeFragment layout	47
5.11	Screens for device selection	48

a	Bluetooth OFF	48
b	Select anchor device	48
c	Anchor device selected	48
5.12	Final screens for configuration	49
a	Bluetooth OFF	49
b	Select anchor device	49
5.13	V2X not yet configured	50
5.14	V2X Available	50
5.15	V2X Not Available	51
5.16	Car Approaching	51
5.17	Overlaying screen	52
6.1	Example of message sent by the <i>anchor</i>	53
6.2	Service flow chart	56
9.1	Control Loss Warning event	66
9.2	Emergency Electronic Brake Light event	67
9.3	Forward Collision Warning event	68
9.4	Left Turn Assistant event	68
9.5	Intersection Movement Assist event	69
9.6	Stationary Vehicle event	69
10.1	Hardware architecture of the V2X technology on the Ego vehicle	72
10.2	Software architecture of the V2X Communication Board	73
10.3	Instrument Cluster of Ego vehicle	74
10.4	Display modes on Ego vehicle	74
a	Mode: grid	74
b	Mode: popup	74
10.5	V2X_INFO_1 Message	75
10.6	Images for the V2X state	76
a	State: <i>V2X Available</i>	76
b	State: <i>V2X Not Available</i>	76
c	State: <i>V2X Not Present</i>	76
10.7	V2X_INFO_2 Message	77
10.8	Three icons for <i>Stationary Vehicle</i> use case	77
a	<i>Stationary Vehicle</i> UC: left	77
b	<i>Stationary Vehicle</i> UC: center	77
c	<i>Stationary Vehicle</i> UC: right	77

10.9	V2X_INFO_3 Message	78
10.10	Two icons for the <i>Front Collision Warning</i> use case: (a) yellow lines and background for the low criticality alert; (b) red lines and background for the high criticality warning.	79
a	<i>Front Collision Warning</i> UC: low criticality	79
b	<i>Front Collision Warning</i> UC: high criticality	79
11.1	Context diagram of the system	81
11.2	V2X HMI Use case: Connection Handling	84
11.3	V2X HMI Use case: Display V2X Events	85
11.4	V2X HMI Use case: Display Logs	85
12.1	Elements composing the V2X HMI application	88
12.2	App's Navigation Drawer	88
12.3	Views of CockpitFragment : Grid	89
a	Grid view on V2X HMI application	89
b	Grid view on dashboard	89
12.3	Views of CockpitFragment : Popup	90
c	Popup view on V2X HMI application	90
d	Popup view on dashboard	90
12.4	Layout bounds for grid and popup	90
a	Grid Layout	90
b	Popup layout	90
12.5	Views of SplashFragment	91
a	Wi-Fi Disabled	91
12.5	Views of SplashFragment	92
b	App not yet configured	92
c	No Wi-Fi networks available	92
d	Remote peer seems to be offline	92
12.5	Views of SplashFragment	93
e	Connecting	93
12.6	LogcatFragment	93
12.7	Log priority filters	94
12.8	V2X HMI <i>Settings</i> screen	94
12.9	Threads of the V2X HMI application	95

Chapter 1

Introduction

1.1 The Company

Magneti Marelli S.p.A. is an international company committed to the design and production of hi-tech systems and components for the automotive sector in the following business areas: Electronic Systems, Automotive Lighting, Powertrain, Suspension Systems, Exhaust Systems, Motorsport, Plastic Components and Modules and After Market Parts and Services[21].

The plant in Venaria Reale, Italy, also have a research department called Technology Innovation, which is composed by four groups, each committed to a specific mission.

1.1.1 Innovation & Connectivity

Innovation & Connectivity is one of the four groups of Technology Innovation and its mission is to analyze all the aspects concerning connectivity from a 360 degrees point of view.

Currently the team is committed to the development of a vehicular On-Board Unit implementing the V2X DSRC technology, which allows equipped vehicles to communicate to each other for increasing the drivers' awareness of possible road hazards that may be out of line-of-sight[25].

Alongside V2X DSRC, the team is also investigating new connectivity technologies like 5G, in order to design the evolution of V2X DSRC, Cellular-V2X (CV2X).

1.2 C-ITS and V2X

Nowadays road safety is mostly managed by stand-alone driver assistance, which helps drivers to maintain safe behaviors at the wheel[15]. However, road safety could be further increased if vehicles were able to communicate with each other and with the road infrastructure.

Cooperative Intelligent Transport Systems, C-ITS, represent the next revolution in this sense: through digital connectivity, vehicles will be able to communicate and share information to coordinate their actions, improving road safety and traffic efficiency[16]. The cooperation among the road entities (i.e. vehicles, infrastructure and other road users) is also the intermediate step for allowing the further full integration of future automated vehicles in the overall transport system.

For what concerns the automotive field, the communication between vehicles and other road entities is called Vehicle-to-Everything or V2X. This technology provides drivers with a 360 degrees awareness of similarly equipped vehicles within a range of about 300 meters[25].

Thanks to a wireless Dedicated Short Range Communication, DSRC, each vehicle can broadcast its own data to nearby vehicles, which use that information to identify potential hazards and warn drivers to avoid imminent crashes[25].

The National Highway Traffic Safety Administration of the United States estimates that Vehicle-to-Everything could reduce “*the severity of up to 80 percent of non-impaired crashes, including crashes at intersections or while changing lanes*”[24].

In order to communicate, vehicles must be equipped with a radio interface, called On Board Unit (OBU), which allows them to access the network created by other similarly equipped vehicles and Road Side Units (RSUs) such as, for example, intelligent traffic lights. Such a network is usually referred to as Vehicular Ad-hoc NETwork (VANET) and is based on decentralized nodes, represented by vehicles and Road Side Units. This means that the infrastructure is not fixed, since vehicles are continuously moving, hence traffic must be routed by the nodes themselves[3].

Currently VANETs rely on dedicated short range communication technologies, such as IEEE 802.11p, but other access methods are being investigated by research institutions, universities and automobiles manufacturers.

The most promising path for new communication technologies is represented by 5G, which would be a more reliable and efficient technology for the realization of Cellular V2X. C-V2X is the evolution of V2X based on 802.11p and exploits the cellular network to extend the range of awareness of road entities up to one mile[14].

Moreover, since the 5G provides higher throughput, vehicles will not only be able to

share simple messages, but also other kinds of information, from data collected by sensors, radars and LIDAR to map or multimedial data sharing[14].

1.2.1 V2X Standards and Legislation

Currently, V2X communication is mostly described by two main standards: ETSI and WAVE, which respectively serve Europe and United States.

Both standards rely on a wireless transport layer especially designed for enabling connectivity in vehicular environments: IEEE 802.11p (called ITS-G5 in Europe). The two standards are quite similar because the European one is a variant derived from US WAVE, adapted to Europe's requirements[17] but, for what concerns the work done for this paper, the differences between the European and American standard are not important as they are managed by the modules on Magneti Marelli's V2X Communication board and do not affect the mobile applications.

At the moment, the V2X technology is not ruled by any legislation as it is still evolving. However, the U.S. National Highway Traffic Safety Administration is trying to propose a rulemaking standard for what concerns motor vehicle safety, in order to force all the car manufacturers to install DSRC systems on all new vehicles, starting from 2020[18].

Such a proposed standard also states how those systems have to implement security and privacy protection: V2X may store lots of personal information and vehicular trajectory data, which, if not correctly protected, could disclose the users' habits and traces. A system that does not protect the user's privacy is hardly accepted by the majority of people[3].

For this reason, the IEEE 1609.2 standards states that DSRC shall provide authentication and optional encryption of messages, based on digital signatures and certificates[14]. Such certificates however are frequently changed and do not contain driver's data, in order to make tracking more difficult[14].

1.3 Goals of this thesis

The goal of this thesis is the development of two applications for bringing the Vehicle-to-Everything communication on Android devices.

The research team is developing the prototype and is also trying to expand the usages of the V2X in order to encompass other entities, such as for example, pedestrians.

Part I of this work is dedicated to this specific type of V2X communication, which is called Vehicle-to-Pedestrian or V2P, and, after a brief introduction to the topic, it will illustrate the design and development details of the related application.

Part II illustrates the design and development of an application concerning the most important type of V2X communication: Vehicle-to-Vehicle, V2V. The goal of this application is to reproduce, on a mobile device, the behavior of the dashboard of a vehicle mounting Magneti Marelli's V2X communication board. For a better understanding of the mechanisms at the base of the application, a chapter will be dedicated to the description of the interactions between the board and the dashboard of the vehicle.

The third and last part of this work reports the overall results and the possible further expansions of both applications.

1.4 Tools

Beside code, the projects of this thesis required the creation of images both to be used as icons and backgrounds in the applications and to illustrate some V2X use cases during internal demonstrations.

Android Studio

Android Studio is the official IDE for developing Android applications. Beside the base functionalities provided by common integrated development environments, it also provides a layout editor, for building Graphical User Interfaces in a what-you-see-is-what-you-get way, an Android emulator and a profiling tool, for real-time monitoring of CPU, memory and network usages[11].

Adobe Illustrator

Illustrator is a program of the Adobe Suite that is meant for creating and editing vector images.

Vector images easily allow to generate the set of scaled images the operating system

needs for correctly displaying the interface on the different screen sizes and resolutions of the devices running Android[4].

Indeed, starting from a vector image, i.e. an icon, it can be scaled up and down without losing quality.

Blender

Blender is an open source software for managing the whole 3D pipeline, from modeling, to rendering and compositing[12].

Blender has been used for the renderings of the images used in the demonstrations, but also for the photo-realistic images used in both applications.

The 3D models have been mostly created specifically for the renderings, while some have been found on the Internet.

1.5 Development methodology: Agile

The projects of this thesis have been developed by following the Agile approach, which requires close cooperation between the customer and the development team. The goal is to organize the process in order to periodically deliver a working version of the product, starting from a prototype implementing basic features and gradually refining the solution. This allows the requirements to be defined or modified based on priority during the development process, by adding more and more details[29].

Since the applications developed for this thesis are intended for internal use, the customer was represented by the researchers of the Innovation & Connectivity team, who defined the requirements mostly through user stories¹ and periodically reviewed both applications to report problems.

The exchange of information was done mostly face to face, but also via conference-call with the colleagues of the team at Magneti Marelli in Auburn Hills, USA, for what concerns the V2P application.

The whole development process was based on two Agile strategies: product backlogs and Exploratory testing. The first one lists, according to a priority, the required changes and the known bugs that need respectively to be implemented and fixed. The second one carries out the test of the application by having the tester using it both in the standard way and also in a clever one, in order to break the software and disclose misbehaviors[26].

Graphical user interfaces fall in the umbrella of interaction design, which deals with the design of the graphical layout and the behavior of the graphical elements when they are being used. Interaction design requires feedback from the end users in order to test usability and can be integrated in the Agile development with some difficulty. In fact it assumes that requirements are not being changed any more, but this does not happen in Agile[28]. For this reason, for both applications, most of the graphical interface has been designed before starting the implementation of the code, making little changes after each further delivery.

¹User stories represent an informal way for defining requirements starting from few details written from the stakeholder's perspective, which are going to be refined in further iterations[1][30].

1.6 Graphical User Interface

Graphical User Interface deserves special attention, because is one of the key element that establishes success for an application. In fact, a product whose interface is not tailored for its target user and does not provide a good user experience, is likely to be quickly left behind.

The applications designed for this thesis are not intended for the big audience of Google Play Store, as the infrastructure they rely on is still a prototype and it is not yet on the market. Yet, this does not exempt the application from compliance with UI design and usability guidelines.

1.6.1 Usability

The ergonomics of human-computer interaction standard, ISO 9241-11, defines usability as “*the extent to which a product can be used by specified users to achieve specified goals with effectiveness, efficiency and satisfaction in a specified context of use*”[20].

Usability studies take into account the human psychology for creating products that are easy to use, other than beautiful to see. In other words, user interfaces must be designed in a way that allows the user to quickly and easily learn (and then recall) the sequence of actions required for achieving a specified goal.

Since usability of a product is tied to its specific category of users, the ease of use of the interface is tested through the observation of a group of people trying to complete a specific task. Usually, the test is repeated many times during the development of the product, as the earlier a problem is identified the cheaper is the correction, both in terms of time and work[19].

The applications developed for this thesis have been tested following the above approach, by having the researchers of the team attempting to complete some tasks during all the phases of the development process, since the prototyping.

1.6.2 Material Design

Nowadays mobile devices are widespread and used by a great number of people having different levels of expertise. Mobile application must consider this fact and provide interactions and graphical layouts that accomodate the majority of the users.

Material Design is a collection of guidelines for creating simple, clean and intuitive interfaces for different platforms and device sizes.

Material Design helps the application to implement usability since it is based on the material metaphor, inspired by paper and ink: surfaces, edges, lights, shadows and

movement are visual cues that recall the patterns of physics reality, leading the user to understand how the layout components move and interact with each other[23]. The hierarchies within the layout are also highlighted by specific color choices combined with white spaces and typography[23].

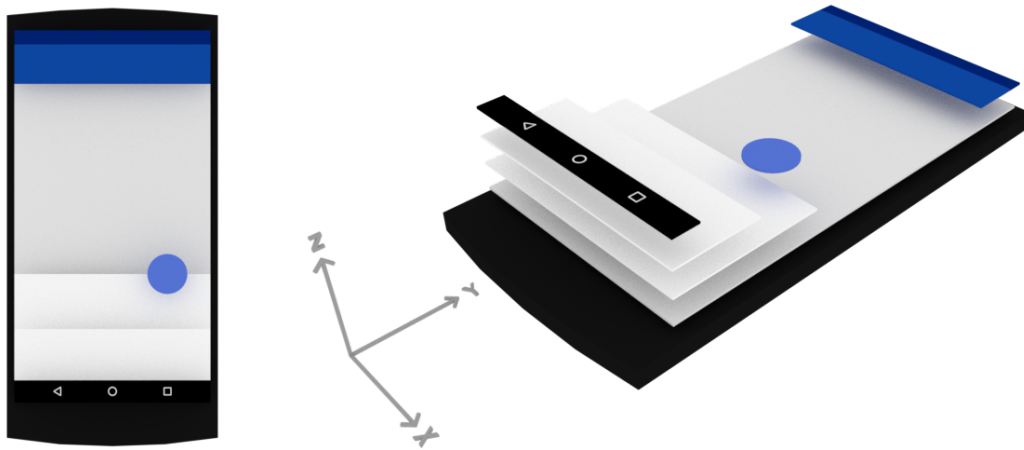


Figure 1.1: The layout components are placed in a three-dimensional space as if they were built and assembled in the real-world[22]

Part I

V2P application

Chapter 2

Vehicle-to-Pedestrian

Vehicle-to-pedestrian, V2P, technology is a more-specific type of V2X communication that focuses on vehicles and Vulnerable Road Users such as pedestrian, bicyclists, people using wheelchair, etc.

This kind of communication is bidirectional and aims at notifying both parties about each others' presence through the dashboard, on vehicles, and through a wearable or handheld device for non-motorized road users, in order to increase road safety by reducing the number of fatalities due to poor visibility.

A typical use case scenario concerns a person, who is going to cross the road in front of a bus idling on the stop. Due to the presence of the bus, such a person is hidden from view for the vehicles incoming in the same travel direction, thus creating a dangerous situation both for the pedestrian and for the vehicles that may hard brake and possibly incur in rear-end collisions.

The goal of the first part of this thesis is to develop a smartphone application that is able to connect to the V2X network and warn the pedestrians against the presence of vehicles near them. It is supposed that the vehicles implement the dual counterpart of this application in order to warn the drivers against the presence of pedestrians so that they can adjust the speed and be ready to brake.

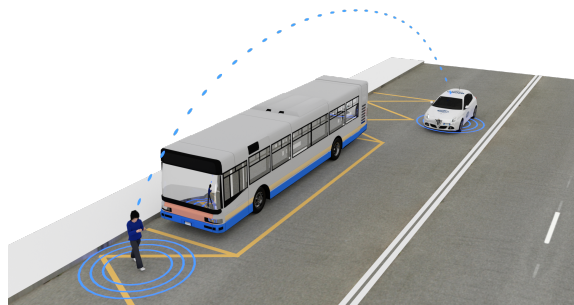


Figure 2.1: V2P use case scenario

Chapter 3

Architecture

This chapter will briefly introduce the architecture of the V2P system in order to describe how it works and how it has been customized for meeting the goals of this paper.

3.1 V2P Architecture

Vehicle-to-Everything is characterized by a distributed architecture, that means each peer receives the messages of all the other nodes and broadcasts its own information.

In order to have the pedestrian-entity be compliant to this paradigm, Magneti Marelli designed an electronic device that works as plug-in extension for the smart-phone, providing the connectivity towards the V2X control units installed on vehicles. The device appears as an additional shell enveloping the phone and it contains a little electronic board that is connected to the phone through the latter's USB-C port and sends it the data received from other V2X entities.

Such data is then processed by the V2P application running on the phone and, when necessary, a notification or warning is sent to the user.

As other V2X entities, also the V2P board attached to the phone communicates via DSRC, which requires GPS signals in order to provide the other entities' locations.

The V2P technology was one of the innovations that were going to be presented in January at Magneti Marelli's booth at the Consumer Electronic Show 2018 in Las Vegas, USA, hence, for demonstration purposes, GPS could not be used because of the shielding of the signal and the small distances of the indoor environment.

For the occasion the V2P Communication board has been replaced with a custom

solution able to provide a good indoor positioning. The choice fell on Skypersonic's¹ Skyloc™ system, which provides indoor triangulation through a set of little electronic devices, *tags*, that communicate with each other via Ultra-Wide Band (UWB). The software loaded on the *tags* is customized for transmitting CSV-formatted messages to other devices either via UWB or Bluetooth.

3.2 Customized V2P Architecture

The implementation with the Skyloc™ system required some changes to the V2P architecture. The most important one is that the whole system becomes centralized: the real-time location is performed by a special *tag* called *anchor*, which collects the data transmitted by all the *tags*, computes their position with respect to itself and sends a custom message via Bluetooth to a paired device: the pedestrian's smartphone. Due to the kind of Bluetooth module attached to the *anchor*, the only type of communication allowed is one-to-one instead of one-to-many, which means that at any time there can be at most one device connected to the master.

Of course, this limitation would be unreasonable for a real-use system but, for the purposes of the demonstration, it was not a problem since Magneti Marelli was equipped with two smartphones, used one at a time for battery efficiency.

3.2.1 Skyloc™ devices

To summarize what has been written before, the Skypersonic's indoor positioning system is made of a set of *tags*, which share the same hardware but provide different functionalities depending on the software loaded into memory. The two most important behaviors are:

- *anchor*, the center of the whole system. It is the one that collects all the data sent by the *tags* attached to moving entities, i.e. smartphone, vehicle, and builds the output messages to be transmitted via Bluetooth to the paired device every 90 ms. Being the central element, the *anchor* also represents the origin of the coordinate reference system for real-time location, so its location is $(0,0,0)$.

The *anchor* is able to triangulate the position of all the simple *tags* thanks to two

¹Skypersonic is a company specialized in designing and producing drones and real-time location systems for indoor applications[31]

special *tags* called “eyes”, which are placed at the corners of Magneti Marelli’s booth.

- simple *tag*, whose position gets tracked by the *anchor*, with precision in the order of centimeters, and communicates with it via UWB.

Simple *tags* are attached to the moving entities in the V2P system, i.e. pedestrians and vehicles.

For what concerns the booth at the CES2018, such entities were represented by the shell enveloping the smartphone, carried around by a person simulating the pedestrian, and a Chrysler Pacifica, which actually did not move as it was also used to show innovations in its interior.

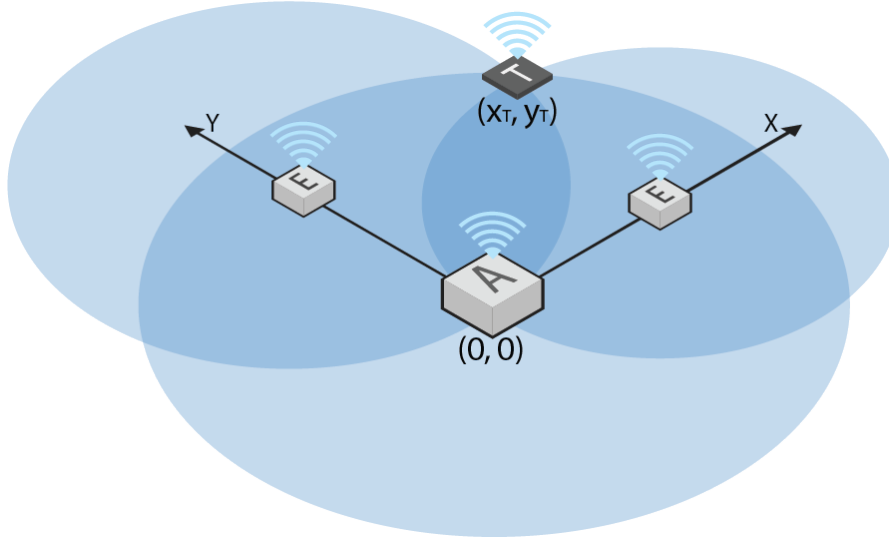


Figure 3.1: The SkylocTM infrastructure. For sake of simplicity, the z coordinate has not been considered.

