Escuela de
Ingeniería y Arquitectura
Universidad Zaragoza

DTU Danmarks Tekniske Universitet

Safety-Critical Platooning Function Based On Wireless Communication Using Cooperative Robots

Trabajo de fin de grado (12 ECTS) realizado en la Universidad técnica de Dinamarca (DTU) entre el 1 de febrero de 2017 y el 31 de mayo de 2017

realizada por Ana Lasheras Mas

Trabajo supervisado en DTU por Paul Pop y Fotis Foukalas
Ponente en la Escuela de ingeniería y arquitectura (EINA) Ana Cristina Murillo

TRABAJOS DE FIN DE GRADO / FIN DE MÁSTER

Escuela de Ingeniería y Arquitectura
Universidad Zaragoza

DECLARACIÓN DE AUTORÍA Y ORIGINALIDAD

(Este documento debe acompañar al Trabajo Fin de Grado (TFG)/Trabajo Fin de Máster (TFM) cuando sea depositado para su evaluación).

D./Dª. Ana Lasheras Mas,

con nº de DNI 73134833P en aplicación de lo dispuesto en el art. 14 (Derechos de autor) del Acuerdo de 11 de septiembre de 2014, del Consejo de Gobierno, por el que se aprueba el Reglamento de los TFG y TFM de la Universidad de Zaragoza,

Declaro que el presente Trabajo de Fin de (Grado/Máster) Grado, (Título del Trabajo) "Safety-Critical Platooning Function Based on Wireless Communication Using Cooperative Robots",

es de mi autoría y es original, no habiéndose utilizado fuente sin ser citada debidamente.

Zaragoza, 12 de Julio del 2017

Fdo: [illegible]

Abstract

Platooning functionality is based on a string of vehicles, driving in the same direction, where each vehicle drives one after another keeping the smallest security distance possible to avoid collisions. This functionality can be provided in self-driving, but also, human driving vehicles. Communication between vehicle allows to keep a small security distance inter-vehicles, while it reduces fuel consumption and it makes more compact the traffic in the roads without increasing the risk of collisions.

This thesis proposed some basic platooning scenarios and defines a distributed control plane, based on a communication and a controller protocol, to achieve some scenarios using wireless communication between the entities in the system. The scenarios and the protocol stack proposed are based on previous work from academic articles. These papers explain and prove that the standards and technology used is adequate for the use case developed in this thesis.

The main task of this thesis is to develop a high-level communication protocol that provides a set of messages to achieve platooning functionality and supports the use cases defined in the thesis. In addition, the designed communication protocol is tested in a real platform using simple controller functionalities.

This thesis also analyses the protocol and finds solutions to support communication failures, while it evaluates the protocol with evaluation metrics.

Tests on the real platform and metrics show that the communication protocol is valid to achieve the defined use cases.

Resumen

Platooning define la conducción de un grupo de vehículos con un destino o dirección común, en la que el primer vehículo decide sobre la conducción (p. ej. aceleración, cambio de carril, desvío, etc.) y los demás se restringen a seguirlo. Estos vehículos pueden ser conducidos por un ser humano o de forma automática. Además, los vehículos comparten información entre sí permitiendo mantener una distancia de seguridad mínima entre ellos, lo que reduce el consumo de combustible y compacta el tráfico en las carreteras sin incrementar el riesgo de colisiones.

Esta tesis propone una serie de escenarios basados en platooning y define un plano de control distribuido entre los vehículos que integra un protocolo de comunicación y un protocolo de control necesarios para conseguir realizar los escenarios de platooning, propuestos. Tanto los escenarios como la pila de protocolos para la comunicación que se han propuesto están basados en trabajo previo descrito en artículos de investigación. Estos artículos explican y prueban estándares y tecnologías adecuadas para desarrollar las funcionalidades de platooning definidas en la tesis.

La principal tarea en esta tesis es el diseño de un protocolo de comunicación en alto nivel, capa de aplicación del modelo OSI, el cual define un conjunto de mensajes que al combinarlos permiten implementar una serie de funciones de platooning, en concreto permiten la implementación de los escenarios descritos en la tesis. Además de diseñar este protocolo de comunicación, se han implementado y probado en una plataforma real sencillas funciones de platooning.

En la tesis también se analiza el protocolo, se buscan soluciones para soportar fallos de comunicación en la implementación y se evalúa la actuación de la implementación en base a unas métricas previamente definidas.

Los test realizados sobre la implementación y los resultados obtenidos en la evaluación muestran que el protocolo diseñado permite ejecutar correctamente los escenarios de platooning deseados.

Acknowledgements

First, I would like to thank my supervisor Paul Pop and co-supervisor Fotis Foukalas for giving me the opportunity to work on this interesting project, and also for their instructions and advises to can finalise my report properly.

I also would like to thank Ana Cristina Murillo for being my presenter in my home university, University of Zaragoza, and for being so helpful giving recommendations about the project.

Furthermore, I thank my family for supporting me always, and for letting me make the most of the opportunity to go one year to DTU in order to finalise my studies and do my thesis.

Finally, I would like to thank my wonderful boyfriend David Nicuesa for always being there when I need him, and for encouraging me during the development of this thesis.

Contents

List of Figures

List of Tables

1. Introduction

This chapter introduces the terms of platooning and vehicular cyber-physical system (vCPS) used through the thesis. Then, there is a section reviewing previous work related to the objectives of the thesis, and a review of standard messages to be exchanged within a platoon. Next section reviews the many possible communication protocols and standards to use in the implementation of the Cooperative Driving Messages (CDM) protocol and selects a certain protocol to use in the design and the implementation. Later, platoon scenarios are defined to provide a design that can support them. Finally, the last section describes the objectives of the thesis and summarises the structure of this report.

1.1 Platooning

Platooning is a formation where a set of vehicles follows one another in the same direction. These vehicles communicate between them sharing breaking and leaving information to the other vehicles in the platoon. This information helps vehicles to take decisions in its outputs (eg. its speed) to maintain a safe driving while keeping a small distance between vehicles. This small distance allows increasing traffic throughput without compromising safety reducing road congestion. It also decreases fuel consumption by reducing the aerodynamic drag when vehicles drive very close one another.

The use of platoon is mainly focused on road/highways where the number of interruptions is limited as there are no traffic lights, pedestrian crossings or stop signs that could split the platoon. Platoons can be also used in other fields like agriculture, where the obstacles are also limited.

Platoons are composed of vehicles that can play the role of *leader* or *follower*(figure 1.1). The role of leader can be just played by one vehicle, in concrete by the first vehicle in the platoon. The leader vehicle drives freely, it can choose the desired velocity and change that velocity whenever it wants. On the contrary, the follower vehicles, that drive behind the follower, must follow the speed indicated by the leader vehicle to avoid collisions and keep a defined security distance between vehicles. For that reason, the leader vehicle communicates with the follower vehicles to indicate its speed and the possible changes in its movement. As it seems, the leader is a coordinator that gives some rules about how the follower must move. The leader coordinates also the maneuvers[1] within the platoon, by receiving maneuver's petitions, computing these requests and answering back.

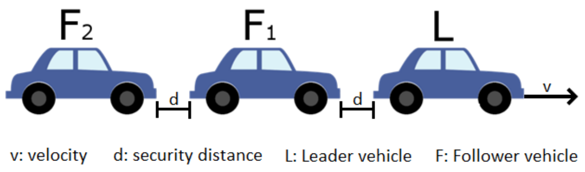


Figure 1.1: Platoon of three vehicles

[1]Maneuver: Change of a vehicle position in the platoon. Generally, a vehicle joining or leaving the platoon.

1.2 Vehicular Cyber-Physical System (vCPS)

A Cyber-Physical System (CPS) is computer-based machines that integrate physical and computational elements that interact between them in order to provide a task. CPS can integrate communication mechanisms and input/output sensors allowing to interact with other CPSs using the network connection. This system can compute actions based on algorithms taking inputs from their knowledge (ie. sensors or previous actions) and from the communication network.

Vehicular Cyber-Physical System (vCPS) are CPS integrated into a vehicle or mobile robot. The term vCPS is used in this thesis to avoid the distinction between vehicles and mobile robots and to indicate that these devices have some embedded computation mechanisms.

1.3 Bibliography review

Nowadays many researchers are focused on the study of vehicular Cyber-Physical Systems (vCPS) cooperating with each other to provide a specific task as a way to improve driving remaining safety in scenarios like platooning and vehicle-to-vehicle (V2V) or vehicle-to-infrastructure (V2I), but also to automate and improve tasks' execution of tractors in agriculture or forklifts in a warehouse.

Initially, it has been done a paper research to study the different use cases and system models discussed and proved in the literature to understand better the topic and know more about the most recent investigations. The paper research is mainly focused in papers about platooning and formation control theme, chiefly communication-based papers. Moreover, it has tried to read papers as recently as possible, to get an idea of what is really interesting now and what is left to be done to try to achieve it in the thesis.

In [14] different updating schemes are proposed to communicate between vehicles within the platoon. These schemes allow to join and leave the platoon taking into account that if the leader vehicle leaves the platoon then another vehicle must assume the leadership.

A chain of platoons (multi-platooning) that communicates in both directions using 802.11p and DCF for the inter-platoons' communication, where vehicles can join/leave a platoon in terms of the following the same destination is described in [17].

[20] describes a platoon management protocol for Cooperative Adaptive Cruise Control (CACC) vehicles using wireless communication that allows merging two platoons into one, split one platoon into two new platoons and lane-change (join or leave a platoon).

In conference [21], it is described a use case of a platoon in a traffic jam situation where the leader vehicle generates traffic shock waves by changing its cruising speed frequently. It also provide simulation with hundreds of platoons, but without inter-platooning communication.

The use case of platoons' emergency braking scenario with human-driving vehicles around is experimented with real vehicles and simulated in [22] after the investigation of different communication strategies taking into account controller requirements and the reliability of wireless communications for platooning under high channel load.

[23] analyses the impediments that human-driven vehicles can cause while a vehicle is trying

to join a platoon in the road, defines a communication protocol on the application layer that considers packet failures and simulates the execution of this protocol in some join scenarios.

In [30], vehicular communication is shown as a particular case of ad-hoc networks and it is reviewed many possibilities for group formation described in the literature, to define the interconnection and exchange of information between vehicles.

1.4 Standard messages' review and application discussion

The main messages exchange between vehicles in a platooning scenario are ones that contain vehicle profile's information to keep a certain distance between vehicles and to avoid collisions while string stability[2] is provided.

Cooperative Awareness Messages (CAM)[12] standard, defined by the European Telecommunication Standards Institute (ETSI), determines the relevant information to exchange from vehicle to vehicle (V2V) and between vehicles and roadside infrastructure(V2I/I2V). This *roadside* information represent knowledge about stations on the roadside that assist the communication between vehicles and provide a general view's information about roads to improve traffic (eg. reduce traffic jams).

This project does not consider roadside communication, for that reason, CAM messages are not used exactly like they are defined in the standard but *simpler* messages with just data needed to keep the distance between vehicles and not roadside information. The information communicated between vehicles in addition to data collected from onboard sensors helps to improve the controller output's accuracy reducing the propagation of inaccurate information which could lead into changes in the distance kept between vehicles and in the worst-case collisions.

The data included in the messages that the studied papers define and CAMs information fields have been analysed to decide which data is needed to include in the messages for this project. Moreover, the sending frequency used in the implementation is based on the CAM standard[12]. This standard states that CAM messages need to be sent in an interval between 0.1 and 1 second.

Apart from these *information* messages, it is also needed to have a set of *event* messages to notify vehicles about formation changes or maneuvers (eg. join/leave platoon), group definition events (eg. change id information) and emergency messages (eg. decelerate order because there is an obstacle in the way). ETSI does not have a standard defining the messages to achieve these cases, but it defines Decentralised Environmental Notification Messages(DENM)[13] to alert vehicles of potentially dangerous events (eg. constructing zones or road narrowing) that can abort maneuvers requests in case of danger conditions.

This project assumes that the road is free of obstacles, being the vehicles the only *potential* obstacles in case of stop. For that reason, DENM messages are not considered in the project. Although, it is needed to define another kind of messages to can achieve maneuvers and to determine the platoon's group and leadership.

[2]String stability: Reduction of disturbances propagated through the platoon.

1.5 Communication protocols search

There are many protocols and standards possibilities to implement the communication for a platooning case. Papers based on real vehicles platooning use a modification of the WiFi standard (IEEE 802.11) in the physical (PHY) and medium-access control (MAC) layers. This amendment, called IEEE 802.11p, adds wireless access in vehicular environments (WAVE) stack. This standard provides a 75 MHz bandwidth in 5.9 GHz (5.85-5.925 GHz) frequency band. The channels provided in IEEE 802.11p have a size of 10 MHz to support many parallel applications at a time, but because of this size, there can be channel congestion that is solved combining two of these channels to have a channel of 20 MHz. It maintains the Orthogonal Frequency Division Multiplexing (OFDM) technique provided in 802.11 standard. The main change in IEEE 802.11p standard comparing to IEEE 802.11 is the use of a communication Outside the Context of BSS(OCB), where BSS is a Basic Service Set that is a set of stations that exchange data information. It also introduces a new time management frame based on Timing Advertisements (TA) and a frame to identify organisations called Vendor-Specific Action(VSA). But, the devices used to test the platooning function does not use IEEE 802.11p, but another WiFi protocol, that it cannot be modified to emulate the IEEE 802.11p. For that reason, the search for a communication protocol is based on upper layers of the communication stack, in particular, network, transport, and application layers.

Network layer protocol

The main protocol used in the network layer is IP, but also WSMP is a possibility defined by WAVE[19] and used in [20].

IP protocol is a very known and easy to way to communicate two entities, while WSMP is not to spread and there are no many documentation and examples to apply it. So, as both of them are solutions defined by the Dedicated Short Range Communication (DSRC) in the WAVE stack the IP protocol has been selected because the simplicity and knowledge to use it.

Transport layer protocol

The main protocols for transport layer are UDP and TCP, but also WSMP. The protocol WSMP must be used in the network and transport layer so as IP protocol has been selected in the network layer, then WSMP has been discarded to use it in the transport layer as it cannot be combined with IP.

UDP is a non-connection oriented protocol, what means that every message is sent independently of the previous one. In addition, it is an unreliable protocol so it does not guarantee package delivery, ordering, or duplicates. However, it can be a good solution to communicate, as it is a non-connection oriented, it is not possible to lose the connection between devices and if the devices move out of the communication range they just lost the reception of some messages but when they move again inside the range they can receive messages again. Nevertheless, this is not a problem in our system as the communication range for 802.11n goes between 70 and 250 meters[31] and the mobile robots are not going to use such a distance in the experiment as the real vehicle could do.

TCP is a connection-oriented protocol, and for that reason, provides reliable communication. It makes sense to use it in the application that will be developed as the communication links between vCPSs are previously-defined. In addition, TCP/IP is mostly used in multi-agent communication as [32] cites. Also, it is a good option because it should not be a problem losing the connection at least not from a loss of communication range point of view as the robots will not move too far as real vehicles could do.

Moreover, paper [32], presents an overview of the communication protocols used between mobile robots, states that TCP/IP is mostly used in multi-agent communication as it ensures stability and reliability. But UDP is a possible solution to support real-time systems and it is a more lightweight protocol comparing to TCP/IP. So both protocols can be a good option.

Application layer protocol

There are many options of application protocols developed for Machine to Machine (M2M) communication or Internet of Thing (IoT). One of the most popular message-Oriented framework based on the publisher-subscriber paradigm is MQTT.

MQTT allows receiving data only from the interested nodes send. This way is very easy to send information to more than one node in a group, like in a multicast communication. MQTT is a *light weight* messaging protocol based on TCP/IP. It is designed to be used in networks where the bandwidth is limited and it allows to have quality of service in the communication. Papers like [29] and [28] proposed a solution for mobile robots using publish/subscribe solutions. Also, the community of the operative system selected, ev3dev, suggest the use of MQTT for communication between robots[27].

There is an extension of MQTT called MQTT-S or MQTT-SN[18] focused on wireless sensor networks, that allows to use UDP in the transport layer what remove handshake and make it more lightweight. But the problem is that there is no many implementation possibilities and documentation about this extension as it is quite new comparing to the original MQTT.

Another possible communication approach is a blackboard model where there is no direct communication between nodes so the robots must not be notified when a message arrives. This one is not a good solution as it is important to know exactly when a message has arrived, mainly in emergency message, notification events, etc.

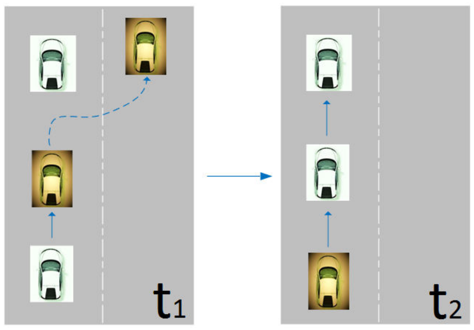
One more communication solution can be the Constrained Application Protocol (CoAP) that uses a similar communication to HTTP, called REST. This protocol is based on IP network layer protocol and UDP transport layer and it supports a reliable mode by using confirmation messages.

Paper [25] evaluate (CoAP) and MQTT-SN protocols concluding that MQTT-SN has better average time for message transmission than CoAP. It also shows that CoAP is more power efficient than MQTT but CoAP consumes more bandwidth than MQTT.

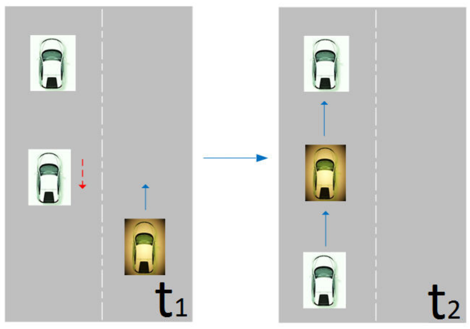
For that reason, MQTT has been selected in the application layer to implement the communication protocol that supports platooning functionality.

1.6 Definition of platooning scenarios

This section defines two platooning scenarios (figure 1.2) taking ideas from the papers reviewed in the previous section 1.3. These scenarios are the objective of the thesis in terms of design, as the distributed control plane must be able to execute these scenarios by exchanging a sequence of messages defined in the communication protocol that triggers the execution of certain controller functionalities.



(a) Scenario: vCPS leaves and joins again the platoon at the back



(b) Scenario: vCPS joins the platoon in the middle

Figure 1.2: Objective platooning scenarios

The scenario defined in figure 1.2a shows how a vCPS that is part of the platoon leaves the platoon, and after leaving, it joins again at the back. This scenario can be useful to exchange the position of vCPSs with different quantity of fuel left, so the vCPS with more fuel should be at the beginning of the platoon and the vCPS with less at the end as the aerodynamic drag is bigger for the first vCPS what increases its fuel consumption. The vCPS with less fuel needs to refill earlier what implies to leave the platoon sooner, and it is always easy to leave from the back as this maneuver does not interfere with the other vCPSs in the platoon.

Figure 1.2b describes a scenario where a vCPS outside the platoon wants to join a certain platoon. This joining can be done in any position of the platoon, but to *complicate* the maneuver the join has been depicted in the middle. This scenario is very common, as a vCPS that wants to join a platoon can be in any location and the platoon can be quite long so it is not efficient to make that vCPS wait until it can join the platoon at the back. It is better to create a gap in the more suitable position for that vCPS to join.

1.7 Objectives and structure of the thesis

The main objectives of the thesis are the design of a communication application protocol to communicate vehicular profile information and events between vCPSs in order to provide platooning functionality, and the implementation and test of this protocol on a real device.

The initial tasks needed to provide these objectives is the study of previous work related with platooning and cooperative task topics based on mobile robots, vCPS implementation or simulation and mainly focused on the communication between the entities. After reviewing the literature, section 1.3, it is needed to define the integration of the communication protocol and the controller functionality required to achieve platooning, even the controller is not the main focus of this thesis. This definition is done in section 2. Then, it comes the main task of designing the communication application protocol, section 3, by indicating the type of communication links needed between the vCPSs in order to provide the platooning functionality safely and to define the messages that must be exchanged between these entities while the content of these messages and how to use them is explained. After, it must be defined the architecture of the controller and some control equations or rules to follow, section 4. Next, the integration of the messages with the controller must be defined clearly indicating the importance of the communication input for the controller to can achieve a certain maneuver or just the platooning movement of the vCPSs, section 5. Later, another main objective that is the implementation of the system in a certain device, must be done and documented well, section 6. In section 6, it is needed to study the device possibilities and limitations to define what it is possible to implement in the device. Also, to decide and study the available libraries that can be used to make the implementation easier. Once the implementation scenarios are defined, it must be described how to provide the communication and controller functionality defined for these scenarios based on the device's limitations. Finally, the performance of the designed protocol must be measure, section 7. The measurements of the performance must be done using the device implementation as no simulation is done in this thesis, and this involves some drawbacks due to the limitations of the implementation.

The tasks of the thesis have been divided in the study of previous work, the study of programming libraries and the device, the iterative design of the communication and controller protocols and its integration, the software implementation embedded in a particular device, the evaluation performance and the writing of this report. Appendix A.1 contains a Gantt chart indicating the time spent in each one of the tasks.

2. Distributed control plane for cooperative vCPSs

The main objective of the thesis is to support platooning scenarios by using wireless communication and controller functions. In order to achieve this complex model, it is proposed a distributed control plane (DCP) design based on three layers that support the cooperation among the vCPSs. This chapter defines the DCP that integrates the communication and the controller protocols required to provide platooning functionality (figure 2.1).

DCP maps the vCPS information into particular local control tasks supporting a cooperative global task, it provides point-to-point connectivity to ensure communication between vCPSs and it controls the performance of a task by the integration of the global communication messages with each local distributed controller.

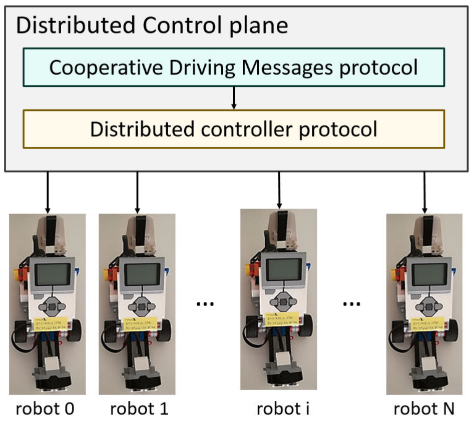

Distributed Control plane
Cooperative Driving Messages protocol
Distributed controller protocol
robot 0
robot 1
...
robot i
...
robot N

Figure 2.1: Overview of the distributed control plane diagram

The architecture of the proposed DCP, depicted in figure 2.2, integrates three levels of functionalities. On top is a cooperative driving messages (CDM) application protocol defined in section 3. CDM protocol has the responsibility of exchanging information between the vCPSs to support platooning cooperative tasks. The CDM application protocol is implemented on top of a communication protocol that is more likely a machine-to-machine (M2M) protocol. At the bottom of the architecture, there is a distributed control protocol, defined in section 4. The distributed controller is responsible for mapping the CDM messages received by the vCPSs to a particular control functionality which objective is to accomplish the cooperative task.

