



André Ribeiro Martins
Marques Mourato

Cidades, pessoas e veículos como uma plataforma
City, People and Vehicles as a Platform



Universidade de Aveiro
2021

**André Ribeiro Martins
Marques Mourato**

**Cidades, pessoas e veículos como uma plataforma
City, People and Vehicles as a Platform**

Dissertação apresentada à Universidade de Aveiro para cumprimento dos requisitos necessários à obtenção do grau de Mestre em Engenharia de Computadores e Telemática, realizada sob a orientação científica do Doutor Pedro Filipe Vieira Rito, Investigador Auxiliar do Instituto de Telecomunicações de Aveiro, da Doutora Susana Isabel Barreto de Miranda Sargento, Professora catedrática do Departamento de Eletrónica, Telecomunicações e Informática da Universidade de Aveiro, e colaboração do Engenheiro Francisco Manuel Malheiro de Castro, V2X Product Owner da Bosch Car Multimédia Portugal, S.A.

Dedico este trabalho à minha irmã, Marta

o júri / the jury

presidente / president

Professor Doutor Paulo Jorge Salvador Serra Ferreira
Professor Associado, Universidade de Aveiro

vogais / examiners committee

Professor Doutor Fernando Pedro Lopes Boavida Fernandes
Professor Catedrático, Faculdade de Ciências e Tecnologia da Universidade de Coimbra

Professora Doutora Susana Isabel Barreto de Miranda Sargento
Professora Catedrática, Universidade de Aveiro

agradecimentos / acknowledgements

Agradeço acima de tudo à minha família que tanto sacrificou para que eu pudesse concretizar os meus sonhos. Queria agradecer em especial ao meu pai, à minha mãe e à minha irmã por todo o apoio e carinho que me deram, sem o qual nada disto seria possível.

À professora Susana Sargento, agradeço todas as oportunidades que me proporcionou ao longo do meu percurso académico e que culminaram no desenvolvimento deste projeto. Estas oportunidades permitiram-me adquirir competências extremamente valiosas e tornaram-me num profissional melhor.

Ao Doutor Pedro Rito, agradeço pela sua disponibilidade e pelo excelente trabalho de mentoria que me proporcionou ao longo do desenvolvimento desta dissertação.

Ao Francisco Castro, pela oportunidade de poder estagiar na Bosch e por todo o acompanhamento ao longo deste projeto.

Agradeço também à equipa fantástica do Instituto de Telecomunicações com quem tive o prazer de trabalhar, em especial ao Rodrigo Rosmaninho e ao Gonçalo Vitor por toda a ajuda e por toda a paciência que tiveram comigo.

Ao João, Vasco, Miguel e Marcelo, irmãos de mães diferentes, por todo o apoio, especialmente nos momentos mais difíceis.

Aos meus afilhados Lúcia, Raquel e Guilherme, por todo o carinho que me deram. Por fim, gostava de agradecer à Joana, pela paciência que teve para ouvir as minhas frustrações, hesitações e desabafos. Por estar lá quando as coisas corriam bem e quando as coisas corriam mal. Um obrigado por tudo o que me ensinou nestes últimos 2 anos.

Este trabalho contribuiu para a plataforma Aveiro Tech City Living Lab, desenvolvida através da European Urban Innovative Actions programme Aveiro STEAM City (UIA03-084).

Palavras Chave

Internet das Coisas, Comunicação V2X, Cidades Inteligentes, Publish-Subscribe, Computação no Edge, Informação Histórica, Informação em Tempo Real

Resumo

A evolução dos mecanismos de comunicação sem fios e o aparecimento de sistemas embebidos com cada vez maior capacidade de computação permitem a criação de uma infraestrutura que, de forma confiável e distribuída, consiga agregar a informação proveniente de vários dispositivos móveis ou estáticos. Comunicação V2X é uma tecnologia emergente e inovadora que permite a troca de mensagens de cooperação, alerta e multimédia num ambiente veicular sem fios. Esta tecnologia conecta um veículo, não só a outros veículos, mas também a infraestruturas e utilizadores rodoviários vulneráveis (motociclos, velocípedes e/ou peões), cobrindo assim, um número vasto de aplicações desde os tradicionais casos de infotainment até às funções mais avançadas de condução cooperativa e assistida. Além dos nós móveis referidos, é também possível usufruir de informação proveniente de nós estáticos, como por exemplo, radares e câmaras que sejam instalados em determinadas zonas da cidade. De forma a complementar os diversos mecanismos de comunicação sem fios, é também possível a instalação de unidades de comunicação fixas, posicionadas perto da estrada, cujo objetivo é recolher a informação proveniente de veículos com comunicação V2X que estejam ao alcance dos recetores. Este tipo de nó recetor pode também servir como um nó lógico, que não só encaminha a informação recebida para a infraestrutura, mas também faz um processamento distribuído (edge computing) desta informação, podendo ele mesmo determinar se um veículo está a deslocar-se a uma velocidade acima da velocidade permitida para aquele local, por exemplo, e gerar um aviso se isso acontecer. Nesta dissertação foram desenvolvidos serviços que permitem a comunicação V2I e I2V em veículos, usando as normas definidas para os sistemas de transporte inteligentes, e também foram desenvolvidos serviços que permitem à infraestrutura receber informação heterogénea e agregá-la num grupo de base de dados. Esta informação é, também, disponibilizada a serviços de alto nível tanto sob a forma de informação em tempo real como sob a forma de informação histórica. A informação é recolhida por estações fixas instaladas no testbed do Aveiro Living Lab que estão distribuídas pela cidade. A solução proposta disponibiliza informação e eventos em tempo real, prontos a serem consumidos por serviços de alto nível, assim que forem recebidos pelas estações fixas. Para demonstrar a versatilidade deste tipo de informação, desenvolvemos a dashboard Aveiro in Real-Time, que pode ser usada por cidadãos ou por operadores da cidade e permite mostrar aos utilizadores o estado atual da cidade. Esta solução também disponibiliza informação histórica, que pode ser usada em diversos casos de uso. Para demonstrar o quão potente e versátil esta solução é, desenvolvemos o serviço Incident Replayer, que permite a reprodução da informação dos veículos que estiveram envolvidos num acidente.

Keywords

Internet of Things, V2X communication, Smart cities, Publish-Subscribe, Edge Computing, Historic information, real-time information

Abstract

The evolution of the wireless communication mechanisms and the appearance of embedded systems with growing computational power allow the creation of an infrastructure that, in a reliable and distributed way, is able to aggregate data acquired from multiple static and dynamic devices. The V2X communication is an emerging and innovative technology that allows the exchange of cooperative alert and multimedia messages in a wireless vehicular environment. This technology connects a vehicle, not only to other vehicles, but also to the infrastructure and to vulnerable road users (motorcycles, bicycles and pedestrians), covering a vast number of applications from the traditional cases of infotainment to the most advanced operations of cooperative and assisted driving. It is also possible to make use of the data acquired from the static nodes, such as, radars and video cameras, which are installed in certain zones of the city. To better complement the multiple wireless communication mechanisms, it is also possible to install roadside units, located near the road, whose main objective is to gather data from the V2X vehicles that are within range of the receptors. This type of receptor node can also work as a logic node that, not only routes the received information to the infrastructure, but also performs distributed computation (edge computing) on this data, such as detecting if a vehicle is moving at a speed that is above the speed limit for that road, and generating an alert if that happens. In this dissertation we developed services that enable V2I and I2V communication in vehicles, making use of the standards for Intelligent Transportation Systems, as well as services that allow the infrastructure to receive heterogeneous data and aggregate it in a database cluster. We also make this data available to high level services both as real-time data and historic data. The data is acquired by roadside units that were installed in the Aveiro Living Lab testbed and are distributed throughout the city. The proposed solution leverages real-time data and events, which are ready to be consumed by services as soon as they are received by the roadside units. To demonstrate the versatility of this type of information, we developed, the Aveiro in Real-Time dashboard that can be used by citizens or by city operators that want to visualize the state of their cities. This solution also leverages historic data, which can also empower multiple use cases. To show how powerful and versatile our solution is, we have developed the Incident Replayer service, that allows users to replay the information of the vehicles that were involved in an accident.

Contents

Contents	i
List of Figures	v
List of Tables	vii
Glossary	ix
1 Introduction	1
1.1 Context and Motivation	1
1.2 Objectives	2
1.3 Dissertation Structure	2
2 Background and Related Work	5
2.1 Vehicular communication	5
2.1.1 V2I Communication	5
2.1.2 I2V Communication	6
2.1.3 V2V Communication	7
2.2 Vehicular-based Messages	8
2.2.1 Cooperative Awareness Messages (CAM)	8
2.2.2 Decentralized Environmental Notification Message (DENM)	9
2.2.3 Collective Perception Messages (CPM)	10
2.3 Related Work	10
2.4 Aveiro Tech City Living Lab	13
2.4.1 On Board Units	13
2.4.2 Roadside Units	14
2.4.3 Infrastructure	15
2.5 Summary	16
3 Architecture	17
3.1 Requirements	17

3.1.1	Vehicular data acquisition	17
3.1.2	Sensor integration	18
3.1.3	Access to real-time and historic data	18
3.1.4	I2V communication	18
3.2	Proposed architecture	19
3.2.1	On board unit	20
3.2.2	Edges	21
3.2.3	Core	23
3.3	Module communication	24
3.3.1	Message types	24
3.3.2	Message formats	25
3.3.3	Sending CAMs and DENMs through an RSU edge	26
3.3.4	Sending CAMs through a gateway edge	27
3.3.5	Sending CAMs from the OBU to the MQTT Bridge directly	27
3.3.6	Sending CAMs and DENMs to the core	28
3.3.7	Sending DENMs from the core to the RSUs	29
3.3.8	Sending DENMs from the RSUs to the OBUs	29
3.4	Summary	30
4	Implementation	31
4.1	ITS modules	31
4.1.1	ITS Router	32
4.1.2	MQTT Service	36
4.1.3	Vehicle Service	41
4.1.4	Eth2Udp & Udp2Eth	45
4.2	Message Queuing Telemetry Transport	48
4.2.1	MQTT Edge	48
4.2.2	MQTT Bridge	49
4.3	Core	50
4.3.1	City Manager	51
4.3.2	Mapping Service	52
4.4	Notification App	57
4.5	Development, deployment & testing	58
4.6	Summary	58
5	Aveiro in Real-Time	61
5.1	State of the city in real-time	61
5.2	Dashboard	61

5.2.1	Side Menu	62
5.2.2	The map	63
5.2.3	The entity page	63
5.3	Entities	64
5.3.1	Communication Vehicles	64
5.3.2	Vulnerable Road Users	68
5.3.3	Road Side Units	69
5.3.4	Radar Vehicles	71
5.4	Data flows	72
5.4.1	Dynamic data	73
5.4.2	Static data	74
5.5	Sending alerts to vehicles	75
5.5.1	Generating DENMs	76
5.6	Receiving data using V2V communication and LTE/5G	77
5.7	Summary	78
6	Incident Replayer	79
6.1	Replaying data	79
6.1.1	Message sniffer	79
6.1.2	Message generator	80
6.2	Requirements	81
6.2.1	Allow users to replay incidents	81
6.2.2	Allow the user to control the replay	82
6.2.3	Store the historic data in an efficient way	82
6.2.4	Access the historic data in an efficient way	82
6.3	Graphical User Interface	82
6.3.1	Incident menu	83
6.3.2	Replay menu	84
6.4	Data flow	85
6.4.1	Storing historic data	85
6.4.2	Accessing historic data	85
6.5	Summary	86
7	Results	87
7.1	Use cases	87
7.1.1	Scenario 1 - Sending vehicle data to the infrastructure via DSRC	87
7.1.2	Scenario 2 - Sending alerts to the vehicles	88
7.2	Testing environment	89

7.2.1	V2I communication	89
7.2.2	I2V communication	90
7.3	Equipment	91
7.4	Results	92
7.4.1	Scenario 1 - Sending vehicle data to the infrastructure via DSRC	92
7.4.2	Scenario 2 - Sending alerts to the vehicles	93
7.5	Summary	97
8	Conclusion and Future Work	99
8.1	Conclusion	99
8.2	Future Work	100
	References	103

List of Figures

2.1	Vehicular communication - Vehicle data is sent to the infrastructure	6
2.2	Vehicular communication - Infrastructure data is sent to the vehicles	7
2.3	Vehicular communication - Vehicle data is sent to other vehicles	7
2.4	General structure of a CAM (from [7])	9
2.5	General structure of a DENM (from [8])	9
2.6	General structure of a CPM (from [9])	10
2.7	On Board Unit	13
2.8	The board of the RSU without the case protection	14
2.9	Wall Boxes. Design and deployment in the city	15
2.10	Smart Lamp Post	15
2.11	Roadside Units in Aveiro	16
3.1	Architecture - Overview	19
3.2	Architecture - On board unit	21
3.3	Architecture - Edges	22
3.4	Architecture - Core	23
3.5	Wireshark capture of a CAM	26
3.6	OBU/RSU Edge connection	27
3.7	OBU/Gateway Edge connection	27
3.8	OBU/MQTT Bridge connection	28
3.9	Core connections	28
3.10	Data flow between the core and the RSUs	29
3.11	Data flow between the RSU an the OBU	30
4.1	MQTT Service data flow	41
4.2	Vehicle Service data flow	45
4.3	MQTT Bridge fetches information from all RSUs, which the services can then collect	50
4.4	City Manager - Architecture	52
4.5	Area of effect - Car 33, Bus 60 and Bus 50 received the DENM, since they were within the area of effect.	53

4.6	Geographic Mapping Strategy and Last Seen Mapping Strategy implement specific algorithms for routing the ITS message	54
4.7	Active RSUs within 300 meters of the epicenter of the DENM were chosen	56
4.8	Mapping Service - Data Flow Diagram	57
4.9	Pipeline of the ITS Router that contains the build, test and deployment stages	58
5.1	Aveiro in Real-Time - Dashboard	62
5.2	ART - Side Menu	62
5.3	ART - Map	63
5.4	ART - Entity Page	64
5.5	Communication Vehicles with lines drawn	65
5.6	Passenger Car (5) displayed in the dashboard	67
5.7	Buses (6) displayed in the dashboard	67
5.8	Two moliceiros sailing side by side in Ria de Aveiro	68
5.9	Vulnerable Road Users near P3 - Ponte Dubadoura	69
5.10	Radar Vehicles detected in P3 - Ponte Dubadoura	71
5.11	Aveiro in Real-Time - Architecture	73
5.12	DENM generated from the Aveiro in Real-Time dashboard	76
5.13	Alert displayed in the notification app	77
5.14	Bus with 5G sends vehicle data of bus with no internet connection	78
6.1	Message sniffer and generator flow	81
6.2	Replay of the approach of an emergency vehicle on 15-10-2021 15:10:22	83
6.3	IR - Incident Menu	84
6.4	IR - Replay menu	84
6.5	Incident Replayer - Data Flow	86
7.1	V2I - Sending vehicle data to infrastructure	88
7.2	I2V - Sending alerts to the vehicles	89
7.3	Both V2I and I2V data flows	91
7.4	Scenario 1 - Analysis	93
7.5	Scenario 2 - Analysis	94
7.6	E2E test 1 - Performance	96
7.7	E2E test 2 - Performance	97

List of Tables

4.1	ITS Router - Configuration parameters	34
4.2	ITS Router - Input and output port mapping	36
4.3	MQTT Service - Configuration parameters	38
4.4	MQTT Service - Input and output port mapping	40
4.5	Vehicle Service - Configuration parameters	43
4.6	E2U - Configuration parameters	46
5.1	Communication Vehicle types	66
5.2	Communication Vehicle types	69
5.3	RSU selected menu	70
5.4	Roadside Unit types	71
5.5	Roadside Unit types	71
5.6	Radar Vehicle types	72
7.1	RSU/OBU - Hardware specifications	91
7.2	Core - Hardware specifications	92
7.3	Scenario 1 - Results	93
7.4	Scenario 2 - Results	94
7.5	E2E test 1 - Input parameters	95
7.6	E2E test 2 - Input parameters	97

Glossary

AOE	Area of Effect	I2V	Infrastructure-to-vehicle
API	Application Programming Interface	IoT	Internet of Things
ART	Aveiro in Real-Time	IR	Incident Replayer
ASCII	American Standard Code for Information Interchange	IT	Institute of Telecommunications
BTP	Basic Transport Protocol	ITS	Intelligent Transportation Systems
CAGR	Compound annual growth rate	JSON	JavaScript Object Notation
CAM	Cooperative Awareness Message	LTE	Long-Term Evolution
CM	City Manager	MQTT	Message Queuing Telemetry Transport
CPM	Collective Perception Message	OBU	On Board Unit
CV	Communication vehicles	OEM	Original equipment manufacturer
DENM	Decentralized Environmental Notification Message	REST	Representational State Transfer
DSRC	Dedicated short-range communications	RSSI	Received signal strength indication
E2U	Ethernet-To-Udp	RSU	Roadside Unit
ETSI	European Telecommunications Standards Institute	SEO	Search Engine Optimization
GDP	Gross Domestic Product	SLP	Smart Lamp Post
GPS	Global Positioning System	U2E	Udp-To-Ethernet
GUI	Graphical User Interface	UDP	User Datagram Protocol
HMI	Human to Machine Interface	V2I	Vehicle-to-infrastructure
HTTP	Hypertext Transfer Protocol	V2V	Vehicle-to-vehicle
		V2X	Vehicle-to-everything
		VAM	VRU Awareness Message
		VRU	Vulnerable Road User

Introduction

This chapter provides the context and motivations for the development of this dissertation, as well as its objectives and the contributions that were achieved. After that, it will explain the structure of this document.

1.1 CONTEXT AND MOTIVATION

According to [1], in 2018, an estimated 55.3% of the world's population lived in urban settlements, and by 2030, it is estimated that this percentage will rise to 60%.

This increase of the urban population leads to a higher percentage of economic growth generated by cities. According to [2], the most dense areas, not surprisingly, represent the world's megacities and each of which represents a very large percentage of national GDP, thus having efficient infrastructures and smart technology can have a very positive economic impact[3].

According to Berg Insight's [4], more than 51% of all new cars sold worldwide in 2019 were equipped with an OEM embedded telematics system, up from 38% in 2018, and expected to reach up to 86% by 2025, with companies such as GM, BMW, MercedesBenz and PSA being the leading adopters of embedded telematics, offering the technology as a standard feature across models and geographies. The global automotive telematics market size was valued at \$50.4 billion in 2018, and is projected to reach over \$100 billion by 2022 and \$320.6 billion by 2026, registering a compound annual growth rate (CAGR) of 26.8% from 2019 to 2026. The OEM segment was the highest revenue contributor in 2018, accounting for \$33.7 billion, and is estimated to reach \$225.6 billion by 2026, registering a CAGR of 27.9% during the forecast period.

There is a clear upward trend toward the inclusion of embedded telematics in every car and appears to be growing every year. The growing support for the inclusion of this technology opens up new and interesting possibilities to improve people's lives, whether it is by providing new ways of entertainment and social engagement, or by providing new solutions that will improve driver safety.

V2X communication has been proven with day-one applications, where data related to hazardous situations can be transmitted to nearby vehicles and to the infrastructure that will be able to relay that information to vehicles that are farther away. Still, the full potential of the V2X technology is yet to be unlocked, as the real amount of data available to the vehicles is much larger than the data that is exchanged between them.

V2X can be used to get and send data in an automotive environment, both to prevent problems such as accidents, and also to provide knowledge of what happened when an accident is in place.

1.2 OBJECTIVES

The main objective of this dissertation is to develop services that enable V2X communications in vehicles, collect data from heterogeneous sources, such as vehicles, radars and cameras, aggregate it in a centralized way, where they can be later consumed by other high-level applications. To test the versatility of this platform, this dissertation also proposes and develops some services that consume the vehicle and sensor data, and make use of both the V2I and Infrastructure-to-vehicle (I2V) technologies. The V2I communication will allow to display information in real-time to the user, and will also allow to store historic information so that it can be used later. The I2V communication will allow to notify drivers of potential dangerous driving conditions, which may prevent accidents.

Both of these use cases will be developed using existing and well defined standard formats for vehicle data, like the Cooperative Awareness Messages (CAM) and Decentralized Environmental Notification Message (DENM).

The objectives of this dissertation can be summarized as follows:

- Definition and extension of communication messages between the vehicles and the infrastructure.
- Development of services that enable V2I and I2V communication in vehicles and in the infrastructure.
- Development of a data aggregator where all the vehicle and mobility sensor data will be made available, both as real-time and as historic data.
- Development of a GUI that displays both the real-time and non real-time information in a clear and concise way.
- Creation of a test suite to test the proposed solution.

1.3 DISSERTATION STRUCTURE

The content of this dissertation is organized into multiple chapters.

Chapter 2 (Background and Related Work) provides the context to this dissertation. It explains the work that has previously been developed, and also provides an overview of similar or related solutions in the state of the art.

Chapter 3 (Architecture) lists the necessary requirements that the solution must meet and proposes a possible architecture for this solution. Then, it provides an in-depth analysis of the different data flows that this architecture supports.

Chapter 4 (Implementation) provides an in-depth analysis of all the individual modules, while explaining their function in the overall architecture. In the end it analyses the development, deployment and testing environments that allowed to work fast and efficiently while mitigating the number of issues in the development, by submitting the software to rigorous tests and following some of the most well known good practices of software development.

Chapter 5 (Aveiro in Real-Time) provides the first functional results, by presenting a web app that displays the real-time information about the city, and also to allow the user to send alerts to the vehicles. In the end we also show a second web app that was developed, the notification app, that runs in the OBU and whose purpose is to serve as an HMI that visually notifies the driver after receiving an alert from the infrastructure.

Chapter 6 (Incident Replayer) presents a platform that allows users to replay certain incidents, by accessing historic data that was received from multiple data sources, like vehicles, radars, cameras, and others.

Chapter 7 (Results) presents the results obtained after doing performance tests against the proposed solution. In this chapter we describe our testing environment, the laboratory testbed and we also present the specifications of the hardware that was used.

Finally, chapter 8 (Conclusion) summarises the work that was developed, how it met our initial expectations and how it innovates with respect to current platforms. We also present ideas for future work.

Background and Related Work

In this chapter, we provide the context in which this dissertation is inserted, and also explain some background concepts that are necessary to understand it. We will present an overview of the V2X technology and what it can achieve along with this growing demand for more telematics in personal vehicles. Then, it presents similar solutions in the related work. Finally, it presents the Aveiro Tech City Living Lab.

2.1 VEHICULAR COMMUNICATION

Manned and autonomous vehicles can talk to each other, to pedestrians and the infrastructure using direct link communication technology called V2X, or Vehicle-to-everything communication. V2X is a growing area of communication between vehicles, either directly or through the so called roadside units, which are, as the name suggests, equipments usually placed near the side of the road, and may serve to relay vehicle messages to other vehicles or to relay these messages to a cloud infrastructure. It can prevent road accidents and save lives and, in addition, improve traffic congestion, enhance mobility and reduce emissions.

If coupled with autonomous vehicles, V2X allows the communication with other unmanned vehicles, which have a huge amount of applications.

The inverse flow of data is also possible, also known as I2V, where messages are sent from the infrastructure, not necessarily in the cloud, and are received by the vehicles. As we have mentioned previously, these messages could be used to give the driver some sort of information about the current driving conditions.

2.1.1 V2I Communication

Vehicle-to-Infrastructure, or V2I communication, is the wireless exchange of data between vehicles and the road infrastructure. V2I alone only describes the inward flow of data, i.e., from the vehicles to the infrastructure, as can be seen in figure 2.1, but when used with I2V, it becomes more powerful, because it not only allows the infrastructure to receive data from

the vehicles, it also allows it to respond. Very generally, this road infrastructure could be used to process some logic on the received data and help the driver make more educated decisions.

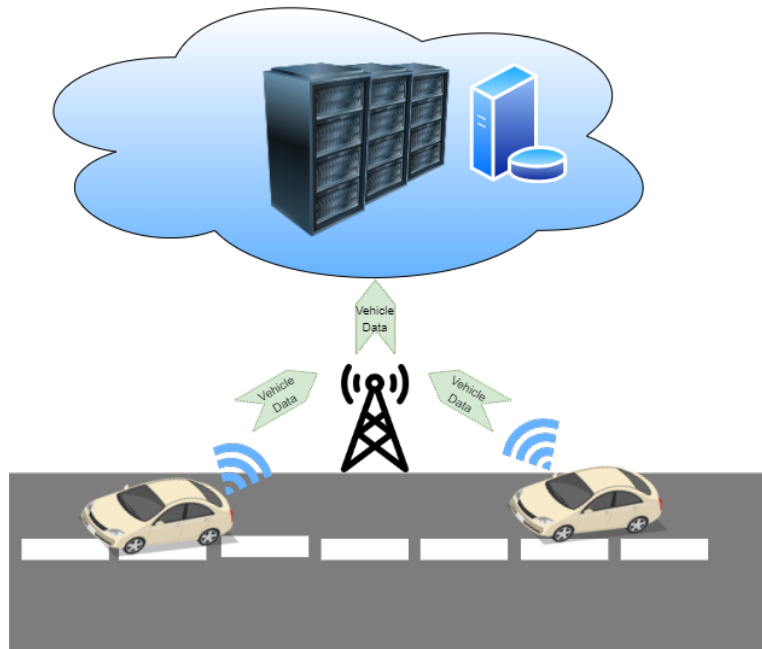


Figure 2.1: Vehicular communication - Vehicle data is sent to the infrastructure

2.1.2 I2V Communication

Infrastructure-to-vehicle communication, or I2V, consists in flow of data from the infrastructure to the vehicles, as seen in figure 2.2. I2V communication has a very large amount of real world applications.

These applications are seen as one of the key challenges to ensure safe and efficient driving through the glowingly overloaded road infrastructure. This technology enables the creation of new use cases, now that the infrastructure is able to send messages to the vehicles. It could be used to provide information to the vehicle drivers, such as the speed limit of the current road, notify about a possible accident ahead, road works, and many other cases. It could also be used in more complex ways, like for example, for ensuring the synchronous communication between vehicles, i.e., it could be used for operations that require coordination between vehicles like truck platooning [5] or lane merging [6].

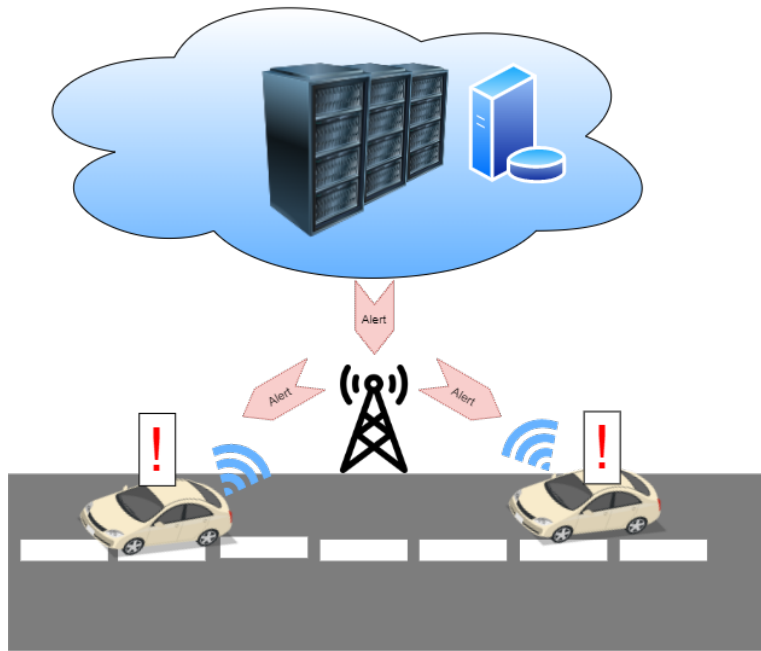


Figure 2.2: Vehicular communication - Infrastructure data is sent to the vehicles

2.1.3 V2V Communication

Vehicle-to-vehicle communication, or V2V, is another type of V2X technology which allows vehicles to send messages directly to each other without the need to use a third party to relay the messages, as seen in figure 2.3. This technology is different from the previous types of V2X technology that we presented, because of its decentralized nature, which can fully work without the infrastructure.

This opens up very interesting use cases like the detection of potential collisions with another vehicle that supports V2V communication. This is interesting because both vehicles may be out of sight of each other and may be completely unaware of a potential collision. V2V allows messages to be exchanged between the two, and makes it possible to the vehicles OBU to calculate if a collision may happen. Like with the I2V, V2V also makes it possible for vehicles to perform coordinated maneuvers, like the platooning or the lane merging.

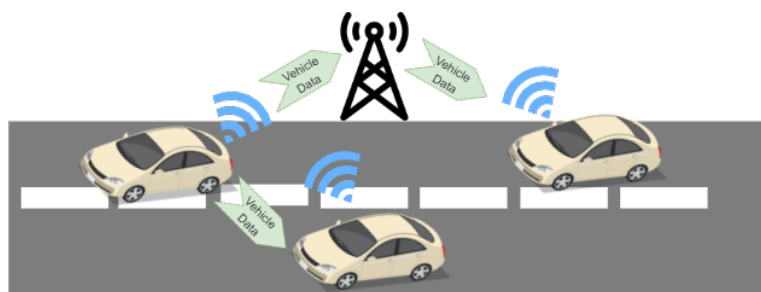


Figure 2.3: Vehicular communication - Vehicle data is sent to other vehicles

2.2 VEHICULAR-BASED MESSAGES

Vehicular messages can either be sent periodically or asynchronously. They essentially fall into two groups:

- **Periodic** - these messages are usually generated by the vehicles and contain its information, such as the current speed, heading, identifier or location. These messages are usually broadcasted to all the listening entities and may be received by both the infrastructure or by other vehicles. They are often generated as CAM.
- **Asynchronous** - these messages can be generated by both the infrastructure or by the vehicles, depending on the situation, and contain information about the origination station, such as the current speed and heading (if it is a vehicle), identifier or location. These messages are usually broadcasted to all the listening entities and may be received by both the infrastructure or by other vehicles. They are often generated as DENM. Asynchronous messages could effectively be used for the communication between 2 vehicles, between a vehicle and the infrastructure and between the infrastructure and a vehicle. In order for a DENM to be generated, there must be usually a trigger that starts the DENM generation process.