Chapter 4

V2P App Requirements

Before describing the user requirements, it is necessary to introduce the context in which the application is going to operate and how it shall interact with the external entities depicted in Figure 4.1. The interactions can be of two types: *control flow* and *data flow*. The first one describes the actions that an actor may perform on the system and vice versa; the latter draws the path followed by the data exchanged between the system and the actors. Thanks to such flows it is possible to draw the main requirements: the application shall communicate with the *anchor* and retrieve the data. After some processing, if certain conditions are met, a notification is sent to the user.

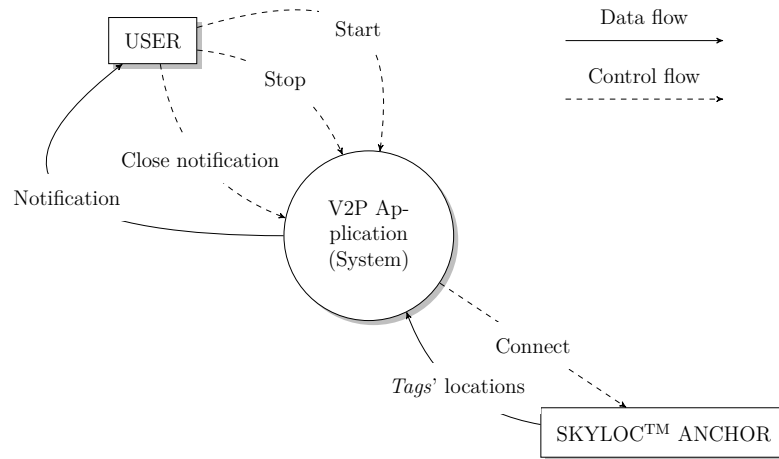


Figure 4.1: Context diagram of the V2P system

The next two paragraphs will describe more in details what the application should

do and how it should behave, but, before proceeding, it is important to notice that the only SkylocTM entity communicating with the application is the *anchor*. In fact, as previously said, it represents the center of the customized V2P infrastructure and it is the one that collects the data about all the *tags*.

4.1 V2P Functional Requirements

The functional requirements, i.e. what the system should be able to do[27], are listed in tables 4.1 and 4.2: the first one concerns the interaction between the system and the user, the second one describes the communication between the system and the *anchor*. The two tables only describe the most important functional requirements, which are the ones that have been defined through the user stories.

Function	Warn the user whenever a possible hazardous road event occurs
Description	Send a notification to the user whenever a vehicle is approaching and its distance from the user is below a certain threshold
Input	Formatted string containing the positions of the <i>tags</i> assigned to the pedestrian's phone and to the vehicle
Source	Skyloc TM <i>anchor</i> via Bluetooth
Output	Issue a warning notification to the user about the incoming vehicle
Action	The user is notified if and only if the distance between the <i>tag</i> assigned to the vehicle and the one assigned to the phone is smaller than a certain threshold and he/she has not already dismissed a notification about the same vehicle. If the distance increases and becomes greater than the threshold while the notification is still up, clear it
Requires	The current distance between the <i>tags</i> and the old one
Pre-condition	The received string is correctly formatted
Post-condition	The new distance replaces the old one if it is below the threshold, otherwise the old distance is reset

Table 4.1: V2P Functional Requirements for the interaction between the system and the user

Function	Retrieve data from the Skyloc TM <i>anchor</i>
Description	Connect to the <i>anchor</i> via Bluetooth and keep listening for incoming messages containing the real-time locations of both <i>tags</i>
Output	Values extracted from the string
Action	Once the connection has been established, keep listening for string messages If connection is dropped, try reconnecting until success
Pre-condition	Bluetooth is enabled, the phone has already been paired to the <i>anchor</i> , and the latter is within the Bluetooth range

Table 4.2: V2P Functional Requirements for the interaction between the system and the *anchor*

4.2 V2P Non-Functional Requirements

Table 4.3 lists the non-functional requirements, which state how the system should achieve the result, by also considering the user interface and the general performance.

NF1	User eXperience (UX)	The application must require little interaction with the user
NF2	UX	Notification shall catch the user's attention even if the mobile device is stored into a bag or a pocket, hence it must use vibration, sounds and lights (if available)
NF3	UX	Vibration shall not be turned off by the user
NF4	UX	When the screen is interactive, the warning about the incoming vehicle should visually interrupt what the user is doing, by dragging a screen over the application currently in foreground
NF5	Connectivity	Connection issues shall be automatically handled by the application

Table 4.3: V2P Non-functional Requirements

4.3 Use Cases

In order to better define how the application should behave, two use cases are presented. They will mainly consider the normal flow followed on normal use of the application, when everything goes well and no exceptions or errors occur.

Each use case, however, will also introduce the possible alternate flows that generate from issues.

4.3.1 Use case: Connection Handling

This use case is triggered when the user enables Bluetooth on the smartphone and, of course, if the application has already been started.

In the basic flow the application succeeds in connecting and starts to receive messages.

If errors or exceptions occur during this process, the use case falls into an alternative flow, which usually warns the user and then handles the issue.

In order to have the centralized architecture be invisible, connection issues like out-of-range or connection lost must be automatically handled by the application, without requiring any action by the user. This means that each time the connection is dropped for some reason other than Bluetooth disabled, the application must try to reconnect until success.

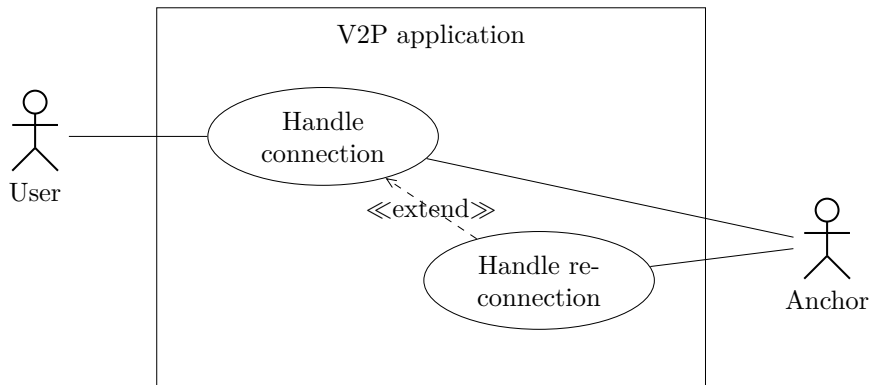


Figure 4.2: V2P Use case: Connection Handling

4.3.2 Use case: Send/Clear Notification

As soon as the system receives a string from the *anchor*, it analyzes the positions of the two *tags* and, if they are below a certain threshold, the notification is sent to the user.

The notification must capture the attention of the user, even if the phone is stored in a pocket or in a bag, so it must make use of vibration, sounds and visual effects.

In the basic flow, the warning notification is issued and the user taps the screen to agree and clear it.

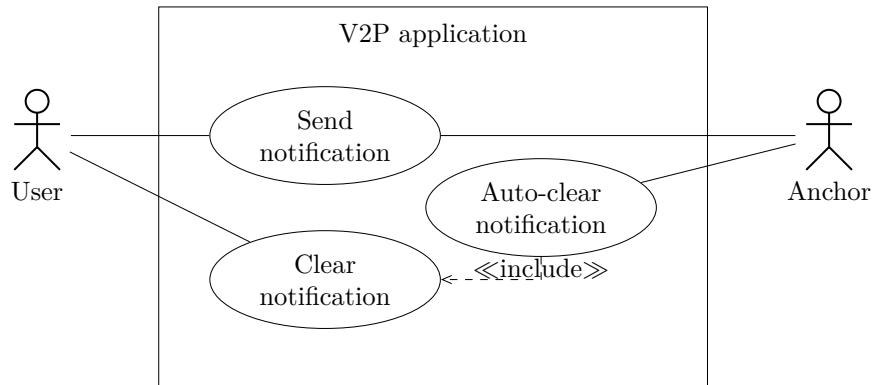


Figure 4.3: V2P Use case: Send/Clear Notification

An alternative flow concerns the case in which the event disappears, i.e. the vehicle moves away from the user without the latter having cleared the notification. In this case the notification clears itself as the event is no more interesting.

Another possible flow is represented by the user stopping the application right when a notification is issued. This is a rare case but it must be managed as well in order to avoid a dangling notification that can never be cleared.

Chapter 5

Overview of the V2P app

This chapter, after some brief considerations about the tools provided by the Android APIs, will present the main components of the application, along with their Graphical User Interface, describing how they are connected to each other in order to satisfy the requirements presented in the previous chapter.

The application has been developed as a *home screen widget* provided with two activities for first-run configuration and settings.

A *home screen widget*¹ can be described as a summarizing view of the most important features offered by an application directly on the Android Home Screen of the smartphone[32][9].

Since a *widget* is a summary, it cannot exist as standalone element without an underlying application as the only action it is allowed to perform is the update of its graphical appearance. Moreover, any action performed by the *widget* must necessarily complete within five seconds, otherwise the Android operating system will kill it automatically[32].

For these limitations, heavy computation and other potentially long-taking operations, such as the ones concerning the Bluetooth socket, have been moved into a **Service**.

From the point of view of Android OS, a *home screen widget* is a **BroadcastReceiver** that captures the action `android.appwidget.action.APPWIDGET_UPDATE` and inflates its layout into a **RemoteViews** object, which is then delivered to the operating system

¹According to the Android documentation, *home screen widgets* are different from *widgets*: the first one are "at-a-glance" views of the most important data and functionality of an application[9], while *widgets* are the objects of class **View**, which are used to build Graphical User Interfaces. To avoid confusion, this text is going to call *widget* the *home screen widgets*, dropping the usage of the word *widget* for referring to **View** objects.

and placed on the home screen[32].

The *widget* can also work as a listener for clicks, so if the user taps it, the *Settings Activity* is launched. From there the user can see the state of the connection towards the *anchor* and change the settings about the notifications.

From the *Settings Activity* is also possible to reach a hidden menu which has been developed for debug purposes and shall not be used by common users.

Each time the *Activity* is launched, an if statement checks whether the V2P application has already been configured, otherwise a first-configuration *Activity* is launched to help the user in the set up through a step-by-step wizard.

Besides the *widget*, there are two more *BroadcastReceivers*, whose goal is to listen for Bluetooth state changes, i.e. *enabled*, *disabled*, etc., and for *intents* driving the notifications, i.e. *issue a notification*, *clear notification*, *show overlay*, *hide overlay*, etc.

The following paragraphs are going to describe the main elements that compose the application. Figure 5.1 shows the connections between such elements in order to give a clearer overview of the whole system.

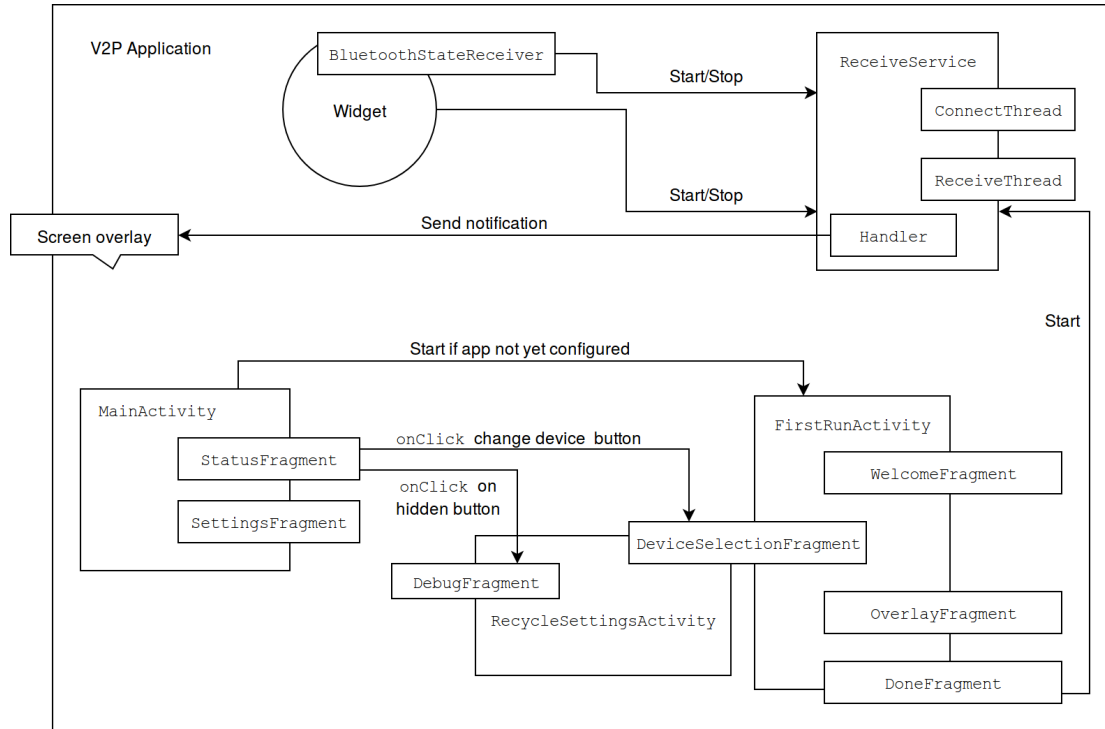


Figure 5.1: Main components of V2P application

5.1 *Widget*

The *widget* is the most important part of the application since it is the one that enables the V2P connectivity on the phone: when the user drags it on the home screen, the `BroadcastReceivers` for Bluetooth state changes and notification events are registered.

As previously introduced, the *widget* itself is a `BroadcastReceiver` and it reacts to actions regarding its graphic layout being dragged on the screen.

The Android operating system allows multiple instances of a *widget* to be placed on the screen, but, for this application, this functionality is useless, as the connection towards the V2X infrastructure is at most one. For this reason, the only actions that are captured by the V2P *widget* are the ones concerning the first instance dragged on the screen, `ACTION_WIDGET_ENABLED`; the last one that is removed, `ACTION_WIDGET_DISABLED`; and the update of the graphic layout of all the instances, `ACTION_APPWIDGET_UPDATE`. The following paragraphs better describe the behavior of the *widget* based on the kind of captured action.



(a)



(b)

Figure 5.2: (a) The layout of the V2P *widget* , (b) the V2P *widget* (with the red badge) as it appears on the home screen

5.1.1 ACTION_WIDGET_ENABLED

As previously introduced, this action is captured when the first instance of the *widget* is dragged on the screen[10].

If the Bluetooth is currently on and the application has already been configured, the connectivity `Service` is started. Otherwise, the *widget* issues a local broadcast event to signal the absence of V2P connectivity to the `LocalNotificationManager`, which will issue a notification and put a badge on the *widget* for warning the user about the issue.

If the application was not yet configured when `ACTION_WIDGET_ENABLED` was captured, the badge is shown on the layout and the action associated to the click on the layout is toggled to head to the *Configuration Activity*, so, whenever the user taps the *widget*, the *Configuration Activity* is launched.

The `ACTION_WIDGET_ENABLED` broadcast is also sent each time the phone is rebooted, therefore it is the action that provides the persistence of the V2P functionality, without requiring the user to manually enable the application each time the phone is turned off.

5.1.2 ACTION_APPWIDGET_UPDATE

`ACTION_APPWIDGET_UPDATE` is captured each time a layout refresh is required[10].

For what concerns the V2P application, a refresh is required each time the badge must be shown or hidden, that means each time the connection is lost or gained or the status of Bluetooth changes between On and Off or vice versa.

The Android OS provides two kinds of triggers for layout updates: a periodic one, occurring after a specific amount of milliseconds, and a “manual” one, issued by a call performed by some other components of the application[10].

The V2P application relays on the second case because there is no reason for having periodic updates, since the *widget* only needs to signal changes about the connectivity state.

Moreover periodic updates are heavy for the operating system, as they wake it up even when the device is sleeping, resulting in a quick battery drain if they are not properly managed[10].

5.1.3 ACTION_WIDGET_DISABLED

This action is captured whenever the last instance of the *widget* is thrown away from the screen[10]. In fact, by throwing away the *widget*, the user means to stop receiving V2P

notifications, therefore the connectivity towards the infrastructure must be stopped based on the current state of Bluetooth.

In the scope concerning this action, both the **BroadcastReceivers** are unregistered, since they are no more needed and would cause a memory leak.

5.2 ReceiveService

The core component that provides connectivity towards the V2P infrastructure is **ReceiveService**, which extends the Android **Service**.

ReceiveService must be started every time there is at least a *widget* instance is on the screen, the Bluetooth is on and the application has already been configured.

Since the goal of the V2P application is to warn the user about a possibly nearby hazardous event, the **Service** must keep running both in memory critical conditions and also when the user is not interacting with the phone.

When the first situation occurs, the Android system automatically selects, among the running applications, a victim to kill in order to free resources[7].

For preventing the system from selecting the V2P application as a victim, the **Service** has been marked as foreground. In fact, foreground **Services** differ from

background ones because they can be marked as victims only if the system is in desperate memory conditions. Moreover they are components which the user is aware of and provide an *ongoing* notification for the status bar.

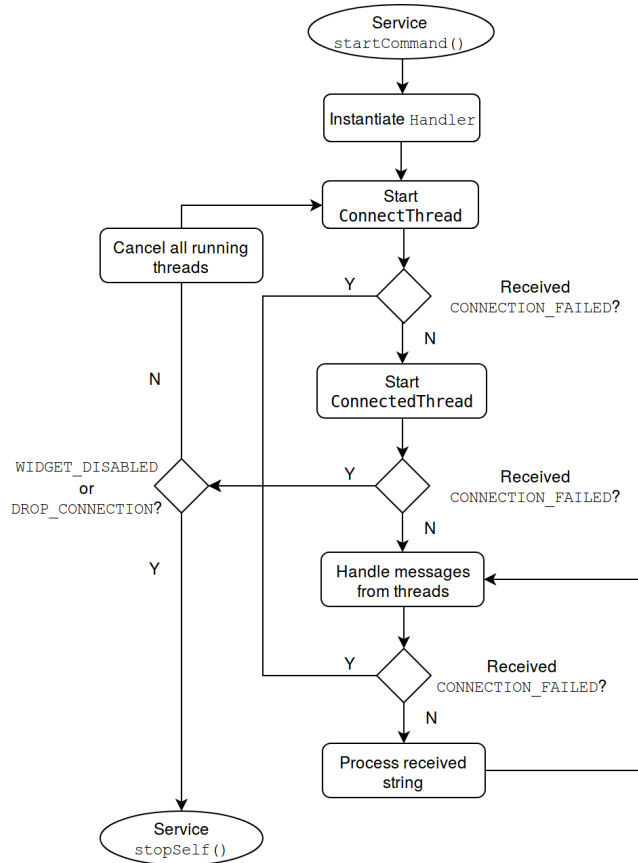


Figure 5.3: Service flow chart

Android **Services** usually run on the main thread hence they cannot be used for performing blocking operations or heavy-computation. For this reason all the actions concerning sockets have been delegated to two separate threads: one for instantiating the connection towards the SkylocTM device, **ConnectThread**, and one for receiving data over the socket stream, **ReceiveThread**. If the connection fails, the corresponding thread sends a **Message** object to the main thread, the **Service**, to communicate the issue before stopping and getting garbage collected.

The main thread collects all the **Messages** sent by the secondary threads with the help of a **Handler** object, which allows the inter-thread communication through a **MessageQueue**.

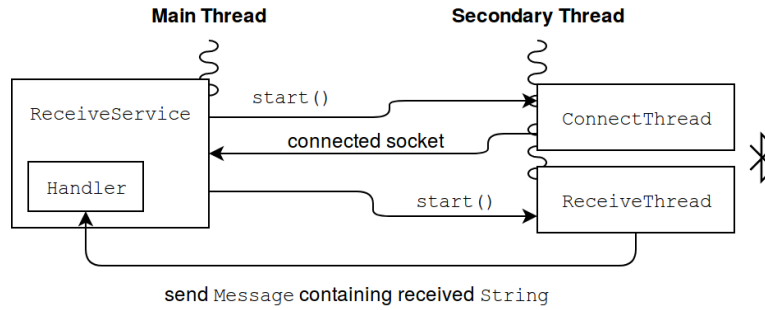


Figure 5.4: Threads of the V2P application

Such a **Handler** can manage two kinds of **Message**²: one containing the data received from the socket stream, **HANDLE_MESSAGE**, and one containing the state of the connection and possible issues, **HANDLE_CONNECTION_FAILED**. The first one processes the CSV-formatted **String** contained in an extra field of the **Message** object and determines if the notification should be triggered. The algorithm that drives this choice will be described in details in the next chapter.