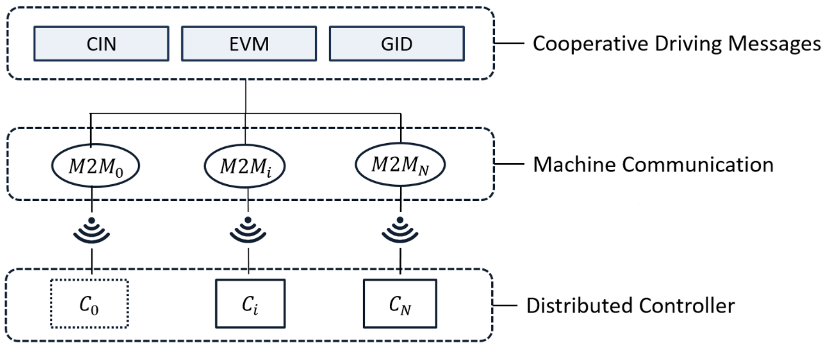


Figure 2.2: Distributed control plane diagram in details

3. Cooperative Driving Messages (CDM) protocol

This section defines a protocol based on the definition of a set of messages, and how to combine them to keep a certain distance between vCPSs maintaining string stability and to support maneuvers in a platooning scenario. First, there is a discussion on the communication links needed to exchange vehicular profile's information between vCPSs. Then, the different types of message supported by the protocol are defined. Next, states machines illustrate how to combine a sequence of messages in order to achieve different platooning scenarios. In the end, there is a section defining how to achieve discovery of a platoon.

3.1 Communication links

There are two main variations of communication links to send CAM messages between vCPSs. One is the communication proposed in [20] that it uses a *one-vCPS look-ahead* communication where every vCPS receives information just from its vCPS in front as figure 3.1 shows. And the other one, shown in figure 3.2, is a combination of *leader-follower* plus *one-vCPS look-ahead* communication where the leader vCPS communicates with every follower and each follower communicates with its following vCPS. Papers [14],[21] and [22] defines this communication.

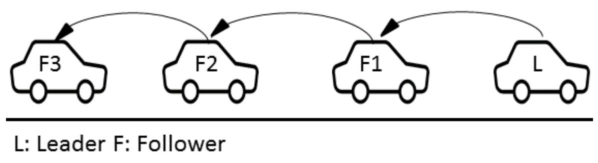


Figure 3.1: Platoon using one-vCPS look-ahead communication

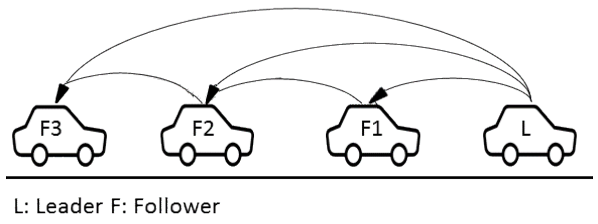


Figure 3.2: Platoon using leader-followers and one-vCPS look-ahead communication

The advantages of the communication drawn in figure 3.1 is lower bandwidth consumption as less messages are exchange between vCPSs comparing with figure 3.2. The possibility of messages collision is the same as the messages are exchanged in the same communication medium in both cases.

The messages to support maneuvers need a communication link with the leader as every follower must be able to communicate with the leader because the leader is the one who manages the maneuvers and who accepts or denies a certain maneuver. The only exception is the case where the leader does a *leave* maneuver, as it can accept that is leaving but it has to indicate that to the followers so they can elect a new leader. Therefore, it is required to have bidirectional links between the leader and the followers vCPSs.

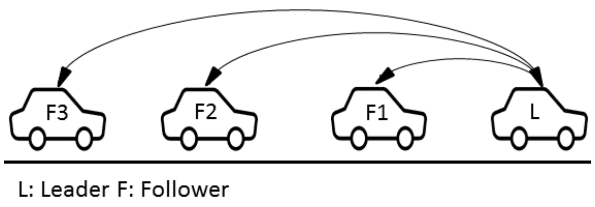


Figure 3.3: Communication links between vCPSs for messages that provide maneuvers

All these links must be reorganised in the case of the leader vCPS *leaving* the platoon and also in the case of *crash failure* of a vCPS, meaning with *crash failure* the suddenly stop of sending messages from the *crashed* vCPS, .

3.2 Type of messages

The communication protocol is based on a set of different messages that will be exchanged between the vCPSs. These messages communicate information about the speed profile of a vCPS and also about it desires of joining or leaving a platoon.

To ensure string stability is necessary to share information about the vehicular profile of a vCPS what allows the receiver's vCPSs to anticipate speed changes. This way the vCPS has a more precise control of the security distance maintained and it makes easier to avoid jam on the brakes.

The messages to maintain a certain distance between vCPSs within the platoon while string stability is provided has been defined as Context Information (CIN) messages, figure 3.4. This kind of message contains speed, location and acceleration information about the sender. This information combined with the input information from onboard sensors (eg. sensor that returns the distance between vCPSs, GPS, etc.) help to accurate the output of the receiver vCPS providing a more secure driving.

INFO	speed	location_x	location_y	acceleration

Figure 3.4: Structure of CIN messages

Apart from the messages to ensure string stability, messages to provide maneuvers in the platoon are needed. The idea is to have a *generic* type of message that allows sharing information supporting diverse movements within the platoon. These messages are defined as Event Maneuver (EVM) messages, figure 3.5. EVM messages inform other vCPSs of the desire of *JOIN*, *LEAVE* or *ACCELERATE*. This message can be used directly to join or leave a platoon but also to do more complex maneuvers as merging two platoons, split the platoon in two or change the position within the platoon as figure 1.2a shows.

Every EVM message includes the id of the sender and the receiver and the type of the petition done. The id of the sender and the receiver is an unique value for every vCPS in the platoon that represents the position of that vCPS within the platoon. It has decided to give the id value 0 to the leader, and a whole number starting from 1 to the follower. This value for the follower indicates its position in the platoon (eg. the first follower has id=1, the second follower has id=2,..., the last follower has id=n being n the total number of follower in the platoon that it is equal to the total number of vCPSs in the platoon minus one). The type of the petition indicates if the messages is requesting something (*REQ* value), or if it is just accepting (*ACK* value) or rejecting that petition (*NACK* value).

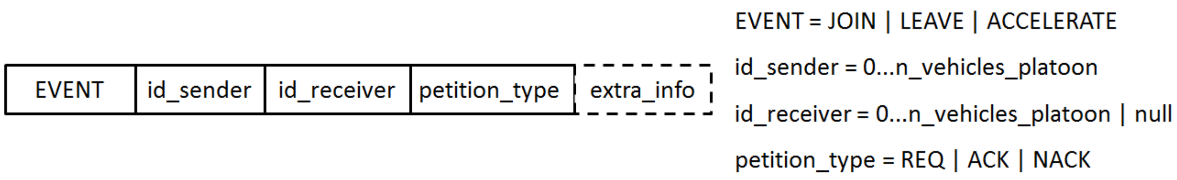
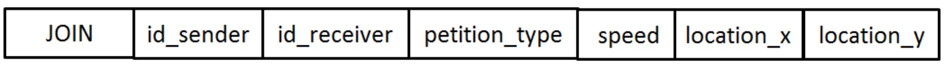
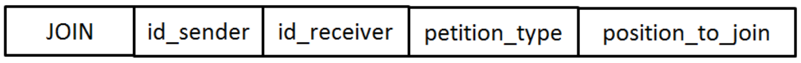

EVENT
id_sender
id_receiver
petition_type
extra_info
EVENT = JOIN | LEAVE | ACCELERATE
id_sender = 0...n_vehicles_platoon
id_receiver = 0...n_vehicles_platoon | null
petition_type = REQ | ACK | NACK

Figure 3.5: Structure of EVM message

JOIN message, defined in figure 3.6, includes the mandatory fields of an EVM message (id sender, id receiver and petition type) in addition with other fields. There are two different packet structures, one for the request message and another for the answer message. The request message, figure 3.6a, adds to the mandatory fields of an *EVENT* message the fields *speed*, *location_x* and *location_y*, previously defined the *CIN* messages of section 3.2. These fields are used to allow the leader vCPS indicate where the vCPS requesting has to join the platoon, as in [23]. The answer message just includes the *position_to_join* field that indicates the position where the vCPS has to join the platoon.



(a) Join request



(b) Join answer

Figure 3.6: Structure of EVM message of type join

LEAVE message, figure 3.7, is really simple and it just includes the mandatory fields of an EVM message. The idea is that a vCPS in the platoon sends a *LEAVE* message of the type request (*REQ*) and it waits for an acknowledgement (*ACK*) answer from the leader to can leave the platoon. After that, the leader is in charge of reformulating the group changing the ids of the other followers if it is needed.

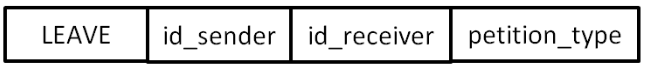


Figure 3.7: Structure of EVM message of type leave

ACCELERATE message, figure 3.8, includes an acceleration field apart from the mandatory fields of an *EVENT* message. This acceleration fields indicates how much a vCPS must accelerate or decelerate if that value is negative. The leader uses this message to can create gaps so a new vCPS can join the platoon in this gap and to remove this gap after a follower in the middle leaves the platoon. In a regular scenario, this message is not needed as the onboard sensors compute the distance with the car in front and automatically accelerate or decelerate depending on the distance. But, it can be useful in emergency braking scenarios where there is an obstacle in the road, and also in case the sensors giving the distance just work in a certain range then when vCPSs are out of range they just will know if they need to accelerate or not by communication.

ACCELERATE	id_sender	id_receiver	petition_type	acceleration

Figure 3.8: Structure of EVM message of type accelerate

Group Identifier message (GID) is used to indicate which role a vCPS has in the platoon and also its position within the platoon. At the beginning of the platoon formation, every vCPS should have an id that can change anytime a *GID* message arrives. If a process receives a *GID* message where the field *old_id* is equal to its id, then this vCPS must change its id with the value in *id_new* field and send back to the leader an acknowledge of this message. A *GID* message, figure 3.9, is composed with a *petition_type* field, a field with the old id of the receiver and a field with the new id of the receiver. The petition type can be *REQ* for request messages or *ACK* for acknowledge message.

ID	petition_type	old_id	new_id

Figure 3.9: Structure of GID message

Election of a new leader is done by using *LEADER* messages, this messages, defined in figure 3.10, are included in the GID kind of messages as its functionality is also related to the definition of the platoon group. This type of message can be request message using *REQ* in the *petition_type* field, to indicate the other vCPSs that they have to change its leader. And, It can also acknowledge messages to accept the new leader. An acknowledge message has the value *ACK* in the *petition_type* field. There is also a field *address_leader* to indicate the IP address of the new leader, this way the other vCPSs know the new leader with whom they have to communicate.

LEADER message:

LEADER	petition_type	address_leader

Figure 3.10: Structure of GID message of type leader

3.3 Protocol application

In this section, some state machines are defined to show how the protocol is applied in concrete scenarios (eg. join a platoon). These state machines are represented as Mealy state diagrams[24] where the output is defined based on the actual state and the input as figure 3.11 shows.

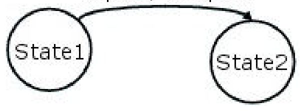
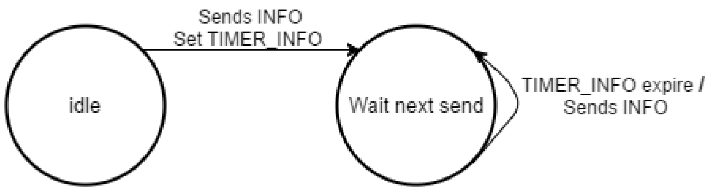
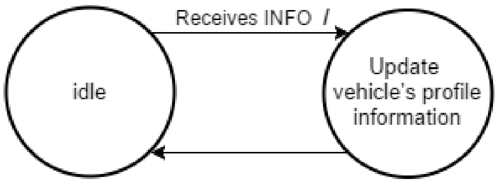


Figure 3.11: Transition between states using Mealy machine

In a platooning scenario, messages where the vCPSs exchange velocity, location and acceleration information are needed to maintain string stability through the platoon. This message is defined in section 3.2 as *CIN* messages. Figure 3.12 shows the states a vCPS follows to send and receive *CIN* messages. These messages are sent from a vCPS to its follower. In concrete, a vCPS sends a message periodically to its follower, this regularity is represented using a timer variable, *TIMER_INFO*, to show that every time the timer expires a new message is sent, figure 3.12a. At the same time, the vCPS can receive these messages as figure 3.12b shows. Every time a vCPS receives a message, it updates its controller information to ensure string stability by keeping the correct distance and following the given velocity.



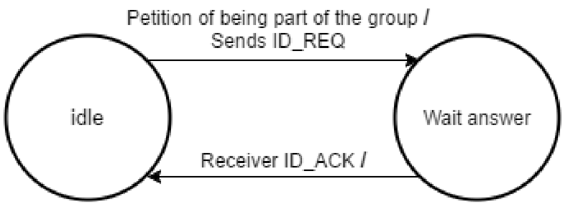
(a) Sender



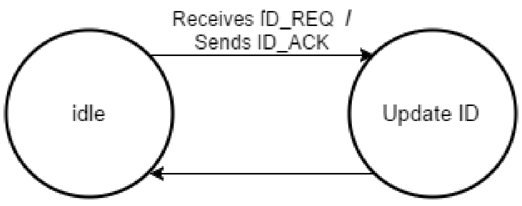
(b) Receiver

Figure 3.12: State machine for the exchange of CIN messages between vCPS

The vCPSs within a platoon form part of a group and this group must be defined and known by the leader. Group messages as the one defined in the previous section as *GID* messages are needed to indicate the id of a vCPS when it joins the platoon for the first time or to change the ID of a vCPS that is already in the platoon (eg. there are three vCPSs in a platoon with ids 0, 1 and 2 respectively and a new vCPS joins the platoon between vCPSs 1 and 2; the vCPS joining needs to have id 2 and the actual id 2 vCPS needs the new id 3). Figure 3.13 shows a state machine where the platoon's leader indicates a new id to a follower sending an *ID_REQ* message, and a state machine for the follower where it updates its id after receiving an *ID_REQ* message. The follower accepts the request, after changing its id, by sending an *ID_ACK* back to the follower.



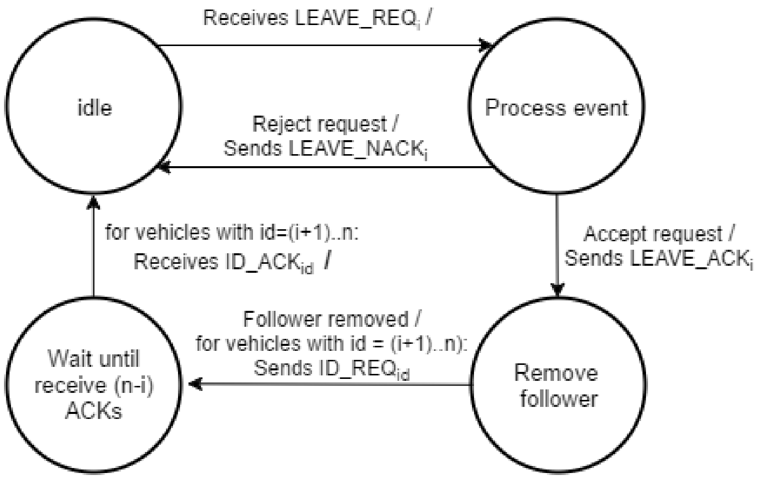
(a) Leader



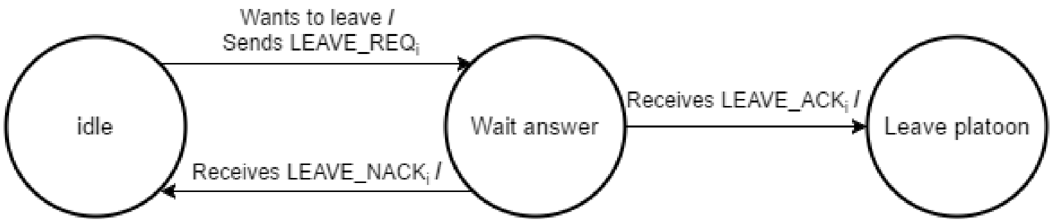
(b) Follower

Figure 3.13: State machine to define a group identifier by using GID messages

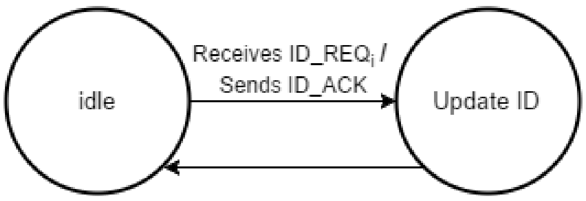
Figure 3.14 shows a more complex scenario where a follower vCPS within a platoon wants to leave the platoon. In this scenario, the follower that wants to leave needs to start sending a *LEAVE_REQ* message to the leader. Then, the leader vCPS receives the leave petition message that can accept sending a *LEAVE_ACK* or reject sending a *LEAVE_NACK* message. The follower that wants to leave will leave if it receives a *LEAVE_ACK* message, but if it gets a *LEAVE_NACK* the follower will have to send a new request to can leave. After sending a *LEAVE_ACK*, the leader must redefine the group by sending *ID_REQ* messages to all the follower vCPSs with an id bigger than the vCPS that have left the platoon.


Receives LEAVE_REQ$_i$ /
idle
Process event
Reject request / Sends LEAVE_NACK$_i$
for vehicles with id=(i+1)..n: Receives ID_ACK$_{id}$ /
Accept request / Sends LEAVE_ACK$_i$
Follower removed / for vehicles with id = (i+1)..n): Sends ID_REQ$_{id}$
Wait until receive (n-i) ACKs
Remove follower

(a) Leader


Wants to leave / Sends LEAVE_REQ$_i$
idle
Wait answer
Receives LEAVE_ACK$_i$ /
Leave platoon
Receives LEAVE_NACK$_i$ /

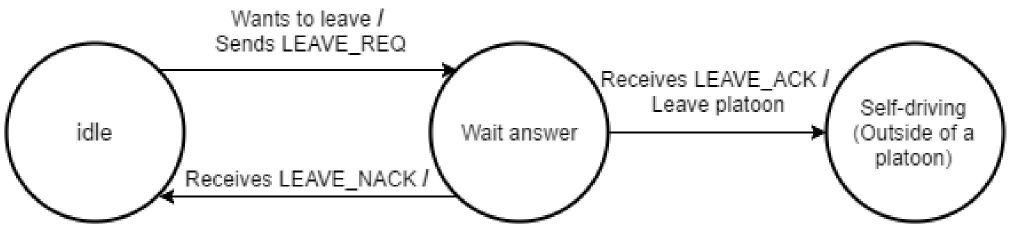
(b) Follower that wants to leave (id = i)



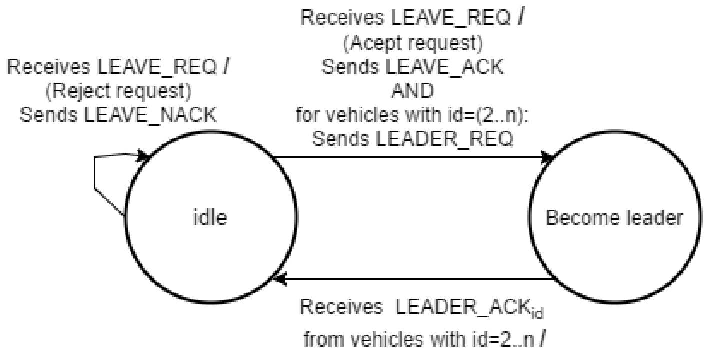
(c) Followers in the platoon with id from (i+1) to n

Figure 3.14: State machine for a follower leaving the platoon

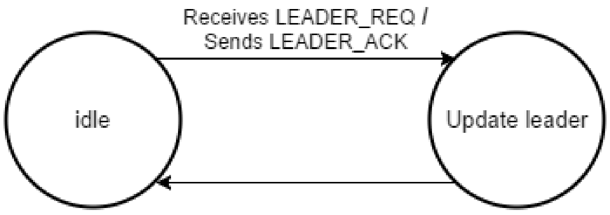
A specific scenario of a platoon's vCPS leaving, it is one where the leader of the platoon wants to leave. In this scenario, the communication will be different as the leader needs to communicate to the followers its leaving request and it is needed a new leader to be elected. Figure 3.15 shows the state machines to can achieve the scenario where the leader leaves the platoon for the different roles of vCPSs. At the beginning, the leader sends a *LEAVE_REQ* message to the first follower (id = 1) and waits for an acknowledge message. The first follower receives the *LEAVE_REQ* message and accepts or rejects the request with a *LEAVE_ACK* or a *LEAVE_NACK* message, respectively. If the leader receives a *LEAVE_ACK* message it can just leave the platoon, as the first follower has already information about the platoon (eg. platoon size, the actual speed of the platoon, etc). After sending a *LEAVE_ACK* messages, the first follower has to impose itself to be leader sending a *LEADER* message to every follower in the platoon. The followers in the platoon receive a *LEADER* message and accept the new leader replying with a *LEADER_ACK* message, and update their leader and their communication channels as now there is a new leader and one vCPS less in the platoon.