The two previously mentioned types of message were defined by ETSI and are meant to be used in the context of vehicular networks. These message types contain complex but complete schema, that cover a large set of datapoints that could be extracted from the vehicles.

2.2.1 Cooperative Awareness Messages (CAM)

Vehicle data is periodically broadcasted in the CAM format, which is a standard used across multiple OEMs. The mass adoption of this type of message mitigates the risk of incompatibility of the solutions between two different OEM, i.e., if cars that contain a Bosch OBU transmit data as CAMs, any vehicle that supports ITS technology, independent of its manufacturer will be able to interpret that message.

Figure 2.4 shows the structure of a CAM as defined by ETSI [7]. The CAM is a very complex entity, so we will highlight the most important attributes of this message. Note that the CAMs do not support the inclusion of floating point numbers, so all the attribute values must be encoded and passed as integers. These attribute values will then be decoded and converted back to floating point once the message is received.

- **Station ID** - this is the station's unique identifier, i.e., each station (vehicle, roadside unit, etc.) has an integer that uniquely identifies it.
- **Station Type** - this indicates the station's type. There is a large amount of data types, but the most common are:
 - **1** - Pedestrian
 - **2** - Cyclist
 - **5** - Passenger car
 - **6** - Bus
- **Latitude and Longitude** - this is the vehicle's position represented in the form of the latitude and the longitude. Both of these values must be represented as integers.

- **Speed** - the vehicle's current speed, in m/s.

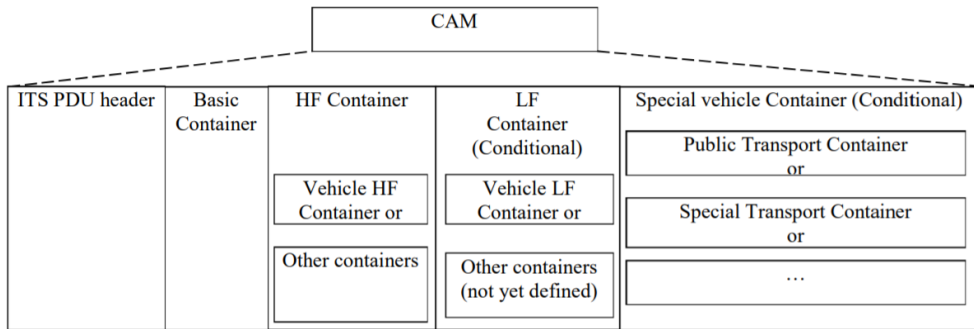


Figure 2.4: General structure of a CAM (from [7])

2.2.2 Decentralized Environmental Notification Message (DENM)

As we mentioned previously, some messages can be sent asynchronously, i.e., they are only sent when something triggers them. These messages are often sent in the DENM format, that is defined as the standard by ETSI¹. Like with CAMs, the mass adoption of this standard leads to the possibility of cross-platform compatibility, i.e., the vehicle from one manufacturer may send an alert to a vehicle of a different manufacturer. These messages are aimed to describe multiple types of events, not only crashes. DENMs are structured as shown in figure 2.6 and are divided into multiple containers. Since they have a very complex structure, we will just highlight the most important attributes:

- **Station ID** - this is the station's unique identifier, i.e., each station (vehicle, roadside unit, etc.) has an integer that uniquely identifies it.
- **Station Type** - this indicates the type of the station that generated this message.
- **Latitude and Longitude** - this corresponds to the coordinates of the epicenter of the DENM.
- **Cause Code** - the code that provides the cause/reason of why the event was generated.
- **Sub Cause Code** - similar to the cause code, but provides the sub type of the event.

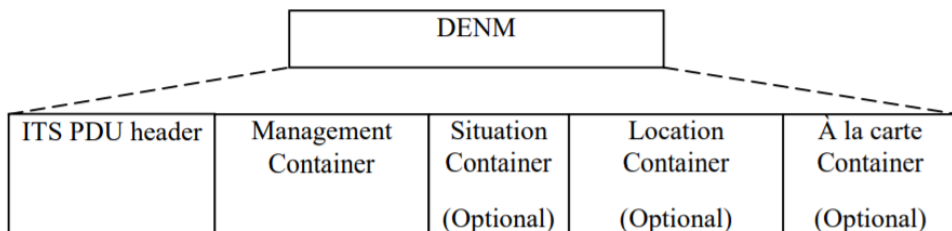


Figure 6: General structure of a DENM

Figure 2.5: General structure of a DENM (from [8])

¹https://www.etsi.org/deliver/etsi_en/302600_302699/30263703/01.02.01_30/en_30263703v010201v.pdf

2.2.3 Collective Perception Messages (CPM)

Besides CAMs and DENMs, there are many other types of messages that are exchanged between vehicles. One of these types are the Collective Perception Messages (CPM). As the name suggests, collective perception consists in the concept of sharing the perceived environment of a station based on perception sensors [9]. Contrary to Cooperative Awareness, where a station broadcasts information about its current driving environment, the collective perception offers the possibility to share information about objects in the surroundings, which have been detected by different sensors, cameras or other information sources. This type of information is specially beneficial in some use cases:

- **Detection of Non-Connected Road Users** - Road users that do not communicate with the system, e.g., drivers of vehicles that do not support V2X technology, pedestrians or cyclists, can only be detected by the environment perception sensors of stations that support this type of technology. With the collective perception service, the number of road users recognized can be increased significantly.
- **Detection of Safety-Critical Objects** - Some objects may pose a significant risk to the safety of road users and it is desirable to avoid them. These objects can include, for example, lost cargo, a tree limb or debris located on the road which may lead to a puncture in vehicles or even an accident.

It may also make sense to merge sensor data with the information received from CAMs before generating and sending out a CPM. This allows stations to have increased awareness about the surrounding environment, which may be specially useful in intersections.

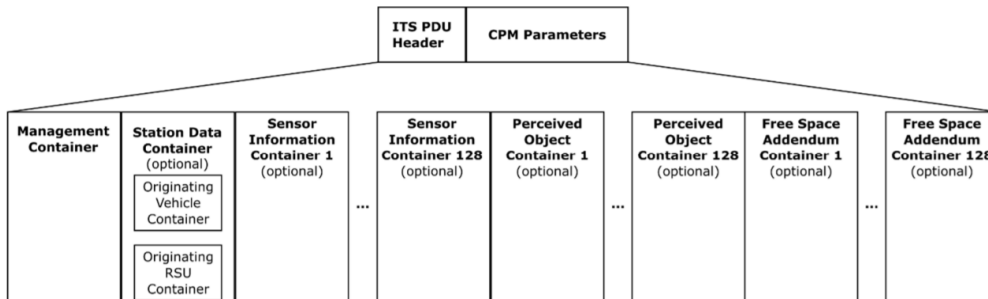


Figure 2.6: General structure of a CPM (from [9])

2.3 RELATED WORK

The rapid increase in the number of vehicles on the roads, as well as the growing urban populations have worsened existing problems such as traffic congestion and accidents. These problems along with other factors, such as the increasing adoption of embedded telematics in vehicles by OEMs, have motivated researchers to develop new smart city concepts and technologies aiming at providing new and useful solutions for these problems.

Several systems have been proposed to tackle traffic congestion in big cities by making use of smart city capabilities. The work in [10] proposes a framework to reduce the latency of emergency services for vehicles such as ambulances and police cars, while trying to minimize

the disruption to the regular traffic. This framework also supports some security mechanisms to prevent these functionalities from being exploited by malicious players.

Collision avoidance is also another major problem being addressed by researchers. The work in [11] aims to address the specific case of a collision between a vehicle and a Vulnerable Road User (VRU) by using the user's smartphone as the main communication device between the user and the infrastructure. Messages containing the position, the heading, along with many other types of data, are sent to remote servers through edge nodes. Once this information reaches the collision detectors, i.e., the remote servers, a collision algorithm determines if any two users are in a collision course, and an alert will be generated and sent.

Tesla [12] has developed its autopilot functionality that allows the vehicle to drive autonomously with minor user interaction. The autopilot contains features like automatic lane changing that can handle lane merges, exits and overtaking. It also contains an auto-parking system, traffic lights and stop-sign recognition, and a feature that makes the car navigate out of the parking space and straight to the driver.

The work in [13] proposes the Cooperative Intersection Collision Avoidance Persistent (CICAP), a collision avoidance system that aims to prevent collisions between vehicles on intersections. This system assumes that two vehicles, equipped with automatic actuators, are able to communicate with each other using V2V technology. The CICAP runs on each vehicle. Based on information such as the position, the speed and the direction of all neighboring vehicles, the CICAP calculates if both vehicles are on a collision course. If a collision course is detected, then the vehicle with the lowest priority will reduce its speed and only increase it again once the vehicle with the highest priority has passed. This may be specially useful in scenarios that involve emergency vehicles. This system was not tested in a real environment, which makes it hard to verify its effective reliability and scalability.

The work in [14] addresses the scenario of vehicle/vehicle collision, more specifically to prevent rear end collisions, when visibility is limited because a vehicle is present in between two other vehicles. Assume a chain of vehicles, A, B and C, where vehicle C is at the back, vehicle B is in the middle and vehicle A is at the front of the queue. If A breaks, then the proximity sensors of B will detect that breaking action and quickly react by breaking, to prevent a crash. By similar analysis, when vehicle B breaks, then the proximity sensors of vehicle C will detect this and break, but [14] proposes a way that allows vehicle C to be notified of the breaking action of vehicle A faster. This is done by using V2V technology and allowing vehicle A (at the front of the queue) to exchange messages with vehicle C (at the back of the queue), allowing vehicle C to have more time to react to the breaking action of vehicle A.

Several projects have focused on the infrastructure, data collection and aggregation on smart cities. The work in [15] describes the implementation of CiDAP and its deployment in a large scale smart city testbed in the city of Santander, Spain. This testbed involved more than 15,000 sensors attached to 1,200 sensor nodes that have been installed around an area of approximately 13.4 square miles in the city. These nodes are hidden inside white boxes and are attached to the street infrastructure. The communication among sensor nodes,

repeaters and gateways is carried out through IEEE 802.15.4 interface, while gateways use Wi-Fi, GPRS/UMTS or Ethernet interfaces to connect with the SmartSantander backbone. CiDAP, a live smart city big data platform, focuses on providing both historic data and real-time data.

The MLK Smart Corridor [16] is the example of another smart city testbed deployed in Chattanooga, TN, USA, which aims at gathering data from heterogeneous sources, aggregating it in a central system and then making it available to the public, so that other researchers can use it. This work also adopts the concept of edge computing, by distributing multiple edge nodes throughout the city. These edge nodes are responsible for structuring the data and sending it off to the central system. This data can be consumed, by high level applications, in real-time, but it is also stored in a data lake, where it can be used for later analysis.

Much like the works mentioned previously, [17], is a live testbed deployed in the city of Uppsala, Sweden, which also aims at providing open data to all users. This testbed, including the open data and open APIs, allow third parties to develop and experiment new sensing products and services that could be exported to the international market. This testbed uses the concept of sensor gateways which are devices that receive the data from the sensor network and send it to the cloud services. The data is then stored centrally where it can be consumed later by other applications and services.

The work in [18] provides a smart city architecture that focuses on the data ingestion and aggregation aspects, which collects data from citizens, commuters, social media crawling, IoT sensors, city operators. It can provide detailed information about the city and its public services (available parking lots in any specific area, traffic flows and collapsing area, people flows, triage usage of hospitals, incidents in the streets), allows users (technical and non-technical) to detect early warnings about specific occurrences through the analysis of historic data, and also provides use cases for entertainment, such as booking tickets. The smart city API provides data to mobile, web and third party city operators.

The work in [19] proposes the Core Telematics Platform (CTP), a high-scalable platform for Intelligent Transport Systems that can optimize the data transmission under linearly increased time complexity by employing a high-scalable architecture design when a large number of devices connect to the application servers. The authors also performed simulated and experimental tests to verify the effectiveness and efficiency of this high-scalable CTP design. All the data communications and probe information are encrypted using a 128-bit advanced encryption standard (AES) algorithm, for security concern.

The work in [20] proposes IoTDA, an integrated IoT data service system, that aims to overcome the heterogeneity of various IoT data to enable them to be integratable and more useful. IoTDA collects different types of IoT data independently and then merges them according to the time when the data was acquired. The authors use an example of a car that is equipped with a camera and takes pictures of the road, so that the AI-based service can label any peculiar objects on that image. At the same time, other times of data are also being collected, such as pollution data and the position of the car. After the data is collected, it's aggregated into a single table.

All these works represent platforms developed in the cities. A full vehicular and sensing platform is much less common. The next section describes the Aveiro Tech City Living Lab.

2.4 AVEIRO TECH CITY LIVING LAB

The technological lab, Aveiro Tech City Living Lab, is an advanced large scale infrastructure, spanned all over the city of Aveiro, at the service of researchers, digital industries, start-ups, scaleups, R&D centres, entrepreneurs and other stakeholders interested in developing, testing or demonstrating concepts, products or services. This infrastructure integrates people, through their mobile phones, sensors and vehicles, that support ITS technology, such as automobiles, bicycles in the city and "moliceiros" in the Aveiro Lagoon, aerial and aquatic drones.

This infrastructure acquires data from vehicles and sensors, through roadside units, which are special equipments that are usually placed near a road. After the data is received and processed by the RSUs, it is then aggregated in an MQTT instance, which is a publish-subscribe system.

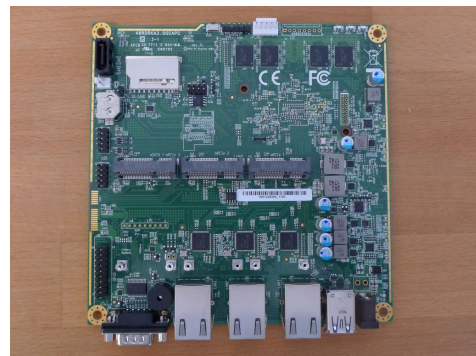
2.4.1 On Board Units

In order to send their data, the vehicles must have a module that collects the data from the vehicle sensors, encodes them into a CAN and sends them to the RSUs of the Aveiro Living Lab. For this task, special boards, called On Board Units (OBUs) must be installed in the vehicle for the data to be sent. Note that the concept of an OBU is too generic, and these boards can be manufactured and configured by different OEMs. The OBUs are computers that can be placed on any type of vehicle (cars, buses, boats, etc.) and will communicate with the roadside units when they are in range of the receptors.

These boards contain modules that allow the OBU to communicate with the roadside units via WAVE but also contain Wi-Fi modules that are used for multiple purposes. Wi-Fi communication is used to allow the OBU to exchange information with other embedded system that are located on the vehicle and to provide communication to users inside the vehicle. Some OBUs can also communicate with the infrastructure directly, via LTE and 5G, without communicating with a Roadside Unit, but this may not be available in every board.



(a) An OBU placed inside of a car



(b) An OBU without the case protection

Figure 2.7: On Board Unit

2.4.2 Roadside Units

Unlike the OBUs, the RSUs are small computers that can be placed in strategic locations throughout the city and are connected by fiber to the datacenter of Instituto de Telecomunicações. RSUs are responsible for collecting data from the OBUs and sending it to the infrastructure, but in order to accomplish this, both boards must be in range of each other to communicate via WAVE. In order to maximize the range of RSUs, we have decided to place these equipments as close to the road as possible. Since we also wanted to make use of external sensors, such as radars and cameras, we decided to create two types of RSUs: Murals and Smart Lamp Posts.

Much like the OBUs, the RSUs contain modules that allow the RSU to communicate with the on board units via WAVE but also contain Wi-Fi modules that can be used for other purposes.

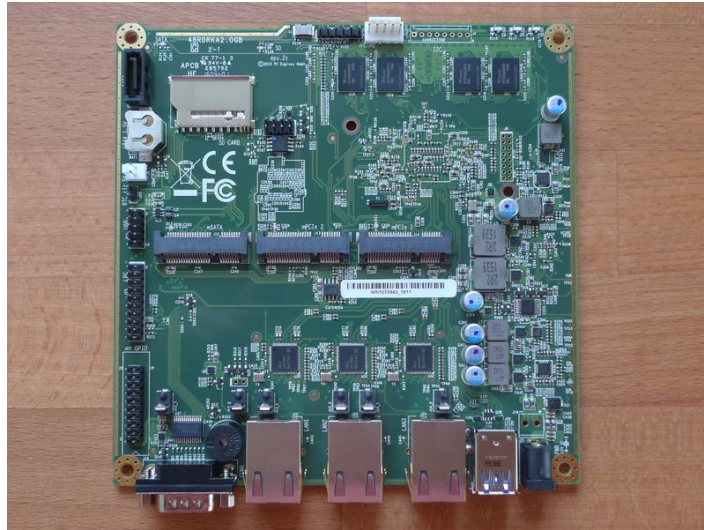


Figure 2.8: The board of the RSU without the case protection

Wall Boxes

Wall Boxes are a type of RSU that is placed in a building (or structure), most commonly, on the roof, and is used to extend the coverage of the network. This equipment offers a more limited set of functionalities than the Smart Lamp Posts, since it does not support the integration with other sensors, such as radars. In order to collect data from these sensors, another type of RSU was created, the Smart Lamp Post.

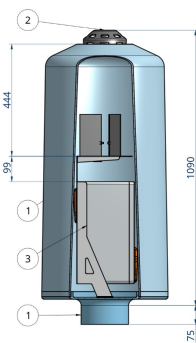


Figure 2.9: Wall Boxes. Design and deployment in the city

Smart Lamp Posts

Smart Lamp Posts, a type of RSU, are large lamp posts that contain a set of sensors on the tip of the post and are, most commonly, placed on the side of the road. Figure 2.10a contains a diagram with the design of this type of RSU and a picture of a smart lamp post deployed near Instituto de Telecomunicações in Aveiro.

Figure 2.10b shows a smart lamp post that collects information from a radar, camera and from WAVE communication. This data is displayed, in real-time, by the City Manager (CM) web app which will be presented later in this document.



(a) Smart Lamp Post. Design and deployment in the city



(b) Smart Lamp Post location - Ponte Dubadoura, Aveiro

Figure 2.10: Smart Lamp Post

2.4.3 Infrastructure

There are currently 13 active wall boxes and 8 active smart lamp posts, in the city of Aveiro, although this testbed is being expanded up to a total of 44 RSUs. Figure 2.11 shows the location of all active (highlighted markers) and inactive (transparent markers) roadside units. Wall boxes are represented by a green marker (with the icon of a house) and smart lamp posts are represented by a purple marker.

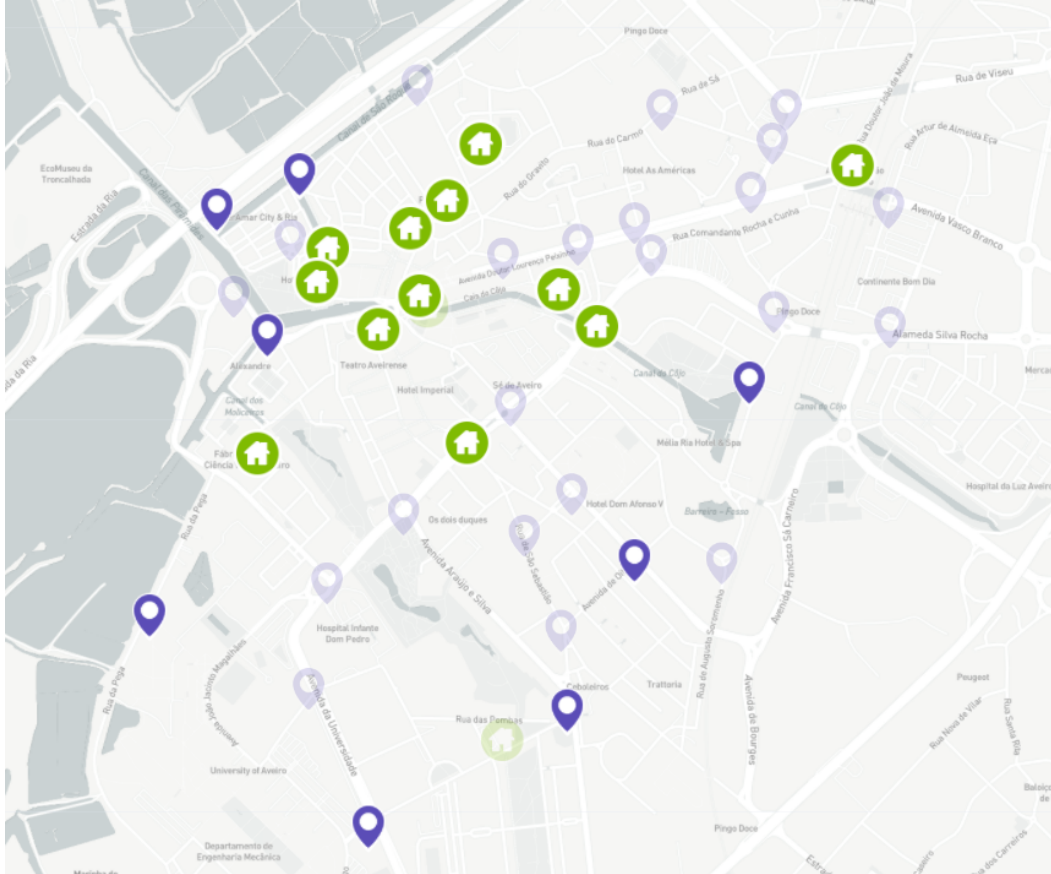


Figure 2.11: Roadside Units in Aveiro

2.5 SUMMARY

In this chapter, we started by providing the context and the motivation for this dissertation.

In section 2.1 we presented the different types of V2X communication along with an explanation about each one. We also provided examples of potential use cases that could be developed with each type of communication.

In section 2.2, we presented the ITS message standards and analysed the use cases and the most important attributes of each message. We also made the distinction between periodic and asynchronous messages, and explained in what scenarios each one of these is used.

In section 2.3, we presented the related work in the literature and discussed different approaches to the concept of data aggregation and centralization. Finally, in section 2.4 we presented the concept of the OBUs and the RSUs in the Aveiro Tech City Living Lab. We also explained the different types of RSU that were used by the IT. After that, we performed an analysis about the infrastructure in Aveiro.

Architecture

Historic information can be used in a plethora of applications and use cases and is commonly used for data analytics. Storing the historic data of a whole city could provide a better insight into the behavior of its citizens, for example.

It would also be useful to have access to real-time data which could be used to complement the historic data. This type of data could be used instead for a more dynamic monitoring of the city.

In this chapter, we propose the architecture for a platform that gathers real-time and non real-time data from vehicles and displays it in a dashboard. We will list and explain the main goal of each service that makes up the infrastructure, using a bottom-up approach, which starts in the modules that are closer to the vehicles, and then follow the data flow until it finally reaches the City Manager App (CM). Some modules are very complex and an in-depth analysis will be provided in the following chapters.

3.1 REQUIREMENTS

Before proposing an architecture, we need to create a list of well-defined requirements that we will use to guide the development of this platform. These requirements are directed towards the creation of a base infrastructure that can be used for multiple use cases. This must be a more generic system to allow to later expand and implement specific features, such as, for example, displaying the vehicular data in real-time in a dashboard or by persisting this data to later perform some computation on it.

3.1.1 Vehicular data acquisition

One of the most important parts of the proposed system is acquiring data from the vehicles. This data must be retrieved from each vehicle's system and then aggregated somewhere. For this to work, these vehicles must contain some computer or board that collects the vehicle telemetry and possesses a wireless interface that allow it to send data to the platform, either to the cloud or through a proxy.

3.1.2 Sensor integration

As mentioned previously, this platform should be made as generic as possible. Another one of the main features of this system is the possibility of retrieving and aggregating information from different sensors, and not only from the vehicle's internal system, such as radar and video cameras. We could go further and later merge all of this information to get more accurate data, e.g., for a particular vehicle we could collect its data from a radar and from a video camera. After merging the information from these two sensors we could calculate the coordinates of the vehicle with more accuracy. This concrete example is out of the scope of this dissertation, but we believe that it is important to demonstrate the versatility of the system that we are proposing.

3.1.3 Access to real-time and historic data

The previous requirements focused on describing the platform's behaviour regarding the acquisition of data, but we must also make sure that the system is able to provide the client with two main interfaces: an interface that allows access to real-time data, and an interface that allows access to the historic data. There is a countless number of scenarios where it is useful not only to have access to persisted historic data but also to real-time data. This could be useful for an app that allows users to see the position of the vehicles, in a map, in real-time. Persisting this data also creates a lot of new and interesting scenarios. One of these scenarios could be, for example, allowing users to replay the state of a city on demand. By integrating these two interfaces, the system becomes more versatile, since it can serve as the support to a higher number of use cases.

3.1.4 I2V communication

With the previous requirements it is possible to receive data from the vehicles and consume it, but it is also important for the system to allow the inverse flow of data, I2V communication, which consists in ensuring that the infrastructure is able to send messages to the vehicles. This feature makes the proposed system even more versatile, because it allows information to be sent to the vehicles themselves. This would allow, for example, the development of a service that notifies OBU of dangerous driving conditions. We could go further and create a GUI that allows the user to generate and send alerts to the OBUs.

3.2 PROPOSED ARCHITECTURE

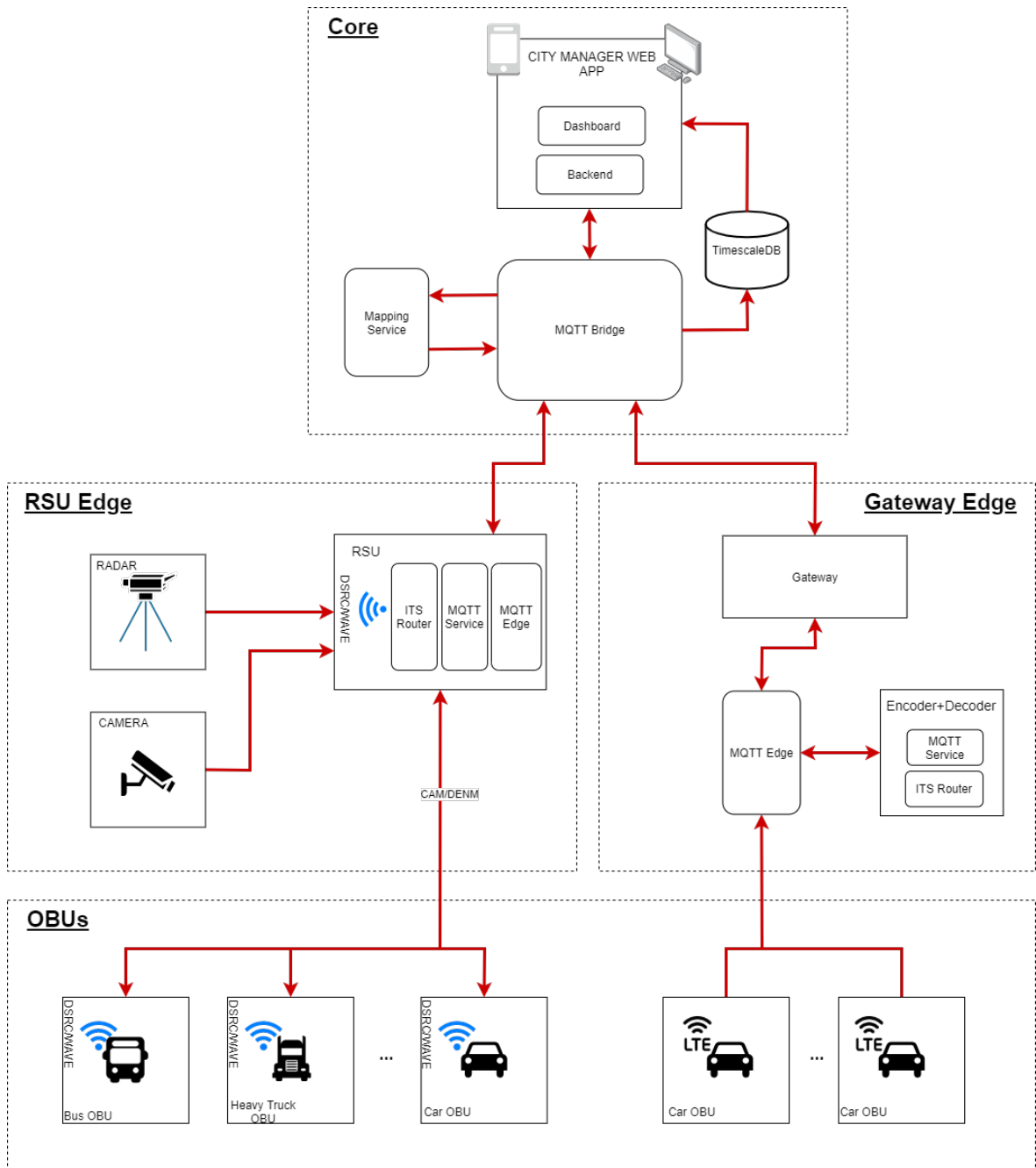


Figure 3.1: Architecture - Overview

To satisfy the requirements, we designed the architecture, seen in figure 3.1. This architecture consists of three major components: the **on board units**, which produce and send the vehicle data; the **edges**, which serve as proxies between the data sources and the core but can also perform some computation on that data; the **core**, which consumes the data collected from vehicles and sensors. There are also two types of edges: **RSU edges**, which can be deployed near a road, allowing it to acquire data through direct communication with the

vehicles; and **gateway edges** which can be located in the cloud. The architecture focuses on allowing the flow of data from the vehicles to the high level applications like the City Manager, and also allows messages to be sent from the CM to the OBUs. It is important to mention that, in this diagram, we present the City Manager Web App as a possible consumer of the data, but this system also allows other services to consume that same data by subscribing to topics in the MQTT bridge or querying the TimescaleDB database.