The latter type is used for issuing the automatic reconnection after an **IOException** occurred on the socket.

However, since the **Service** can be stopped from other components, such as **Widget** and **BluetoothStateReceiver**, the secondary threads must stop as well, without sending any **HANDLE_CONNECTION_FAILED Message**. In fact, if the user throws away all the instances of the *widget* or if Bluetooth is turned off, all the resources used by the **Service** and its threads must be released.

²Actually what distinguishes a **Message** from another is the user-defined code contained in the **what** field of the object[5]. In this case, **what** can assume two values: **HANDLE_CONNECTION_FAILED** and **HANDLE_MESSAGE**

5.2.1 ConnectThread

`ConnectThread` is the thread that is in charge of starting the connection towards the device.

This is a blocking operation since the `connect()` method implements a select, which has the thread block until the connection succeeds or until the timeout (12 seconds) expires. If any exception occurs, `ConnectThread` closes the socket and, before, terminating, checks whether the issue is due to the user having thrown away the *widget* or turned off Bluetooth. If any of the two conditions is met, the thread simply terminates. Otherwise, it sends a `Message` to the main thread's `Handler` with `HANDLE_CONNECTION_FAILED` code.

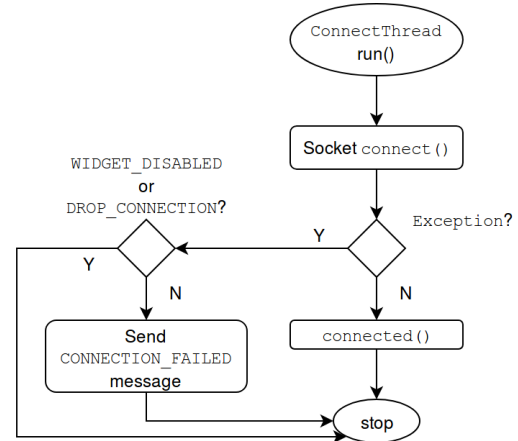


Figure 5.5: `ConnectThread` flow chart

5.2.2 ReceiveThread

`ReceiveThread` is the thread that reads data from the `InputStream` and passes the received `String` to the main thread as an argument of the `HANDLE_MESSAGE` `Message`.

The reading process can be interrupted if the counterpart goes offline or if the user throws the *widget* away, turns off Bluetooth or changes the *anchor* device's address. In order to decide how to behave, the thread checks whether the variable storing the connection state equals either to `WIDGET_DISABLED` or to `CONNECTION_DROPPED`. These states respectively address the Bluetooth Off case and the *widget* thrown away one. If none of the two conditions is met, then the connection should be re-established, hence the thread sends a `HANDLE_CONNECTION_FAILED` `Message` to the `Handler`.

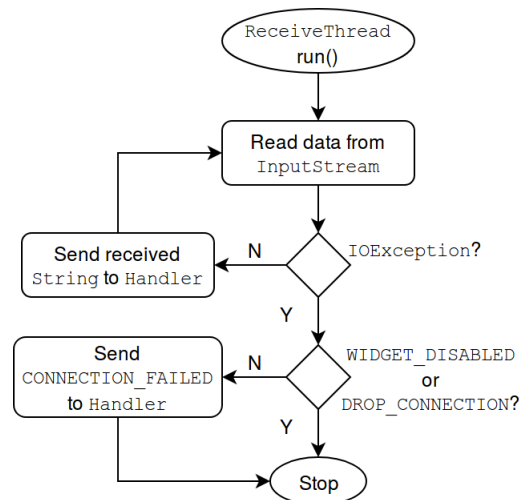


Figure 5.6: `ReceiveThread` flow chart

5.3 BluetoothStateReceiver

`BluetoothStateReceiver` is a `BroadcastReceiver` that is dynamically registered³ when the first instance of the *widget* is dragged on the home screen. The two Bluetooth broadcasts captured by this receiver are:

- `STATE_ON`, broadcast when the Bluetooth is turned on.
This broadcast is not sticky, hence it is captured by `BluetoothStateReceiver` only if it is issued after the *widget* has been dragged on the screen.
As soon as `STATE_ON` is captured, an if statement checks whether the application has already been configured, i.e. the device has already been selected: if it is, then the `ReceiveService` is started.
- `STATE_OFF` broadcast is captured when the Bluetooth is turned off. The `Service` is stopped and a local broadcast is sent to the `LocalNotificationManager`, for having it cancel the ongoing notification in the status bar.

5.4 LocalNotificationManager

`LocalNotificationManager` is a local⁴ `BroadcastReceiver` that also provides static methods for managing the notifications to show on the status bar. This class can capture the following broadcast intents:

- `SHOW_BADGE`. This action sets the visibility of the badge (Figure 5.7a) to `VISIBLE` and triggers the `APPWIDGET_UPDATE` broadcast for refreshing the layout of all the instances of the *widget* on the home screen.
- `HIDE_BADGE`. This action is the opposite of the previous: it sets the badge visibility to `INVISIBLE` (Figure 5.7b) and triggers the `APPWIDGET_UPDATE` broadcast for refreshing the layout of all the instances of the *widget* on the home screen.

³Dynamic registration of a `BroadcastReceiver` means that it is not declared in the Manifest, hence it is instantiated only when the application is alive.

⁴Local `BroadcastReceivers` are intended for exchanging informations among components of an application, without having the intents broadcast in the whole operating system. This ensures more efficiency and security for the application, as a local `BroadcastReceiver` does not react to intents sent by other applications installed on the phone.



Figure 5.7: V2P *Widget* (a) with and (b) without badge

- **SHOW_OVERLAY**. This action is captured when a vehicle is close to the user and the notification must be shown in the status bar, along with a screen that is dragged on whatever application currently on foreground.

The overlaying screen must be explicitly authorized by the user in the application's page in Android Settings as it can be very annoying, if not properly managed by the application.

When a **SHOW_OVERLAY** broadcast is captured, the notification text and icon are changed respectively to "*car approaching*" and to the car icon. If the permission to drag overlays was not granted, the notification is marked as clickable, in order to allow the user to clear the notification and restore it to the previous value.

The **SHOW_OVERLAY** broadcast also carries a **Bundle** with **extras** fields, which provide informations about the current distance between the user and the vehicle.

- **HIDE_OVERLAY**. This broadcast is sent when the V2P event clears itself because the distance between the vehicle and the user has become greater than the threshold.

If the permission to drag screen onto other applications had been granted, the overlay screen is removed and the notification is cleared and restored to the value describing the current status of the service: *V2X available* or *V2X not available*.

- **RESTORE_V2X_NOTIFICATION**. This broadcast restores the notification after an event has been canceled. Depending on the current state of Bluetooth, either On or Off, it sets the notification to *V2X available* or *V2X not available* due to Bluetooth disabled. Normally the notifications can never be cleared by the user, because they are linked to the underlying **Service** and its activity, in fact they are all marked with the *ongoing* flag.

All the above mentioned broadcasts make use of notifications, hence the `LocalNotificationManager` class provides methods for issuing them with different parameters. The most important one regards the vibration, which can be either short or long based on the screen state. If it is interactive then it is probable that the user is currently watching it or even tapping on it, meaning that the vibration shall not be long, as the attention of the user is already on the phone. Differently, if the screen is not interactive, it is probable that the phone is currently stored in a bag or a pocket, hence the vibration shall be longer in order to catch the user's attention.

5.5 The Settings Activity (`MainActivity`)

The *Settings Activity* is the `Activity` that supports the V2P *widget* and is the one that is always called when the user taps the layout of the *widget*. The Graphical User Interface of this `Activity` is a `TabLayout` made of two tabs hosting two `Fragments`: one showing the current status of the connection one for the application *Settings*. Each time `MainActivity` is started, an if statement checks whether the application has already been configured, otherwise the configuration `Activity`, `FirstRunActivity`, is started, pushing `MainActivity` on the backstack.

The following paragraphs are going to illustrate the layout of the two `Fragments` that can be attached to `MainActivity`.

5.5.1 `StatusFragment`

This `Fragment` allows the user to check the current state of the connection and provides two buttons with different levels of visibility: from Figure 5.8a it is possible to see the standard button that allows the user to change the *anchor* device, but, actually also the splash screen can react to clicks. From there, in fact, it is possible to reach a debugging screen where the expert user (i.e. a Magneti Marelli researcher) can tune the parameters of the algorithm that computes the distance between the *tags*. In order to prevent a common user from entering such a screen, the image must be tapped four times before the secondary `Activity` is launched.

The `Activity` launched by both the buttons of the `StatusFragment` is the same, but loads the proper `Fragment` based on an extra field passed with the `Bundle` of the calling `Intent`.

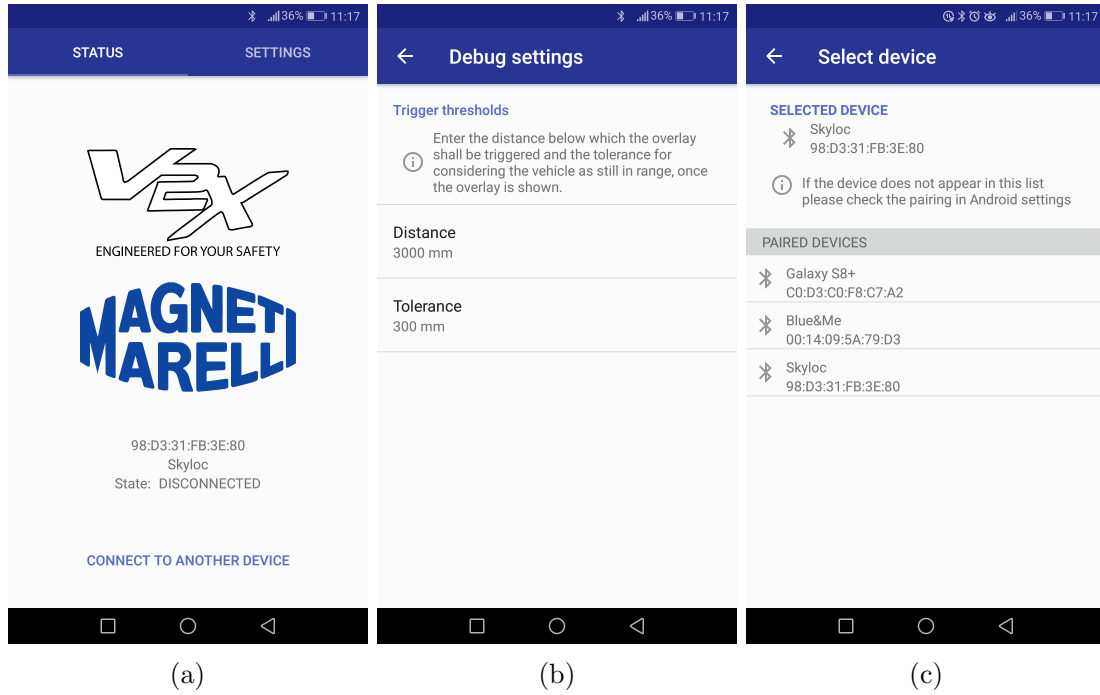


Figure 5.8: (a) the `StatusFragment` attached to the `MainActivity`, (b) the debug screen reached through the button hidden in the splash image, (c) the change *anchor* device screen reached by clicking the button in the `StatusFragment`

5.5.2 RecycleSettingsActivity

`RecycleSettingsActivity` is the `Activity` that act as container for either the device selection `Fragment` or the debugging one. The correct `Fragment` is loaded based on an extra field in the `Bundle` passed with the calling `Intent`.

Since the device selection `Fragment` is the same used for the configuration steps of the application, it will be described together with the `FirstRunActivity` in Section 5.6.2.

5.5.3 DebugSettingsFragment

`DebugSettingsFragment` is the hidden debug screen that can be reached by tapping multiple times the V2X and Magneti Marelli logos depicted in Figure 5.8a in the *Status* tab.

Here the user can tune the parameters driving the algorithm that manages the triggering of the notifications.

The way this screen can be accessed is inspired by the way the easter eggs of Android and the *Developer Mode* are enabled: the user has to navigate in the system *Settings*

and tap multiple times on the right item in the menu list. During this operation, a toast appears to tell the user how many times left the button must be tapped in order to activate the feature.

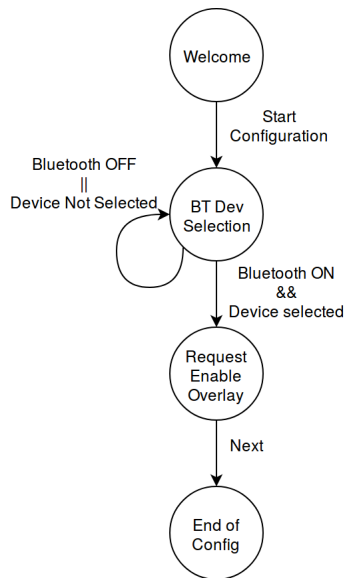
In order to speed up the access to the menu, in the V2P application, the number of taps required to enable the hidden screen is lower than the one required by the hidden features in Android *Settings*. This is to simplify the tuning of the parameters at Magneti Marelli's booth at CES2018.

5.6 The Configuration Activity (**FirstRunActivity**)

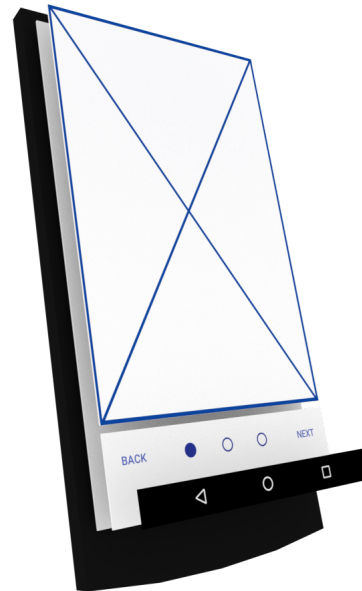
The goal of **FirstRunActivity** is to lead the user through the steps required for configuring the application.

Such steps are represented as nodes in the state diagram depicted in Figure 5.9a and each of them is managed by a specific **Fragment**.

Figure 5.9b illustrates the layout of the **Activity**, where it is possible to notice a sort of navigation bar at its bottom, which has been conceived for telling the user which is the current step of the configuration and how many ones are left.



(a)



(b)

Figure 5.9: (a) Configuration steps. Each node represents a **Fragment**; (b) Layout of **FirstRunActivity**: the bottom bar and the fragment stub

5.6.1 WelcomeFragment

WelcomeFragment (Figure 5.10) is the first screen of the configuration process. It is made of an image, which depicts a classical scene where a user is crossing the road in front of a car equipped with V2P, and a button. When the user taps the button, the **Fragment** triggers the transition to the next screen in the underlying **Activity**.

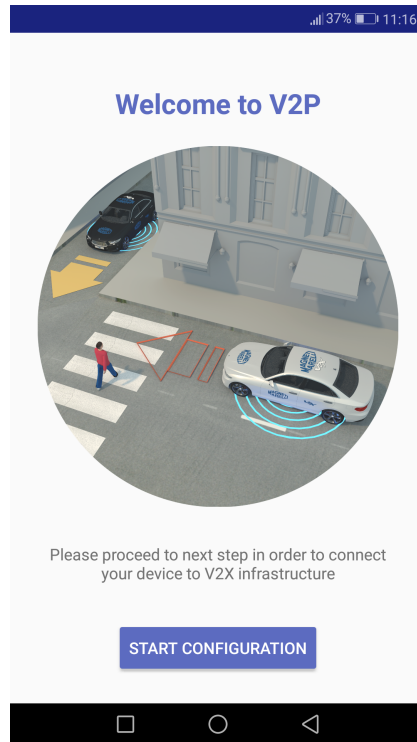


Figure 5.10: Layout of the **WelcomeFragment**

5.6.2 DeviceSelectionFragment

DeviceSelectionFragment is the second screen of the process and asks the user to select the Bluetooth device that acts as *anchor*.

The layout is made of two overlapping **FrameLayout** objects to cover the cases Bluetooth On and Bluetooth Off. As soon as the **Fragment** is attached to the **Activity**, an if statement checks the current Bluetooth state. If it is off, a dialog appears to ask the user to turn it on, and the **FrameLayout** depicted in Figure 5.11a is set to visible. As it is possible to see, a textual description explains the user why the Bluetooth is needed, together with an image depicting the V2P infrastructure.

As soon as Bluetooth is turned on, the **FrameLayouts** are swapped: the one shown in

Figure 5.11a is set to invisible, and the new one is set to visible. The transition occurs smoothly, by having both layouts coexisting with inverted values of transparency for a while. This **Fragment** can also be recalled from the *Settings Activity*, if the user wishes to change the *anchor* device.

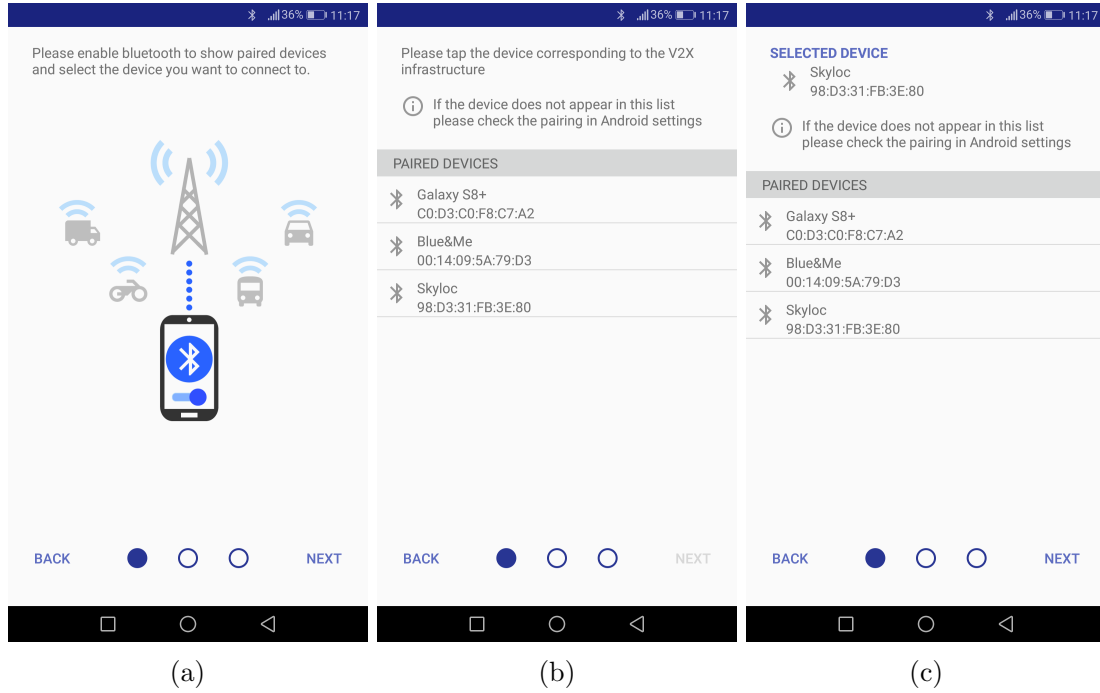


Figure 5.11: (a) **FrameLayout** for Bluetooth disabled, (b) **FrameLayout** for Bluetooth enabled and device not yet selected, (c) *Anchor* device selected and “Next” button enabled

Figure 5.11b shows the **FrameLayout** set to visible when the Bluetooth is on: the devices currently paired with the phone are listed, and the user has to select the one corresponding to the *anchor*.

As soon as the device is selected (Figure 5.11c), the underlying **Activity** is notified and enables the button to proceed to next step.

5.6.3 RequestEnableOverlayFragment

The third step asks the user to allow the application to draw screens over other applications (Figure 5.12a). This feature needs the user to explicitly grant the permission in the system Settings, because of security reasons. As usual, the screen describes the feature together with an image. In order to increase the probabilities that the user grants the permission, there is a button that leads to the Settings menu of the

application, in system *Settings*, where the user just has to tap on the toggle button.

Since this feature can be turned on and off by the user, the application must be able to work in both cases. For this reason, the *Next* button is not disabled but only renamed to *Skip*.

5.6.4 EndOfConfigFragment

EndOfConfigFragment (Figure 5.12b) is the last step of the configuration, and it is the one that sets the variable indicating the end of the configuration to **true** when the user taps the *Done* button. If the user closes the **Activity** before having tapped the *Done* button, the variable is not set and the configuration is not considered as finished, and will start again next time the user launches the application. As soon as the user taps the *Done* button, **FirstRunActivity** terminates and the *Settings Activity* is popped from backstack.