(a) Leader



(b) Follower (id = 1)

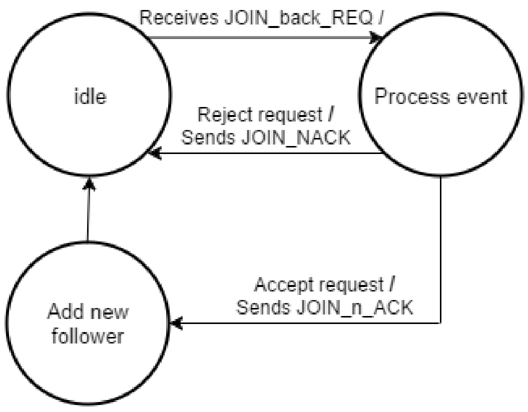


(c) Followers in the platoon with id > 1

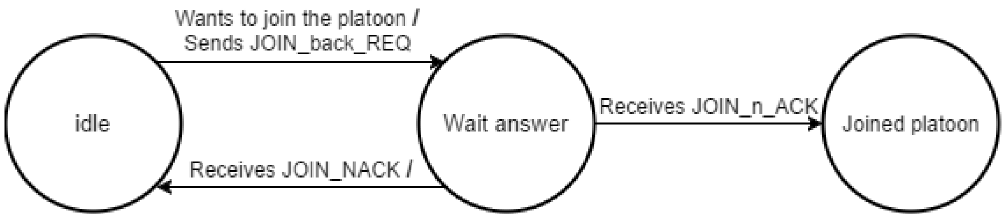
Figure 3.15: State machine for a leader leaving the platoon

Another common maneuver in a platooning scenario is the action of a new vCPS joining a platoon. The new vCPS can join the platoon in the back or in any other place.

Figure 3.16 defines the state machines that indicate the communication messages to support a new vCPS joining the platoon at the back scenario. First, the new vCPS needs to request the platoon leader that it want to join the platoon at the end by sending a *JOIN_back_REQ* message. Using the message defined in the previous section, it can be assumed that this *JOIN_back_REQ* is a *JOIN* message where *position_to_join* field has a special value that indicates the last position of the platoon. This value will not be the real value of the last position of the platoon, but a pre-defined value as the vCPS that wants to join does not know the size of the platoon. Then, when the leader receives the request message it will reject the petition by sending *JOIN_NACK* or accept by sending *JOIN_n_ACK* this acknowledge message contains the id assigned to the vCPS joining the platoon. The vCPS that wants to join receives a *JOIN_n_ACK* message and it updates its id with the value contained in the message to form part of the platoon group, in this case, *n* that represents the last position of the platoon.


Receives JOIN_back_REQ /
idle
Process event
Reject request / Sends JOIN_NACK
Accept request / Sends JOIN_n_ACK
Add new follower

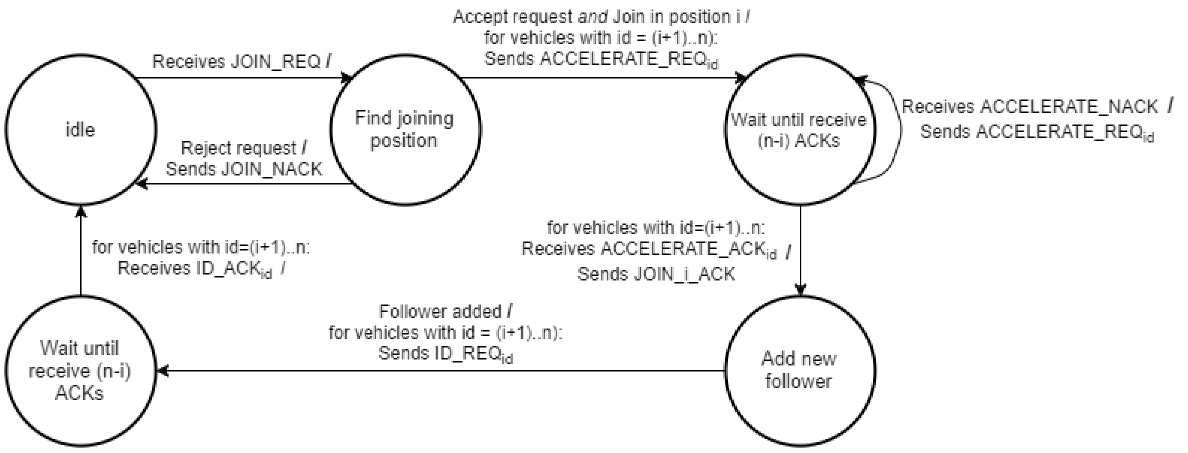
(a) Leader



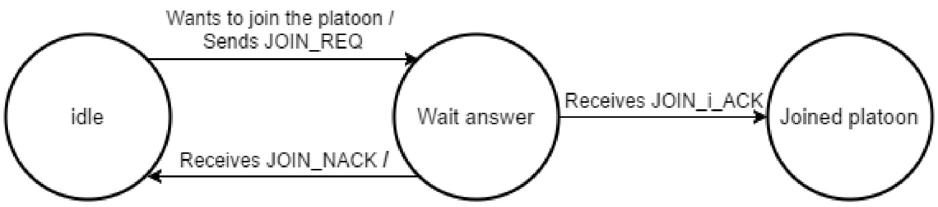
(b) New follower (vCPS that wants to join back)

Figure 3.16: State machine for a new vCPS joining the platoon at the back

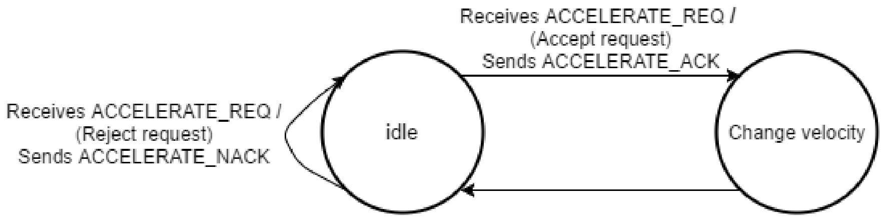
Figure 3.17 shows the general joining scenario where a new vCPS can join the platoon anywhere. In this scenario, anywhere does not mean that the joining vCPS chooses where to join; it means that the vCPS joining the platoon requests the entry to the platoon group and depending on its position the leader selects the optimal position to join. To initiate the scenario, the new vCPS that wants to join the platoon sends a *JOIN_REQ* to the leader. Then, the leader receives that request that can reject sending back a *JOIN_NACK* message, or can accept. If the leader accepts the petition then it calculates where the vCPS must join the platoon and send to a follower *i* that will be behind this new vCPS an *ACCELERATE_REQ* to deccelerate and make a gap where the new vCPS will join. The vCPS receiving this *ACCELERATE_REQ* message can reject the petition answering with a *ACCELERATE_NACK* or accept the petition with a *ACCELERATE_ACK* message and changing its velocity to make the gap. Once the leader has received the confirmation of the gap with the *ACCELERATE_ACK* message from the follower *i*, the leader can reply the vCPS that want to join the platoon with a *JOIN_ACK* message indicating the location where it must join and its id to be part of the platoon. The leader also sends new ids, *ID_REQ*, to the followers behind the new vCPS to redefine the group. These followers receive the *ID_REQ* and update its id with the value of the new one indicated in the request message.


idle
Receives JOIN_REQ /
Find joining position
Reject request / Sends JOIN_NACK
Accept request *and* Join in position i / for vehicles with id = (i+1)..n): Sends ACCELERATE_REQ$_{id}$
Wait until receive (n-i) ACKs
Receives ACCELERATE_NACK / Sends ACCELERATE_REQ$_{id}$
for vehicles with id=(i+1)..n: Receives ACCELERATE_ACK$_{id}$ / Sends JOIN_i_ACK
Add new follower
Follower added / for vehicles with id = (i+1)..n): Sends ID_REQ$_{id}$
Wait until receive (n-i) ACKs
for vehicles with id=(i+1)..n: Receives ID_ACK$_{id}$ /

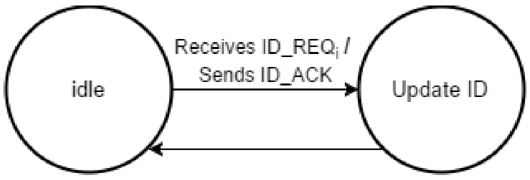
(a) Leader



(b) New follower (vCPS that wants to join the platoon)


Receives ACCELERATE_REQ / (Reject request) Sends ACCELERATE_NACK
idle
Receives ACCELERATE_REQ / (Accept request) Sends ACCELERATE_ACK
Change velocity

(c) Follower in the platoon (id = i)



(d) Followers in the platoon with id from i to n

Figure 3.17: State machine for a new vCPS joining the platoon in any position

To conclude this section, two figures are included to show, in a more graphical way than using Mealy state machines, how the vCPSs perform the scenarios in figure 1.2a and 1.2b, respectively, using a sequence of messages defined in the communication protocol.

The figure depicted in 3.18 shows the scenario defined in figure 1.2a where a vCPS that is already in the platoon, in a middle position, leaves and joins again the platoon at the end.

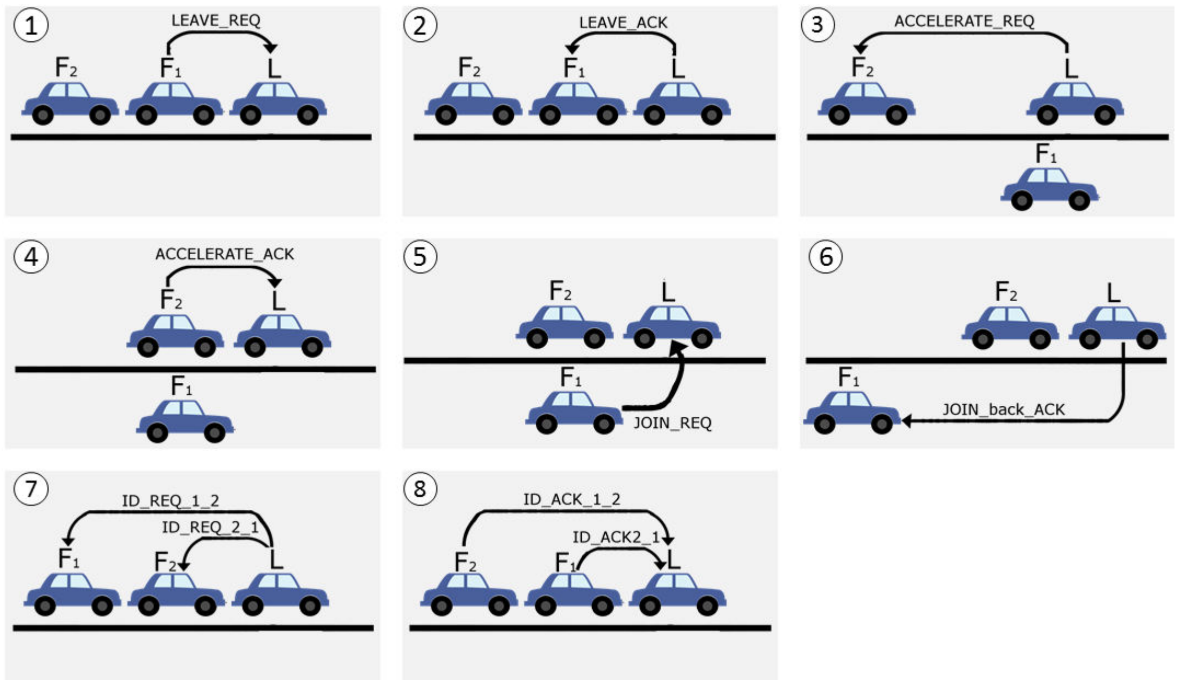


Figure 3.18: Sequence of pictures showing a vCPS leaving the platoon and then joining it aging at the back scenario

The scenario defined in figure 1.2b where a vCPS that is not part of a platoon joins a close platoon is represented in figure 3.19.

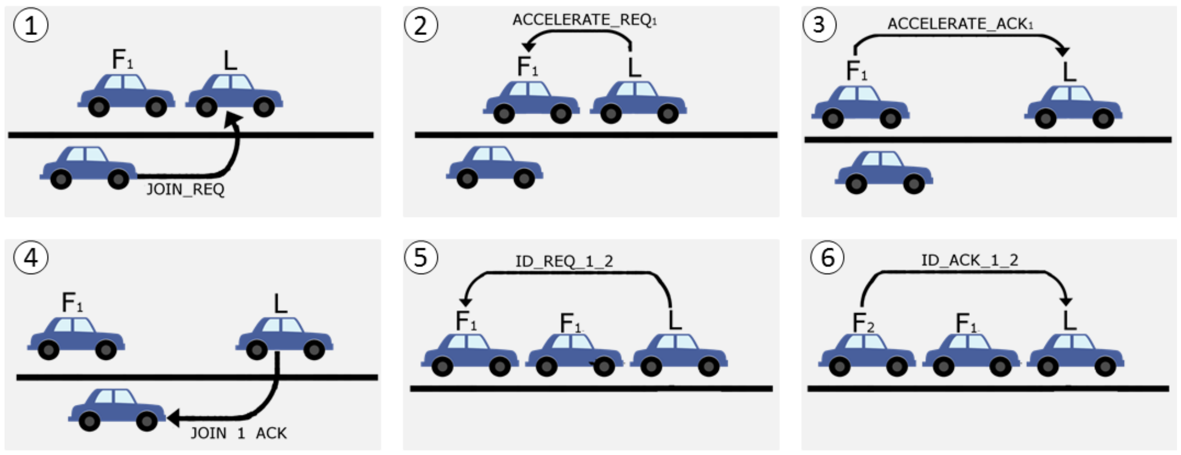


Figure 3.19: Sequence of pictures showing a vCPS joining a platoon in the middle scenario

3.4 Leader discovery

In section 3.3 every scenario assumes that the vCPSs know the members of the platoon to can communicate with them. But in a real situation where a vCPS in the road wants to join a platoon, it will not know its members as platoons are something dynamic that can be created, modified and dissolved over the time.

For that reason, a discovery mechanism should be provided to allow new vCPSs to discover the platoon, to recognise the leader and to communicate with it in order to join the platoon.

A solution for discovery is to use broadcast messages in the network to find the leader. This way, an independent vCPS that wants to join a platoon broadcasts a message in the network and waits an answer from the leader or any member of the platoon. This answer should include the address of the leader to indicate the vCPS outside the platoon to whom it must request the access to the platoon. As a broadcast message is sent to every node in the network, it is possible that the vCPS outside the platoon receives more than one answer, if that is the case the vCPS just discards the duplicates.

Figure 3.20 shows a possible definition of the request and acknowledge discovery message. In the request message the vCPS that is on the road but does not know the platoon, broadcast a message like the one in figure 3.20a to indicate that it wants to be part of the platoon. Be part means that it wants to know how to communicate with the platoon not to join. Indeed, the vCPS needs to send a *EVM* message of type *JOIN* to can join the platoon, this *DISCOVERY* message is just used to know the address of the leader. The leader or any other vCPS that receives this message answers back with a *DISCOVERY_ACK* message, like the one defined in figure 3.20b, indicating the address of the platoon leader.

DISCOVERY	REQ	address_request

(a) Discovery request

DISCOVERY	ACK	address_request	address_leader

(b) Discovery answer

Figure 3.20: Structure of GID message of type discovery

Broadcast messages are sent within a network, so it must be assumed that every vCPS is on the same network. This is a reasonable assumption if it is assumed that the vCPSs connects to a network provided by some infrastructure in the road. This way vCPSs close to the same infrastructure will be in the same network. But even if this is assumed, this road infrastructure that provides the network connectivity will not be the same during the whole platoon's trip. Consequently, vCPSs needs to use a mobile IP protocol that allows vCPSs to keep its same IP even if they move from one network to another or to have a mechanism that triggers broadcast messages informing of the new IP when there is a change on the IP. So, If just half of the vCPSs within a platoon move to a new network, they can still communicate with the vCPSs in the previous network. Figure 3.21 shows how the discovery scenario could be achieved using the messages defined in figure 3.20 considering the network assumptions and the mobile IP protocol described previously.

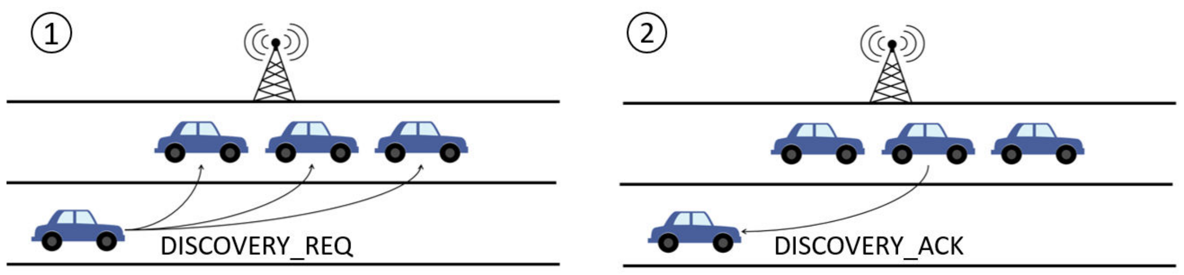


Figure 3.21: Sequence of pictures showing discovery of a platoon

4. Distributed Controller protocol

This chapter defines the controller functionalities each vCPS needs to execute in order to provide string stability and maneuvers. First, some figures depict block diagrams to describe the controller architecture and system model. Then, there is an analysis of the controller functionality needed to provide string stability through a platoon of vCPSs.

4.1 Controller model

First, the architecture of the controller is defined in figure 4.1 showing how onboard sensors and a communication unit create an output that the controller uses as input to generate an output that is the input of the actuators. To support platooning in real vehicles sophisticated sensors like radar or lidar are used to measure the distance between vCPSs instead of infrared sensors that are used in the implementation done in the thesis. VCPSs are also equipped with a sensor to get a location reference, for example, a global positioning system (GPS) to provide the position and do a more accurate control of the platoon movement. In addition, there is a wireless communication unit compliant that can use the IEEE 802.11p standard or another vehicular communication technology like 5G in real vehicles platooning, or the IEEE 802.15 standard used in mobile robots' implementations as it consumes less power. The actuators control the acceleration of the vCPS setting the compute acceleration from the controller to the motor.

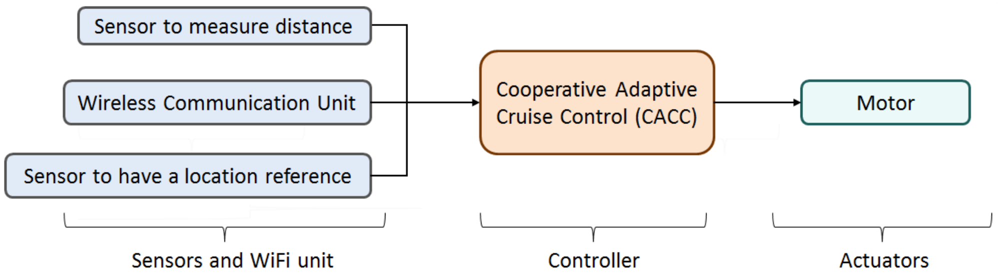

Sensor to measure distance
Wireless Communication Unit
Sensor to have a location reference
Cooperative Adaptive Cruise Control (CACC)
Motor
Sensors and WiFi unit
Controller
Actuators

Figure 4.1: Control architecture block diagram

The control architecture in figure 4.2 shows the outputs and inputs from every functional block (sensors, controller and actuators) specified in figure 4.1, where a block diagram shows the components that each vCPS needs to provide platooning functionality based on keeping a distance between vCPSs while string stability is preserved. The input and output from the wireless network show the content of the *CIN* messages, defined in section 3.2, to keep a distance between vCPSs while string stability is maintained. The maneuvers input achieved by *EVM* and *GID* messages are omitted in this figure to express the diagram with clarity but express in figure 4.3. The solid lines indicate that the input/output is sent/received periodically.

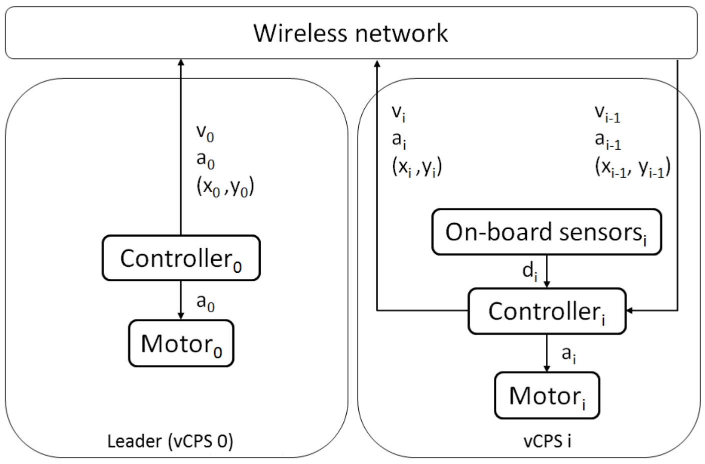


Figure 4.2: Control system block diagram

It is assumed that every vCPS is identified by a unique value of i being i = [0..N] where N is the number of the follower vCPS in the platoon and the vCPS with id=0 is always the leader. Each vCPS receives a communication input from the vCPS it has in front, except the leader as it is supposed that it has no vCPS in front, and an input from its onboard sensors. From the communication each vCPS receives v_{i-1}, a_{i-1} and (x_{i-1},y_{i-1}) that correspond to the velocity, acceleration and position of the sender vCPS, respectively. From the sensors, the vCPS receives the distance to the vCPS in front. It is assumed that the leader has no vCPS in front within the platoon, for that reason its *on-board sensors* are not depicted as the leader does not need to know the distance d_i to the vCPS in front to keep the controller objective. Despite the fact that leader does not need onboard sensors to provide string stability and a certain distance within the platoon, the leader can include sensors to measure the distance like the followers. The leader uses these sensors in case there is an obstacle in the road to detect it, or if another leader is elected and it becomes follower. This distance from the sensors, d_i, is used by the controller to accelerate or decelerate in the case of being too close or far from the vCPS in front. The controller computes the acceleration, a_i to be applied to the motor taking into account the input from the communication and the onboard sensors. In the case of the leader, it just decides the velocity it wants for the platoon and indicates to the gas injector and pressure regulator the acceleration required to run the motor at that certain speed.

The objective of the controller is still the same during maneuvers, vCPSs must keep a certain distance between vCPSs and string stability even if there are changes in the platoon due to maneuvers. To provide maneuvers, the vCPSs receive additional information related to the maneuver request and acknowledgements. This information is provided by the communication protocol to the controller as the distributed control plane defines, chapter 2. The distributed control plane integrates the messages sent based on the communication protocol with the control protocol to provide platooning functionality. Figure 4.3 shows how the messages used to provide maneuvers, *EVM* messages mainly, come from the wireless network to be computed in the controller and goes to the wireless network as a request or an answer compute by the controller. The dotted lines indicate that this input/output is asynchronous what means that the messages could be sent/receive at any time and to any vCPS that matches the id specified in the message.

