POLITECNICO DI TORINO

Master of science-level of the Bologna process in
Communication and Computer Networks Engineering

Final project work

Experimental analysis of 802.11p boards on real vehicles

Advisors
prof. Claudio Ettore Casetti
dott. Francesco Raviglione
dott. Gianluca Rizzo

Saira Bano
Student ID: 252873

July 2020

Summary

In this master degree thesis, tests are planned for real-world automotive scenarios using APU1D embedded boards, built by PC Engines, a Swiss producer.

On these boards an embedded Linux operating system (OpenWrt) is used, providing an open-source development environment with a high degree of customizability.

The IEEE 802.11p protocol an automotive version of ”WiFi” was already implemented on these boards.
These PC Engines boards are used as On-Board Units integrated in an electric vehicle, which was made available for this work.

The main focus is to test an open implementation of IEEE 802.11p and to experimentally investigate its characteristics in real-world settings, with the main focus on Key Performance Indicators such as latency and throughput.

The goal is to test our implementation and to analyze the performance of the system from the collected data, highlighting strengths and possible problems that could arise in a real world mobility scenario. A GPS device is also used in this work in order to obtain measurements of the speed and distance covered by the car.

The CAN network interface of the electric vehicle was also used to get the Engine RPM values. These RPM values can be used to determine a real-time estimate of the vehicle speed by using a GPS and CAN data, by applying different speed filtering methods, as it was not directly available through CAN.

The chapters are organized in the following order:

1. The first chapter introduces the concept of Intelligent transportation systems (ITS) and its use cases, together with the related protocols for vehicular communications. It also describes the introduction of VANETs (Vehicular Ad-hoc Networks) and some available embedded solutions for vehicular networks.

2. The second chapter is related to the WAVE (Wireless Access in vehicular Environment) stack, describing the functionality of each layer in detail. It also describes the tool used in this work for latency measurements, called LaTe.

3. The third chapter is related to the Controller Area Network, describes the working principles of the CAN protocol and how the Engine RPM values are extracted from a CAN socket. It also describes the methodology of getting the speed of the car by using the Engine RPM values and GPS.

4. The fourth chapter is related to the Global Positioning System (GPS), describing in detail the NMEA-0183 standard used to generate GPS sentences. Some of the GPS sentences are used to measure the speed of the electric vehicle. The distance covered by the vehicle is also calculated by using the latitude and longitude values and this procedure is also detailed in this chapter.

5. The final chapter is related to the test planning, in order to determine the 802.11p performance in mobility scenarios. The tests are planned to be performed by using the APU1D boards as On Board Units and Road Side Units, the GPS device and an electric vehicle. These tests are used to determine different system parameters well suited for automotive scenarios and to analyze the performance of 802.11p protocol. Unfortunately the activity was limited to test planning and software development, as it was not possible to perform the measurements on the actual vehicle due to the COVID-19 emergency.

6. The chapter 6 is related to the conclusion and the future work.

Contents

Abbreviations

Organizations

- **ARIB**: Association of Radio Industries and Businesses
- **ETSI**: European Telecommunications Standards Institute
- **FCC**: Federal Communications Commission
- **IEEE**: Institute of Electrical and Electronic Engineers
- **NMEA**: National Marine Electronics Association
- **SAE**: Society of Automotive Engineers

Vehicular communications

- **CAN**: Controller Area Network
- **DSRC**: Dedicated Short Range Communications
- **ECU**: Electronic Control Unit
- **HDV**: heavy Duty Vehicle
- **ITS**: Intelligent Transportation System
- **MANET**: Mobile Ad hoc Networks
- **OBU**: On Board Unit
- **OCB**: Outside the Context of BSS

- **OEM**: Original Equipment Manufacturer
- **PF**: Platoon Follower
- **PL**: Platoon Leader
- **RPM**: Revolutions Per Minute
- **RSU**: Road Side Unit
- **V2I**: Vehicle to Infrastructure
- **V2P**: Vehicle to pedestrian
- **V2V**: Vehicle to Vehicle
- **V2X**: Vehicle to everything
- **VANET**: Vehicular Ad-hoc Network
- **VCAN**: Virtual Controller Area Network

Navigation systems

- **DOP**: Dilution of Precision
- **GLONASS**: Global Navigation Satellite System
- **GNSS**: Global Navigation Satellite System
- **GPS**: Global Positioning Systems
- **HDOP**: Horizontal Dilution of Precision (Global Positioning System)
- **UTC**: Universal Time Coordinated

Telecommunications

- **ICT**: Information and Communication Technology
- **ISI**: Inter Symbol Interference
- **KPI**: Key Performance Indicators
- **LAN**: Local Area Network
- **LLC**: Logical Link Control
- **MAC**: Medium Access Control (layer)
- **NRZ**: Non Return-to-Zero
- **PHY**: Physical (layer)
- **RTT**: Round Trip Time
- **SDU**: Service Data Unit
- **SOF**: Start Of Frame

Standards (IEEE WAVE & ETSI) and protocols

- **AC**: Access Category (EDCA)
- **AIFS**: Arbitration InterFrame Space
- **BSM**: Basic Safety Message
- **CAM**: Cooperative Awareness Message
- **CCH**: Control Channel
- **DCC**: Decentralized Congestion Control
- **DENM**: Decentralized Environment Notification Message
- **EDCA**: Enhanced Distributed Channel Access
- **ICMP**: Internet Control Message Protocol

- **IP**: Internet Protocol
- **LaMP**: Latency Measurement Protocol
- **LaTe**: Latency Tester
- **SAM**: Service Announcement Message
- **TCP**: Transmission Control Protocol
- **TWAMP**: Two Way Active Measurement Protocol
- **UDP**: User Datagram Protocol
- **WAVE**: Wireless Access in Vehicular Environments
- **WSMP**: WAVE Short Message Protocol

WiFi or IEEE 802.11

- **AP**: Access Point
- **BO**: BackOff (procedure)
- **BSS**: Basic Service Set
- **DCF**: Distributed Coordination Function
- **DIFS**: Distributed InterFrame Space
- **ESS**: Extended Service Set
- **IBSS**: Independent Basic Service Set
- **OFDM**: Orthogonal Frequency-Division Multiplexing
- **TDMA**: Time Division Multiple Access

Hardware systems

- **COM**: Communication port
- **DRAM**: Dynamic Random Access Memory
- **LPF**: Low Pass Filter
- **PCI**: Peripheral Component Interconnect
- **USB**: Universal Serial Bus

Chapter 1

Introduction

1.1 Intelligent Transportation Systems

In 1991 the United States Congress demanded the formulation of the IHVS (*Intelligent Vehicle Highway Systems*) program in order to seek improvements in their transportation infrastructure [1]. The services which are defined by this program constitute the so-called **ITS** (*Intelligent Transportation Systems*). The ITS defined the usage of ICT technologies that are applied to transportation systems to exchange information among vehicles, in order to create smarter transportation systems for an efficient use of the road infrastructure, with the addition of providing safety and comfort to users, reducing fuel consumption and also road fatalities.

According to [2] around 1.24 millions fatalities occur every year, and leave millions of people disabled. The work on ITS started back in the 90s to improve traffic safety but this area is still under development and leading researchers, car manufacturers, network operators, transport planners, public authorities and ITS organizations to work together. European commission also took initiatives like the "*i2010 Intelligent Car Initiative*" to support research in the area of transportation.
Some use cases for ITS, in order to provide road safety and to improve overall traffic flow management have been reported in [3]:

- **Platooning**: which can be implemented in any kind of vehicle but which appears to be more effective in the case of an HDVs (Heavy Duty Vehicles). On highways HDVs will travel in the right-most lane and can form a platoon of vehicles that can

communicate with each other, creating an increased group awareness. There is normally one Platoon Leader (PL) and many Platoon Followers (PF) that are mimicking the PL; this allows the vehicles to maintain a smaller inter-vehicle distance, still without any safety concern as all the maneuvers are performed keeping the same speed between vehicles, thus increasing highway capacity.
Platooning also helps in reducing fuel consumption by reducing aerodynamic resistance. By 2050 gas emissions shall be reduced by 60% for tenable environment as stated by European Commission in [4], and platooning can help in reducing fuel consumption as mentioned before.

- **See Through**: also called ”non line of sight”. As far on a road with only one lane per the direction of travel is concerned, overtaking can be challenging when there is a vehicle reducing the visibility in front of the car, such as train trucks whose length typically ranges from 18 to 25 meters. In this scenario inter-vehicle communication enables the driver to see what the vehicle in front sees, thanks to a video wireless transmission, thus increasing drivers awareness.

- **Collision prevention systems**: annually millions of dollars are spent in the research activities for cooperative systems in the field of ITS, for the comfort and safety they provide to passengers and drivers. Also in autonomous driving a lot of research has been carried out to apply collision avoidance algorithms for vehicles safety at intersections and for pedestrians that are more susceptible to road fatalities.

  Some recent work make use of GPS receivers to determine precise location and to continuously update the driver about possible collisions, transmitting information about hazards warnings to other vehicles [5]. Autonomous cars are also equipped with a large number of sensors, from which data is collected and can be used in collision avoidance algorithms.

1.2 VANETs

One of the major components of the ITS are the so called VANETs. VANETs (Vehicular Ad-hoc Networks) are a specific type of MANETs (Mobile Ad-hoc Networks).

MANETs are self organizing and dynamic collections of mobile nodes; these mobile nodes can act both as hosts and routers to communicate with other nodes, following a

multi hop approach until the packets are received at the destination; based on the destination address the final node can retain the packet or discard it. Due to dynamic nature of these nodes, communication should be adapting to changes in the location of the nodes and in the nearby environment [6]. In case of VANETs the mobile nodes are the vehicles, which exchange information among themselves and with the infrastructure; this kind of communication is generally referred as V2X (*Vehicle to everything*), setting up a collective intelligence in transportation systems, due to the formation of VANETs [7].
The major characteristics of VANETs are:

- The network topology in VANETs changes very rapidly as the vehicles may move even at very high speed, so this continuous change in position can lead to frequent topology changes [8].
- VANETs are ad-hoc networks, which means that they are created and terminated at any time when its needed and can exchange the information with other vehicles and Road side units [8].
- Due to frequent changes of topology routing decisions are difficult to make in VANETs.
- Vehicular Ad-hoc Networks are used to provide safety and comfort to drivers, so it is necessary to develop a low-latency communication due to delay constraints in safety critical applications, such as in collision prevention systems.

1.2.1 Components of VANETs

The major components of the architecture of a VANET are shown in Figure 1.1. In particular, they are described in the following list:

- **On Board Units**: these are devices located in a car and containing a Resource Command Processor (RCP), Memory, and network interface to connect to other OBUs and RSUs based either on the IEEE 802.11p protocol designed for short range communications or on C-V2X (Cellular vehicle-to-everything) that is based on the cellular networks, but which is out of scope of present thesis work [9].
- **Road Side Units**: these devices are situated at the edge of roads and are fixed; they are used to provide broader connectivity in VANETs, connecting farther OBUs. RSUs also act as routers or gateways and provide internet connectivity and other infrastructure related services to the vehicles.

- **Application Units**: these are sophisticated devices usually embedded with On Board Units to provide safety or infotainment applications by communicating with RSUs or other OBUs.
- **GPS**: acronym for Global Positioning System, it is used in order to gather the precise position of vehicles and to inform other vehicles about accurate hazard locations to avoid accidents. GPS is also used in position based routing protocols, which will be discussed later on in this chapter. Moreover these devices can also be used to determine the speed of vehicles.

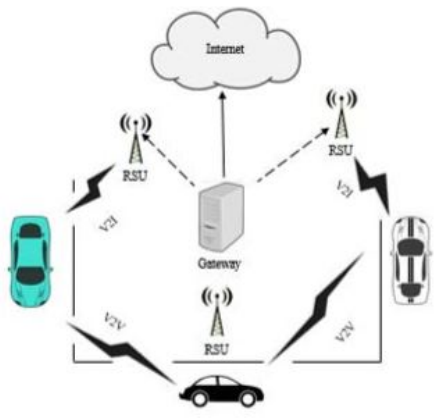


Figure 1.1: Architecture of a VANETs. Taken from [9]

1.2.2 Types of Communication in VANETs

There are various kinds of communication in VANETs such as:

- **V2V**: Vehicle to vehicle communication, in which OBUs are directly communicating with each other.
- **V2I**: Vehicle to infrastructure communication or vice versa, in which vehicles communicate with road side units.
- **V2P**: Vehicle to pedestrian communication, as one of the major goals of V2X is to protect pedestrians and to save their lives. Nowadays, almost all pedestrians carry smartphones, and using cellular technology, a V2P communication can happen helping the vehicle-side recognition of the pedestrians.

- **V2N**: Vehicle to Network communication, it will connect the vehicles to the cellular infrastructure and clouds, and constitute the so called C-V2X (Cellular-Vehicle to Everything) [10]. It would provide the commercial services such as infotainment applications and navigation functions (Google Maps).

1.2.3 Routing in VANETs

In VANETs routing is not so obvious as compared to classical routing approaches in mobile networks, considering the fact that the topology continuously changes due to the vehicles and their movements.

Many routing algorithms have been proposed in recent years and classified in five major categories as mentioned in [11] and as shown in Figure 1.2.

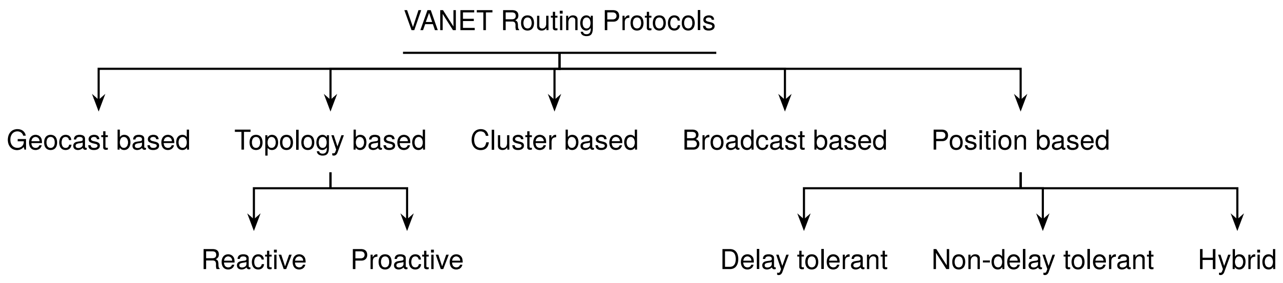


Figure 1.2: Taxonomy of VANET routing protocols

- **Topology based routing**: this family of routing protocols is based on the use of routing tables for destinations, and it is further characterized as:
  - *Proactive routing*: routes to the destination are already computed; each node maintains the routing table towards the destination but, as the topology in the VANET changes, these routing tables need to be re-computed or updated.
  - *Reactive routing*: rather than maintaining the routing tables, whenever nodes need to exchange data, route towards the destination are computed at run time; in case of safety related applications this is not suitable because of the high latency in computing paths.
- **Position based routing**: this protocol is based on the geographic position of nodes that can be determined using GPS, so that each node have knowledge of its location

and its nearby nodes. This location information is inserted in packet headers and, using a multi hop approach, packets can reach the destination node without any insight into the network topology [9]. This routing scheme is further characterized as:

  - *Delay tolerant*: it is used when there are frequent disconnections within the network, in this case a carry and forward scheme will be used in order to save the packet and deliver them to the neighboring nodes.
  - *Non-delay tolerant*: it is also called minimum delay protocol, it works well in dense environments, with the main goal of delivering packets from one to node to another in shorter time by using the shortest path algorithms.
  - *Hybrid*: it is a mixture of more than one routing protocols (either delay tolerant or non-delay tolerant); it also combines more than one routing protocol based on the topology [11].

- **Cluster based routing**: in cluster based routing the network is divided into different clusters formed by vehicles, that are travelling in same direction and with the same speed. Particularly communications within and outside the cluster are managed by a cluster head that is unique, and which broadcasts the packets to all the nodes within the cluster; however, if the cluster is large there are large delays with increased overhead [11].

- **Geocast routing**, also called *"Topology-Assisted Geo-Opportunistic Routing"*: as among all the routing protocols, geocast is contemplated to be the best [11]. It is a multi-cast routing protocol that uses GPS to know the position of the vehicles as well as their neighboring nodes. The geographical area is divided into different zones and message are sent to all the vehicles in that zone [12].

- **Broadcast routing**, which is also called flooding: in order to avoid the overhead of finding paths when an immediate response to events is needed, and to reach longer ranges, information is broadcasted to all the neighboring nodes. This works well when the density of vehicles is small, while on the contrary, in dense scenarios, vehicles are further broadcasting data creating a broadcast storm which basically means echoing the same message over and over, thus flooding the network and as a result, inducing problems such as packet collisions, network congestion, interference with nearby systems and redundant rebroadcasts [13].

Many broadcast suppression techniques have been proposed in recent years to face this issue; one of them is called ”timer based intelligent flooding”, in which each forwarder possibly wait for a certain time before rebroadcasting a packet; in particular this time is inversely proportional to the distance from the sender node, such that farther nodes have a higher probability of re-transmission; moreover, if the timer of a node hasn’t expired yet and it overhears the same message it will discard any successive copy of this message [14].

1.2.4 Standardization of VANETs

In order to promote safety applications in transportation, the Federal Communications Commission (FCC) of The United States, in 1999, has allocated a bandwidth of 75 MHz around 5.9 GHz for vehicular applications.

Around this spectrum IEEE developed a DSRC (Dedicated Short Range Communication) set of protocols also named WAVE (Wireless Access in Vehicular Environments) to enable vehicle to vehicle (V2V) and vehicle to infrastructure communications (V2I) [15].

Different countries have worked for the standardization of DSRC including USA, Europe and Japan; while US and Europe are using the same frequency bands (5.850-5.925 GHz), Japan uses a different approach and instead of 5.850-5.925 GHz it uses a band around 700 MHz, which is standardized as ARIB STD-T109 [16]. These standards are also summarized in Table 1.1.

Excluding Japan there are two main standardization efforts: IEEE developed WAVE that

Country	**Frequency band(MHz)**	**Standards**
Europe	5875-5925	ETSI ITS-G5
Japan	700	ARIB STD-T109
America	5850-5925	WAVE

Table 1.1: DSRC range and standards in different regions

offers a whole stack including physical layer, MAC layer and upper layers standards, while in Europe, ETSI (European Telecommunications Standards Institute) has developed ITS-G5 that uses the same physical and MAC layers as WAVE, and which builds a variant of the upper layer protocols. The foundation of both stacks is detailed by IEEE 802.11p, which is an amendment of IEEE 802.11; used as WAVE PHY layer and WAVE Lower MAC.

IEEE WAVE

The WAVE standard is released by the combined efforts of the IEEE 1609 and IEEE 802.11p groups, to support DSRC operations in 802.11p: the timing parameters are doubled to tackle problems such as delay spread, fast fading and Doppler effect, this also reduces the channel bandwidth from 20 MHz to 10 MHz [16]. While Japan uses a different approach and instead of OFDM it uses a mixture of CDMA/CATDMA scheme. However, we are not going to discuss in detail about the Japanese standards.
In WAVE MAC layer channel access is similar to 802.11 but there is no BSS (Base Service Set) establishment and a new mode is defined, called OCB (Outside the context of BSS), in order to support immediate data exchange without performing the authentication and association steps required in 802.11 networks that are AP$-$based. It also introduces congestion based selection of EDCA (Enhanced distributed channel access) parameters that will be described later on in chapter 2.

The 75MHz bandwidth (5.850-5.925 GHz) reserved for DSRC is further divided into seven 10 MHz channels. Out of these seven channels, one is a control channel (CCH) and six are service channels (SCH); in particular there are two IEEE 802.11 MAC entities: one for SCH and one for CCH. A guard interval of 5 MHz at the beginning of 75 MHz DSRC band is also present.
The control channel is used to carry high priority and non-IP data and also to advertise different services that are present on service channels, together with management data, while other data transmissions occur on SCH. The standardized formula mentioned in [17], to derive channel numbers is:

$$f(CN) = 5000 + 5CN(MHz) \tag{1.1}$$

Table 1.2 shows the different frequency ranges assigned to each channel number.

The upper layers in the IEEE protocol stack have been defined collectively by the IEEE 1609.x-2016 standards and the SAE (Society of Automotive Engineers) J2735 standard. The upper layers of this stack will be defined more in detail in chapter 2. J2735 defines 16 different types of messages of which the important one is **BSM** or Basic Safety Messages; which is defined, just like CAMs (Cooperative Awareness Messages)[2], to report safety events and to inform other vehicles about speed, position and heading.

[1]This band is used as a broad band radio access network, further details are available in [18]

[2]CAM will be describe in next section of ETSI ITS-G5

Frequency [GHz]	**DSRC Channels**
5.895-5.905	CCH-180
5.885-5.895	SCH2-178
5.875-5.885	SCH1-176
5.865-5.875	SCH3-174
5.855-5.865	SCH4-172
5.470-5.725	SCH7-94 to 145 [1]
5.905-5.915	SCH5-182
5.915-5.925	SCH6-184

Table 1.2: DSRC channels and their channels frequency range

The fundamental technology for short latency requirements and for road safety is DSRC/WAVE [17].

ETSI ITS-G5

With respect to IEEE WAVE, ETSI ITS-G5 introduces variants on how the PHY and MAC layer parameters are chosen in order to support the dynamic selection of the range at which the vehicles can communicate, according to the channel conditions, and set up the so called DCC (Decentralized Congestion Control) system that is one of the core features of ITS-G5, in which the congestion can be reduced by lowering the transmission power or transmission rate. It also introduces one more layer between the networking one and the application one called **Facilities** layer, providing additional application, information and communication support. The main type of messages defines by the application support facilities are:

- **CAM**: which stands for Cooperative Awareness Messages. The purpose of these messages is to let vehicles know about each other's location, their speed, and the direction in which nearby nodes are going to move. The frequency of these messages is usually 1-10Hz, which means that CAM are sent periodically at 1 Hz but it can increased up to 10 Hz under some particular conditions such as: if the car is accelerating over a certain acceleration and the difference between the current and the previous reported speed value by CAM is more than 0.5 m/s , or if the car is turning and the angle difference in the heading between position sent in the previous CAM and the current position is more than 4 degrees.

- **DENM**: which stands for called Decentralized Environmental Notification Message.

These messages are triggered by an event, and among other information, they will transmit latitude and longitude of the place where some event occurred. These kind of messages have a local scope based on the topology or area.

- **SAEM**: which stands for Services Announcement Essential Messages, the transmission and reception of these messages are handled by (SA) Services Announcement protocol over the facilities layer [19], to periodically advertise services available on different service channels.

1.3 Embedded solutions for VANETs

Although researchers mostly rely on simulations to test and study different protocols, it is still necessary, for more reliable results, to test V2X protocol stacks on real time embedded devices. There are many embedded solutions available in the market but some of them are expensive while some offers a partially support to full IEEE WAVE stack. 802.11p radios have been developed by many manufacturers including Cisco/Cohda Wireless, Commsignia, Denso, Savari, Kapsch, Siemens, Unex, Autotalks and Arada [20]. Some solutions are listed here, taking as reference the list in [3]:

- **Cisco/Cohda Wireless**: Cohda Wireless is a vendor of automotive application equipment in the Cooperative Intelligent Transport Systems (ITS) market [21]. Cisco instead is the world leader in providing networking solutions and producing them; Cisco made a joint efforts with Cohda to develop smart city embedded systems.

  Cohda provides complete hardware boards (MK5) for both OBUs and RSUs. Both boards are using the same chipset provided by NXP semiconductors and named *"RoadLINK"*. These boards are suitable to work in harsh environments and are equipped with dual antennas, a GNSS receiver and support all the DSRC (802.11p) communications for vehicles and for smart city deployments. It is also stated on their website that MK5 OBUs show better performance with respect to other producers for non line of sight scenarios [22].

- **Arada systems**: they develop OBUs called "Arada LocoMote" that are based on the IEEE 802.11p specifications to operate on 5.9 GHz and provide vehicle to vehicle communications; they offer low latency for V2V and V2I connectivity, high data rates, and an integrated GPS device [23].