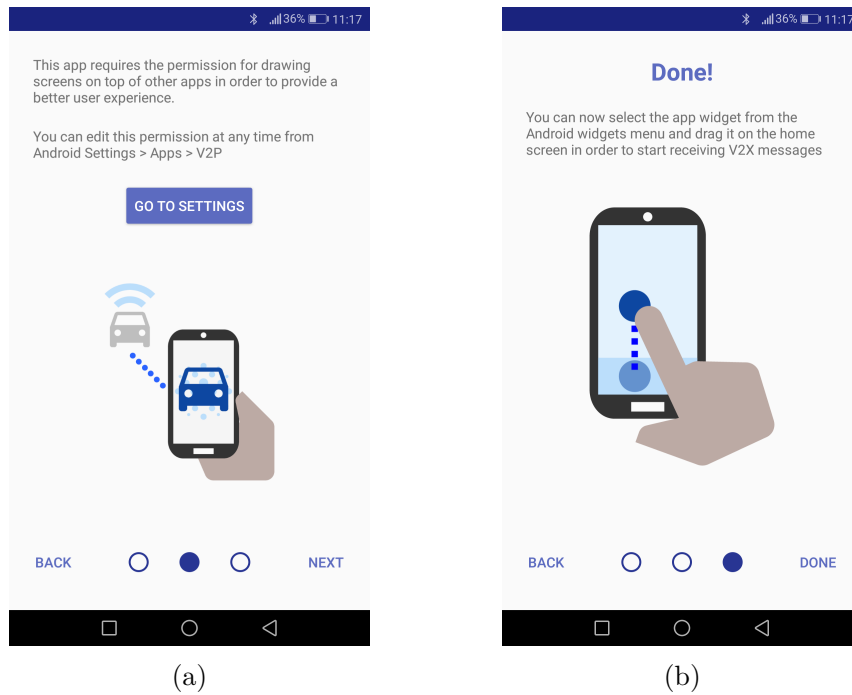


Figure 5.12: (a) Screen for overlay permission request, (b) Screen for teaching the user how to enable the V2P.

5.7 Notifications & Screen Overlay

Notifications and the screen overlay are the final goal of the V2P application.

5.7.1 Notifications

Notifications accompany the whole life of the V2P *widget*, from the moment its first instance is dragged on the home screen until its last deletion. This is mainly due to the fact that the `ReceiveService` is marked as foreground, hence it requires a non-dismissable notification to be issued as soon as its `onStartCommand` method is executed. In order to cover also the cases in which the `Service` is not running due to absence of connection, an *ongoing* notification is also issued when at least an instance of the *widget* is on the screen.

Below are reported the values that a notification can assume:

- *V2X not yet configured* (Figure 5.13). This happens when the user drags the first instance of the *widget* on the home screen without having first launched the application from the app menu. This notification is clickable, that means if the user taps it, it will launch the configuration wizard.

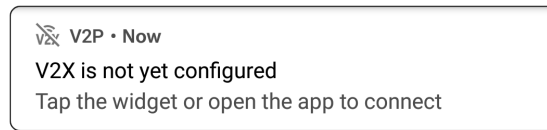


Figure 5.13: V2X not yet configured

- *V2X available* (Figure 5.14). In this case `ReceiveService` is running as the Bluetooth is on, the connection with the *anchor* is established and `ReceiveThread` is reading from the `InputStream`. Data is being received and processed but the car and the user are far from each other.



Figure 5.14: *V2X Not Available*: connected to the *anchor*

- *V2X not available* (Figure 5.15). This case signals either a problem in the connection towards the *anchor* or that the Bluetooth is currently disabled.

The first case can occur when the `ReceiveService` is trying connect to the *anchor*, either for the first time or after the connection was lost. The second one, tells the user that the `Service` can not start because the Bluetooth is currently disabled.

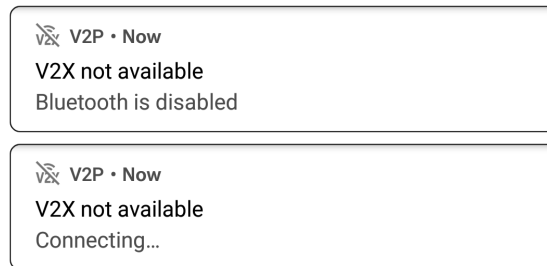


Figure 5.15: *V2X Not Available*: Bluetooth is disabled or the device is trying to establish a connection

- *Car approaching* (Figure 5.16). This notification is issued every time a car is in the user's area of interest. If the application is allowed to drag screens over other applications, this notification comes with the screen overlay, otherwise, the notification is given a parameter that allows the user to reset it with a tap.



Figure 5.16: *Car Approaching*

5.7.2 Overlays

Figure 5.17 shows the warning screen that is drawn onto whatever application is currently on foreground when the vehicle enters the pedestrian's area of interest.

The screen is very simple and is designed to catch the eye of the user. The peculiarity of screens dragged on top of other applications is that they disable all the buttons but theirs. For this reason, in order to dismiss the warning, the user has to tap on the *Got it* button. This is also the reason why the overlay feature requires an explicit authorization in Android Settings.

If the event disappears without the user having seen it, the screen is automatically dismissed together with the notification.

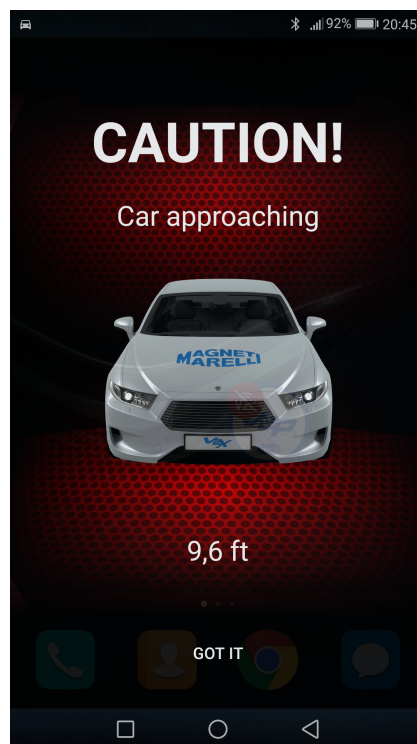


Figure 5.17: Overlaying screen

Chapter 6

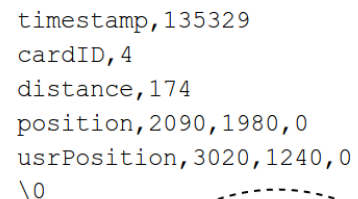
V2P Algorithm

This chapter illustrates the algorithm that triggers the V2P events on the pedestrian’s smartphone. The corresponding code is part of the `ReceiveService` class and is run every time a correctly formatted message from the *anchor* is received.

To summarize, notifications shall be sent every time a vehicle enters the pedestrian’s area of interest, which is defined as the radius of a variable amount of millimeters centered on the user.

Such an amount is the sum of two measures, one called *distance* and one called *tolerance*. Both will be discussed later, after having introduced the format of the message sent by the *anchor*.

The Skyloc™ *anchor* device broadcasts a CSV-formatted message every 90 milliseconds with a connection-less protocol. The connected device, i.e. the smartphone, has to listen to the stream and concatenate the read characters until they form a suitable string starting with the word “`timestamp`” and ending with the sequence “`\0`”. The complete message is shown in Figure 6.1 and it is possible to notice that it is composed by the following key-value pairs:



```
timestamp,135329
cardID,4
distance,174
position,2090,1980,0
usrPosition,3020,1240,0
\0
```

Figure 6.1: Example of message sent by the *anchor*

- `timestamp` (not used). The value represents the number of milliseconds elapsed since the *anchor* device has been turned on. The `timestamp` field has been prepared for further expansions of the application but currently the version described in this work does not use it.

- **cardID**. It corresponds to the **carID** field of the V2X Communication board installed on vehicles. In this case, it represents the identifier associated to the SkylocTM device.
- **distance** (not used). This field represents the point-to-point distance between two *tags* (the vehicle's and the pedestrian's) measured by the *anchor*. This field has not been used as it is not as accurate as the distance obtained by triangulating each *tag*.
- **position**. This value is made of three coordinates that represent the location of the *tag* associated to the vehicle in the Cartesian coordinate system centered on the *anchor*. Their value is expressed in millimeters.
- **usrPosition**. This field represents the coordinates of the *tag* associated to the pedestrian. They are expressed in millimeters and belong to the Cartesian coordinate system centered in the *anchor*.

Skypersonic's SkylocTM system can triangulate real-time indoor locations with an accuracy of $\pm 20cm$, but in particular situations it can decrease due to interference. For this reason the radius of the user's are of interest also had to include the *tolerance* value, in order to reduce the noise affecting the values of consecutive messages.

To support the message processing, the V2P application defines a custom class, named **Vector3D**, to store the received coordinates of each *tag* and compute the distance between them by using the Pythagorean theorem. Moreover, it also uses an **int** type variable, **oldMin**, to store the distance that triggered the notification and a **boolean** one to remember whether the vehicle is within the pedestrian's area of interest.

It is important to highlight that this algorithm has been prepared for supporting more than one vehicle, but for demonstration purposes a single vehicle was enough. The following description will then make the assumption that there is at most a vehicle at a time approaching the pedestrian.

Figure 6.2 illustrates the flow of the algorithm, which starts as soon as the **Handler** of **ReceiveService** receives a well-formatted string from the **ReceiveThread** running on a secondary thread.

First of all, if the overlay screen is not currently displayed, that is the vehicle is not near the pedestrian, the variable **oldMin** is reset to the maximum value admitted for the **int** type.

Now the **usrPosition** and **position** coordinates are extracted from the string, inserted into two **Vector3D** objects and the distance between the two is computed.

This distance is then compared against the threshold distance, whose value can be

modified by the expert user in the *Debug Settings* screen: if the computed one is lower then the vehicle has just entered the pedestrian's area of interest.

Before triggering the notification, however, it is necessary to verify that the vehicle is heading towards the user, hence the distance is also compared to the `oldMin` value, which represents the distance between the two *tags* in the previous cycle. If also this condition is true, then the `SHOW_OVERLAY` broadcast is sent, after having verified that in the meantime the *widget* has not been thrown away. In fact, if all the instances of the *widget* are deleted from the screen, the `SHOW_OVERLAY` broadcast shall not be sent, since the user is no more interested in receiving V2P notifications.

As the reader may recall, the overlaying screen also shows the distance between the vehicle and the pedestrian. This is possible because the `SHOW_OVERLAY` broadcast can also carry extra fields, such as the distance indeed.

The `LocalNotificationManager`, while drawing the warning screen layout, also converts the distance expressed in millimeters to either meters or feet, based on the measure the user selected in the application's *Settings*.

If the overlay is currently displayed when a new message is received, the `oldMin` variable is not reset, as the vehicle is currently within the pedestrian's area of interest. If the newly computed distance is greater than the threshold then the vehicle may be getting far from the user. To ensure that two conditions are checked: the vehicle is in range and the distance is greater than the sum of the threshold distance and the tolerance. If both are verified then the `HIDE_OVERLAY` broadcast must be sent after having reset `oldMin` and after having marked the vehicle as outside the range. Beside removing the warning screen, this broadcast also has to reset the notification to the proper value, based on the current state of Bluetooth. However, there is still a third case to take into account: if the last instance of the *widget* is thrown away while the warning screen is drawn on the current application, the `onDestroy` method of the `ReceiveService` will take care of sending a `HIDE_OVERLAY` broadcast. In this case, the notification should be definitely canceled and not updated, as as soon as the `Service` dies, the user will no longer be able to clear it due to its *ongoing* flag.

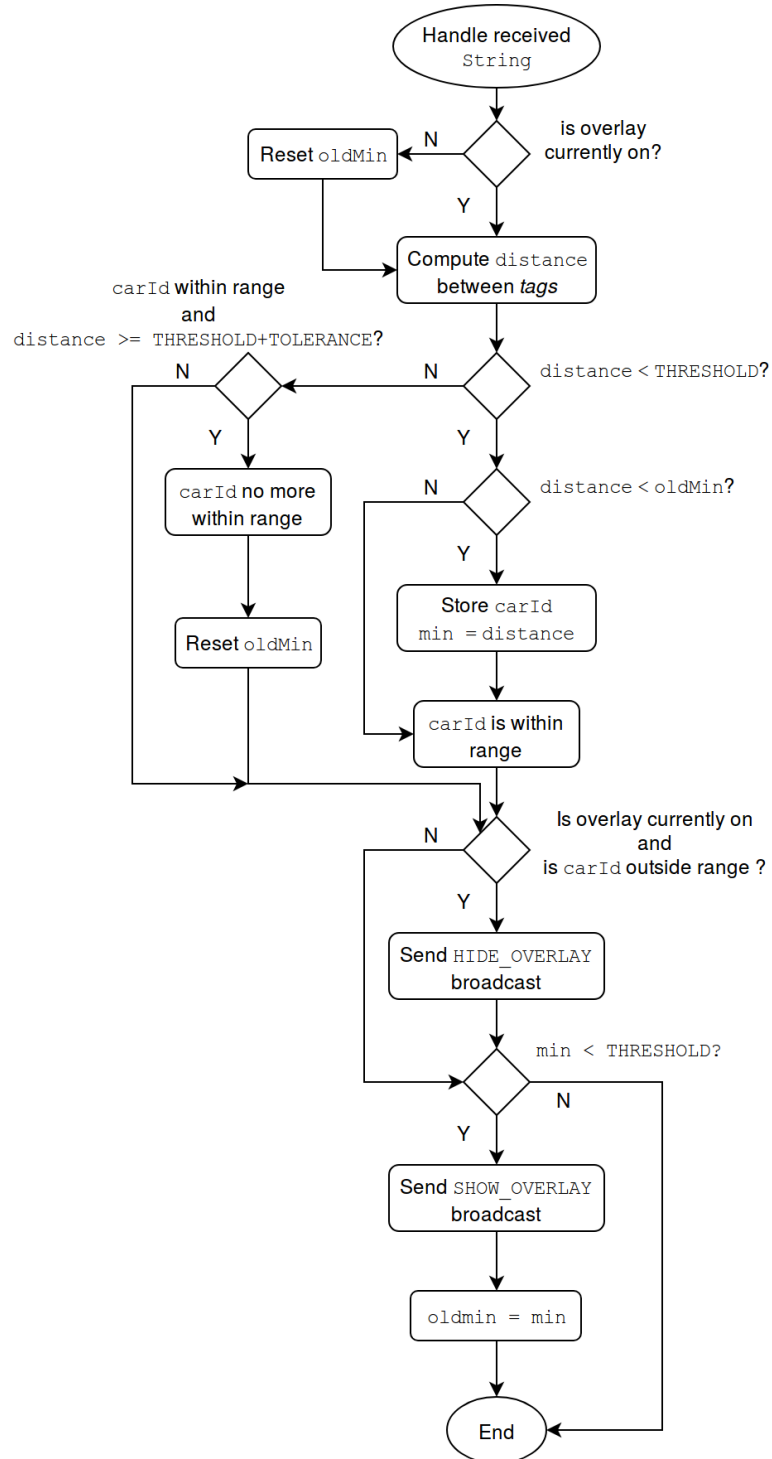


Figure 6.2: Service flow chart

Chapter 7

Design Choices for the V2P App

This chapter is going to illustrate the chosen design strategies and compare them against alternative ones, focusing on the pro and cons. For better clarity, it will be divided into two sections, one for the hardware and one for the software.

7.1 Hardware choices

As said in the previous chapters, the hardware architecture for the demonstration was composed by a set of real-time indoor positioning devices made by Skypersonic whose Skyloc devices were able to communicate with the phone via Bluetooth. Two considerations can be done about the hardware: one concerning the wireless communication and the other concerning the choice of the SkylocTM system to provide the indoor positioning.

The booth at the CES2018 in Las Vegas was set up for showing innovations of all the departments of Magneti Marelli, hence the displacement of the devices was not managed by the researchers of the Innovation & Connectivity group. This means that they had to stick to some space and logistic limitations, such for example the computers that ran the demonstration software and the number of screens.

7.1.1 Bluetooth

The wireless communication between the phone and the *anchor* device could either use Bluetooth or WLAN. The main reason Bluetooth was chosen in spite of WLAN is that

Bluetooth modules provide a method to easily detect the signal strength of another Bluetooth device and thus detect the approximate distance of two devices. Another reason is that the room where the booth was placed had many wifi networks that may have interfered with the WLAN used for V2P.

7.1.2 The infrastructure

The solution using Bluetooth RSSI would have required a program or a script on the computer behind the booth, which continuously checked the signal strength and sent the information to the phone. This solution is affected by two issues: the first one is that the RSSI value, for Android devices, differs from device to device, based on the chipset vendor[13], hence the value needs to be normalized in order to make the V2P application portable; the other issue is that RSSI varies depending on interference, such as a body between the two devices. This would have been a big issue at the CES2018, where there was a lot of people.

These reasons made the choice of the infrastructure fell on the SkylocTM system, because it can provide very precise indoor positioning and it already came with a ready-to-use software that sent the information required by the application.

7.2 Software choices

The next subsection will analyze how the application was designed and why some implementations have been preferred over others.

7.2.1 Android OS support

Starting from Android 6.0 Marshmallow, the operating system introduced some major changes in the permission management. One of them was indeed the permission that allows an application to draw screens over another one, which was not supported by older versions. This reason initially led to the choice of developing the V2P application for Android M, however, on some smartphones running such a version, were affected by a bug which prevented that permission to be shown on the system's Settings screen. The support for Android M was then definitely dropped in favor of Android 7.0 Nougat. Another reason that made the older versions to be abandoned is that they do not support the Material Design theme and they would have required a lot of efforts to develop a backwards-compatible Graphical User Interface.

7.2.2 Home Screen Widget vs. auto-starting Activity

The reason why the application has been developed as a home screen widget instead of a simple **Activity** is that the main requirement was that the V2P functionality should have been always running once started, without needing the user to manually start it every time. Once widgets are dragged on the home screen, they are under the control of the home screen application, which automatically takes care of restarting them every time the phone is rebooted.

Instead, a standard **Activity** would have required the user to manually start it at every reboot or, in order to start automatically, should have required special permissions. Moreover, an **Activity** would require more memory, as it is a standalone element, while the widget lives as a **BroadcastReceiver** and little graphics on the home screen application.

Another advantage of widgets is that their view is directly visible on the home screen and can communicate issues through changes in their layouts without being as invasive as an **Activity** brought to foreground would be.

7.2.3 The debug screen

Since the infrastructure of the V2P in Venaria Reale was different from the one set up at the CES2018, the thresholds triggering the V2P notifications could have been very different from each other. For this reason and for avoiding bad performances during the demonstration, such thresholds have been made editable. Since in outdoor usage, V2P is based on DSRC, such parameters have no reason to exist anymore, hence they have been placed in a hidden debug screen, in order not to be seen by common users.

7.2.4 V2P algorithm & Pythagorean theorem

As previously discussed, the algorithm that computes the distance between two *tags* uses the Pythagorean theorem to obtain the result:

$$\sqrt{a^2 + b^2} = c$$

Many other strategies could have been used for that, for example the Pythagorean trigonometric identity:

$$\sin^2 \theta + \cos^2 \theta = 1$$

However, even though the result is the same, from the computational point of view, the Pythagorean trigonometric identity is heavier than the Pythagorean theorem

because of the trigonometric functions, sine and cosine.

Computational load is important to take into account when programming for mobile devices as the more the CPU works, the more the battery is drained quickly.

The V2P application already requires a lot of processing for managing the both the Bluetooth and the communication between the receiving thread and the main one. These operation could cause some delays since they also depend on the general load of the operating system so, if computationally heavy operations are executed many times per second, delays can increase and cause a bad user experience.

One could argue that modern high-end devices may have a special processor for optimizing trigonometric operations, but mobile applications must be programmed to work smoothly also on low-end devices, so a careful design will avoid bad behaviors.

Chapter 8

V2P Testing

This chapter concludes the first part of this work by illustrating the test methodology adopted for the V2P application and the issues that have been found.

The overall results, as well as further developments, will be discussed in Part III.

The testing of the V2P application has been done according to Agile principles, hence it has been carried out before each delivery of intermediate versions of the product.

The Graphical User Interface has been tested in collaboration with other members of the team with the Exploratory testing strategy.

Exploratory testing consist trying to break the software by using it in a clever way, without following scripted tests. This operation mimics in some way the usage of the application in real life, also including accidental actions, which, if not correctly managed by the code, could lead to misbehaviors or glitches[26].

Since the very first tests of the connectivity between the V2P application and the *anchor* it was found that with the connection-less protocol adopted by the *anchor* about two-thirds of the messages were not captured by the V2P algorithm. This was mainly due to the fact that the application knows neither when the stream starts nor how many bytes compose the message. For this reason it has to read a bunch of data and then start concatenating further characters after having found the `timestamp` sequence.