3.2.1 On board unit

As mentioned previously, the OBUs are responsible for sending the vehicle data to the infrastructure. This can be done by sending the data directly to the cloud via LTE or to the RSUs that are distributed throughout the city. Figure 3.2 shows the internal modules and their interaction. These main modules are:

- **Vehicle Service** - Has access to the vehicle data and generates ITS messages with that information. Generates CAMs periodically and generates DENMs when certain events occur.
- **ITS Router** - Serves as a message router between different services. Responsible for decoding incoming ITS messages and encoding outgoing ITS messages. Due to the versatility of this module, we use it in other components of the architecture as well.
- **MQTT Service** - Serves as the data handler between the ITS Router and the MQTT. Receives decoded messages from the ITS Router and publishes them in the MQTT, in the JSON format. Listens for JSON messages in the MQTT and sends them to the ITS Router. Due to the versatility of this module, we reuse it in other components of the architecture, much like the ITS Router.
- **Eth2Udp** - Receives packets on the wireless interface and sends the underlying payload to a udp port.
- **Udp2Eth** - Receives udp packets and sends the the packet through the specified network interface.
- **Gateway Service** - This service is responsible for listening for messages in the internal MQTT and sending them via LTE, when available, to a gateway edge. This type of communication serves as an alternative communication mechanism, other than simply sending ITS messages through a roadside unit. It can be turned off to prevent unnecessary data from being sent.
- **Notification App** - A GUI that allows users to be notified when the OBU receives an alert, for example. Can be accessed through the browser, on a desktop and on mobile. This app will listen for alerts that are published in the internal MQTT
- **MQTT OBU** - The MQTT that runs in the OBU. This module serves as a data broker between the data producers, MQTT Service, and the data consumers, Gateway Service and Notification App

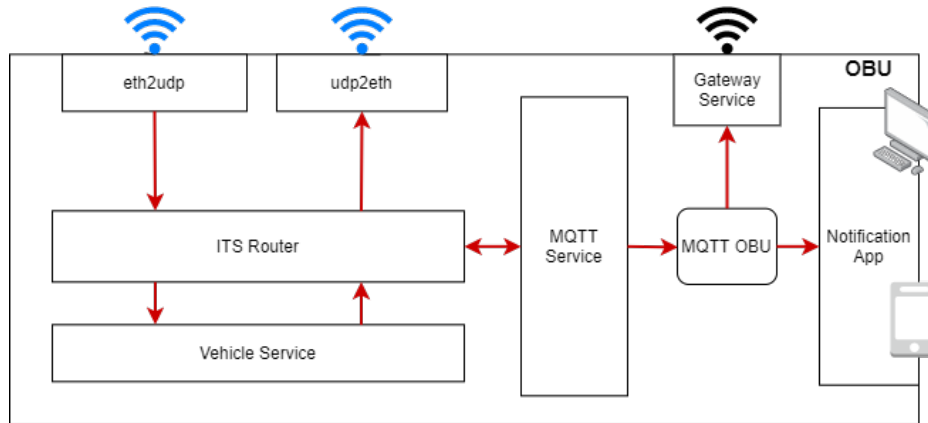


Figure 3.2: Architecture - On board unit

3.2.2 Edges

The edges can receive information from multiple data sources, such as vehicles, radars and cameras. Because of the diversity of data, we designed a type of RSU that was able to perform some computations, such as the normalization of that data, for example, before allowing it to be consumed by the high level services in the core. These edges, called **RSU edges**, can communicate via DSRC with nearby vehicles that support ITS technology. Even though these edge computation capabilities are useful, a vehicle is not going to have connectivity to RSU edges constantly and may need to communicate with the core even if it does not have connectivity. To address this limitation, we have developed the **gateway edge** which receives vehicle data sent, via LTE, to its MQTT, then decodes it and sends it to the core, where it will be consumed by the high level services. To sum up, there are two types of edge:

- **RSU edge** - Receives ITS messages containing the vehicle data, via DSRC, decodes the messages and sends it to the core. This edge can also receive messages from the high level services, encode them and send them to the OBUs. For the messages to be received or sent, the vehicle must be in range of the DSRC communication. To maximize the connection time between the OBU and the edge, the edge can be placed near a road. One of the biggest advantages of these types of edges is that they allow an easier integration with external sensors, like radars and cameras, which require the hardware to be physically pointing towards the road. Because of these constraints, we decided to create SLP, which are posts that contain not only the hardware where the RSU software will run, but also contain the external sensors which will send data to the RSU.
- **Gateway RSU** - Receives ITS messages containing the vehicle data, via LTE, decodes the messages and sends it to the core. This edge does not allow messages to be sent via LTE to the vehicles, but this could be added in the future as an improvement to the system. For the messages to be sent, the vehicle must allow LTE communication. Unlike the RSU, this edge does not need to be physically located near a road. It can instead be deployed to the cloud.

Figure 3.3 shows the internal modules of both types of edges and their interactions. Notice that some modules are used in both edges and in the OBUs. These services, the Vehicle

Service, ITS Router and MQTT Service, are used to generate, encode and decode ITS messages. Because of the importance of these modules, we have developed an extensive test suite for them that extends the tests that were already developed by the authors of the original version of the ITS Router. This test suite contains unit, integration and end-to-end tests but also contains performance tests. These tests allow to test each module separately. Most of these modules have already been explained previously, so we will provide an explanation for the modules that are exclusively used in the edges.

- **MQTT edge** - Each edge contains a MQTT edge, which aggregates the data received by that edge, i.e., the data received from the sensors that are connected to it and data from nearby vehicles. The data that is published in this module will be fetched by the MQTT bridge and made available to the high level applications.
- **Gateway** - Relays all messages published in the MQTT edge to the MQTT bridge in the core. We decided to chose this approach instead of configuring the MQTT Bridge to fetch all messages from the MQTT edge automatically, like in the RSU edges, because we wanted to be able to filter confidential data and prevent it from being sent to the MQTT Bridge.

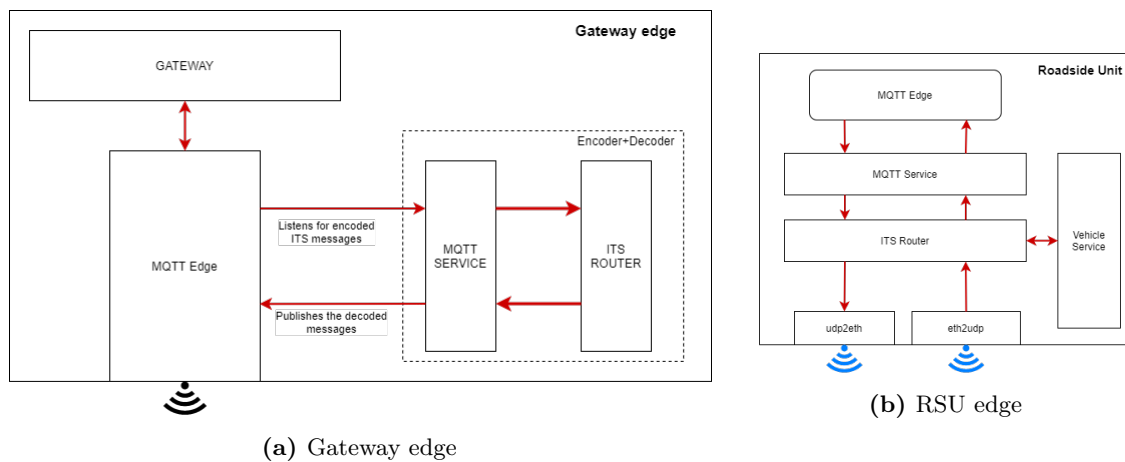


Figure 3.3: Architecture - Edges

These edges can receive data directly from the vehicles via DSRC when the OBUs are in range, or via LTE, as we have explained previously. For this reason, RSU edges should be placed in a location that maximizes the connection time, to allow the OBUs to send as much data as possible to the infrastructure, and that minimizes the number of obstacles between the OBU and the RSU, which would allow a higher number of OBUs to have connection with the RSU.

RSUs can also receive data from external sensors, such as radars and cameras, for example. The data from these sensors is published to the MQTT edge, where it will be collected by the MQTT bridge and made available for the CM.

3.2.3 Core

We have explained how the data is generated by the OBUs and collected by the edges, but it is also important to explain how that data is aggregated and how it can be used. For this process, we designed **the core**, which is located in the cloud, and is responsible, among other things, for making both real-time and historic data available to high level applications and services. Figure 3.4 shows the inner modules of the core.

- **MQTT Bridge** - The MQTT bridge allows to connect multiple MQTT edges together, i.e., it fetches information from all the existing MQTT edges and makes this data available in one place. The high level apps like the CM only need to subscribe to the topics of the bridge, instead of individually subscribing to the topics of each MQTT edge.
- **TimescaleDB** - A database that contains the historic data from CVs, sensors and VRUs. When the data is published in the bridge, a process fetches it and adds it to the database.
- **Mapping Service** - This module routes the DENMs that are sent from the infrastructure to the vehicles, called target DENMs. It contains the logic that will determine, the best RSUs to receive the target DENM. Since this is a complex module, we will provide a more detailed explanation later on.
- **City Manager Backend** - The backend contains a RESTful API and a websocket server. When the data is published in the bridge, it is immediately sent to the dashboard via websocket. The API allows the client side to fetch the historic data, as well as static information, such as the list of RSUs and their respective information (name,geographic location,etc.).
- **Dashboard** - The dashboard is a GUI that allows users to visualize the real-time data as well as the historic information. A more detailed explanation will be provided later on.

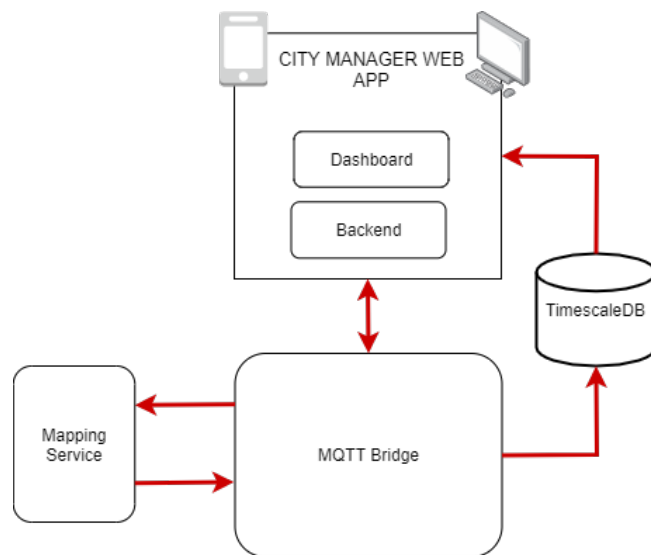


Figure 3.4: Architecture - Core

3.3 MODULE COMMUNICATION

In section 3.2, we presented three major components of the architecture - the core, the edges and the OBUs - and their inner modules. In this section we will analyse the message types, message formats and later we will list and explain the different data flows.

3.3.1 Message types

Different types of messages are used to exchange data between the OBUs, the edges and the core: vehicle data, sensor data and event notification messages.

- **Vehicle data** - These messages contain information about the current state of the vehicle, such as its id, type, speed, heading, along with many more attributes. This information is usually represented as either a CAM or a JSON, depending on the service that is processing it.
- **Sensor data** - The information contained on these messages can vary depending on which sensor is producing it. Radars periodically publish message containing information about the entities that they detect, such as their position, speed or heading. Much like radars, cameras also periodically publish messages containing information about the object that they detect, but these messages contain a more limited array of attributes. They do not contain the speed or heading of the object, but instead contain information about the type, or label, the confidence level and an estimated geographic location of the objects. This information is published in the MQTT edge.
- **Event notification messages** - These messages are used to notify drivers of potential dangers on the road. They are often represented as DENMs, but can also be represented in the JSON format. These messages can be generated by the CM and sent to the vehicles, through the RSU edges, or directly generated in the RSU edges. Sending DENMs to vehicles via LTE is out of the scope of this dissertation, but could be implemented in the future, which would allow drivers from vehicles that do not have direct connection with the RSUs to be notified as well.

Regardless of their type, all messages end up being published to the Message Queuing Telemetry Transport (MQTT) bridge, and will be used by the core services. It is important to include some mechanisms that allow these services to uniquely identify the type of data and its source. To address this we decided to include metadata in the messages and include the source of the data in the topic. In sum, the data must respect these properties:

- Each topic must only contain one and only one type of data.
- It must be possible to identify the source of the data.
- It must be possible to identify the exact time when the messages arrived at the source.

Now, that we have listed the main requirements that the data must meet, we will explain how the proposed solution addresses them.

One important requirement that was taken into account when developing this architecture is that each MQTT topic must not contain different types of data, e.g., topic p3/apu/cam cannot contain messages received from a radar and messages received from a vehicle. We use clearly defined names for each topic according to the type of data in it. This is done to ensure

that each topic contains only one type of information, and it makes it easier for high level application to access that data.

It is also important to identify from which source (edge) the information was received from. This is done by using the topic prefix notation, i.e., messages that are received from the edge P3 - Ponte Dobadoura must have the prefix p3/. For example, in the MQTT bridge, topic p3/apu/cam could be used to store vehicle data received in the edge P3.

Finally, it is important to be able to include the timestamp when the messages arrive on the edge (notice that this timestamp is different from the timestamp generated by the vehicle or sensor). When the vehicle data arrives in the RSU, the E2U appends the timestamp to the packet. This timestamp is later interpreted by the ITS router and included in the message itself, where it can be used by other services.

3.3.2 Message formats

There are different types of messages, but these messages can have multiple formats depending on which services process them at any given moment, as we have previously seen. We will list and explain each format that the messages can have. This will allow to easily explain the data flows later on. These main message formats are:

- **Simple ITS Message** - this type of message is composed of the ItsPduHeader and the payload of the message. It can be divided into two sub types. This messages doesn't contain any additional headers.
 - **Simple CAM** - Simple ITS messages that contain the ItsPduHeader and the payload of a Cooperative Awareness Messages.
 - **Simple DENM** - Simple ITS messages that contain the ItsPduHeader and the payload of a Decentralized Environmental Notification Messages
- **Proper ITS message** - ITS messages that contain contain the BTP header¹ and the Geonetworking header². It can also be divided into two subtypes.
 - **Proper CAM** - Proper ITS messages that contain a simple CAM.
 - **Proper DENM** - Proper ITS messages that contain a simple acsdenm.
- **Full ITS Message** - Proper ITS messages that are ready to be sent from the wireless interface. It can also be divided into two subtypes.
 - **Full CAM** - Full ITS Message that contains a proper CAM.
 - **Full DENM** - Full ITS Message that contains a proper DENM.
- **Json ITS Message** - Simple ITS Message in the JSON format. It can also be divided into two subtypes.
 - **Json CAM** - Simple CAM in the JSON format.
 - **Json DENM** - Simple DENM in the JSON format.

¹https://www.etsi.org/deliver/etsi_ts/102600_102699/1026360501/01.01.01_60/ts_1026360501v010101p.pdf

²https://www.etsi.org/deliver/etsi_en/302600_302699/3026360401/01.02.01_30/en_3026360401v010201v.pdf

Note that some message types may include other message types. For example, a proper CAM contains a simple CAM. Another important aspect to point out is that ITS messages, like CAMs and DENMs, can easily be interpreted and displayed in packet-analyser tools such as Wireshark, since its format is well known and well defined. Figure 3.5 contains a wireshark capture of a full CAM. The fields of the simple CAM are represented in green, the attributes of the proper CAM are represented in blue and finally the attributes of the full CAM are represented in red. Note that the full CAM is the most extensive subtype and includes the attributes of all other subtypes of CAM.

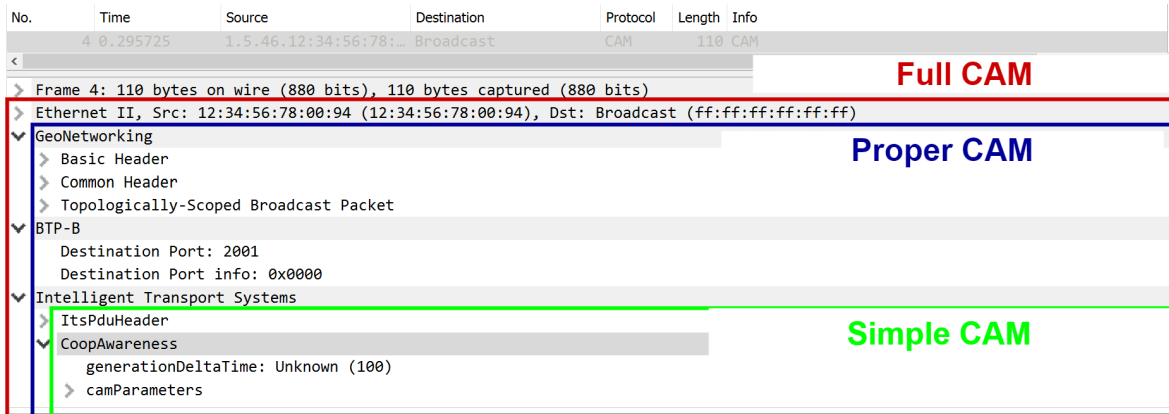


Figure 3.5: Wireshark capture of a CAM

3.3.3 Sending CAMs and DENMs through an RSU edge

To send the vehicle data through a roadside unit, the messages must be generated, encoded and then sent via DSRC. The OBUs can generate two types of messages: CAMs, which are generated and broadcasted periodically and contain the vehicle's telemetry, and DENMs, which are only generated when certain events occur, such as an accident, for example.

Figure 3.6 shows the data flow from CAMs and DENMs. Both of these messages are generated by the Vehicle Service, which has direct access to the vehicle data, such as the GPS data, in the Simple ITS format. They are then sent to the ITS Router, which is responsible for upgrading them to Full ITS messages, encoding and routing these messages to the U2E, where they will be broadcasted, via DSRC.

Once these messages are received by the RSU, via DSRC, they are processed by the E2U, which captures the packets in the wireless interface and sends them to the ITS router, via UDP. The messages are then decoded, interpreted, converted into the JSON format and then sent to the MQTT service, which will publish them in the MQTT edge.

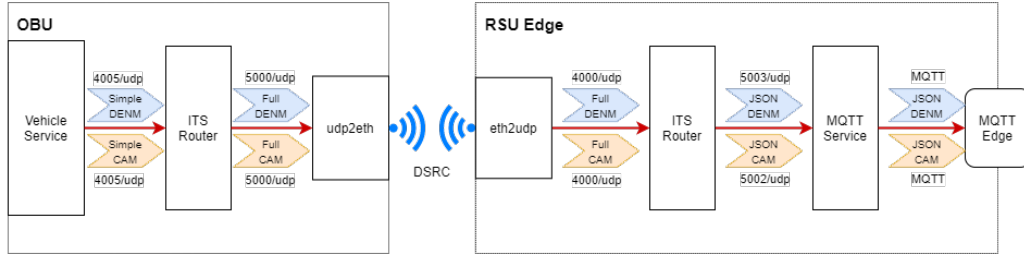


Figure 3.6: OBU/RSU Edge connection

3.3.4 Sending CAMs through a gateway edge

This data flow was created for OBUs that only publish CAMs in the proper CAM format. This was the case of the OBU used for testing at Bosch, whose inner modules are unknown to us. Note that we did not have control over these OBUs, and so we were unable to change the format in which CAMs were published, thus we needed to adapt our design to allow these messages to be decoded and interpreted. Figure 3.7 contains a diagram of this flow, where proper CAMs are generated and published directly to topic geral/cams of the MQTT edge, via LTE. These messages are then fetched by the MQTT service and sent to the ITS router to be decoded. they are then published back to another topic, boschBraga/cam_decoded, in the MQTT edge. After this, the gateway simply relays the messages published in a pre-selected list of topics, i.e., the gateway only relays the messages published in topic boschBraga/cam_decoded and not message published in topic geral/cams. The gateway allows to implement these types of mechanisms, which prevent sensitive data from being relayed to the MQTT bridge. For this dissertation, this data flow is only used with the OBUs from Bosch which are predefined to publish CAMs in the Proper CAM format. It is also important to mention that this data flow does not support DENMs, because the OBUs from Bosch did not generate these types of messages, but could be added in the future.

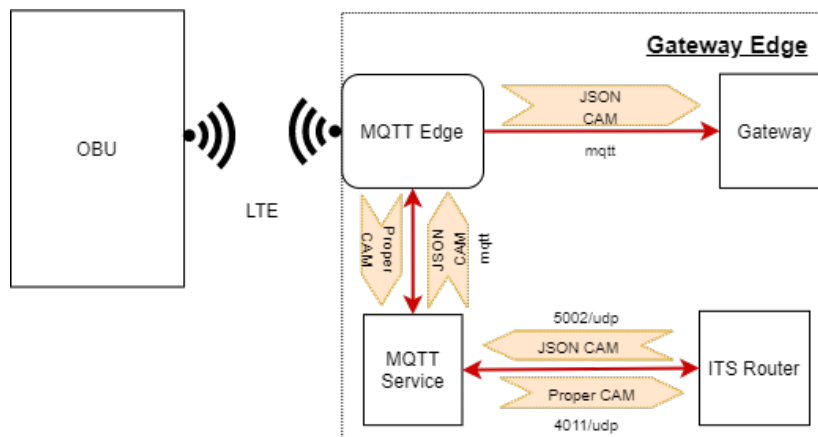


Figure 3.7: OBU/Gateway Edge connection

3.3.5 Sending CAMs from the OBU to the MQTT Bridge directly

Our goal is to make the core architecture flexible and capable of receiving information directly from the OBUs, so we adapted the architecture of the OBU to include the Gateway

Service. Note that the author of this dissertation did not develop the Gateway Service, and this service was added after the other inner modules of the OBU were implemented. Similar to the data flow 3.3.3, the CAMs are generated in the Vehicle Service and sent to the ITS router, where they are decoded and interpreted. After this, they are published in the MQTT, fetched by the Gateway Service and published directly to the MQTT bridge via LTE.

This approach is aimed to be an alternative to sending CAMs via DSRC, because it allows OBUs to transmit data even if they are not in range of a RSU.

Another important aspect to note is that no security protocol is being used to ensure the validity and authenticity of the data published in the MQTT bridge, since this is out of the scope of this dissertation, but this could be a potential improvement in the proposed system.

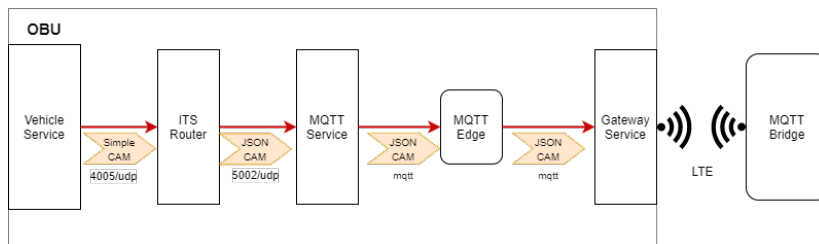


Figure 3.8: OBU/MQTT Bridge connection

3.3.6 Sending CAMs and DENMs to the core

We have previously seen that data, whatever its source, is ultimately published to the MQTT bridge in the JSON format, which allows the high level services to consume this data from one place. Note that this data flow alone does not describe the entire flow of data from the OBUs until they reach the core. Instead it should be seen as the last part of the process until the data is ready to be consumed.

In figure 3.9, it is possible to observe that, after the data is published to the MQTT bridge, it is persisted in the TimescaleDB database and is consumed by the CM backend which then sends it to the dashboard via websocket. This allows the CM to have access to both real-time and historic data.

It is also important to mention that the CM is used as one possible consumer, but the data in the MQTT bridge can be consumed by other services as well.

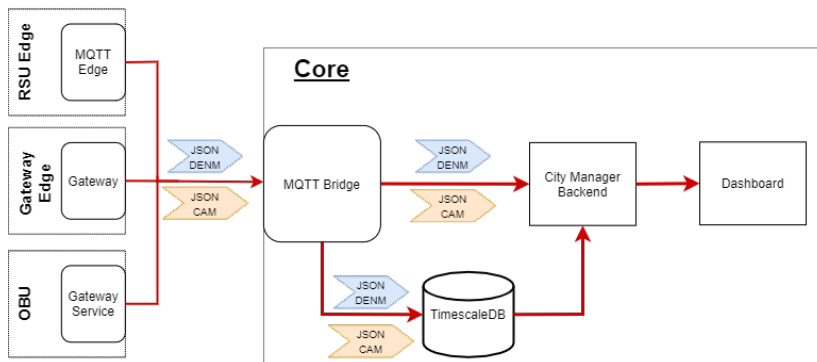


Figure 3.9: Core connections

3.3.7 Sending DENMs from the core to the RSUs

One of the most important requirements of the system is its support for I2V communication. Figure 3.10 contains a diagram of the proposed data flow. This architecture allows users to send alerts to the vehicles, through a web app, the CM, that generates a DENM and publishes it on topic target/denm, on the MQTT bridge. Once the DENM is published in the bridge, we must decide to which RSU(s) it will be sent. To solve this problem, we developed the Mapping Service which computes the list of target RSUs, i.e., the RSUs to where the incoming DENM will be sent, based on a set of metrics, such as the geographic location of the RSUs and the real-time connections between RSUs and OBUs. Since this module is very complex, we will provide a more in-depth explanation later on. Once the Mapping Service has chosen the target RSUs, it will publish the target DENM to the topic RID/target/denm, in the MQTT bridge, where RID is the ID of the RSU, e.g., for the target DENM to be published in the RSU P3 - Ponte Dobadoura, it must be published in the topic p3/target/denm of the MQTT bridge. After this, they are relayed to the MQTT edge of their respective RSU.

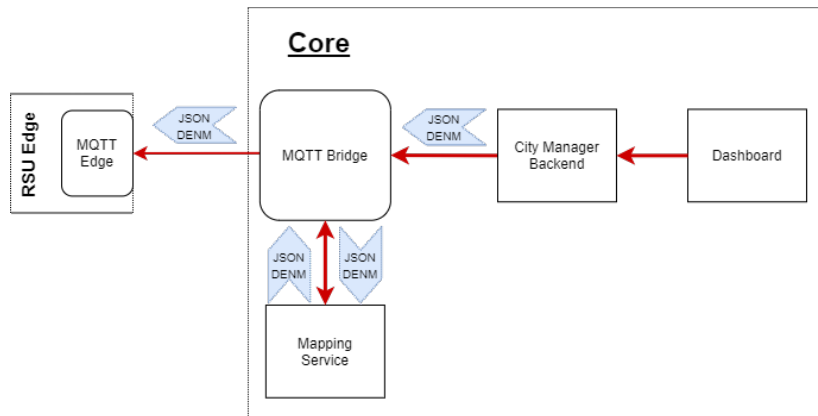


Figure 3.10: Data flow between the core and the RSUs

3.3.8 Sending DENMs from the RSUs to the OBUs

The developed architecture only supports sending DENM from the infrastructure to the vehicles through RSUs, although as a future improvement it could be adapted to support sending DENMs to OBUs, via LTE, for example.

As previously explained, the target DENMs are published in the MQTT edge of each target RSU. After this, they are read by the MQTT service and routed to the ITS router, which will interpret them in the JSON format, encode them and send them to the E2U module. This module will take the Full DENM generated and broadcast it via DSRC. These messages will be captured by the OBUs of nearby vehicles. The U2E will listen for the packets in its wireless interface and redirect them to the ITS router, which will decode, interpret and sent the target DENM to the MQTT service. The MQTT service will publish this message in the local MQTT of the onboard unit, where it will be fetched by the Notification App and finally displayed in the Notification dashboard.

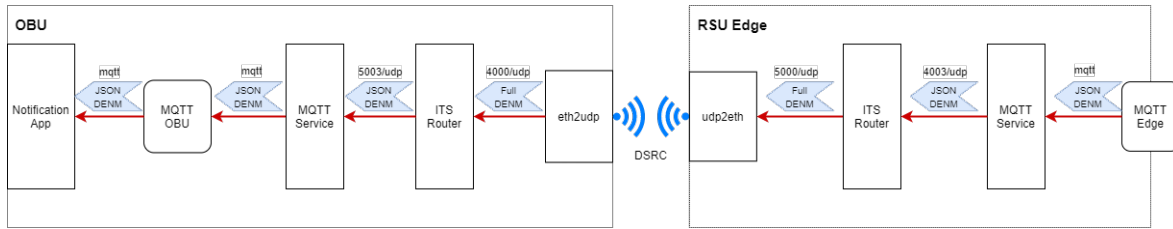


Figure 3.11: Data flow between the RSU and the OBU

3.4 SUMMARY

This chapter started with the description of the requirements, followed by a presentation of the high level components of the architecture - OBUs, edges and core - and later provided an analysis of the communication between modules and the flows of data. We will now summarise everything that has been previously explained and we will discuss how the requirements are addressed by the proposed architecture.

The vehicular data is collected by the OBU of the vehicle and can be sent to the infrastructure through a RSU edge via DSRC, through a gateway edge via LTE or directly to the MQTT bridge via LTE.

The sensor data is collected by the sensors themselves and is published in the MQTT edge of the RSU, where it is then relayed to the MQTT bridge.

After the data reaches the MQTT bridge, it can be consumed by the high level applications and services. This data is automatically persisted in the TimescaleDB database, where it can be accessed later. This allows the data to be consumed in real-time directly from the bridge or as historic data by accessing the database.

The proposed architecture supports I2V communication by allowing the user to generate an alert through a dashboard. This alert is generated in the form of a DENM and sent to the MQTT bridge, where the Mapping Service will compute the target RSUs that will receive this message. When the message reaches each RSU, it will be encoded and broadcasted via DSRC to nearby vehicles.