- More open and customizable solutions are also available, including the DCMA-86P2 IEEE 802.11p modules made by UNEX, Taiwan. These cards operate in the 5.850-5.925 GHz frequency range, and use an Atheros AR5414 RF transceiver produced by Qualcomm, in order to enable communications according to IEEE 802.11p; they are mini-PCI modules supported by the *ath5k* Linux driver, that can be used with any compatible board and can be powered by a single 3.3V supply [15].
- **PC Engines**, they make and sell small single board computers for networking solutions. The APU1D boards which are used in this work, are from PC Engines, using an AMD G-series processor with 64 bit support, providing two miniPCI express slots and operating on 12V DC power supply [24]. The APU boards provide better performance as compared to the ALIX series boards, which are also developed by PC Engines, as embedded solutions for networking using *AMD Geode* processors [25].
- **Kapsch TrafficCom**: it's an Austrian group also providing commercial solutions for vehicle to vehicle and V2I communications in the DSRC frequency range around 5.9 GHz, supporting both the IEEE and ETSI standards; one of their products are the KVE-3320 V2XECU OBUs, which are using ARM processors (ARM Cortex A7) [3].
- **Savari**:It is an USA based company, providing the solutions for V2X communication such as: V2V, V21, V2P and I2P (Infrastructure-to-Phone) [26]. The software stack provided by them is radio-agnostic and can be fit to any radio module, thus making it easy for the V2X stack to adapt to the future changes.

1.3.1 Objectives and used hardware and software system

The main objectives of this work are:

- First of all, the objective is to test and evaluate IEEE 802.11p in a real world automotive scenario, involving an electric vehicle which was available at the time this work was developed; the focus was put on Key Performance Indicators (KPIs) such as throughout and latency.
- Another objective is to perform a comparison with static IEEE 802.11p measurements, using the APU1D boards and results which came from a previous work of

Politecnico di Torino [3].

- By using the interface with the CAN in-vehicle network and a GPS device, a methodology for evaluating the vehicle speed is proposed, as this variable can be very useful in different scenarios, including the possibility to enrich the previously mentioned measurements, correlating, for instance, the measured latency with the variable speed between an OBU, on the vehicle, and a fixed RSU.

The hardware used in this work, as OBUs and RSUs, are APU1D boards built by PC Engines, embedding a 1 GHz AMD G series of processor and mounting UNEX DHXA-222 wireless cards. These boards are equipped with IEEE 802.11p functionalities, developed in a Politecnico di Torino thesis work titled as *"Implementation of the IEEE802.11p protocol on embedded systems"* [3]. On these boards a patched Linux distribution for embedded devices has been used [27] i.e. OpenWrt, that can be easily modified according to different requirements providing a good degree of flexibility as it is an open-source platform.

Using OpenWrt with the APU1D boards provides a customizable and effective low-cost solution for networking experimentation.

The vehicle we used is an electric vehicle that can reliably reach the speed of 40 to 45 km/h. It provides the CAN bus[3] to read the data along with CAN IDs that are described in a partial database, that we got from the producer. However, we do not have all the information about the mechanical system of the electric vehicle.

[3]We had to manually interface with the CAN bus by dismounting the car, described later in chapter 3

Chapter 2

IEEE Standards and protocols

2.1 IEEE WAVE

The Institute of Electrical and Electronics Engineers (IEEE) has defined a family of standards for vehicular communications i.e. the 802.11p/1609.x protocol stack, also called WAVE, that was first released in 2006. This standard defines the architecture and a standardized set of services for wireless communications [28].

IEEE took the existing 802.11 standard and carved out a dedicated set of rules that would apply to the specific environment of vehicular communications, to tackle the high mobility and dynamics of vehicular networking. So an amendment was made to 802.11, based on 802.11a and called 802.11p in order to target the requirements of vehicular networking [16].

The WAVE stack includes IEEE 802.11p, used as WAVE PHY layer and WAVE Lower MAC, thus IEEE 802.11p details the foundation on which the WAVE stack resides. The DSRC protocols for the upper layers are instead managed by the IEEE 1609 family of standards that will be discussed later in this chapter.

In particular, the IEEE 1609.3 standard is used to provide networking services. The IEEE 1609.4 standard that is placed on top of the IEEE 802.11p and enables the multi-channel operation [29] and 1609.2 provides security related functions. The WAVE protocol stack components are shown in Figure 2.1.

In this chapter the IEEE WAVE protocol stack will be described in detail, focusing on its usage in vehicular communications.

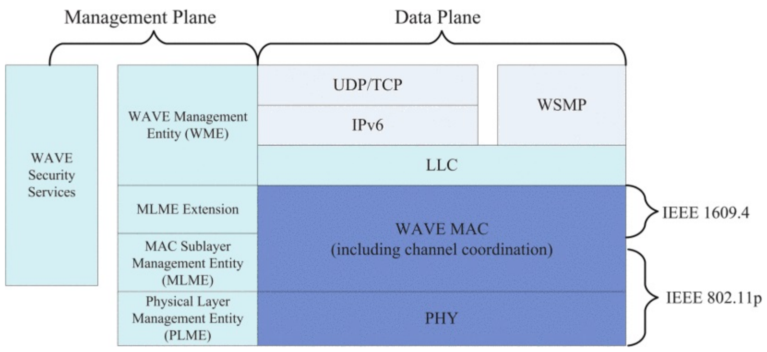

Management Plane
Data Plane
WAVE Security Services
WAVE Management Entity (WME)
UDP/TCP
IPv6
WSMP
LLC
MLME Extension
MAC Sublayer Management Entity (MLME)
Physical Layer Management Entity (PLME)
WAVE MAC (including channel coordination)
PHY
IEEE 1609.4
IEEE 802.11p

Figure 2.1: WAVE Architecture. Taken from [30]

2.1.1 WAVE PHY layer

802.11p defines the physical layer of the WAVE stack and it is derived from the existing 801.11a standard, that uses a transmission technique, called ”OFDM”, to support different data rates. OFDM stands for Orthogonal Frequency Division Multiplexing and it provides high aggregated data rates by transmitting simultaneously over different frequencies at the lower data rates by using different sub carriers. OFDM is also very well tolerant to multipath distortion.

As already discussed in chapter 1, the 75 MHz band is divided into seven channels; one CCH mapped to channel 178 and 6 SCH, as shown in Figure 2.2. The channels are 10 MHz wide as compared to 20 MHz in 802.11a to counteract with Doppler effect (a change in frequency when data is delivered and received from a moving source) and with ISI (Inter Symbol Interference) induced by multipath fading. A 5 MHz guard band is also introduced to avoid interference with nearby radio technologies. In IEEE 802.11p,

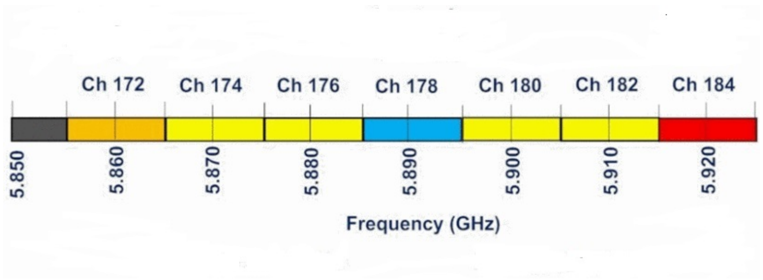


Figure 2.2: WAVE channels .Taken from [17]

the data rate ranges from 3 Mb/s to 27 Mb/s depending on the used modulation and coding rate, while in 802.11a the maximum data rate is twice with respect to 802.11p i.e.

$27 \times 2 = 54 Mb/s$; this is due to the fact that, in IEEE 802.11p, all the timing parameters are doubled to counteract with the multipath fading effect.

IEEE 802.11p also introduces some improved requirements for the receiver performance for cross channel interference that was not present in the 802.11a standard [29].

2.1.2 WAVE MAC layer

IEEE WAVE MAC layer encompasses 802.11p and some additions defined by the IEEE 1609 group. In the 1609.4 standard, WAVE MAC layer provides:

1. Upper MAC layer: it provides a channel access scheme
2. Lower MAC layer: it provides channel coordination

Channel Access

In IEEE 802.11 there are three kind of service sets: Basic Service Set (BSS), IBSS (Independent Basic Service Set) and ESS (Extended Service Set); however, in IEEE 802.11p a new mode has been defined, called OCB (Outside the Context of BSS) in order to support the timing constraints of VANETs, in which there are no authentication and association steps in order to reduce message latency. The stations in OCB mode can transmit frames without forming or being a member of any BSS [17].

The channel access is based on a variant of DCF (Distributed Coordination Function) called EDCA, supporting different priority levels, with the aim of allowing the provision of traffic with different quality of services.

This mechanism is based on IEEE 802.11e and it has been also applied to 802.11p.

EDCA

It is a scheme used by the WAVE MAC layer for channel access based on CSMA/CA. EDCA allows the transmitter to differentiate between different data priorities coming from the upper layers and from different applications that have different requirements. This can be done by using different priority queues inside the MAC layer. High priority data will go to high priority queues and low priority data into low priority queues. There are four ACs (*Access Categories)* defined by EDCA corresponding to four different priority levels. These are:

1. AC_BK: defined for background data with a very low priority

2. AC_BE: defined for best effort traffic, it is normally used as the default one, atleast under Linux as mentioned in [3].

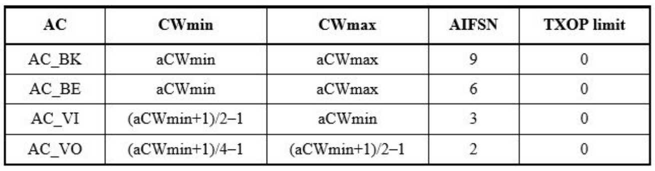
3. AC_VI: defined for video

4. AC_VO: defined for the voice data with the highest priority

These *Access Categories* names are not used to characterize the data, but they rather refer to priority levels. The data comes out from these 4 queues by having internal contention in a sort of embedded network inside the wireless card in which these 4 queues act as stations. Each of these stations perform, typically, DCF operations to access the channel but with different values of some crucial parameters, as shown in Figure 2.3.

According to the DCF scheme the stations sense the channel for a certain time before declaring it as idle: this interval, needed to declare the channel as idle, is called AIFS (*Arbitration Inter-frame Space)* instead of DIFS that was used in 802.11 for channel contention. The value of AIFS depends on the four different ACs, corresponding to four different queues. Higher priority ACs will have a shorter AIFS so that they sense the channel for a shorter time and if the channel is idle they will send their frames on the physical channel with a higher transmission probability with respect to low priority ACs, which have to listen to the channel for more time, before declaring it as idle.

AIFS is the amount of time data frames wait before they can try to access the physical channel, higher priority queues will have a shorter AIFS and lower priority will have higher values. AIFS is defined in terms of AIFSN $*$ time slot; whereas the time slot in IEEE 802.11p is 13 micro seconds [31]. Each AC has different transmission parameters as described in Figure 2.3:



AC	CWmin	CWmax	AIFSN	TXOP limit
AC_BK	aCWmin	aCWmax	9	0
AC_BE	aCWmin	aCWmax	6	0
AC_VI	(aCWmin+1)/2–1	aCWmin	3	0
AC_VO	(aCWmin+1)/4–1	(aCWmin+1)/2–1	2	0

Figure 2.3: WAVE access categories.Taken from [32]

The transmission parameters of Access Categories are:

- **CW_MIN, CW_MAX**: the contention window is the range of integers from which we can draw a random value to perform the Back Off (BO) procedure. The BO procedure is performed when any station receives a packets from higher layers to be transmitted, but the channel is busy, in this case the stations has to wait for a random time called ”Backoff interval” computed by multiplying this random integer with the slot time, before it can contend the channel again, in order to avoid collisions.

  The contention window minimum and maximum values define the range from which the BO value is extracted, for each access category. The higher priority ACs will extract from a set of smaller values so that they will send their frame with larger probability while lower priority ACs have a larger set of values; therefore higher priority ACs will likely transmit before lower priority ones. In case of collision the CW is doubled so that the probability of further collisions is reduced as in DCF.

- **TX_OP**: (Transmission Opportunity) it defines the number of maximum frames that can be sent without re-contending the channel; if its value is zero, only one frame can be sent by the station [33].

- **AIFSN**: it defines the number of slots after a SIFS (*Short Interframe Space*) before a channel can be declared idle.

All of these 4 queues are contending internally when we have data from different ACs and then there is external contention; in the internal contention these collisions among different stations are virtual, only the stations winning the internal contention will be able to contend the physical channel and transmit their data frames, in which they might collide with another frames transmitted over the same frequency and on the same channel.

Channel coordination

The upper MAC layer is defined by the IEEE 1609.4 standard and it defines the multi-channel operations, channel routing and management services. It specifies on which channel the data should be transmitted, as in a vehicular environment there is no BSS and the vehicles are not coordinated; in fact, if they have only one transceiver, they have to choose on which channel to transmit among 7 different physical channels. For this reason the channels are defined as CCH and SCH:

- **Control Channel**: it is used to advertise different services available on service channels.

- **Service Channels**: they actually provide those services advertised by control channel.

If there is only one transceiver, as foreseen by the IEEE standards, only one channel can be accessed at a time and for this reason IEEE 1609.4 defines different switching modes among these channels, defining a time slot, which in this case is not the 802.11 one and it is equal to 50 ms.

- **Continuous channel access**: in which we always listen to one channel
- **Alternative channel access**: in which every time slot the station switches between CCH and SCH.
- **Immediate channel access**: in which the station can switch to any channel at any time.

According to the IEEE standards each WAVE card should have two implementations of the MAC layer: one for CCH and another one for the SCHs, and each of them should have 4 queues for the EDCA mechanism, as shown in Figure 2.4.
CCH carries only non-IP messages, while SCHs carries both IP and non-IP data. It is a task of the 1609.4 standard to decide which of these MAC layer should be used for transmission. This operation is called channel selection.
Non-IP packets are packets coming from other vehicles and end up on both SCH and

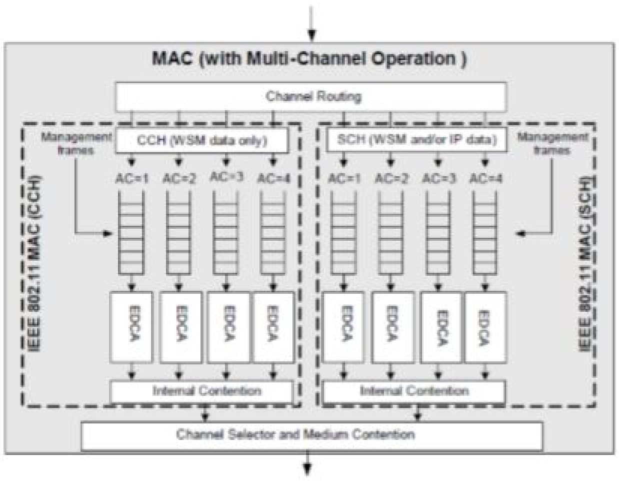


Figure 2.4: MAC layer channel selection Taken from [34]

CCH. The SCH and CCH are used to carry different kinds of data: SCH is used for non crucial information, while CCH carries instead crucial information, advertisements and alerts.

Time synchronization

An important time reference is coming from GPS that provides UTC in *hhmmss.ss* (hours minutes seconds) format that will be explained in chapter 4. There is a 2ms tolerance on the obtained UTC values.

With the start of every UTC second, the channel coordination time slot also starts, the guard interval is introduced to ensure that there is no misalignment between the slots, in different vehicles for channel coordination. So GPS is an important element in VANETs and it is used for positioning and time synchronization. All the synchronization features are based on a common time reference, which is UTC (Universal Coordinated Time) time modulo 1 second.

2.1.3 WAVE networking

It is defined by the 1609.3 standard. The stack is divided into two planes to implement two different functionalities: one is the data plane to carry data and the other is used to provide the management services such as IPv6 configuration and WSAs (WAVE Service Advertisements) monitoring, so that the data is properly carried according to the standard, and it is called management plane. Data services rely on two protocols at the network layer:

- IP: the standard, in particular, foresees the usage of IPv6.
- WSMP (WAVE Short Messaging Protocol): it is an alternative to the full TCP/IP stack and carries WSAs. Each WAVE service has an allocated PSID (Provider Service ID), broadcasted over the CCH by the provider, the users can then utilize this service if they want and forward the WSMP contents to higher layers.

The **LLC sub-layer** is used to tag the data by using a short field to state from which protocol stack the data is coming as it can come from either IPv6 or WSMP: this enables the receiver to know to which protocol stack it has to send its data. In fact LLC is used for protocol multiplexing functions.

2.1.4 WAVE security

It is defined by the 1609.2 standard. The standard describes the security and privacy services and implements these services independently from the IP or WSMP protocol stack. The standard foresees 5 basic services such as:

- **Authenticity**: set of procedures to clearly identify the sender that is sending data.
- **Authorization**: set of procedures performed to ensure that the user is authorized to use a specific service.
- **Integrity**: set of procedures assuring that the packet sent by sender has not been amended.
- **Non-repudiation**: the messages are signed by a third-party CA (Certificate Authority) so that we are sure that the message is sent by trusted users.
- **Confidentiality**: set of procedures to ensure that only the intended recipient can read the contents of the packets.

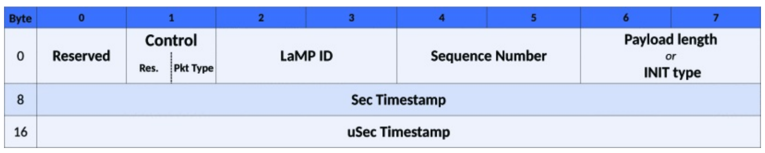
2.1.5 WAVE application layer

The IEEE WAVE application layer is based on a message set defined by the SAE J2735 standard, specified by SAE (Society of Automotive Engineers), which defines a fixed set of messages that are universally recognizable. These messages are of 16 different types including Basic Safety Messages (BSM) that are used as heartbeat to keep notifying the speed and heading of the vehicle to nearby nodes.

2.2 LaMP protocol

Latency is one of the key parameters in vehicular networks or in any other network system. There are many protocols available for latency measurements such as *ICMP* (Internet Control Message Protocol) and *TWAMP* (Two Way Active Measurement Protocol). The `ping` tool, which is commonly used, computes latency by using *ICMP* and by timestamping the packets, but it does not provide end-to-end latency measurements at the application layer as the ICMP reply mechanism is typically implemented in the kernel.

One of the groups in Politecnico di Torino introduced a new protocol called LaMP (Latency Measurement Protocol) to measure the end points latency at the application layer [35]. LaMP can be encapsulated inside any lower layer protocol and it works in a client-server architecture in which a server replies back to the client requests and the client computes the RTT (Round Trip Time) or uni directional latency by using the timestamps inserted inside each LaMP packet. Beforehand a session is established between the client and the server for accurate measurements. LaMP foresees a total of 16 bytes for its header,



Byte	0	1	2	3	4	5	6	7
0	Reserved	Control Res. | Pkt Type	LaMP ID		Sequence Number		Payload length *or* INIT type	
8	Sec Timestamp							
16	uSec Timestamp							

Figure 2.5: LaMP header. Taken from [35]

as shown in Figure 2.5. They are respectively divided into:

- One byte for a reserved header field, to identify a LaMP SDU.
- One byte of a control field, to find out the LaMP packet type (request, reply, acknowledgment or connection initialization)
- 16 bytes used for timestamps, one for a seconds and another for a microseconds timestamp.
- Two bytes for a LaMP ID, used to identify each client server session.
- Two bytes of sequence number for enabling the association between each request and reply and to compute some metrics, such as estimated ”out of order” count.
- Two bytes to store the payload length of the LaMP packet, as LaMP can also be used to carry any arbitrary payload, up to 65535B.

2.2.1 LaTe

LaTe stands for ”Latency Tester” and it is a flexible latency measurement tool developed by the same research group [35] that designed LaMP, following all its specifications.

They validated the tool by performing several tests with LaMP over UDP and IPv4, to measure different types of latency such as user to user (either RTT or one-way latency) and what they called KRT (Kernel Reception Timestamp), in which received packets are timestamped when they are being passed from the hardware to the kernel stack. LaTe also supports the computation of confidence intervals according to the T-Student distribution. LaTe can be used as a basic latency measurement tool for vehicular networks.

Chapter 3

CAN and speed estimation

3.1 Introduction

Before the 80's the electronic devices within the vehicles were connected through point to point wires, making the overall system heavy, complex and expensive. In 1985 Bosch Germany developed CAN (Controller Area Network) to solve the problem of point-to-point dedicated wires between electronic devices and replace them with a full in-vehicle network [36]. This in-vehicle network approach has been widely adopted by many automotive industries and emerged as an international standard known as ISO 11898 [36].

CAN is based on a serial communication two-wire bus, relying on asynchronous communication; it is used to connect all the parts of the vehicle and can reach a throughput of 1 Mb/s. This CAN bus can be used to connect up to 2032 devices in a single network.

There is no addressing for bus nodes and a filtering process can be used to get the relevant data based on CAN IDs. Any data can be sent using CAN frames that are 8 bytes long, atleast in a basic version of standard, with a "Start of Frame" (SOF) bit that will identify the starting of each new frame. In addition to the usage in cars, CAN applications are widespread and include:

- Railways, trams and underground metros
- In aircrafts it is used to connect different sensors and navigation systems in the cockpit
- Marine electronics

- Industry

In this work, we are reading the CAN data from the electric vehicle; each CAN frame has an identifier, which can be used to determine which ECU sent the frame and which information is packed inside. The way in which the data is actually coded depends on the CAN network designer, as each OEM has its own way of coding the transmitted information starting from a CAN network database. In order to decode the information contained inside CAN frames, we could rely on a partial database coming from the producer.

One of the main objectives of using CAN in this work, is to get the estimate of the speed of the car, as it was not directly available from the CAN of the used vehicle. So we decided to read the RPM values from the CAN to infer a possible linear relationship between the measured engine RPM of the car and the actual speed of the car, which we are getting from the GPS as explained in chapter 4.

The aim is, after verifying if this relationship is linear and computing the right coefficients to compute a speed estimation from the RPM values even in the absence of GPS coverage.

To find the relationship between the speed and RPM values, we also introduced a filter, which is explained in more detail in section 3.4. The data received from the CAN of the car (RPM values) and from the GPS (speed values), which normally comes at a different frequency. The use of a filter has the following goals:

- The first goal was to get both of these values at the same frequency so the filtering mechanism is used to downsample the RPM value which was received at a quite high frequency (around 50 Hz) with respect to the GPS, which is sending data at the frequency of 5 Hz.

- The second goal was to reduce the RPM oscillations in order to try to keep the speed of the car as much constant as possible (as it is difficult to maintain the speed of the car manually at one value, for example, at 25km/h) and trying to relate this speed with the speed we read from the GPS.

3.1.1 CAN architecture

A scheme of the ISO 11898 standard which defines the CAN protocol is shown in Figure 3.1.

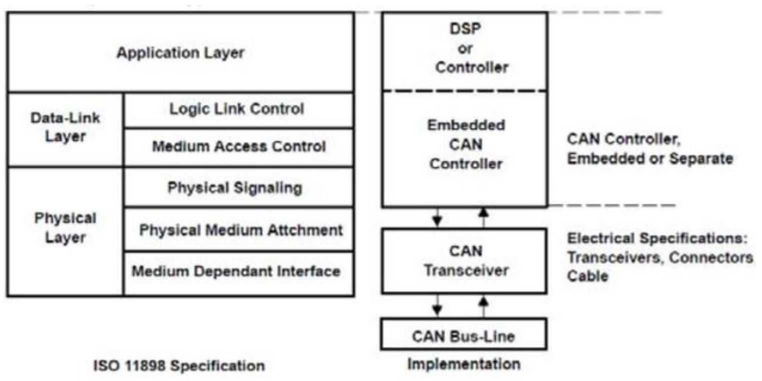


Figure 3.1: ISO 11898 layered architecture. Taken from [37]

It is based on the ISO/OSI model but, instead of seven layers it consists of three main layers that are:

- **Application layer**: it is up to the application developer to define the application layer messages and to defines the CAN message map, link speed and different CAN configurations; the developer also defines the network architecture and type of information that shall be exchanged among ECUs (Electronic Control Units) and the priority of different messages.

- **Data Link layer**: all the operations related to the transmission and reception of CAN frames will be implemented in this layer. It also implements the CAN bus arbitration mechanism, which is a mechanism to prioritize different kinds of data based on the CAN ID and decide which node can transmit when multiple nodes are transmitting at the same time.

  The arbitration field in the CAN frames is used for prioritizing the data. Each data frame in the arbitration field has an associated CAN ID and another field, which is called RTR (Remote Transmission Request), and is used to manage the re transmissions.

  There are four different types of CAN frames defined in the Data Link Layer:

  – **Data frames:** these CAN frames are used to transfer data from one node to another in the CAN network. As already mentioned, there is no addressing mechanism in the CAN protocol, therefore it is up to the node to accept or discard the data it received.