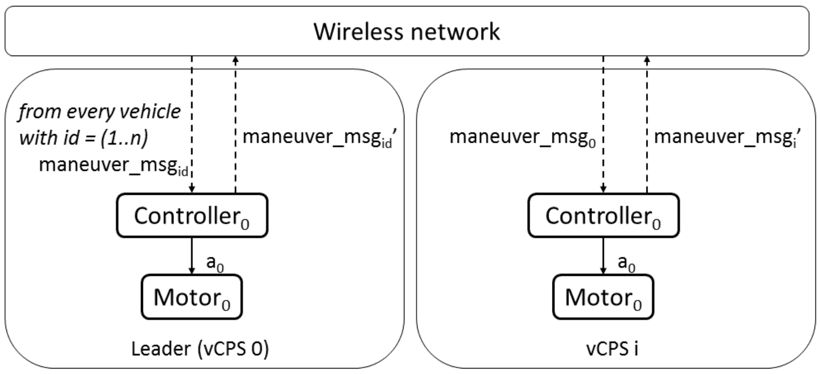


Figure 4.3: Maneuver's control system block diagram

4.2 Controller analysis

The main function of the controller is to keep a certain distance between vCPSs, avoiding collisions, while string stability is kept in a network with communication delays.

Let D be the desired distance to keep, knowing d$_i$ from the infrared sensors. The controller goal is to maintain the distance D, what means that

$$D - d_i = 0 \tag{1}$$

based on (1) the controller computes a velocity value for the actuators with the purpose of reducing the gap error between vCPS i and the vCPS in front, i-1. The action of accelerating or decelerating to keep the desired distance is shown in

$$\begin{aligned} if\ d_i > D\ then\ v_i = v_i * A \\ else\ if\ d_i < D\ then\ v_i = v_i * D \\ else\ v_i = v_{i-1}; \end{aligned} \tag{2}$$

where A is a predefined value that indicates the acceleration percentage and D is another predefined value that indicates the deceleration percentage.

String stability is one of the main factors to provide by the control system. By definition, string stability is the faculty to reduce perturbations, velocity and acceleration, throughout the platoon. According to [16] string stability can be defined as

$$|SS(t)| = \frac{X_i(t)}{X_{i-1}(t)} \leq 1, \quad i \geq 1 \tag{3}$$

with X$_i$(t) and X$_{i-1}$(t) being the positions of the vCPS i and i-1 in a certain time t, respectively. This equation shows the need of having communication, to know the previous vCPS position and with it can achieve string stability.

The position X$_i$(t) of a vCPS can be determined by using a positioning system like GPS but also using odometry equations to estimate relative position based on a starting position. But, not every device integrates a positioning system so, for that devices, the position can be estimated by odometry, if the trajectory angle is close to 0, then

$$\begin{aligned} \Delta x_i = \Delta s_i \cos(\theta_i + \frac{\Delta\theta_i}{2}) \\ \Delta y_i = \Delta s_i \sin(\theta_i + \frac{\Delta\theta_i}{2}) \end{aligned} \tag{4}$$

where Δx_i, Δy_i are the difference of the robot's coordinates in a Cartesian plane all over the time, and Δs_i is the distance travelled by robot i. Assuming that the robots move following a straight line the trajectory angle $\theta = 0$ and the Δy_i is a constant based on the initial y-coordinate y_{i_0}. Based on this assumption, (4) can be simplified to

$$\begin{aligned} \Delta x_i &= \Delta s_i \\ \Delta y_i &= 0 \end{aligned} \tag{5}$$

Δs_i can also be defined as $si(t)$, like the distance travelled after a time t, and it can be easily calculated by multiplying the speed by the time like

$$s_i(t) = v_i(t)t = \Delta x_i \tag{6}$$

Using (5) and (6) the position of a robot Xi based on its initial position x_{i_0} and its traveled distance s_i is calculate with

$$X_i = x_{i_0} + \Delta x_i(t) \tag{7}$$

Equations from (3) to (7) show how the controller model can keep the string stability through the platoon based on its knowledge received from the on-board sensors and the communication unit.

It is assumed that the communication network used to send the vehicle profile information cannot deliver the messages instantly, so the messages can be delayed. For that reason, the output computed by the controller must have into account this variable delay.

The communication delay must be taken into account in a way that delay information does not compromise safety while keeping a certain distance between vCPSs and the platoon stable. In addition, maneuvers can be aborted if the delay in the communication exceeds a certain limit.

VCPSs can realise of a certain delay by computing the difference between its position and the position sent by the previous vCPS in a certain sample

$$|X_{i-1}| = |X_i + d_i| \tag{8}$$

This way, the vCPS knows that there is a certain time delay if (8) is not satisfied. In concrete, the communication delay for that sample is

$$\tau = \frac{v_i}{|X_{i-1} - X_i + d_i|} \tag{9}$$

(9) can be used to abort a maneuver in case this time delay is over a threshold and also for controller arrangements to keep string stability.

In a leave maneuver execution where a follower leaves the platoon, it is assumed that messages are exchanged between the vCPS that wants to leave and the leader. After, the vCPS can leave, without new controller requirements, the vCPS in the platoon just needs to still keep a certain distance and string stability. Then, when the vCPS has left the platoon completely the vCPSs behind it must cover the remaining gap. To achieve this the controller just act as normal, it receives an input from the sensors saying the distance between this vCPS and the preceding one. The vCPS that receives this distance computes that this distance is bigger compared to the desired distance so it accelerates. In this scenario, the leader could also send *ACCELERATE* messages to indicate that they have to accelerate to cover the gap, but anyway, the controller is already aware of that.

In a join maneuver execution, the vCPS that wants to join sent that request to the leader. Then, the leader computes that request to decide where this vCPS should join the platoon as

$$x_{join} = x_m + \frac{v_0 - v_m}{t + \tau} \tag{10}$$

where x_{join} is the position where the vCPS must join, x_m and v_m is the position and velocity of the vCPS when it requests to join, respectively, and τ is the communication delay. After computing this position, the leader sends a message to the vCPSs behind this position to decelerate and create a gap. This gap is covered by the vCPS joining the platoon, and if this gap is too big then the controller will get a distance bigger than the desired distance that it will accelerate the vCPS and cover the gap completely to keep the desired distance.

Due to communication delays, the execution of a maneuver could compromise safety if the actions taken to execute the maneuver are not achieve on time. For that reason, it is important to measure the communication delay and conclude if a maneuver should execute it or if it has to abort it. A safe time ratio that indicates if the maneuver is still executable can be defined as in [22]. Assuming that the maximum tolerable delay, τ_{safe}, to execute a maneuver can be defined as in [22]

$$\tau_{safe} = \{\tau_{msg} : T \in \tau_{safe} \wedge \tau_{msg} \leq \delta_{req} + \Delta\} \tag{11}$$

where τ_{msg} is a certain delay of a message, T is the inter-message delays collected, δ_{req} is the maximum allowable inter-message delay and Δ is a grace time period in which the information received is still useful.

Then, the safe time ratio, τ_{safe_ratio}, is

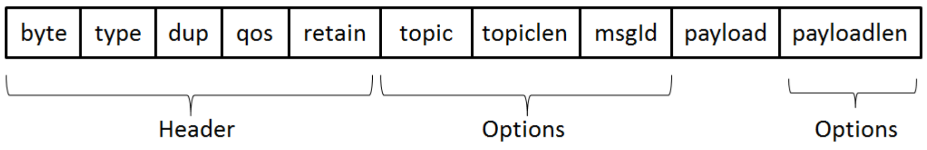
$$\tau_{safe\_ratio} = \frac{\sum_{\tau_s \in \tau_{safe}} \tau_s}{\sum_{\tau_{msg} \in T} \tau_{msg}} \tag{12}$$

5. Integration of CDM protocol into the distributed controller

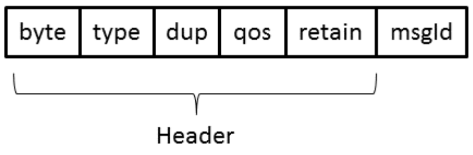
This chapter describes the integration of the communication protocol, defined in chapter 3, and the distributed controller protocol from chapter 4 to build the distributed control plane described in chapter 2. So basically, the chapter defines the interaction between the communication and the distributed controller protocol to provide platooning.

In concrete, it is defined how the CAM protocol is incorporated into the M2M protocol, and how the M2M protocol works over the wireless network and gives input to the distributed controllers to achieve the cooperative task.

The CDM protocol is implemented using the machine-to-machine (M2M) application protocol called MQTT, as section 1.5 defines. The structure of the packets and its maximum size is defined in [26]. There are many types of packets in MQTT, but publish and ACK packets are the only ones analysed (figure 5.1) as these packets are the ones more important to define the performance of the communication. Publish packet carries, inside the *payload* field, the messages defined in the CDM application communication protocol from chapter 3 (ie. CIN, EVM and GID messages). ACK packet is used by the MQTT mechanism to send back an acknowledgement of the publish packet reception right after the sending of a publish packet. It is important to remark that ACK packet is not always sent, they are sent depending on the quality of service (QoS) configured in MQTT.



(a) Publish packet



(b) ACK packet

Figure 5.1: MQTT packet structure

MQTT has three different levels of QoS: QoS0, QoS1 and QoS2[11]. QoS0 is like having no QoS at all, it ensures that the packet is delivered *at most once*, what means that it can be delivered once or not be delivered if the packet is lost in the network. QoS1, *at least once*, ensures that the packet is delivered but it can be delivered more than one time, so the receiver gets duplicates. This is possible because the sender sends the packet, activate a timer and then waits for an ACK packet to be sure that the packet was delivered. If the time-out is trigger before receiving the ACK that means that the packet was not delivered and the sender has to retransmit again the packet until it gets an ACK packet before time-out. QoS2 ensures not only that the package is delivered correctly, but also that the receiver does not get duplicates of the message. This type of quality is also called *exactly once* and it is the safest, but the slowest mode of transfer.

MQTT runs over TCP protocol that ensures reception of packets by sending TCP-ACK packets, but this ACK just ensures that the reception of the packet not its delivery to the application as the QoS1 and QoS2 options of MQTT provide.

Figure 5.2 shows the messages defined in the CDM protocol that can be included in the payload field of a MQTT publish packet. Just one of these type of messages is included in this field per packet, meaning that it is not possible to include more than one different message in this field. The inclusion of the selected message from the CDM protocol is implemented in the experimental testbed by building a JSON format string based on the CDM protocol and including this string in the payload field of the MQTT publish packet.

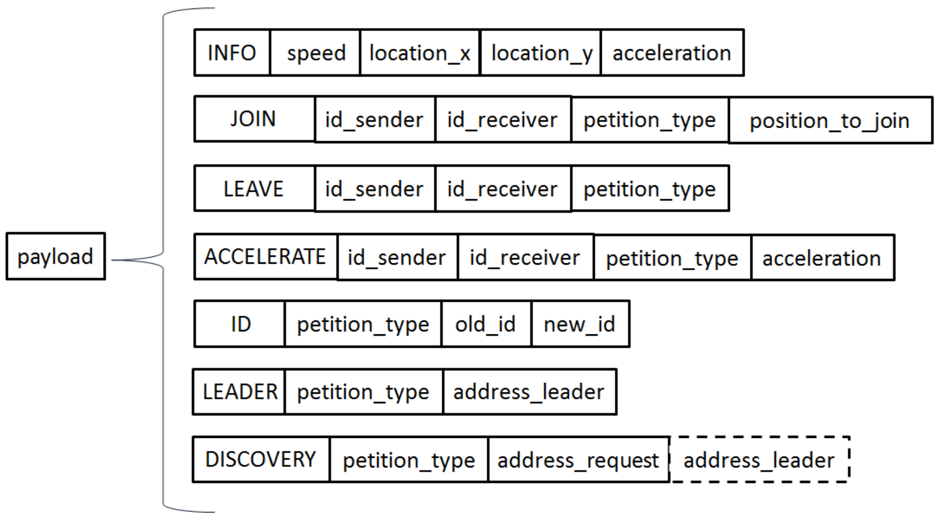

payload
INFO | speed | location_x | location_y | acceleration
JOIN | id_sender | id_receiver | petition_type | position_to_join
LEAVE | id_sender | id_receiver | petition_type
ACCELERATE | id_sender | id_receiver | petition_type | acceleration
ID | petition_type | old_id | new_id
LEADER | petition_type | address_leader
DISCOVERY | petition_type | address_request | address_leader

Figure 5.2: Possible CDMs to include in the payload of a MQTT publish packet

MQTT packets are encapsulated inside TCP and IP packets as figure 5.3 shows, where the *payload* field indicates the CDM message included. These packets are sent through the network and received by the distributed control plane application defined in 2, that each device is running as it can be seen in figure 2.2. The encapsulation of a CDM message into a MQTT packet shows the integration of the two top layers (ie. Cooperative driving messages and Machine communication) defined in figure 2.2. This packet is sent through the network and received by the controller as it is defined in figure 4.2 and 4.3. When a packet is received, it triggers a function that extracts the *payload* of the packet and it parses the content to get the packet fields defined as part of the CDM protocol, in section 3.2. Once the fields of the message are obtained the distributed controller protocol of chapter 4 uses this input to compute the output to be sent to the actuators based on the controller model defined in section 4.1.

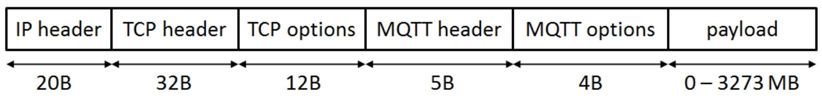

IP header | TCP header | TCP options | MQTT header | MQTT options | payload
20B | 32B | 12B | 5B | 4B | 0 – 3273 MB

Figure 5.3: Packet exchanged through the distributed control plane

The distributed control plane of each vCPS sends and receives packets, using the format shown in 5.3, in order to compute a certain controller action. Next, some temporal diagrams show how packets are exchanged through the network and how these packets affect the controller of each vCPS requiring a computation in order to provide platooning as it is defined by the coordination control plane in chapter 2.

Figure 5.4 shows a temporal diagram that indicates the transmission and reception of *CIN* messages and the computation in a 500 ms frame taking into account that *CIN* messages are sent every 100 ms using one-vCPS look-ahead communication. In this case, the controller computation refers to the motors' speed change based on the speed indicated in the message received and the input from the sensors.

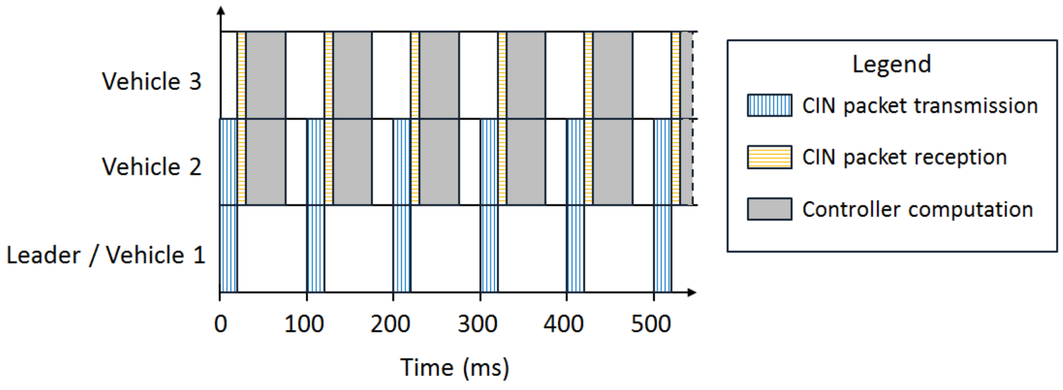


Figure 5.4: Packet data exchange between platoon vCPS

An execution of the maneuvers *join* and *leave* is depicted in figure 5.5 and 5.6, respectively, showing the exchange of *CIN* messages to maintain a safety distance between vCPSs while string stability is provided, and *EVM* messages to perform the maneuver required. At the same time, it is depicted the controller computation required to maintain the platoon driving while the maneuver is executed safely. In this case, the controller computation represented in the figures refers mainly to the change of the speed in the motors that will be its stop or set to zero in the case of the *leave* maneuver in figure 5.6 for *vehicle 3*. But, in figure 5.5 the controller computation depicted for *vehicle 3* after the LEAVE packet reception's event represent the computation required to become a member of the platoon by subscribing to the MQTT topic in order to receive *CDM* and to register the platoon id.

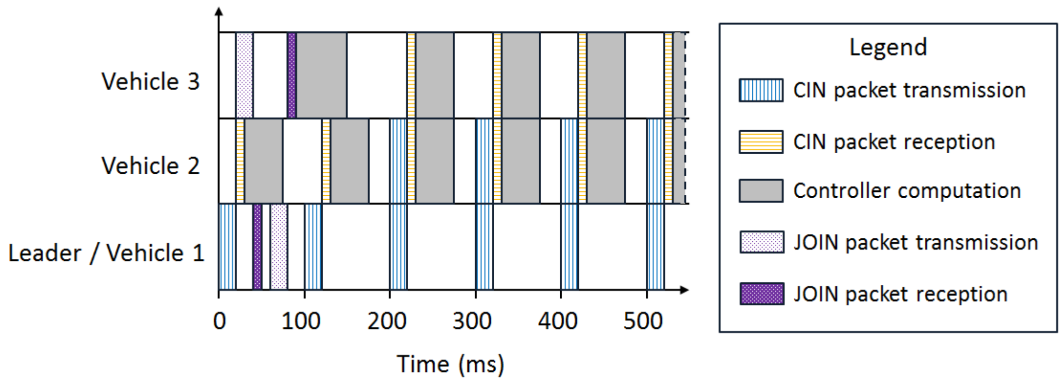


Figure 5.5: Packet data exchange between platoon vCPS to perform a join follower maneuver

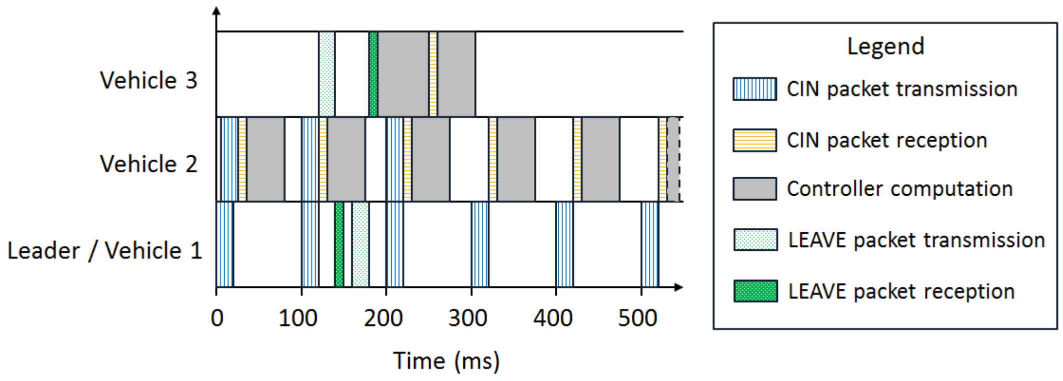


Figure 5.6: Packet data exchange between platoon vCPS to perform a leave follower maneuver

The *CDM protocol* defined in chapter 3 to support the cooperation between vCPSs using the distributed control plane allows more maneuver scenarios (eg. leader leave) but all of them follows the same idea of executing a controller functionality and provide that output to the actuators based on the received messages. For that reason, only the more common scenarios of a follower joining or leaving the platoon are depicted as this figures look enough to describe how the communication is integrated into the distributed controller.

6. Experimental testing

This chapter, first, describes the devices used in the experimental testbed. Then, it is defined the communication protocol stack to use based on the communication protocols reviewed and the protocol selected in section 1.5. After there is a subsection that summarises the failures that will affect the communication and it is described how to implement the previous communication protocol to be reliable against these failures. Later, section 6.4 indicates how the distributed controller protocol, defined in chapter 4, is applied to the devices used in the experimental testing. Next, there is a description of the device's libraries and network configuration needed for the implementation. Finally, the last section defines the scenarios implemented in the Lego Mindstorms EV3 robots and describes the decisions taken during the implementation.

6.1 Definition of the device

The distributed coordination plane defined in section 2, and designed in sections 3, 4 and 5 is implemented and tested in a particular device to evaluate its execution. The device selected to do the implementation is the Lego Mindstorms EV3 robot model 31313. This device is based on a programmable brick running in a microprocessor ARM9-based Sitara AM1808 system-on-chip from Texas Instrument. This brick can be programmed using a Lego programming software but this software can be replaced and be programmed in almost any other programming language.

As every Lego product, this device can be customised with many different Lego pieces. Between these Lego parts, there are wheels and motors that can be connected to the brick to build a mobile robot. This device also has some sensors that can be connected to the brick and programmed.

In particular, three Lego Mindstorms are used to test some simple platooning scenarios. These three robots are built the same way and all of them have the same sensors except the leader.

6.1.1 Lego Mindstorms EV3 hardware

The robots are built with two wheels that are connected each one to a motor, and another small wheel just needed to maintain the stability of the robot. The connection of the motors to the brick and the wheels allows the robot to move. This motor runs at 160-170 RPM and the wheels have a radius of 2.15cm, what allows to run a robot with a maximum speed of 0.36-0.38 meters per second. In addition, the robots are equipped with sensors. In concrete, all of them has two colour sensors and the followers, two out of three robots, have an infrared sensor. The structure built for the robots, including the sensors and motors, can be seen in figure 6.1.

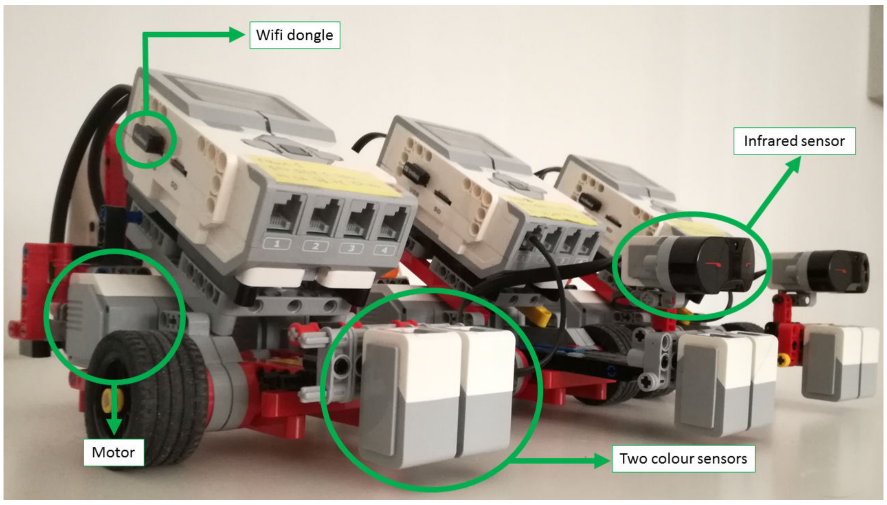


Figure 6.1: Picture of the three robots with its peripherals highlighted

Two of the robots has an infrared sensor that is used to measure the distance between robots. This way it is possible to keep a certain distance between them and make them accelerate or decelerate avoiding collisions. The infrared sensor is placed in the front of the robot to measure the distance between it and the robot in front of it. The leader does not have this sensor as it is assumed that it has no robot in front, and also in the test done there are not obstacles in the way where the robots move.