The loss of messages may seem huge but, as described in the previous chapters, the *anchor* sends a message every 90 milliseconds, which is a lot, even for the purpose of the V2P application. This is the reason why the issue was ignored. In fact, in real life, neither vehicles nor pedestrian move so fast to cause big differences among two consecutive messages.

Part II

V2X HMI application

Chapter 9

Vehicle-to-Vehicle

The mobile application developed for the second part of this thesis has been conceived for allowing the researchers to easily debug the software loaded on the control units of the test cars. Currently the debug requires a laptop to be connected to the V2X board, taking a significant amount of space in the small interior of the vehicle.

Moreover, in order to further enhance this operation, the application was also required to display, on a mobile device, the same V2X event alerts that are shown on the car's dashboard. In fact, those tests are carried out by groups of three or four researchers per vehicle, who can hardly see the events displayed on the dashboard in front of the driver.

The Innovation & Connectivity team is equipped with two cars for the tests, whose Electronic Control Units have been conveniently modified to integrate with the V2X Communication board made by Magneti Marelli. One of the two cars is also able to display the V2X events on its dashboard, in the form of text and icons.

Before proceeding with the description of the mobile application, a quick reference to the functioning of the Vehicle-to-Vehicle communication on the company cars and to the test methodology should be given.

9.1 Vehicle-to-Vehicle Communication

Vehicle-to-Vehicle communication denotes the exchange of information between two vehicles that is then translated into alerts for the driver.

9.1.1 V2X Use Cases

The use cases currently implemented on the V2X communication board of Magneti Marelli are Control Loss Warning (CLW), Emergency Electronic Brake Light (EEBL), Forward Collision Warning (FCW), Left Turn Assist (LTA), Intersection Movement Assist (IMA), Stationary Vehicle warning (SV).

Additional use cases are Road Side Unit (RSU) and IVRS, which concern the communication between a vehicle and the road infrastructure, also called Vehicle-to-Infrastructure, V2I.

The icons and the texts for the alerts have already been arranged on the control units in contemplation of a further software implementation.

The following paragraphs are going to illustrate the V2V use cases listed above.

Control Loss Warning (CLW)

Control Loss Warning is an event generated and broadcast by a vehicle whose sensors detect loss of traction. Upon receiving such information, the surrounding vehicles evaluate the relevance of the event and, if appropriate, provide a warning to the driver^[2].

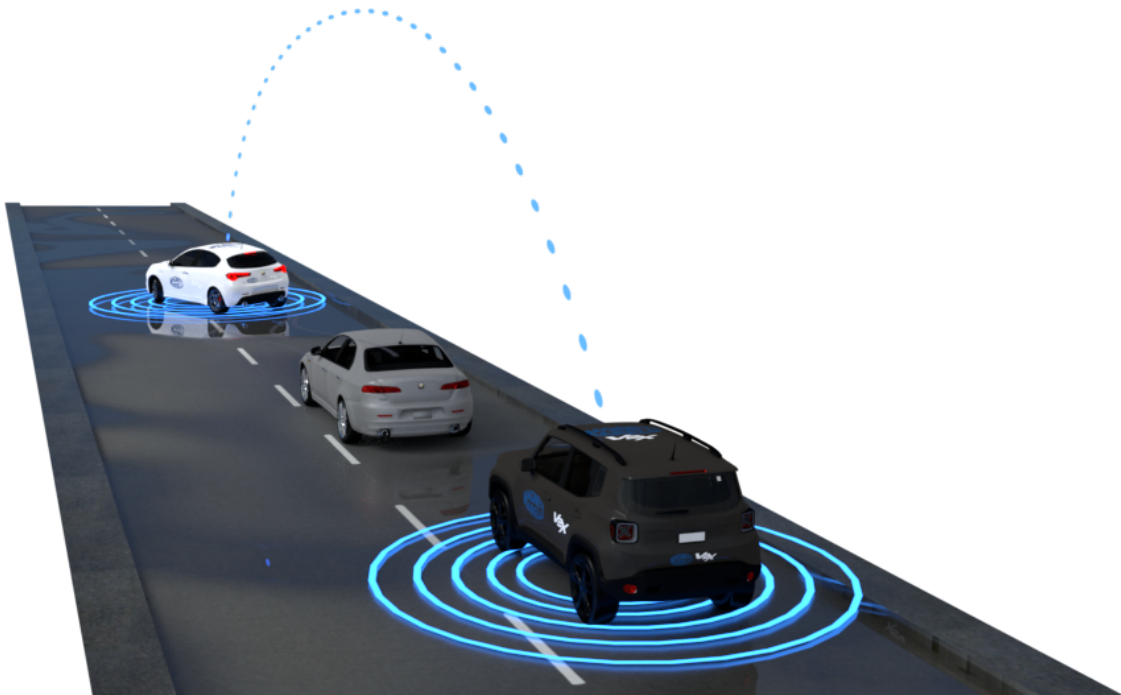


Figure 9.1: Control Loss Warning event

Emergency Electronic Brake Light (EEBL)

Emergency Electronic Brake Light is an event generated and broadcast by a vehicle performing an emergency brake. Upon receiving such information, the surrounding vehicles evaluate the relevance of the event and, if appropriate, provide a warning to the driver. This event is particularly important when the transmitting vehicle is out-of-sight of other drivers[2].

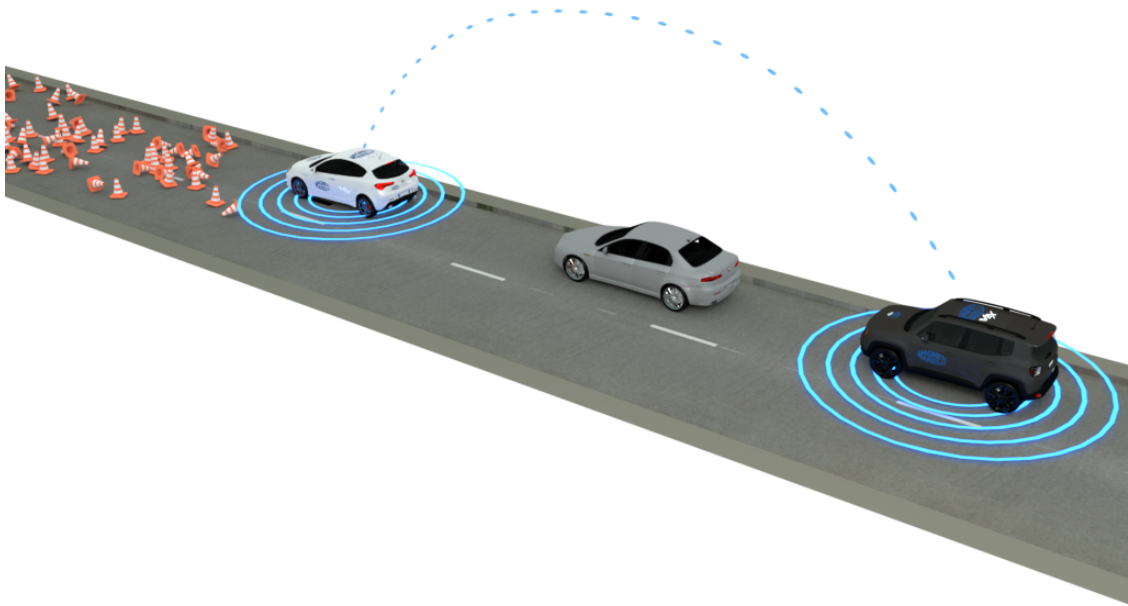


Figure 9.2: Emergency Electronic Brake Light event

Forward Collision Warning (FCW)

Forward Collision Warning is an event generated by a vehicle whose sensors detect an impending rear-end collision with the vehicle ahead in traffic, in the same lane and direction of travel[2]. This event is intended for the driver of the vehicle detecting the impending collision.

Left Turn Assist (LTA)

Left Turn Assist is generated by a vehicle attempting to turn left at an intersection in order to warn its driver an oncoming vehicle that may be out-of-sight. As for FCW, the LTA event is intended for the driver of the vehicle generating the event.

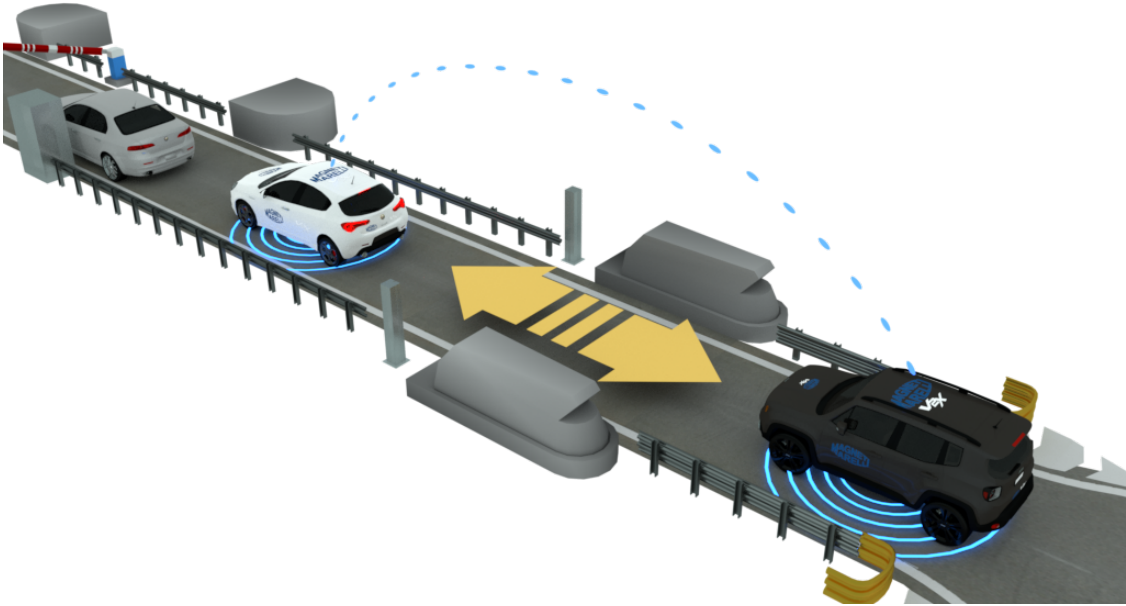


Figure 9.3: Forward Collision Warning event

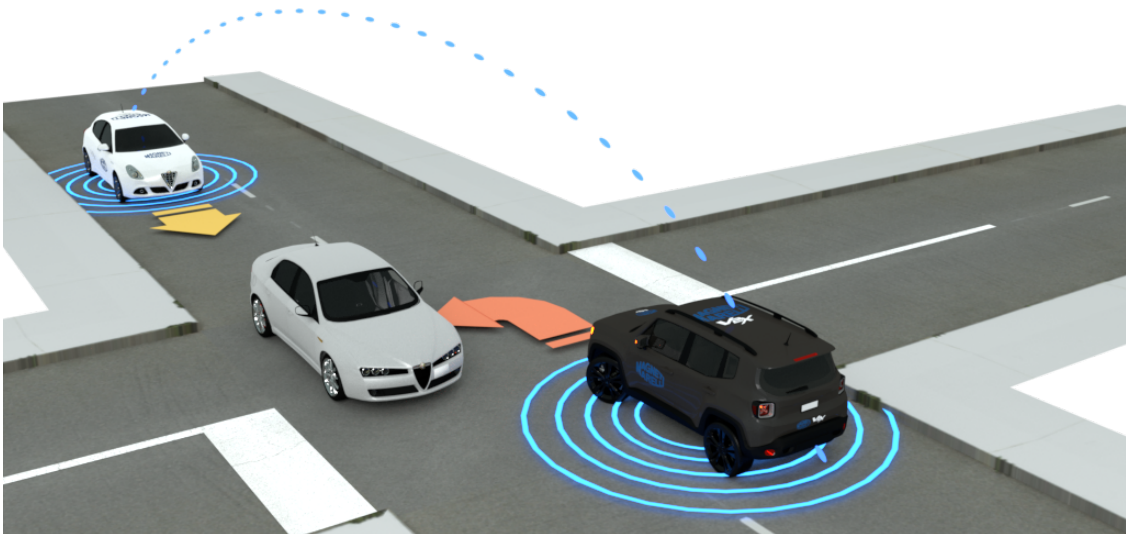


Figure 9.4: Left Turn Assistant event

Intersection Movement Assist (IMA)

Intersection Movement Assist is an event generated by a vehicle approaching an intersection, which detects a crossing vehicle. The goal is to warn the driver about the high collision probability[2].



Figure 9.5: Intersection Movement Assist event

Stationary Vehicle (SV)

Stationary Vehicle is an event generated and broadcast by a vehicle having flashing emergency lights on. The goal is to warn other drivers and have them reducing their speed and possibly making a lane change.

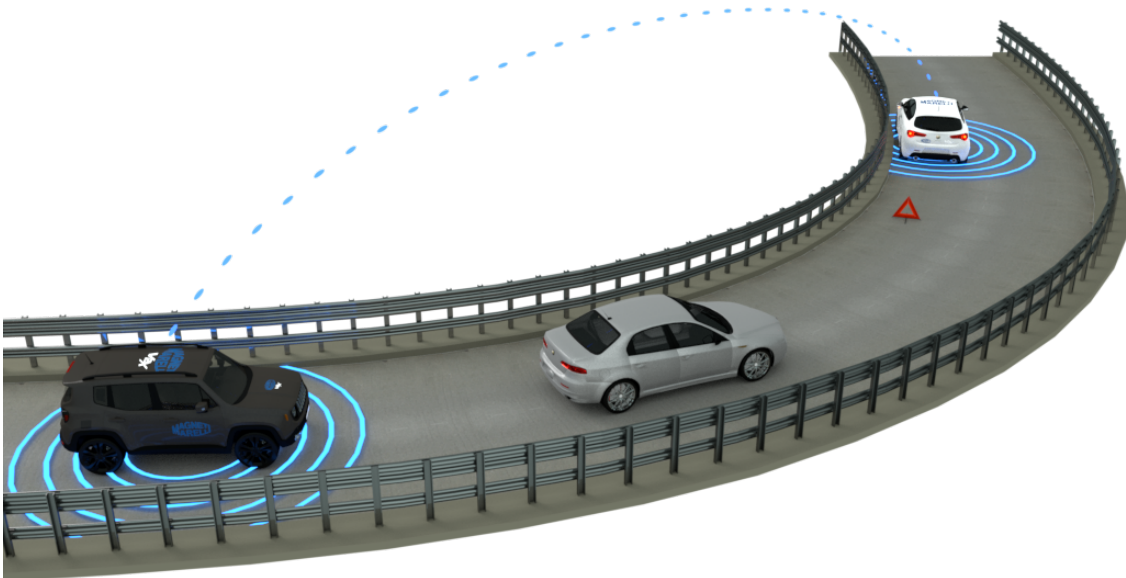


Figure 9.6: Stationary Vehicle event

9.2 Test methodology

The tests are carried out using two vehicles in the following way: one acts like the vehicle out of the other driver’s line-of-sight, hence it is used as the V2X event “generator” and for this reason is called Remote vehicle; the other is the car that receives the events and displays the alerts on the dashboard, hence it is called Ego vehicle.

Of course the above is a simplification as both cars are actually able to transmit and receive V2X events. In fact, in some use cases, the events are captured and displayed by both vehicles.

Since the Remote vehicle is used as “transmitter”, it usually does not require passengers but the driver, hence the testers ride on the Ego vehicle, to which they can connect with a laptop in order to read the logs.

The majority of the tests is carried out in the private parking lot of the company. The tests requiring special conditions, such as for example GPS coordinates far from each others, are done either on the road or on track.

9.3 A note about the images for the use cases

The scenes illustrating the use cases presented above have been created with Blender in order to help the researchers explain the Vehicle-to-Everything events to other colleagues during internal demonstration.

The majority of the 3D meshes has been modeled for the purpose, while some have been downloaded from the Internet. The texturing, as well as the lighting and the material set up, are original works set up for making the renderings more photo-realistic. The set up of these scenes is a minor part of the work done for this thesis, which however required a special dedication.

Chapter 10

V2V Architecture

This chapter is going to describe the hardware and software architecture of the mobile application and the V2X communication board mounted on the Ego vehicle.

The latter will be illustrated first, by paying specific attention to the interaction between the board and the instrument cluster of the vehicle, so that the reader can easily spot the similarities with the software architecture of the mobile application.

10.1 Hardware Architecture

The V2X Communication board is a control unit integrated on the vehicle and equipped with a GPS, that is able to interface both with the CAN bus and the dashboard, and with near vehicles. The goal is to know at any time the position and the state of the vehicle on which the board is mounted but also of the surrounding vehicles, in order to show warning alerts to drivers about the V2X events extrapolated from received data. This operation is carried out by comparing the values of some types of CAN messages and the GPS positions of each vehicle with threshold values: based on the result, the control unit can understand whether there is a hazardous situation. If the answer is positive, the board tells the dashboard which event has to be displayed and its level of criticality.

At the same time, the data read from the CAN bus and the GPS, is wrapped into a message and periodically broadcast to near vehicles, in order to have them processing the informations and possibly alert their drivers.

Figure 10.1, beside showing the architecture discussed so far, also illustrates how the application has been integrated in the system: whenever the V2X control unit detects an event, it sends a message both to the instrument cluster and also to the mobile

devices connected to the Wi-Fi hotspot of the vehicle.

It is important to notice that the application is currently unable to damage the vehicular system because the V2X framework is not able to receive messages from the outside and is not directly connected to the CAN bus anyway, for obvious security reasons.

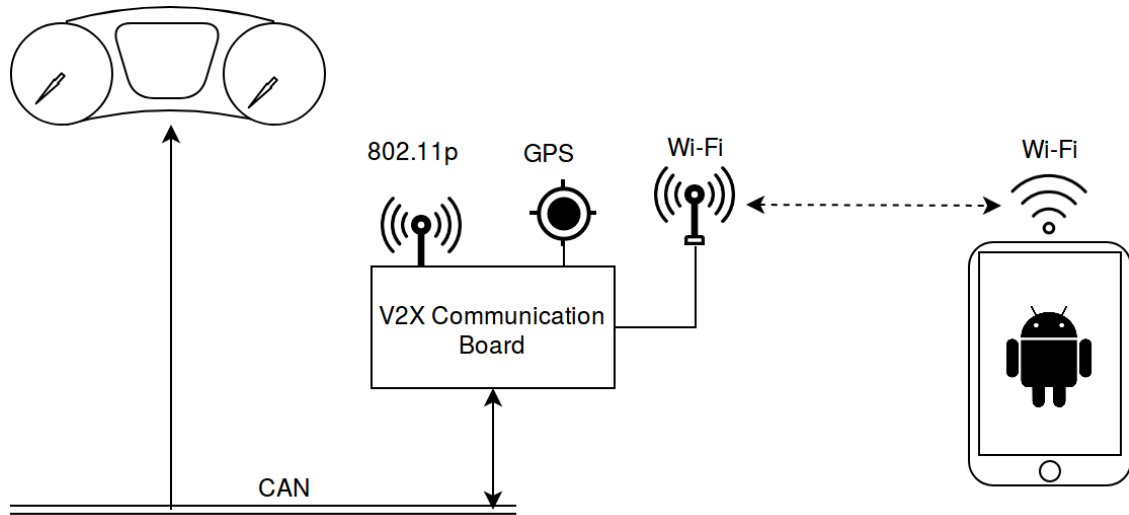


Figure 10.1: Hardware architecture of the Vehicle-to-Everything technology installed on the Ego vehicle. The Android mobile devices connect to the V2X Communication board via Wi-Fi

10.2 Software Architecture

The software loaded on the V2X Communication board is made of many modules, each managing a specific type of V2X event. Such modules are independent from each others, that means each of them can be either activated or deactivated at will, without the functioning of the board being compromised.

In order to manage the communication towards the mobile devices, it was necessary to add an additional module, called dispatcher, to the framework to broadcast via Wi-Fi the messages meant for the dashboard.