  - **Remote frames:** these frames are used by a node to request the data from another node in the CAN network, hence these frames only carry the request data and do not carry any payload.
  - **Error frames:** When an error in the communication is discovered on the bus, these frames are sent in order to inform all the other nodes to stop their transmissions.
  - **Overload frame:** these frames are used whenever the network is overloaded by any one of the nodes that always tries to send something, so these frames are sent to the transmitter node to slow down its transmission of frames, in order to avoid overloading the CAN network. These frames can only be transmitted during IFS (Inter Frame Space)[1].

- **Physical layer**: it is based on a bus with two wires: CAN_H and CAN_L, which are both terminated with a 120 ohm resistance. The physical layer includes the clock synchronization and bit coding management using NRZ (Non-Return-to-Zero). There are two logic states for the CAN bus:

  - **Dominant state:** when the CAN_H and CAN_L are either at 5V or 0V, the CAN bus will be in the dominant state, when a node is trying to send a dominant value and another node is sending a recessive value, the bus carries the dominant value
  - **Recessive state:** when both CAN_H and CAN_L are at 2.5V the bus will be in a recessive state.

3.2 Reading Data from CAN socket

As explained before we are interested in taking the RPM values from the CAN socket so that we can find out the trend between the speed of the car that we get from the GPS (as explained in chapter 4), and the RPM values.

In order to read the data from the vehicle's CAN network, the first step is to connect with the CAN_H and CAN_L wires. For this purpose we opened the car to identify these

[1]It is used to separate a data and remote frames

two wires in order to make the physical connection of them with a DB-9 connector, as shown in Figure 3.2.

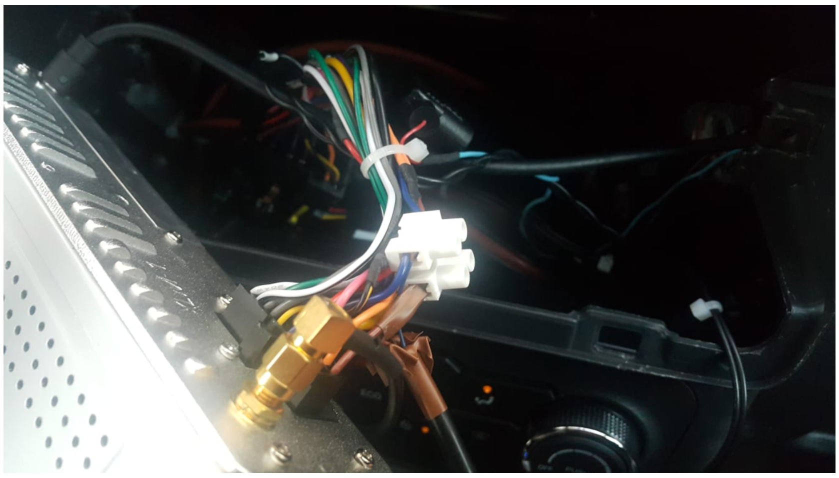


Figure 3.2: CAN bus wires

CAN_H is represented by the blue wire while CAN_L by the brown wire. The two identified cables were then soldered with a DB-9 ports. Now, to get some useful data, the following steps have been performed using *Linux* and relying on basic C socket programming.

A Kvaser Leaf Light device, which is shown in Figure 3.3 was connected to the DB-9 connector mentioned before. We are using a Kvaser device as it provides an USB connection to DB-9 port.

Figure 3.3: Kvaser leaflight device. Taken from [38]

The CAN interface was then set up using the Linux commands mentioned in [39], and shown below:

Listing 3.1: Setting up the CAN interface

```
$ sudo ip link set can0 type can bitrate 125000
$ sudo ip link set up can0
```

The steps for reading CAN data using C sockets programming, as we did in this work, are:

Step 1: In order to get the data first we need to open a socket for the CAN network, in the same way as in TCP or UDP sockets. This can be done by using the `socket()` system call that returns the file descriptor of the newly created socket.

There are three arguments to be passed to this system call in order to read data from a CAN bus: the first argument is PF_CAN and it represents the protocol family of the CAN network, while the second argument defines the type of communication semantics. In particular SOCK_RAW provides the access to the raw network protocol stack. The third argument defines the specific protocol which should be used. At present CAN socket supports two protocols: CAN_RAW and CAN_BCM.

- **CAN_RAW**: the socket in the CAN Raw protocol receives only those frames which have valid data, a single filter can be set in order to receive the CAN frames of interest. It also enables the usage of loopback interface but does not allow the socket to receive the frames that are sent in loopback mode [40], instead it is useful for the analyzer tools such as ”cansniffer” [41].
- **CAN_BCM**: the CAN Broadcast Manager protocol can manage to have a large number of transmissions and receptions based on multiple applied filters handled at the same time. The CAN_BCM protocol provides a command based interface for the filtering and sending of messages in the kernel space.

In our case we will rely on ”CAN_RAW”.

Listing 3.2: **Step 1**: CAN Socket Opening

```
Socket_D = socket(PF_CAN, SOCK_RAW, CAN_RAW);
//check if the socket is open
if (Socket_D < 0)
{
  perror(PROGNAME ":socket ");
  return errno;
}
```

Step 2: After the socket has been created and opened, it should be bound with the right CAN interface by using the `bind()` system call. In order to receive the data from all the CAN interfaces the "can0" interface must be used. Interface indices can be found out by relying on `ioctl()`.

Listing 3.3: **Step 2**: CAN Socket Binding

```
strcpy(ifr.ifr_name, "can0" );
ioctl(Socket_D, SIOCGIFINDEX, &ifr);
addr.can_family = AF_CAN;
addr.can_ifindex = ifr.ifr_ifindex;
bind(Socket_D,(struct sockaddr *)&addr, sizeof(addr))
```

Step 3: CAN filters can also be applied to get specific CAN frames based on the desired CAN IDs. A filter is basically matching when the AND operation on the received CAN ID with some mask bits is corresponding to the AND operation between the specified filter ID and the same mask bits.

This CAN filter is then applied to the socket by using the `setsockopt()` system call this allow us to receive only the required CAN frames based on their ID as shown in the listing below:

Listing 3.4: **Step 3**: Applied filter for RPM values

```
struct can_filter rpmfilter[1];
rpmfilter[0].can_id   = 0x100817EF; //filter id for RPM
rpmfilter[0].can_mask = 0xFFFFFFF;
int enable=1;
setsockopt(Soclet_D,SOL_CAN_RAW, CAN_RAW_FILTER,
&rpmfilter, sizeof(rpmfilter));
```

Step 4: In order to timestamp each received CAN frame, iovec arrays and ancillary data can be used. The full code is available in Appendix A.

Listing 3.5: **Step 4**: Settings of the CAN socket for timestamp

```
if(setsockopt(Socket_D, SOL_SOCKET, SO_TIMESTAMP,
&enable, sizeof(enable))<0)
```

```
{
  perror("std error");
  exit(EXIT_FAILURE);
}
```

Step 5: In order to read the CAN frames, two system calls can be used; one is `read()` and the other is `recvfrom()`. `recvfrom()` should be used when the CAN frames are received from any existing CAN interface.

Listing 3.6: **Step 5**: Reading of timestamped CAN data

```
long *t; //for timestamp values
t=(long*)malloc(size* sizeof(long));
while (sigval== 0)
{
  nbytes = recvmsg(Socket_D, &msg, 0); /*receiving data from CAN
      interface of car */
  for(cm=CMSG_FIRSTHDR(&msg);cm;cm=CMSG_NXTHDR(&msg,cm))
  {
  if (cm->cmsg_type == SO_TIMESTAMP)
    {
    tv = (struct timeval *) CMSG_DATA(cm);
    t[k]=tv->tv_usec ;
    k++;
    printf("timestamp %ld ",tv->tv_usec); //microsecond timestamp
    }
  }
}
```

Step 6: After getting the CAN data from the CAN socket, the first two bytes, depending on the CAN ID and according to the electric vehicle CAN data specifications, contain the RPM value of the electric engine. We can parse them with a code as the one reported below:

Listing 3.7: **Step 6**: RPM values

```
printf("can_id: %X data: ", frame.can_id);
for (int i = 0; i < 2; i++)
{
  if(i==0)
    {
       printf("%X ", frame.data[i]);
       rpm1=frame.data[i];
    }

  else

    {
        printf("%X ", frame.data[i]);
        rpm0=frame.data[i]; //read as 0-1 H-L byte
    }
}
```

Step 7: After reading all the data for the time needed to gather all the measurements, the CAN interface socket can be closed, and the CAN Kvaser Leaf Light cable can be disconnected.

Listing 3.8: **Step 7**: CAN Socket Closing

```
if (close(Socket_D) < 0)
{
  perror(PROGNAME ": CAN socket closed");
  return errno;
}
return EXIT_SUCCESS;
```

3.3 Virtual Controller Area Network

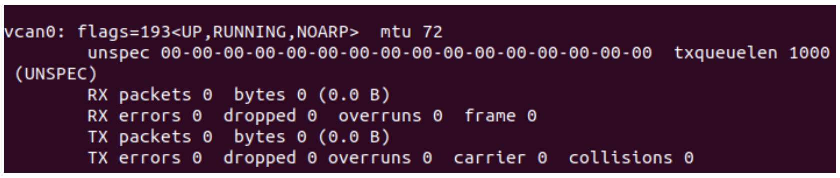
Linux also provides a facility to use a virtual CAN (vCAN) to test the correctness of a code using AF_CAN sockets without the need of the actual hardware. Therefore, when we could not physically access the electric vehicle, our algorithms were tested using the virtual CAN interface. The support for virtual CAN can be enabled using the *can-utils* package that can be installed on Linux using the following command:

```
$ sudo apt-get install can-utils
```

Now a virtual CAN device can be created by using the following commands as mentioned in [42]:

```
$ modprobe vcan
$ sudo ip link add dev vcan0 type vcan
$ sudo ip link set up vcan0
```

After runnning these command on a Linux terminal window, the interface should be up and running as shown in Figure 3.4. It should be called "`vcanx`", where *x* represents the



```
vcan0: flags=193<UP,RUNNING,NOARP>  mtu 72
        unspec 00-00-00-00-00-00-00-00-00-00-00-00-00-00-00-00  txqueuelen 1000
 (UNSPEC)
        RX packets 0  bytes 0 (0.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 0  bytes 0 (0.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

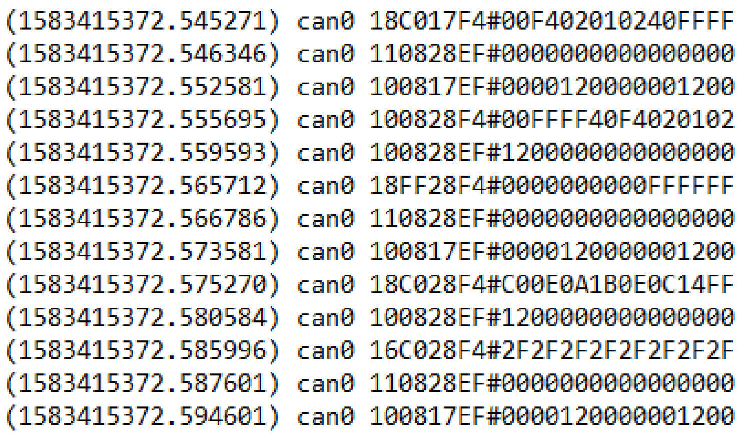
Figure 3.4: Virtual CAN is up and running

number of the interface actually in use (in our case it was called `vcan0`).

To run the code for testing, CAN frames were sent by virtual CAN and received by our CAN socket, which is listening to the `vcan` interface. To dump the CAN traffic, two options can be utilized:

- **"`cangen`"** can be used to generate a random and continuous stream of CAN frames that can be sent to the CAN interface listening in our C program. This random stream can be stopped by pressing ctrl+c.

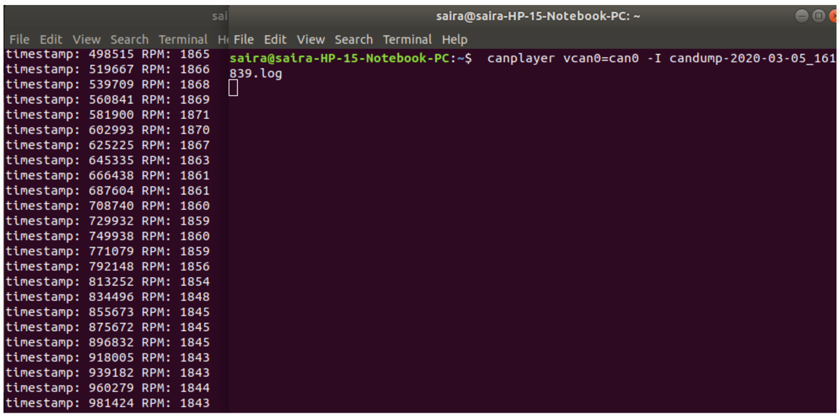
- "`candump -l vcan0`" command is used to record the log file of CAN traffic. Log files can be created by connecting to a real CAN hardware and they can then be used for later testing. Candump files can be recorded with or without timestamps by using ”-t” option, and can be reproduced using the `canplayer` command. For this work a log file was dumped with timestamps and it can be seen in Figure 3.5:



```
(1583415372.545271) can0 18C017F4#00F402010240FFFF
(1583415372.546346) can0 110828EF#0000000000000000
(1583415372.552581) can0 100817EF#0000120000001200
(1583415372.555695) can0 100828F4#00FFFF40F4020102
(1583415372.559593) can0 100828EF#1200000000000000
(1583415372.565712) can0 18FF28F4#0000000000FFFFFF
(1583415372.566786) can0 110828EF#0000000000000000
(1583415372.573581) can0 100817EF#0000120000001200
(1583415372.575270) can0 18C028F4#C00E0A1B0E0C14FF
(1583415372.580584) can0 100828EF#1200000000000000
(1583415372.585996) can0 16C028F4#2F2F2F2F2F2F2F2F
(1583415372.587601) can0 110828EF#0000000000000000
(1583415372.594601) can0 100817EF#0000120000001200
```

Figure 3.5: Log file of the CAN

`canplayer` was used to replay this log file and send the CAN frames to the CAN socket used by our C code. As we were interested in taking the rpm values, our output is filtered based on the CAN ID of the frames containing the RPM values and each value is timestamped as shown in Figure 3.6.


sai
saira@saira-HP-15-Notebook-PC: ~
File Edit View Search Terminal H
File Edit View Search Terminal Help

```
timestamp: 498515 RPM: 1865
timestamp: 519667 RPM: 1866
timestamp: 539709 RPM: 1868
timestamp: 560841 RPM: 1869
timestamp: 581900 RPM: 1871
timestamp: 602993 RPM: 1870
timestamp: 625225 RPM: 1867
timestamp: 645335 RPM: 1863
timestamp: 666438 RPM: 1861
timestamp: 687604 RPM: 1861
timestamp: 708740 RPM: 1860
timestamp: 729932 RPM: 1859
timestamp: 749938 RPM: 1860
timestamp: 771079 RPM: 1859
timestamp: 792148 RPM: 1856
timestamp: 813252 RPM: 1854
timestamp: 834496 RPM: 1848
timestamp: 855673 RPM: 1845
timestamp: 875672 RPM: 1845
timestamp: 896832 RPM: 1845
timestamp: 918005 RPM: 1843
timestamp: 939182 RPM: 1843
timestamp: 960279 RPM: 1844
timestamp: 981424 RPM: 1843
```

```
saira@saira-HP-15-Notebook-PC:~$  canplayer vcan0=can0 -I candump-2020-03-05_161
839.log
```

Figure 3.6: Running code using virtual CAN

3.4 Filtering Results

In this work, one of our main goals was to get the speed of the car through the CAN interface. However, the CAN network we had access to was not providing any frames containing the desired speed values, so as an alternative, we decided to take the RPM values.

The engine RPM values can be converted to vehicle speed but in our case, as mentioned before, we are trying to experimentally infer a relationship between the engine RPM and the longitudinal speed of the car.

It would be very complex to build a full mechanical model of the electric vehicle because we don't know how the mechanical system of the electric vehicle is built and it would require us to know the mechanical insights and different physical quantities of the vehicle which are not known, such as axle ratio and gear ratio.

So we are still not able to analytically calculate the exact speed of the car, but what we expect is that the plot for the RPM values will show the same trend as for the speed, if the relation is really linear.

Figure 3.7 show the main test that we performed in order to gather the data[2] from point A to point B and vice versa, we have to slow down two times during a turn.
As expected during the experimentation, the RPM values were increasing and decreasing

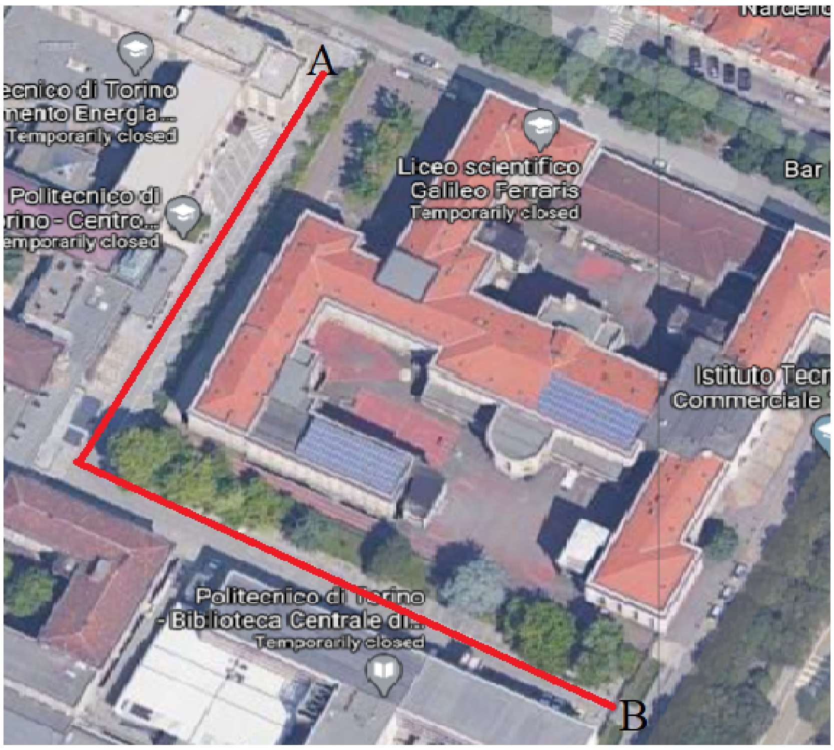

A
B
Liceo scientifico Galileo Ferraris Temporarily closed
Politecnico di Torino - Biblioteca Centrale di... Temporarily closed
Istituto Tecn Commerciale
Bar

Figure 3.7: Path taken by the car during the experiment

quite linearly according to the speed that we were able to keep while performing the test as shown in Figure 3.8.

Furthermore, when the car was stopped, the RPM, as expected, was also zero. In order to confirm that the obtained trend of RPM correlates to the speed, we attached a GPS to the car. This allowed us to obtain a vehicle's speed value which was later compared with the RPM values.

The data being obtained from the car was at a higher frequency (around 50 Hz) with respect to the data obtained through GPS i.e. 5 Hz. The right frequency for the received RPM values is computed through the log file, by using a simple C function based on the CAN ID for the RPM. The frequency of the GPS was instead set manually using its

[2]The same data we are using in virtual CAN

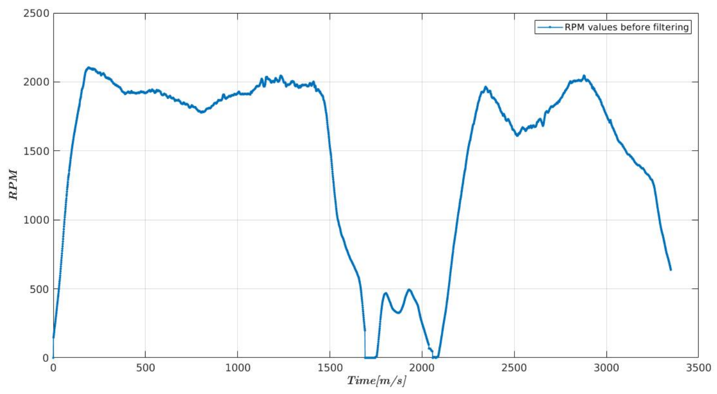


Figure 3.8: RPM graph

software, as described later in chapter 4.

Therefore, we needed to apply a low pass averaging filter so that we have the RPM and speed values at the same frequency. This would allow us to find the relationship between these two values as mentioned earlier.

Another goal of using the LPF was also to reduce the oscillations and to keep the speed of the car constant as much possible, as it was planned to perform tests at different speeds of the car (maintaining one value of the speed at a time in a test such as 25 km/h), get the RPM values and try to perform the linear regression but we were not able to perform this due to the Covid-19 emergency.

3.4.1 Low pass averaging filter

As already mentioned we use an LPF (Low Pass Filter) for averaging over the values and to downsample the RPM data. The values are downsampled with a factor of 11 in such a way that the first 10 values are neglected and every 11th value will be taken into account. In order to mitigate the effect of these missing 10 values the LPF is computing the average of past values. The more values are averaged, the better are the results but we cannot average over more than 10 values as we have to choose each 11th sample.

The factor of 11 is used in such a way that it will downsample the data from a frequency of 50 Hz to approximately 5 Hz.

The data is sent to the LPF in the form of blocks; various block sizes can be used such as (50, 100, 200, 300), the bigger the block size is the better the computational efficiency.

In order to avoid wasting any sample, block sizes multiple of 11 have been used so that all the samples take part in the filter computation. The code implementing the filter is shown in the listing below:

Listing 3.9: Calling the filter

```
{
  if ((i >=   M)) //M is the block size
    {
      i=0;
      xm1 = simplp(x, y, M, tmp,fptr); /* x is the input
      and y is the output array */
      tmp=xm1;
    }

  else if (i<M)

    {
      x[i]=rpm_f; /* each rpm value is saved
      inside the input array */
    }
    i++;

}
M=i;
xm1 = simplp(x, y, M, tmp,fptr); // for the last left block
```

The data taken from the CAN interface, containing the RPM values is sent to the filter in blocks by using the arrays defined in the program. Some delay however is also introduced as the larger the data blocks are the more time we have to wait for gathering these samples. The filter code is shown in the following code listing:

Listing 3.10: Filter code