Lego Mindstorms EV3 robots do not include any sensor to measure position (eg. GPS). So, the position is not taken into account in the implementation and tests. Also, they do not have a gyroscope or compass sensor that could help to maintain the trajectory of the robots straight, reorienting the robot if it turns. But, Lego EV3 robots include colour sensors that indicate which colour is seen through the sensor. This way, it is possible to use colour sensors to make the robots follow a straight trajectory driving them through a straight black line on the floor. So, the colour sensors obtain the colour seen if its black that means that the robot is moving straight but if it is white that means that the robot is not straight and it has to be reoriented to run straight again. Figure 6.2 shows the black line used in the tests to make the robots run straight using the colour sensors.

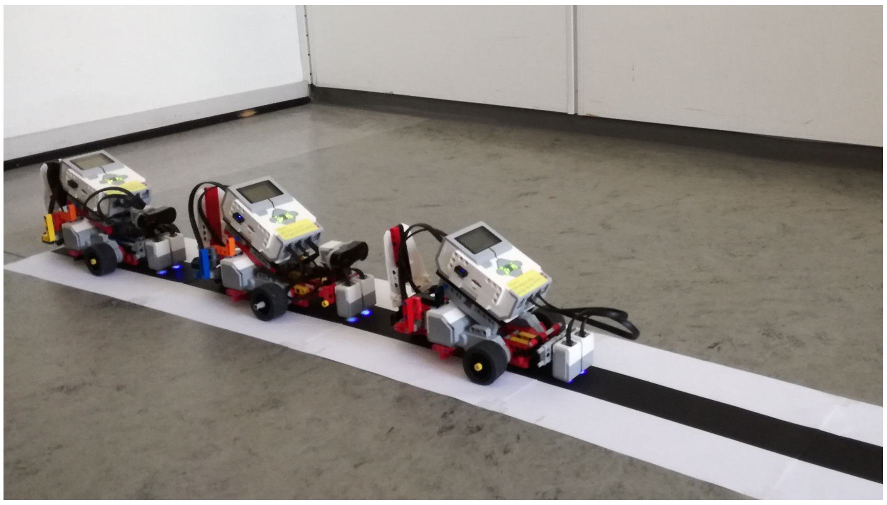


Figure 6.2: Picture of the three robots running through a black line

The WiFi communication is provided to the Lego EV3 robot by connecting a Wifi dongle to the brick. In concrete two different WiFi dongles with the same characteristics has been used, the Edimax EW-7811Un and the Sandberg micro WiFi USB. Both are compatible with IEEE 802.11b/g/n standards, and it is not possible to do network modification on the dongles to simulate the IEEE 802.11p standard used in vehicular platooning scenarios, so it has just been used like this.

6.1.2 Lego Mindstorms EV3 software

LEGO Mindstorms EV3 (31313) allows using its default operative system and firmware to program it. The official EV3 programming software has an integrated development environment to implement programs in a visual way by selecting and mixing blocks of actions. These blocks are a set of instructions that hide the programming code needed to provide a certain action, making easy to non-programmers to develop features but reducing the programming possibilities. For that reason, it has done a search in different software to implement functionalities in Lego Mindstorms that allows developing the features by coding instead of using a visual interface, to have more flexibility.

As it is said before, it is possible to use many others operative systems and firmwares to implement the code in a regular way and not in a graphical way moving blocks. These different firmwares allow to use different programming languages and provide a library with some functions that make easy to provide a functionality.

Ev3dev

Ev3dev[2] is an operative system based in Debian-Linux compatible with the LEGO Mindstorms EV3 robots and other devices as Raspberry Pi.

This operative system provides libraries for programming languages as C, C++, Python, JavaScript, etc. In addition, as it is Debian-Linux based, it has the flexibility to develop anything supported by this operative system. Also, it is possible to install a firmware in the top of this operative system as ROS or LeJOS to use the functions that this firmware provides in their libraries. Ev3dev libraries are available on GitHub and there is a big community working on it, to increase the functionalities, improve them but also to help using them.

Ev3dev also provides a debugger and cross-compiler called *brickstrap*. The library provides many functions to run easily the Lego's peripherals (eg. motor, infrared sensor, etc) and much more sensors that could be connected to the brick. And, it has also a big flexibility to communicate via WiFi the robots by using TCP/IP, UDP, etc. But, ev3dev proposed to use MQTT to communicate the bricks between them as it is simple to use and lightweight to exchange information between several devices.

RobotC

RobotC[9] is a programming platform based in C. It provides a library with functions and instruction to do an implementation in the Lego Mindstorms robots but also other devices. It has many tutorials and examples available and also a forum to ask doubts or report a bug to the community.

But the main disadvantage is that it is not open source and not free software. In addition, there is not a direct command to connect EV3 bricks via WiFi. So, it is not possible to create customised communication as referred on many forum posts present it as a constraint.

LeJOS

LeJOS[5] is a firmware that runs a JAVA virtual machine on the LEGO Mindstorms robots. It allows to do custom connections between the bricks via WiFi, and it has a library with many functions for the device peripherals.

This firmware has an active forum to submit problems or difficulties using this firmware, as well as tutorials to get started. It is easy and flexible to debug the code with this firmware by using the Java Integrated Development Environment (IDE).

Monobrick

Monobrick[6] is a free software, developed in Denmark, to provide functions that help to control robots over the Internet. The library provided is C#-based but it allows to program in C#, F# and IronPython. As the other software mentioned, it has examples to get started and a forum to solve problems. It allows establishing a TCP/IP connection easily between devices by using the WiFi dongle like another *sensor*.

Development environment selected

After reviewing the software possibilities, the software selected to use in the robots is ev3dev because it looks to be the most flexible as it is based in Debian-Linux. Also, because it brings libraries that can help dealing with the controller and communication as well as a developer's community working with this operative system that can help during the implementation. In addition to the big community using a Debian-Linux distribution that can also help.

6.2 Protocol stack

This sections defines, in figure 6.3, the protocol stack to use in the experimental testbed based on the analysis done in section 1.5 and the device limitations.

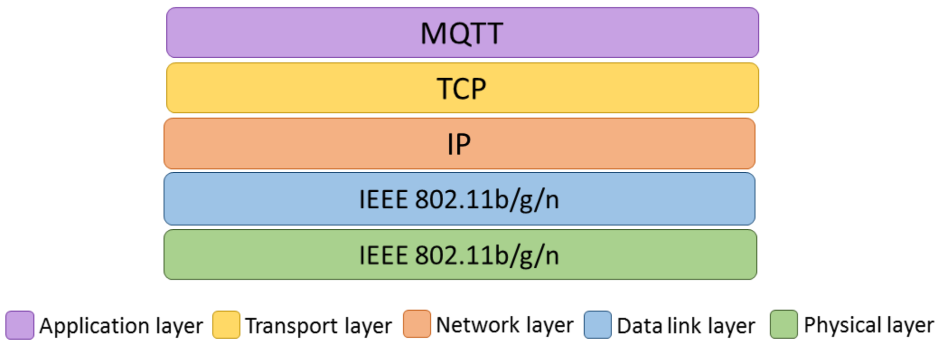
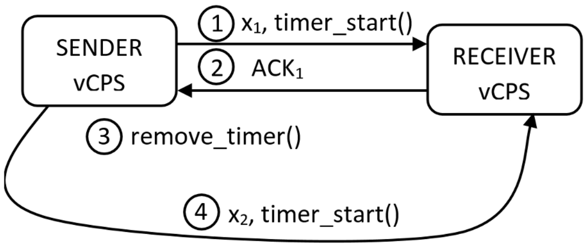


Figure 6.3: Protocol stack for mobile robots' implementation

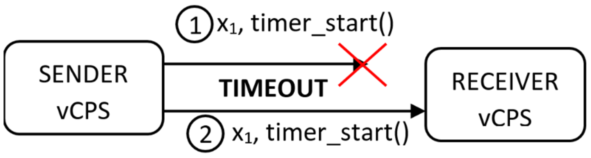
6.3 Mechanisms for the CDM to support crash failures

The platooning communication system uses a hybrid architecture where the maneuvers are achieved by requesting the leader vCPS an always waiting for its acknowledge to can perform that maneuver like in a server/client architecture, in this case, the *leader* vCPS acts like a server and the *follower* vCPSs act like clients. But, the exchange of *CIN* messages, containing vehicle profile's information to achieve string stability, uses a peer-to-peer architecture where there are no request messages just information messages exchange between vCPSs as equals.

In a distributed system like platooning, it is not enough to just send the defined messages in section 3.2 in a certain sequence to achieve the maneuvers. If the messages are sent directly like they are defined messages can be lost, duplicate and corrupt. Depending on the scenario and type of message these failures can be significant or not (eg. If an info message is duplicated there is no much problem as this kind of message is received continuously, but the duplication of an accelerate request can be critical as a the vCPS could collide with another vCPS if it accelerates more than it should). For that reason, it is necessary to use a transmission protocol that eliminates these failures. The Positive Acknowledgement with Re-Transmission (PAR) protocol, used in TCP, eliminates duplication messages, losses using retransmission and corruption using acknowledge messages. But this protocol still has the problem of *floating corpses* that are messages lost in the network for a considerable period of time and suddenly are received confusing the receiver. This problem can be solved extending the PAR protocol with numbered acknowledge messages. So, if a vCPS receives an old message, lost in the network, it can accept it numbering its acknowledge message with the number of the message to inform the sender process that this message was received it does not have to be retransmitted anymore. But in a *real time system* like the platoon is, it has more sense to discard that messages and send a not-acknowledgement, *NACK*, message. Figure 6.4 shows the implementation of the PAR protocol to provide a reliable connection ensuring the reception of messages with no duplicates, x_1 *and* x_2 represents any message type defined in section 3.2.



(a) Execution of PAR protocol with no failures



(b) Execution of PAR protocol where the receiver does not get the first message in time

Figure 6.4: Implementation of PAR protocol

Another important issue in the system is to be fault-tolerance, meaning that the platoon must be running correctly even if a process fails. For that reason, vCPSs should be able to find out crashed vCPSs that cannot communicate anymore and act to reorganise the platoon if it is necessary (eg. If the leader crashes one or more followers should realize of that failure and notify the others to elect a new leader or just to impose the follower just behind the crashed leader to be the new leader; If a follower in the middle of the platoon crashes other vCPS should aware and notify to the leader so the leader can decide to divide the platoon in two avoiding the crashed vCPS that should not continue being a member of the platoon.).

In distributed systems, there is well-known mechanism called *heartbeat* that consists in exchange messages between the members of a group to notify the others that its correct execution. That way, if a node stops sending these messages for a while then the other nodes can assume that this node has crashed. In the platooning communication protocol, the vCPSs are already sending periodic messages to other vCPSs, the *CIN* messages to keep a certain distance and string stability, that can somehow inform the receiver that this sender is alive. If the *CIN* messages are replied with an acknowledge message, as the PAR protocol indicates, then the senders can be aware of the crash failures of its receivers, and also of the failure of its senders (eg. In a platoon where *CIN* messages are sent as figure 6.5 defines, and the first follower fails. The leader, as the sender, will realise that this vCPS has crashed because it will not receive an acknowledge of its *CIN* message. But also, the follower *F2* will realise that this vCPS has crashed as it should periodically receive *CIN* messages from the crashed vCPS.).

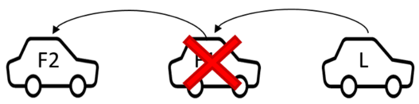


Figure 6.5: Follower crashes in a platoon using one-vCPS look-ahead communication

To provide the PAR protocol it can be used the quality of service possibility that MQTT provides, in concrete QoS2 that ensures the reception of a message *exactly once* is needed to provide the same reliability as the PAR protocol does. The different types of quality of service that MQTT provide are described in section 5.

However, the *heartbeat* systems would need to be implemented on top of the distributed control plane to provide resilience to failure. But, as it is assumed in the experimental testbed that a device is not going to crash indefinitely, the *heartbeat* mechanism has not been implemented.

6.4 Application of the distributed controller protocol

The distributed controller protocol, section 4, is applied during the experimental testbed in the Lego Mindstorms EV3 31313 robots. But, some changes in the controller has been done as this device has some limitation and less sophisticated hardware.

Figure 6.6 shows the controller architecture based on the hardware included in the Lego Mindstorms EV3 31313 robots, defined in chapter 6.1, that is used to test the designed protocol.

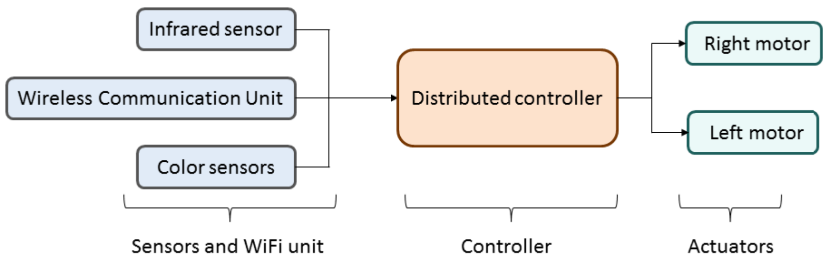


Figure 6.6: Lego EV3's control architecture block diagram

The controller system for the Lego EV3 devices is shown in figure 6.7. Figure 6.7 shows exactly how the system is defined for a platoon of three robots, being *robot 0* the leader of the platoon, and *robot 1* and *robot 2* its followers.

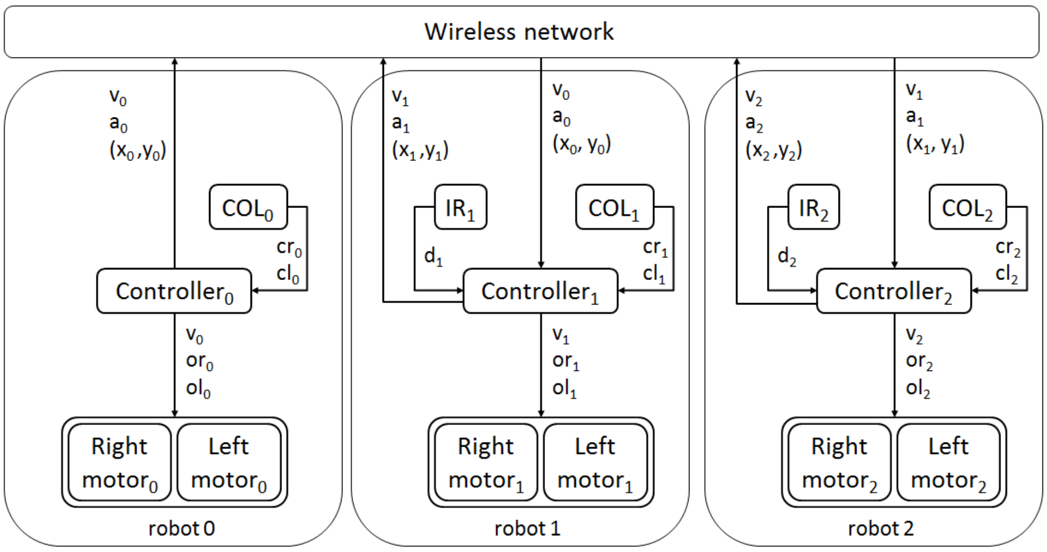


Figure 6.7: Lego EV3's control system block diagram

It is assumed that the integer value close to the *robot* word is a unique value of *i* being i = [0..N] where 0 always indicate the leader of the platoon and N is the number of follower robot in the platoon, in this case N=2. This way, the output from the communication, represented by the *wireless network* input, contains v_{i-1}, a_{i-1} and (x_{i-1}, y_{i-1}) that express the velocity, the acceleration and the position of the robot in front, respectively. The output from the sensors is defined in IR_i, infrared sensor of i, and COL_i, colour sensors of i. Infrared sensor has d_i as output, what means the distance measured between the vCPS in position i and the previous vCPS. Colours sensors produce cr_i and cl_i where cr_i represents the colour obtain from the right sensor of the vCPS i, and cl_i represents the colour obtained from the left sensor of the vCPS i. These values from the sensors and the communication are the input of the *controller$_i$* block that using this input calculates v_i, or_i and ol_i representing the desired velocity of robot i and the desired orientation of the robot i for the right and the left wheel, respectively. The output from the controller is the input of the right and left motors that control the right and the left wheel, respectively.

6.5 Device and network configuration

The first step to take is the installation of the software selected in the subsection 6.1.2 for the implementation, that it is the ev3dev operative system from [2]. Also, it is needed to install a library provided by the operative system that includes functions to use sensors and actuators easily. Ev3dev provide libraries in many programming language, but only the C-library[1] has been installed as the implementation is coded in C. Then, the MQTT broker called Mosquitto has been installed and configured[8]. There are many MQTT broker possibilities, this one is selected because of a recommendation. Next, it is needed to install the MQTT publisher-subscriber library in C language. Like the broker, there are many different libraries and from all of them the Eclipse Paho library has been selected[7] as it has good documentation. Finally, the JSON-C library has been installed, as the messages are written using JSON format and decode using this library[4].

For the network communication, it has set up a wireless access point and configure a DHCP server to provide a fixed IP address for each LEGO Mindstorms robot based on its MAC address. This way it is possible to include the IP in the code of the implementation without being to be change, and also to make easy the communication via ssh with the devices by always using the same IP for each one. The hotspot's configuration steps can be checked easily from an Internet web page like [3] that is why it is not specify here, but the modified system files are in the appendix A.2.

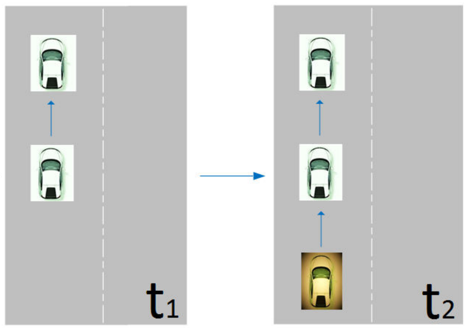
6.6 Distributed control plane implementation

First, figure 6.8 defines the platooning scenario implemented in the Mindstorms EV3 robots based on the limitations of the devices, as the previously defined scenarios in section 1.6 would need a more sophisticated and accurate hardware to be implemented (eg. GPS needed to get the accurate position of vCPSs).

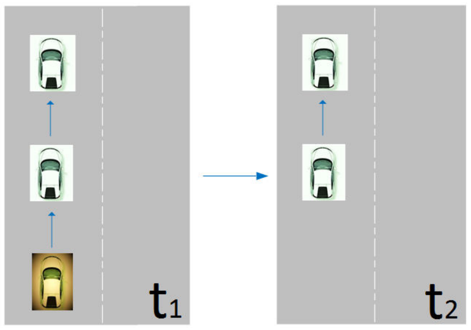
The scenarios to be implemented are a simplification of the scenarios defined in the introduction, but their combination allows to achieve the scenario of figure 1.2. In concrete, it is implemented a scenario where a vCPS joins from the back (figure 6.8a) that it is a simplification of the 1.2b defining a vCPS joining in the middle. A join in the middle is much more complex than the join in the back because it needs to compute the position to join in very accurately to avoid collisions with other vCPSs in the platoon during the maneuver or to avoid the join in an area outside the platoon. For that reason, the join at the back maneuver has been implemented as it only has to take care of the distance with the last vCPS of the platoon to join without crash with it. In addition, the join back maneuver is an interesting scenario that forms part of the vCPS leaves and joins again the platoon scenario defined in figure 1.2a so it can help in the implementation of this more complex scenario.

The other two scenarios implemented describes a leave follower maneuver, the leaving can be from the back (figure 6.8b) or from an intermediate position in the platoon (figure 6.8c). These scenarios has been defined in order to provide the scenario of figure 1.2a, at least separately by having the leave in the middle from figure 6.8c and the join in the back from figure 6.8a that could be connected to achieve the whole scenario, but as it is said before to can compute the exact time and position to join the platoon back from a different lane it is needed a positioning system like a GPS to measure accurately the position of the vCPSs, that is why the whole scenario of figure 1.2a is not implemented.

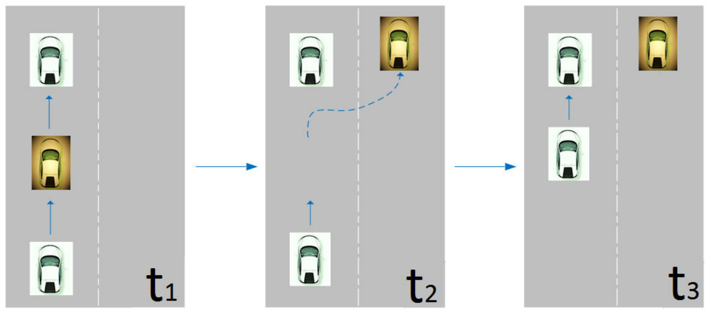
These three scenarios has been achieved by using the sequence of messages defined in section 3.3. In concrete, messages from figure 3.16 are used for the joining at the back scenario of figure 6.8a, and messages of figure 3.14 are used to achieve the leaving scenarios represented in figure 6.8b and 6.8c.



(a) Scenario: vCPS joins at the back a platoon of two vCPSs



(b) Scenario: Last vCPS of a three vCPSs platoon leaves the platoon



(c) Scenario: Middle vCPS of a three vCPSs platoon leaves the platoon

Figure 6.8: Platooning scenarios implemented

The implemented code uses functions from the ev3dev operative system C-library[1] to configure and move the motors, and to configure and read data from the sensors. Also, it uses functions from the MQTT library[7] in order to exchange message between the LEGO Mindstorms EV3 robots. And, the functions of JSON library[4] are used to parse the content of the messages that is formatted using JSON.

Apart from the functions in the libraries, it has coded a header file and a C-file with other functions required in order to integrate the communication and the controller, implementing this way the distributed control plane. Listing A.4 of appendix A.3 contains the header file that defines functions to be use in order to communicate but also to control the movement of the vCPSs.