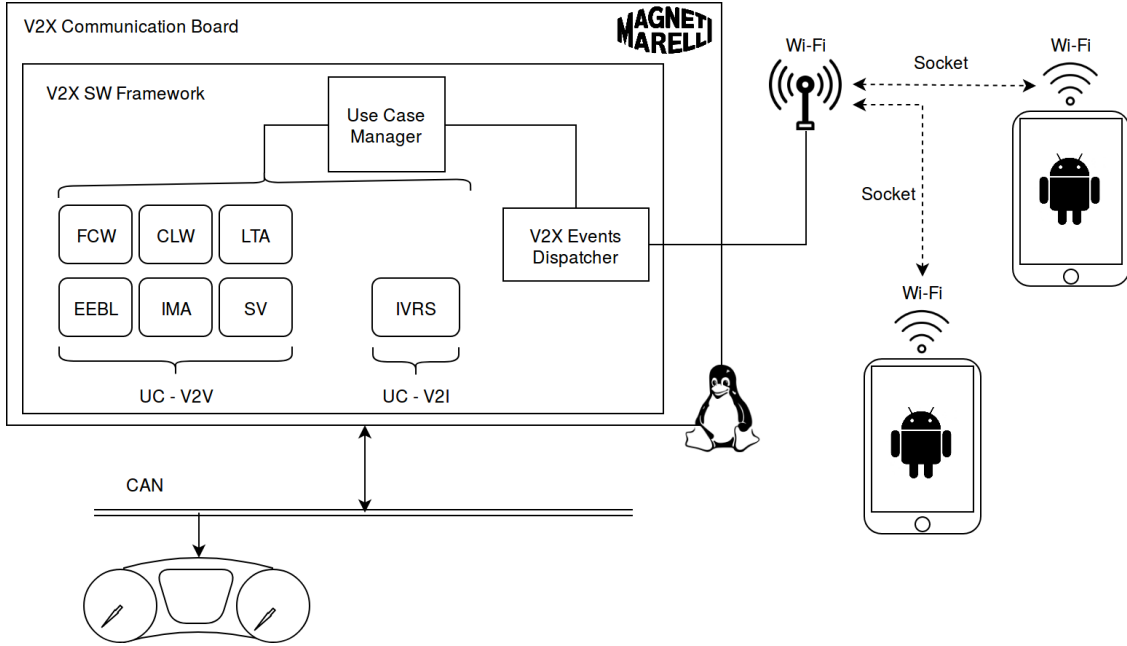


Figure 10.2: Software architecture of the framework loaded onto the V2X Communication Board

10.3 The Instrument Cluster of the Ego vehicle

The instrument cluster of the Ego vehicle assigned to the team is equipped by a 7 inches display, premium version¹, which is mainly composed by three areas, as shown in figure 10.3: the central part (B) is committed to the visualization of alerts in form of images and texts.

The presence of the V2X control unit on the Company’s Ego vehicle brings with it a few changes to the alerts shown on the central area of the display, in order to allow also the visualization of V2X events. Usually, in the central area of the display, the original Ego vehicle, i.e. the one without the V2X control unit, shows the vehicle speed. The modified one instead shows a little Ego vehicle on a road. This kind of layout is able to show both the state of the V2X connectivity and the detected events, beside, of course, the classic telltales for faults.

The layout showing the little car on the road has been named “grid”, since the V2X events can be placed in fixed cells of an invisible grid overlapping the road image. That

¹The premium display differs from the base one because it can show color images and simple animations. Moreover it can be reconfigured by adding/editing image mappings in the control unit.

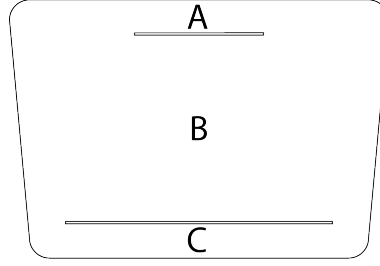


Figure 10.3: The multifunctional display of the Ego vehicle is divided into three areas:
 (A) Upper zone, where the date, titles and other short informations are displayed;
 (B) Central zone, used for displaying menus and V2X events;
 (C) Bottom zone: displays the fuel level and the temperature

grid, shown in figure 10.4a, is made of two rows and three columns in order to represent far and near events (rows), whose position can be on the right, on the left or at center with respect to the road (columns).

There is a second way to show the events to the driver: whenever the level of criticality is high and the distance of the events from the vehicle is below a certain threshold, the grid is temporarily replaced by a popup (Figure 10.4b), which shows an icon representing the event, a text message and a sort of compass to indicate the origin of the event.

The V2X connectivity state is represented through the icon of the car in the grid layout, together with a telltale indicating also the V2X standard currently used by the car.

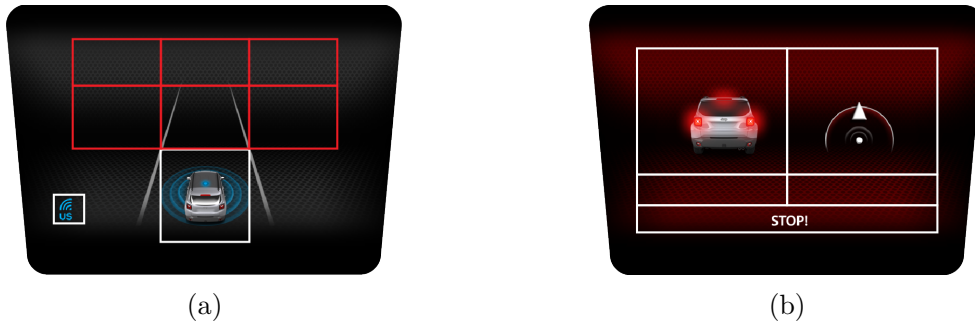


Figure 10.4: The display modes for V2X events on the Ego vehicle's display: (a) grid;
 (b) popup.

10.3.1 Functioning

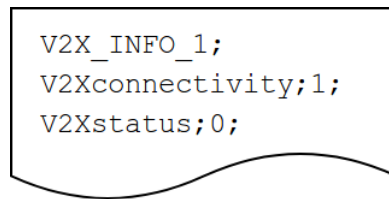
Both the images and the texts used to form the alerts are stored as values in a key-value map data structure, where the key is represented by an integer contained in a specific range.

The selection of the elements to be displayed is made by a CSV formatted string that is sent to the instrument cluster, which contains, for each field, either the key corresponding to the specific value that must be retrieved from memory or a numeric variable to indicate distance or time.

Internally, the logic of the dashboard requires the last message to be repeated every 100 milliseconds for maintaining images and texts on the display. In this way, the instrument cluster can notice any possible failure of the underlying control units, because they would stop updating the messages. This concept will be clarified later with an example even if it does not directly affect the way the application and the framework module work. The next three paragraphs are going to describe the three types of V2X messages the dashboard is able to understand.

V2X_INFO_1

The V2X_INFO_1 message type (Figure 10.5) controls the image of the car at the bottom of the grid view and the icon for the V2X standard. The dashboard can autonomously load two kinds of state: *not present* and *not available*, since the V2X Communication board usually takes few seconds to become operational and to start sending messages. On startup the dashboard displays the *not present* configuration, represented by the yellow car with the yellow exclamation mark and the red state icon (Figure 10.6c).



```
V2X_INFO_1;  
V2Xconnectivity;1;  
V2Xstatus;0;
```

Figure 10.5: Syntax of the V2X_INFO_1 message

As soon as the V2X framework is ready, it sends a V2X_INFO_1 message for notifying that the state has now been changed to *available*. The dashboard then loads the grey car over blue waves and the blue icon displaying the V2X standard currently in use (Figure 10.6a).

If the instrument cluster does not receive any V2X_INFO_1 message for at least 30

seconds, the state is automatically set to *not available*, which is represented by the grey car and the yellow state icon (Figure 10.6b).

If one of the modules of the V2X framework crashes, the state will be changed to *not present*.



Figure 10.6: Images for the V2X state

V2X_INFO_2

The `V2X_INFO_2` message type (Figure 10.7) controls the images shown on the grid, which can manage up to three icons at the same time.

The dashboard has three slots for representing likewise events. Each slot is assigned to an index, from 1 to 3, to which an icon and the coordinates of a cell of the grid can correspond.

For what concerns each slot, the dashboard is restricted to turn on and off the corresponding cell, based on what has been read from the received CSV formatted string. It is the sender's duty to correctly manage the set and reset operation for the cells, because for each slot the dashboard keeps in memory the coordinates of the last cell that has been set. For this reason, before turning on another cell, it is mandatory to turn off the current one, otherwise the coordinates of the latter will be overwritten. This behavior is due to the fact that the reset of a cell is done by setting to zero all the fields assigned to the slot. For example, in order to turn off slot number one, the fields `CodeIcon1`, `RowIcon1` and `ColumnIcon1` must be set to zero.

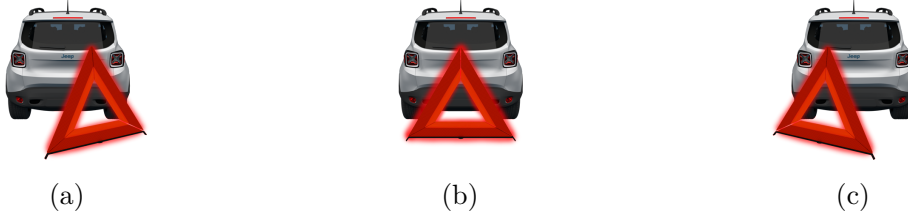
Some use cases are represented by more than one icon so that, when it is displayed on the grid, the whole screen appears more graceful. An example is given by the Stationary Vehicle use case depicted in Figure 10.8, which is mapped to three icons where the emergency warning triangle is placed in different positions.

When an event mapped on multiple icons is received, the dashboard retrieves the proper one based on the cell in which the icon must be placed. For what concerns the Stationary Vehicle use case, the icon with the emergency warning triangle is placed on the right is meant to be placed on the leftmost cells, the one having the triangle at


```
V2X_INFO_2;  
CAR_colour;0;  
RowIcon1;1;  
ColumnIcon1;2;  
CodeIcon1;1;  
RowIcon2;0;  
ColumnIcon2;0;  
CodeIcon2;0;  
RowIcon3;0;  
ColumnIcon3;0;  
CodeIcon3;0;
```

Figure 10.7: Syntax of the V2X_INFO_2 message

center must be placed in the central cells and the one with the triangle on the right is meant to be placed in the rightmost cells.

Figure 10.8: The three types of icon for the *Stationary Vehicle* use case

V2X_INFO_3

The V2X_INFO_3 message type controls the popups, that are the full screen alerts appearing when the distance of a V2X events is below a certain threshold.

The syntax of the message (Figure 10.9) has been set up for supporting also the messages about V2I events, such as the RSU and IVRS use cases. In particular, `TrafficLight_GreenSpeed`, `TrafficLight_TimeToChange` and `TrafficLight_Distance` fields are dedicated to the RSU event, respectively describing the duration of the red light, the duration of the green light and the distance from the traffic light.

As for the V2X_INFO_2 message type, also for the V2X_INFO_3 ones the dashboard

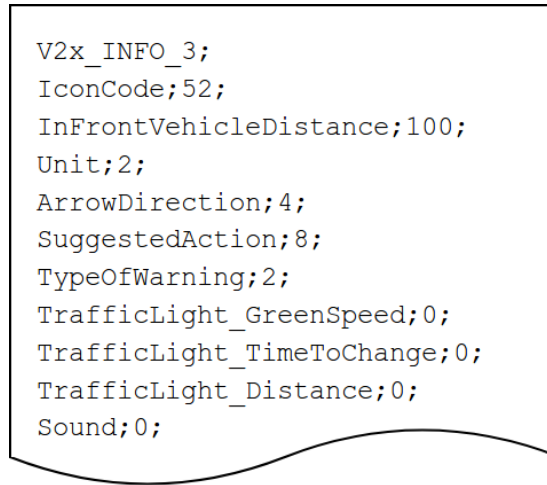


Figure 10.9: Syntax of the V2X_INFO_3 message

is committed to carry out the switch between the grid view and the popup and vice versa. The field acting as switch between the two views is **TypeOfWarning**, whose value modifies the background of the popup screen. In fact, based on the level of criticality, the background can be either yellow (low criticality) or red (high criticality). There is an additional yellow background for V2I events, but currently it is not used.

Whenever the instrument cluster receives a message of type **V2X_INFO_3**, it displays the popup screen, updating it if other messages of the same type are received, but at the same time it also keeps updating the events on the grid, if **V2X_INFO_2** messages are received.

The transition from popup to grid view takes place when the dispatcher module sends a **V2X_INFO_3** message where the value of the **TypeOfWarning** field is set to zero.

Multiple images mapped to the same events also exist for popup messages. This is the case of the *Front Collision Warning* event (Figure 10.10): based on the background type, either yellow or red, the dashboard loads the proper image.

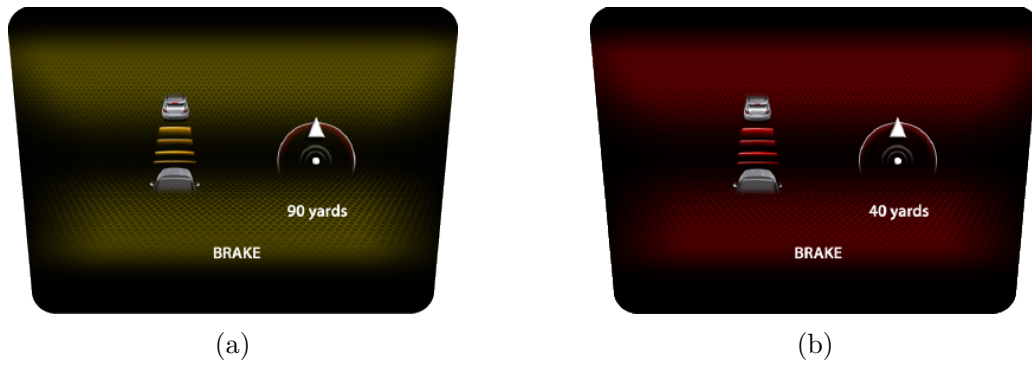


Figure 10.10: Two icons for the *Front Collision Warning* use case: (a) yellow lines and background for the low criticality alert; (b) red lines and background for the high criticality warning.

Chapter 11

V2X HMI App Requirements

This chapter is going to describe the requirements for the V2X HMI application, which has to reproduce, on a mobile device, the instrument cluster display of the Ego vehicle, for what concerns the V2X functionality.

The diagram of Figure 11.1 depicts the interactions between the application and external entities, represented by the user and the dispatcher module loaded of the V2X Communication board. As usual, the diagram illustrates both the control flows and the data flows.

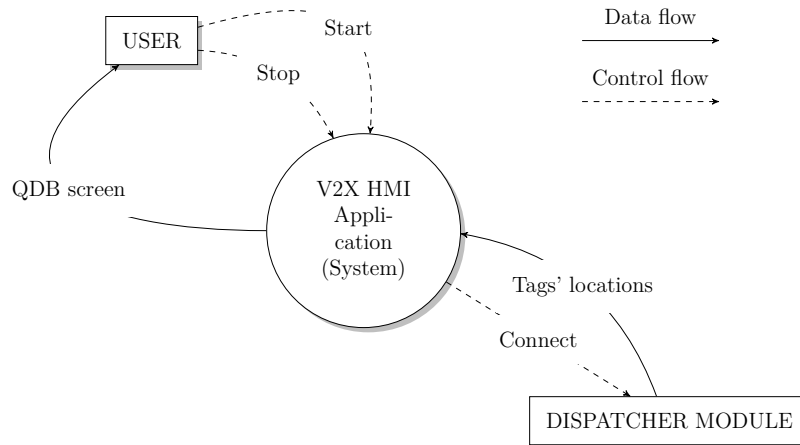


Figure 11.1: Context diagram of the system

The next two sections will list the functional and non-functional requirements for the V2X HMI application that have been pulled out from the user stories defined during the conversations with the team researchers.

11.1 V2X HMI Functional Requirements

Tables 11.1 and 11.2 list the functional requirements for the two most important interactions between the application and the external entities, which are the visualization of the V2X state and events and the reading of the string messages sent by the dispatcher module. Since the application has to communicate with an external module, it must be compliant to the communication method of the latter, which sends the messages via Wi-Fi. This requirement falls in the Functional Requirements category because it is imposed by the an entity that is external to this system.

Function	Display V2X events
Description	Display on mobile device the same V2X events displayed on the instrument cluster
Input	Formatted string containing the informations to display the correct images and texts that describe the event
Source	Dispatcher module on the V2X Communication board
Output	Retrieve images and texts from memory and display alerts on the screen
Action	Based on the type of message received from the dispatcher, retrieve and display the proper icons and texts
Requires	Depending on the type of message, either the current message or the current message and informations of the previous one
Pre-condition	The received string is correctly formatted
Post-condition	Info about a specific message type must be stored into memory

Table 11.1: Functional Requirements for the interaction between the system and the user

11.2 V2X HMI Non-Functional Requirements

Table 11.3 lists the most important non-functional requirements for the application. Most of them concern the features that make the application usable and describe how it has to perform the required actions.

11.3 V2X HMI Use Cases

The Use Cases presented in the following paragraphs will help the reader clarify how the requirements are linked together by mainly illustrating the basic flow. Next to

Function	Retrieve messages from the dispatcher module
Description	Connect to the dispatcher module via WLAN and keep listening for incoming messages
Output	Values extracted from the received string
Action	Once the connection has been established, keep listening for string messages If connection is dropped, attempt to reconnect for n times: if all of them fail, wait for the user to explicitly ask to try again
Pre-condition	Wi-Fi is enabled on the device and the mobile device is within the range of the car's Wi-Fi hotspot

Table 11.2: V2X HMI Functional Requirements for the interaction between the system and the dispatcher module

NF1	User eXperience (UX)	The appearance shall recall the one of the instrument cluster display of the Ego vehicle: same icons, same texts and same backgrounds
NF2	UX	The log screen shall recall the Linux terminal
NF3	UX	Logs shall be filterable based on their priority: debug, info, verbose, etc.
NF4	Architecture	The application shall be implemented in a modular way, to allow further expansions with new features
NF6	Connectivity	The communication between the application and the dispatcher module shall be connection oriented and possibly offer reliability
NF5	Connectivity	The application shall provide a menu for allowing the user to change the IP and port for the connection
NF6	Connectivity	Connection issues shall be reported as much precisely as possible, covering the most frequent cases: i.e. connection dropped by server, by client; connection lost due to low signal strength; Wi-Fi disabled; connection parameters not configured;

Table 11.3: V2X HMI Non-Functional Requirements

that, exceptions and other issues are treated as alternative flows.

11.3.1 V2X HMI Use Case: Connection Handling

This Use Case will follow the basic flow of actions taken when the user enables the Wi-Fi to connect to the V2X dispatcher module. It makes two assumptions:

- the application has been started and has already been configured, that means the user has already provided the IP address and the port of the server;
- no error occur during the whole process, that means that at the end the application succeeds in connecting and starts to receive messages.

If the connection fails for some reason other than Wi-Fi disabled by the user, the Use Case falls in the alternative flow and warns the user through a specific screen. In the meantime the application attempts to reconnect to the dispatcher module for three times.

If all the attempts fail then the user can choose to manually reconnect by tapping a specific button on the screen.

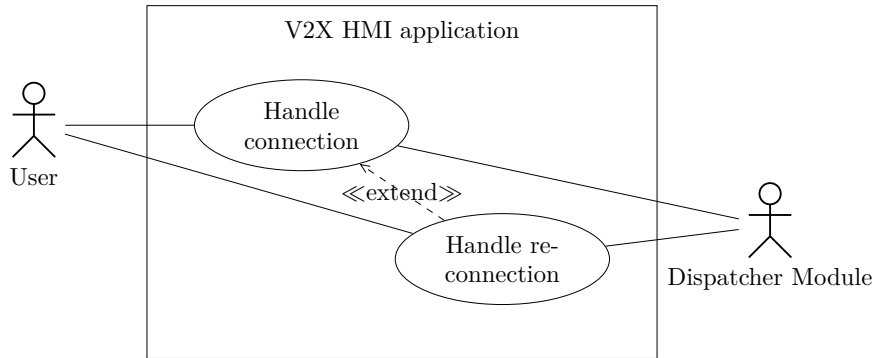


Figure 11.2: V2X HMI Use case: Connection Handling

11.3.2 V2X HMI Use Case: Display V2X Events

As soon as the connection between the mobile device and the dispatcher module has been established, the latter starts sending the messages to set up the mirrored view. The messages belong to three different type of V2X information and form the two kind of V2X alert layouts on the car's instrument cluster display.

The basic flow for this Use Case assumes that received the message is well formed. The values of the fields composing the string are pulled out and used as indices for retrieving the images and the texts from the data structures of the application. Then, based on the kind of message, the elements are depicted on or removed from the screen.

If a string were not well formed, the application would simply ignore that message.

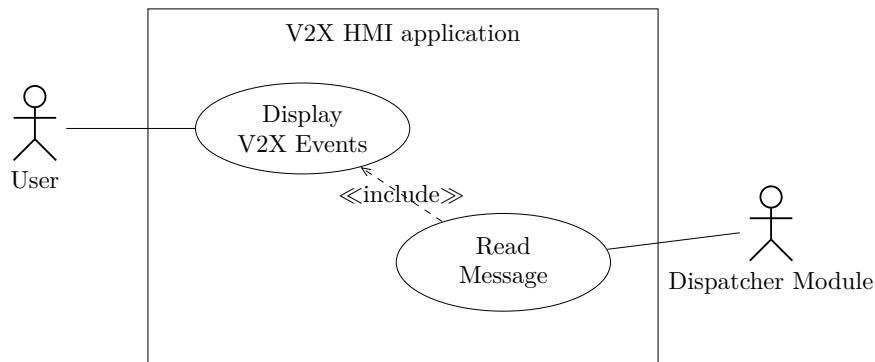


Figure 11.3: V2X HMI Use case: Display V2X Events

11.3.3 V2X HMI Use Case: Display Logs

This Use Case is triggered as soon as the application is started. Logs are messages issued by the normal functioning of the application, which also include the strings received by the dispatcher module. They are stored in a temporary data structure to allow the retrieval and printing as soon as the user navigates to the log screen in the application.