```
int simplp (int * x, int  * y, int M, int xm1, FILE *fin)
{

 int n;

 y[0] = x[0] + xm1; //xml is the intial value
 y[1]=x[1]+x[0];
 y[2]=x[2]+x[1];
 y[3]=x[3]+x[2]+x[1];
 y[4]=+x[4]+x[3]+x[2]+x[1];
 y[5]=x[5]+x[4]+x[3]+x[2]+x[1];

 for (n=6; n < M ; n++)
  {
   y[n] =  x[n-6]+x[n-5]+x[n-4]+x[n-3]+x[n-2]+x[n-1]+x[n]; //used for
       avaeraging
   printf("%d\n",y[n] );
   y[n]=y[n]/7; //avearging over past 7 values

   if(z==10) // as we are satrting from zero so 11th value is 10
     {
       fprintf(fin, "%d\n", y[n] ); //writing the results to the file
     }
z++;
  }
return x[M-1]; /* the last value is return
back to the filter calling variable */
}
```

3.4.2 Filter results

The results of the filtered data over the tests mentioned before are shown in Figure 3.9, on which the original RPM values and the filtered RPM values are shown on the same plot.

Figure 3.10 shows the averaged value of nearby samples. It also shows that the values are downsampled and each 11th sample is selected from the given data set.

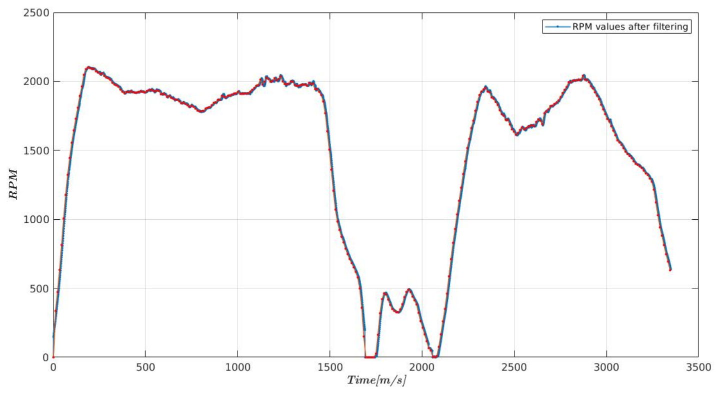


Figure 3.9: RPM values after filtering and downsampling

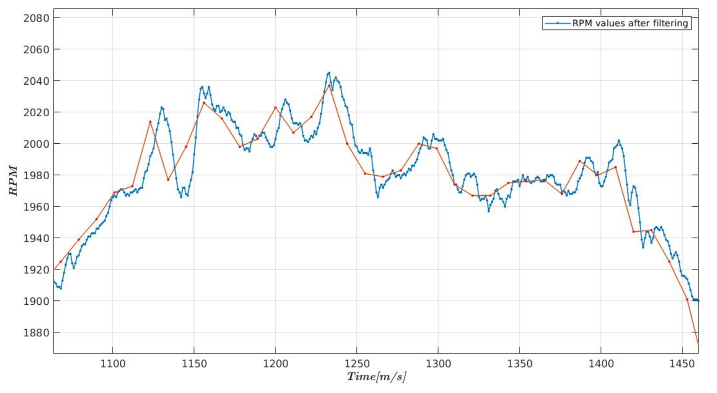


Figure 3.10: Figure shows the averaging of values

Chapter 4

Global Positioning System

4.1 Introduction

In recent days GPS has been widely used in almost all applications for positioning and tracking purposes.

GPS stands for *Global Positioning System*, developed by the United States and operated by the United States Air force. Firstly it was solely used for military purposes but later on it was also made available for civilians. A GPS device is a small handheld device that can be easily placed anywhere. It provides accurate and precise position of the user under any weather circumstances.

The term GPS and GNSS are often used interchangeably. **GNSS** stands for *Global Navigation Satellite System*, but these two have one major difference: a GPS receiver receives signals from up to 31 satellites [43], while a GNSS system can opt to receive signals from more than one satellite system[1] including *GLONASS* by the Russian Federation, *Galileo* by the Europeans Union, *GPS* by the United States and *Beidou* by China. So it can receive signals up to four satellite systems and provide more accurate results. GNSS applications are widespread and they include agriculture, transportation, naval usage and mobile phones. A number of use-cases are described below:

- **Automotive**: one of the most common use cases for GNSS receivers is in autonomous driving in which GPS can be used for vehicle navigation systems and to inform other

[1]If it fails to take the data from one satellite system, it can pick the location from another one

vehicles about the current position and speed through wireless communication. GPS is typically used as a data source for the positioning fields of CAM messages, and thus playing an important role in intelligent transportation systems.

- **Mobile Communication**: GPS receivers, embedded in mobile phone devices, are being tested by mobile phone industries to support 911 emergency. Qualcomm also developed a technology named **gpsOne** for position and location based services using wireless networks [44].
- **Marine and Aviation**: it is used both for marine and aviation navigation, increasing the level of safety and efficiency by providing efficient traffic routing and mapping [45].
- **GPS Time Synchronization**: in addition to precise and accurate positioning it also provides accurate timing and hence it can be use to synchronize any kind of device in which the clock synchronization is a requirement.

Due to its several applications in different sectors, European Companies urge different manufacturers, researchers and system integrators to have a big GNSS downstream market. According to [46], up to 2023, revenues will be very high for this industry as the shipment of GNSS receivers are expected to be doubled, increasing revenue up to 20 billion euros.
In this work the major goal of using GPS is to get the speed of the car in order to find the relationship between the speed and RPM values as already mentioned in chapter 3. The use of GPS has the following goals in our work:

- To get the speed of the car as already stated in chapter 3, for the comparison of speed and RPM values.
- To get the Latitude and Longitude values and calculate the distance covered by the car, other than for logging its position.

4.1.1 GPS working principle

GPS satellites are orbiting around the Earth in such a way that, in order to have three position coordinates (latitude, longitude, altitude) and time, we need to have at least four satellites in the direct LOS (Line Of Sight). However, the more satellites are used, the better is the accuracy.

These GPS satellites are equipped with an atomic clock and are continuously broadcasting the navigation data in the form of radio signals that are received by our navigation devices (GPS, GNSS receivers) with an accurate timestamp from the atomic clock of the satellite. By computing the difference in the timestamps (at which the signal is sent and received by our device), the distance of the receiver with respect to the satellite is calculated by using the distance formula:

$$Distance = Speed \cdot Time \tag{4.1}$$

The speed represents the speed of light[2], and using this distance from each satellite, the position of the receiver is figured out with a potential error of around 10 meters, thanks to the trilateration method, in which each satellite will draw circle based on the above calculated distance and then intersection point of all these circles represents the location of the GPS receiver.

The GPS receivers capture the radio signals and act as *Listeners* while the satellites are acting as *Talkers* (each navigation system has an associated talker-ID). These radio signals are then converted to particular sentences by the receiver, followed the protocol described in the NMEA-0183 standard, and, using the serial data interface, they can be transferred to a connected computer.

4.2 NMEA-0183

This standard was first published in 1983 and it has been updated many times. All kind of GPS and GNSS receivers use the NMEA-0183 standard[3] to transmit information about position (Latitude, Longitude), timing, velocity and so on.

NMEA stands for *National Marine and Electronic Association,* based in the United States. It is a non-profit association with a task to define standards and protocols for communication between marine devices [47]. This standard enables the exchange of information between marine devices, and it also made standard for GPS receivers.
It is important to specify that:
The full copy of the NMEA-0183 standard is present on their official website [48] for

[2] 3×10^8 m/s

[3] Data format followed by worldwide GPS manufacturers

purchasing, however we are following the specifications that came along with our device manual of the respective manufacturer.

4.2.1 NMEA-0183 Data Format

NMEA-0183 uses a serial data interface to connect with the GPS devices. It also provides the compatibility with RS232 serial port protocols, however it is not RS232 and it complies to *"EIA-422"*.

This standard provides a higher speed[4] and better noise immunity, compared to the previous RS-232C standard [49].

The interface speed foreseen by the NMEA standard is 4800 bits per second or 4800 baud rate with no parity bits, but in our device, the baud rate can be adjusted up to 115200 bits per second.

GPS receivers capture data in the form of self contained sentences, in which each sentence show different information. Each sentence is 82 bytes long in which each byte represent one character and uses the ASCII encoding, according to NMEA-0183 standard.

The data within a sentence are separated by commas. Sentences start with '$' and end with <CR><LF> (Carriage Return/ Line Feed). Following the '$' symbol there are two letters representing the talker-ID, telling from which navigation system we are receiving the data, as shown in Table 4.1:

Talker-ID	**Used Navigation Systems**
GP	GPS
GL	GLONASS
GN	GPS+GLONASS

Table 4.1: Talker-ID of Navigation Systems

In this work our device was configured to use both GPS and GLONASS, so the talker-ID of our sentences is **GN**. After the talker-ID there are three letters that describe the type of information contained within each sentence, as shown in table 4.2. Each sentence also contains a checksum value at its end, represented in hexadecimal numbers (0-9)(A-F). A checksum can be optionally computed and it does not include the '\$' and '$*$' characters in its computation. All the values are separated by "," and if a sentence doesn't have any

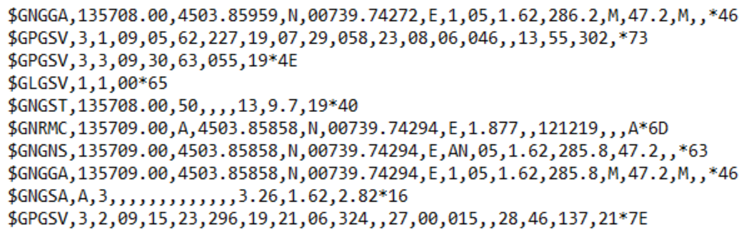
[4]High data rates up to 10 MB/s

valid information or if there is no data, it just shows delimiters ",," without putting any data or zeros in between.

Sentences type	Specifications
GNS	Fix information gives position and time
GGA	Same as GNS
GLL	Longitude/Latitude and Time
GSA	No.of satellites involved for this data and DOP[5] values
GSV	No.of satellites in view, Elevation,Azimuth,SNR
RMC	Speed over ground in knots ($1\,knot = 1.852\,km/h$)
VTG	Speed over ground in knot and km/h.
ZDA	Time (Local TimeZone) and Date
GST	Standard deviation of latitude, longitude and altitude error

Table 4.2: Type of sentences

The Figure 4.1 shows different sentences coming from the GNSS device. The sentences that do not have any data are represented by ",," as mentioned before, and the checksum value of some sentences are shown after '$*$'. The detail of all these sentences will be explained later in this chapter.



```
$GNGGA,135708.00,4503.85959,N,00739.74272,E,1,05,1.62,286.2,M,47.2,M,,*46
$GPGSV,3,1,09,05,62,227,19,07,29,058,23,08,06,046,,13,55,302,*73
$GPGSV,3,3,09,30,63,055,19*4E
$GLGSV,1,1,00*65
$GNGST,135708.00,50,,,,13,9.7,19*40
$GNRMC,135709.00,A,4503.85858,N,00739.74294,E,1.877,,121219,,,A*6D
$GNGNS,135709.00,4503.85858,N,00739.74294,E,AN,05,1.62,285.8,47.2,,*63
$GNGGA,135709.00,4503.85858,N,00739.74294,E,1,05,1.62,285.8,M,47.2,M,,*46
$GNGSA,A,3,,,,,,,,,,,,,3.26,1.62,2.82*16
$GPGSV,3,2,09,15,23,296,19,21,06,324,,27,00,015,,28,46,137,21*7E
```

Figure 4.1: Output sentences from GNSS device

4.3 GNSS Receiver

The GNSS receiver, which is used in this work, is produced by Navilock (*USB 2.0 Multi GNSS Receiver u-blox 8),* with the following specifications, as mentioned in the U-blox Product Summary [50]:

[5]It stands for Dilution Of Precision, it states the effect of measurement errors in the final estimation

- This receiver is based on a u-blox 8 chip set (UBX-M8030-KT) with a built in antenna providing a maximum sensitivity up to 167 dbm and a USB 2.0 Type-A male connector.
- It can receive signals from up to 72 satellites.
- The needed power supply is 5V DC that can be provided from USB, and it has a current consumption of 45 mA.
- The maximum update frequency is 18 Hz when only GPS is configured, and it must be lowered down to 10 Hz when using both GPS and GLONASS.
- It supports baud rates up to 115200 bps, but the default value is, however, 9600 bps for this GNSS receiver.
- It can operate in a temperature range from -20° to 60°.

4.3.1 GNSS Sentences

There are various type of sentences that we can get from a GNSS receiver, which are briefly described in table 4.2. However, the main sentences we used in our work for different computations are: ”GNGNS” and ”GNRMC”. These sentences are described in more details in the next subsections.

GNGNS

It contains fixed GNSS data that provides latitude, longitude and timing according to the local time zone. For instance, we can have a look at an example:
`GNGNS,135659.00,4503.85938,N,00739.74133,E,AN,05,1.62,288.8,47.2,,*63`
This sentence contains the following information, that are explained one by one:
GNGNS, UTC, latitude, latitude direction, longitude, longitude direction, mode indicator, no.of satellite, HDOP, antenna altitude, geoidal separation in meters, age of differential data, reference station ID and checksum.

- ***UTC:*** stands for Coordinated Universal Time, it shows the timing in ”hhmmss.ss format”. For instance, looking at the example we have: hour=13, minutes=56 and seconds=59

- ***Latitude:*** after timing, we have a value of latitude which is equal to 45 degrees, 03 minutes, while 0.85938 is the decimal part of the minutes.

- ***Latitude direction:*** N show that it is in the North direction.

- ***Longitude:*** it is provided in the same format as latitude.

- ***Longitude direction:*** it is related, in this case, to the East direction.

- ***Mode Indicator:*** the first character is for GPS and the second for GLONASS. A means ”Autonomous” [6] and N, means ”no valid fix”.

- ***No. of Satellites:*** it tells number of satellites that are in use for calculating the longitude/latitude values, in this case 5 satellites are in use.

- ***HDOP:*** acronym for *”Horizontal Dilution Of Precision”*. Factor for relative[7] accuracy of a horizontal position and can be computed by all the satellites that are in use. It defines the effect of errors and determine the precision in the receiver measurements. The error should be small, so that the changes in the measurements does not show a complete change of location.

- ***Antenna Altitude:*** it shows the height of the antenna from mean sea level: in this case its value is 288.87 meters

- ***Geoidal separation in meters:*** whose value is 47.2 in meters. It is computed by taking the difference of the mean of the sea level and the ellipsoid surface of the Earth, which is the approximated geoid because of its irregular shape as shown in Figure 4.2.

- ***Age of differential data:*** it will take a null value if the GN is used as talker-ID.

- ***Reference station ID:*** which will be null in case of GLONASS and represented by ”,,” in the example sentence above.

[6]Autonomous means that the satellite system is in non-differential mode (which means that no ground-based system is used for error corrections)

[7]Relative with respect to one satellite to another

- ***Checksum:*** at the end of the sentence, following '∗' there is a checksum value i.e 63. The checksum as already mentioned, is computed over the 80 bytes and it depends on the user whether he wants to check it or not.

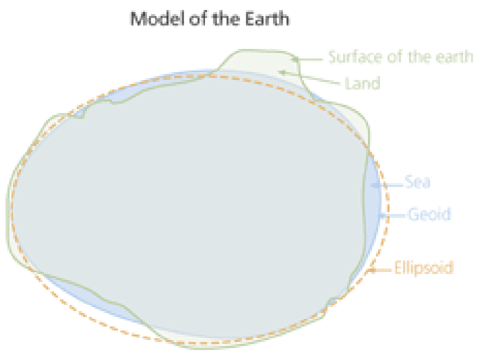


Figure 4.2: Mean sea level is approximated by geoid. Taken from [51]

GNRMC

It stands for Recommended Minimum Navigation Information; the important information that we can get from it is the speed of the object to which GNSS receiver is attached. In this work we use RMC sentences to get the speed of car. An example is shown below:
`GNRMC,135658.00,A,4503.85944,N,00739.74119,E,0.398,,121219,,,A*69`
The information in this sentence are, respectively:

- ***UTC:*** it is representing the time in the same hhmmss.ss format as already mentioned for the GNGNS sentences.
- ***Status:*** it is showing a flag of status which can be either A or V, A means "Active status" and V, which means "Void status"; respectively telling if data is valid or not.
- ***Latitude:*** it is showing the latitude in the same format as described earlier in GNGNS, with the direction of latitude towards North.
- ***Longitude:*** representing a longitude value along with its direction, i.e. East.
- ***Speed in knots:*** it gives the speed over the ground, which is measured in knots. In this case the speed is 0.398 knots.

- ***Track angle:*** it is measured in degrees, and in this case a null value was transmitted.
- ***Date:*** it shows the date in a ”ddmmyy” format.
- ***Magnetic variations:*** it is also computed in degrees, it is the value that can be added or subtracted in the magnetic compass to determine the true North direction. It depends on the location.
- ***Checksum value:*** checksum is computed for the whole sentence (its value is 69, in this case).

The other sentences provided by a GNSS receiver, are briefly described below:
GNVTG: in these sentences, the useful data that can be extracted is speed over ground both in kilometers per hour and in knots. In the first two fields it shows the course over ground in degrees either True or Magnetic.
GNGGA: in these sentences, GGA yields almost the same information as GNGNS sentences such as time, date and longitude, latitude values. However, it contains the GPS quality indicator that tells whether the position fix is valid or not.
GNZDA: these sentences are used to provide the date and time including the day, month and year. Local zone hours and minutes are also present in ZDA sentence.
GNGSV: these sentences give the information about the number of satellites that are used, however, 4 satellites must be used in order to get reliable information such as: elevation, azimuth and SNR (Signal to Noise Ratio) are also present.
GNGST: these sentences are used to give information about the error statistics of position and state the standard deviation of latitude, longitude and altitude error and indicate the order of accuracy of the position information.

4.4 Installing GNSS Software

Our GNSS receiver came with a packaging including the device and a compact disk with windows drivers, user manual and an evaluation software named ”U-Center”. This software is only available for the Windows Operating System. Using this software, the device settings can be changed. The main steps to use the GNSS software for our purposes, as mentioned before, include:

- Using a compact disk, drivers were installed in the Windows Operating System.

- The GNSS receiver was connected with the PC. Then using the software, we selected the right COM port. After selecting the right COM port the device was connected for communication with the computer as shown in Figure 4.3.

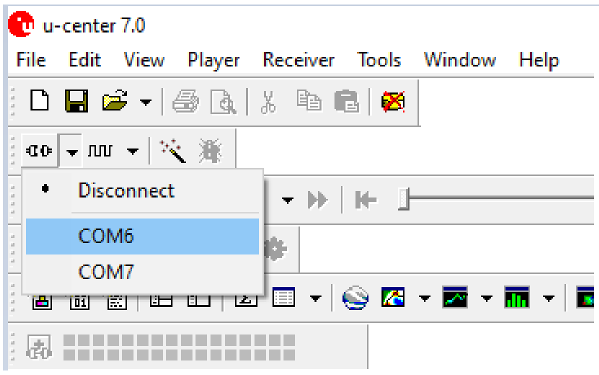


Figure 4.3: Select the right COM port

- Then the baud rate was selected, the most commonly used is 9600 but we can select up to 115200, as shown in Figure 4.4.

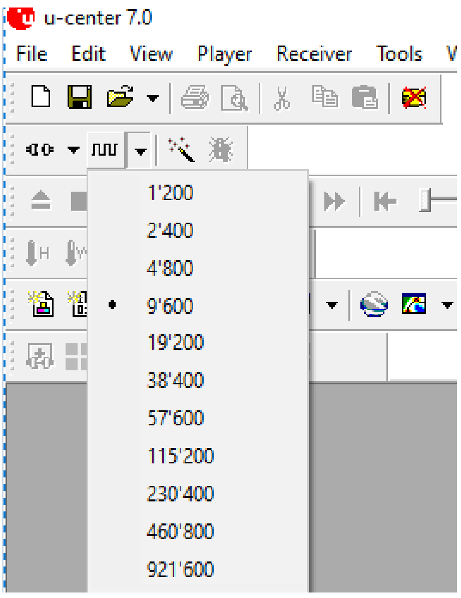
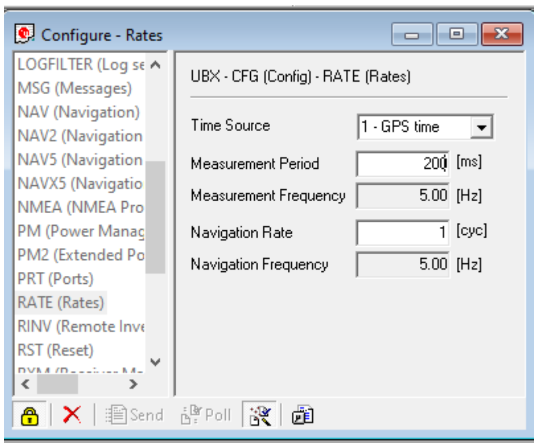


Figure 4.4: Select the Baud Rate

- The rates at which the device updates its information can be changed by clicking on View $-$ > Configuration View $-$ > Rates as shown in Figure 4.5.



UBX - CFG (Config) - RATE (Rates)

Time Source 1 - GPS time

Measurement Period 200 [ms]

Measurement Frequency 5.00 [Hz]

Navigation Rate 1 [cyc]

Navigation Frequency 5.00 [Hz]

Figure 4.5: Frequency setting of getting GPS data

4.4.1 Configuring a GNSS receiver in Debian and OpenWrt

In this work the GNSS receiver was configured to work both with Ubuntu (18.04 LTS) and the OpenWrt Linux distribution, that was installed on the APU1D boards.

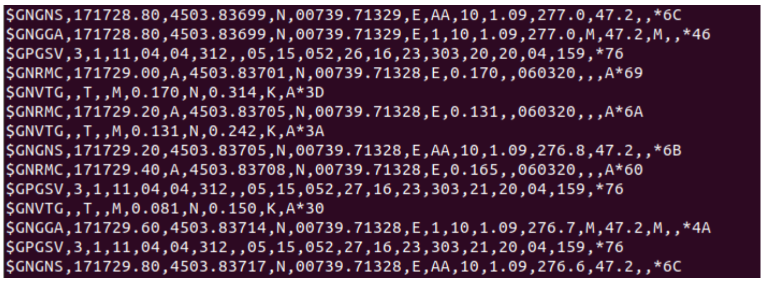
Debian

In order to configure the device to work with a Debian distribution focusing on Ubuntu, first we checked at which COM port it was attached, by using **'dmesg'**.
The following bash script was then used:

```
#!/bin/bash
sudo stty -F /dev/ttyACM0 9600
sudo cat /dev/ttyACM0
```

The `stty` command was used to set the terminal settings so that it can get the data from the specified device file instead of getting it from the standard input. ”F” was used to specify the device file name and 9600 is the baud rate at which we should be receiving the data from the GNSS.

The GPSD daemon can also be used to receive the sentences from a GNSS receiver. The result of the reception of the GNSS sentences in a Debian OS is shown in Figure 4.6:



```
$GNGNS,171728.80,4503.83699,N,00739.71329,E,AA,10,1.09,277.0,47.2,,*6C
$GNGGA,171728.80,4503.83699,N,00739.71329,E,1,10,1.09,277.0,M,47.2,M,,*46
$GPGSV,3,1,11,04,04,312,,05,15,052,26,16,23,303,20,20,04,159,*76
$GNRMC,171729.00,A,4503.83701,N,00739.71328,E,0.170,,060320,,,A*69
$GNVTG,,T,,M,0.170,N,0.314,K,A*3D
$GNRMC,171729.20,A,4503.83705,N,00739.71328,E,0.131,,060320,,,A*6A
$GNVTG,,T,,M,0.131,N,0.242,K,A*3A
$GNGNS,171729.20,4503.83705,N,00739.71328,E,AA,10,1.09,276.8,47.2,,*6B
$GNRMC,171729.40,A,4503.83708,N,00739.71328,E,0.165,,060320,,,A*60
$GPGSV,3,1,11,04,04,312,,05,15,052,27,16,23,303,21,20,04,159,*76
$GNVTG,,T,,M,0.081,N,0.150,K,A*30
$GNGGA,171729.60,4503.83714,N,00739.71328,E,1,10,1.09,276.7,M,47.2,M,,*4A
$GPGSV,3,1,11,04,04,312,,05,15,052,27,16,23,303,21,20,04,159,*76
$GNGNS,171729.80,4503.83717,N,00739.71328,E,AA,10,1.09,276.6,47.2,,*6C
```

Figure 4.6: NMEA data from a Debian based Linux distribution (Ubuntu 18.04 LTS)

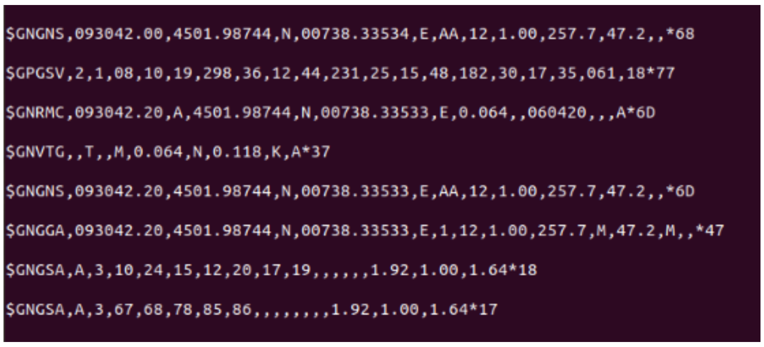
OpenWrt

OpenWrt was installed on the APU1D boards. The main purpose of configuring GNSS with these boards, as already mentioned in chapter 3, that was needed to get the speed and distance from the GPS device. The speed values are used to make a linear regression with the RPM values provided by the CAN bus of the electric vehicle used in this work. While the distance calcualtion is used for performing many tests as described later on in chapter 5. On these boards the IEEE 802.11p protocol was already implemented. In order to make the GNSS device work with the OpenWrt operating system, certain packages need to be installed from the main OpenWrt package respository. These packages are:

- kmod-usb-core
- kmod-usb2
- kmod-usb-serial-pl2303
- `stty` was not working in OpenWrt, so to make it work the **coreutils-stty** package has been installed

`opkg install` was used to install all the aforementioned packages (on these boards internet was not available so these packages were firstly downloaded on a development PC and then transferred to OpenWrt using the `scp` command for secure transfer of data).

Once all the necessary packages were installed, it was possible to use the same commands listed before to listen to an ACM port, and as before, it was possible to read the GNSS receiver NMEA sentences. These sentences are shown in Figure 4.7 :