The other files implemented contain the *main* function executed by a vCPS. All these files have the same structure and contain the functions *send_info*, *check_events*, *drive_straight* and *main*. *Send_info*, *check_events* and *drive_straight* are executed concurrently as each one is executed in a different thread created in the *main* function. *Send_info* function sends *CIN* messages to the vCPS just behind. *Check_events* function tests if there is any messages of type *EVM* or *GID* received, and if so it executes a functions that acts based on the messages received. *Drive_straight* function assures that the vCPS moves through a black line straight and if the line is left then the vCPS correctes its position by turning the driving in order to reach the line again. And, *main* function sets up the communication, configures the sensors and actuators of the LEGO Mindstorms, creates threads for the other three functions and indicates how the vCPS should move.

In the appendix A.3 there is one of these main files, in concrete listing A.5, that contains the code of the leader vCPS increasing its speed after twenty seconds for three times and then stopping. The other main files contain the code of a follower vCPS where the speed of the motors is based on the received *CIN* messages and where the distance between vCPSs is kept based on the values read from the infrared sensors. There is a file for a follower just following the leader vCPS and keeping a distance between itself and the vCPS in front while string stability is provided. Another file has the code of the last vCPS joining the platoon following the same code as the one for the follower but including the creation and sending of a *JOIN* EVM message of type request in order to become a member of the platoon. There is also a file containing the code of a follower vCPS leaving the platoon from the back that it includes the creation and sending of a *LEAVE* EVM message of type request in order to depart. And, some other file that describes the leaving from the middle maneuver including the same code as the file where a vCPS leaves from the back but adding the controller part to change the lane after leaving the platoon from the middle.

The code files implemented, that is the one in appendix A.3, includes the modifications needed to evaluate its performance in section 7.2. This code needs to be compiled with the libraries previously installed in the device like section 6.5 indicates, a possible Makefile[1] for the compilation has been implemented and it can be seen in the link included in appendix A.3. To execute the code, it is just needed to run the executable file as it is done in a Linux terminal (ie. using ./name_file being *name_file* the executable file to be run).

[1]Makefile: File containing directives used with the make build automation tool to compile a set of files.

7. Performance evaluation

First, evaluation metrics has been selected getting ideas from the literature[20][22][23]. Then, these metrics has been defined to indicate how they are going to be tested. Finally, the evaluation metrics has been tested during the LEGO Minstorms EV3 robots' execution and the results has been analysed.

7.1 Definition of evaluation metrics

One of the points to evaluate is the *maneuver performance* to analyse how the maneuvers influence the execution. The maneuver performance is going to be tested using different metrics: duration, failure rate and packet loss, bandwidth consumption and throughput impact.

Duration means the time it takes to achieve a maneuver, and it can be defined like

$$duration_{maneuver} = t_{m_n} - t_{m_1} \tag{1}$$

For a maneuver that needs to exchange n messages, its duration, $duration_{maneuver}$, is the time difference between the first message sent, t_{m_1}, and the last message received, t_{m_n}, to achieve the maneuver. As it is assumed that the communication medium can have delays, the duration of a maneuver can be different from on measure to another. For that reason, the test of this metric should be done as the average of a certain number of measurements to provide a valid result.

Failure rate gives the quantity of packages lost during a concrete maneuver scenario. In this metric is assume that only messages related with the tested maneuver are sent. Failure rate can be defined as

$$failure\_rate_{maneuver} = \frac{m_{maneuver}}{m_{sent_{maneuver}}} \tag{2}$$

where $m_{maneuver}$ is the number of messages needed to achieve the maneuver defined for a certain vCPS and $m_{sent_{maneuver}}$ is the number of messages sent through the network in order to achieve that *maneuver*.

Another metric is the packet loss over the wireless link to measure the communication failure. This way, the packet loss is the difference between the packets received and the packets sent.

$$packet\_loss_{maneuver} = m_{sent_{maneuver}} - m_{received_{maneuver}} \tag{3}$$

where m_{sent} is the number of messages sent through the network and $m_{received}$ is the number of messages received.

Another metric is the bandwidth consume during a maneuver. This metric indicates the number of bits sent in the sequence of messages needed to execute a certain maneuver. To do this first, it must be measured the size of the different kind of messages proposed in the communication protocol. Then, it has to be identified which sequence of messages are needed to achieve a maneuver. Finally, the total bandwidth is the addition of the sizes of every message required plus the size of the acknowledge messages sent from the TCP protocol. Equation (4) reflects the bandwidth if there is no QoS selected in MQTT, QoS0, as it does not take into account the additional ACK packets sent by MQTT with QoS1 or QoS2.

$$bandwidth\_consumed_{maneuver} = 2 * \sum_{i=1}^{m_{maneuver}} (msg\_size[i] + tcp_{ack}\_size) \tag{4}$$

where it is assumed that a maneuver is defined by a number of $m_{maneuver}$ messages and its size is contained in an array called *msg_size* where i indicates the position of that message i in the sequence of messages. The sum value is multiplied by two because it has to be taken into account the sending of the message to the broker from the publisher and the reception of the message by the subscriber (ie. broker sends the message to the vCPSs subscribed to that topic).

The last metric to evaluate is the impact of the maneuvers in the throughput. This can be measure knowing the bandwidth consumed by a maneuver and the time duration of a maneuver, $duration_{maneuver}$.

$$throughput\_impact_{maneuver} = \lfloor \frac{bandwidth\_consumed_{maneuver}}{duration_{maneuver}} \rfloor \tag{5}$$

where the $bandwidth\_consumed_{maneuver}$ is defined in (4), and the definition of $duration_{maneuver}$ is in (1).

7.2 Performance results

Every metric has been evaluated for the three scenarios defined for implementation in section 6.6 with three LEGO Mindstorms EV3 robots using an area composed of two black lines of 570 cm long, that simulate the road where the vCPSs drive.

First to evaluate the $duration_{maneuver}$ performance metric defined in (1), the follower code that achieves the maneuver needs to include the C-function *gettimeofday* in order to measure the time interval between the request for a maneuver and its resolution, appendix A.3. This metric is measured ten times to get an average time of the $duration_{maneuver}$ value because there can be some value differences from one measurement to another due to the communication medium. Tables 7.1, 7.2 and 7.3 show the results of the ten executions, and the average of the $duration_{maneuver}$ time for each one of the maneuvers measured. This time defines how fast the sequence of messages to achieve a certain maneuver is received by the follower requesting that maneuver, allowing the follower to compute the action with its distributed controller based on the information received in the messages of the CDM protocol.

	Join back maneuver	
	Communication	**Communication and reduce gap**
Test 1	0.187753	18.518347
Test 2	0.202330	20.387474
Test 3	0.162184	18.52486
Test 4	0.214999	20.463871
Test 5	0.155732	17.534069
Test 6	0.187827	18.462982
Test 7	0.163555	17.896031
Test 8	0.202067	20.471877
Test 9	0.209586	20.627352
Test 10	0.178953	18.420495
Average	0.186499	19.1307358

Table 7.1: Results of the $duration_{join\_back}$ metric evaluation, measured in seconds.

	Leave back maneuver
	Communication
Test 1	0.155197
Test 2	0.155696
Test 3	0.140895
Test 4	0.108600
Test 5	0.123346
Test 6	0.121391
Test 7	0.130693
Test 8	0.198835
Test 9	0.145909
Test 10	0.156928
Average	0.143749

Table 7.2: Results of the $duration_{leave\_back}$ metric evaluation, measured in seconds.

	Leave middle maneuver	
	Communication	**Communication and lane change**
Test 1	0.284857	6.480074
Test 2	0.252923	5.602009
Test 3	0.263346	6.330987
Test 4	0.233882	6.056165
Test 5	0.223925	5.093712
Test 6	0.262286	5.226361
Test 7	0.2521985	5.466213
Test 8	0.214019	4.754800
Test 9	0.233491	5.437649
Test 10	0.205285	4.618048
Average	0.242621	5.506602

Table 7.3: Results of the $duration_{leave\_middle}$ metric evaluation, measured in seconds.

The results in tables 7.1 and 7.2 show that the performance of a join back maneuver and a leave back maneuver need an average of 0.19 and 0.15 seconds to be executed, respectively. It has sense that these results are similar as both maneuvers need just two messages to be executed, *JOIN_REQ* and *JOIN_ACK* messages for a join back maneuver, and, *LEAVE_REQ* and *LEAVE_ACK* for the leave back maneuver when the follower leaving is the last one. The difference between times could appear because in the join maneuver the messages are sent to the leader after the leader has walk between 110 and 120 cm but the vCPS joining has not move, so they are separated this distance; while in the leaving maneuver they are just separated by the size of the vCPSs plus the inter-vCPSs distance kept. This assumption in the difference of times has not been proved, checking different responses time for different distances but it seems a fair assumption. Table 7.3 shows the results of the execution of a leave from the middle maneuver, this maneuver need a three messages to be exchanged, *LEAVE_REQ*, *LEAVE_ACK* and *ID_REQ*, instead of two like the leave from the back and join maneuver that is why its duration time for the communication is slightly higher than the others.

The join maneuver and the leave from the middle maneuver also measure the time from the request message sent to the vCPS controller finishing the maneuver, what means to cover the gap actually forming part of the platoon in the join maneuver and to change the driving lane in the leave from the middle maneuver. As it can be seen in table 7.1 the resulting times are quite similar and it takes around 19 seconds to cover the whole gap because the Lego Mindstorms robots are driving at 6-6.6 cm/s. This speed is computed by the controller based on the distance to the vCPS in front and the speed received through the communication and could change depending on these parameters. Also, it could be possible to reduce the time to cover the gap increasing the speed but this could lead into undesirable collisions if the vCPS get too close with a high velocity and it has not enough time to brake or reduce its speed before reaching the vCPS in front. In table 7.3 the results to change the lane are also quite similar between them, taking around 5.5 seconds to perform the change. This value can be used to analyse how the system performs and if it changes a lot from one execution to another, as it is reflected it does not really change. But, it is something dependent on the speed and the distance from one lane to another so it is not representative of global scenario as these parameters could change leading into a change of the *communication and lane change* time.

Then, the $failure\_rate_{maneuver}$ for a certain maneuver has been measured using (2). The $m_{maneuver}$ variable indicates the number of messages needed to achieve that maneuver, this values can be obtained from figures in section 3.3 that defines state machines with the sequence of messages required for some maneuvers. The $m_{sent_{maneuver}}$ variable defines the number of messages sent through the network in order to achieve that maneuver, in the optimal case this value is the same than the one in the $m_{maneuver}$ variable giving a $failure\_rate_{maneuver}$ value of 1 what indicates that messages sent are exactly the same as the ones that should be sent. Another possibility is to have the value of $m_{sent_{maneuver}}$ lower than the value on $m_{maneuver}$ what means that not all the messages has been received due to network failures so the maneuver cannot be achieved. And as the last option, if $m_{sent_{maneuver}}$ is higher than the value on $m_{maneuver}$ that means that there has been sent duplicates of the packets caused by network failure or delays in order to can achieve the maneuver. The equations 7.2 and 7.2 show the $failure\_rate_{maneuver}$ compute for the join back, the leaving from the back and the leaving from the middle maneuvers. For the maneuver join back and leave back is used the value *2* for the variables m_{join_back} and m_{leave_back} as only two messages are needed to achieve each maneuver, *JOIN_REQ* and *JOIN_ACK* messages, and *LEAVE_REQ* and *LEAVE_ACK* messages respectively. However, the leave middle maneuver needs three messages to achieve the maneuver, *LEAVE_REQ*, *LEAVE_ACK* and *ID_REQ*, that is why m_{leave_middle} has the value *3*. The values of $m_{sent_{join\_back}}$, $m_{sent_{leave\_back}}$ and $m_{sent_{leave\_middle}}$ has been measured by executing both scenarios in the LEGO Mindstorms robots ten times and doing the average of these results as in the executions there are no failures in the network and the messages sent are always two in the case of the join back and leave back maneuver and three in the case of the join middle maneuver. These values has been obtained by using the variables *sent_msg* and *recv_msg* included in the code. For that reason, it is avoided a table with the results of each test and the average as it would contain always the same value. Just counting the messages sent and receive cannot be a good solution as MQTT is running over TCP what means that some of the packets can be retransmitted over TCP to ensure its reception, but in MQTT are just seen like one transmission. For that reason, it has also run Wireshark over the network where the robots are sending/receiving the messages in order to check for TCP retransmissions of the *EVM* messages. But, the network used to evaluate the performance is really reliable so there was no TCP retransmission of *EVM* messages during all the tests done. One output of the Wireshark captures for each maneuver is attached in the appendix A.4, only one capture is included for each maneuver since all reflect no retransmission being equal outputs.

$$failure\_rate_{join\_back} = \frac{m_{join\_back}}{m_{sent_{join\_back}}} = \frac{2}{2} = 1$$

$$failure\_rate_{leave\_back} = \frac{m_{leave\_back}}{m_{sent_{leave\_back}}} = \frac{2}{2} = 1$$

$$failure\_rate_{leave\_middle} = \frac{m_{leave\_middle}}{m_{sent_{leave\_middle}}} = \frac{3}{3} = 1$$

Next, the $packet\_loss_{maneuver}$ has been measured by counting the number of messages sent and the messages received and calculating the difference between these two values as 3 defines. From the $failure\_rate_{maneuver}$ results it seems like the number of packet loss in the network is none, as the network has not many interference and as the LEGO Mindstorms robots are at most 570cm far so it is not as much distance to can lose the connectivity.

$$packet\_loss_{join\_back} = m_{sent_{join\_back}} - m_{received_{join\_back}} = 2 - 2 = 0$$

$$packet\_loss_{leave\_back} = m_{sent_{leave\_back}} - m_{received_{leave\_back}} = 2 - 2 = 0$$

$$packet\_loss_{leave\_middle} = m_{sent_{leave\_middle}} - m_{received_{leave\_middle}} = 3 - 3 = 0$$

The *bandwidth_consumed* has been measured by measuring the size of the communication messages, and then applying the (4) using these values. A traffic network analyser, Wireshark[15], has been used to measure the real size of the packets sent.

The packet's size changes depending on the type of the message sent. For that reason, it has been measured the size of the messages used in the implementation. Section 3.2 defines *CIN*, *EVM* and *GID* messages, but not all the sub-messages included in these messages has been measure. Only, *INFO* message defined as a *CIN* message; *JOIN* and *LEAVE* message from the *EVM* messages and not *ACCELERATION* message as it is not used in the implementation; and *ID* message from the defined *GID* messages avoiding the measurement of *NEW_LEADER* and *DISCOVERY* messages as these messages are not implemented nor used in the testbed scenarios. In addition, ACK packets from MQTT has also been measured as these packets are received when QoS is configured, and TCP acknowledgement packets as they are also received after every message sent. Table 7.4 shows the packet's size (ie. size of the IP, TCP and MQTT header, TCP options and CDM from 5.3) and the message size (ie. CDM field from 5.3) obtained using Wireshark, appendix A.5 contains the screenshots of the Wireshark's outputs.

Message type	**Packet size (bytes)**	**Message size (bytes)**
Context Information Message (CIN)		
INFO message	144	71
Event Maneuver Message (EVM)		
JOIN message	142	69
LEAVE message	128	55
Group Identifier message (GID)		
ID message	120	47
MQTT ACK	70	
TCP ACK	66	

Table 7.4: Packet size

The *bandwidth_consumed* of a join back, a leaving follower from the back and a leaving follower from the middle maneuver is computed by the application of (4). The values of the variables $m_{maneuver}$ for m_{join_back}, m_{leave_back} and m_{leave_middle} has been obtained from the states machines in 3.3, but they could have also been obtained from the values used for the failure rate maneuver metric. The values for these variables are the size of two *JOIN* messages for the join back maneuver, two *LEAVE* messages for the leaving back maneuver and two *LEAVE* messages and one *ID* message for the leaving middle maneuver, respectively.

$$bandwidth\_consumed_{join\_back} = 2 * (142 + 66 + 142 + 66) = 832\ bytes$$

$$bandwidth\_consumed_{leave\_back} = 2 * (128 + 66 + 128 + 66) = 776\ bytes$$

$$bandwidth\_consumed_{leave\_middle} = 2 * (128 + 66 + 128 + 66 + 120 + 66) = 1148\ bytes$$

The bandwidth consumed it has been measured using QoS0 in MQTT, what means that there are no MQTT ACK messages sent. These values of the bandwidth will increase if QoS1 or QoS2 is provided by MQTT.

To reduce the bandwidth, it is possible to reduce the size of the messages (eg. not use JSON and just add the content directly in the message without indicating what each value means), and also by using a lower level protocol to send the packets as TCP or UDP sockets or an application level that sends directly the messages to the receiver and does not use a third party like the broker.

Finally, the $throughtput\_impact_{maneuver}$ has been measured by inserting the results, obtained in this section, of $bandwidth\_consumed_{maneuver}$ and $duration_{maneuver}$ in the equation (5) as follows.

$$throughput\_impact_{join\_back} = \lfloor\frac{bandwidth\_consumed_{join\_back}}{duration_{join\_back}}\rfloor = \lfloor\frac{832}{0.186499}\rfloor = 4461\ bytes/s$$

$$throughput\_impact_{leave\_back} = \lfloor\frac{bandwidth\_consumed_{leave\_back}}{duration_{leave\_back}}\rfloor = \lfloor\frac{776}{0.143749}\rfloor = 5398\ bytes/s$$

$$throughput\_impact_{leave\_middle} = \lfloor\frac{bandwidth\_consumed_{leave\_middle}}{duration_{leave\_middle}}\rfloor = \lfloor\frac{1148}{0.242621}\rfloor = 4731\ bytes/s$$

These results show a different on the bandwidth of 1000 bytes/s approximately what can be led by the increased of the distance between two vCPSs, but it has been said during the measure of the duration that this cannot be proved. And, it can just be that during the measure of one maneuver and another the network was more overloaded.

7.3 Study and comparison of the results

This section review the results obtained in section 7.2 and compared them with results obtained in the literature for the same scenarios and with the hypothetical results defined by the model in chapter 5.

The results obtained for the duration metric defined in (1) can be compared with the results of *Fig. 17.* of [20]. This figure shows the average time need it to perform different platooning maneuvers based on the communication messages defined in the paper. In [20] the duration time for the last follower leaving is less than 4 seconds while the results of this thesis' evaluation, figure 7.2, is less than 0.15 seconds in average and the time for a middle follower leaving in the paper is between 7.5 and 8 seconds while the proposed system gives a time of fewer than 0.25 seconds for the communication and 5.5 seconds to perform the whole maneuver taking into account the communication and the lane change of the vCPS. So, as it can be seen the distributed control plane proposed and evaluated gives better results for the leaving maneuver. The join maneuver cannot be compared with [20] as it has not been evaluated the join maneuver in this paper. Even though this comparison gives better results to the proposed system, it cannot be ensure that the distributed control plane defined in this thesis performs better as the evaluation of the defined systems is done with three LEGO Mindstorms ev3, using the WiFi standard 802.11b/g/n and a speed for each device changes is around 0.072 - 0.076 m/s. But, the evaluation done using VENTOS' simulator[10] in [20] uses a platoon of ten vehicles, the WiFi amendment for vehicular systems 802.11p and a speed for each vehicle of 20 m/s.

The integration of messages and the controller computation needed to provide maneuvers in the system is defined in figures 5.5 and 5.6 showing the approximate *duration* must take theoretically. In theory, this duration is around 0.15 seconds to execute the communication of a join maneuver that will be at least 0.2 seconds until the controller will receive a *CIN* message indicating the speed of the platoon and therefore accelerate the vCPS that wants to join in order to reach the platoon and keep the safety distance and not more. And, it is around 0.1 seconds to can leave the platoon what implies to send a *leave* request and wait for a *leave* acknowledge message to can understand by the controller that the vCPS can remove its platoon membership by stopping its communication and modifying its driving. Comparing this theoretical results with the performance results it is seen that the *join* and the *leave* maneuvers are slightly more costly in the communication duration.

8. Conclusion

After doing a review of the literature there are many proposals to provide platooning functionality based on simulations, but not such many experimenting with mobile devices as the LEGO Mindstorms robots used in this thesis. In addition, most of the papers are focused on the controller or the communication part but they do not explain in details how to integrate both protocols to achieve platooning. For that reason, this thesis has focused on the definition of a distributed control plane defining how to achieve platooning through communication but also describing how communication is connected with the controller protocol in order to drive vCPSs keeping a certain distance while supporting string stability through the platoon and performing maneuvers.

The objectives of the thesis has been achieved by defining the distributed control protocol in chapter 2, designing the Cooperative Driving Messages (CDM) protocol in chapter 3 to communicate vCPSs through an application layer protocol, analysing the controller formulas to be taken into account to provide platooning in chapter 4, describing the integration of the communication and the controller in the distributed control protocol in chapter 5, and implementing the distributed control protocol defined in the previous sections using the LEGO Mindstorms EV3 devices as chapter 6 contains.

Finally, it has been proposed and evaluated some metrics to measure the performance of the distributed control protocol and its implementation using the LEGO Mindstorms and Wireshark tool in chapter 7. The evaluation of these metrics is tied to certain network and to a reduce number of executions, due to the use of a device implementation to test it and not simulation what it will allow running more tests with a wider selection of the parameter, and to evaluate different network characteristics giving a more variety of results and more realistic because of this diversity. Nevertheless, the performance evaluation can give an idea of the distributed controller performance and the possible enhancements that can be done to improve it.

8.1 Future work

Suggested work to improve the content of this thesis is the incorporation of a GPS module into the LEGO Mindstorms EV3 robots to implement completely the desirable scenarios defined in 1.2. Also, the use of a simulator or the increase of the evaluation scenarios, using the LEGO Mindstorms EV3 robots, in terms of different communication networks and areas to provide a more complete evaluation of the distributed control plane for cooperative vCPSs.

Bibliography

[1] Ev3dev library in C. `https://github.com/in4lio/ev3dev-c`. [Online; accessed 25-April-2017].

[2] Ev3dev operative system. `http://www.ev3dev.org`. [Online; accessed 05-March-2017].

[3] Guide to configure a hotspot. `https://www.maketecheasier.com/set-up-raspberry-pi-as-wireless-access-point/`. [Online; accessed 20-April-2017].

[4] JSON-C library. `https://github.com/json-c/json-c`. [Online; accessed 25-April-2017].

[5] LeJOS firmware. `http://www.lejos.org/`. [Online; accessed 05-March-2017].

[6] MonoBrick firmware. `http://www.monobrick.dk/`. [Online; accessed 05-March-2017].

[7] MQTT client - Paho. `https://www.eclipse.org/paho/`. [Online; accessed 25-April-2017].