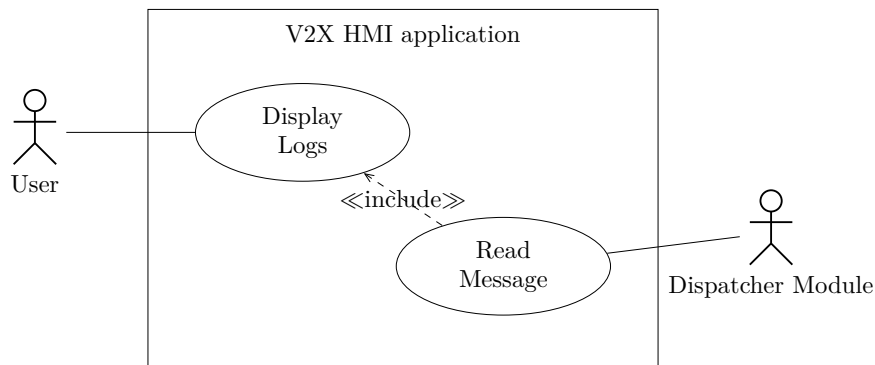


Figure 11.4: V2X HMI Use case: Display Logs

Chapter 12

Overview of the V2X HMI app

This chapter will present the main components, depicted in Figure 12.1, of the V2X HMI application along with their Graphical User Interface, describing how they are linked together to fulfill the requirements.

Differently from the V2P application presented in Part I, from the graphical point of view, this application appears simpler as it is made of a set of standard **Activities**, each made of one or more **Fragments**. Since **Activities** and **Fragments** are run on the main thread, that is the one also carrying out the Graphical User Interface, the connectivity towards the dispatcher module on the V2X Communication board required additional components to be implemented:

- a **Service**, for managing the secondary threads operating on the sockets;
- two **BroadcastReceivers**, for displaying either on a dashboard-like screen or on the log console the information received by the Service's threads and for managing the state of Wi-Fi.

12.1 MainActivity

MainActivity is the **Activity** that supports the core of the application and is the one that is being run when the user taps the launcher in the apps drawer of the phone.

The layout is made of **Fragments** and a navigation drawer, which can be opened either by tapping the icon in the leftmost part of the application bar or through a sliding gesture from the leftmost part of the screen towards the center.

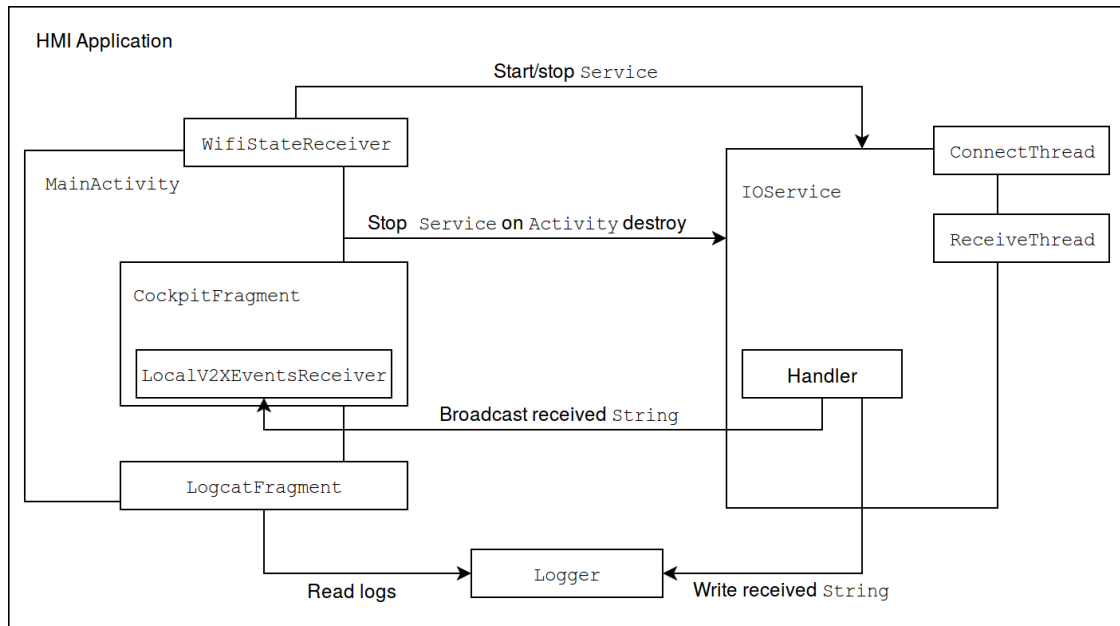


Figure 12.1: Elements composing the V2X HMI application

The navigation drawer (Figure 12.2) contains four items: *Cockpit*, *Logcat*, *Modules* and *Settings*, which, if tapped, trigger the transition to the correspondent *Fragment*.

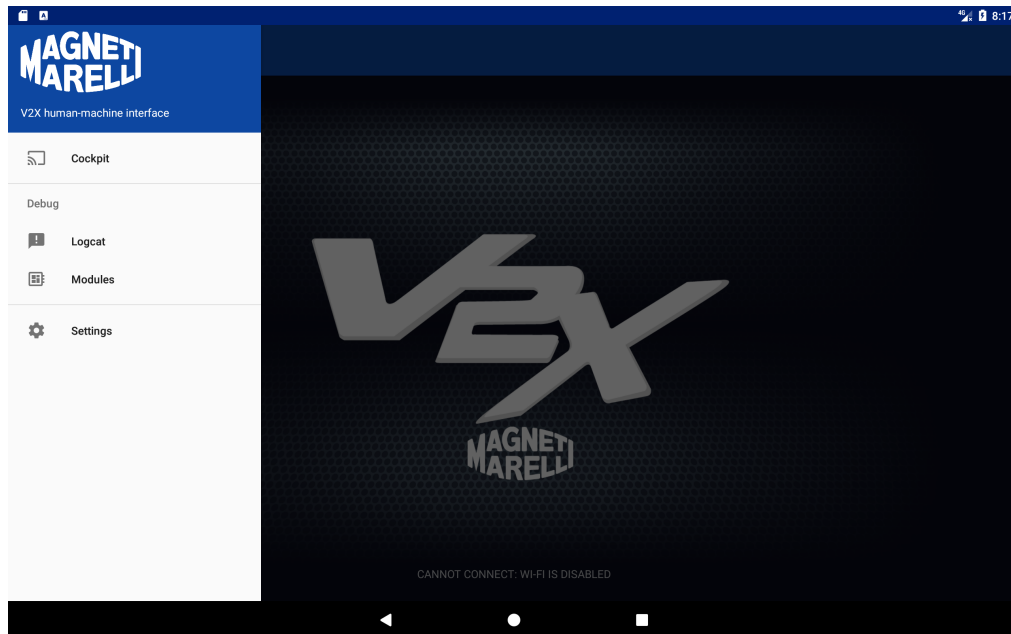


Figure 12.2: App's Navigation Drawer

The *Modules* item is currently not attached to any special screen, as it is meant for a functionality that has not been developed yet, but will be presented in Part III of this paper.

On start, `MainActivity` instantiates a `BroadcastReceiver` to listen for Wi-Fi state changes, in order to be able to start and stop the `Service` handling connection towards the dispatcher module. The functioning of both the latter and the Wi-Fi state receiver will be discussed later, in separate sections.

Instead, the following paragraphs will describe the layout of the `Fragments` corresponding to the items *Cockpit* and *Logcat* of the navigation drawer.

12.1.1 CockpitFragment

The `CockpitFragment` reproduces the same layout of the instrument cluster display of the Ego vehicle and is loaded as soon as the connection with the dispatcher module has been established. As it happens for the display of the car, the view is initially set on the grid with the little car on the road.

The car is set through a message of type `V2X_INFO_1` sent by the dispatcher. Any other event will trigger the fetching and the visualization of the proper image or message, either on the grid layout or on the popup layout.

Popup and grid have been placed on two different `FrameLayouts`, for a better performance. Figure 12.3 shows the comparison between the screens displayed by the application and the ones displayed on the instrument panel of the Ego vehicle.



Figure 12.3: Stationary vehicle use case displayed on the application (left) and on the dashboard (right). (a)(b) show the grid view.

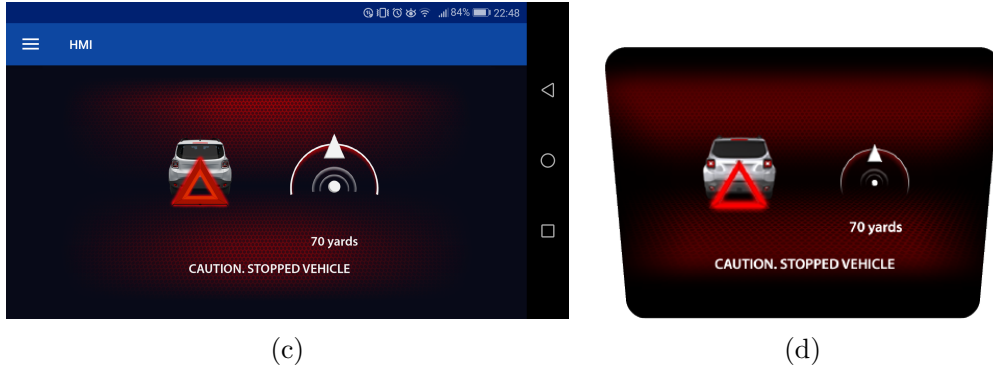


Figure 12.3: Stationary vehicle use case displayed on the application (left) and on the dashboard (right). (c)(d) show the popup view.

Figure 12.4 shows the layout bounds for each element that may be displayed. In fact, each of them has a fixed position on the screen and is stored in a specific data structure. It can happen that an image is stored in two or more key-value maps, in order to distinguish whether it has to be fetched for a popup screen or for the grid one.

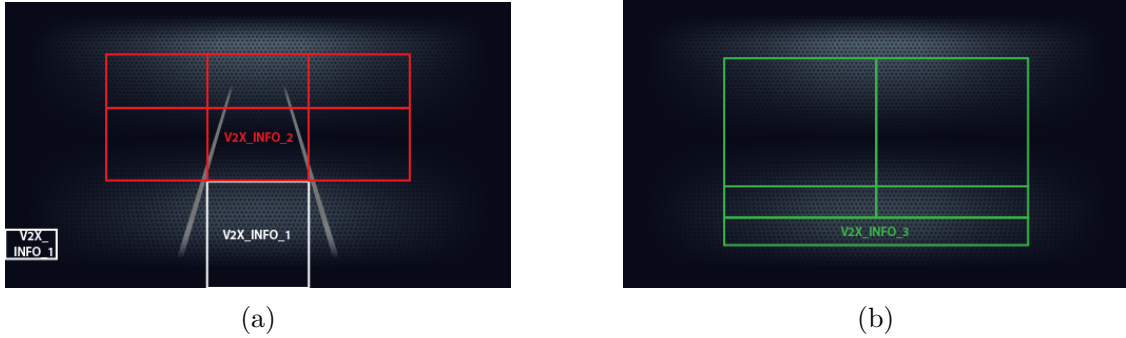


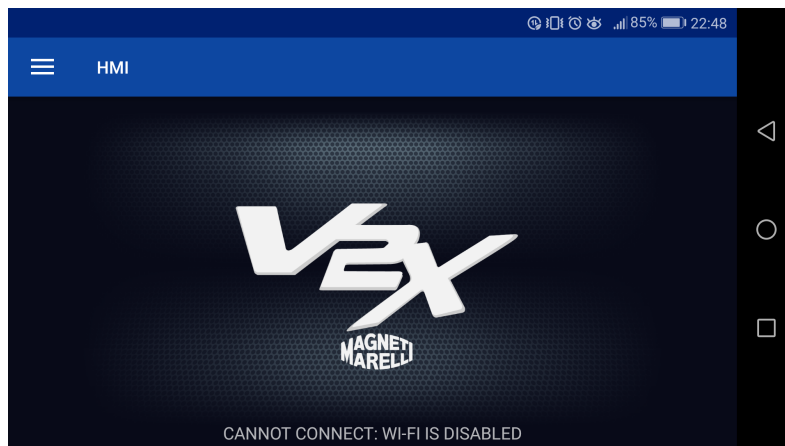
Figure 12.4: (a) Layout bounds for grid (a) and popup (b). The different colors represent a specific V2X_INFO message

12.1.2 SplashFragment

The **SplashFragment** is a **Fragment** that is used for warning the user about the connection issues that may arise when the Wi-Fi state of the mobile device changes. It can load different images and backgrounds for providing detailed information about the issue. Each screen is listed below, together with the condition that triggers its visualization:

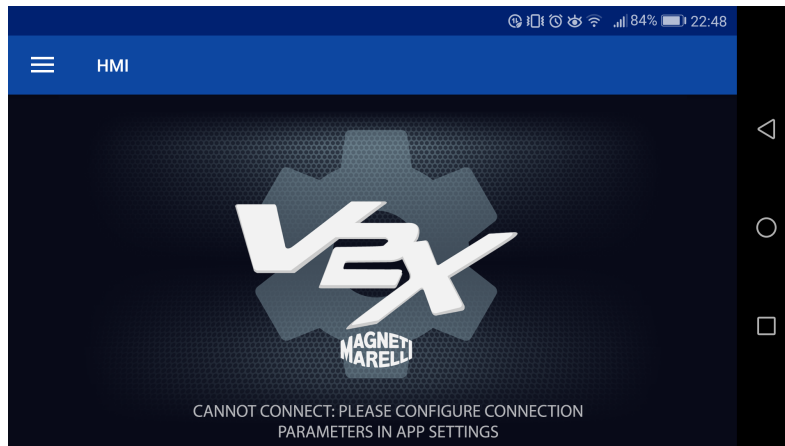
- *Wi-Fi disabled* (Figure 12.5a). The Wi-Fi radio of the mobile device is turned off.

- *No Wi-Fi networks available* (Figure 12.5c). The Wi-Fi radio is on but the device has not yet established a connection to a network or is performing the handshake for obtaining the IP address and other parameters.
- *App not yet configured* (Figure 12.5b). The Wi-Fi is on and connected but the application cannot establish the connection towards the dispatcher module because the parameters, that are the IP and the port, have not yet been configured.
- *Remote peer seems to be offline* (Figure 12.5d). The Wi-Fi is on and connected but the **Service** did not succeed in establishing the connection with the dispatcher module, probably because the latter is either not listening on the specified port or is offline. This screen appears after a certain number of connection attempts. In order not to drain the battery of the device, the **Service** is stopped and it can be restarted manually by the user by tapping the screen.
- *Connecting* (Figure 12.5e). The Wi-Fi is on and connected and the **Service** is trying to establish the connection towards the dispatcher module. The splash screen is animated in order to indicate that the application “is doing something” and it is not frozen.

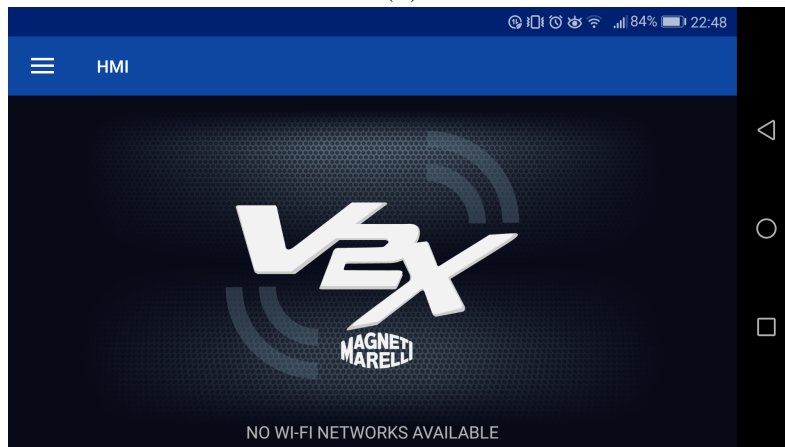


(a)

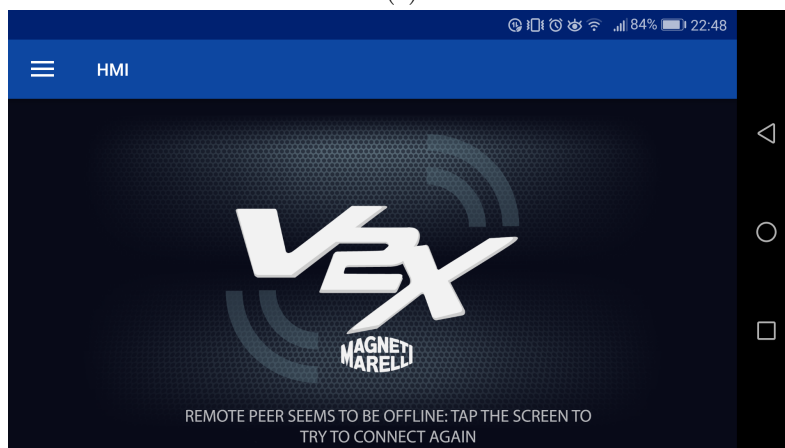
Figure 12.5: (a) Wi-Fi disabled;



(b)

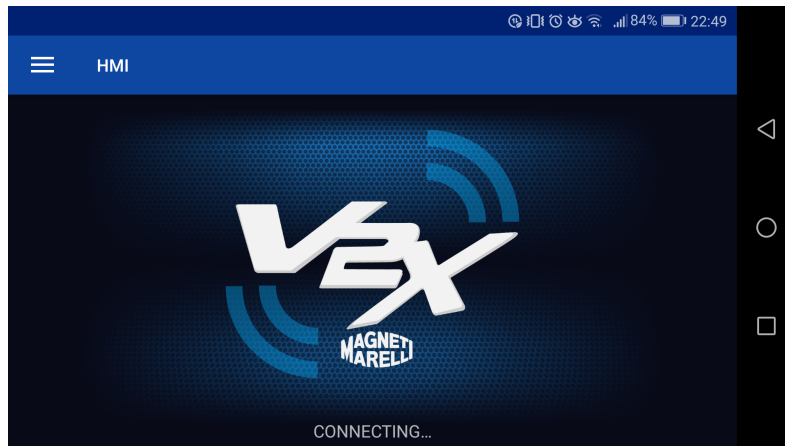


(c)



(d)

Figure 12.5: (b) No Wi-Fi networks: the grey waves signal the failed parameters negotiation attempts; (c) App not yet configured: the configuration problem is signaled through the gear behind the V2X logo; (d) Cannot connect due to peer probably offline: in this case the grey waves signal the failed connection attempts towards the dispatcher;



(e)

Figure 12.5: (e) Connecting: waves are blue and they are animated for signaling the ongoing connection attempt.

12.1.3 LogcatFragment

The `LogcatFragment`, shown in Figure 12.6, reproduces a console that captures all the logs printed through an interface overriding the `Android.Log` one. Currently all the messages received from the dispatcher are printed here, together with some debug messages concerning the state of the connection.

This `Fragment` adds two buttons to the application bar for allowing the user to filter and search for messages. The latter function has not yet been implemented.

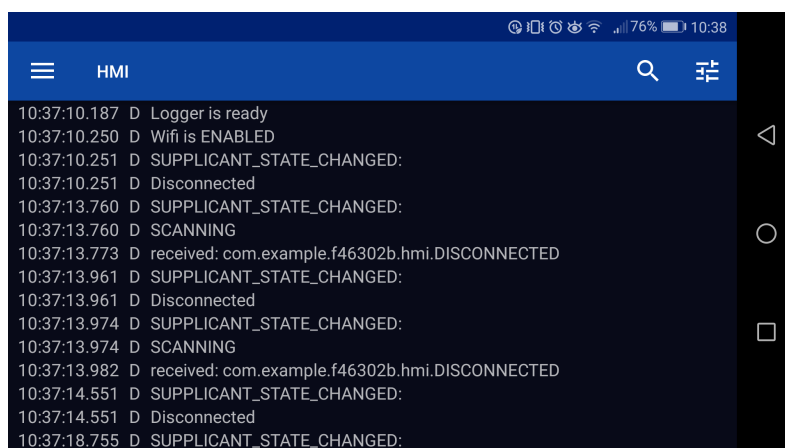


Figure 12.6: The console-like screen showing application logs.