```
$GNGNS,093042.00,4501.98744,N,00738.33534,E,AA,12,1.00,257.7,47.2,,*68

$GPGSV,2,1,08,10,19,298,36,12,44,231,25,15,48,182,30,17,35,061,18*77

$GNRMC,093042.20,A,4501.98744,N,00738.33533,E,0.064,,060420,,,A*6D

$GNVTG,,T,,M,0.064,N,0.118,K,A*37

$GNGNS,093042.20,4501.98744,N,00738.33533,E,AA,12,1.00,257.7,47.2,,*6D

$GNGGA,093042.20,4501.98744,N,00738.33533,E,1,12,1.00,257.7,M,47.2,M,,*47

$GNGSA,A,3,10,24,15,12,20,17,19,,,,,,1.92,1.00,1.64*18

$GNGSA,A,3,67,68,78,85,86,,,,,,,,1.92,1.00,1.64*17
```

Figure 4.7: NMEA data as printed from OpenWrt

4.5 Distance and speed calculation from NMEA sentences

As mentioned earlier, the NMEA sentences used in this work are: ”GNGNS” and ”GN-RMC”. ”GNGNS” is used to get the longitude and langitude values for the distance calculation while ”GNRMC” gives the direct speed after parsing the sentence.

4.5.1 Distance calculation

After getting data from the GNSS receiver, some post processing was done in C language. Latitude and longitude values were extracted from GNGNS sentences. In the code we developed, these values were firstly converted into decimal degrees and then to radians to put them in the so called Haversine formula, to calculate the distance between two points on Earth [52]. The following trignometric equations are used to calculate the distance between two points:

$$a = sin^2(\Delta\phi/2) + cos\phi1 \cdot cos\phi2 \cdot sin^2(\Delta\lambda/2) \tag{4.2}$$

$$c = 2\,\mathrm{arctan}\,2(\sqrt{a}, \sqrt{1-a}) \tag{4.3}$$

$$d = R \cdot c \tag{4.4}$$

ϕ is used to denote latitude, λ is used for longitude. $\Delta\phi$ is used to denote the difference between the latitude of the two points, between which we are interested to compute the difference. Likewise, $\Delta\lambda$ is used to calculate the difference of the two longitude values. ”R” represents the Earth radius i.e. 6,378,137 meters. The function used for distance calculation is shown in the listing below. The full code is available in Appendix B.

Listing 4.1: Function for calculating distance from longitude and langitude values of two points

```
double distance(double lat_1, double long_1, double lat_2,double
    long_2)
{
  double R = 6371e3; /* in meters and so the distance will also be in
      m/s */
  double latitude1,latitude2,longitude1,longitude2;
  latitude1=lat_1;
  latitude2=lat_2;
  longitude1=long_1;
  longitude2=long_2;
  latitude1=toRadians(latitude1);
  latitude2=toRadians(latitude2);
  longitude1=toRadians(longitude1);
  longitude2=toRadians(longitude2);

  double dlong = longitude2 - longitude1;
  double dlat = latitude2 - latitude1;
  double ans = pow(sin(dlat / 2), 2) +
  cos(latitude1) * cos(latitude2) *
  pow(sin(dlong / 2), 2);
  ans = 2 * asin(sqrt(ans));

  ans = ans * R; //ans variable represents the distance

    printf("distance is %lf\n",ans);
  return ans;
}
```

4.5.2 Speed from GNSS receiver

As we already discussed in the beginning of this chapter, we can retreive the GNRMC sentences from the GNSS device, that provide the speed in knots over the ground. By using the post processing code, first we get the GNRMC sentences and then split the string to extract the speed data, converting it from knot to meters per second by multiplying the extracted value by 1.852. By relying on this procedure, we can retrieve a measurement of speed, which was not possible to obtain from the CAN network interface. These speed values had the aim of helping us to determine their relation with the RPM values that were extracted using the CAN network interface in chapter 3. Though we were able to get both the RPM values and speed from GPS separately, we were not able to perform tests in a real time scenarios with GPS and CAN data available at the same time, for the comparison among them, due to the COVID-19 emergency. Nevertheless, it was expected to observe a quite linear trend between GPS speed and engine RPM.

Chapter 5

Measurements related to 802.11p system parameters

In this work, all the planned tests are based on different V2I scenarios. These tests can be performed using all the devices and software tools described earlier including the APU boards (used as OBU and RSU receivers), the GNSS receiver for logging distance and speed values, and using the electric vehicle.

All these tests are created to analyze the performance of the IEEE 802.11p protocol in real world automotive scenarios and to investigate the effects of different system parameters such as data rates, the offered traffic and the transmission power. Unfortunately, not all the measurements could be performed due to the prevailing COVID-19 emergency and we were not able to gather and analyze the data in real world scenarios. However, they were, nevertheless, planned and explained in this chapter along with their scripts. Our activity thus focused on test planning and on the preparation and validation of scripts and software tools to perform mobility measurements with 802.11p.

All the tests are planned according to the following scheme:

- All the tests will be performed on APU1D boards, on which OpenWrt (18.06.1) has been installed as already mentioned in chapter 1.

- Two APU1D boards will be used to perform experiments. These APU1D boards, just for the sake of convenience are named as "APU_104" and "APU_105" in order to easily recognized and reference them during the tests.

- The boards are assigned the IP addresses of 10.10.6.104 and 10.10.6.105 respectively for 802.11p connection. In order to connect the boards with the development PC using SSH, the wired Ethernet IP addresses assigned are: 192.168.1.184 and 192.168.1.185 for APU_104 and APU_105 respectively as shown in Figure 5.1.

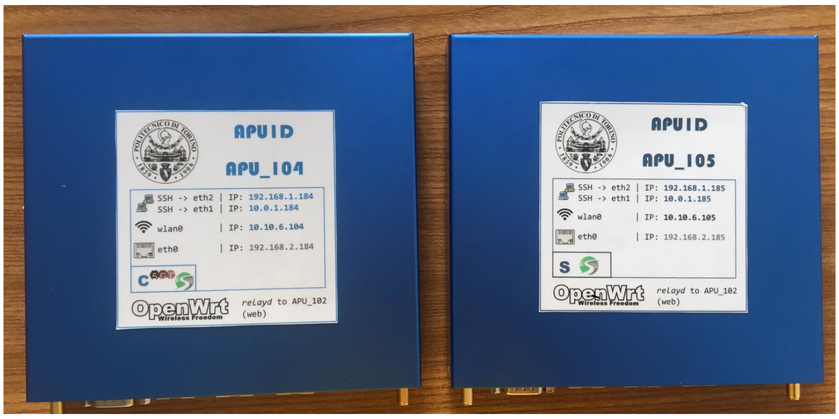


Figure 5.1: APU boards used for testing

- In all the tests APU_104 is used as a server and APU_105 as a client for iPerf and LaTe measurement tools.
- On each of the APU boards, 5 dBi antennas are connected.
- All the tests will be performed over a UDP protocol, as it will report a packet loss measurements that cannot be obtained using TCP.
- The used transmission mode is OCB, according to IEEE 802.11p protocol.
- Each measurement will be performed three times and the final results would be the average of all these three measurements in order to have the reliable results.
- All of these measurements will be performed on the suitable road segments. One of the proposed testing location is shown in Figure 5.2, this path is around 210 meters long.
- The antennas will be placed on top of the car using some strong magnet while the position of RSU can be changed depending on the test being performed.
- All the experiments will be performed with a vehicle speed ranging from 5 km/h to 45 km/h, which is the maximum reachable speed for our electric vehicle.

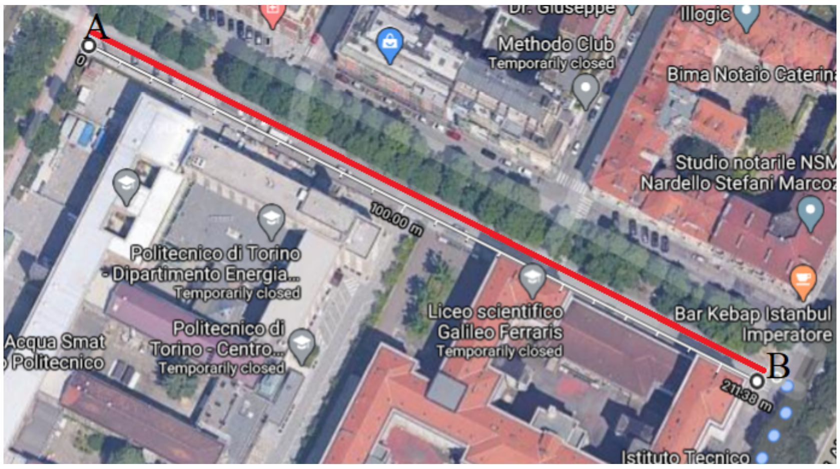

A
B
100.00 m
211.38 m
Methodo Club
Illogic
Bima Notaio Caterina
Studio notarile NSM Nardello Stefani Marco
Politecnico di Torino - Dipartimento Energia...
Politecnico di Torino - Centro...
Liceo scientifico Galileo Ferraris
Bar Kebap Istanbul Imperatore
Acqua Smat Politecnico
Istituto Tecnico

Figure 5.2: Path used for testing

- The measurements will be performed in both directions while moving away from the RSU and in opposite direction i.e. towards the RSU.
- In each test, different parameters will be used that will be changed several times to analyze the different aspects of each scenario.
- Each script will save its output in a ”csv” like format, in order to be able to easily perform analysis on the data later on.
- All the scripts mentioned in this chapter, are tested in a static environment, in order to validate the developed software. Thus all the outputs shown in this chapter have static results.

5.1 Throughput measurements

The first set of tests are planned with the objective to find a set of parameters that are more robust, in order to obtain the largest throughput values for 802.11p and to investigate the results in a mobility scenario such as with respect to the speed and distance. These tests can be repeated by using different physical data rates, over different offered traffic and UDP socket window sizes.

While computing the throughput values with respect to the speed and distance covered by the car, changing system parameters can help us to understand which are the best combination of parameters for a given distance and speed and to experimentally evaluate an 802.11p system on a moving car.

The throughput values are measured using the iPerf measurement tool in UDP mode. The mentioned system parameters can be changed as follows:

- In order to set the bit rates on the boards, we used the command "`iw dev wlan0 set bitrates legacy-5 x`". Here, x denotes the value for a certain bit rate. For example in order to set the bit rate of 3 MB/s, value 6 is used. This is because of the fact that APU1D boards are using a patched version of Linux, in which 802.11p is built upon 802.11a. Due to the patch limitations we have to specify the physical bit rate value as double with respect to what we will physically achieve with 802.11p.

- In order to change the socket buffer size, the `-w` option is used. For UDP the default window size provided by iPerf is `208 KB`.

- In order to change the offered traffic, the `-b` option is used. The default offered traffic over UDP is `1 Mbit/sec`.

The script used for RSU in both tests is shown in Listing 5.1.

Listing 5.1: iPerf client set the values of different data rates, offered traffic and window sizes

```
#!/bin/bash

if [ $# -ne 3 ]; then
echo "Three arguments expected"
echo "1. Bit rate"
echo "2. Offered traffic"
echo "3. Window Size"
exit 1
fi

rate=$1
bw=$2
window=$3

echo "Setting physical layer data rate to $rate Mbit/s at client"
phyrate=$( expr 2 '*' "$rate")
iw dev wlan0 set bitrates legacy-5 $phyrate
```

```
sleep 1

echo "--Running Test with Bitrate $rate, Offered traffic $bw, Window
   $window Client Side--"

iperf -c 192.168.1.184 -u -i 1 -p 5001 -w $window -b $2M
```

The iPerf client will run on the RSU. The test will be repeated many times for different combinations of the mentioned parameters.

The window sizes that can be used for testing purpose are:
`(1K, 2K, 5K, 10K, 20K, 50K, 60K, 80K, 150K and 200K)`
The physical data rates that can be used are:
`(3 MB/s, 6 MB/s, 12 MB/s)`
The offered traffic flow values are:
`(5 MB/s, 10 MB/s, 15 MB/s, 20 MB/s)`

In order to reach the maximum reachable throughput values, the offered traffic must be kept higher then the physical data rates.

As already mentioned, the server will run on APU_104, used as OBU. The tests are designed in such a way that the iPerf server will run on the OBU, and will be placed inside car logging the throughput values over the UDP layer 4 protocol on port 5001 with the default payload size of 1470 B. It will report the values for the whole test duration[1] with the reporting interval of 0.2 sec that is set by using `-i` option.

5.1.1 Throughput with respect to speed of the car

In order to evaluate the throughput of 802.11p with respect to the speed of the car, the following test is planned.

In this experiment two data sources on the OBU need to be synchronized: speed values coming from a GPS and throughput values from the iPerf server. For the synchronization of both of these values, we used the concept of threading in C language, as it is necessary to track the throughput changes as the car moves towards or away from the RSU.

The GPS thread is continuously getting the speed values at a rate of 5 Hz as mentioned into the previous chapters. When the speed value is received, it is also timestamped as shown in the Listing 5.2.

[1]We stop the iPerf, when we stopped the car

Listing 5.2: Part of C code to compare the throughput with the speed of car

```c
pthread_mutex_t mutex;
//initializng the values with 0

float gpsValue = 0.0;
float iperfValue = 0.0;

unsigned long gpsTime = 0;
unsigned long iperfTime = 0;
//GPS thread to read speed values
void *gps(void *arg)
{
  FILE *pipe;
  float var;
  char line1[BUFSIZE];
  pipe= popen("./run.sh","r");
  if (pipe == NULL)
  {
    printf("CANT FIND DEVICE");
    pthread_exit(0);
  }

while (fgets(line1,sizeof(line1), pipe) != NULL)
{

  pthread_mutex_lock(&mutex);
  gpsTime = time(NULL); //timestamping the GPS value


  var = atof(line1); //saving the value coming from GPS sentences

  gpsValue = var;
  pthread_mutex_unlock(&mutex);


  pclose(pipe);
  pthread_exit(0);

}
```

Another thread is used to run the iPerf server to get the throughput values at a frequency of 5 Hz as shown in Listing 5.3. This can be achieved by using interval of periodic reports at 0.2 seconds, so that we can have the two values (speed & throughput) at the same frequency.

Listing 5.3: iPerf thread receiving the throughput values at 5 Hz

```
void *iperf(void * arg)
{
    FILE *fptr;
    float value;
    char line [BUFSIZE];

    fptr=popen("stdbuf -o L iperf -s -u -i 0.2 -p 5001 -f m | stdbuf -
        o L sed 's/\\s\\s*/ /g' | awk '/MBytes/{print $7}'","r"); //
        reading the throughput value from iperf

    if (fptr== NULL)
    {
      printf("server not runnning");
      pclose(fptr);
      pthread_exit(0);
    }
    while (fgets(line,sizeof(line), fptr) != NULL)
    {
      pthread_mutex_lock(&mutex);
      iperfTime = time(NULL);

      value = atof(line);  //throughput value

      iperfValue = value;
      pthread_mutex_unlock(&mutex);

    }

  pclose(fptr);
  pthread_exit(0);
}
```

The measurements are then logged in the main function by comparing the timestamps of both values, so that for each speed value we can have the corresponding throughput value as depicted in Listing 5.4 .

Listing 5.4: Compare the timestamps of values received from two threads and display them

```
int main ()
{

  FILE *file;
  pthread_t tid1;
  pthread_t tid2;
  pthread_create(&tid1,NULL,gps,NULL);
  pthread_create(&tid2,NULL,iperf,NULL);

  printf("after running thread.");
  file=fopen("result.csv","w");

  while(1)
  {
    //comparing the two timestamps
    if ( abs(speedTime - gpsTime) == 0 )
    {
      printf("data %f, speed %f \n", speedValue, gpsValue);
      printf("speedTime %lu, gpsTime %lu \n", speedTime, gpsTime);
    }

  usleep(200 * 1000); //read each value after every 0.2s

  }
  //waiting for both threads to finish
  pthread_join(tid1, NULL);
  pthread_join(tid2, NULL);
  pthread_cancel(tid1);
  pthread_cancel(tid2);
  fflush(stdout);
  fclose(file);

  return(0);
}
```

For communication, user will provide the connectivity parameters that will be pre-negotiated. Once all the arguments are decided, client will try to communicate with the iPerf server.

The expected output of this test is shown in Listing 5.5.

Listing 5.5: Throughput values with respect to the speed of a car

```
throughput 1.060000, speed 0.248000
throughput 1.060000, speed 0.166000
throughput 1.060000, speed 0.115000
throughput 1.060000, speed 0.179000
throughput 1.060000, speed 0.148000
throughput 1.000000, speed 0.112000
throughput 1.060000, speed 0.144000
throughput 1.060000, speed 0.178000
throughput 1.060000, speed 0.233000
throughput 1.060000, speed 0.167000
throughput 1.000000, speed 0.171000
throughput 1.060000, speed 0.214000
throughput 1.060000, speed 0.154000
throughput 1.060000, speed 0.157000
throughput 1.060000, speed 0.345000
```

5.1.2 Throughput with respect to distance

The software tools and script for this test are created in the same way as in the previous section by using threads; however, now in GPS thread we are calculating the distance. The function for calculating the distance is the same as already mentioned in chapter 4.

In the script shown in Listing 5.6, it is shown that after getting the values of latitude and longitude, we can convert them into a decimal degrees format and send it to the distance function. This will compute the distance using latitude and longitude values using Haversine formula. The main purpose of this test are:

- To determine the expected communication range at which a reliable communication can be held between RSU and OBU by using the basic data rate and at different data rates. The communication range is defined as the maximum distance at which packets are received successfully.

- To see the impact of the direction of the car, i.e. how the direction affects the throughput when we are approaching the RSU and going away from it
- Furthermore, we can also place the RSU at different points of the path used for testing.

It is probable that, as the distance increases, the throughput will decrease. However the true results can only be obtained through real experimentation. The expected output should look like this as shown in Listing 5.7.

Listing 5.6: C code to call the distance function for computing distance and compare with the throughput values

```c
 if ( lat1 != 0 && long1 != 0 && lat2 != 0 && long2 != 0)
{
  // this place will run for second reading, because at first reading
     lat2 and long2 will be 0
  double lat1_deg = decimal_deg(lat1);
  double lat2_deg = decimal_deg(lat2);
  double long1_deg = decimal_deg(long1);
  double long2_deg = decimal_deg(long2);
  float res=distance(lat1_deg,long1_deg,lat2_deg,long2_deg);
  //calling the distance function here and saving the distance in res
     variable
  printf("%f\n", res );
  fprintf(fptr,"%f\n",res );
}
```

Listing 5.7: throughput with respect to the distance covered by car

```
throughput 0.000000, distance 0.000000
throughput 1.060000, distance 0.765304
throughput 1.060000, distance 0.765304
throughput 1.060000, distance 0.765304
throughput 1.060000, distance 0.765304
throughput 1.060000, distance 0.765304
throughput 1.060000, distance 0.765304
throughput 1.060000, distance 0.765304
throughput 1.000000, distance 0.796763
throughput 1.060000, distance 0.796763
throughput 1.060000, distance 0.796763
throughput 1.060000, distance 0.796763
throughput 1.060000, distance 0.828722
throughput 1.000000, distance 0.828722
```

5.1.3 Throughput measurements for different access categories

After determining the best set of parameters, the tests presented earlier can be repeated by sending the traffic on different access categories. In particular, to recap the test we are referring to are:

- Throughput with respect to the speed of car
- Throughput with respect to the distance covered by car

The iPerf client will run on the RSU. For the iPerf server the scripts would be the same, however now the client will run the following script as shown in Listing 5.8 instead of the script mentioned in Listing 5.1.

Listing 5.8: iPerf client setting data rates and different access categories

```
#!/bin/bash

if [ $# -ne 1 ]; then
echo "1. Access Category to use : BE,VO,VI,BK"
exit 1
fi

access_c=$1

iperf -c 192.168.1.184 -u -i 1 -p 5001  -A $access_c
```

The output will have throughput values with respect to speed and distance, for each access category.

5.2 Measurements for channel reliability

5.2.1 Throughput, txpower and RSSI measurements

The results of these measurements will be used to assess the performance of 802.11p in real conditions.

These tests help us to determine the maximum transmission power that can be used in order to determine the coverage range, at which we can have the reliable connectivity. The factor of RSSI is also considered so that we can have an ideal 3D matrix based upon the throughput values, receiver sensitivity and transmitted power. This matrix will be created in order to better establish, which combination of the parameters provides a reliable connectivity.

The goal of having 3D matrix of these values is to have throughput values to understand when the channel is reliable or un-reliable based on transmission power and RSSI.

In this testing scenario, the iPerf server will be placed inside the car. And iPerf client will be run on RSU. The OBU will be recording the values for transmission power, RSSI (Received Signal Strength Indicator) and throughput values.

Listing 5.9: Script to run on OBU

```
#!/bin/bash
echo "Starting iPerf server placed inside the car as OBU"
if [ $# -ne 2 ]; then
echo "Two arguments expected"
echo "1. Bit rate"
echo "2. Bandwidth"
exit 1
fi

rate=$1

echo "Setting physical layer data rate to $rate Mbit/s"

phyrate=$( expr 2 '*' "$rate")
iw dev wlan0 set bitrates legacy-5 $phyrate
sleep 1

echo "Reading throughput from iperf server"

stdbuf -o L iperf -s -u -i 1 -p 5001 -t 60 -f m  |  grep 'Mbits/sec'
   --line-buffered |
while IFS= read -r line
do
  throughput=$(echo "$line" |  grep -Po '[0-9.]*(?= Mbits/sec)')
  txpower=$(iw dev |awk '/txpower/{print $2}')
  rssi=$(iw dev wlan0 station dump | awk '/signal:/{print $2}')
  echo "$txpower,$rssi,$throughput"


done
```

The client will set different values for the transmission power by using the following command:
`iw dev wlan0 set txpower fixed $txpower`
The value of 'txpower' is set by the user for each experiment. To set a value of 10 dBm, a value of 1000 will be used without considering the gain of antenna that is 5 dBi.

Listing 5.10: iPerf client running on RSU

```
#!/bin/bash
echo "Client as RSU"
if [ $# -ne 3 ]; then
  echo "Three arguments expected"
  echo "1. Bit rate"
  echo "2. Bandwidth"
  echo "2. Tx power"
  exit 1
fi

rate=$1
txpower=$3

echo "setting tx power"
iw dev wlan0 set txpower fixed $txpower
sleep 1

echo "Setting physical layer data rate to $rate Mbit/s at client"
#phyrate =$((rate*2))
phyrate=$( expr 2 '*' "$rate")
iw dev wlan0 set bitrates legacy-5 $phyrate
sleep 1

echo "iperf client"
iperf -c 10.10.6.104 -u -i 1 -t 60 -b $2M -p 5001
```

The final output values are shown in Listing 5.11 in the following columns:

1. Transmitted power
2. RSSI
3. Throughput

Listing 5.11: Static output results of the mentioned parameters

```
Starting server placed inside the car as OBU
Setting physical layer data rate to 3 Mbit/s
Reading throughput from iperf server
5.00,-47,2.63
5.00,-47,2.61
5.00,-47,2.61
5.00,-47,2.61
5.00,-48,2.62
5.00,-48,2.61
5.00,-48,2.62
5.00,-48,2.61
5.00,-48,2.62
5.00,-48,2.61
5.00,-48,2.61
5.00,-48,2.62
5.00,-47,2.61
5.00,-47,2.61
5.00,-48,2.62
5.00,-47,2.62
5.00,-48,2.61
5.00,-48,2.61
5.00,-48,2.62
5.00,-48,2.61
5.00,-47,2.62
5.00,-47,2.61
5.00,-48,2.61
```

5.2.2 Measurements for link stability with respect to distance

This test is planned in order to evaluate the 802.11p performance based on the distance. Another objective is to determine the maximum distance at which the connection is reliable and the communication is not interrupted.

The test can be divided into two scenarios: one is the static one in which the vehicle is not moving, and the other is a real-world experiment to see how the distance affects the received power.

To find the link stability, the two parameters that are used to assess the channel reliability with respect to distance are RSSI and transmission retries. In the developed script, both these values are measured by using the following command:

```
iw dev wlan0 station dump
```
(5.1)

The distance values will be given by a GNSS receiver as already mentioned in chapter 4. The testing scenario is the following:

- In order to calculate the value of RSSI, we need the server or RSU to continuously send the unicast ICMP packet (with `ping`), so that client can receive the messages from the station. The RSU is set to send ping messages at 5 Hz as shown in Listing 5.12 .
- `iw` will be launched at the client side which is acting as OBU on which we can log the RSSI and transmission retries values at the frequency of 5 Hz.
- GNSS receiver is giving the distance values at 5 Hz.
- The transmission retries (txretries) tells how much re-transmissions happened in 802.11 with respect to the distance. As if the RSU will not get the ack back, it will send the message again in unicast OCB mode. So it is the measure of how much the channel is reliable.
- The test can be performed at different speed values; however, the speed of the car must be kept constant during a single test.

Listing 5.12: Road Side Unit continously sending ping messages

```
#!/bin/sh
while true;
do
  ping 192.168.1.184
  ./millisleep 0.2  # 5Hz
  clear
done
```

At the OBU, as shown in listing 5.13 the difference of **tx retries** will be computed by considering the fact that it may not start from zero.

Listing 5.13: OBU computing the result for distance, signal strength and transmission retries

```
#!/bin/bash

echo ‘clear‘

i=0;  # set i=0 to read the first value of txretries

stdbuf -oL ./distance |
while IFS= read -r distance
do

#echo "Distance=$distance"
a=($(iw dev wlan0 station dump | awk ’/signal:/{print $2 } /tx retries
    :/{print $3}’) ) ;

if [[ $i -eq 0 ]]
then
{
  tmp=${a[0]};
  echo "$distance,${a[1]},$tmp" #for the first values
  i=1;
}
else
{
  txretries=${a[0]};

  diff_txretries=$(($tx - $tmp)) ;

  tmp=txretries;
}
fi

echo "$distance,${a[1]},$diff_txretries" >> result.csv #distance,
    signal strength, difference in txretries
done
```

Results of the scripts are shown in Listing 5.14 based on static measurements. The output is shown as:

1. Distance
2. RSSI
3. Transmission retries

Listing 5.14: Signal strength and transmission retries with respect to the distance

```
distance,RSSI,txretries
0.037470,-47,0
0.037470,-47,0
0.037470,-47,0
0.430744,-47,0
0.430744,-47,0
0.430744,-47,0
0.449877,-47,0
0.449877,-47,0
0.449877,-47,0
0.449877,-47,0
0.449877,-47,0
0.449877,-47,0
0.463708,-47,0
0.449877,-47,0
0.861488,-47,0
0.861488,-47,0
0.861488,-47,0
0.861488,-47,0
0.861488,-47,0
0.861488,-47,0
0.861488,-47,0
```

5.3 Latency measurements

Latency is one of the most important performance metrics for safety linked applications in vehicular networks. We are measuring the network latency by computing the delay of packet generation at RSU to the reception of the packet at OBU and way back. The measurements are designed to determine how much the latency is stable with respect to the speed of a car and the distance in mobility conditions.

In this work, to measure the latency we are using the LaTe tool as described earlier in chapter 1. This tool works in a client server architecture, in which a server replies back to the client requests. All the information related to the compilation of LaTe on Linux devices and its source code is available at [53], as it is an open source project.

The main usage aspects of this tool are briefly described below and are mentioned in detail in LaTe help [54].