[8] MQTT server - Mosquitto. `https://projects.eclipse.org/projects/technology.mosquitto`. [Online; accessed 25-April-2017].

[9] RobotC firmware. `http://www.robotc.net/`. [Online; accessed 05-March-2017].

[10] M. Amoozadeh. Vehicular network open simulator (ventos), website. `http://maniam.github.io/VENTOS/`. [Online; accessed 17-May-2017].

[11] Eclipse. Mqtt quality of service levels. `http://www.eclipse.org/paho/files/mqttdoc/MQTTClient/html/qos.html`. [Online; accessed 06-May-2017].

[12] European Telecommunications Standards Institute (ETSI). *ETSI EN 302 637-2 V1.3.2 - Intelligent Transport Systems (ITS); Vehicular Communications; Basic Set of Applications; Part 2: Specification of Cooperative Awareness Basic Service.* ETSI, 2014.

[13] European Telecommunications Standards Institute (ETSI). *ETSI EN 302 637-3 V1.2.2 - Intelligent Transport Systems (ITS); Vehicular Communications; Basic Set of Applications; Part 3: Specifications of Decentralized Environmental Notification Basic Service.* ETSI, 2014.

[14] P. Fernandes and U. Nunes. *Platooning with IVC-Enabled Autonomous Vehicles: Strategies to Mitigate Communication Delays, Improve Safety and Traffic Flow.* IEEE Transactions on intelligent transportation systems, vol. 13, no. 1, 2012.

[15] Wireshark Foundation. Wireshark network analyzer. `https://www.wireshark.org/`. [Online; accessed 06-May-2017].

[16] J. Ploeg M. Van de Molengraft G. Naus, R. Vugts and M. Steinbuch. *Towards on-the-road implementation of cooperative adaptive cruise control.* Proc. 16th World Congr. Exhib. ITS World, 2009.

[17] K. Abboud H. Zhou H. Zhao W. Zhuang H. Peng, D. Li and X. Shen. *Performance analysis of IEEE 802.11p DCF for Multiplatooning Communications with Autonomous Vehicles.* IEEE Globecom, 2015.

[18] Hong Linh; Stanford-Clark Andy Hunkeler, Urs; Truong. *MQTT-S - A publish/subscribe protocol for Wireless Sensor Networks.* 3rd International Conference on Communication Systems Software and Middleware and Workshops, pp. 791-798, 2008.

[19] John B. Kenney. *Dedicated Short-Range Communications (DSRC) Standards in the United States.* Proceedings of the IEEE | Vol. 99, No. 7, July 2011.

[20] C. Chuah-H. Zhang M. Amoozadeh, H. Deng and D. Ghosal. *Platoon management with cooperative adaptive cruise control enabled by VANET.* Elsevier Vehicular Communications, 2015.

[21] H. Tsai M. Segata, R. Lo Cigno and F. Dressler. *On Platooning Control using IEEE 802.11p in Conjunction with Visible Light Communications.* 12th Annual conference on wireless on-demand network systems and services (WONS), 2016.

[22] S. Joerer C. Sommer M. Gerla R. Lo Cigno M. Segata, B. Bloessl and F. Dressler. *Toward Communication Strategies for Platooning- Simulative and Experimental Evaluation.* IEEE Transactions on vehicular technology, vol. 64, no. 12, 2015.

[23] S. Joerer F. Dressler R. Lo Cigno M. Segata, B. Bloessl. *Supporting platooning maneuvers through IVC: An initial protocol analysis for the join maneuver.* IEEE Xplore, 2014.

[24] George H. Mealy. *A Method for Synthesizing Sequential Circuits.* Bell System Technical Journal. pp. 1045-1079, 1955.

[25] Mohd Saufy Rohmad Muhammad Harith Amaran, Nazmin Arif Mohd Noh and Habibah Hashim. *A Comparison of Lightweight Communication Protocols in Robotic Applications.* Elsevier - IEEE International Symposium on Robotics and Intelligent Sensors, 2015.

[26] OASIS. Mqtt standard definition. `http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.pdf`. [Online; accessed 06-May-2017].

[27] Jorge Pereira. Sending and receiving messages with mqtt. `http://www.ev3dev.org/docs/tutorials/sending-and-receiving-messages-with-mqtt/`. [Online; accessed 02-March-2017].

[28] M. Petrov R. Kazala, A. Taneva and St. Penkov. *Autonomous robot integration in Suveillance System. Architecture and communication protocol for system cooperation.* International Power Electronics and Motion Control Conference and Exposition, 2014.

[29] M. Petrov R. Kazala, A. Taneva and St. Penkov. *Wireless Network for Mobile Robot Application.* IFAC (International Federation of Automatic Control) conference paper, 2015.

[30] Anna Maria Vegni and Valeria Loscrí. *A Survey on Vehicular Social Networks.* IEEE communication surveys tutorials, vol. 17 no. 4, 2015.

[31] Wikipedia. IEEE 802.11. `https://en.wikipedia.org/wiki/IEEE_802.11`. [Online; accessed 16-March-2017].

[32] Yong-qiang Zhang Zhou Yun and Zhou Wan-zhen. *Present Situation and Future Development of Multiple Mobile Robot Communication Technology.* International Conference on Computer Application and System Modeling (ICCASM), 2010.

A. Appendix

A.1 Gantt chart

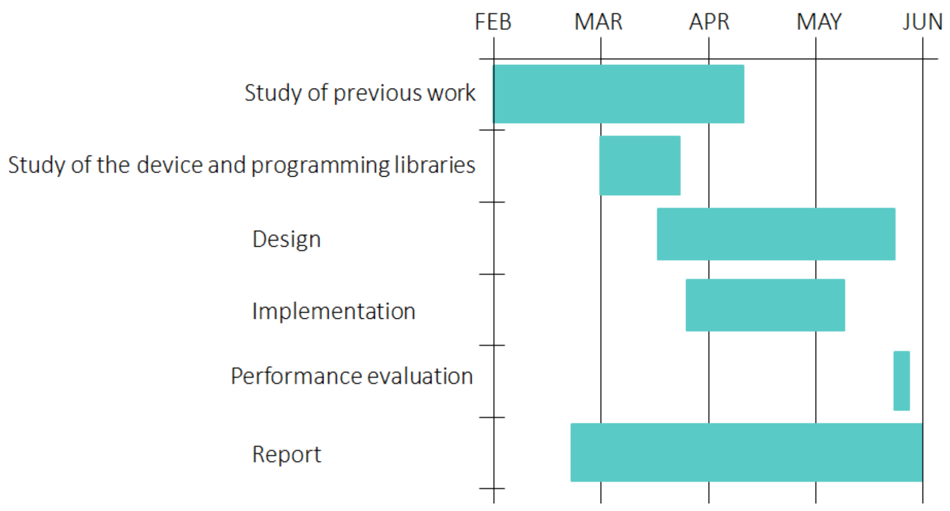


Figure A.1: Gantt chart defining the tasks achieved in the thesis

A.2 Configuration files to configure a hostspot with fixed IPs

Listing A.1: File: /etc/network/interfaces

```
auto lo
iface lo inet loopback
allow-hotplug wlan1

iface wlan1 inet static
address 192.168.1.249
netmask 255.255.255.248
gateway 192.168.1.46

pre-up iptables-restore < /etc/iptables.ipv4.nat
```

Listing A.2: File: /etc/dhcp/dhcpd.conf

```
# The ddns-updates-style parameter controls whether or not the server will
# attempt to do a DNS update when a lease is confirmed. We default to the
# behavior of the version 2 packages ('none', since DHCP v2 didn't
# have support for DDNS.)
ddns-update-style none;

default-lease-time 600;
max-lease-time 7200;

# If this DHCP server is the official DHCP server for the local
# network, the authoritative directive should be uncommented.
#authoritative;
```

```
# Use this to send dhcp log messages to a different log file
log-facility local7;

# Internal subnet
subnet 192.168.1.248 netmask 255.255.255.248 {
   range 192.168.1.253 192.168.1.254;
   option subnet-mask 255.255.255.248;
   option routers 192.168.1.249;
   option broadcast-address 192.168.1.255;
   default-lease-time 600;
   max-lease-time 7200;
   group{
      host robot1 {hardware ethernet 74:DA:38:7E:C7:40;
      fixed-address 192.168.1.250;}
      host robot2 {hardware ethernet 80:1F:02:EA:8F:DA;
      fixed-address 192.168.1.251;}
      host robot3 {hardware ethernet 74:DA:38:7E:C7:73;
      fixed-address 192.168.1.252;}
  }
}
```

,

```
# Path to dhcpd's config file (default: /etc/dhcp/dhcpd.conf).
#DHCPD_CONF=/etc/dhcp/dhcpd.conf

# Path to dhcpd's PID file (default: /var/run/dhcpd.pid).
#DHCPD_PID=/var/run/dhcpd.pid

# Additional options to start dhcpd with.
#Don't use options -cf or -pf here; use DHCPD_CONF/ DHCPD_PID instead
#OPTIONS=""

# On what interfaces should the DHCP server (dhcpd) serve DHCP requests?
#Separate multiple interfaces with spaces, e.g. "eth0 eth1".
INTERFACES="wlan1"
```

Listing A.3: File: /etc/hostapd/hostapd.conf

```
interface=wlan1
driver=nl80211
ssid=robots-router
hw_mode=g
channel=6
macaddr_acl=0
auth_algs=1
ignore_broadcast_ssid=0
wpa=2
wpa_passphrase=robots-router
wpa_key_mgmt=WPA-PSK
wpa_pairwise=TKIP
rsn_pairwise=CCMP
```

A.3 Code of the distributed control plane for cooperative vCPSs

This appendix includes a sample of the implemented code, in concrete the header file describing the functions implemented, and one of the main files. To see all the files implemented go to: `https://gist.github.com/almsv/22067424c2692082ce411aa766d9f4f2`. In the link, there is a comment at the end describing in details the content of each file, how to compile them and how to execute them.

Listing A.4: Header file that defines the communication an controller functions implemented

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <MQTTClient.h>
#include <pthread.h> //To can use pthread_mutex_t
#include <unistd.h>
#include <json-c/json.h> //To decode messages

#define QOS         0
#define TIMEOUT     10000L

#define MOTOR_LEFT      OUTA
#define MOTOR_RIGHT     OUTB
#define MOTOR_BOTH      ( MOTOR_LEFT | MOTOR_RIGHT )
#define MAX_SPEED       1050 //tacho_get_max_speed( MOTOR_RIGHT, 0 )
#define MIN_SPEED       0
#define MIN_SECURITY_DISTANCE 5
#define MAX_SECURITY_DISTANCE 10

#define COLOR_SENSOR_LEFT       INPUT_2
#define COLOR_SENSOR_RIGHT       INPUT_3

#define IR_CHANNEL      0
#define COLOR_CHANNEL   0

#define _SNC_TURN_ANGLE_ 10
#define DEGREE_TO_COUNT( d )  (( d ) * 260 / 90 )

#define SIZE_QUEUE 20

typedef enum {INFO, JOIN, LEAVE, ACCELERATE, ID, LEADER} cdm;
typedef enum {REQ, ACK, NACK} petition;

typedef struct{
    int speed;
    int location_x;
    int location_y;
    int acceleration;
} CIN_struct;

typedef struct{
   cdm msg_type;
    int id_sender; //ID of the vehicle that has initiate the request
    int id_receiver; //ID of the vehicle that must receive the request
    petition petition_type;
    int position_join;
    int acceleration;
} EVM_struct;

typedef struct{
    cdm msg_type;
    petition petition_type;
    int old_id;
    int new_id;
    char address_leader[25];
    char address_request[25];

} GID_struct;

extern const char* msg_name[];
extern const char* petition_name[];

/**
 * Function that returns the speed of an CIN_struct structure.
 *
 * @return : Integer value with a speed value.
 */
int get_speed();

/**
 * Function that translate the proximity given in the parameters to a
 * distance in cm.
 *
 * @param prox: Integer with the proximity given by the infrared sensor.
 * @return : Integer that contains a distance in cm.
 */
int get_distance(int prox);

/**
 * Function that modifies the platoon_id of with the value given in the
 *       parameters.
 *
 * @param id: Integer that contains the identifier for this vehicle.
 */
void set_platoon_id(int id);

/**
 * Function returns the platoon_id of the vehicle.
 *
 * @return : Integer that contains the platoon id of the vehicle.
 */
int get_platoon_id();

/**
 * Function that returns a value that indicates if the vehicle must stop
 * or must keep running.
 *
 * @return : Integer with value 1 if the vehicle must STOP, and 0 if
 *       it should keep running.
 */
int stop();
```

```
/**
 * Function that returns a value that indicates if there is any evm
 * message in the queue that must be processed.
 *
 * @return : Integer with value 1 if there are messages in the queue,
 * and 0 if the queue is empty.
 */
int evm_to_process();

/**
 * Function that returns a value that indicates if there is any gid
 * message in the queue that must be processed.
 *
 * @return : Integer with value 1 if there are messages in the queue,
 * and 0 if the queue is empty.
 */
int gid_to_process();

/**
 * Function returns the size of the vehicles platoon.
 *
 * @return : Unsigned integer that contains the platoon  size.
 */
unsigned int get_size_platoon();

/**
 * Function that increase the velocity of the vehicle.
 */
void accelerate();

/**
 * Function that decrease the velocity of the vehicle.
 */
void decelerate();

/**
 * Callback function to enable asynchronous notification of delivery
 *  of messages. The function is called after publishing a message.
 *  It is just use it there is QoS configure.
 *
 * @param context: Pointer to the context value passed to
 *       MQTTClient_setCallbacks(), which contains the
 *       application-specific context.
 * @param dt: MQTTClient_deliveryToken associated with the
 *       published message. To check that all messages have
 *       been correctly published.
 */
void delivered(void *context, MQTTClient_deliveryToken dt);

/**
 * Callback funtion execute when a new message, publish in a topic which
 * the client is subscribed, has been received from the server.
 * This function is executed on a separate thread.
 * @param context: Pointer to the context value passed to
 *       MQTTClient_setCallbacks(), which contains the
 *       application-specific context.
 * @param topicName: Topic of the received message.
 * @param topicLen: Length of the topic.
 *       If 0 the value returned by strlen(topicName) can be trusted.
 *       If greater than 0, the topic name can be retrieved by accessing
 *       the variable as a byte array.
 * @param message: MQTTClient_message that contains the message payload
 *       and attributes.
 *
 * @return : Integer value indicating whether or not the message has been
 *       safely received by the client application.
 *       Return 1 means that the message has been successfully handled.
 *       Return 0 indicates that there was a problem.
 */
int msgarrvd(void *context, char *topicName, int topicLen,
    MQTTClient_message *message);

/**
 * Function that process evm messages previously received.
 */
void process_evm();

/**
 * Function that process gid messages previously received.
 */
void process_gid();

/**
 * Callback function to enable asynchronous notification of the loss of
 * connection to the server.
 * The function is called when the connection is lost to the server.
 *
 * @param context:  Pointer to the context value passed to
 *       MQTTClient_setCallbacks(), which contains the
 *       application-specific context.
 * @param cause: Reason for the disconnection.
 */
void connlost(void *context, char *cause);


/**
 * Function that establish a connection between a client and
 *       the broker to allow publish/subscribe messages.
 *
 * @param id: String identifier of a vehicle, must be unique.
 *
 */
void set_connection(char* id);

/**
 * Function that close a connection between a client and the broker
 */
void close_connection();

/**
 * Function that calls the function MQTTClient_subscribe()
 *       to subscribe a client to a certain topic indicated
 *       in the parameters.
 *
 * @param stopic:  Name of the topic to be subscribed.
 */
void subscribe_to_topic(char *stopic);

/**
 * Function that calls the function MQTTClient_unsubscribe()
 *       to unsubscribe a client from a certain topic indicated
 *       in the parameters.
 *
 * @param stopic:  Name of the topic to be subscribed.
 */
void unsubscribe_to_topic(char *stopic);

/**
 * Function that creates a message of type "CIN".
 *
 * @param msg: Structure of type CIN_struct with the content of
 *       the message to be sent.
 */
void create_cin_msg(CIN_struct msg);

/**
 * Function that creates a message of type "EVM".
 *
 * @param ev: Value that indicates the type of message to be send.
 * @param msg: Structure of type EVM_struct with the content
 *       of the message to be sent.
 */
void create_evm_msg(cdm ev, EVM_struct msg);

/**
 * Function that creates a message of type "GID".
 *
 * @param ev: Value that indicates the type of message to be send.
 * @param msg: Structure of type GID_struct with the content
 *       of the message to be sent.
 */
void create_gid_msg(cdm ev, GID_struct msg);

/**
 * Funtion that publish a new message in a certin topic.
 *
 * @param ptopic:  Name of the topic where to publish.
 */
void publish_msg(char* ptopic);
```

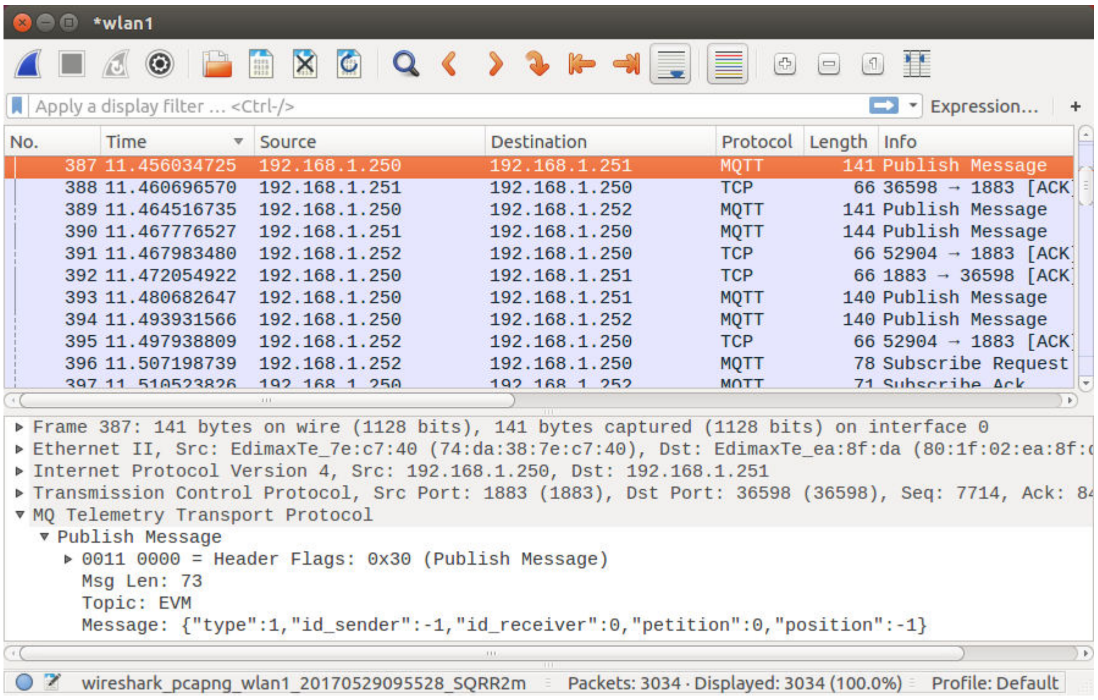
Listing A.5: C file that implements the leader's vCPS code

```
#include "v2v_comm.h"
#include "coroutine.h"
#include "brick.h"

pthread_t t_info, t_event, t_color_sensors;
```

```
char* mqtt_id = "robot_1";

char publish_topic[16];
char subscribe_topic[16];

CIN_struct s_inf;

//Sends a message every second
void* send_info(void *arg){
    //Publish in your INFO topic
    snprintf(publish_topic, sizeof(publish_topic), "CIN_%d",
    get_platoon_id());
    while(true){
        create_cin_msg(s_inf);
        publish_msg(publish_topic);
        sleep_ms(1000);  // 1 second
    }
}

void* check_events(void *args){
    while(true){
        if(evm_to_process()){
            process_evm();
        }
        if(gid_to_process()){
            process_gid();
        }
    }
}

void* drive_straight(void * args){
    int left_sensor = -1, right_sensor = -1, angle;
    while(s_inf.speed!=0){
        left_sensor = sensor_get_value(COLOR_CHANNEL,
        port_to_socket(COLOR_SENSOR_LEFT), IR_REMOTE__NONE_);
        right_sensor = sensor_get_value(COLOR_CHANNEL,
        port_to_socket(COLOR_SENSOR_RIGHT), IR_REMOTE__NONE_);

        //Left color sensor doesn't see black - Need to turn right
        if ((left_sensor != 1) && (right_sensor == 1)){
            do{
                angle=-_SNC_TURN_ANGLE_;

                tacho_set_speed_sp(MOTOR_LEFT, s_inf.speed);
                tacho_set_speed_sp(MOTOR_RIGHT, s_inf.speed);
                tacho_set_position_sp(MOTOR_LEFT, DEGREE_TO_COUNT(-angle));
                tacho_set_position_sp(MOTOR_RIGHT, DEGREE_TO_COUNT(angle));
                tacho_run_to_rel_pos(MOTOR_BOTH);
                tacho_run_forever(MOTOR_BOTH);

                left_sensor = sensor_get_value(COLOR_CHANNEL,
                port_to_socket(COLOR_SENSOR_LEFT), IR_REMOTE__NONE_);
            }while(left_sensor!=1);
        }
        //Right color sensor doesnt see black - Need to turn left
        else if ((right_sensor != 1) && (left_sensor == 1)){
            do{
                angle=_SNC_TURN_ANGLE_;

                tacho_set_speed_sp(MOTOR_LEFT, s_inf.speed);
                tacho_set_speed_sp(MOTOR_RIGHT, s_inf.speed);
                tacho_set_position_sp(MOTOR_LEFT, DEGREE_TO_COUNT(-angle));
                tacho_set_position_sp(MOTOR_RIGHT, DEGREE_TO_COUNT(angle));
                tacho_run_to_rel_pos(MOTOR_BOTH);
                tacho_run_forever(MOTOR_BOTH);

                right_sensor = sensor_get_value(COLOR_CHANNEL,
                port_to_socket(COLOR_SENSOR_RIGHT), IR_REMOTE__NONE_);
            }while(right_sensor!=1);
        }
    }
    tacho_stop(MOTOR_BOTH);
}

int main(void){
    char error;
    s_inf.speed = -1;
    set_platoon_id(0);
    set_connection(mqtt_id); //Initialize MQTT communication
    brick_init(); //Initialize robot sensors and motors

    //Subscribe in the previous vehicle EVENT topic
    snprintf(subscribe_topic, sizeof(subscribe_topic), "EVM");
    subscribe_to_topic(subscribe_topic);