Overall, the architecture that we propose is able to meet all of the requirements presented in the start of this chapter. In the next chapter, we will dive deep into the implementation of the major modules.

Implementation

To implement the proposed solution, we needed to first approach the problem from a high level perspective and abstract from the complexity the individual modules. After that, we started defining the responsibility and functionalities of each module. In this chapter, we describe the implementation of the system proposed in the previous chapter.

We will start by documenting the implementation of the ITS modules, which are used repeatedly throughout the architecture and are considered the most important parts of the system. After this, we will provide an in-depth analysis to all the core's inner modules, followed by the implementation of the edges' inner modules and the implementation of the OBU's inner modules. Note that some modules may be omitted because they are trivial to understand, like the gateway. After this, we will introduce the test environment that we developed to validate the system's most important features.

4.1 ITS MODULES

The communication between the OBUs and the core can be performed either directly or through a RSU. In this section we will analyse the specific case, where this communication is done via a roadside unit, where these entities exchange ITS messages like CAMs and DENMs.

These messages contain multiple headers that must be interpreted and removed before the payload itself can be interpreted by the infrastructure. In order to address this problem we developed a central module that is responsible for encoding, decoding these messages and routing them from one port to another, the ITS router. We also developed a service that is responsible for generating ITS messages from the vehicle's telemetry, the Vehicle Service, and a service that publishes any messages received from the router in an MQTT broker and vice-versa, the MQTT service. In order for the generated ITS messages to be broadcasted from a vehicle and received by it, it was necessary to inject these messages into the wireless interface of the on board unit, and it was also necessary to packets that are received by the wireless interface of the board. To solve this task, we extended the modules E2U and U2E which support the described behavior.

The set of modules that were just described are designated as the ITS modules and together they perform the core ITS operations, which allow OBUs to send and receive CAMs and DENMs to and from RSUs.

4.1.1 ITS Router

One of the most important modules of this architecture, the ITS router, written in Java, encodes and decodes CAMs and DENMs and forwards them to other services. Note that the scope of this dissertation only covers these two types of ITS messages, but the ITS router was developed with the aim to easily allow the support of additional types of messages.

Initially this module was implemented as a simple python script, which listened for incoming ITS messages in the wireless interface and ignored the BTP and geonetworking headers by removing every byte before the byte sequence `b'\x07\xd1\x00\x00'`, thus retrieving only the payload, i.e., the ITS message itself. This solution proved to be unreliable, since all data contained in the headers was being discarded and this script could only be used to decode messages and not to encode and send them. An alternative solution would be to develop an encoder and decoder from scratch, but this would be too time-consuming and require manually implementing basic features and then adapting it to our use cases. Instead, we decided to use the Rendits Geonetworking library¹ which was developed from a previous library², and contains an extensive tool set that allows basic operations to be performed on ITS message, such as encoding and decoding, but also allows more complex operations as well. Note that this library is quite extensive and contains features that were not used in this dissertation, but it makes it easier to implement more complex operations in the ITS router in the future. The geonetworking library allows the interpretation of ITS messages but it requires high level modules to invoke it and use its methods and classes. For this reason, we decided to use and extend the Rendits Router³, which is an open-source module that already implements some of the basic routing features required for receiving and sending ITS messages. Both of these modules together create a more complex module, the ITS router, that is able to receive ITS messages, decode them and route them to the service that will use this data.

Apache Maven

The Apache Maven⁴ is a software project management and comprehension tool. It makes the build process easier for developers, by providing a uniform build system and encourages better development practices. One of the most important features of the maven apache is that it streamlines the build process, as mentioned, and this allows tests to be performed on the modules.

¹<https://github.com/rendits/geonetworking>

²<https://github.com/alexvoronov/geonetworking>

³<https://github.com/rendits/router>

⁴<https://maven.apache.org>

Compilation

Before being able to run on the hardware (OBUs, RSUs, etc.), the ITS router must be compiled. Since the ITS router uses an external submodule, the geonetworking library, this library must be compiled first as an ad-hoc module, and later included in the compilation of the ITS router. Maven is used to compile the module and it produces an output .jar file containing the binaries that can be executed in the respective hardware. Since each RSU and OBU runs the ITS router, and it would be very time consuming to compile this module in each one of them, we have decided to have a dedicated board that compiles it. The binaries are then sent to the remaining boards, allowing new changes to be deployed in the whole system.

Configuration

The ITS router can adopt different behaviours according to the values of its input parameters. Table 4.1 contains the full list of input parameters, their respective default value along with a brief description about each parameter.

This set of parameters can be loaded from a configuration file. Since the router reads these parameters on startup, it allows to change the behavior of the router without having to recompile, which is a time consuming task. Code 1 shows the content of the router.properties file which contains the main input parameters of the ITS router. If one of these parameters is omitted from the configuration file, that parameter will assume the default value.

The parameters can be divided into 3 groups: the input ports, the output ports and the state parameters, which contain identifying information about the board and will be included in the generated messages. Note that, for convenience, we have decided to assign input ports values in the range 4000-4999 and output ports values in the range 5000-5999.

```
1      localPortForUdpLinkLayer=4000
2      localPortForUdpLinkLayerNoEther=4011
3      portRawCam=4002
4      portRawDenm=4003
5      portRcvFromVehicle=4005
6
7      portLinkLayer=5000
8      portSendCamJson=5002
9      portSendDenmJson=5003
10     SDNCamPort=5004
11     vehicleCamPort=5009
12     vehicleDenmPort=5010
13
14     macAddress=6e:06:e0:01:00:70
15     stationType=15
16     additionalFieldsEnabled=1
17     sendToSDNService=1
```

Code 1: ITS Router - Configuration file

Table 4.1: ITS Router - Configuration parameters

Parameter	Type	Default	Description
localPortForUdpLink-Layer	integer	4000	Port that receives full ITS messages
localPortForUdpLink-LayerNoEther	integer	4011	Port that receives proper ITS messages
portRawCam	integer	4002	Port that receives JSON CAMs
portRawDenm	integer	4003	Port that receives JSON DENMs
portRcvFromVehicle	integer	4005	Port that receives simple its messages
portLinkLayer	integer	5000	Output port to where the full ITS messages will be sent
portSendCamJson	integer	5002	Output port to where JSON CAMs will be sent
portSendDenmJson	integer	5003	Output port to where JSON DENMs will be sent
SDNCamPort	integer	5004	Output port to where JSON CAMs will be sent. Used for the SDN service. Works as a secondary port of portSendCamJson.

Parameter	Type	Default	Description
vehicleCamPort	integer	5009	Output port to where simple CAMs will be sent. Used to sent messages to the Vehicle Service.
vehicleDenmPort	integer	5010	Output port to where simple DENMs will be sent. Used to sent messages to the Vehicle Service.
macAddress	string	12:34:56:78	Mac address of the machine that is running the ITS router
stationType	integer	15	Type of the station as defined by ETSI ⁵ . Has value 15 if the current board is used as an RSU or 5 if the board is used in a passenger car, for example.
additionalFieldsEnabled	integer	1	Has value 1 if the received timestamp and RSSI are included as the last bytes of the payload and 0 otherwise.
sendToSDNService	integer	1	Will send JSON CAM to port SD-NCamPort if it has value 1 and will not send anything otherwise.

Port mapping

The ITS router has input ports which receive ITS messages in different formats, and output ports to where the converted messages will be sent. Its main goal is to route the messages from the input ports to the appropriate output ports. Table 4.2 contains the mapping of the input and output ports. Each row contains the input port where the messages are received and

⁵https://www.etsi.org/deliver/etsi_en/302600_302699/3026360401/01.02.01_30/en_3026360401v010201v.pdf

the respective format, and the output port to where the messages are sent and the respective format, e.g., it is possible to observe that the router receives full CAMs in UDP port 4000, decodes them, converts them into the JSON format and sends them to UDP port 5002.

Default input port	Input format	Default output port	Output format
4000	Full DENM	5003	JSON DENM
4000	Full CAM	5002	JSON CAM
4002	JSON CAM	5000	Full CAM
4003	JSON DENM	5000	Full DENM
4005	Simple CAM	5000	Full CAM
4005	Simple DENM	5000	Full DENM
4011	Proper CAM	5002	JSON CAM

Table 4.2: ITS Router - Input and output port mapping

4.1.2 MQTT Service

The MQTT service, part of the ITS module set, is a simple module that works as the interface between the ITS router and a mosquitto instance. This module listens for incoming JSON messages in the input topics for CAMs and DENMs, converts them to a byte stream and sends CAMs to portRawCam and DENMs to portRawDenm, as seen in table 4.1.

This module was developed from scratch, i.e., it is not an extension of a previously implemented module. It does not contain any logic related to the encoding or decoding of ITS messages and, for this reason, it does not depend on the geonetworking library, so we had the flexibility to choose the programming language that we wanted to use to develop it. We chose to develop this module in Java since it is faster than interpreted languages like Python ⁶, and this is a low-level module and will handle the exchange of CAM messages with a frequency of 10hz for each vehicle, which can become a bottleneck for a large number of vehicles.

⁶<https://medium.com/swlh/a-performance-comparison-between-c-java-and-python-df3890545f6d>

Compilation

The MQTT service needs to be compiled with Maven before being run on the hardware. Unlike the ITS router, this module does not contain any logic to handle ITS messages; in fact, the concept of an ITS message is unknown in the context of the MQTT service. Without this dependency, this service can be compiled without including the geonetworking module and works as a standalone module whose only two main tasks are to read JSON messages, whatever their content is, and route them to a UDP port and vice-versa.

After the compilation, the generated .jar file can be sent to the board where it will be run. Since the compilation is a time-consuming task and the generated binaries are compatible with all RSUs and OBUs, we use a dedicated board that compiles the MQTT service and sends the generated binaries to the remaining boards, effectively deploying changes to the whole system.

Configuration

Much like the ITS router, the MQTT service adopts different behaviors according to its input parameters. Table 4.3 contains the full list of input parameters, their respective default values along with a brief description about each parameter.

This set of parameters is loaded from a configuration file when the module is executed, allowing to change its behavior without having to recompile the code, which is a time-consuming task. Code 2 shows the content of the `mqttservice.properties` file which contains the parameters of this service. Since this module is connected to both the ITS router and the mosquitto instance, we can divide these parameters into 2 separate groups: the ports that are connected to the router's input and output ports, and the input and output topics of the mosquitto instance. If one of these parameters is omitted from the configuration file, that parameter will assume the default value.

It is also worth pointing out that a certain port may be an output port of the MQTT service and an input port for the ITS router, which is the case of the `routerInputPortWithGeonetHeader`.

```
1   routerInputPortWithGeonetHeader=4011
2   routerInputPortCam=4002
3   routerInputPortDenm=4003
4   routerOutputPortCam=5002
5   routerOutputPortDenm=5003
6
7   mqttBrokerUrl=tcp://127.0.0.1:1883
8   inEncodeWithGeonetCamsTopic=geral/cams
9   inEncodeDenmsTopic=lab/target/denm
10
11  outEncodeCamsTopic=cam_decoded
12  outEncodeDenmsTopic=denm_decoded
```

Code 2: MQTT Service - Configuration file

Table 4.3: MQTT Service - Configuration parameters

Parameter	Type	Default	Description
routerInputPortWith- GeonetHeader	integer	4011	Port to where proper ITS messages will be sent, once they are published as a byte stream in the mosquitto instance
routerInputPortCam	integer	4002	Port to where the JSON CAMs will be sent once they are received in the input CAM topic
routerInputPortDenm	integer	4003	Port to where the JSON DENMs will be sent once they are received in the input DENM topic
routerOutputPortCam	integer	5002	Port where the JSON CAMs will be received from. These messages will then be published in the output CAM topic
routerOutputPortDenm	integer	5003	Port where the JSON DENMs will be received from. These messages will then be published in the output DENM topic

Parameter	Type	Default	Description
inEncodeWithGeonet-CamsTopic	string	geral/cams	Topic where the proper CAMs will be received
inEncodeDenmsTopic	string	target/denm	Topic where the target DENMs are received. These messages are used to warn vehicles about potential dangers in their environment
outEncodeCamsTopic	string	cam_decoded	Topic to where the decoded JSON CAMs will be published
outEncodeDenmsTopic	string	denm_decoded	Topic to where the decoded JSON DENMs will be published

Port mapping

The MQTT service serves as an interface between the ITS router and the MQTT. Its main goal is to route the messages from the router ports to the appropriate MQTT topics and vice-versa. Table 4.4 contains the mapping of the ports and topics. Each row contains the input where the messages are received and the output to where the messages are sent. Note that the MQTT service, not only contains input ports, but also contains input topics, so we will treat both of these simply as inputs. We will likewise treat output ports and output topics simply as outputs. Ports are represented as integers, e.g., port 5002 and topics are represented as string, e.g., geral/cams.

Default input	Default output	Message format
geral/cams	4011	Proper CAM
target/denm	4003	Simple DENM
5002	cam_decoded	JSON CAM
5003	denm_decoded	JSON DENM

Table 4.4: MQTT Service - Input and output port mapping

Data flow

Figure 4.1 shows the flow of data of the MQTT service module, when integrated with other ITS modules. By analysing the figure we can observe that, in data flow (1) both JSON CAMs and JSON DENMs are sent to the ITS router, where they will be decoded and sent to the wireless interface through U2E.

It is also possible to observe that a different flow exists on port 4011, data flow (2), where proper ITS messages are received on the MQTT instance and are sent to the ITS router, where they will be properly decoded and then republished in the MQTT instance, through the MQTT service.

Finally, there is one remaining flow, data flow (3), which occurs, when ITS messages are received by the ITS router, on port 4001, are decoded, interpreted, and then published in the respective topic, i.e., one topic for CAMs and another topic for DENMs.

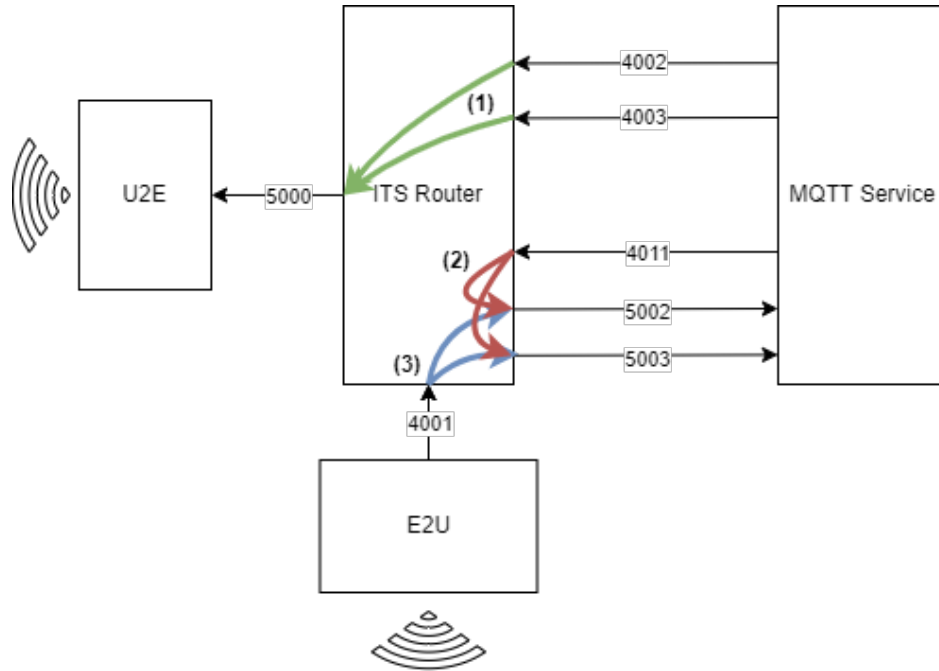


Figure 4.1: MQTT Service data flow

4.1.3 Vehicle Service

The Vehicle Service, part of the set of ITS modules, is used to generate ITS messages. It will generate CAMs periodically with a frequency of 10 Hz, if the device where it is being executed is an OBU, and with a maximum frequency of 1 Hz if the device where it is being executed is a RSU. DENMs are generated when a trigger message is received by the Vehicle Service. This service is based on a test suite that was developed previously to test the Rendits Router. This test suite generated CAMs in a simpler format than the one that we present in this dissertation. These messages only contained a small subset of attributes. The remaining attributes would later be added by the rendits router to make them ETSI compliant. We decided to completely eliminate this simplified version of CAMs, and delegated the CAM generation process fully to the Vehicle Service. CAMs are then sent to the ITS router where they will be encoded and broadcasted through the wireless interface of the board.

This module can be used in OBUs, which will allow the vehicle to send its telemetry and generate alerts, but it can also be used in RSUs for the same purpose. At the time of writing of this dissertation, the CAMs generated by the RSUs are discarded by the high level applications, since they do not provide additional information about the roadside unit, but the DENMs generated are broadcasted via the wireless interface and are received by the vehicle's OBU. This message can then be displayed as an alert to the user, in the Notification App that runs in the OBU itself. In the current implementation of the infrastructure, the DENMs generated by RSUs are not sent to the MQTT bridge via the ITS router, because it does not currently support this. Instead, these DENMs must be published directly in topic `px/denm_decoded`, where `px` is the ID of the RSU that generated the DENM. A new route could be added to the ITS router that received Simple DENMs, converted them to JSON

DENMs and sent them to the MQTT service that would publish them in the edge MQTT. These messages would later be fetched by the MQTT bridge and made available for the high level applications.

Configuration

The Vehicle Service can adopt different behaviors depending on the values of its input parameters, which can be loaded from a configuration file, like the other ITS modules. Table 4.5 contains the full list of input parameters, their respective default values along with a brief description about each parameter.

This set of parameters is loaded from a configuration file when the module is executed, allowing us to change its behavior without having to recompile the code, which is a time consuming task. Code 3 shows the content of the `vehicleservice.properties` file which contains the parameters of this service.

Another important thing to note is that ports `portRcvFromRouterCam` and `portRcvFromRouterDenm` allow the Vehicle Service to receive CAMs and DENMs, respectively, from the router and be processed by this service at runtime, but they aren't being used at the time of writing of this dissertation. Even though they are not used now, this makes the Vehicle Service more flexible, since this feature may be useful in the future.

As mentioned previously, this service can be used in both OBUs, which change their position in real-time, and RSU, which cannot change their position in real-time. Because roadside units cannot move, thus their coordinates will not change, we have decided to include their coordinates - latitude and longitude - in their configuration file. Note that the latitude and longitude configuration parameters may only be included in the configuration file of RSUs.

```
1      portRcvFromRouterCam=5009
2      portRcvFromRouterDenm=5010
3      portSendCam=4005
4      portSendDenm=4005
5      portToListenAccident=9999
6
7      stationId=15
8      stationType=5
9      frequency=10
10
11     latitude=40.63476
12     longitude=-8.66038
```

Code 3: Vehicle Service - Configuration file

Table 4.5: Vehicle Service - Configuration parameters

Parameter	Type	Default	Description
portRcvFromRouterCam	integer	5009	Port from where incoming decoded CAMs are received. This port is connected to the ITS router.
portRcvFromRouterDenm	integer	5010	Port from where incoming decoded DENMs are received. This port is connected to the ITS router.
portSendCam	integer	4005	Port to where generated CAMs will be sent. This port is connected to the ITS router.
portSendDenm	integer	4005	Port to where generated CAMs will be sent. This port is connected to the ITS router.
portToListenAccident	integer	9999	Port to where messages can be sent to trigger the generation of a DENM by this service.
stationId	integer	No default	The ID of the current station. This ID uniquely identifies a station (OBUs, RSUs, etc.).

Parameter	Type	Default	Description
stationType	integer	15	Type of the station as defined by ETSI ⁷ . Has value 15 if the current board is used as a RSU or 5 if the board is used in a passenger car, for example.
frequency	integer	10	Frequency, in Hz, of the generation of CAMs, e.g., if it has value 10, then the Vehicle Service will generate 10 CAMs per second.

Data flow

Figure 4.2 shows the different flows of data of the Vehicle Service.

It is possible to observe data flow (1), where the vehicle data is generated by the Vehicle Service and is then sent to the ITS router, through port 4005, where it will be decoded and then sent to the wireless interface through U2E. This flow of data is triggered, when the Vehicle Service generates a CAM, for example.

It is also possible to observe another flow of data, data flow (2), where ITS messages are received by the ITS router, are decoded and then sent to the Vehicle Service. This data flow is not currently being used but opens up the possibility for future extension.

⁷https://www.etsi.org/deliver/etsi_en/302600_302699/3026360401/01.02.01_30/en_3026360401v010201v.pdf

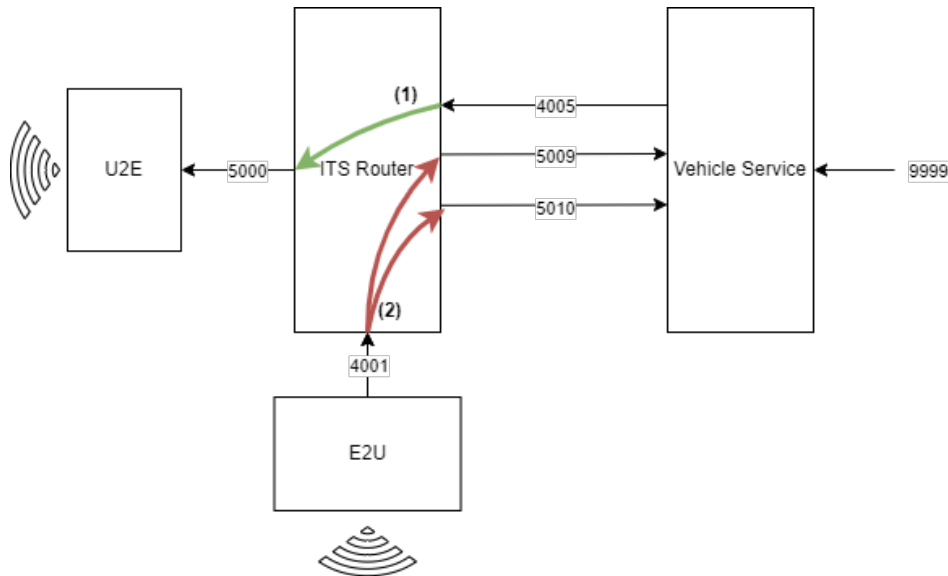


Figure 4.2: Vehicle Service data flow

4.1.4 Eth2Udp & Udp2Eth

These two low level modules, also known as the utoepy module (E2U+U2E), are responsible for routing packets received on the wireless interface of the board to a UDP port, E2U, and routing messages received in a UDP port and transmit them via the wireless interface. These modules are the extension of previously developed open-source code ⁸. The original utoepy required its dependencies to be installed with `easy_install`, which is deprecated. Furthermore, this module can only be executed using `python2` which has also been deprecated and, while it is possible to install `python2`, it is a complex and time-consuming task. For this reason we decided to rewrite the utoepy in `python3` instead, using the `scapy` library.

Since the utoepy module will handle a load as big as the ITS router, it would be important to implement this module using a well performing (fast) compiled programming language, like C or C++, but since the module was already implemented in Python we decided to develop it in this language to save some time, but it could be rewritten in C++ for better performance in the future.

Since this module was developed in Python and this is an interpreted language, it is not required to be compiled before being executed. For this reason, we must only send the `.py` file to the boards and execute it.

Configuration

Much like the other ITS modules, the utoepy loads multiple input parameters from a configuration file, which can be passed as a program argument. Since the configuration files are in the INI format, both the E2U and the U2E can read their input parameters from the same configuration file. Code 4 contains the example of a configuration file that contains the parameters of both of these modules. Table 4.6 contains the name, type, default value and description of the utoepy parameters.

⁸<https://github.com/alexvoronov/utoepy>

```

1      [eth2udp]
2      address = 127.0.0.1:4000
3      interface = wlan0
4      mode = raw
5      keep_own_frames = no
6      include_extra_fields = yes
7      station_type = rsu
8
9      [udp2eth]
10     port = 5000
11     interface = wlan0
12     mode = raw

```

Code 4: UTOEPY - Configuration file

Table 4.6: E2U - Configuration parameters

Parameter	Used by	Type	Default	Description
address	Eth2Udp	string	127.0.0.1:4000	The UDP message received in the wireless interface will be sent to this ip and port
interface	Eth2Udp	string	wlan0	The name of the wireless interface where the packets will be received
mode	Eth2Udp	string	raw	Cooked packet mode prepares the incoming packet to be sent to a UDP port. Raw packet mode assumes that UDP payload is already ready to be sent to the UDP port.

Parameter	Used by	Type	Default	Description
keep_own_frames	Eth2Udp	string	no	If it has value 'yes' keeps frames originating from the wireless interface and discards them otherwise
include_extra_fields	Eth2Udp	string	yes	If it as value 'yes', the timestamp and the rssi of arrival will be appended to the end of the payload, otherwise the packet will be forwarded as is.
station_type	Eth2Udp	string	rsu	Used to chose the mac address of the hardware. It will append a different mac address in the incoming packets according to the type of hardware being used: RSU or OBU
port	Udp2Eth	integer	5000	UDP port from where the outgoing messages will be read. These messages will then be transmitted through the wireless interface.
interface	Udp2Eth	string	wlan0	The name of the wireless interface where the packets will be received

Parameter	Used by	Type	Default	Description
mode	Eth2Udp	string	raw	Cooked packet mode prepares the incoming packet to be sent to a UDP port. Raw packet mode assumes that UDP payload is already ready to be sent to the UDP port.
mode	Udp2Eth	string	raw	Cooked packet mode prepares the incoming UDP packet to be sent from the wireless interface. Raw packet mode assumes that UDP payload is already ready to be sent.

4.2 MESSAGE QUEUING TELEMETRY TRANSPORT

As explained previously, each edge contains a MQTT instance, which stores the local data of the edge, and the MQTT bridge which aggregates the data contained in all of the RSUs. In this section we will explain why we chose to use this approach instead of just having the MQTT edge instances.

4.2.1 MQTT Edge

Each RSU has a local Message Queuing Telemetry Transport (MQTT) instance running, that contains topics where the data from vehicles and sensors (radar, camera, etc.) is published. The data from the MQTT edges is then fetched by the MQTT bridge.

For this dissertation, we used Eclipse Mosquitto⁹ which provides an open source MQTT broker. Its lightweight MQTT protocol implementation makes it suitable for full power machines, as well as for the low power and embedded ones, like the OBUs and RSUs.

In order to explain why we chose to create a MQTT instance for each edge and aggregate that data in the MQTT bridge, let us consider a set of local MQTT instances, P_1, P_2, \dots, P_N , where P_i is the i th MQTT instance and N is the total number of RSUs (assuming that each RSU has one MQTT instance).

⁹<https://mosquitto.org>

Consider also that we chose not to aggregate the data in the bridge. If, for example, the CM wanted to collect data from all N RSUs, then it would have to perform $K \cdot N$ subscriptions, where K is the number of subscriptions done to each MQTT instance. Every time a new RSU was added, the CM would have to be changed to include K new subscriptions.

Consider a set of services that make use of this data, S_1, S_2, \dots, S_X , where S_i is the i th service that uses the data from the RSUs and X is the number of available services. If every service (CM, VRU Safety App, etc.) requires the information available on all these RSUs, there would be a total of $X \cdot N \cdot K$ active subscription. This approach would quickly become very difficult to manage when the number of services and RSUs increases.

4.2.2 MQTT Bridge

MQTT Bridge is a MQTT instance whose main purpose is to subscribe to all of the topics from all the RSUs, thus allowing to gather all the available data in one place. This process can be visualized in figure 4.3. Consider the example used in 4.2.1. Using the bridge approach, every time a new RSU is added, we just need to change its configuration to also fetch data from the new RSU. The services only perform K subscriptions to the bridge and they do not have to be changed when new RSUs are added. If, like in the previous example, every service, requires the data available on these RSUs, there would be a total of $X \cdot K + K \cdot N = K \cdot (N + X)$ subscriptions, which is a significant reduction from the number of active subscriptions in the previous approach, and also makes the system much more manageable. Note that this solution does not decrease the amount of messages exchanged between RSUs and services, it simply allows an easier management of the active subscriptions.

Code 5 shows a possible configuration file for the MQTT bridge. In this file it is possible to observe that the bridge will be running on port 1883 on the machine where it is hosted, and will fetch information from the MQTT instance running on `atcll-p1-jetson.nap.av.it.pt:1883`, and will make it available in topics with the prefix `p1/`, i.e., if a message is published on topic `denm_decoded` of the instance running in `atcll-p1-jetson.nap.av.it.pt:1883`, then that message can be accessed by subscribing topic `p1/denm_decoded` of the bridge. We can also observe that this configuration file allows the bridge to fetch information from instances `atcll-p2-jetson.nap.av.it.pt:1883` and `atcll-testbed-rsu.nap.av.it.pt:1883`.

Apart from fetching information, the bridge also allows to propagate messages to and from the core to the edges. It is possible to observe that the configuration of bridge-lab contains additional logic for outgoing messages, i.e., all messages published to topic `lab/target/denm` of the bridge will be published in topic `lab/target/denm` of the MQTT instance running on `atcll-testbed-rsu.nap.av.it.pt:1883`.

For this dissertation we only used a small subset of configurations for the bridge, but there are many more available. The full list of commands can be found online ¹⁰.