```
LaTe [-c <destination ip address> [mode] [protocol] [options]]
```

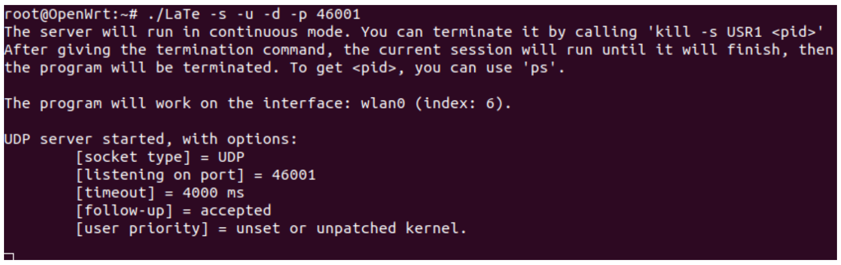
- In order to run the LaTe on client side, the `-c` option is specified
- To run the protocol as a server, `-s` is used.
- The destination address specifies the address of the machine where the server is running.
- There are two different modes defined in LaTe: one is unidirectional and the other is ping-like bidirectional mode, however we are using the ping-like mode as unidirectional requires the two hardware's clock to be synchronized.
- In the protocol section we can define the protocol to be used for latency measurements, `-u` is used for the UDP protocol.
- There are several other client and server options are available that are optional.

5.3.1 LaTe server

The LaTe server is running on APU_104 with following options as shown in Figure 5.3. In particular we are relying on the following command:

```
./LaTe -s -u -d
```

- `-s`: LaTe server
- `-u`: UDP protocol (layer 4)
- `-d`: server running in continuous daemon mode
- `-p`: it specifies the port to be used, as in our case we are using the port 46001.



```
root@OpenWrt:~# ./LaTe -s -u -d -p 46001
The server will run in continuous mode. You can terminate it by calling 'kill -s USR1 <pid>'
After giving the termination command, the current session will run until it will finish, then
the program will be terminated. To get <pid>, you can use 'ps'.

The program will work on the interface: wlan0 (index: 6).

UDP server started, with options:
        [socket type] = UDP
        [listening on port] = 46001
        [timeout] = 4000 ms
        [follow-up] = accepted
        [user priority] = unset or unpatched kernel.
```

Figure 5.3: LaTe server running on port 46001

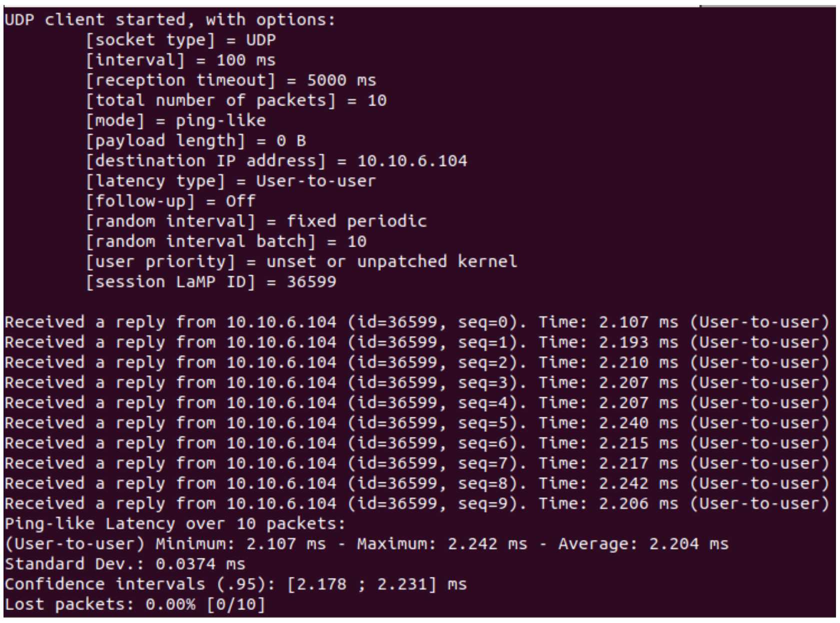
5.3.2 LaTe client

We are using the LaTe client on APU_105 as shown in figure 5.4. In particular we are relying on the following command:

```
./LaTe -c 10.10.6.104 -u -n 10 -t 200 -B -w 192.168.1.9:46001
```

The used options in the above command are:

- `-c`: client side
- `-u`: UDP protocol
- `-n`: it defines the total number of packets to sent, default is 600
- `-t`: it is sending each packet after 200 ms
- `-B`: bidirectional mode
- `-P`: payload size in bytes, default is 0
- `-w`: this option is used to report the per packet data and will be described later in details.
- The two IP address that are mentioned in the command are:
    1. 10.10.6.104: this is WiFi 802.11p address, on which the LaTe server is running.
    2. 192.168.1.9: This is the IP address of the development PC, in which the application getting the measurements being run.



```
UDP client started, with options:
        [socket type] = UDP
        [interval] = 100 ms
        [reception timeout] = 5000 ms
        [total number of packets] = 10
        [mode] = ping-like
        [payload length] = 0 B
        [destination IP address] = 10.10.6.104
        [latency type] = User-to-user
        [follow-up] = Off
        [random interval] = fixed periodic
        [random interval batch] = 10
        [user priority] = unset or unpatched kernel
        [session LaMP ID] = 36599

Received a reply from 10.10.6.104 (id=36599, seq=0). Time: 2.107 ms (User-to-user)
Received a reply from 10.10.6.104 (id=36599, seq=1). Time: 2.193 ms (User-to-user)
Received a reply from 10.10.6.104 (id=36599, seq=2). Time: 2.210 ms (User-to-user)
Received a reply from 10.10.6.104 (id=36599, seq=3). Time: 2.207 ms (User-to-user)
Received a reply from 10.10.6.104 (id=36599, seq=4). Time: 2.207 ms (User-to-user)
Received a reply from 10.10.6.104 (id=36599, seq=5). Time: 2.240 ms (User-to-user)
Received a reply from 10.10.6.104 (id=36599, seq=6). Time: 2.215 ms (User-to-user)
Received a reply from 10.10.6.104 (id=36599, seq=7). Time: 2.217 ms (User-to-user)
Received a reply from 10.10.6.104 (id=36599, seq=8). Time: 2.242 ms (User-to-user)
Received a reply from 10.10.6.104 (id=36599, seq=9). Time: 2.206 ms (User-to-user)
Ping-like Latency over 10 packets:
(User-to-user) Minimum: 2.107 ms - Maximum: 2.242 ms - Average: 2.204 ms
Standard Dev.: 0.0374 ms
Confidence intervals (.95): [2.178 ; 2.231] ms
Lost packets: 0.00% [0/10]
```

Figure 5.4: LaTe client

5.3.3 Latency measurement with respect to speed

In order to measure the latency with respect to the speed of the car and to report a packet-wise latency information, a receiver application in C is developed. The full code is available in Appendix C.

The testing scenario is shown in Figure 5.5, in which the two APU boards are exchanging 802.11p data.

There is a `-w` option in LaTe that will send the measurement data to a receiver application which IP address is mentioned in the command. In our case the application is running on 192.168.1.9 and on port 46001 that is also by default port. On this development PC the GPS device is also running so that we can log the latency measurement with respect to the speed.

`’-w 192.168.1.9:46001,eth2’`

If the interface is not specified with `-w` option as it is defined above as ’eth2’, it will be bound to all the interfaces.

On the receiver application two sockets must be created and bound according to the requirements described in LaTe tool help options: one is TCP and other is UDP socket. The TCP socket is running as TCP server and wait for the TCP client to connect (Listing 5.15). The TCP client here is the Late client, with the -w option.

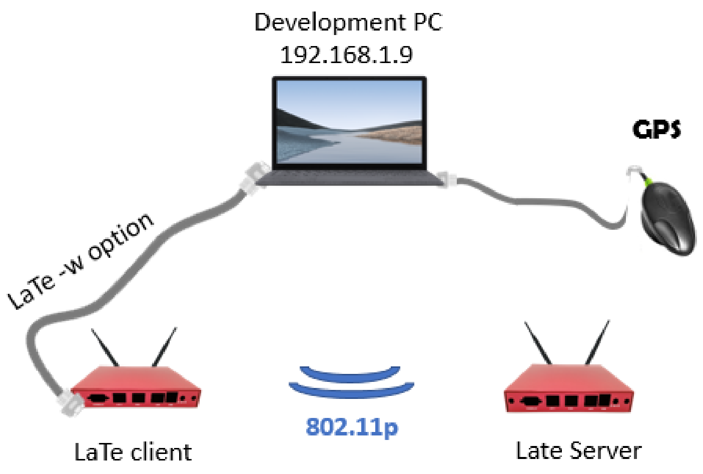


Figure 5.5: LaTe testing

Listing 5.15: Creating a TCP socket and binding it with the IP address of the receiver application device

```
//tcp socket create

if ((server_fd = socket(AF_INET, SOCK_STREAM, 0)) == 0)
{
  perror("socket failed");
  exit(EXIT_FAILURE);
}

//set the socket options for TCP
if (setsockopt(server_fd, SOL_SOCKET, SO_REUSEADDR | SO_REUSEPORT,
&opt, sizeof(opt)))
{
  perror("setsockopt");
  exit(EXIT_FAILURE);
}

// Filling server information
servaddr.sin_family  = AF_INET; // IPv4
servaddr.sin_addr.s_addr = INADDR_ANY;
servaddr.sin_port = htons(46001);
```

```
//binding TCP socket
if (bind(server_fd, (struct sockaddr *)&servaddr,
sizeof(servaddr))<0)
{
  perror("bind failed");
  exit(EXIT_FAILURE);
}

printf("Before listen server\n");

if (listen(server_fd, 3) < 0)
{
  perror("listen");
  exit(EXIT_FAILURE);
}
/*OPENING UDP SOCKET AND BIND IT*/
printf("Before listen server\n");

// Creating socket file descriptor for UDP
if ( (sockfd = socket(AF_INET, SOCK_DGRAM, 0)) < 0 )
{
  perror("socket creation failed");
  exit(EXIT_FAILURE);
}

if (bind(sockfd, (const struct sockaddr *)&servaddrudp,
sizeof(servaddrudp)) < 0 )
{
  perror("bind failed");
  exit(EXIT_FAILURE);
}
```

The TCP socket is used for sensitive data such as the connection information, LaTe fields and LaMP ID or LaTe test ID. While all the other data or per packet latency information will be send on UDP.

The first packet that we receive from TCP is 'LaTeINIT', in which we have information related to the different fields of data which will be received through UDP, along with LaTe test ID.

This LaTe ID will be passed to the UDP thread in which we are reading data from the

UDP socket as shown in the listing 5.16:

Listing 5.16: Checks on the received ID

```
if(sizeof(buffertcp) == 0)
{
  printf("Error: LaTe was disconnected too early.\n");
  continue;
}
//extracting the LaTe ID
lateSplit = str_split(buffertcp, ',');
int lamp_session_id=atoi(lateSplit[1]);

if ((strcmp(lateSplit[0], "LaTeINIT") != 0 ) && (strcmp(lateSplit[0],
    "LaTeEND") != 0 ) )
{
  printf("Expected a LaTeINIT or LateEND packet, but received %s \n",
      lateSplit[0]);
  close(new_socket);
  continue;
}
else if ((strcmp(lateSplit[0], "LaTeEND") == 0 ))
{
  finalate_fields = str_split(str_split( lateSplit[2],'=')[1],';');
  printf("LateEnd \n");
  close(new_socket);
  continue;
}
else
{
  printf("Received a %s \n",lateSplit[0]);
}

//after extracting infromation from TCP packet send it to UDP thread
    using structures

late_fields = str_split(str_split( lateSplit[2],'=')[1],';');
UDPThreadArgs udpTArgs;
udpTArgs.lamp_session_id = lamp_session_id;
udpTArgs.late_fields1 = malloc(sizeof(late_fields));
udpTArgs.late_fields1 = late_fields;
udpTArgs.udp_sessioin_id = sockfd;
```

If the packets received on UDP packet have different ID as mentioned in 'LaTeINIT' then these packets will be discarded.

When we receive a 'LaTeEND' packet via TCP, no more data from the UDP socket will be received and the TCP socket can be closed. The LaTeEND packet shows the final statistics such as minimum, maximum and average values of the latency.

Listing 5.17: Received the LaTeEND packet and stop receiving data from UDP socket

```
if (strcmp(lateSplit2[0], "LaTeEND") != 0 )
{
  printf("Expected a LaTeEND packet, but received %s ",lateSplit2[0]);
  printf("No final report data can be obtained.");

}
//if the packet received is LaTeEND, stop receiving data from UDP
if( (strcmp(lateSplit2[0], "LaTeEND") == 0) )
{

  printf("Received a %s packet", lateSplit2[0]);

  if(write(pipeDescriptor[1],"\0",1)<0) {
  fprintf(stderr,"Its termination will be forced.\n");
  printf("loop cancel\n");
}

if ( atoi(lateSplit2[1]) != lamp_session_id  )
{
  printf("Warning: data ignored. Expected test ID %d but received %d",
       lamp_session_id, atoi(lateSplit[1]));

}
else
{
  if ( strcmp(lateSplit2[2], "srvtermination") != 0  )
  {
     // final_test_data_operations( lateSplit);
    printf("Here ");
    break;
  }
}

}
```

The testing scenario using the LaTe receiver application and GPS device as shown in Figure 5.5 is the following one:

- Start the LaTe server on RSU in continuous daemon mode
- Start the receiver application on the development PC on which the TCP server is listening for new connections from LaTe.
- Start the LaTe client on the OBU, when the car is started
- As the car starts moving the GPS device will start logging the speed values
- Both the speed values and LaTe values are received every 0.2 seconds, but still we used the timestamping in order to be sure that for each speed value we have the corresponding latency value.

Listing 5.18: Latency vs speed values

```
//if both values received at the same time then they will be printed
if ( abs(LateTime - gpsTime) == 0 )
{
  printf("latency %f, speed %f \n", LateValue, gpsValue);
}
```

The real measurements of this test were not computed. However, the expected results will be look like this as shown in Listing 5.19.

Listing 5.19: Latency measurements with respect to speed

```
  latency 2.282000, speed 0.062000
  latency 2.286000, speed 0.029000
  latency 2.248000, speed 0.029000
  latency 2.217000, speed 0.029000
  latency 2.232000, speed 0.108000
  latency 2.248000, speed 0.108000
  latency 2.240000, speed 0.081000
  latency 2.202000, speed 0.081000
  latency 2.236000, speed 0.060000
```

```
latency 2.270000, speed 0.060000
latency 2.268000, speed 0.060000
latency 2.239000, speed 0.060000
latency 2.249000, speed 0.060000
latency 2.239000, speed 0.091000
latency 2.263000, speed 0.091000
latency 2.270000, speed 0.022000
latency 2.232000, speed 0.022000
latency 2.298000, speed 0.030000
latency 2.299000, speed 0.030000
latency 2.192000, speed 0.030000
Received a LaTeEND packet
```

5.3.4 Latency measurement with respect to distance

In this test, the aim is to find the maximum range at which the latency is stable. For this purpose we are using the same receiver application on the development PC as we used in above test. But now instead of using GPS function to compute speed we are calculating the distance from the point when the car is start moving.

The distance function used for computation is the same as mentioned in chapter 4. In order to synchronize the two data sources in this test, we are using the distance function within the thread. The GPS thread for distance calculation is shown in Listing 5.20. However, for the sake of completeness the full code is also available in Appendix D.

The testing scenario is this one:

- GPS is running on development PC, logging distance values.
- Receiver application is also running on development PC as described in Figure 5.5.
- LaTe client is running on APU_105 board along with -w option in order to send the latency data to the receiver application (192.168.1.9) as described in the following command:
  `./LaTe -c 10.10.6.104 -u -n 10 -t 200 -B -w 192.168.1.9:46001.`
- LaTe server is running on APU_104.
  `./LaTe -s -u -d`

- On the development PC we have the latency values from LaTe and distance values from GPS.

Listing 5.20: GPS thread calculating distance

```
void *gps(void *arg)
{
  printf("Running in thread.");
  FILE *pipe ;
  gps_data data1;
  char line [BUFSIZE];
  char** tokens;

  pipe= popen("./gpsscript.sh","r");
  char str[7]="$GNGNS,";
  double lat1=0.0,lat2=0.0,long1 = 0.0,long2 = 0.0;
  int R=6371e3;
  float rad=3.14/180;
  printf("%f\n", rad);
  if (pipe == NULL)
    {
    printf("CANT FIND DEVICE");
    pthread_exit(0);
  }


  while (fgets(line,sizeof(line), pipe) != NULL)
  {
    if (strncmp(line,str,7) == 0)
    {
      tokens = str_split(line, ',');

      if (tokens == NULL)
        {
        continue;
      }

      for (int i = 0; *(tokens+i); i++)
      {
        char *singleString = *(tokens + i);
      if (i==0)
      {
        continue;
      }

      else if(i==2)
      {
        if (lat1 == 0) {
        lat1 = strtod(singleString, NULL);
      }
      else
      {
        data1.latitude = strtod(singleString, NULL);
        lat2 = data1.latitude;
      }

      }
      else if (i == 4)
      {

        if (long1 == 0 ){
        long1 = strtod(singleString, NULL);
        }
      else
        {
        data1.longitude =  strtod(singleString, NULL);
        long2 = data1.longitude;
        }

      }

    }

    // checking if we got both first location and second location,
    if ( lat1 != 0 && long1 != 0 && lat2 != 0 && long2 != 0)
    {
      // this place will run for second reading, because at first
         reading lat2 and long2 will be 0
      double lat1_deg = decimal_deg(lat1);
      double lat2_deg = decimal_deg(lat2);
```

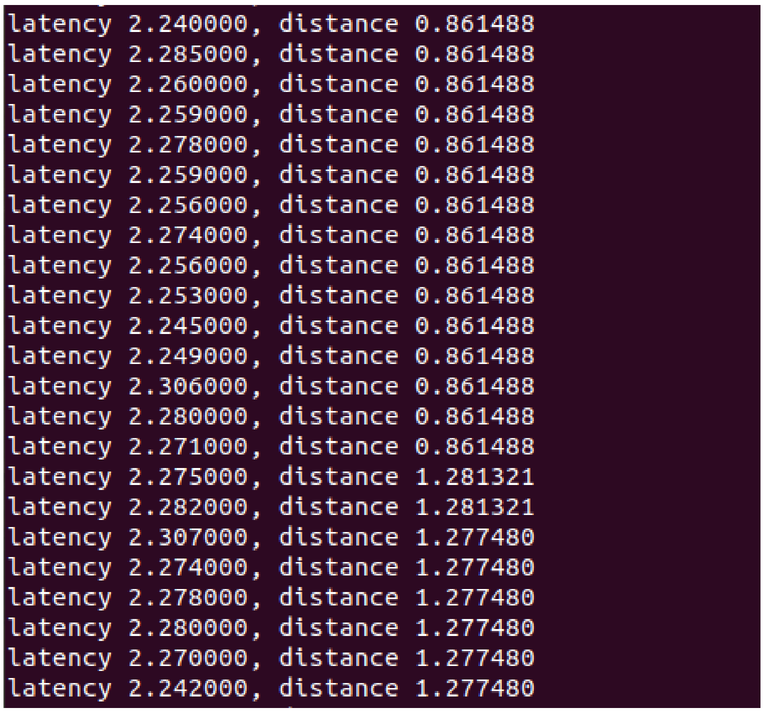
```
      double long1_deg = decimal_deg(long1);
      double long2_deg = decimal_deg(long2);

      pthread_mutex_lock(&mutex);
      gpsTime = time(NULL);
      gpsValue =distance(lat1_deg,long1_deg,lat2_deg,long2_deg);

      pthread_mutex_unlock(&mutex);
      // printf("%f\n", res );

    }
    free(tokens);
    }
  }
  pclose(pipe);
  pthread_exit(0);
}
```

The final output is expected in this form as shown in Figure 5.6:



```
latency 2.240000, distance 0.861488
latency 2.285000, distance 0.861488
latency 2.260000, distance 0.861488
latency 2.259000, distance 0.861488
latency 2.278000, distance 0.861488
latency 2.259000, distance 0.861488
latency 2.256000, distance 0.861488
latency 2.274000, distance 0.861488
latency 2.256000, distance 0.861488
latency 2.253000, distance 0.861488
latency 2.245000, distance 0.861488
latency 2.249000, distance 0.861488
latency 2.306000, distance 0.861488
latency 2.280000, distance 0.861488
latency 2.271000, distance 0.861488
latency 2.275000, distance 1.281321
latency 2.282000, distance 1.281321
latency 2.307000, distance 1.277480
latency 2.274000, distance 1.277480
latency 2.278000, distance 1.277480
latency 2.280000, distance 1.277480
latency 2.270000, distance 1.277480
latency 2.242000, distance 1.277480
```

Figure 5.6: Latency measurements with respect to the distance

Chapter 6

Conclusions and future work

6.1 Conclusions

In this thesis work, our goal was to evaluate the performance of an 802.11p technology in a mobility V2X scenario. We planned several tests for the 802.11p technology assessment with the main focus on throughput and latency measurements by using iPerf and LaTe measurement tools.

Our planned tests are based on real world experiments in order to indicate how the speed and distance covered by a vehicle impact the throughput and latency measurements.

These tests also include the steps to determine which system parameters are more suitable for mobility scenarios. For this reason, we have proposed comparing measurements with different data rates, offered traffic and UDP socket window sizes, and analyze how these parameters will affect the throughput and coverage range.

We also proposed the tests for determining the channel reliability based on RSSI, transmission retries and transmitted power.

Furthermore, we developed a receiver application on a development PC using C language. This application is able to receive the latency data coming from LaTe tool and combine it with a GPS data.

In this work, we have also proposed a method of determining the possible relation between the speed of the electric vehicle which was available during this work and Engine RPM values, extracted from the CAN socket.

We have also presented how to configure a GPS device on APU1D embedded boards, for the calculation of distance by using longitude and latitude values, and how to extract the speed information from GPS sentences.

6.2 Future work

The work done so far can still be enhanced and developed further, especially in executing other V2X features.

- All of these tests can be executed in a real world scenario using a car, embedded boards and a GPS device.
- While measuring the throughput and latency, we have not considered the payload size or packet sizes. Aforementioned results can be further improved with the consideration of packet size.
- All of these tests can be repeated on V2V scenarios in order to determine the IEEE 802.11p protocol performance in some relevant use cases such as in collision avoidance and congestion.
- These tests can be repeated by using a vehicle that is able to reach higher speed with respect to the maximum speed of the electric vehicle we had available for this work in order to see the impact of higher speed on 802.11p connectivity.
- In future, we also plan to increase the number of embedded boards in our experiment which will allow to look into the capacity of 802.11p protocols and analyze its scalability when it comes to the hardware and software solutions we proposed in this work.
- Finally, one goal could be to plan the tests on OpenC2X ETSI ITS-G5 platform developed by CCS Labs in University of Paderborn (Germany) and to evaluate the platform when sending CAM and DENM messages.

6.3 Acknowledgments

First of all I would like to thank Allah Almighty for keeping me save throughout COVID-19 pandemic. It is due to His blessings that I am healthy and have worked efficiently on my thesis.

My whole hearted thanks goes to my project supervisor Prof. Claudio Ettore Casetti, for inspiring my interest in the field of vehicular networks. His support and guidance through every stage has been greatly helpful and he was always very kind to me.

I would also like to extend my thanks of gratitude to my co-supervisor Francesco Raviglione, without whom this thesis work would never have reached to accomplishment. His elevating motivation, inspiration, active availability either in lab or on Emails and constant support allowed me to successfully complete my thesis.

Moreover, I must express my deep and honest gratitude to my loving father without whose affection, concern and constant effort, it would not have been possible for me to bear prolonged experience of this phase of study and research.

Heartiest gratitude to my sister Asma for her provision of assistance whenever required and for being a constant support throughout this journey.

I also pay my deep respect to all my teachers who have paved the path of knowledge for me all my academic life. Thanks to my friends who always supported me and lifted me up with their love and motivation.

Appendices

Appendix A

Listing 1: Reading RPM values from CAN interface