The filtering button starts an **Activity** providing a list of priority types with check buttons (Figure 12.7). Each check allows the messages with that priority to be displayed in the console.

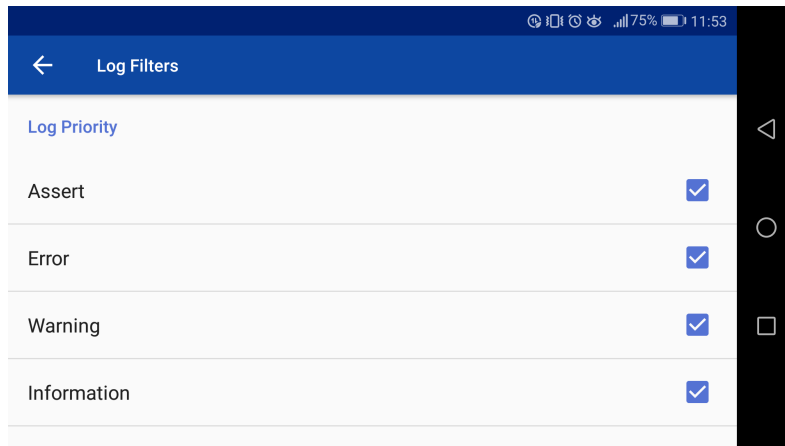


Figure 12.7: Log priority filters

12.2 SettingsFragment

The **SettingsFragment** (Figure 12.8) is the screen that lists the Settings of the application. Currently, such a list is made of only one entry that is *Connection parameters*, which opens a dialog for editing the IP address and port of the dispatcher module.

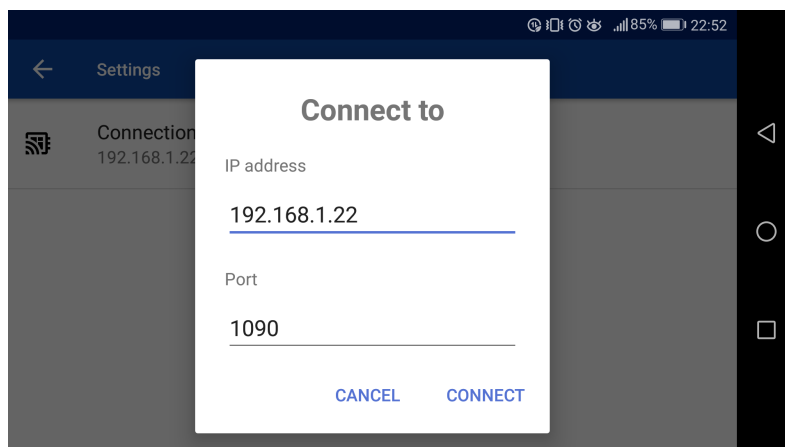


Figure 12.8: V2X HMI *Settings* screen

This **Fragment** is not attached to the **MainActivity** but to another one, in order to suggest that the Settings screen concerns the settings of all the application and not

just the **Fragment** currently on display. This cue is conveyed to the user thanks to the animated transition occurring between two **Activities**, which visually separates them.

12.3 IOService

The **IOService** is the one that manages the connection towards the dispatcher module, hence is started every time the Wi-Fi is turned on. Differently from the V2P application, it does not require to run also in memory critical conditions: the V2X HMI application is intended as a support for debugging, hence it assumes the user keeps it in foreground. When a low memory situation occurs the Android system tries to free memory by selecting as victims applications that are in background, hence it should not kill neither the V2X HMI application nor its **Service**.

Since Android **Services** run on the same thread of the Graphical User Interface, the operations on socket are delegated to two separate threads, which communicate with it through a message queue. Figure 12.9 shows the components instantiated by the **IOService**: the two threads and the handler of the message queue.

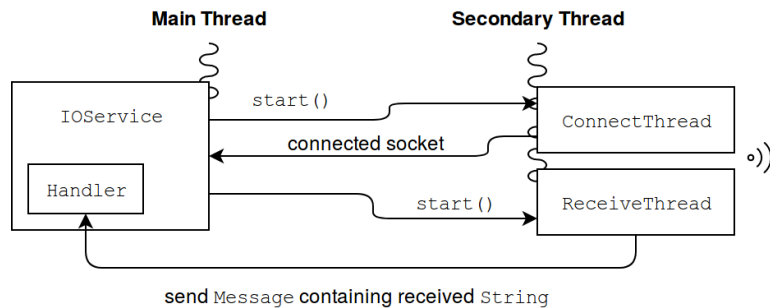


Figure 12.9: Threads of the V2X HMI application

12.4 WifiStateReceiver

The **WifiStateReceiver** is an extension of **BroadcastReceiver** that listens for state changes of Wi-Fi and, based on that, start or stops the **Service**. This **BroadcastReceiver** is dynamically instantiated by **MainActivity**, because it is not needed if the application is not running.

12.5 LocalV2XEventsReceiver

When the `IOService` is running and the dispatcher sends a message, the latter is passed from the receiving thread to the message handler. In order to pass that message to the cockpit, the handler exploits a local `BroadcastReceiver`: `LocalV2XEventsReceiver`. This `BroadcastReceiver` is a private class of `CockpitFragment` and it is registered and unregistered respectively when the `Fragment` is attached to and detached from the `MainActivity`. In fact, there is no need to process the `V2X_INFO` messages if the cockpit is not currently on display.

Since the `IOService` cannot know which `Fragment` is currently attached to `MainActivity`, it broadcasts the message anyway. On the other side, the broadcast will be received if the `LocalV2XEventsReceiver` is currently listening and properly processed through specific methods that will be discussed in the next chapter.

Chapter 13

Algorithm

This chapter will discuss the algorithm that allows the application to display the same alerts shown on the instrument cluster display of the *ego* car.

As introduced in Chapter 10, the dispatcher module of the V2X Communication board sends the same messages intended for the dashboard to all the devices connected to its port. This means that the syntax of such messages has been chosen based on the features provided by the dashboard, therefore the mobile application should implement similar ones.

Images and texts are saved on key-value maps, from which they are retrieved through an integer index used as key.

The message handler in the `Service` broadcasts the received strings as `Bundle` extras of the `RECEIVED Intent`.

If the `CockpitFragment` is currently attached to the `MainActivity`, the `Intent` will be received by the `LocalV2XEventsReceiver`, which extracts the information from the `Bundle` and, based on the type of the message, fills the suitable object: `V2XInfo1`, `V2XInfo2` or `V2XInfo3`.

The constructors of those objects are designed to automatically retrieve the correct resource ID from the maps, based on the specific use case. The list below describes the operations that each constructor takes when initializing the object:

- `V2XInfo1`. Since the *not present* icon uses two images (the car and the exclamation mark), the one depicting the exclamation mark must be loaded on cell (2,2) of the grid, hence there must be a variable in the `V2XInfo1` object that signals whether the object is representing such a use case or not;
- the grid-related objects, `V2XInfo2`, are characterized by the three images mapped to the *Stationary Vehicle* use case. The constructor assigns the resource ID to the

images accordingly to the column value in which the icon should be placed;

- also the popup-related objects, `V2XInfo3`, have a use case which is mapped to two images. The Front Collision Warning has two images, representing two different level criticalities for the distances between the two cars: a yellow one and a red one. The constructor loads the correct image by considering the color of the screen background: if it is yellow, then the level of criticality is medium, hence the yellow FCW image is loaded; if the background is red, then the situation is dangerous and the image should be red.

Once the received string has been translated into a specific object, the latter has to be drawn on the screen by calling the suitable method. For what concerns the three type of objects, `V2XInfo1`, `V2XInfo2`, `V2XInfo3`, the above-mentioned methods are `nomimetodi`.

setV2XStatusCar

The `setV2XStatusCar` method is in charge of setting the little car image below the grid view, based on the boolean variable of the `V2XInfo1` object passed as parameter.

setGridEvent

As the name suggests, the `setGridEvent` method manages the three images that can be loaded on the grid view. Each `V2XInfo2` message can contain the reference to up to three events. Based on the values of the coordinates of each cell, the method can either set or reset the image on the cell.

Since the reset operation requires that the coordinates are both set to zero, the application must remember the old ones in order to reset the correct cell.

setPopupEvent

This method is called when the received string is of type `V2X_INFO_3`. A popup is drawn on top of the grid view, since its layout is on a different `FrameLayout` object. Popups are displayed and removed from screen based on the value that sets the color of the background: yellow and red mean “*issue the popup*”, grey, instead, means “*delete popup*”.

Chapter 14

Design choices for the V2X Mirroring App

This chapter will illustrate the software choices concerning the development of the V2X Mirroring application, focusing on the reasons that led to select a strategy in spite of another.

14.1 Communication protocol: TCP

According to the requirements of this application, the communication with the dispatcher module shall use Wi-Fi and be connection oriented and reliable. For these reasons, the choice of the protocol fell on TCP, which also offers, beside the above-mentioned features, the ordered delivery of packets and error detection.

Since the car environment where the application will be used is very noisy, TCP is the best choice because it automatically handles transmission-level issues.

However, all those features increase a lot the overhead of the transmission, as the two peers need to send many service packets, which can have an impact on performances. The overhead of TCP led the researchers to consider also UDP for its higher speed in transmission. However, UDP is not connection-oriented, hence two peers just listen for datagrams. This is a drawback since the team is planning to add a feature to the application for remotely enabling and disabling the use case modules on the V2X Communication board and this would require the dispatcher to know which device is sending the instruction.

Moreover, UDP would have required both the dispatcher module and the application to implement error detection, reliability and strategies for packet reordering, which

would have ended in less performant programs.

14.2 Recycled Bitmap Images

One of the features of Java language is the automatic management of memory through the use of the garbage collector, which allows the programmer to forget about freeing memory allocated for no-more-needed variables. This characteristic is very useful but has a drawback: the garbage collection can be done at any time but not so frequently[6] hence, if the application loads many images, an out-of-memory exception can occur in the meantime.

Furthermore, Android applications must also take portability into account as they can be run on a big number of devices with different screen sizes and densities, therefore images and icons must fit all the situations.

The Android system can automatically adapt the layouts (and therefore the images) to the screen dimension, by reducing their sizes, even if this operation can be unefficient. For this reason the majority of the images used by the V2X Mirroring application has been managed with methods for bitmaps recycling made available by the Android APIs, which allow the reusing of the same blocks of memory to load different images. In fact, once the current image is no more needed, instead of having it garbage collected, its memory is reset and used for loading a new one.

Moreover, before being loaded into memory, images are sampled based on a reduction factor that fits the dimensions of the layout, allowing an additional saving of RAM.

14.3 Image and texts maps

Images and texts for the *cockpit* have been stored into a custom class extension of `SparseArray` data structure: `ImmutableSparseArray`.

`SparseArray` is an Android-typical key-value map, having integer numbers as keys and any `Object` as value. The peculiarity of this structure is that, unlike normal arrays of `Objects`, it allows to have gaps between indices and is more memory efficient than `HashMaps`, because it avoids auto-boxing of the keys and does not require an additional object to support each entry of the data structure[8].

Since, for the purposes of the V2X Mirroring application, the content of this structure should be immutable, the `SparseArray` class has been extended in order to override the methods that allow the editing of the entries.

A drawback of this structure is that instead of providing $O(1)$ access to its elements,

like `HashMaps`, it relies on binary search, whose access cost is $O(\log_2 N)$.

For what concerns this application however, this is not a problem because the number of entries for each structure goes from about ten to sixty, which makes the binary search more affordable with respect to the copious boxing and unboxing operations required to transform the `int` keys into `Integers` and vice versa for each map access.

Chapter 15

V2X HMI Application Testing

This chapter concludes the second part of this work by illustrating how the tests on the application have been carried out and which are the issues that emerged. The further improvements and future developments, as well as the obtained results, are discussed in Part III.

The testing on the application has been mainly done with the help of a stress test readjusted from the one used for the instrument cluster display of the Ego vehicle. The goal was to have all the use cases displayed sequentially in all the cells of the grid with an update rate of few tens of milliseconds, in order to test the response of the application.

At first, the same test used for the Ego vehicle was used, but it sent messages too fast, ending up in saturating the buffer of the TCP socket of the application.

The test has then been modified by increasing the lapse between two messages until the TCP buffer became able to handle all of them.

The reason why the test for the Ego vehicle has not been used is that the lapse between two messages was too low and the TCP buffer of the application was not able to manage all that traffic, resulting in a denial-of-service even for the graphical user interface. In fact, the main thread was overloaded too because it had to process all the broadcast `Intents` coming from the background `Service`.

The test written specifically for the application, instead, simulated a more realistic situation, where no more than five or six messages are sent in a second.

In this case, the application behaved correctly, with rare delays due to the instability of the Wi-Fi signal.

Later, the application was also successfully tested on the car, by simulating all the available use cases.

15.1 Graphical User Interface Testing

The user interface was tested by following the Agile Exploratory testing strategy, by using the application trying to break it[26]: i.e. by inserting alphabetic characters in text fields expecting only numeric inputs, etc.

Tests have been carried out before each intermediate delivery.

15.2 *Logcat* Testing

The testing of the log screen gave some problems of visualization when the events were too fast, even when running the app-customized test. Those issues were due to the implementation of the priority filter, which does not allow the addition of new lines while performing the filtering operations.

This is not acceptable but, currently, it does not represent a big issue since the main goal of the application, that is the mirroring of the instrument cluster display, has been achieved.

Part III

Results

Chapter 16

Results

This chapter concludes the work done for this thesis by first recalling the goals and then by reporting the obtained results and introducing further improvements and possible future developments.

This thesis required the development of two mobile applications addressing two aspects of the V2X technology.

The first one, was related to the communication between a vehicle and a pedestrian, with the help of a mobile device. The goal of the first application was to bring the V2P connectivity on smartphones through a special communication board developed by Magneti Marelli, in order to notify users about the presence of nearby vehicles. As a requirement, once started by the user, it had to require little maintenance and be able to auto-start after phone reboot. For this reason it has been implemented as a home screen widget, so that it is handled by the system's home screen application and its state is always visible both as notification and on the phone's main screen.

The second aspect concerned the V2X system installed on test cars, which needed to be debugged in a more comfortable way, by allowing the researchers to see the V2X events displayed by the instrument cluster also on a tablet. The mirroring of the Human-Machine-Interface of the instrument cluster was indeed the goal of the second application, which was also prepared for displaying log messages. For a better user experience, the application is only available in landscape mode, due to the massive use of images and texts.

Both applications met all the defined requirements and, once tuned, they behaved correctly during the tests. In addition, thanks to the modular design, their components can be easily modified or enhanced without requiring changes to the whole architecture.

16.1 Further Improvements

Some examples of improvements can be made for both applications.

The V2P one, for example, could optimize the RAM usage by managing images through the recycle bitmap methods made available by the Android APIs and could enhance user experience by showing badges signaling approaching vehicles onto its widget layout.

For what concerns the V2X HMI application, the most straight away improvement could be the addition of regular expression-based filtering and logs highlighting features to the *logcat* screen.

16.2 Future Developments

The addition of more complex features can be scheduled as a future development.

Thanks to its modularity, the V2P application can easily switch from the SkylocTM architecture to the DSRC one and, later, even to the C-V2X communication. The latter would allow it also to abandon the V2P DSRC board in favor of the cellular antenna of the mobile device itself. Beside that, the support for different vehicle categories could be added, together with their priority levels, in order to have the application behave as its counterpart on vehicles already does.

Also the V2X HMI application can be expanded with the addition of further modules. In order to ease the debug of the V2X Communication board installed on test vehicles, the *Modules* management screen could be added, allowing the researchers to remotely enable and disable specific parts of the V2X Framework, without needing to physically connect to the car's control unit with their laptops.

As soon as C-V2X will become widespread, this application could take advantage of that by implementing a top view on a map of V2X events, potentially by giving the user the possibility to filter them based on different choices.

Bibliography

- [1] *AgileAdvice / User Stories and Story Splitting*. 2014. URL: <http://www.agileadvice.com/2014/03/06/referenceinformation/user-stories-and-story-splitting/>.
- [2] Ahmed-Zaid et al. *Vehicle Safety Communications – Applications (VSC-A) Final Report*. Tech. rep. DOT HS 811 492A. NHTSA, Sept. 2011.
- [3] Wenshuang Liang et al. “Vehicular Ad Hoc Networks: Architectures, Research Issues, Methodologies, Challenges, and Trends”. In: *International Journal of Distributed Sensor Networks* Volume2015 (2014).
- [4] *Android Developers / Icon Design*. 2018. URL: https://developer.android.com/guide/practices/ui_guidelines/icon_design.html.
- [5] *Android Developers / Message*. 2018. URL: <https://developer.android.com/reference/android/os/Message.html>.
- [6] *Android Developers / Overview of Android Memory Management*. 2018. URL: <https://developer.android.com/topic/performance/memory-overview.html>.
- [7] *Android Developers / Service*. 2018. URL: <https://developer.android.com/guide/components/services.html>.
- [8] *Android Developers / SparseArray*. 2018. URL: <https://developer.android.com/reference/android/util/SparseArray.html>.
- [9] *Android Developers / Widgets*. 2018. URL: <https://developer.android.com/design/patterns/widgets.html>.
- [10] *Android Developers / Widgets*. 2018. URL: <https://developer.android.com/guide/topics/appwidgets/index.htm#AppWidgetProvider1>.
- [11] *Android Studio*. 2018. URL: <https://developer.android.com/studio/index.html>.
- [12] *Blender / About*. 2018. URL: <https://www.blender.org/about>.

- [13] *Bluetooth Technology Website / Proximity and RSSI*. 2015. URL: <http://blog.bluetooth.com/proximity-and-rssi>.
- [14] *Electronic Design / DSRC vs. C-V2X: Looking to Impress the Regulators*. 2017. URL: <http://www.electronicdesign.com/automotive/dsrc-vs-c-v2x-looking-impress-regulators>.
- [15] *ETSI / Automotive Intelligent Transport Systems*. 2018. URL: <http://www.etsi.org/technologies-clusters/technologies/automotive-intelligent-transport>.
- [16] *European Commission / Cooperative, connected and automated mobility (C-ITS)*. 2018. URL: https://ec.europa.eu/transport/themes/its/c-its_en.
- [17] Andreas Festag. "Cooperative Intelligent Transport Systems in Europe". In: *IEEE Communication Magazine* 53 (2015), pp. 64–70.
- [18] *Innovation Destination Automotive / 5 Things You Should Know about the DOT V2V Mandate*. 2017. URL: <http://innovation-destination.com/2017/10/16/5-things-know-dot-v2v-mandate/>.
- [19] *Interaction Design / What is Usability Testing?* 2018. URL: <https://www.interaction-design.org/literature/topics/usability-testing>.
- [20] *ISO / Definitions: 3.1 usability*. 1998. URL: <https://www.iso.org/obp/ui/#iso:std:iso:9241:-11:ed-1:v1:en>.
- [21] *Magneti Marelli / Company*. 2018. URL: <http://www.magnetimarelli.com/company>.
- [22] *Material Design / Environment*. 2018. URL: <https://material.io/guidelines/material-design/environment.html>.
- [23] *Material Design / Guidelines*. 2018. URL: <https://material.io/guidelines/>.
- [24] *NHTSA / U.S. DOT advances deployment of Connected Vehicle Technology to prevent hundreds of thousands of crashes*. 2016. URL: <https://www.nhtsa.gov/press-releases/us-dot-advances-deployment-connected-vehicle-technology-prevent-hundreds-thousands>.
- [25] *NHTSA / Vehicle-to-Vehicle Communication / What is V2V?* 2016. URL: <https://www.nhtsa.gov/technology-innovation/vehicle-vehicle-communication>.
- [26] *QASymphony / Agile Methodology: The Complete Guide to Understanding Agile Testing*. 2018. URL: <https://www.qasymphony.com/blog/agile-methodology-guide-agile-testing/>.

- [27] *ReQtest / Functional Requirements vs. Non Functional Requirements*. 2012. URL: <https://reqtest.com/requirements-blog/functional-vs-non-functional-requirements/>.
- [28] *ReQtest / Integrating interaction design in agile development*. 2014. URL: <https://reqtest.com/agile-blog/integrating-interaction-design-in-agile-development>.
- [29] *ReQtest / The Goal of Agile Methodology*. 2015. URL: <https://reqtest.com/agile-blog/requirements-in-agile-software-development>.
- [30] *ReQtest / Using User Stories to Document Requirements*. 2014. URL: <https://reqtest.com/agile-blog/using-user-stories-to-document-requirements>.
- [31] *Skypersonic / Safe Drone Technology*. 2018. URL: <http://www.skypersonic.com/>.
- [32] *Vogella / Android (Home Screen) Widgets*. 2016. URL: <http://www.vogella.com/tutorials/AndroidWidgets/article.html>.