¹⁰<http://www.steves-internet-guide.com/mosquitto-bridge-configuration>

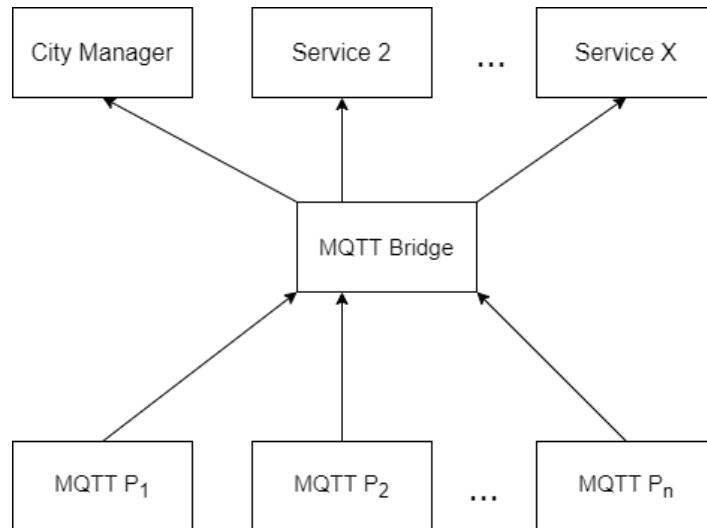


Figure 4.3: MQTT Bridge fetches information from all RSUs, which the services can then collect

```

1  port 1883
2
3  connection bridge-1
4  address atc11-p1-jetson.nap.av.it.pt:1883
5  topic # in 0 p1/ ""
6
7  connection bridge-2
8  address atc11-p2-jetson.nap.av.it.pt:1883
9  topic # in 0 p2/ ""
10
11 connection bridge-lab
12 address atc11-testbed-rsu.nap.av.it.pt:1883
13 topic # in 0 lab/ ""
14 topic lab/target/denm out "" ""

```

Code 5: MQTT Bridge - Configuration file

4.3 CORE

As previously hinted, the core hosts an eco-system of services, such as the mapping service, high level applications, like the city manager and the main data aggregator, the MQTT bridge.

The data is fetched from the RSUs by the MQTT bridge, which makes the data available for consumption by the high level applications and services. The core is located in the cloud and, for this dissertation, we chose to implement it in a virtual machine in the IT data center, but it can easily be hosted in another data center, like the Bosch data center.

Note that the core contains a large number of services and some of them were not implemented by the author of this dissertation. The ones developed by the author are the CM and the Mapping Service. The data aggregator service, that fetches data from the MQTT bridge and persists it in the database cluster was not developed by the author.

4.3.1 City Manager

The City Manager is a high level application that consumes the vehicle and sensor data available in the MQTT bridge. The CM consumes this data in two ways, by listening for real-time data and presenting it in a dashboard, the Aveiro in Real-Time (ART) and by querying historic information and replaying that data through a GUI, the Incident Replayer (IR).

The CM includes a backend application that serves information for both frontend application - the ART and the IR. These frontend applications are add-ons to the already existing frontend app Aveiro Open Lab¹¹.

We wanted to use this web app to showcase other projects as well, not only internally, but also to the public. Our goal was to make the app accessible for everyone, independently of the equipment, operating system or specifications that they were using, so we decided to adopt a user driven testing methodology from the beginning, by asking feedback from non-technical users, that would still allow to have the flexibility of making fast changes whenever we needed to. In order to achieve this, we focused more on non-functional tests, while performing functional tests whenever possible.

The backend of the City Manager contains a Node.js server that exposes a REST API, that allows the client side to make HTTP requests in order to fetch historic information, for example. The backend also contains a websocket server which is used to send data in real-time to the client. Figure 4.4 shows the architecture of the CM. The frontend was developed in Nuxt.js, which is a free and open source web application framework based on Vue.js, Node.js, Webpack and Babel.js. Nuxt.js offers development features such as server side rendering, automatically generated routes, search engine optimization (SEO) improvement and is, in our opinion, easier to learn when compared to other alternatives such as Angular or React.

¹¹<https://aveiro-open-lab.pt>

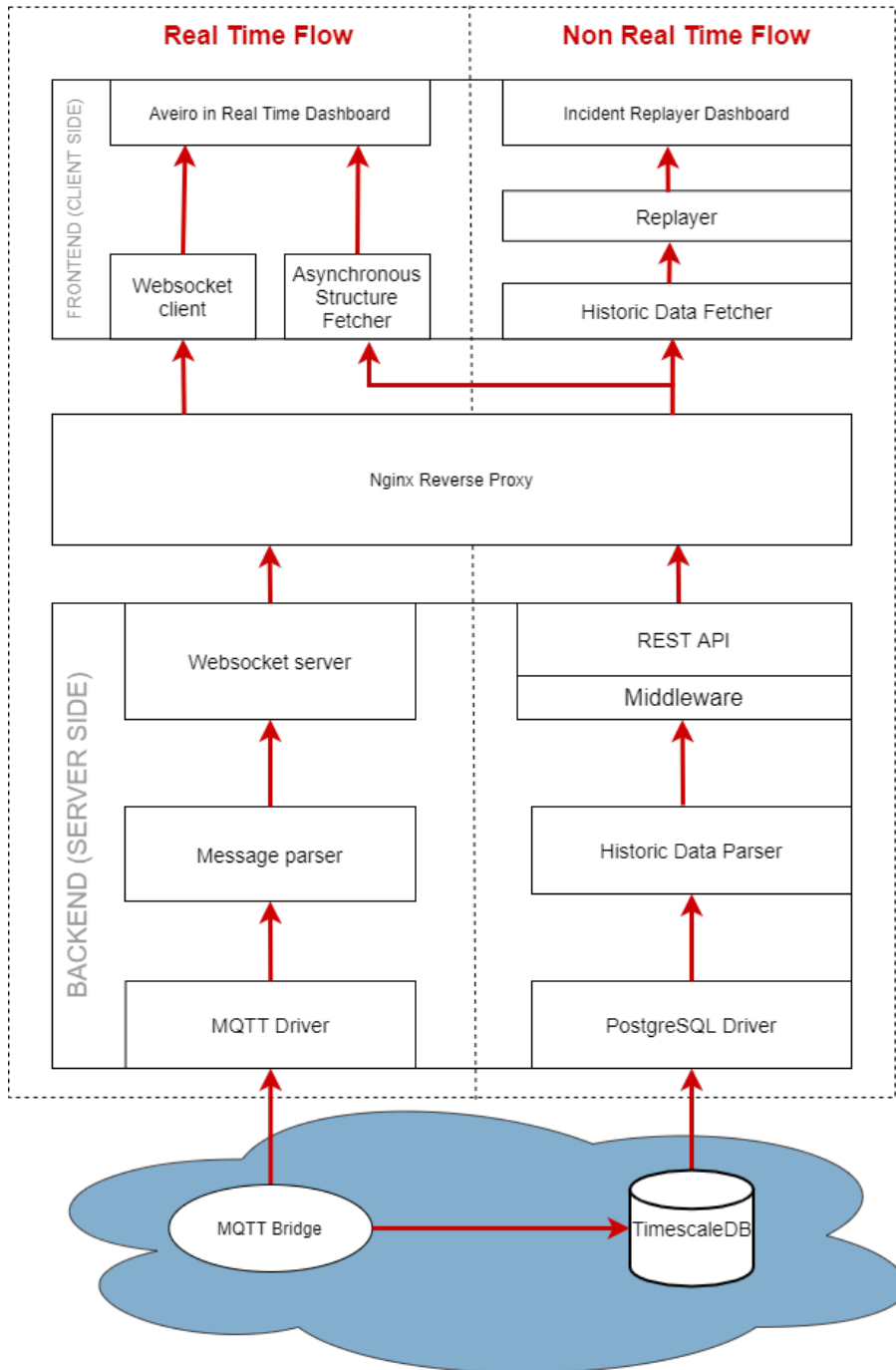


Figure 4.4: City Manager - Architecture

4.3.2 Mapping Service

Intelligent Transportation System (ITS) messages (CAM, DENM, etc.) can be sent from a service through the infrastructure to either a specific OBU or to an area of effect (AOE), where all OBUs within that area will receive and interpret the message. In order to send an ITS message from the infrastructure to the target OBU, the infrastructure must be able to route these messages to an RSU first, which will then transmit the messages via WAVE to the OBU itself.

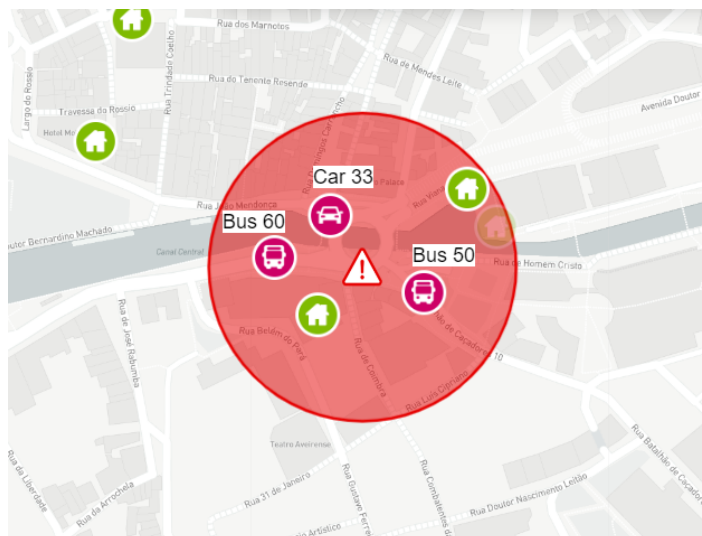


Figure 4.5: Area of effect - Car 33, Bus 60 and Bus 50 received the DENM, since they were within the area of effect.

One of the main challenges of this process is to determine which RSU candidates will maximize the probability of the messages reaching the destination OBU. Since there are many algorithms that could be used to determine this, we chose to delegate this decision making responsibility to a single entity, the Mapping Service.

Selection of the algorithm

Because different algorithms can be used, we have decided to adopt the abstract concept of a Mapping Strategy and implement it in different ways. Figure 4.6 illustrates this. We could also see MappingStrategy as the equivalent of an interface in a programming language such as Java, for example, and GeographicMappingStrategy and LastSeenMappingStrategy as classes that implement that interface. This abstraction allows to quickly test different mapping strategies with minimal code changes, making the development process less prone to bugs.

Mapping Strategy

DENMs contain the coordinates and the radius of their Area of Effect (AOE), which means that we could consider sending messages to RSUs that are geographically closer to the coordinates of the center of this area of effect, the epicenter. But this would not work for every ITS message, because, for example, CAMs, do not have such fields. Another strategy that could be used, could be routing the messages to the RSUs that have recently received messages from the destination OBU. While it is not guaranteed that both the RSU and OBU will still have a connection between each other, there is some probability that this will be the case. This last strategy would only work in the case where we only want to send messages to a specific OBU and not to an AOE, so we must make sure that the Mapping Service has different behaviours according to the input message.

The code for the MappingStrategy interface/class is presented bellow.

Listing 4.1: MappingStrategy class

```
class MappingStrategy:

    def best_connections(self, topic, message):
        pass
```

This is a simple class that defines the method *best_connections*, which will be implemented by sub classes of the MappingStrategy class. This method will be responsible for the decision making and heuristics selection process that will determine the best destination RSUs, in order to maximize the probability of the DENM reaching the destination OBUs.

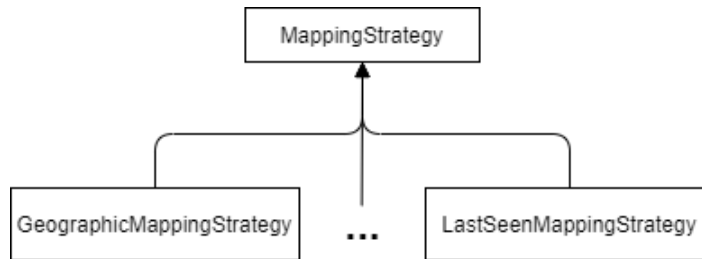


Figure 4.6: Geographic Mapping Strategy and Last Seen Mapping Strategy implement specific algorithms for routing the ITS message

Geographic Mapping Strategy

This strategy chooses to send the messages through RSUs that are within a certain distance, `MAX_DISTANCE_FROM_TARGET`, represented in meters from the center of the coordinates specified by the DENM. It also fetches the location of all the available RSUs by performing an HTTP GET request to an Airtable API that contains this and more information about the RSUs available in the city of Aveiro. This information is fetched periodically, which allows the strategy to automatically be aware of new RSUs and requiring no code changes. It is also aware of which RSUs are active, i.e., which RSU locations have equipment installed and are able to receive messages, and which are inactive.

This algorithm can be implemented with the following pseudo code. Note that some methods were omitted for simplification purposes.

Listing 4.2: GeographicMappingStrategy class

```
class GeographicMappingStrategy(MappingStrategy):

    function best_connections(topic, message):
        target_position = message.get_event_position()
        target_coordinates = (
            target_position.latitude(),
            target_position.longitude()
        )
```

```

best_rsu_connections = []
for rsu_id in road_side_units:
    rsu_info = road_side_units[rsu_id]
    rsu_coordinates = (
        rsu_info.latitude(),
        rsu_info.longitude()
    )
    distance_between_rsu_and_target = calculate_distance(
        target_coordinates,
        rsu_coordinates
    )
    if distance_between_rsu_and_target <=
        MAX_DISTANCE_FROM_TARGET:
        best_rsu_connections.append(rsu_id)

return best_rsu_connections

```

Note that with this strategy, the Mapping Service fetches the information about all the available RSUs, every 5 minutes using a daemon thread. This is done to ensure that the Mapping Service is able to make calculations with the most up to date information available without requiring a manual restart every time a new RSU is added.

In order to compare the performance of this strategy to other strategies, is it important to perform the time and space complexity analysis of this approach. For simplification purposes, we will consider that the time and space complexity of the computation of the geographic distance between two points is $O(1)$. Consider N to be the total number of active roadside units.

We will now present the time and space complexity analysis of this algorithm:

- **Worst case** - In the worst case, the time complexity is $O(N)$, since the algorithm calculates the distance between the epicenter of the alert and each RSU. The space complexity is $O(N)$, since, in this case, all RSUs are at a distance of less or equal to `MAX_DISTANCE_FROM_TARGET` meters, therefore the algorithm would have to keep track of all the N RSUs in the final list. In sum:
 - **Time complexity** - $O(N)$
 - **Space complexity** - $O(N)$
- **Best case** - In the best case, the time complexity is $O(N)$, since it will still have to calculate the distance for all RSUs, and there is no condition that would allow it to exit early. The space complexity is $O(1)$, since, in this case, all RSUs would be at a distance of more than `MAX_DISTANCE_FROM_TARGET` meters from the target coordinates and no elements would be added to the list. In sum:
 - **Time complexity** - $O(N)$
 - **Space complexity** - $O(1)$

Figure 4.7 illustrates an example where this strategy was used. For this example the strategy was provided with the location of the RSUs, a `MAX_DISTANCE_FROM_TARGET=300` (meters) and the coordinates of the DENM, which is represented by the purple circle in the middle (close to P6). It is possible to observe from the figure that, for this example, RSUs P16, P17, P5, P6, P20 are considered the best candidates and the messages will be routed to them. It is worth mentioning that P7, P42 and P14 are within range, but were not selected because these are considered to be inactive and are thus unable to receive any message. It is also worth mentioning that P8 was not selected because the RSU is not inside the circle, since its coordinates are located in the middle of the icon displayed (green circle with a house in the middle).

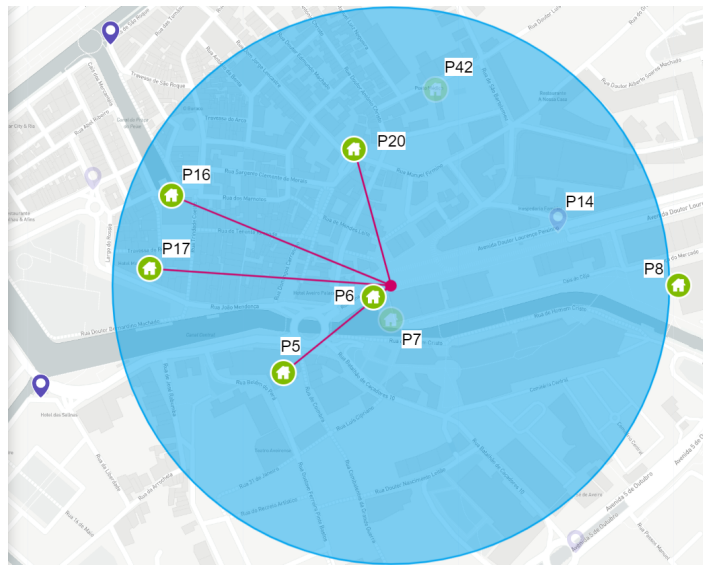


Figure 4.7: Active RSUs within 300 meters of the epicenter of the DENM were chosen

Implementation

The Mapping Service is divided into two main sections: the updater and the trigger. The updater section is composed by a MQTT Client and a Mapper module, which are responsible for keeping track of all connections between RSUs and OBUs. The trigger section is composed by a MQTT Client and the Mapping Strategy, which send the incoming DENM to the destination RSUs.

Updater section

As previously mentioned, the updater section keeps track of the connections between RSUs and OBUs. Every time an ITS message, that was sent from an OBU, is received in the central MQTT bridge, the Mapper module is updated with that connection. Note that, even though the Mapping Service only allows DENMs to be routed, we keep track of both incoming CAMs and DENMs, because the main objective of the Mapper module is to keep track of all available connections. Other modules can lookup all connections at any point in time, by fetching this information from the Mapper module.

Trigger section

This section allows the Mapping Service to trigger the computation of the best candidates and the subsequent propagation of the incoming DENM. This process is triggered when a DENM is published in the topic `target/denm`. After the target RSUs are found, the incoming DENM is routed to the destination topics `pi/target/denm`, where p_i is the ID of i th roadside unit.

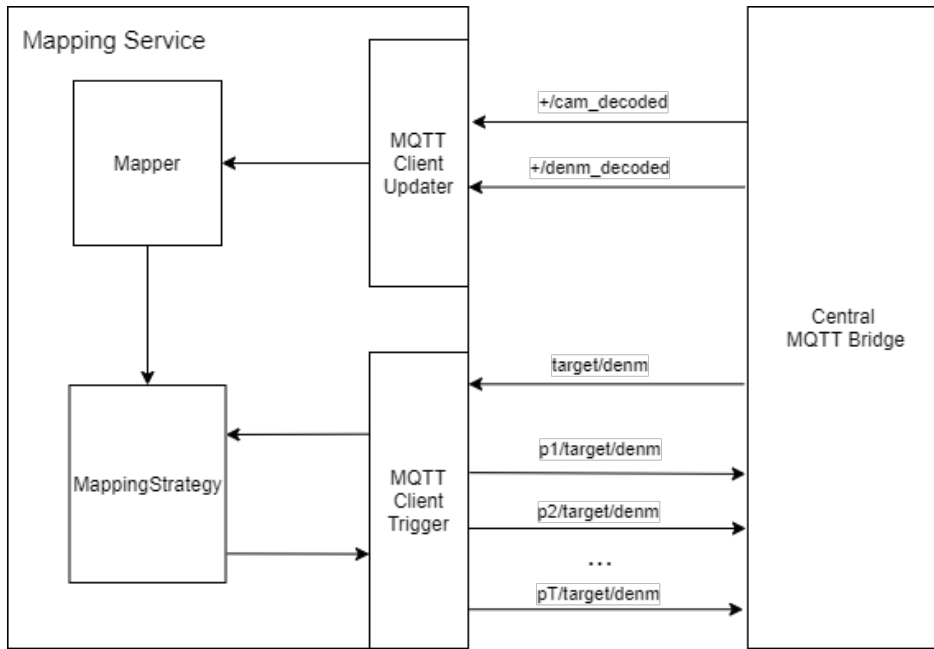


Figure 4.8: Mapping Service - Data Flow Diagram

4.4 NOTIFICATION APP

The proposed architecture allows the user to generate DENMs and send them to the OBUs of vehicles that are inside the DENM's AOE, but the only way to know if an alert reached a vehicle was to listen to the MQTT on the OBU itself. For this reason, we decided to create the Notification App, which is a web app, accessible through a browser, that allows users to be notified when the vehicle receives an alert, whether this alert came from the infrastructure or not.

The Notification App runs in the OBU of each vehicle and is divided into two submodules: the Notification Server and the Obu Alert Web App:

- **Notification Server** - Consists of a simple Node.js server that listens for incoming DENMs in topic `denm_decoded` on the OBU MQTT instance. It also contains a websocket server that sends an alert to the frontend whenever a DENM is received in the MQTT. This alert contains the information of the DENM (coordinates of the epicenter, timestamp of its creation, etc.).
- **Obu Alert Web App** - Consists in a frontend application that contains a websocket client that is connected to the backend websocket server. When an alert is received, its information is displayed in a GUI.

It is possible to access the Notification App through a browser, on a mobile and on a desktop. Since this app was also developed for mobile, this opens up the possibility to have a tablet that displays this GUI to the driver. This is possible because the OBU was also configured as an access point, so that the tablet would be able to connect to it. Once this connection is established, the user can simply connect to the web app via the browser.

4.5 DEVELOPMENT, DEPLOYMENT & TESTING

During the development of this dissertation, the CM was publicly accessible, which allowed users to interact with the most stable and up-to-date version of the CM. Since the architecture of the core is quite complex, we decided to implement the DevOps best practices and implement these services as docker containers. We also decided to use Gitlab CI/CD to create one pipeline for each service, i.e., one pipeline for the frontend, one pipeline for the backend of the CM, and one pipeline for the Mapping Service.

We also decided to follow the 5 SOLID principles of Object Oriented Design: Single-responsibility, open-close, liskov substitution, interface segregation and dependency inversion. The use of these principles allowed to greatly mitigate the number of bugs in the code, and also allowed to implement new features much faster.

Because of our choice to follow the SOLID principles, this allowed to separate the code into different contexts, by using classes, which allowed to easily produce unit, integration and end-to-end tests.

Figure 4.9 shows the pipeline of the ITS router that contains the build stage, which consists in the compilation of the geonetworking library and then the compilation of the ITS router, the test stage which performs the unit tests, implemented with JUnit, and the deployment stage, which deploys the MQTT service and the ITS. Note that this pipeline is only used for gateway edges, since it does not deploy the vehicle service. This pipeline also contains a downstream that triggers the pipeline of the geonetworking library.

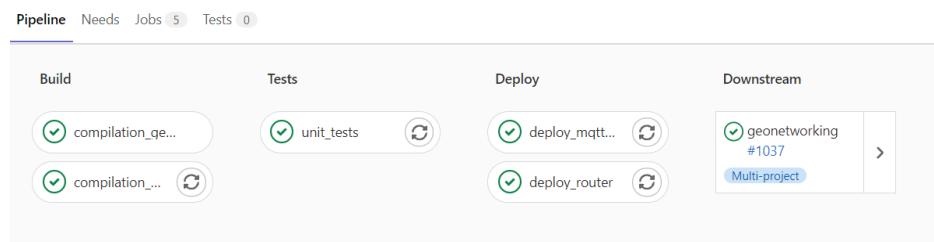


Figure 4.9: Pipeline of the ITS Router that contains the build, test and deployment stages

4.6 SUMMARY

We successfully implemented the features of the architecture proposed in chapter 3. The current platform allows to display vehicle and sensor data in real-time in a dashboard, ART, allows to generate alerts and notify the drivers of nearby vehicles, and also allows to replay incidents.

In this chapter, we started by performing an in-depth analysis of the ITS modules, followed by an analysis of the MQTT instances and the bridge. We also explained why using the MQTT bridge is a better approach than only using the MQTT edges. The first approach allows an easier management of the subscriptions when the number of RSUs increases.

After that, we explained the implementation of the City Manager App, as well as performed an in-depth analysis of the complex Mapping Service. Note that the Mapping Service was implemented using the strategy design pattern, which makes it easier to extend this module, if for example, in the future we want to add a new way of choosing the best candidate RSUs.

We also explained the need for a GUI that allowed the driver to be notified when an alert was sent to the vehicle, and also explained how the Notification App fits that purpose.

Finally, we explained the development, deployment and testing methodologies that were used in the implementation of this infrastructure, which allowed to mitigate the number of bugs in the code, which greatly improved the project efficiency.

Aveiro in Real-Time

This work considers the visualization of the mobility data available in the "living lab" in the city of Aveiro. The City Manager web app (CM) has been developed in the framework of this dissertation, and is composed of two services: the Aveiro in Real-Time service (ART) and the Incident Replayer service (IR).

This chapter describes the context and implementation of the Aveiro in Real-Time App. We will start by analysing the existing testbed in Aveiro, and then the GUI of the app. In the end, we will explain the architecture of this app and how it integrates with the architecture of the whole system.

5.1 STATE OF THE CITY IN REAL-TIME

The Aveiro Living Lab integrates data from people, through their mobile phones, sensors and vehicles. The data collected by the living lab infrastructure can be accessed through a web app ¹ that exposes multiple features to the user. This web app allows a user to visualize the position of all RSUs that are installed in Aveiro in a map, and also allows the user to fetch the sensor and vehicular data gathered in the past. Although it is possible to access the historic information, this task is not performed by the ART, instead it is performed by the IR, as we will explain in the next chapter. The ART displays the data received from vehicles in a clear and concise way, and allows users to have an overview of the state of the city of Aveiro in real-time.

5.2 DASHBOARD

Figure 5.1 shows the ART dashboard. The dashboard contains three main sections: the left side menu, the map and the entity page.

¹<https://aveiro-open-lab.pt/>

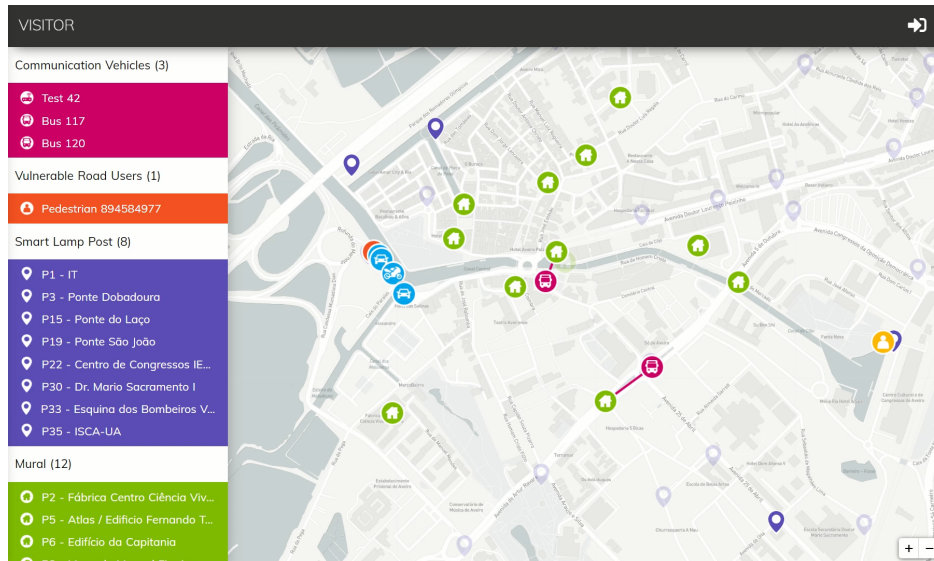


Figure 5.1: Aveiro in Real-Time - Dashboard

5.2.1 Side Menu

The ART side menu displays the list of entities that are currently active in the map. An active entity is an entity that is transmitting or receiving information. This menu is divided into 4 main categories: Communication Vehicles (CV), Vulnerable Road Users (VRU), Smart Lamp Posts (SLP) and Murals. Figure 5.2 shows that, at the moment of capture, there were 4 CV (1 test device and 3 buses), 1 VRU (1 pedestrian), 8 active SLP and 12 active murals. Note that only the active RSUs appear in the side menu, so there may exist more.

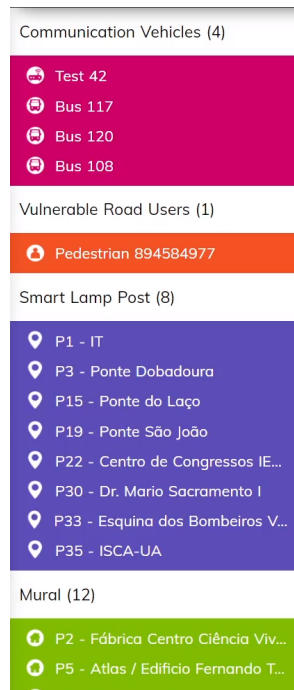


Figure 5.2: ART - Side Menu

5.2.2 The map

The ART map displays the vehicles as markers. Each source type is represented by a different color, for example, radar vehicles are presented by blue markers. Each entity type is represented by a different icon, for example, buses are represented by a bus icon. In figure 5.3 it is possible to observe that, at the moment of capture, there were 2 buses, 5 radar vehicles and 1 pedestrian. We will provide a more in-depth analysis, in section 5.3, of all the different color codes and icons that the ART supports. Note that the RSUs that appear transparent in the map are inactive, which means that they are not receiving information from vehicles.

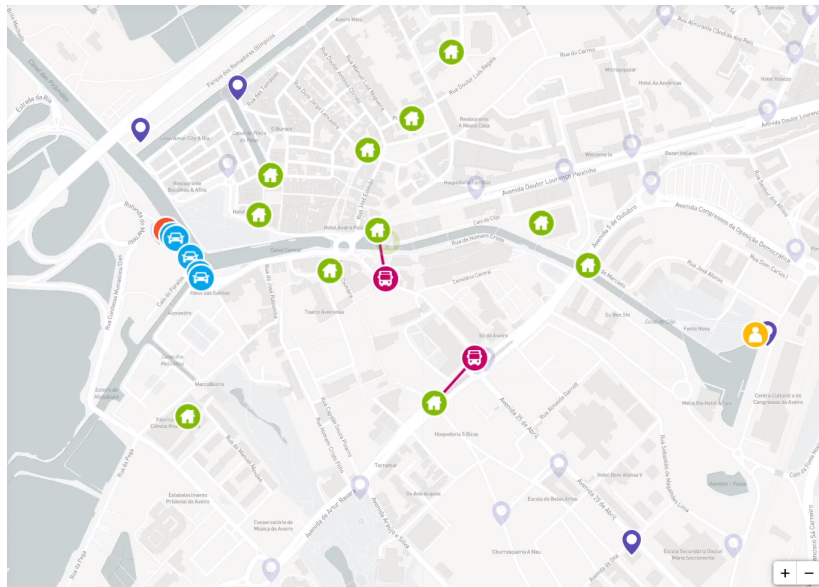


Figure 5.3: ART - Map

5.2.3 The entity page

The entity page is displayed when a user clicks on a marker. All the information about that entity will be displayed in the interface. Figure 5.4 shows the entity page of Bus 120. It is possible to observe information, such as the speed and the heading of the bus, the RSSI of the communication, the RSU that is receiving this information, the timestamp of the last received message and also the delays in the infrastructure, i.e., the time elapsed since the message was received by the RSU until it is displayed in the front-end. A more in-depth analysis of these fields will be provided in the next sections.