```
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <stdlib.h>
#include <stdbool.h>
#include <time.h>
#include <assert.h>
#include <errno.h>
#include <sys/time.h>
#include <sys/socket.h>
#include <sys/select.h>
#include <sys/ioctl.h>
#include <arpa/inet.h>
#include <sys/ioctl.h>
#include <inttypes.h>
#include <linux/errqueue.h>
#include <signal.h>
#include <fcntl.h>
#include <sys/types.h>
#include <linux/can.h>
#include <pthread.h>
#include <linux/can/raw.h>
#include <net/if.h>
#include <linux/sockios.h>
#include <math.h>
#include <linux/if.h>
#define size 65537
#define _POSIX_C_SOURCE 200809L
```

```
#define PROGNAME "socketcan"

static sig_atomic_t sigval;

//fucntioning for converting rpm values to decimal also concatenation
    high and low bytes
int concatenate(int a, int b)
{

  int num1,num2,num3;
  num1=a;
  num2=b;
  num3=(num1<<8)|(num2);
  //printf("%d",num3);
  return num3;
}
int z=2;
int simplp (int * x, int  * y, int M, int xm1, FILE *fin)
{
  int n;
  y[0] = x[0] + xm1; //xml is the intial value
  y[1]=x[1]+x[0];
  y[2]=x[2]+x[1];
  y[3]=x[3]+x[2]+x[1];
  y[4]=+x[4]+x[3]+x[2]+x[1];
  y[5]=x[5]+x[4]+x[3]+x[2]+x[1];
  for (n=6; n < M ; n++)
  {
    y[n] =  x[n-6]+x[n-5]+x[n-4]+x[n-3]+x[n-2]+x[n-1]+x[n];
    //used for avaeraging
    printf("%d\n",y[n] );
    y[n]=y[n]/7; //avearging over past 7 values
    if(z==10) // as we are starting from zero so 11th value is 10
    {
      fprintf(fin, "%d\n", y[n] ); //writing the results to the file
    }
    z++;
  }
  return x[M-1];
}
```

```
static void onsig(int val)
{
  sigval = (sig_atomic_t)val;
}

int main()
{

  int Socket_D;
  FILE *fptr;
  char line[16];
  char c;
  int ss;
  int i=0;
  double temp=0,diff=0, diff_div=0;
  int j=0; //for downsampling loop
  int M=50; //blobk size of lpf
  int xm1;
  int tmp=0;
  //below all the variables related to timestamp
  int so_timestamping_flags = 0;
  int so_timestamp = 0;
  int so_timestampns = 0;
  int siocgstamp = 0;
  int siocgstampns = 0;
  int level, type;

  //all variables and array declarartions for low pass filter
  int *x; //input array
  int *y; //output array
  long *t; //for timestamp values
  int num=256;
  int k=0;
  x=(int*)malloc(size* sizeof(int));
  y=(int*)malloc(size* sizeof(int));
  t=(long*)malloc(size* sizeof(long));
  result_lpf = (int*)malloc(size* sizeof(int));

  struct cmsghdr *cm;
  struct msghdr msg;

  struct timespec *ts = NULL;
  struct timeval *tv=0;


  char ctrlBufSw[CMSG_SPACE(sizeof(struct timeval))];

  struct can_frame frame;
  size_t nbytes;

  // struct iovec
  struct iovec iov;

  // Prepare ancillary data structures
  iov.iov_base=&frame;
  iov.iov_len= sizeof(struct can_frame);

  msg.msg_name=NULL;
  msg.msg_namelen=0;
  // Ancillary data (control message)
  msg.msg_control=ctrlBufSw;
  msg.msg_controllen=sizeof(ctrlBufSw);
  // iovec arrays
  msg.msg_iov=&iov;
  msg.msg_iovlen=1; // 1 element for each recvmsg()
  // Ancillary data (control message)
  msg.msg_control=ctrlBufSw;
  msg.msg_controllen=sizeof(ctrlBufSw);
  // iovec arrays
  msg.msg_iov=&iov;
  msg.msg_iovlen=1; // 1 element for each recvmsg()
  msg.msg_flags=0;

  //fptr = fopen("timestamp.csv","w"); //temporary file for writing
      the rpm before filtering
  //fout=fopen("rpmfinal.csv","w"); // writing the result after
      filtering and downsampling will be done by using this file
  struct sockaddr_can addr;
  struct ifreq ifr;
```

```
int b=0x100817EF; //rpm ID
int rpm0,rpm1,rpm_h,rpm_l,rpm_f;
struct can_filter rpmfilter[1];

/* Register signal handlers */
if (signal(SIGINT, onsig)    == SIG_ERR ||
signal(SIGTERM, onsig)   == SIG_ERR ||
signal(SIGCHLD, SIG_IGN) == SIG_ERR)
  {
      perror(PROGNAME);
      return errno;
  }

//open the socket for Can
Socket_D = socket(PF_CAN, SOCK_RAW, CAN_RAW);

//check is it open
if (Socket_D < 0)
  {
    perror(PROGNAME ": socket");
    return errno;
  }

//to get the interface index

strcpy(ifr.ifr_name, "vcan0" );
ioctl(Socket_D, SIOCGIFINDEX, &ifr);

addr.can_family = AF_CAN;
addr.can_ifindex = ifr.ifr_ifindex;
//bind the socket
bind(Socket_D, (struct sockaddr *)&addr, sizeof(addr));

//filter used to get only rpm values

rpmfilter[0].can_id   = b ;
rpmfilter[0].can_mask =0xFFFFFFF;

int enable=1;
```

```
if(setsockopt(Socket_D, SOL_SOCKET, SO_TIMESTAMP, &enable, sizeof(
    enable))<0)
  {
    perror("std error");
    exit(EXIT_FAILURE);
  }

setsockopt(Socket_D, SOL_CAN_RAW, CAN_RAW_FILTER, &rpmfilter, sizeof
    (rpmfilter));

sigval=0;

//read data from CAN frames'
long micro_sec;
int count=0;

while (sigval== 0)
{

nbytes = recvmsg(Socket_D, &msg, 0);
for (cm = CMSG_FIRSTHDR(&msg); cm ; cm = CMSG_NXTHDR(&msg, cm))
  {

    if (cm->cmsg_type == SO_TIMESTAMP)
    {
      tv = (struct timeval *) CMSG_DATA(cm);
      t[k]=tv->tv_usec ;
      k++;
      printf("timestamp: %ld ",tv->tv_usec); //equivalent to fprintf
          (stderr,"timestamp %ld ",tv->tv_usec);
    }
  }

if (nbytes < 0) {
perror("can raw socket read");
return 1;
}

/* paranoid check ... */
if (nbytes < sizeof(struct can_frame)) {
```

```
fprintf(stderr, "read: incomplete CAN frame\n");
return 1;
}

//printing it
printf("can_id: %X data: ", frame.can_id);
for (int i = 0; i < 2; i++)
  {
    if(i==0)
    {
      //printf("%X ", frame.data[i]);
      rpm1=frame.data[i];
    }
    else
    {
      //printf("%X ", frame.data[i]);
      rpm0=frame.data[i]; //read as 0-1 for H-L
    }

  }
  // printf("\n");
  rpm_h=rpm0;
  rpm_l=rpm1;
  rpm_f= concatenate(rpm_h, rpm_l);
  printf("RPM: %d\n",rpm_f );

  //fprintf(fptr, "%d\n", rpm_f);

  if ((i >=   M))
  {
    i=0;
    xm1 = simplp(x, y, M, tmp,fptr);
    tmp=xm1;

  }
  //printf("%d \n",Socket_D);
  else if (i<M)
  {
    x[i]=rpm_f;
    }

    i++;

    }

    M=i; // for the remaining values
    xm1 = simplp(x, y, M, tmp,fptr);
  }
  free(x);
  free(y);

  // Close the CAN interface
  if (close(Socket_D) < 0)
  {
    perror(PROGNAME ": close");
    return errno;
  }
  return EXIT_SUCCESS;

}
```

Appendix B

Listing 2: GPS code for distance and speed

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <signal.h>
#include <limits.h>
#include <math.h>
#include <time.h>
#include <assert.h>
#include <sys/time.h>
#include <string.h>
#define BUFSIZE 256
#define M_PI 3.14159265358979323846
#define MIN(x, y) (((x) < (y)) ? (x) : (y)) //function to calculate
   minimum value

typedef struct{
char name[6];
float timeZone;
double latitude;
char  D1;
double longitude;
float speed_kn;
float speed_km;
char D2;
char mode[2];
int satelites;
float dilution;
float antAltitude;
float GS;
float Differential;
int chksum;
} gps_data;

float decimal_deg(double value)
{
//concatenate int part and dec part
int degrees = floor(value / 100);
//printf("degree %d\n", degrees );
float minutes = value - (degrees * 100);
//printf("%f\n",minutes );
value = degrees + minutes/60;
return value;

}


double toRadians( double degrees)
{
double one_deg = (M_PI) / 180;
return (one_deg * degrees);
}

double distance(double lat_1, double long_1, double lat_2,double
    long_2)
{
double R = 6371e3; //in meters and so the distance will also be in m/s
double latitude1,latitude2,longitude1,longitude2;
latitude1=lat_1;
latitude2=lat_2;
longitude1=long_1;
longitude2=long_2;
latitude1=toRadians(latitude1);
latitude2=toRadians(latitude2);
longitude1=toRadians(longitude1);
longitude2=toRadians(longitude2);



double dlong = longitude2 - longitude1;
double dlat = latitude2 - latitude1;

double ans = pow(sin(dlat / 2), 2) +
cos(latitude1) * cos(latitude2) *
pow(sin(dlong / 2), 2);

ans = 2 * asin(sqrt(ans));

ans = ans * R;

// printf("distance is %lf\n",ans);
return ans;
}

char** str_split(char* a_str, const char a_delim)
{
char** result    = 0;
size_t count     = 0;
char* tmp        = a_str;
char* last_comma = 0;
char delim[2];
delim[0] = a_delim;
delim[1] = 0;

/* Count how many elements will be extracted. */
while (*tmp)
{
if (a_delim == *tmp)
{
count++;
last_comma = tmp;
}
tmp++;
}

/* Add space for trailing token. */
count += last_comma < (a_str + strlen(a_str) - 1);

/* Add space for terminating null string so caller
knows where the list of returned strings ends. */
```

```
count++;

result = malloc(sizeof(char*) * count);

if (result)
{
size_t idx  = 0;
char* token = strtok(a_str, delim);

while (token)
{
assert(idx < count);
*(result + idx++) = strdup(token);
token = strtok(0, delim);
}
assert(idx = count - 1);
*(result + idx) = 0;
}

return result;
}


int main()
{
FILE *pipe, *fptr;
fptr=fopen("time_gps.csv","w");
gps_data data1;
char line [BUFSIZE];
char** tokens;
struct timeval a;

pipe= popen("./gpsscript.sh","r"); //to run the bash script
char str[7]="$GNGNS,";
char str2[7]="$GNRMC,";
char str3[7]="$GNVTG,"; // we also try to get speed from VTG sentences
    but it was not accurate
const char *time_array;
int timeinsec;
int totaltimeinsec=1;
```

```
double lat1=1,lat2=1,long1,long2;
int R=6371e3;
float rad=3.14/180;
printf("%f\n", rad);
if (pipe == NULL) {
printf("CANT FIND DEVICE");
return 1;
}


while (fgets(line,sizeof(line), pipe) != NULL)
{
if (strncmp(line,str,7) == 0)
{
tokens = str_split(line, ',');

if (tokens == NULL) {
continue;
}

for (int i = 0; *(tokens+i); i++)
{
char *singleString = *(tokens + i);

//printf("%s \n", singleString);

if (i==0)
{
continue;
}
else if(i==1)
{
totaltimeinsec=timeinsec;
data1.timeZone = strtod(singleString, NULL);
//printf("TimeZone: %f \n", data1.timeZone);
hour = data1.timeZone/10000;
min = ((int)data1.timeZone % 10000) / 100;
sec = ((int)data1.timeZone % 100);
}
else if(i==2)
{
lat1=data1.latitude;
data1.latitude = strtod(singleString, NULL);
lat2=data1.latitude;
//printf("Lat: %f \n", data1.latitude);
//printf("Lat_tmp: %f \n", Lat_tmp);
}
else if (i == 4)
{
long1=data1.longitude;
data1.longitude = strtod(singleString, NULL);
long2=data1.longitude;

//printf("Long: %F \n", data1.longitude);
//printf("Long_tmp: %f \n", Lon_tmp);
}



}
lat1=decimal_deg(lat1);
lat2=decimal_deg(lat2);
long1=decimal_deg(long1);
long2=decimal_deg(long2);
float res=distance(lat1,long1,lat2,long2);
printf("%f\n", res );

free(tokens);
}

else if (strncmp(line,str2,7) == 0)
{
tokens = str_split(line, ',');


if (tokens == NULL)
{
continue;
}
```

```
for (int i = 0; *(tokens+i); i++)
{
char *singleString = *(tokens + i);
//printf("%s \n", singleString);

if (i==0)
{
continue;
}

else if(i==7)
{

data1.speed_kn = strtod(singleString, NULL);
data1.speed_kn= data1.speed_kn* 1.852; //knots to km/h
printf("RMC: %f \n", data1.speed_kn); //speed in km/h

}
}
printf("\n");
free(tokens);

}

else if (strncmp(line,str3,7) == 0)
{
tokens = str_split(line, ',');

if (tokens == NULL)
{
continue;
}

for (int i = 0; *(tokens+i); i++)
{
char *singleString = *(tokens + i);
//printf("%s \n", singleString);

if (i==0)
{
continue;
}

else if(i==7)
{
data1.speed_km = strtof(singleString, NULL);
printf("VTG: %f \n", data1.speed_km); //it is not giving the accurate
    speed in our case

}
}
printf("\n");
free(tokens);
}

}


//printf("%lf %lf\n",data1.latitude,data1.longitude);

pclose(pipe);
fclose(fptr);
return 0;

}
```

Appendix C

Listing 3: LaTe socket opening both TCP and UDP for latency measurement with respect to speed

```
#include <unistd.h>
#include <stdio.h>
#include <assert.h>
#include <sys/socket.h>
#include <stdlib.h>
#include <netinet/in.h>
#include <string.h>
#include <pthread.h>
#include <time.h>
#include <arpa/inet.h>
#include <poll.h>
#include <sys/timerfd.h>
#include <inttypes.h>
#include <linux/sockios.h>
#include <linux/ethtool.h>
#include <sys/ioctl.h>
#include <fcntl.h>
#include <linux/if.h>
#include <errno.h>
#include <netdb.h>
#include <sys/types.h>
#define MAXLINE 1024
#define PORT 46001
#define BUFSIZE 256

pthread_mutex_t mutex;
float gpsValue = 0.0;
float LateValue = 0.0;
unsigned long gpsTime = 0;
unsigned long LateTime = 0;

char** str_split(char* a_str, const char a_delim);
int exitflag=0;
int pipeDescriptor[2]={0};
struct pollfd socketMon[2];
struct sockaddr_in cliaddr;
socklen_t len;

typedef struct
{
char** late_fields1;
int lamp_session_id;
int udp_sessioin_id;

} UDPThreadArgs;

void *gps(void *arg)
{
FILE *pipegps;
float var;
char line1[BUFSIZE];
pipegps= popen("./run.sh","r");
if (pipegps == NULL)
{
printf("CANT FIND DEVICE");
pthread_exit(0);
}


while (fgets(line1,sizeof(line1), pipegps) != NULL)
{

pthread_mutex_lock(&mutex);
gpsTime = time(NULL);

var = atof(line1);
```

```
gpsValue = var;
pthread_mutex_unlock(&mutex);

}
pclose(pipegps);
pthread_exit(0);

}




static pthread_t tid1;

void *udpthread(void *arg)
{

//struct UDPThreadArgs* udpargs=(struct UDPThreadArgs*)arg;
UDPThreadArgs udpArgs;
udpArgs = *( (UDPThreadArgs*) (arg) );
char buffer[MAXLINE];
char** udp_fields;
char** lateSplit;

socketMon[0].fd=udpArgs.udp_sessioin_id;
socketMon[0].revents=0;
socketMon[0].events=POLLIN;

socketMon[1].fd=pipeDescriptor[0];
socketMon[1].revents=0;
socketMon[1].events=POLLIN;
int n;
printf("i am here in udp\n");
while(1)
{
if(poll(socketMon,2,-1) > 0)
{
//fprintf(stdout,"revents: 0: %d 1: %d\n",socketMon[0].revents,
    socketMon[1].revents);
if(socketMon[0].revents & (POLLIN))
{
```

```
//printf("Read server value.\n");
// read socket
n = recvfrom(udpArgs.udp_sessioin_id, (char *)buffer, MAXLINE,
    MSG_DONTWAIT, ( struct sockaddr *) &cliaddr, &len);

//printf("Client: %s\n", buffer );
udp_fields = str_split(buffer, ',');
pthread_mutex_lock(&mutex);
LateTime = time(NULL);
LateValue=atof(udp_fields[3]);
pthread_mutex_unlock(&mutex);
if ( abs(LateTime - gpsTime) == 0 )
{
printf("latency %f, speed %f \n", LateValue, gpsValue);
//printf("speedTime %lu, gpsTime %lu \n", speedTime, gpsTime);
}
}

if(socketMon[1].revents > 0)
{
printf("Event\n");
break;
}
}
}

printf("POll ended\n");


return 0;
}




int main()
{



int server_fd, new_socket, valread,sockfd;
struct sockaddr_in address;
struct sockaddr_in servaddr,servaddrudp;
int opt = 1;
socklen_t addrlen = sizeof(address);
char buffertcp[1024] = {0};
char bufferudp[1024]= {0};
char ** lateSplit;
// pthread_t tid1;
pthread_t tid2;
char ** late_fields;
int sock_opt;
pthread_create(&tid2,NULL,gps,NULL);


// tcp socket create

if ((server_fd = socket(AF_INET, SOCK_STREAM, 0)) == 0)
{
perror("socket failed");
exit(EXIT_FAILURE);
}


if (setsockopt(server_fd, SOL_SOCKET, SO_REUSEADDR | SO_REUSEPORT,
&opt, sizeof(opt)))
{
perror("setsockopt");
exit(EXIT_FAILURE);
}

// Filling server information
servaddr.sin_family  = AF_INET; // IPv4
servaddr.sin_addr.s_addr = INADDR_ANY;
servaddr.sin_port = htons(46001);

//binding TCP socket
if (bind(server_fd, (struct sockaddr *)&servaddr,
sizeof(servaddr))<0)
{
perror("bind failed");
exit(EXIT_FAILURE);
```

```
}

printf("Before listen server\n");

if (listen(server_fd, 3) < 0)
{
perror("listen");
exit(EXIT_FAILURE);
}
/*OPENING UDP SOCKET AND BIND IT*/
printf("Before listen server\n");

// Creating socket file descriptor for UDP
if ( (sockfd = socket(AF_INET, SOCK_DGRAM, 0)) < 0 )
{
perror("socket creation failed");
exit(EXIT_FAILURE);
}

memset(&servaddrudp, 0, sizeof(servaddrudp));

servaddrudp.sin_family  = AF_INET; // IPv4
servaddrudp.sin_addr.s_addr = INADDR_ANY;
servaddrudp.sin_port = htons(46001);

// Bind the socket with the server address
if (bind(sockfd, (const struct sockaddr *)&servaddrudp,
sizeof(servaddrudp)) < 0 )
{
perror("bind failed");
exit(EXIT_FAILURE);
}

sock_opt=fcntl(sockfd,F_GETFL,NULL);
if(sock_opt<0) {
exit(EXIT_FAILURE);
}

// Set the non-blocking flag to the socket
if (fcntl(sockfd,F_SETFL,sock_opt | O_NONBLOCK)<0) {
```

```
exit(EXIT_FAILURE);
}


while(1)
{

if ((new_socket = accept(server_fd, (struct sockaddr *)&address,(
    socklen_t*)&addrlen))<0)
{
perror("accept");
exit(EXIT_FAILURE);
}

exitflag=0;

valread = read(new_socket, buffertcp, 1024);
printf("First Value: %s\n",buffertcp);

if(sizeof(buffertcp) == 0)
{
printf("Error: LaTe was disconnected too early.\n");
continue;
}

lateSplit = str_split(buffertcp, ',');
int lamp_session_id=atoi(lateSplit[1]);

if ((strcmp(lateSplit[0], "LaTeINIT") != 0 ) && (strcmp(lateSplit[0],
    "LaTeEND") != 0 ) )
{
printf("Expected a LaTeINIT or LateEND packet, but received %s \n",
    lateSplit[0]);
close(new_socket);
continue;
}
else if ((strcmp(lateSplit[0], "LaTeEND") == 0 ))
{
```

```
// finalate_fields = str_split(str_split( lateSplit[2],'=')[1],';');
    l_test_data_operations(decoded_data); // still have to define this
    function
printf("LateEnd \n");
close(new_socket);
continue;
}
else
{
printf("Received a %s \n",lateSplit[0]);
}

late_fields = str_split(str_split( lateSplit[2],'=')[1],';');
UDPThreadArgs udpTArgs;
udpTArgs.lamp_session_id = lamp_session_id;
udpTArgs.late_fields1 = malloc(sizeof(late_fields));
udpTArgs.late_fields1 = late_fields;
udpTArgs.udp_sessioin_id = sockfd;

if(pipe(pipeDescriptor)<0) {
fprintf(stderr,"Error: could not create the pipe for the graceful
    termination of the flush thread.\n");
return -1;
}

//creating thread for UDP

pthread_create(&tid1,NULL,udpthread,&udpTArgs);

printf("After Creating Thread");

valread = read(new_socket, buffertcp, 1024);
//printf("Second Value: %s\n",buffertcp);


if(sizeof(buffertcp) == 0)
{
printf("Error: LaTe was disconnected before LaTeEND could be received
    ..\n");
// terminate_udpserver(sockfd, lamp_session_id);
```

```
//pthread_join(tid1,NULL);
close(new_socket);
continue;
}


char** lateSplit2 = str_split(buffertcp, ',');

//printf("End Here: %s\n", lateSplit2[0] );


if (strcmp(lateSplit2[0], "LaTeEND") != 0 )
{
printf("Expected a LaTeEND packet, but received %s ",lateSplit2[0]);
printf("No final report data can be obtained.");

}

if( (strcmp(lateSplit2[0], "LaTeEND") == 0) )
{

printf("Received a %s packet", lateSplit2[0]);

if(write(pipeDescriptor[1],"\0",1)<0) {
fprintf(stderr,"Its termination will be forced.\n");
//pthread_cancel(tid1);
printf("loop cancel\n");
}

if ( atoi(lateSplit2[1]) != lamp_session_id  )
{
printf("Warning: data ignored. Expected test ID %d but received %d",
    lamp_session_id, atoi(lateSplit[1]));

}
else
{
if ( strcmp(lateSplit2[2], "srvtermination") != 0  )
{
// final_test_data_operations( lateSplit);
printf("Here ");
break;
}
}

}
}





//  close(pipeDescriptor[0]);
//close(pipeDescriptor[1]);
pthread_join(tid1,NULL);

return 0;
}

char** str_split(char* a_str, const char a_delim)
{
char** result    = 0;
size_t count     = 0;
char* tmp        = a_str;
char* last_comma = 0;
char delim[2];
delim[0] = a_delim;
delim[1] = 0;

/* Count how many elements will be extracted. */
while (*tmp)
{
if (a_delim == *tmp)
{
count++;
last_comma = tmp;
}
tmp++;
}
```

```
/* Add space for trailing token. */
count += last_comma < (a_str + strlen(a_str) - 1);

/* Add space for terminating null string so caller
knows where the list of returned strings ends. */
count++;

result = malloc(sizeof(char*) * count);

if (result)
{
size_t idx  = 0;
char* token = strtok(a_str, delim);

while (token)
{
assert(idx < count);
*(result + idx++) = strdup(token);
token = strtok(0, delim);
}
assert(idx = count - 1);
*(result + idx) = 0;
}

return result;
}
```

Appendix D

Listing 4: C code for latency measurement with respect to distance

```c
#include <unistd.h>
#include <stdio.h>
#include <assert.h>
#include <sys/socket.h>
#include <stdlib.h>
#include <netinet/in.h>
#include <string.h>
#include <pthread.h>
#include <time.h>
#include <arpa/inet.h>
#include <poll.h>
#include <sys/timerfd.h>
#include <inttypes.h>
#include <linux/sockios.h>
#include <linux/ethtool.h>
#include <sys/ioctl.h>
#include <fcntl.h>
#include <linux/if.h>
#include <errno.h>
#include <netdb.h>
#include <sys/types.h>
#include <math.h>
#define MAXLINE 1024
#define PORT 46001
#define BUFSIZE 256
#define M_PI 3.14159265358979323846
#define MIN(x, y) (((x) < (y)) ? (x) : (y))
```

```
pthread_mutex_t mutex;
float gpsValue = 0.0;
float LateValue = 0.0;
unsigned long gpsTime = 0;
unsigned long LateTime = 0;

char** str_split(char* a_str, const char a_delim);
double distance(double lat_1, double long_1, double lat_2,double
    long_2);
double toRadians( double degrees);
float decimal_deg(double value);

int exitflag=0;
int pipeDescriptor[2]={0};
struct pollfd socketMon[2];
struct sockaddr_in cliaddr;
socklen_t len;


typedef struct{
char name[6];
float timeZone;
double latitude;
char    D1;
double longitude;
float speed_kn;
float speed_km;
char D2;
char mode[2];
int satelites;
float dilution;
float antAltitude;
float GS;
float Differential;
int chksum;
} gps_data;

typedef struct
{
char** late_fields1;
```

```
int lamp_session_id;
int udp_sessioin_id;

} UDPThreadArgs;

void *gps(void *arg)
{
printf("Running in thread.");
FILE *pipe ;
gps_data data1;
char line [BUFSIZE];
char** tokens;

pipe= popen("./gpsscript.sh","r");
char str[7]="$GNGNS,";
double lat1=0.0,lat2=0.0,long1 = 0.0,long2 = 0.0;
int R=6371e3;
float rad=3.14/180;
printf("%f\n", rad);
if (pipe == NULL) {
printf("CANT FIND DEVICE");
pthread_exit(0);
}


while (fgets(line,sizeof(line), pipe) != NULL)
{
if (strncmp(line,str,7) == 0)
{
tokens = str_split(line, ',');

if (tokens == NULL) {
continue;
}

for (int i = 0; *(tokens+i); i++)
{
char *singleString = *(tokens + i);

//printf("%s \n", singleString);
```

```
if (i==0)
{
continue;
}

else if(i==2)
{
if (lat1 == 0) {
lat1 = strtod(singleString, NULL);
}
else {
data1.latitude = strtod(singleString, NULL);
lat2 = data1.latitude;
}

}
else if (i == 4)
{

if (long1 == 0 ){
long1 = strtod(singleString, NULL);
}
else {
data1.longitude =  strtod(singleString, NULL);
long2 = data1.longitude;
}



}



}


// checking if we got both first location and second location,
if ( lat1 != 0 && long1 != 0 && lat2 != 0 && long2 != 0)
{
// this place will run for second reading, because at first reading
   lat2 and long2 will be 0
```

```
double lat1_deg = decimal_deg(lat1);
double lat2_deg = decimal_deg(lat2);
double long1_deg = decimal_deg(long1);
double long2_deg = decimal_deg(long2);

pthread_mutex_lock(&mutex);
gpsTime = time(NULL);
gpsValue =distance(lat1_deg,long1_deg,lat2_deg,long2_deg);

pthread_mutex_unlock(&mutex);
// printf("%f\n", res );

}
free(tokens);
}
}
pclose(pipe);
pthread_exit(0);
}





static pthread_t tid1;

void *udpthread(void *arg)
{

//struct UDPThreadArgs* udpargs=(struct UDPThreadArgs*)arg;
UDPThreadArgs udpArgs;
udpArgs = *( (UDPThreadArgs*) (arg) );
char buffer[MAXLINE];
char** udp_fields;
char** lateSplit;

socketMon[0].fd=udpArgs.udp_sessioin_id;
socketMon[0].revents=0;
socketMon[0].events=POLLIN;
```

```
socketMon[1].fd=pipeDescriptor[0];
socketMon[1].revents=0;
socketMon[1].events=POLLIN;
int n;
printf("i am here in udp\n");
while(1)
{
if(poll(socketMon,2,-1) > 0)
{
//fprintf(stdout,"revents: 0: %d 1: %d\n",socketMon[0].revents,
    socketMon[1].revents);
if(socketMon[0].revents & (POLLIN))
{
//printf("Read server value.\n");
// read socket
n = recvfrom(udpArgs.udp_sessioin_id, (char *)buffer, MAXLINE,
    MSG_DONTWAIT, ( struct sockaddr *) &cliaddr, &len);

//printf("Client: %s\n", buffer );
udp_fields = str_split(buffer, ',');
pthread_mutex_lock(&mutex);
LateTime = time(NULL);
LateValue=atof(udp_fields[3]);
pthread_mutex_unlock(&mutex);
if ( abs(LateTime - gpsTime) == 0 )
{
printf("latency %f, distance %f \n", LateValue, gpsValue);
//printf("speedTime %lu, gpsTime %lu \n", speedTime, gpsTime);
}
}

if(socketMon[1].revents > 0)
{
printf("Event\n");
break;
}
}
}
```

```
printf("POll ended\n");


return 0;
}




int main()
{


int server_fd, new_socket, valread,sockfd;
struct sockaddr_in address;
struct sockaddr_in servaddr,servaddrudp;
int opt = 1;
socklen_t addrlen = sizeof(address);
char buffertcp[1024] = {0};
char bufferudp[1024]= {0};
char ** lateSplit;
// pthread_t tid1;
pthread_t tid2;
char ** late_fields;
int sock_opt;
pthread_create(&tid2,NULL,gps,NULL);


// tcp socket create

if ((server_fd = socket(AF_INET, SOCK_STREAM, 0)) == 0)
{
perror("socket failed");
exit(EXIT_FAILURE);
}


if (setsockopt(server_fd, SOL_SOCKET, SO_REUSEADDR | SO_REUSEPORT,
&opt, sizeof(opt)))
{
perror("setsockopt");
exit(EXIT_FAILURE);
}

// Filling server information
servaddr.sin_family  = AF_INET; // IPv4
servaddr.sin_addr.s_addr = inet_addr("192.168.1.9");
servaddr.sin_port = htons(46001);

//binding TCP socket
if (bind(server_fd, (struct sockaddr *)&servaddr,
sizeof(servaddr))<0)
{
perror("bind failed");
exit(EXIT_FAILURE);
}

printf("Before listen server\n");

if (listen(server_fd, 3) < 0)
{
perror("listen");
exit(EXIT_FAILURE);
}
/*OPENING UDP SOCKET AND BIND IT*/
printf("Before listen server\n");

// Creating socket file descriptor for UDP
if ( (sockfd = socket(AF_INET, SOCK_DGRAM, 0)) < 0 )
{
perror("socket creation failed");
exit(EXIT_FAILURE);
}

memset(&servaddrudp, 0, sizeof(servaddrudp));

servaddrudp.sin_family  = AF_INET; // IPv4
servaddrudp.sin_addr.s_addr = INADDR_ANY;
servaddrudp.sin_port = htons(46001);

// Bind the socket with the server address
```

```
if (bind(sockfd, (const struct sockaddr *)&servaddrudp,
sizeof(servaddrudp)) < 0 )
{
perror("bind failed");
exit(EXIT_FAILURE);
}

sock_opt=fcntl(sockfd,F_GETFL,NULL);
if(sock_opt<0) {
exit(EXIT_FAILURE);
}

// Set the non-blocking flag to the socket
if (fcntl(sockfd,F_SETFL,sock_opt | O_NONBLOCK)<0) {
exit(EXIT_FAILURE);
}


while(1)
{

if ((new_socket = accept(server_fd, (struct sockaddr *)&address,(
    socklen_t*)&addrlen))<0)
{
perror("accept");
exit(EXIT_FAILURE);
}

exitflag=0;

valread = read(new_socket, buffertcp, 1024);
printf("First Value: %s\n",buffertcp);

if(sizeof(buffertcp) == 0)
{
printf("Error: LaTe was disconnected too early.\n");
continue;
}

lateSplit = str_split(buffertcp, ',');
```

```
int lamp_session_id=atoi(lateSplit[1]);

if ((strcmp(lateSplit[0], "LaTeINIT") != 0 ) && (strcmp(lateSplit[0],
    "LaTeEND") != 0 ) )
{
printf("Expected a LaTeINIT or LateEND packet, but received %s \n",
    lateSplit[0]);
close(new_socket);
continue;
}
else if ((strcmp(lateSplit[0], "LaTeEND") == 0 ))
{
// finalate_fields = str_split(str_split( lateSplit[2],'=')[1],';');
    l_test_data_operations(decoded_data); // still have to define this
    function
printf("LateEnd \n");
close(new_socket);
continue;
}
else
{
printf("Received a %s \n",lateSplit[0]);
}

late_fields = str_split(str_split( lateSplit[2],'=')[1],';');
UDPThreadArgs udpTArgs;
udpTArgs.lamp_session_id = lamp_session_id;
udpTArgs.late_fields1 = malloc(sizeof(late_fields));
udpTArgs.late_fields1 = late_fields;
udpTArgs.udp_sessioin_id = sockfd;

if(pipe(pipeDescriptor)<0) {
fprintf(stderr,"Error: could not create the pipe for the graceful
    termination of the flush thread.\n");
return -1;
}

//creating thread for UDP

pthread_create(&tid1,NULL,udpthread,&udpTArgs);
```

```
printf("After Creating Thread");

valread = read(new_socket, buffertcp, 1024);
//printf("Second Value: %s\n",buffertcp);


if(sizeof(buffertcp) == 0)
{
printf("Error: LaTe was disconnected before LaTeEND could be received
    ..\n");
// terminate_udpserver(sockfd, lamp_session_id);
//pthread_join(tid1,NULL);
close(new_socket);
continue;
}


char** lateSplit2 = str_split(buffertcp, ',');

//printf("End Here: %s\n", lateSplit2[0] );


if (strcmp(lateSplit2[0], "LaTeEND") != 0 )
{
printf("Expected a LaTeEND packet, but received %s ",lateSplit2[0]);
printf("No final report data can be obtained.");

}

if( (strcmp(lateSplit2[0], "LaTeEND") == 0) )
{

printf("Received a %s packet", lateSplit2[0]);

if(write(pipeDescriptor[1],"\0",1)<0) {
fprintf(stderr,"Its termination will be forced.\n");
//pthread_cancel(tid1);
printf("loop cancel\n");
}
if ( atoi(lateSplit2[1]) != lamp_session_id  )
{
printf("Warning: data ignored. Expected test ID %d but received %d",
    lamp_session_id, atoi(lateSplit[1]));

}
else
{
if ( strcmp(lateSplit2[2], "srvtermination") != 0  )
{
// final_test_data_operations( lateSplit);
printf("Here ");
break;
}
}

}
}






//  close(pipeDescriptor[0]);
//close(pipeDescriptor[1]);
pthread_join(tid1,NULL);

return 0;
}

char** str_split(char* a_str, const char a_delim)
{
char** result    = 0;
size_t count     = 0;
char* tmp        = a_str;
char* last_comma = 0;
char delim[2];
delim[0] = a_delim;
delim[1] = 0;
```

```
/* Count how many elements will be extracted. */
while (*tmp)
{
if (a_delim == *tmp)
{
count++;
last_comma = tmp;
}
tmp++;
}

/* Add space for trailing token. */
count += last_comma < (a_str + strlen(a_str) - 1);

/* Add space for terminating null string so caller
knows where the list of returned strings ends. */
count++;

result = malloc(sizeof(char*) * count);

if (result)
{
size_t idx  = 0;
char* token = strtok(a_str, delim);

while (token)
{
assert(idx < count);
*(result + idx++) = strdup(token);
token = strtok(0, delim);
}
assert(idx = count - 1);
*(result + idx) = 0;
}

return result;
}
```

```
float decimal_deg(double value)
{
//concatenate int part and dec part
int degrees = floor(value / 100);
//printf("degree %d\n", degrees );
float minutes = value - (degrees * 100);
// printf("%f\n",minutes );
value = degrees + minutes/60;
return value;

}
double toRadians( double degrees)
{
double one_deg = (M_PI) / 180;
return (one_deg * degrees);
}

double distance(double lat_1, double long_1, double lat_2,double
    long_2)
{
double R = 6371e3; //in meters and so the distance will also be in m/s
double latitude1,latitude2,longitude1,longitude2;
latitude1=lat_1;
latitude2=lat_2;
longitude1=long_1;
longitude2=long_2;
latitude1=toRadians(latitude1);
latitude2=toRadians(latitude2);
longitude1=toRadians(longitude1);
longitude2=toRadians(longitude2);


double dlong = longitude2 - longitude1;
double dlat = latitude2 - latitude1;

double ans = pow(sin(dlat / 2), 2) +   cos(latitude1) * cos(latitude2)
     *  pow(sin(dlong / 2), 2);

ans = 2 * asin(sqrt(ans));
ans = ans * R;
return ans;
}
```

Bibliography

[1] US Congress, *Intermodal Surface Transportation Efficiency Act. Pub. L. 102-240, 105 Stat. 1914, 1991*, 2008.

[2] *Global status report on road safety 2013*, 2015. [Online]. Available: `https://www.who.int/violence_injury_prevention/road_safety_status/2013/en/`.

[3] F. Raviglione, *Implementation of the IEEE 802.11p protocol on embedded systems*, 2018. [Online]. Available: `https://webthesis.biblio.polito.it/9525/`.

[4] *Roadmap to a Single European Transport Area – Towards a competitive and resource efficient transport system*, 2016. [Online]. Available: `https://www.eea.europa.eu/policy-documents/roadmap-to-a-single-european`.

[5] P. Shrivastava, V. Shubham, S. Vijay Vargiy, and vikramlodhi, «Vehicle to Vehicle Safety Device An Ease for Safe Driving VIKRAM LODHI,» May 2015.

[6] C. Schindelhauer, «Mobility in wireless networks,» in *In 32nd Annual Conference on Current Trends in Theory and Practice of Informatics, Czech*, 2006.

[7] G. Karagiannis, O. Altintas, E. Ekici, G. Heijenk, B. Jarupan, K. Lin, and T. Weil, «Vehicular Networking: A Survey and Tutorial on Requirements, Architectures, Challenges, Standards and Solutions,» *IEEE Communications Surveys Tutorials*, vol. 13, no. 4, pp. 584–616, 2011.

[8] A. D. Devangavi and R. Gupta, «Routing protocols in VANET — A survey,» in *2017 International Conference On Smart Technologies For Smart Nation (SmartTechCon)*, 2017, pp. 163–167.

[9] R. Brendha and V. S. J. Prakash, «A survey on routing protocols for vehicular Ad Hoc networks,» in *2017 4th International Conference on Advanced Computing and Communication Systems (ICACCS)*, 2017, pp. 1–7.

[10] *explore the technology*. [Online]. Available: `https://5gaa.org/5g-technology/c-v2x/`.

[11] S. A. Ahmad and M. Shcherbakov, «A Survey on Routing Protocols in Vehicular Adhoc Networks,» in *2018 9th International Conference on Information, Intelligence, Systems and Applications (IISA)*, 2018, pp. 1–8.

[12] H. Toulni and B. Nsiri, «A hybrid routing protocol for vanet using ontology,» *Procedia Computer Science*, vol. 73, pp. 94–101, 2015.

[13] H. Trivedi, P. Veeraraghavan, S. Loke, A. Desai, and J. Singh, «Routing mechanisms and cross-layer design for Vehicular Ad Hoc Networks: A survey,» in *2011 IEEE Symposium on Computers Informatics*, 2011, pp. 243–248.

[14] G. Ciccarese, M. De Blasi, P. Marra, and C. Palazzo, «A timer-based Intelligent Flooding Scheme for VANETs,» in *2010 7th International Symposium on Communication Systems, Networks Digital Signal Processing (CSNDSP 2010)*, 2010, pp. 377–381.

[15] F. Kamal, E. Lou, and V. Zhao, «Design and validation of a small-scale 5.9 GHz DSRC system for vehicular communication,» in *2012 25th IEEE Canadian Conference on Electrical and Computer Engineering (CCECE)*, 2012, pp. 1–4.

[16] N. Vivek, P. Sowjanya, B. Sunny, and S. V. Srikanth, «Implementation of IEEE 1609 WAVE/DSRC stack in Linux,» in *2017 IEEE Region 10 Symposium (TENSYMP)*, 2017, pp. 1–5.

[17] Y. Li, «An overview of the DSRC/WAVE technology,» vol. 74, Jan. 2012, pp. 544–558. DOI: `10.1007/978-3-642-29222-4_38`.

[18] *Access layer specification for Intelligent Transport Systems operating in the 5 GHz frequency band*, 2012. [Online]. Available: `https://www.etsi.org/deliver/etsi_en/302600_302699/302663/01.02.00_20/en_302663v010200a.pdf`.

[19] *Intelligent Transport Systems (ITS); Facilities layer function*, Jul. 2019. [Online]. Available: `https://www.etsi.org/standards`.

[20] L. Andia, M. Carlsson, and A. Freund, «A reconfigurable IEEE 802.11 p/ARIB RF transceiver for V2X,» 2014.

[21] *Cisco and Cohda wireless*. [Online]. Available: `https://www.chipestimate.com/Cisco-NXP-Invest-in-Cohda-Wireless-to-Enable-the-Connected-Car/Semiconductor-IP-Core/news/24985`.

[22] *Cohda wireless hardware*. [Online]. Available: `https://cohdawireless.com/solutions/hardware/`.

[23] A. Sassi, F. Charfi, L. Kamoun, Y. Elhillali, and A. Rivenq, «Experimental measurement for vehicular communication evaluation using OBU ARADA System,» in *2015 International Wireless Communications and Mobile Computing Conference (IWCMC)*, 2015, pp. 1358–1364.

[24] *APU Boards*. [Online]. Available: `https://www.pcengines.ch/apu.htm`.

[25] *APUID Boards*. [Online]. Available: `https://www.pcengines.ch/apu1d.htm`.

[26] *Savari solutions*, 2016. [Online]. Available: `https://savari.net/solutions/`.

[27] francescoraves483, *francescoraves483/OpenWrt-V2X*, 2019. [Online]. Available: `https://github.com/francescoraves483/OpenWrt-V2X`.

[28] S. Eichler, «Performance Evaluation of the IEEE 802.11p WAVE Communication Standard,» in *2007 IEEE 66th Vehicular Technology Conference*, 2007, pp. 2199–2203.

[29] D. Jiang and L. Delgrossi, «IEEE 802.11p: Towards an International Standard for Wireless Access in Vehicular Environments,» in *VTC Spring 2008 - IEEE Vehicular Technology Conference*, 2008, pp. 2036–2040.

[30] Lusheng Miao, K. Djouani, B. J. Van Wyk, and Y. Hamam, «Performance evaluation of IEEE 802.11p MAC protocol in VANETs safety applications,» in *2013 IEEE Wireless Communications and Networking Conference (WCNC)*, 2013, pp. 1663–1668.

[31] IEEE, «Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications,» p. 2321, 2016.

[32] ——, «IEEEStd802.11-2016Part11:WirelessLANMediumAccessControl(MAC)and Physical Layer (PHY) Specifications, IEEE Std 802.11™-2016, IEEE Computer Society,2016,» *IEEE*, 2016.

[33] Z. Wang and X. Guo, «Priority-based parameter performance optimization for EDCA,» in *Proceedings of 2013 3rd International Conference on Computer Science and Network Technology*, 2013, pp. 685–688.

[34] IEEE, «ieee standard for wireless access in vehicular environments (wave) – multi-channel operation,» *IEEE*, p. 17, 2016.

[35] F. Raviglione, M. Malinverno, and C. Casetti, «A Flexible, Protocol-Agnostic Latency Measurement Platform,» in *2019 IEEE 90th Vehicular Technology Conference (VTC2019-Fall)*, 2019, pp. 1–5.

[36] *Controller Area Network (CAN) overview*. [Online]. Available: `https://www.ni.com/it-it/innovations/white-papers/06/controller-area-network--can--overview.html`.

[37] H. Chen and J. Tian, «Research on the Controller Area Network,» in *2009 International Conference on Networking and Digital Society*, vol. 2, 2009, pp. 251–254.

[38] *KVASER LEAF RANGE*. [Online]. Available: `http://ampliconme.com/product/kvaser-leaf-range/`.

[39] *Setting up the CAN*. [Online]. Available: `https://elinux.org/Bringing_CAN_interface_up`.

[40] *Controller Area Network*. [Online]. Available: `https://www.kernel.org/doc/Documentation/networking/can.txt`.

[41] *CAN loopback interface*. [Online]. Available: `https://www.kernel.org/doc/html/latest/networking/can.html#socketcan-local-loopback1`.

[42] Mortedamos, *mortedamos/vehicle-hacking*. [Online]. Available: `https://github.com/mortedamos/vehicle-hacking/wiki/Vehicle-Hacking-Setup-Guide:-Part-1:-Virtual-CAN-Interface`.

[43] *Space Segment*. [Online]. Available: `https://www.gps.gov/systems/gps/space/`.

[44] *Qualcomm completes assisted-gps test calls for wcdma/gsm/gprs networks*. [Online]. Available: `https://www.spacedaily.com/news/gps-04zzzzh.html`.

[45] *GPS Applications*. [Online]. Available: `https://www.gps.gov/applications/`.

[46] A. Kolomijeca, J. A. López-Salcedo, E. Lohan, and G. Seco-Granados, «GNSS applications: Personal safety concerns,» in *2016 International Conference on Localization and GNSS (ICL-GNSS)*, 2016, pp. 1–5.

[47] *NMEA*. [Online]. Available: `https://web.archive.org/web/20140215150802/http://www.kh-gps.de/nmea.faq`.

[48] *National Marine Electronics Association*. [Online]. Available: `https://www.nmea.org/content/STANDARDS/NMEA_0183_Standard`.

[49] D. DePriest. [Online]. Available: `https://www.gpsinformation.org/dale/nmea.htm`.

[50] U-blox, *Product Summary UBX-M8030 Versatile u-blox M8 GNSS chips UBX-15029937 - R07*. [Online]. Available: `www.u-blox.com`.

[51] *Geoid sepration of Earth*. [Online]. Available: `https://www.esri.com/news/arcuser/0703/geoid1of3.html`.

[52] C. Veness, *Haversine for calculating distance*. [Online]. Available: `https://www.movable-type.co.uk/scripts/latlong.html`.

[53] francescoraves483, *francescoraves483/LaMP*$_LaTe$. [Online]. Available: `https://github.com/francescoraves483/LaMP_LaTe`.

[54] F. Raviglione, *LaTe Help*. [Online]. Available: `https://francescoraves483.github.io/LaMP_LaTe/`.