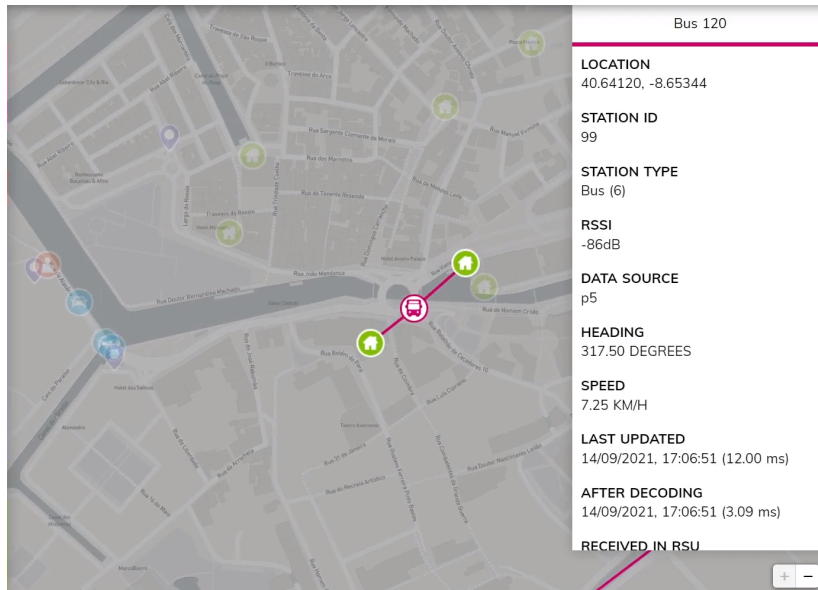


Figure 5.4: ART - Entity Page

5.3 ENTITIES

As previously mentioned, the ART dashboard displays 4 different source types. These types are color coded, which means that, for example, VRUs are displayed as orange icons on the map and are also displayed as orange items in the left menu. We found out, based on the feedback of the designers at Bosch as well as some technical and non-technical users, that this approach is the most intuitive for the end user to understand how to interpret information in the ART.

5.3.1 Communication Vehicles

Communication vehicles ■ are vehicles that transmit CAMs that are received by at least one of the RSU distributed throughout the city. These communication vehicles can be of different types. These types include the station types specified by ETSI ² along with some extra types that were added to the project to address specific use cases, like the Moliceiro type. For example, 5.1 displays three CVs: two buses, of station type 6 (specified by ETSI), and one test on board unit, of station type 18 (non standard type). The main difference between normal vehicles and CVs is that CVs support ITS technology and transmit CAMs periodically, which allow to have access to a vast array of information, while the information about other vehicles can only be extracted by sensors, which collect much less information than CVs.

Table 5.1 contains the list of currently supported types of CVs. Note that ETSI defines more types, but we chose to add support to these because they are the most common types of vehicle, although it is possible to easily add support for more types. Each time a vehicle contains an unsupported station type, the ART will display the Unknown type.

²https://www.etsi.org/deliver/etsi_ts/102800_102899/10289402/01.02.01_60/ts_10289402v010201p.pdf

The pink lines drawn in figure 5.5, represent an active connection between the vehicle and the respective RSU, e.g., it is possible to observe that the passenger car was, at the moment of capture, sending CAMs to two different roadside units.

We will now provide a description of the supported station types. Note that some types may be omitted because they are trivial to understand.

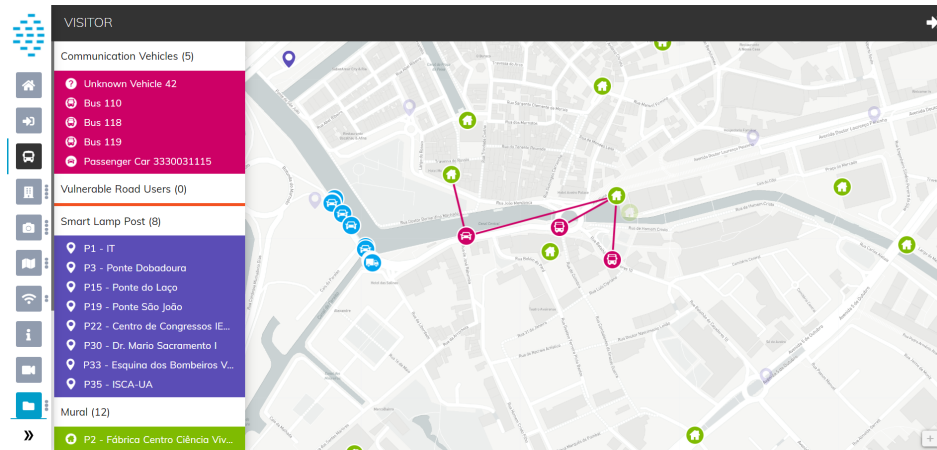


Figure 5.5: Communication Vehicles with lines drawn



















Name	Station Type	Selected Icon	Unselected Icon
Unknown	0		
Motorcycle	4		
Passenger Car	5		
Bus	6		
Light Truck	7		
Heavy Truck	8		
Ambulance	10		
Moliceiro	17		
Test	18		

Table 5.1: Communication Vehicle types

Unknown

Unknown devices, with station type 0, are devices whose type is unknown. The ART will display icons of this type when a vehicle transmits CAMs with an unsupported station type, i.e., a station type that is not listed in 5.1.

Passenger Car

Passenger cars, with station type 5, as the name suggests, are light vehicles that allow the transportation of passengers, that support ITS technology. This type of vehicles frequently appears in Aveiro, but is less frequent than the buses. Some vehicles support ITS technology, but it is possible to make a non-ITS vehicle support ITS technology by placing an OBUs inside of the vehicle. During testing, it was quite often for us to place our OBUs inside of a vehicle, since we had control over the information that was sent in the CAMs, i.e., we could easily change the station type of the vehicle from 5 to 10, and make that vehicle appear as an emergency vehicle.



Figure 5.6: Passenger Car (5) displayed in the dashboard

Bus

Buses, with station type 6, are road vehicle that can carry multiple passengers. ITS support has been added to some buses in Aveiro, by placing OBUs inside of them. These OBUs, transmit CAMs periodically, which allows their telemetry to be transmitted to the infrastructure. This information is then displayed in the dashboard in real-time and can be visualized by the user.

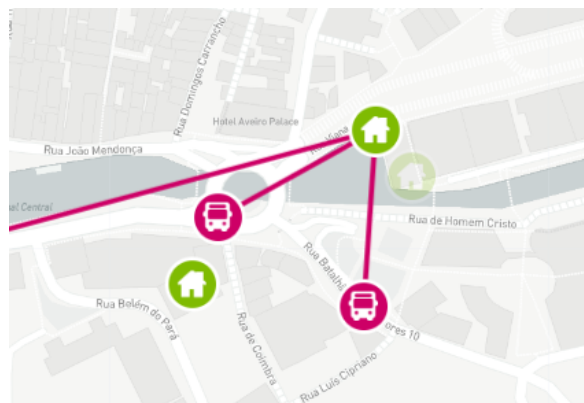


Figure 5.7: Buses (6) displayed in the dashboard

Light Truck

Light truck, with station type 7, commonly are goods vehicle class. A vehicle of this type that supports ITS technology is very rare and has never appeared in the app, but the type is supported by the ART.

Heavy Truck

Heavy truck, with station type 8, is usually used to transport heavy loads. During the development of this dissertation, we only used one OBU of this type, which is located in the installations of Bosch Braga and was sending CAM to a gateway edge via LTE.

Ambulance

Ambulance, with station type 10, is a vehicle that transports patients to treatment facilities and are equipped with medical equipment. According to the ETSI definition, all vehicles with station type 10 must be Special Vehicles, but in the scope of this dissertation, we decided to associate this station type with an ambulance, exclusively.

Moliceiro

Moliceiros, with station type 17, are a special type of boat that sail in the Ria de Aveiro. They were originally used to harvest moliço, but currently they are more commonly used for tourist purposes. This type has been added to the ART because during the development of this dissertation, we placed three different OBUs inside of real moliceiros and tracked their information in real-time.

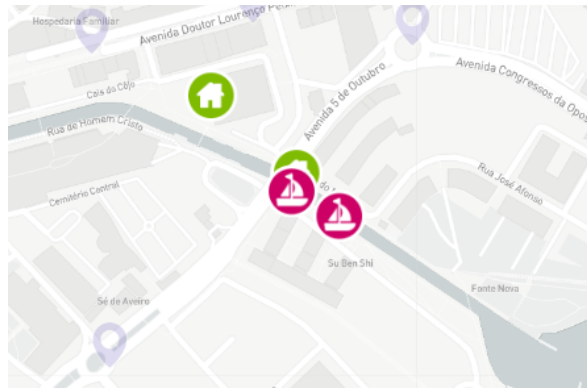


Figure 5.8: Two moliceiros sailing side by side in Ria de Aveiro

Test devices

Test devices, with station type 18, are a special type of OBU, used for testing purposes.

5.3.2 Vulnerable Road Users

Vulnerable Road Users (VRU) ■ are road users that, in the event of a crash, would have little to no protection from crash forces. They are generally considered to include pedestrians, bike riders, motorbike riders and in the case of the CM, we also include some passenger car drivers. The data from the VRU is sent by a mobile app which must be installed in the smartphone of the users. This app will periodically send this data to the CM infrastructure, where it will be both persisted and displayed in real-time in the ART dashboard.

Table 5.2 contains the list of currently supported types of VRUs. Note that ETSI defines more types, but we chose to add support to these because they are the most common types, although it is possible to easily add support for more. Each time a VRU contains an unsupported station type, the ART will display the Unknown type.

Unlike CV, VRUs do not have lines drawn between the VRU and the RSU, since these entities don't communicate using DSRC. Instead the data from the smartphones of the VRU is sent directly to the infrastructure via either wifi or cellular data, thus skipping the edges.

equipment allows to acquire the most amount of information about a vehicle, when compared to the other equipments.

- **Video Camera** - Collects a video feed, normally from a point of interest (park, road,etc.) which is then analysed by an artificial intelligence algorithm that is able to detect and label different objects from this feed, i.e., it can detect how many people, cars, bicycles are present in the video feed.
- **Radar** - Detects three types of vehicles: small, medium and large. It is able to acquire some information about these vehicles, like their speed and heading, without communicating with the vehicles themselves.
- **Lidar** - Detects the number people that are near this sensor.

It is possible to see the list of equipments of a certain RSU by pressing on its marker. In figure 5.3, we can observe the entity page of the SLP P3 - Ponte Dobadoura. This RSU contains ITS-G5, a radar and a camera. Furthermore, it is also possible to observe the cone of vision of the radar, which is divided into 3 distinct areas of confidence. The innermost area has the highest level of confidence,i.e., data received from vehicles that are located in the innermost area is more accurate than data received from vehicles in the outermost area. Note that these areas of confidence are approximations and not exact values. The confidence levels may be affected by multiple factors, like the presence of obstacles inside of these areas.

Table 5.4 contains the list of equipments and their respective icons as they appear in the ART dashboard, while table 5.5 contains the types of supported roadside units and their respective icons. It is possible to notice that figure 5.1 contains 8 active SLPs and 12 Murals.

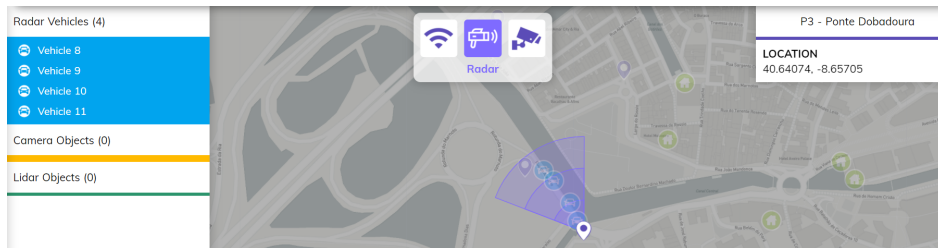


Table 5.3: RSU selected menu





Name	Icon
Cameras	
Lidar	
Radar	
ITS-G5	

Table 5.4: Roadside Unit types





Name	Selected Icon	Unselected Icon
Smart Lamp Post		
Mural		

Table 5.5: Roadside Unit types

5.3.4 Radar Vehicles

Some RSUs have a radar installed on them which gathers information about the vehicles that are in its range. Figure 5.10 shows the RSU P3 selected. The left side menu of the dashboard displays the list of vehicles detected by the radar. The radar is responsible for assigning an ID to each vehicle and by gathering data such as the speed, length and heading of the vehicles. The algorithm used to get this information is out of the scope of this dissertation. Table 5.6 contains the types of radar vehicles that the ART dashboard supports.

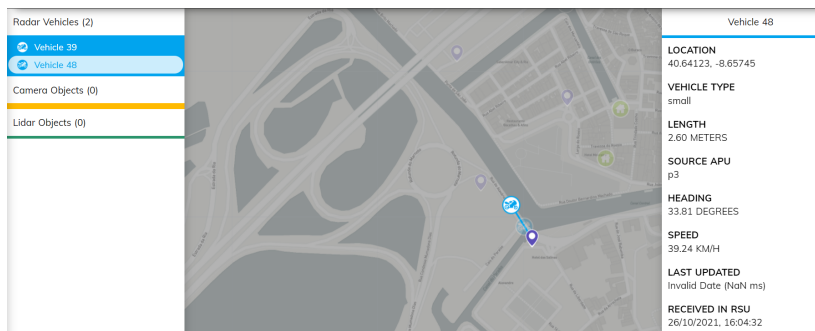


Figure 5.10: Radar Vehicles detected in P3 - Ponte Dubadoura







Name	Selected Icon	Unselected Icon
Large		
Medium		
Small		

Table 5.6: Radar Vehicle types

5.4 DATA FLOWS

In section 4.3.1, we provided a high level overview of the architecture of the CM, which consists in both the real-time and non real-time flows. We also provided an explanation of how the modules communicated with each other. In this section we will analyse the architecture of the ART, which is part of the architecture of the CM but does not contain the IR modules. We will analyse the data flows that allow to populate the ART dashboard.

Figure 5.11 represents the architecture of the ART. This diagram contains the modules responsible for fetching data synchronously and asynchronously.

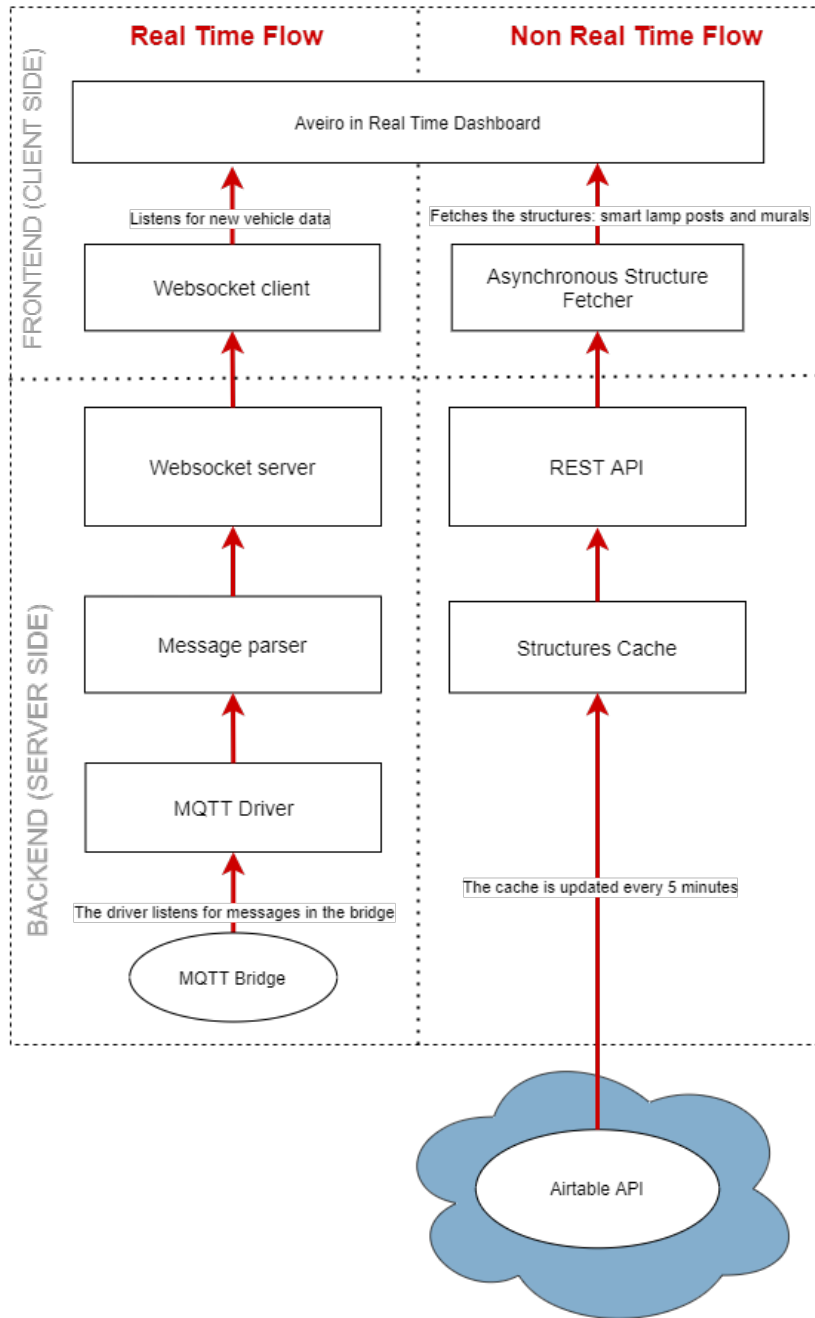


Figure 5.11: Aveiro in Real-Time - Architecture

5.4.1 Dynamic data

Dynamic data, or real-time data, is information that needs to be updated when new data is received, no matter what the frequency is, it must be in the most up-to-date state. Types of dynamic data are the speed and heading of a vehicle which may or may not change depending on whether the vehicle increases its speed or not.

In order to get information from multiple data sources, the ART subscribes to specific topics in the MQTT bridge. Each topic contains messages of one and only one type. There cannot be more than one type of message, for a single topic.

The MQTT driver subscribes to this list of topics and the message parser parses these input messages, according to their input topic, i.e., a different schema is applied to each topic, which allows the message parser to convert the input message, which was published in the bridge, into the output message, which will be sent to the websocket client. The MQTT driver also contains a blacklist of topics and station types, which are ignored when the message is processed, e.g., CAMs from roadside units are received by the MQTT driver but are ignored, since they do not contain relevant information for the ART and their locations are already statically defined in the dashboard. The available topics and the corresponding message types are:

- **apu/cam** - these messages are very similar to CAMs but only contain a subset of fields. Additionally, they also contain the RSSI and the source RSU.
- **cam_decoded** - these are JSON CAMs that are received in their raw format by the bridge. This format has been analysed in 3.3.2.
- **denm_decoded** - these are JSON DENMs that are received in their raw format by the bridge. This format has been analysed in 3.3.2.
- **vam_decoded** - VAMs are standard messages for VRU, which contain information that can be acquired from the smartphones, such as the position, the station type (car, pedestrian, etc.), heading, sizeClass, along with many other attributes. VAMs are received in their raw format by the bridge.
- **jetson/camera/count** - each message contains the label and the number of occurrences of that label, along with more information. Each message contains only one type of label, i.e., one message will contain the count of detected people and another message will contain the count of detected cars. These messages are published only when an object of a certain label type is detected, otherwise, no message will be published in the bridge.
- **jetson/lidar/count** - similar to the camera count messages, the lidar count message contains the count of only one label type, but this type of message is published periodically and not only when an object is detected.
- **jetson/radar** - radar messages contain the information that the radar is able to acquire about the vehicles. This message contains the class (small, medium or large), the position, the speed, the heading and the length of the vehicle as well as the timestamp of reception.

5.4.2 Static data

Some information does not need to be updated in real-time, like the position of the RSUs, because this information is likely not to change over time, since RSUs contain complex equipment and are installed in fixed points throughout the city. Because of this, some data is fetched and updated periodically, which means that temporary inconsistency in the data is tolerated in these cases, where outdated data is displayed in the ART dashboard. For this reason, it is very important to determine which types of information do not need to be updated in real-time.

We have decided to use [Airtable](https://www.airtable.com)³ to store the static information, because it allows to easily change this information by using a GUI. This gives the flexibility to quickly change the data, but it also allows to change the format of the table, i.e., it allows to quickly add more columns, since it does not enforce a schema validation on the data. Another important aspect and maybe the most important, is that [Airtable](https://www.airtable.com) allows to make the data on each table accessible via an API, i.e., in order for us to access the table data, we just need to perform a GET request to this API and the data will be returned in the response body.

Figure 5.11 shows the entities involved in the non real-time data flow. The CM backend periodically fetches static information from the [Airtable](https://www.airtable.com) API and stores it in cache. The client side can then access this information, when the page is loaded, by performing a GET request to the defined endpoint in the API exposed by the backend. As previously mentioned, the client side may eventually fetch outdated information from the backend, but consistency will eventually be reached when the backend performs its next fetch from the [Airtable](https://www.airtable.com) API.

The static information can be divided into the following categories:

- **Event Types** - list of ETSI defined values for the `causeCode` and `subCauseCode` fields of the DENM, i.e., contains a map between the value of the `causeCode` and their full description, e.g., `causeCode` of value 2, represents an accident. The full description is provided to make it easier for the user to understand its meaning.
- **Station Types** - list of ETSI defined values for the `stationType` field of the CAM, i.e., maps the value of `stationType` to its full name. These values are passed as dynamic data, since this allows us to add custom `stationTypes`, like the `moliceiro` type. This way no code changes are necessary for the ART to support a new custom `stationType`.
- **Structures** - full list of existing roadside units. It contains the type (Mural or SLP), description, position, supported equipments and the active status of each RSU.
- **Cameras** - information about the existing cameras. Contains the RSUs where the cameras are installed, the heading, the spread and length of the cone of vision.
- **Radars** - information about the existing radars. Contains the RSUs where the radars are installed, the heading, the spread and length of the cone of vision.
- **Lidar** - information about the existing lidars. Contains the RSUs where the lidars are installed, the heading, the spread and length of the cone of vision. Note that lidars have a 360° cone of vision, but their vision might be impaired by certain obstacles.

5.5 SENDING ALERTS TO VEHICLES

As mentioned in chapter 3, one of the main requirements of this system was the ability to support I2V communication which would allow drivers to be notified of possible dangers on the road. We then explained that the users could send alerts to the vehicles through the CM. In this section we will analyse the GUI that the user interacts with in order to generate DENMs that will later reach the vehicles. We will also present and explain the GUI of the notification app that runs in the OBU and displays the alert to the driver of the vehicle through a web app.

³<https://www.airtable.com>

5.5.1 Generating DENMs

Figure 5.12 shows the ART dashboard that includes a right side menu that allows users to generate DENMs. The users must first specify the alert's cause code, sub cause code and epicenter. After this, they can generate the alert by pressing the **Generate Alert** button.

Figure 5.12 shows the state of the ART dashboard moments after this button has been pressed. As we can observe, the user generated a DENM with cause code 2 and sub cause code 92, which has been detected by the ART. A notification message has been generated by the platform and displayed in the top right corner showing this same information. Furthermore, a red circle has appeared in the map, in the coordinates that were specified by the user and represents the AOE of the generated DENM. In order to prevent the abuse of the system, we provide this feature only to administrators, i.e., to use this feature, users must be logged in an administrator account.

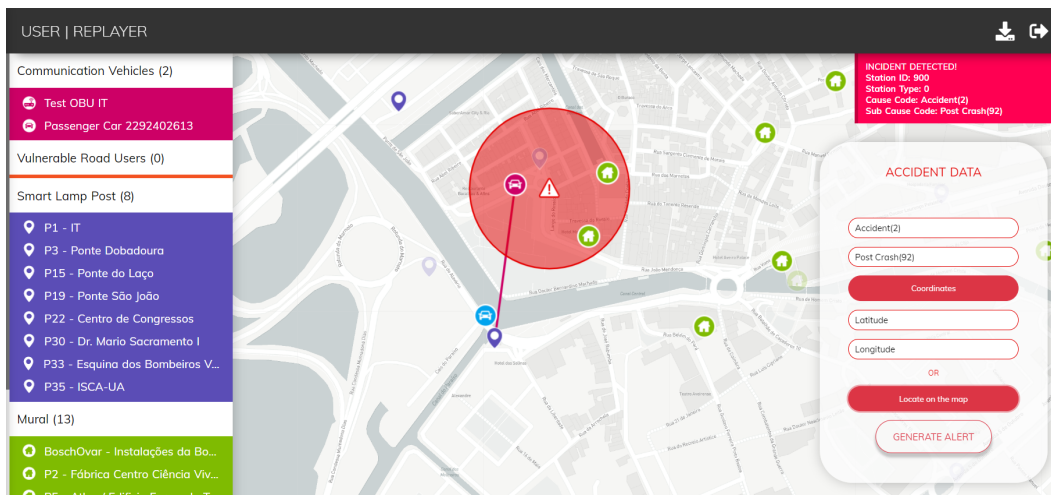


Figure 5.12: DENM generated from the Aveiro in Real-Time dashboard

When a DENM is generated, it will traverse the whole infrastructure until it reaches the vehicle's OBU. This DENM will then be decoded, converted to the JSON format and published in the local MQTT instance, where it will be fetched by the Notification Server and then sent to the client side via websocket. Finally, the alert is displayed to the user in the GUI, along with its attributes, such as the cause code and sub cause code.

Figure 5.13 shows the GUI of the notification app, that displays the information from the DENM generated in the previous example. The driver is able to see that an accident has been detected near him and might adopt a different driving behavior. This alert contains cause code 2, sub cause code 92 and was generated by the station with id 900, which is the ID of the CM.

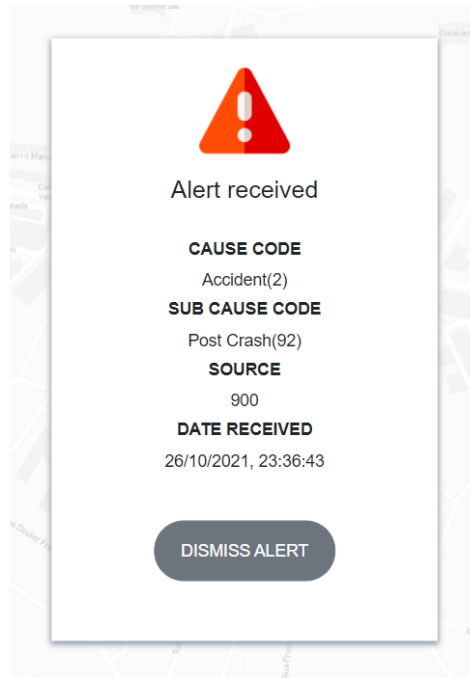


Figure 5.13: Alert displayed in the notification app

5.6 RECEIVING DATA USING V2V COMMUNICATION AND LTE/5G

As we have previously mentioned, it is possible to receive information in the infrastructure, from a vehicle that does not have an internet connection. This is possible only if that vehicle is able to communicate with another vehicle that has an internet connection and that vehicle serves as a proxy. We have tested this scenario in our live testbed.

Figure 5.14 shows two buses, bus 118 and bus 114 (selected in the image). Bus 118 has LTE/5G connection and has V2V connectivity with bus 114. Bus 114, on the other hand, does not have internet connectivity, but also has V2V connectivity with bus 118.

We can observe that bus 114 sends its vehicle data to the infrastructure through bus 118, which works as a "moving RSU". In figure 5.14 it is also possible to observe that the data source of bus 114 is obu50, which is a field that commonly displays the ID of the RSU, but in this case it displays the ID of the OBU of bus 118.

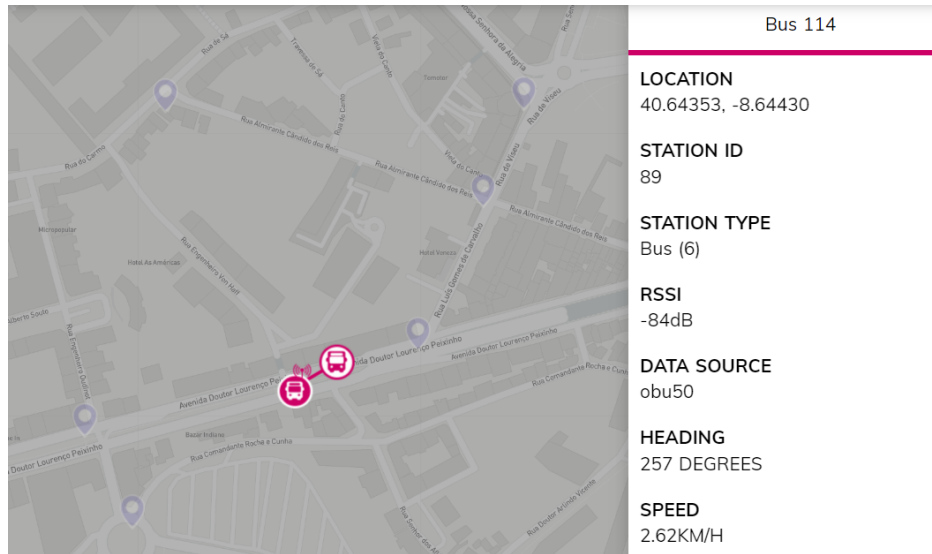


Figure 5.14: Bus with 5G sends vehicle data of bus with no internet connection

5.7 SUMMARY

In this chapter we started by providing the context about the CM and the living lab. We then explained the difference between the ART and the IR, and finally presented the main features of the ART.

In section 5.2 we presented the different sections of the ART dashboard: the side menu, that contains a list of the active entities, grouped by their types, the map, which displays the location of these entities and contains color coded markers for all of them and the entity page, which is activated when a user presses an entity and displays detailed information about the selected entity.

In section 5.3 we listed all the currently supported entity types: CVs, VRUs, RSUs and radar vehicles. Note that camera objects and lidar objects are not considered different entities, since they provide a very limited amount of data compared to the other entities.

In section 5.4 we explained how the ART supports both real-time and non real-time flows of data and the interaction between their inner modules. We analysed the different types of messages that are published in the MQTT bridge and are later used to populate the ART dashboard. We also explained the advantages of storing the static information in cache and listed all the different types of static information that is stored by the CM backend: event types, station types, structures, cameras, lidars and radars.

In section 5.5 we presented the GUI that allows users to notify vehicle drivers, by generating DENMs and sending them to the OBUs. We also presented the GUI of the notification app that allows the drivers to be notified and to have access to the type of alert that was generated.

Incident Replayer

This chapter describes the context and implementation of the Incident Replayer. We will start by explaining the old replay system that was used for testing and how the Incident Replayer was inspired by it. We will then present the GUI of this app and later explain the data flows and how this app integrates in the architecture of the whole system.

6.1 REPLAYING DATA

In chapter 3 we defined the requirements of the system and explained how the proposed architecture addresses them. In chapter 5 we presented the ART platform, which is one of the results of the implementation of this architecture. The ART addresses the real-time data requirements, by displaying sensor and vehicle data in real-time, but it does not allow users to have access to any kind of historic information. The received data is displayed in a dashboard where users can observe the state of the city in real-time. This means that the data is displayed, with minor latencies, as soon as it is received by RSUs, without requiring the user to refresh the page. This creates a very dynamic vision of the city.

During the development of the ART platform, we created a replay system that allowed us to repopulate the ART dashboard for testing, based on previously received messages in the MQTT. This was done because sometimes we needed to test if radar vehicles were being correctly displayed, but at that moment there were no radar vehicles physically detected in the city, thus the need to create a more efficient way to test the dashboard.

This system consisted in two different components: the message sniffer and the message generator.

6.1.1 Message sniffer

The message sniffer consists in a script that subscribes to all topics of a MQTT instance and stores each message received and the respective topic, in each line of an output file. The messages and the respective topics are stored in the ASCII format.

Code 6 shows a portion of the content of a file generated by the message sniffer. Note that the content of the messages is too big, so some information was omitted. It is possible to observe that each row corresponds to a message that was published in the MQTT instance. Each row contains the topic where the message was published and the content of the message, e.g., we can observe that a vehicle was moving at 11.9 m/s and was detected by the radar located at P3 - Ponte Dobadoura, which lead to a message being published in topic Jetson/Radar of the MQTT located in the edge and subsequently published in topic p3/Jetson/Radar of the MQTT bridge, where it was read by the message sniffer and stored in this file. Lastly, it is also possible to observe that the topic format contains the ID of the source as a prefix, i.e., a message that was published in topic p3/Jetson/Radar was received in the RSU with ID p3. Notice that some messages were received from a source with ID obu50. This is the case where messages were received by a "moving RSU" (obu50 in this case), as seen in 5.6.

Note that, while this approach works for testing purposes, it is not viable to use it to store historic information because of the amount of disk storage that is required. The amount of data stored in the file depends on the amount of messages being published in the MQTT bridge at any given moment, but we were able to observe that, on average, a 30 second sample leads to 13.7MB of storage used.

```

1 obu50/cam_decoded {"rssi": -65535, "timestamp": 1632763187.660123, "others": {...}}
2 obu50/cam_decoded {"rssi": -65535, "timestamp": 1632763187.660123, "others": {...}}
3 p1/Jetson/Camara/Objects/People {"detectedPerson": "True", "listOfPeople": [...]}
4 p3/Jetson/Radar {"long": -8.65730527, "lat": 40.64107383, "speed": 11.9, ...}
5 p3/jetson/radar {"longitude": -8.6573052, "latitude": 40.641073, "speed": 11, ...}
6 p3/Jetson/Radar {"long": -8.65765386, "lat": 40.641468, "speed": 10.70000, ...}
7 p3/Jetson/Radar_new {"long": -8.65765386, "lat": 40.641468, "speed": 10.7000, ...}

```

Code 6: Incident Replayer - Message sniffer file format

6.1.2 Message generator

The message generator consists in a script that reads from an input file and publishes in the MQTT instance, the messages in their respective topic, effectively replaying the messages and populating the ART dashboard.

Figure 6.1 shows the flow of data from the moment when the message is published in the MQTT bridge to the moment when the data arrives in the City Manager. We can observe that, as seen previously, the data is published in the bridge, and it is immediately consumed by the City Manager, which allows to display that information in real-time, but there is also another data flow. As we have seen before, the message sniffer is responsible for reading the data from the bridge and storing it in a file. The message generator will, when triggered by the user, read all the messages stored in the file and publish them in the input topic, but in a different MQTT instance, in this case we used the MQTT Bridge for Replays which is similar to the main MQTT bridge but runs in a different port. This way we do not interfere with the production data flow.

This system was initially intended to be used for testing purposes only, but we quickly realised its potential. It was possible to replay historic data but we wanted to extend this solution in order for it to be used not only by developers but also by non technical users.

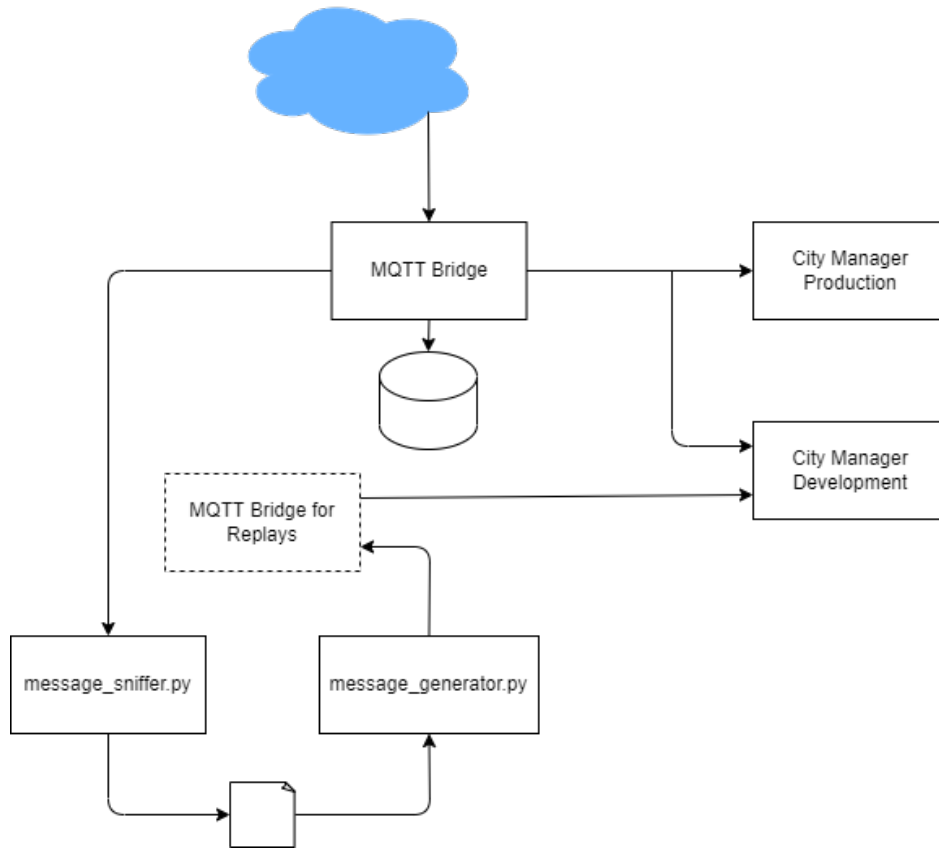


Figure 6.1: Message sniffer and generator flow

6.2 REQUIREMENTS

We wanted to create a way for users to be able to replay the state of the city, in certain points in time. Since this appeared to be a complex system and we did not find a similar solution, we had to develop it from the start, so it was important to define the requirements first.

6.2.1 Allow users to replay incidents

As we mentioned previously, the main goal of this platform was to replay historic information, but we wanted to extend this concept and to develop a solution that allows the state of the city to be replayed, i.e., it must be possible for a user to observe all sensor and vehicle data at a given point in time. Since this is a complex solution, we decided to focus on replaying specific incidents, rather than allowing the user to choose a time window to replay, although this could be a future improvement. For this dissertation we decided to replay 3 minutes before and after the incident alert is generated.

The state of the city must then be displayed to the user through, for example, a GUI similar to the ART dashboard, but would have to include the list of incidents and allow the user to select the one that they want to replay.

6.2.2 Allow the user to control the replay

After selecting the incident that they want to replay, the user should be able to perform standard control operations on the replay, such as starting or stopping the replay, skipping to any point in time or changing the replay speed. The user must also be able to keep track of the current time of replay, i.e., the date that is being replayed should be displayed to the user at all times. Finally, the user should be able to interact with all the entities in order to get their detailed information, by pressing the markers, for example.

6.2.3 Store the historic data in an efficient way

The replay system that was previously explained had one big flaw. Storing all the messages and topics in ASCII format in a file is very inefficient and takes up a lot of space in disk. For large replays, the size of the file can increase very rapidly, thus it is necessary to create another solution to this problem.

6.2.4 Access the historic data in an efficient way

Now that we have stated the need to create a solution to store information in an efficient way, it is also important to consider that the solution must also allow the easy and efficient access of historic data.

The previously implemented replay system contains a very simple solution, which consists in republishing the messages in the MQTT, which on its own appears to be efficient. This is not to say that it is efficient to store messages in the file, but republishing them in the broker again may be a good starting point to find the most efficient solution.

6.3 GRAPHICAL USER INTERFACE

Now that we have defined the requirements, we will present the functional results that were achieved by the development of this platform.

Figure 6.2 shows the IR GUI, which is divided into three main parts: the map, the incident menu and the replay menu. Since the IR map is very similar to the ART map, which was presented in 5.2.2, we will not be analysing it. Instead we will focus on analysing the other two parts.

It is possible to observe that, at the time of capture, the user was replaying the **V2V video demo 1** replay, that involved a passenger car and an emergency vehicle. It is also possible to observe that this figure shows that the replay was paused on the date 15/10/2021 15:10:22 and had the x4 speed selected.

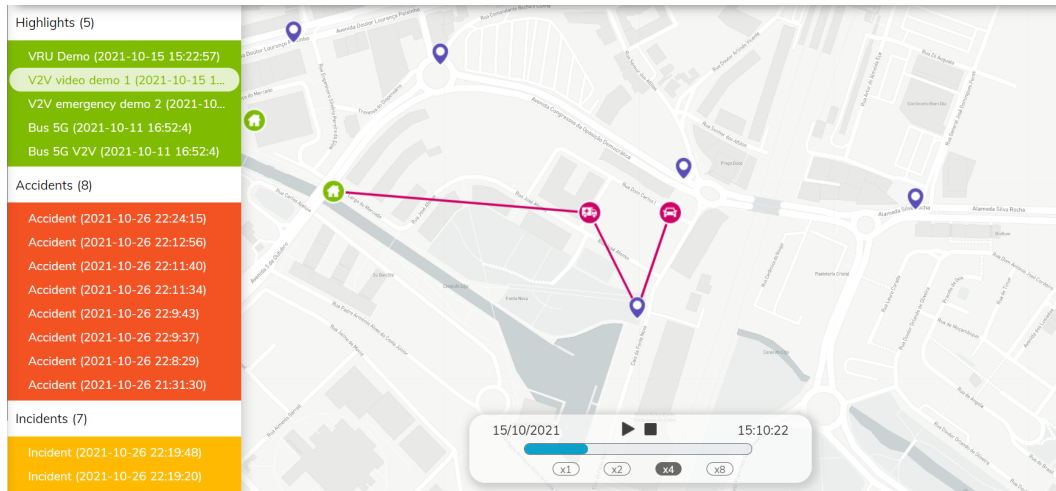


Figure 6.2: Replay of the approach of an emergency vehicle on 15-10-2021 15:10:22

6.3.1 Incident menu

The incident menu displays a list of incidents that were detected by the IR platform. It displays all the highlights and the 15 most recent alerts, which are then divided into two different categories: the accidents and the incidents. In sum, the interface displays the following alert types:

- **Highlights** - Used to bookmark special incidents and display them at the top of the list to make it easier to find and replay them later. They are often used to bookmark special demonstrations, like the ones performed for Aveiro Tech Week ¹. These replays must be specified in the code, but this could be extended in the future by specifying and loading the timestamps of these incidents from the Airtable API.
- **Incidents** - Less severe occurrences. These may be used to alert the driver about potentially dangerous events like road works ahead, obstacles on the road or approaching emergency vehicle.
- **Accidents** - Severe occurrences. May be used to alert the drivers about nearby vehicle crashes.

¹<https://techweek.aveirotechcity.pt/pt>

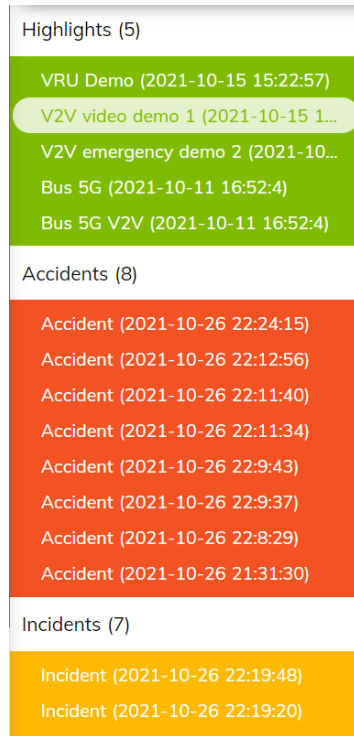


Figure 6.3: IR - Incident Menu

6.3.2 Replay menu

One of the requirements that was presented in section 6.2, was that the IR platform should allow the user to perform standard control operations on the replay, such as starting or stopping the replay, skipping to any point in time, or changing the replay speed. To address this, we decided to implement a replay menu, shown in figure 6.4, that allows the user to perform these operations. In sum, the full list of operations is:

- **Play** - Starts or resumes the replay
- **Pause** - Pauses the replay at a given point in time. If the play button is pressed after, the replay will be resumed in that point.
- **Stop** - Resets replay, i.e., moves the progress bar to the start of the replay.
- **Skipping** - Allows the user to skip to a point in time by pressing te progress bar.
- **Speed** - Allows the user to increase or decrease the speed of the replay. A replay that is played at x2 speed will take half of the time to be reproduced. The IR supports x1, x2, x4 and x8 speeds.

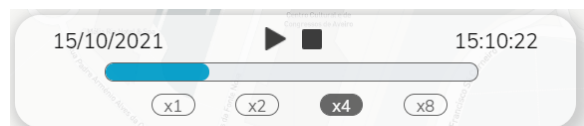


Figure 6.4: IR - Replay menu

6.4 DATA FLOW

In section 4.3.1, we provided a high level overview of the architecture of the CM, which contained both the real-time and non real-time flow of data. Contrary to the ART, the IR only accesses the historic information and does not provide any kind of real-time information.

The IR is composed of two asynchronous data flows:

- **Data acquisition** - The data acquisition part consists of the data sources, which produce the data, the edges, that receive the data and relay it to the core, and the database cluster, where the data is persisted.
- **Data replay** - The data replay part consists in a frontend application, that the user uses to selected and control replays, the CM backend, which exposes an API and is responsible for accessing the databases when an HTTP request is received and the database cluster, where the data is has been stored.

Note that both of these data flows are completely independent of each other, i.e., data can be produced and stored without being accessed, and data can be accessed without being produced. Note that even though this modularity is a good approach, there is still one possible point of failure, the database cluster. If the cluster is offline, then the data can neither be accessed nor produced. This problem could be solved by creating multiple replicas of the databases, which would bring their own problems, such as data duplication and a performance impact, but this would mitigate the risk of bringing the whole system offline.

6.4.1 Storing historic data

Figure 6.5a represents the complete flow of data. This data is produced by the data sources, like pedestrians, vehicle and sensors and is sent to the edges. After this, it is persisted in a cluster of databases. This data is grouped by type and distributed throughout multiple TimescaleDB ² databases, e.g., the CAM data is stored in the CAM database, the DENM data is stored in the DENM database, etc.

Note that we decided to use the previously implemented database cluster, which the author of this dissertation did not create. All of the databases are run in a single machine. While we took this approach, we recognize that there is room for improvement and this solution would greatly benefit from having a distributed database system which would allow a larger number of users to access this information at the same time. Since the focus of this dissertation was not to create a distributed database system, we focused instead on developing a base concept for the replay platform.

6.4.2 Accessing historic data

As we have seen before, the data is persisted in a cluster of databases so that it can be later used.

When a user selects a replay from the incident menu, as seen in 6.3.1, an HTTP GET request is sent to the CM backend which will return the historic data, from 3 minutes before to 3 minutes after the generated incident, to the frontend. Even though we chose this approach,

²<https://www.timescale.com>

we recognize that there are other possible solutions, which cache the database response in the backend and only send the data gradually.

Figure 4.4 demonstrates the interaction between the inner modules of the IR. When the user requests the historic data, the PostgreSQL Driver will fetch the data from the database, which will be parsed by the Historic Data Parser, into the relevant format and finally the data is sent to the client side, where it is replayed for the user.

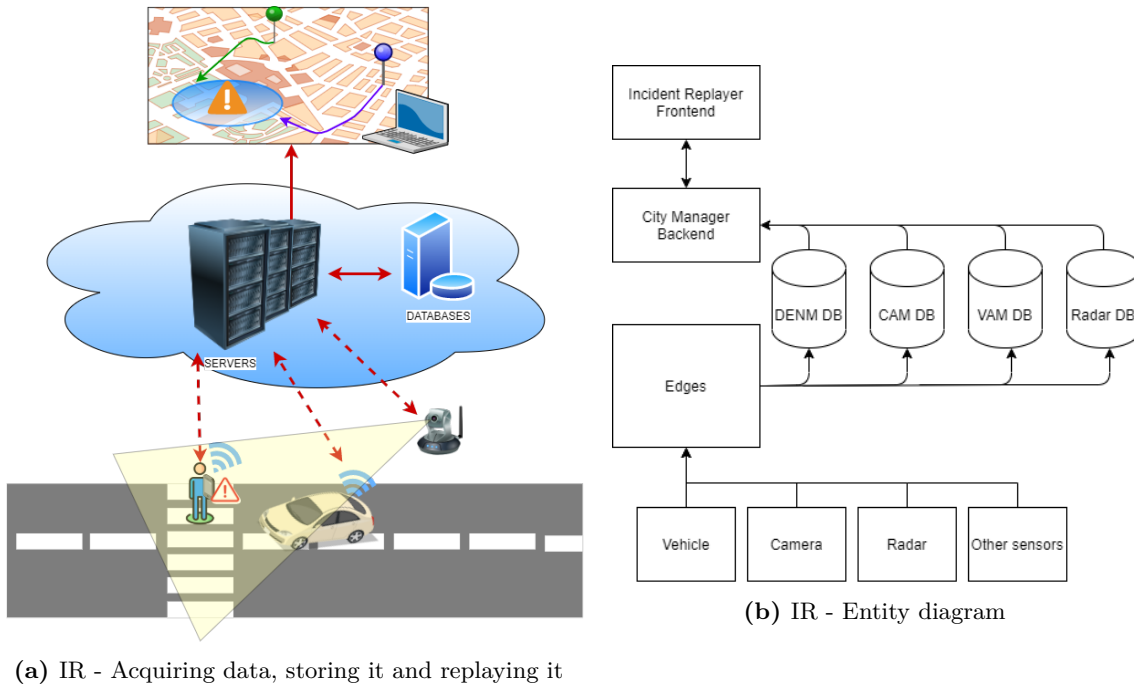


Figure 6.5: Incident Replayer - Data Flow

6.5 SUMMARY

In this chapter we started by presenting how the ART’s replay system served as the starting point to the development of a more complex solution that allowed users to replay the state of the city in a given point in time.

In section 6.3 we presented the GUI of the IR and analysed its parts: the incident menu and the replay menu. The incident menu allows the user to select the replay and the replay menu allows the user to control the replay by performing some operations, such as play, pause, change speed or skipping. In section 6.2, we presented the requirements that the proposed platform must have. The platform must allow users to replay an incident and must allow the user to control this replay. The platform must also have an efficient way to store the data, instead of storing it in a file in ASCII format like in the original replay system and must also have an efficient way of replaying the data. Finally, in section 6.4 we presented the two main data flows of this system: the data acquisition flow and the data replay flow. To acquire data, the edges will receive information from the data sources and persist this data in a cluster of databases TimescaleDB. To replay the data, the CM backend will query these databases and send the data to the client side, where it will be replayed.

Results

In chapters 5 and 6 we presented the functional results of this dissertation, by explaining the main functionalities that were implemented in both the ART and the IR.

In this chapter we will provide the performance results gathered from executing a group of stress tests on both of these platforms. We will start by defining some scenarios and will later provide the performance results of the platform when tested against those scenarios. We will test the infrastructures, including the OBUs, the edges and the core, and later we will focus on performing stress tests of the CM platform.

7.1 USE CASES

As we have previously mentioned, the proposed system is able to perform V2I, which allows vehicles to send their information to the infrastructure, so that it can be consumed by high level application, but it is also able to perform I2V communication, by allowing an operator to send an alert to the drivers, through the infrastructure. For this reason, we have decided to test the infrastructure against some possible scenarios in which both of these types of communication occur.

7.1.1 Scenario 1 - Sending vehicle data to the infrastructure via DSRC

In this first scenario the vehicle periodically generates the vehicle data, in the form of a CAM, and sends it through a RSU edge via DSRC. After this the data is relayed to the infrastructure core where it will be consumed by the CM and displayed in the ART dashboard. Figure 7.1 shows multiple vehicles sending their data to the infrastructure through an RSU, represented by a black antenna.

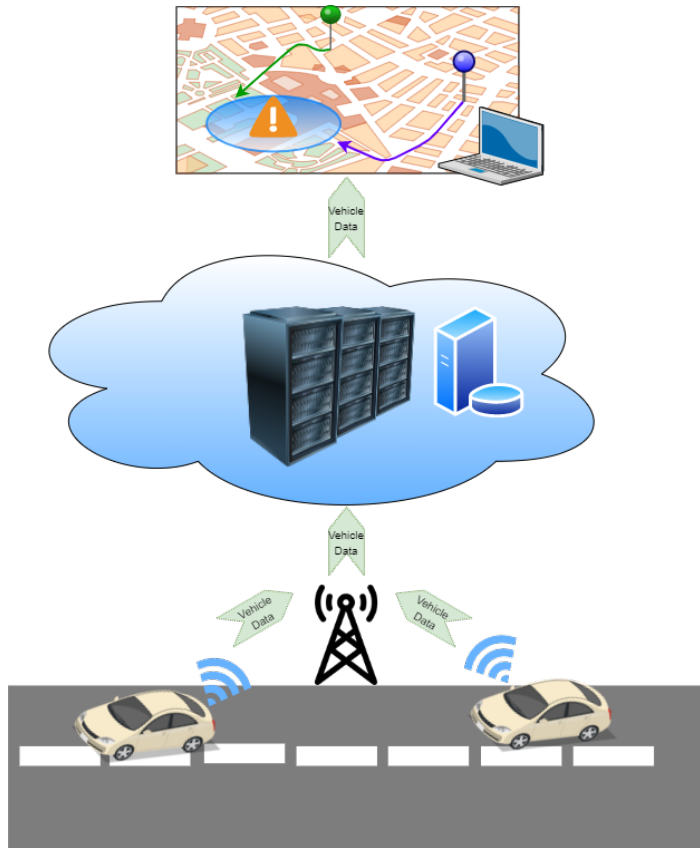


Figure 7.1: V2I - Sending vehicle data to infrastructure

7.1.2 Scenario 2 - Sending alerts to the vehicles

Contrary to the scenario presented in 7.1.1, in this scenario the user generates an alert, which will be sent by the CM to the mapping service. This service will then compute the edges that have the highest probability of being able to reach the vehicles. Note that the generated DENM may not reach the vehicles, so it is up to the Mapping service to determine the edges that have more likely to have a connection with the target vehicles. After the DENM reaches the respective edges, it will be broadcasted and received by the vehicles, which will display its information in the GUI of the notification app, as seen in figure 7.2.

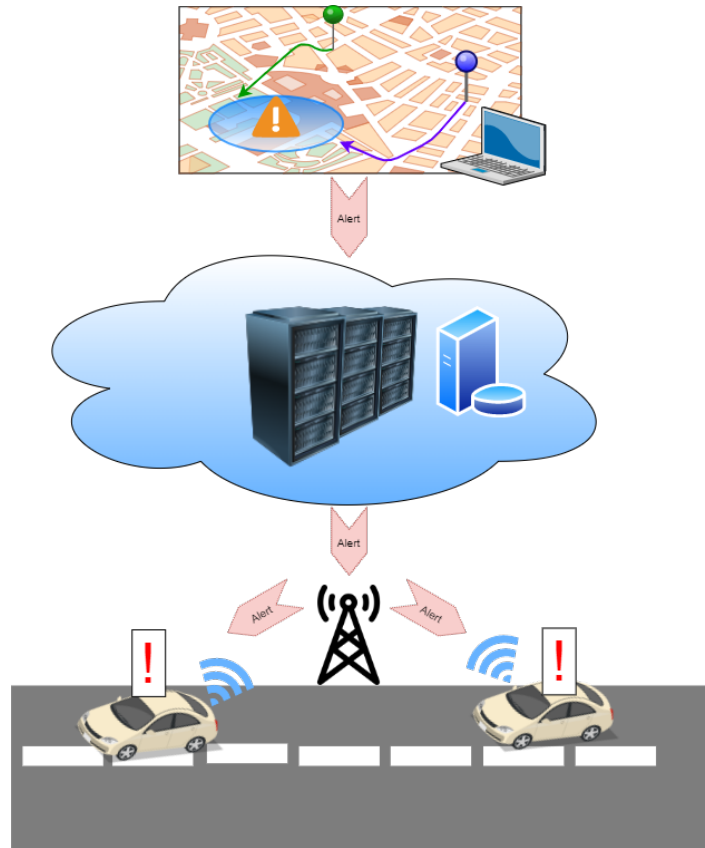


Figure 7.2: I2V - Sending alerts to the vehicles

7.2 TESTING ENVIRONMENT

To test the concepts described earlier, we created a laboratory testbed. This testbed is composed by one RSU and one OBU. Even though the proposed system allows DENMs to be sent through multiple RSUs, we decided to use only one RSU to make it easier to test our solution.

7.2.1 V2I communication

Figure 7.3a shows the setup that was used to test scenario 1. This figure shows the flow of data from the vehicle service, where the CAMs are generated, until it reaches the CM, where it will be displayed to the user by the ART dashboard. We also created 6 checkpoints along this flow, where we measured the timestamp and compared it to the other checkpoints. This allowed us to calculate the latency, i.e., the time elapsed between one checkpoint and another. In order to better explain the time of measurement in each checkpoint, we will proceed to provide a description about each one.

- **Point 1** - Timestamp is measured after the CAM is generated and before it's sent to the ITS router.
- **Point 2** - Measured after the CAM is encoded and before it is sent to the U2E where it will be broadcasted.

- **Point 3** - Measured as soon as the eth2udp receives the packet in the wireless interface of the RSU and before its sent to the router.
- **Point 4** - Measured after being decoded by the ITS router and before being converted to the JSON format.
- **Point 5** - Measured before publishing the JSON CAM in the MQTT edge.
- **Point 6** - Measured when the CM receives the CAM from the MQTT bridge.

7.2.2 I2V communication

Figure 7.3b shows the setup that was used to test scenario 2. This figure shows the I2V data flow, that is triggered when the user generates an alert in the ART dashboard. This test case covers the complete V2X data flow from the CM to the notification app in the OBU. Note that a DENM may not reach its intended target vehicle, as we have previously explained. To eliminate the risk of this happening, we configured the mapping service to send the DENM to the testbed RSU at all times.

For this test case we performed measurements in 8 checkpoints along the flow, which we will proceed to explain.

- **Point 1** - Measured after the CM generates the DENM and before it sends it to the mapping service.
- **Point 2** - Measured after the mapping service receives the DENM and computes the best RSU candidates. This measurement is taken before the DENM is published back into the MQTT bridge.
- **Point 3** - Measured after the DENM is read from the MQTT edge and before it is sent to the ITS router.
- **Point 4** - Measured after the DENM is received from the MQTT service and before it is decoded.
- **Point 5** - Measured as soon as the eth2udp receives the packet in the wireless interface of the RSU and before its sent to the router.
- **Point 6** - Measured after being decoded by the ITS router and before being converted to the JSON format.
- **Point 7** - Measured before publishing the JSON DENM in the MQTT edge.
- **Point 8** - Timestamp is measured when the notification app receives a DENM. This happens after the DENMs is published in the MQTT obu and then read by the notification app's subscriber.

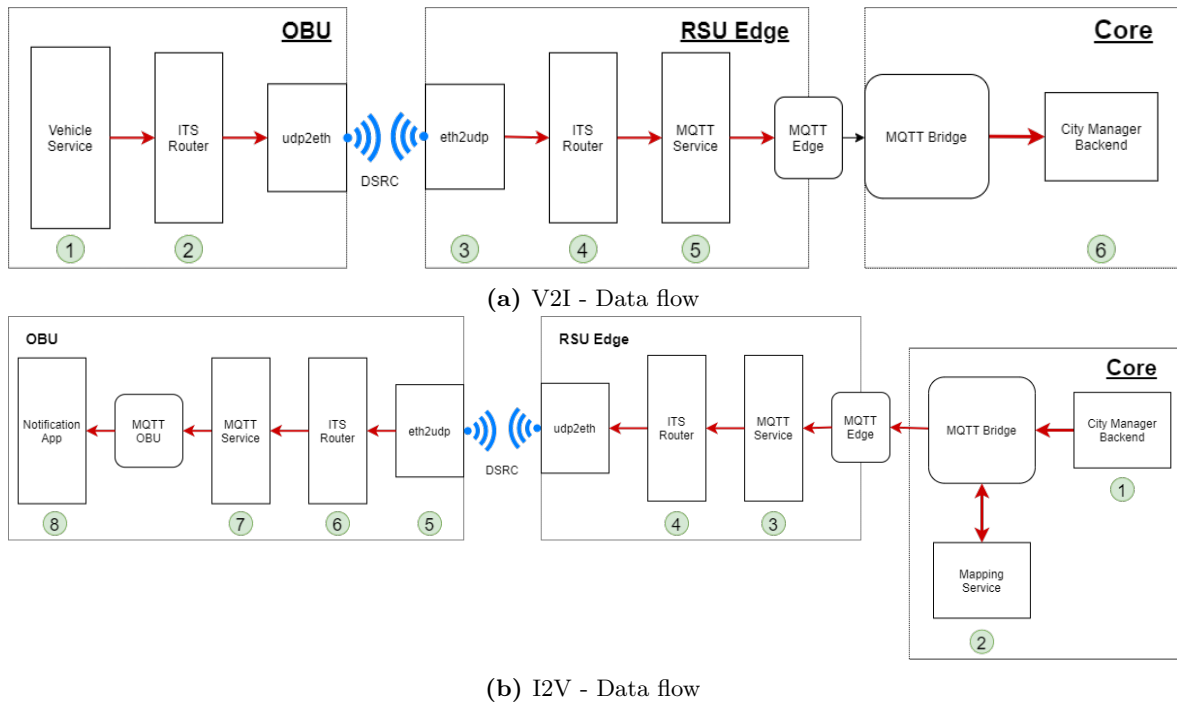


Figure 7.3: Both V2I and I2V data flows

7.3 EQUIPMENT

Both the RSU and the OBU used run in two boards with similar characteristics. Their characteristics can be observed in table 7.1. These boards were prepared in advance and their remote access was made available.

The core services were split across two different virtual machines, with similar characteristics, running in the IT data center. These characteristics can be seen in table 7.2.

RSU and OBU board specifications	
CPU	AMD G series gx-412T 1GHz quad Jaguar core with 64bits
RAM	4GB
Storage	30GB SSD
OS	Debian 10
Linux kernel	5.7.10

Table 7.1: RSU/OBU - Hardware specifications

Core virtual machine specification	
CPU	Intel(R) Xeon(R) Silver 4215 CPU @ 2.50GHz
RAM	16GB
Storage	240GB HDD
OS	Ubuntu 18.04.4

Table 7.2: Core - Hardware specifications

7.4 RESULTS

In this section we will present the performance results obtained by running end-to-end tests on the infrastructure, for each scenario described above. We will also present the performance results obtained from running stress tests on the most time consuming API endpoints of the CM backend. These two types of tests will allow to have a good idea of the overall performance of the infrastructure, but will also provide some details about the performance of the CM as a standalone unit.

7.4.1 Scenario 1 - Sending vehicle data to the infrastructure via DSRC

As mentioned previously, the vehicle data is generated periodically and sent to the infrastructure. In order to simulate that process, we configured the OBU to generate a total of 500 CAMs with a frequency of 10Hz. Every time a CAM was sent, we captured the timestamp in each checkpoint of the flow. Table 7.3 shows the results that were obtained from this process.

First, we started by measuring the timestamps in each point. After that, we computed the difference between the timestamp of one point and the timestamp of the previous point, which corresponds to the elapsed time between one point and the other, or also known as the latency. Note that because of this approach, there is no latency for point 1 because there is no point that comes before it.

The average latency corresponds to the average value between the 500 latencies, that were previously calculated, for each point. The average latency between point 3 and point 2 is 25.86 ms, which means that the message took on average 25.86 ms to go from point 2 to point 3.

After this, we calculated the average cumulative latency, by computing the running sum of the latencies, e.g., point 5 has an average cumulative latency of 38.46ms, which means that the message took that amount of time to go from point 1 to point 5.

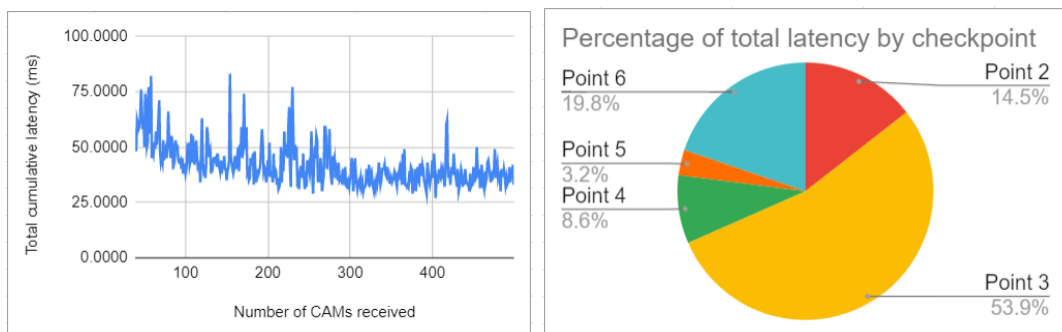
Finally, in order to have a better understanding of the performance of the infrastructure and to make it easier to identify bottlenecks, we computed the percentage of the total latency by point. This was achieved by simply dividing the average latency of each point by the average cumulative latency in point 6, i.e., the average total amount of time that it takes for the message to go from the vehicle service to the CM. These results were then inserted in a pie chart, like the one seen in figure 7.4b .

From these results, it is possible to conclude that, sending the CAM from the OBU, via DSRC, and receiving it on the RSU is the longest step in the process, takes 25.86 ms and corresponds to 53.93% of the total latency.

We also decided to plot the total cumulative latency, as seen in 7.4a, which allows to have a better understanding of the performance of the infrastructure over time. As we can see, the performance of the whole system stays constant over time, and we were able to find that the total latency of the infrastructure reached a minimum of 27.00 ms and a maximum of 83.00 ms.

Point	Average latency	Average cumulative latency (ms)	Percentage of total latency (%)
2	6.95	6.95	14.50
3	25.86	32.81	53.93
4	4.11	36.92	8.57
5	1.53	38.46	3.20
6	9.49	47.95	19.80

Table 7.3: Scenario 1 - Results



(a) Scenario 1 - Relation between total cumulative latency and the number of generated CAMs **(b)** Scenario 1 - Percentage of total latency by point

Figure 7.4: Scenario 1 - Analysis

7.4.2 Scenario 2 - Sending alerts to the vehicles

As mentioned previously, the user can generate alerts, in the form of DENMs, and send them to the vehicles, but in order for this to happen the message needs to be processed by several different services. Unlike the previous scenario, these messages are not generated periodically, instead they are triggered by the user. In this subsection, we will provide the results of the three tests that were performed to test the performance of the infrastructure when generating DENMs: the module latency tests, which measures the latency between the modules along the data flow; the end-to-end test 1, which focuses on testing the performance of the whole system, instead of the individual modules; and end-to-end test 2, which is designed to put the CM backend under a lot of stress.

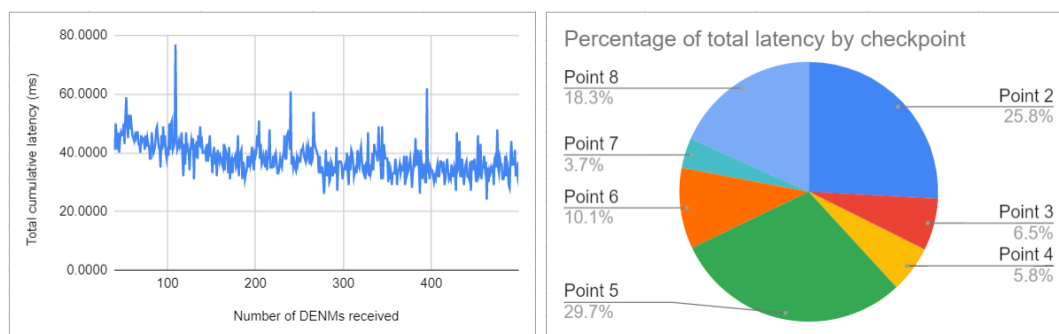
Module latency test

In order to simulate this behaviour, we automated the process of generating a DENM by using Postman ¹, which achieved this by performing POST requests to the POST /generateDENM endpoint of the CM. For this test we generated 500 DENMs and measured the timestamps of the 8 checkpoints along the data flow, explained in 7.2.2. Table 7.4 shows the results obtained from these tests. Much like in 7.4.1, we computed the average latency, the average cumulative latency and then the percentage of total latency. We can observe that the average total amount of latency, i.e., the total amount of time that it takes to generate a DENM and to send it to the vehicle is, on average, 49.82 ms and it reached a maximum value of 76.99 ms and a minimum of 23.99 ms. Figure 7.5a demonstrates the evolution of the performance of the infrastructure over time. This performance stays constant, which means that there are no degradations over time.

Figure 7.5b shows the percentage of total latency by checkpoint. It is possible to observe that point 2 and point 5 took the longest to be reached. Point 2 corresponds to the time that the mapping service took to compute the best RSU candidates, and point 5 corresponds to the time that it took to send and receive the DENM via DSRC. Note that the mapping service is implemented in Python, an interpreted language, which is slower than compiled languages. This may justify the latency of point 2.

Point	Average latency	Average cumulative latency (ms)	Percentage of total latency (%)
2	12.87	12.87	25.83
3	3.24	16.11	6.51
4	2.90	19.01	5.81
5	14.82	33.82	29.74
6	5.02	38.84	10.08
7	1.86	40.70	3.73
8	9.12	49.82	18.30

Table 7.4: Scenario 2 - Results



(a) Scenario 2 - Relation between total cumulative latency and the number of generated DENMs **(b)** Scenario 2 - Percentage of total latency by point

Figure 7.5: Scenario 2 - Analysis

¹<https://www.postman.com>

End-to-end test 1

In subsection we analysed the latency between individual modules of the infrastructure, but now we will provide an overview of the performance of the system as a whole.

For this test, we used JMeter ², which allowed to automate the process of sending HTTP POST requests to the /generateDENM endpoint of the CM backend. Besides this, JMeter also plotted the results in a chart and also provided additional statistics of the test.

JMeter is a very versatile testing program and provided multiple input parameters that could be adjusted in order to change the behavior of the test suite. The most important parameters that were used are:

- **Number of threads** - The number of threads, or virtual users, that will perform HTTP requests to the API.
- **Loop count** - The number of requests that each virtual user will make. A user can only send a request, after a response has been returned to the previous one.
- **Ramp up period** - The total amount of time, in seconds, that it will take to activate the full number of virtual users, e.g., if the test suite has 5 threads and a ramp up period of 10, it means that the test suite will take 10 seconds to turn create 5 virtual users and make them send HTTP requests.

This test is used to test the performance of the CM when the number of active concurrent users is less than the number of cores in the CPU of the machine that is hosting the CM. The input parameters of this test can be seen in table 7.5. Note that there may be a maximum of 2 concurrent users.

Figure 7.6 shows the results of this test. We can see that, for the 700 requests that were performed, the average amount of time that it took to get a response was 636 ms and a median of 514 ms. Notice that in the previous test, the maximum amount of time that it took to generate a DENM and send it to the vehicle was 76.99 ms, which is much lower than the 636 ms that we measured. This is due to the authentication operations performed by the Nginx ³ proxy that is responsible for performing the intermediate operations, such as routing the requests to the authentication service to check if the user has the right permissions to generate a DENM. In the previous test we skipped this proxy and made the requests directly to the machine that is running the CM.

Finally, we can conclude that, with this workload, the CM's performance stays constant over time and does not degrade.

E2E test 1	
NUM_THREADS	2
LOOP_COUNT	350
RAMP_UP_PERIOD	0

Table 7.5: E2E test 1 - Input parameters

²<https://jmeter.apache.org>

³<https://www.nginx.com>

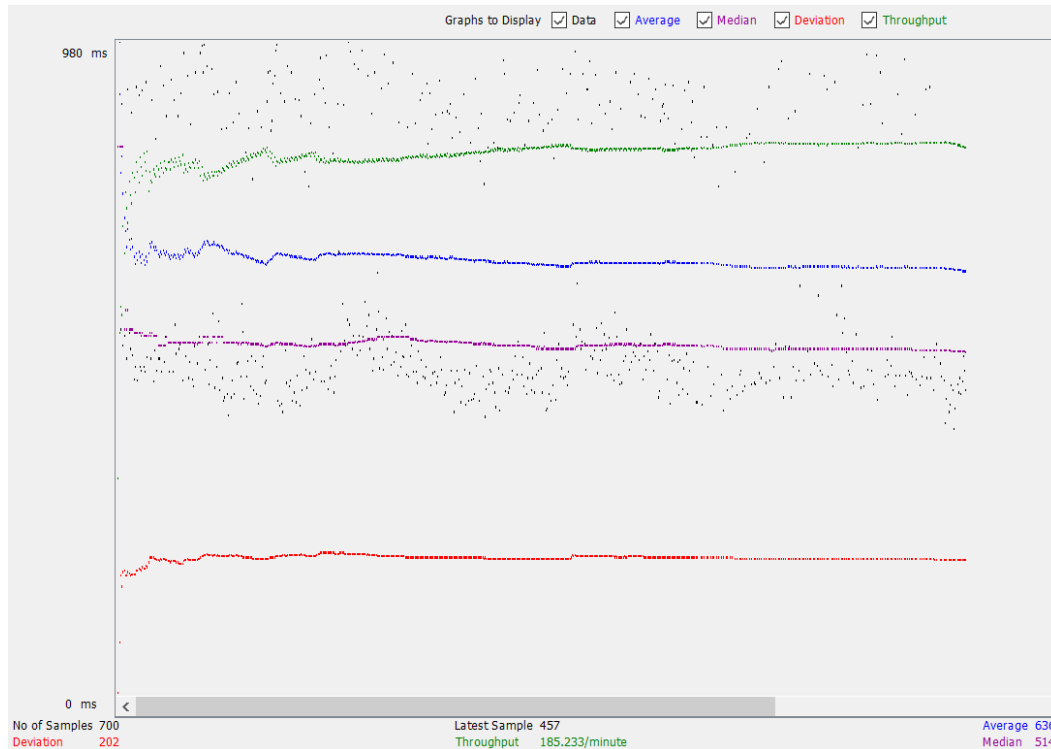


Figure 7.6: E2E test 1 - Performance

End-to-end test 2

Now that we have explained the scope of these end-to-end tests and their difference when compared to the module latency test, we will perform a test that is more computationally heavy than the previous ones.

Table 7.6 contains the input parameters of this test. Notice that this time, the test will launch 20 virtual users which is far greater than the number of cores of the CPU, so there may be a maximum of 20 concurrent users which is far beyond the capability of the machine where the CM backend is hosted.

The results of this test can be observed in figure 7.7. We can see that, for the first few requests, the backend is able to return the response in a reasonable amount of time, but the performance quickly degrades when more virtual users are activated, until it converges to nearly 2000 ms. This degradation happens when the number of concurrent requests surpasses the number of cores in the CPU.

There are multiple ways to solve this degradation in performance. In this case, we are only using one instance of the CM backend, but we could implement multiple instances and control the traffic so that requests are evenly balanced between these instances, using a load balancer. This is known as horizontal scaling. We could, alternatively, improve the resources of the machine that is hosting the CM, like getting a CPU with more cores, for example. This is known as vertical scaling.

E2E test 2	
NUM_THREADS	20
LOOP_COUNT	35
RAMP_UP_PERIOD	0

Table 7.6: E2E test 2 - Input parameters

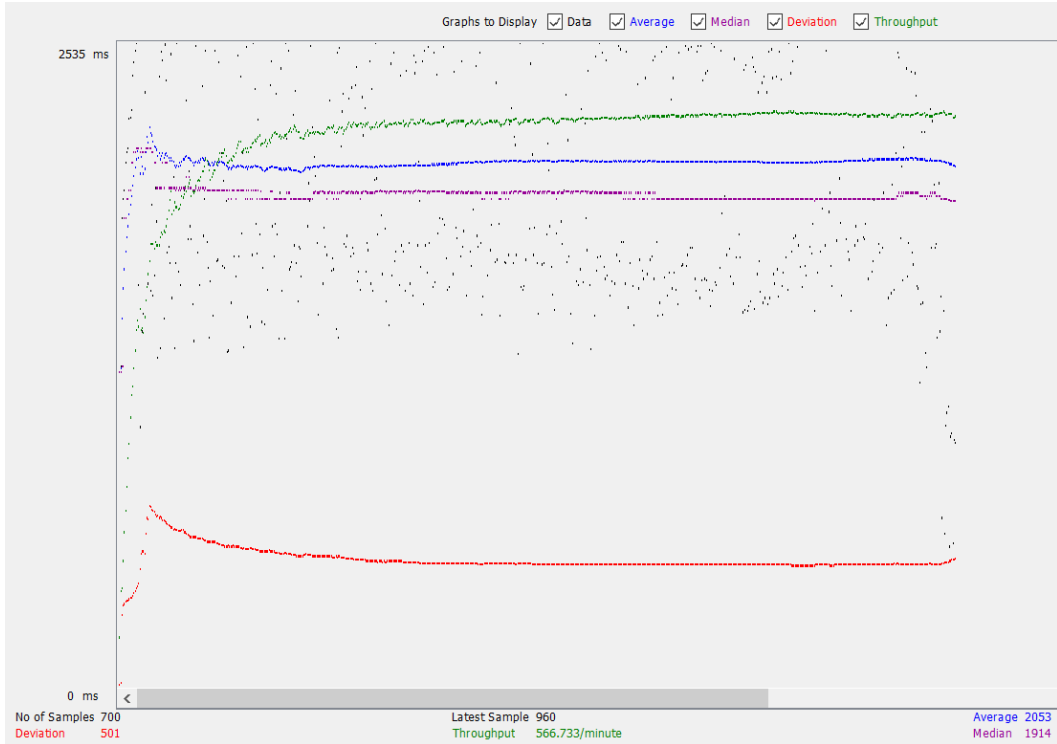


Figure 7.7: E2E test 2 - Performance

7.5 SUMMARY

We started this chapter by introducing two scenarios that were made possible by the implementation of this infrastructure. The first scenario consisted in sending vehicle data, in the form of CAMs, to the CM, through the infrastructure, and display it in the ART dashboard. The second scenario tested the inverse data flow, where an alert was generated, in the form of a DENM, by the user and was sent to the vehicle, through the infrastructure. This alert was then displayed to the driver using the notification app GUI.

In section 7.2 we analysed our testing environment and explained that we used one RSU and one OBU to simulate the connection between a vehicle that is driving on the road and comes near a RSU. We also explained that the mapping service was specially configured to redirect all DENMs to the RSU of this testbed, which allowed to more easily perform these tests. We also went into detail and explained the checkpoints where the timestamps are measured. In section 7.3 we presented the hardware specifications of the boards that were used to run the RSU and the OBU software and also presented the specification of the

machine that hosts the core services, like the mapping service and the CM backend. Finally, in section 7.4 we explained our testing methodology and provided the results of the tests. These results showed the latency between individual modules in the infrastructure, but also showed the performance of the CM app when placed under a high workload.

Conclusion and Future Work

In previous chapters we presented the requirements of the system, proposed an architecture that addresses them, analysed its implementation and presented our testing methodology as well as the results to these tests. In this chapter we will present the final conclusions that were made throughout the this dissertation, and suggest future improvements to the work developed.

8.1 CONCLUSION

The main objective of this dissertation was to developed services that enable V2I and I2V communication in vehicles, making use of the standards for Intelligent Transportation Systems, as well as services that allow the infrastructure to receive heterogeneous data, transform it into uniform data and store it in a database cluster. This data could then be accessed by high level services as real-time data or as historic data.

This concept opened up the possibility for new and interesting use cases, like the creation of a dashboard that displayed real-time information about the state of a city, which ranges from data acquired directly from vehicles to data acquired by other sensors, like radars or lidars. The modularity provided by this approach allows to simplify and isolate the complexity of some tasks, like developing an algorithm to acquire data from a radar, or developing an algorithm that analyses a video feed from a camera and detects how many people there are. Another interesting use for this system is the possibility to store and later access historic information, which leads to the interesting case of the Incident Replayer, which is able to replay the state of the city in any point in time, based on the persisted information. Notice that, before this data is stored, it is displayed in real-time in the ART, which demonstrates the true versatility of this solution. Replaying the state of a city can be particularly interesting if our goal is to analyse a crash, for example, because it gives access to detailed data that we would not have access to otherwise, which opens up the possibility for it to have a real and direct commercial value for companies, such as insurance companies.

Besides supporting the upward flow of information, i.e., from the vehicles to the infrastructure, V2I, this dissertation also focused on providing a system that supported the downward

flow of information, i.e., from the infrastructure itself to the vehicles, I2V. While there are large number of use cases that can take advantage of the downward flow of data, we decided to implement a notification system, that allows users to send alerts to vehicles, through their interaction with the ART dashboard. This alert is propagated through the infrastructure until it finally arrives in the vehicle and is displayed to the driver. This alert provides information to the driver that he may not have had access to otherwise, which has commercial value for companies that specialize in driver safety.

In conclusion, we presented an efficient and well performing solution that allows the generation of vehicle data in the vehicles themselves, allowed data acquisition and persistence by the infrastructure. This solution also made this data available in the form of real-time and historic data, which lead to interesting use cases, that have a real impact on changing driver behavior and improving driver safety. To demonstrate the versatility of the solution, we developed a dashboard that displays real-time data to the user, that allows city operators to monitor their cities more efficiently. This solution also leverages historic data, which allows for interesting use cases such as the IR that allows users to replay the data of vehicles that were involved in an accident.

8.2 FUTURE WORK

The emergent field of intelligent transportation systems is quite vast, and this dissertation covers only some of the possible use cases. Throughout this dissertation, we have mentioned and suggested ways where this work could be improved and extended in the future, but we will introduce them once again, for discussion.

- **Adding security mechanisms** - While the current solution supports authentication in its high level applications, it does not support any kind of authentication in its low level modules. The CAMs that are currently being generated contain all the vehicle information in clear text, which may not be desired.
- **Test V2V support more extensively** - The ITS modules are so versatile that we ended up implementing V2V communication, but since this was out of the scope of this dissertation we decided not to test this mechanism thoroughly, although this could be done in the future.
- **Integrate cooperative V2V communication** - The system currently supports V2V communication, but no logic is processed on the messages that are exchanged between vehicles. V2V has great potential and it could be used to address many use cases, such as lane merge and intersection crossing.
- **Allow DENMs to be relayed using V2V communication** - The current system allows V2V communication between vehicles, but it does not allow a vehicle to relay the DENM sent by another vehicle to the infrastructure, i.e., currently the vehicles receive ITS messages sent by neighbouring vehicles and interpret them, but this functionality could be extended to allow a vehicle to receive a DENM from another vehicle and send it to the infrastructure. This could be useful in the case of a vehicle that had an accident but did not have connectivity to the infrastructure. This way, the vehicle would be able

to notify a city operator, for example, which could dispatch emergency services to its location.

- **Sending DENMs through the platform using different technologies** - While messages are sent to and from the vehicles in different technologies, the transmission of them through the platform is only performed through DSRC. This may lead to undesired situations; for example, an operator wanting to warn the driver of potential dangerous driving conditions ahead, but is not able to send the warning because the OBU does not have this connection. The platform shall allow to communicate alerts through other technologies.
- **Scaling this solution** - As it currently stands, the solution is limited to only one instance of the CM backend and to a database cluster that is hosted in a single machine. The performance of the system can be greatly improved by deploying the CM backend in multiple instances, and redirecting the traffic evenly between them using a load balancer. Furthermore, it is also possible to improve the performance of the database system, by using well known strategies, like sharding, horizontal partitioning or vertical partitioning.
- **Allow users to choose a time window to replay** - If the efficiency of the system is improved, as previously suggested, new use cases become viable. Allowing users to choose a time window is a very time consuming operation if the user chooses a very large time window. This would give the user more freedom with the information that is acquired by the infrastructure, since it is not limited to replaying certain incidents.

References

- [1] U. Nations, *The World's Cities in 2018*. United Nations, 2018. [Online]. Available: <https://www.un-ilibrary.org/content/books/9789210476102>.
- [2] P. Khanna, *How much economic growth comes from our cities?* <https://www.weforum.org/agenda/2016/04/how-much-economic-growth-comes-from-our-cities>, [Online; accessed 05-December-2021], 2016.
- [3] J. R. Short, *Want the economy to grow? We need to make cities more efficient*, <https://citymonitor.ai/economy/want-economy-grow-we-need-make-cities-more-efficient-1872>, [Online; accessed 05-October-2021], 2016.
- [4] B. Insight, *The Global Automotive OEM Telematics Market*, <https://www.berginsight.com/the-global-automotive-oem-telematics-market>, [Online; accessed 05-October-2021], 2020.
- [5] M. Won, "L-platooning: A protocol for managing a long platoon with dsrc," *IEEE Transactions on Intelligent Transportation Systems*, pp. 1–14, 2021. DOI: 10.1109/TITS.2021.3057956.
- [6] G. Domingues, J. Cabral, J. Mota, P. Pontes, Z. Kokkinogenis, and R. J. F. Rossetti, "Traffic simulation of lane-merging of autonomous vehicles in the context of platooning," in *2018 IEEE International Smart Cities Conference (ISC²)*, 2018, pp. 1–6. DOI: 10.1109/ISC2.2018.8656856.
- [7] ETSI, *Cooperative Awareness Messages*, https://www.etsi.org/deliver/etsi_en/302600_302699/30263702/01.04.01_60/en_30263702v010401p.pdf, [Online; accessed 05-October-2021], 2019.
- [8] ———, *Decentralized Environmental Notification Messages*, https://www.etsi.org/deliver/etsi_en/302600_302699/30263702/01.04.01_60/en_30263702v010401p.pdf, [Online; accessed 05-October-2021], 2014.
- [9] ———, *Collective Perception Messages*, https://www.etsi.org/deliver/etsi_tr/103500_103599/103562/02.01.01_60/tr_103562v020101p.pdf, [Online; accessed 05-October-2021], 2019.
- [10] S. Djahel, M. Salehie, I. Tal, and P. Jamshidi, "Adaptive traffic management for secure and efficient emergency services in smart cities," in *2013 IEEE International Conference on Pervasive Computing and Communications Workshops (PERCOM Workshops)*, 2013, pp. 340–343. DOI: 10.1109/PerComW.2013.6529511.
- [11] M. Malinverno, G. Avino, C. Casetti, C. F. Chiasserini, F. Malandrino, and S. Scarpina, "Edge-based collision avoidance for vehicles and vulnerable users: An architecture based on mec," *IEEE Vehicular Technology Magazine*, vol. 15, no. 1, pp. 27–35, 2020. DOI: 10.1109/MVT.2019.2953770.
- [12] Tesla, *Tesla Autopilot project*, <https://www.tesla.com/autopilot>, [Online; accessed 11-October-2021].
- [13] I. Elleuch, A. Makni, and R. Bouaziz, "Cooperative intersection collision avoidance persistent system based on v2v communication and real-time databases," in *2017 IEEE/ACS 14th International Conference on Computer Systems and Applications (AICCSA)*, 2017, pp. 1082–1089. DOI: 10.1109/AICCSA.2017.20.
- [14] N. C. Basjaruddin, Z. I. R. Noor, and D. H. Widyantoro, "Multi agent protocol for cooperative rear-end collision avoidance system," in *2019 2nd International Conference on Applied Information Technology and Innovation (ICAITI)*, 2019, pp. 23–26. DOI: 10.1109/ICAITI48442.2019.8982117.
- [15] B. Cheng, S. Longo, F. Cirillo, M. Bauer, and E. Kovacs, "Building a big data platform for smart cities: Experience and lessons from santander," in *2015 IEEE International Congress on Big Data*, 2015, pp. 592–599. DOI: 10.1109/BigDataCongress.2015.91.

- [16] A. Harris, J. Stovall, and M. Sartipi, “Mlk smart corridor: An urban testbed for smart city applications,” in *2019 IEEE International Conference on Big Data (Big Data)*, 2019, pp. 3506–3511. DOI: 10.1109/BigData47090.2019.9006382.
- [17] B. Ahlgren, M. Hidell, and E. C.-H. Ngai, “Internet of things for smart cities: Interoperability and open data,” *IEEE Internet Computing*, vol. 20, no. 6, pp. 52–56, 2016. DOI: 10.1109/MIC.2016.124.
- [18] P. Bellini, P. Nesi, M. Paolucci, and I. Zaza, “Smart city architecture for data ingestion and analytics: Processes and solutions,” in *2018 IEEE Fourth International Conference on Big Data Computing Service and Applications (BigDataService)*, 2018, pp. 137–144. DOI: 10.1109/BigDataService.2018.00028.
- [19] I.-X. Chen, Y.-C. Wu, I.-C. Liao, and Y.-Y. Hsu, “A high-scalable core telematics platform design for intelligent transport systems,” in *2012 12th International Conference on ITS Telecommunications*, 2012, pp. 412–417. DOI: 10.1109/ITST.2012.6425209.
- [20] R. Lee, R.-y. Jang, M. Park, G.-y. Jeon, J.-k. Kim, and S.-h. Lee, “Making IoT Data Ready for Smart City Applications,” in *2020 IEEE International Conference on Big Data and Smart Computing (BigComp)*, 2020, pp. 605–608. DOI: 10.1109/BigComp48618.2020.00020.