    //Create thread to send INFO messages
    error = pthread_create(&t_info, NULL, &send_info, NULL);
    if(error != 0){
        printf("Thread not created: %s\n", strerror(error));
    }
    //Create thread to process events
    error = pthread_create(&t_event, NULL, &check_events, NULL);
    if(error != 0){
        printf("Thread not created: %s\n", strerror(error));
    }
    //Create thread to handle color sensors making the robot drive straight
    if(sensor_is_plugged(port_to_socket (COLOR_SENSOR_LEFT),
    LEGO_EV3_COLOR) && sensor_is_plugged(port_to_socket(
    COLOR_SENSOR_RIGHT), LEGO_EV3_COLOR)){
        color_set_mode_col_color(port_to_socket(COLOR_SENSOR_LEFT));
        color_set_mode_col_color(port_to_socket(COLOR_SENSOR_RIGHT));
        error = pthread_create(&t_color_sensors, NULL, &drive_straight,
        NULL);
        if(error != 0){
            printf("Thread not created: %s\n", strerror(error));
        }
    }
    else{
        printf("Color sensors NOT plugged\n" );
    }
    //Run motors
    if (tacho_is_plugged(MOTOR_BOTH, LEGO_EV3_L_MOTOR)) {
        s_inf.speed = MAX_SPEED * 0.10;
        // 10% of MAX_SPEED
        tacho_set_speed_sp(MOTOR_BOTH, s_inf.speed);
        tacho_run_forever(MOTOR_BOTH);
        sleep_ms(20000);  // 20 sec
        s_inf.speed = MAX_SPEED * 0.20;
        // 20% of MAX_SPEED
        tacho_set_speed_sp( OUTA | OUTB, s_inf.speed);
        tacho_run_forever(MOTOR_BOTH);
        sleep_ms(20000);  // 20 sec
        s_inf.speed = MAX_SPEED * 0.30;
        // 30% of MAX_SPEED
        tacho_set_speed_sp( OUTA | OUTB, s_inf.speed);
        tacho_run_forever(MOTOR_BOTH);
        sleep_ms(20000);  // 20 sec
        s_inf.speed = 0;
        tacho_stop(MOTOR_BOTH);
    }
    else{
        printf("Motors NOT plugged\n");
    }

    brick_uninit(); //Deactivate robot sensors, motors, etc.
    sleep_ms(1000);
    close_connection(); //Close MQTT connection
    return 0;
}
```

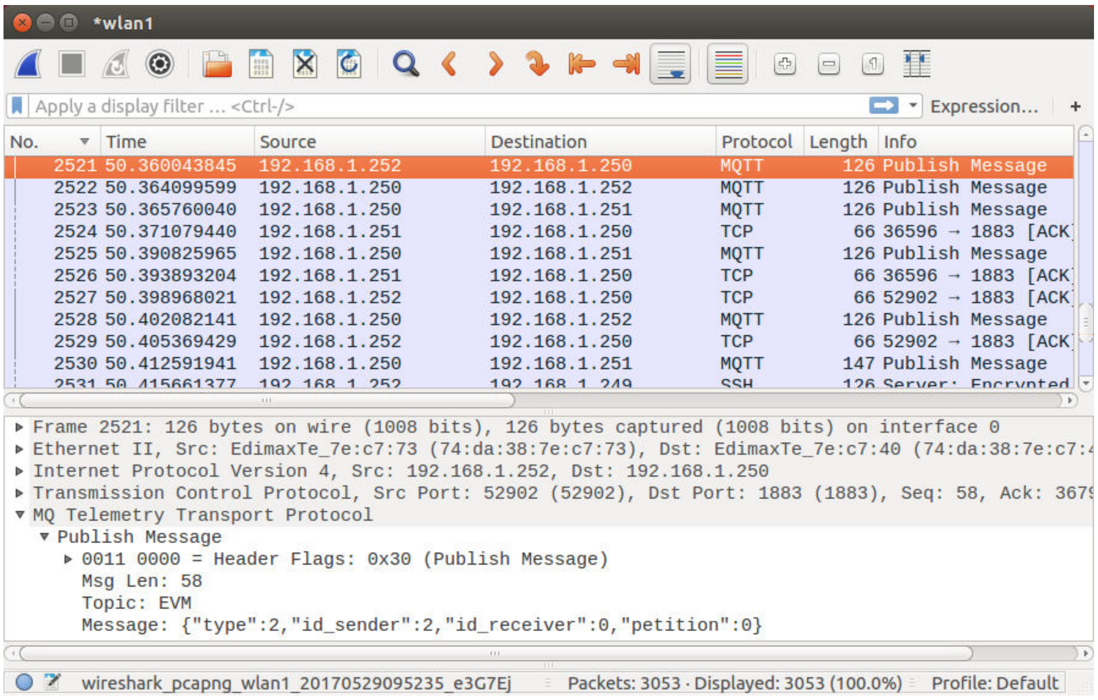
A.4 Retransmission analysis

This section contains screenshots of the network analyser Wireshark, in order to check if the packets required to achieve a maneuver are retransmitted or not by the TCP protocol. This has been done in order to calculate the $failure\_rate_{maneuver}$ metric. But, as it can be seen there is no TCP retransmision of the messages.



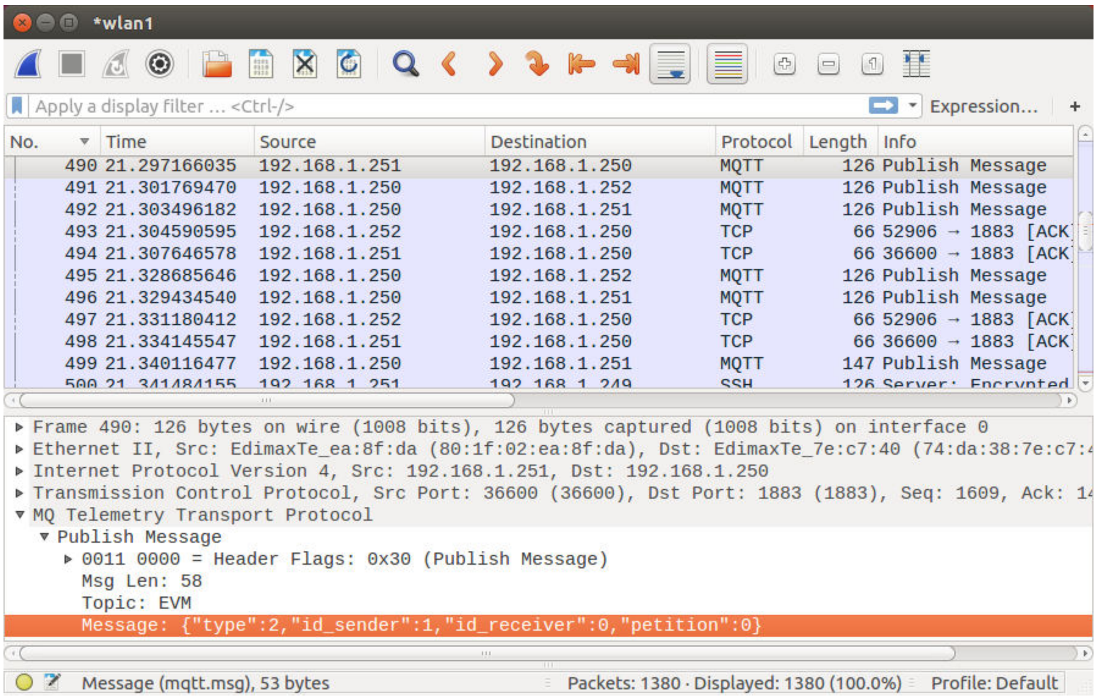
```
*wlan1
Apply a display filter ... <Ctrl-/>                                   Expression...  +
No.   Time          Source          Destination     Protocol Length Info
  387 11.456034725  192.168.1.250   192.168.1.251   MQTT      141 Publish Message
  388 11.460696570  192.168.1.251   192.168.1.250   TCP        66 36598 → 1883 [ACK]
  389 11.464516735  192.168.1.250   192.168.1.252   MQTT      141 Publish Message
  390 11.467776527  192.168.1.251   192.168.1.250   MQTT      144 Publish Message
  391 11.467983480  192.168.1.252   192.168.1.250   TCP        66 52904 → 1883 [ACK]
  392 11.472054922  192.168.1.250   192.168.1.251   TCP        66 1883 → 36598 [ACK]
  393 11.480682647  192.168.1.250   192.168.1.251   MQTT      140 Publish Message
  394 11.493931566  192.168.1.250   192.168.1.252   MQTT      140 Publish Message
  395 11.497938809  192.168.1.252   192.168.1.250   TCP        66 52904 → 1883 [ACK]
  396 11.507198739  192.168.1.252   192.168.1.250   MQTT       78 Subscribe Request
  397 11.510523826  192.168.1.250   192.168.1.252   MQTT       71 Subscribe Ack

▸ Frame 387: 141 bytes on wire (1128 bits), 141 bytes captured (1128 bits) on interface 0
▸ Ethernet II, Src: EdimaxTe_7e:c7:40 (74:da:38:7e:c7:40), Dst: EdimaxTe_ea:8f:da (80:1f:02:ea:8f:d
▸ Internet Protocol Version 4, Src: 192.168.1.250, Dst: 192.168.1.251
▸ Transmission Control Protocol, Src Port: 1883 (1883), Dst Port: 36598 (36598), Seq: 7714, Ack: 84
▾ MQ Telemetry Transport Protocol
  ▾ Publish Message
    ▸ 0011 0000 = Header Flags: 0x30 (Publish Message)
      Msg Len: 73
      Topic: EVM
      Message: {"type":1,"id_sender":-1,"id_receiver":0,"petition":0,"position":-1}

wireshark_pcapng_wlan1_20170529095528_SQRR2m   Packets: 3034 · Displayed: 3034 (100.0%)   Profile: Default
```

Figure A.2: Wireshark output to control retransmission of packets in a join back maneuver



```
*wlan1
Apply a display filter ... <Ctrl-/>                                   Expression...  +
No.   Time          Source          Destination     Protocol Length Info
 2521 50.360043845  192.168.1.252   192.168.1.250   MQTT      126 Publish Message
 2522 50.364099599  192.168.1.250   192.168.1.252   MQTT      126 Publish Message
 2523 50.365760040  192.168.1.250   192.168.1.251   MQTT      126 Publish Message
 2524 50.371079440  192.168.1.251   192.168.1.250   TCP        66 36596 → 1883 [ACK]
 2525 50.390825965  192.168.1.250   192.168.1.251   MQTT      126 Publish Message
 2526 50.393893204  192.168.1.251   192.168.1.250   TCP        66 36596 → 1883 [ACK]
 2527 50.398968021  192.168.1.252   192.168.1.250   TCP        66 52902 → 1883 [ACK]
 2528 50.402082141  192.168.1.250   192.168.1.252   MQTT      126 Publish Message
 2529 50.405369429  192.168.1.252   192.168.1.250   TCP        66 52902 → 1883 [ACK]
 2530 50.412591941  192.168.1.250   192.168.1.251   MQTT      147 Publish Message
 2531 50.415661377  192.168.1.252   192.168.1.249   SSH       126 Server: Encrypted

▸ Frame 2521: 126 bytes on wire (1008 bits), 126 bytes captured (1008 bits) on interface 0
▸ Ethernet II, Src: EdimaxTe_7e:c7:73 (74:da:38:7e:c7:73), Dst: EdimaxTe_7e:c7:40 (74:da:38:7e:c7:4
▸ Internet Protocol Version 4, Src: 192.168.1.252, Dst: 192.168.1.250
▸ Transmission Control Protocol, Src Port: 52902 (52902), Dst Port: 1883 (1883), Seq: 58, Ack: 3679
▾ MQ Telemetry Transport Protocol
  ▾ Publish Message
    ▸ 0011 0000 = Header Flags: 0x30 (Publish Message)
      Msg Len: 58
      Topic: EVM
      Message: {"type":2,"id_sender":2,"id_receiver":0,"petition":0}

wireshark_pcapng_wlan1_20170529095235_e3G7Ej   Packets: 3053 · Displayed: 3053 (100.0%)   Profile: Default
```

Figure A.3: Wireshark output to control retransmission of packets in a leave back maneuver

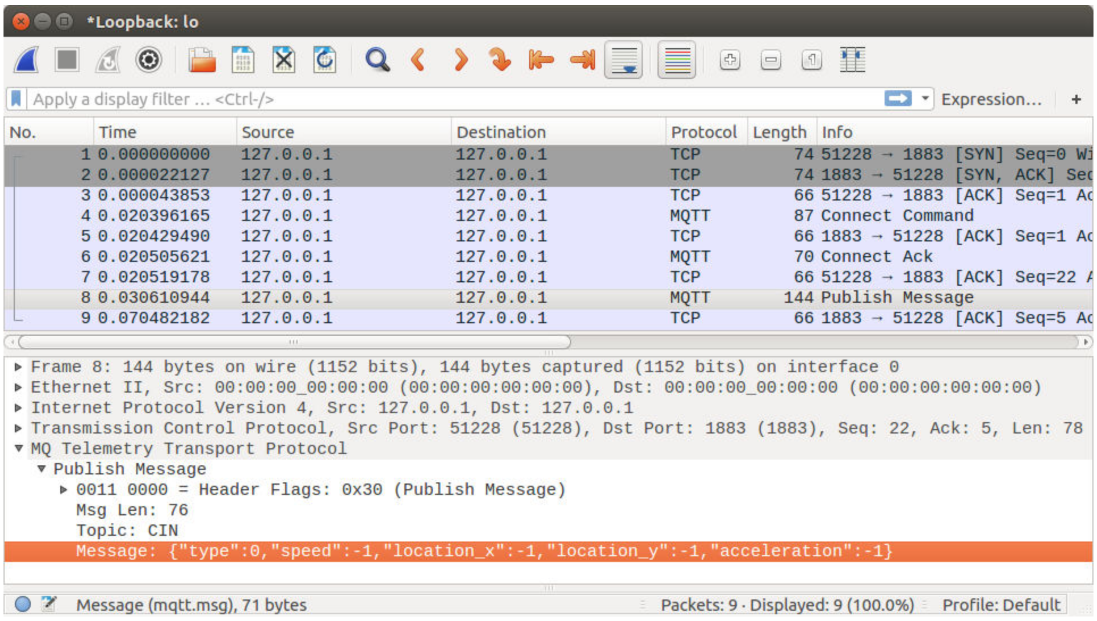


```
*wlan1
Apply a display filter ... <Ctrl-/>                                            Expression...  +
No.    Time            Source           Destination      Protocol Length Info
490 21.297166035 192.168.1.251 192.168.1.250 MQTT      126 Publish Message
491 21.301769470 192.168.1.250 192.168.1.252 MQTT      126 Publish Message
492 21.303496182 192.168.1.250 192.168.1.251 MQTT      126 Publish Message
493 21.304590595 192.168.1.252 192.168.1.250 TCP        66 52906 → 1883 [ACK]
494 21.307646578 192.168.1.251 192.168.1.250 TCP        66 36600 → 1883 [ACK]
495 21.328685646 192.168.1.250 192.168.1.252 MQTT      126 Publish Message
496 21.329434540 192.168.1.250 192.168.1.251 MQTT      126 Publish Message
497 21.331180412 192.168.1.252 192.168.1.250 TCP        66 52906 → 1883 [ACK]
498 21.334145547 192.168.1.251 192.168.1.250 TCP        66 36600 → 1883 [ACK]
499 21.340116477 192.168.1.250 192.168.1.251 MQTT      147 Publish Message
500 21.341484155 192.168.1.251 192.168.1.249 SSH       126 Server: Encrypted

▸ Frame 490: 126 bytes on wire (1008 bits), 126 bytes captured (1008 bits) on interface 0
▸ Ethernet II, Src: EdimaxTe_ea:8f:da (80:1f:02:ea:8f:da), Dst: EdimaxTe_7e:c7:40 (74:da:38:7e:c7:4
▸ Internet Protocol Version 4, Src: 192.168.1.251, Dst: 192.168.1.250
▸ Transmission Control Protocol, Src Port: 36600 (36600), Dst Port: 1883 (1883), Seq: 1609, Ack: 14
▾ MQ Telemetry Transport Protocol
  ▾ Publish Message
    ▸ 0011 0000 = Header Flags: 0x30 (Publish Message)
      Msg Len: 58
      Topic: EVM
      Message: {"type":2,"id_sender":1,"id_receiver":0,"petition":0}

Message (mqtt.msg), 53 bytes        Packets: 1380 · Displayed: 1380 (100.0%)    Profile: Default
```

Figure A.4: Wireshark output to control retransmission of packets in a leave middle maneuver

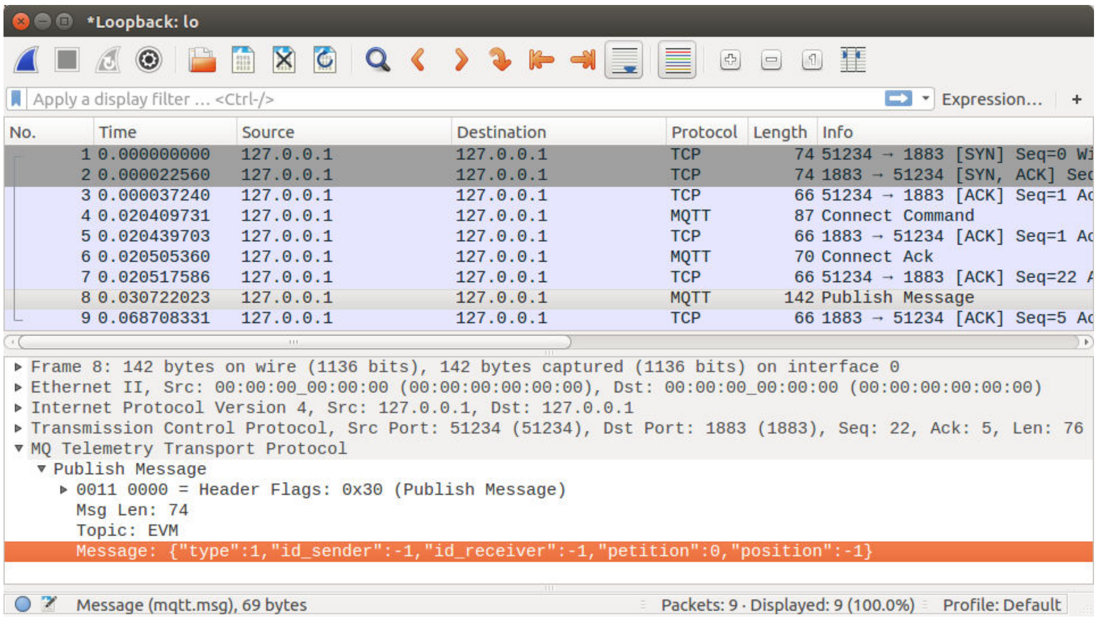
A.5 Packet size measurement

The network protocol analyser Wireshark is used to provide the size of the different packets used in the implementation. Following there are screenshots of the Wireshark outputs over the execution of the code in listing A.6 that defines a code sending CDM messages over MQTT to measure their size.



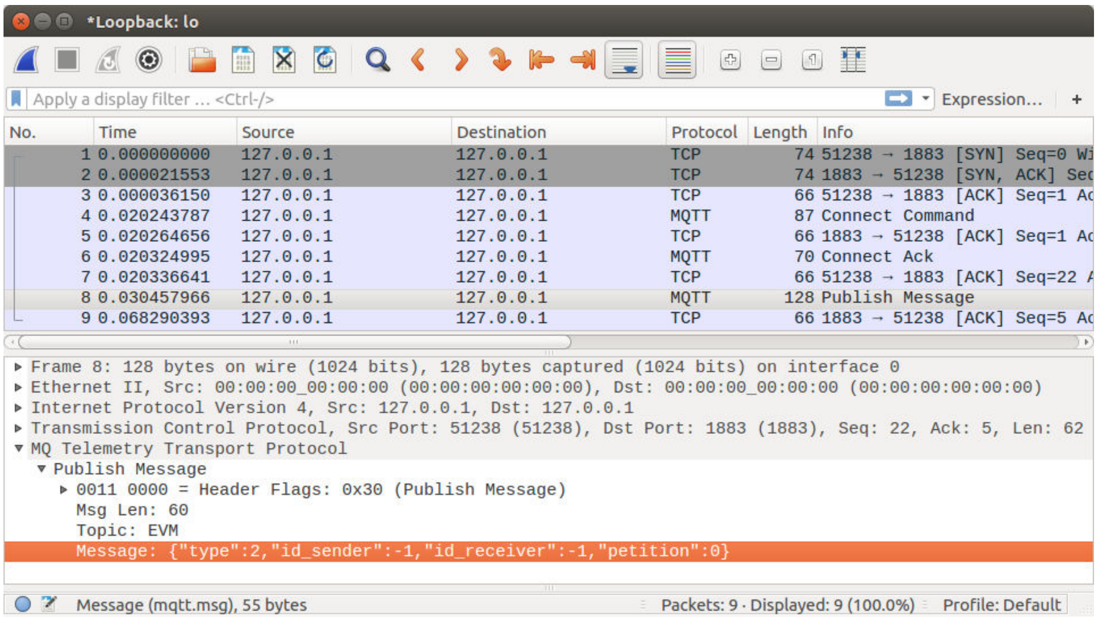
```
*Loopback: lo
Apply a display filter ... <Ctrl-/>                                            Expression...  +
No.  Time         Source      Destination  Protocol Length Info
  1 0.000000000 127.0.0.1   127.0.0.1    TCP        74 51228 → 1883 [SYN] Seq=0 Wi
  2 0.000022127 127.0.0.1   127.0.0.1    TCP        74 1883 → 51228 [SYN, ACK] Sec
  3 0.000043853 127.0.0.1   127.0.0.1    TCP        66 51228 → 1883 [ACK] Seq=1 Ac
  4 0.020396165 127.0.0.1   127.0.0.1    MQTT       87 Connect Command
  5 0.020429490 127.0.0.1   127.0.0.1    TCP        66 1883 → 51228 [ACK] Seq=1 Ac
  6 0.020505621 127.0.0.1   127.0.0.1    MQTT       70 Connect Ack
  7 0.020519178 127.0.0.1   127.0.0.1    TCP        66 51228 → 1883 [ACK] Seq=22 A
  8 0.030610944 127.0.0.1   127.0.0.1    MQTT      144 Publish Message
  9 0.070482182 127.0.0.1   127.0.0.1    TCP        66 1883 → 51228 [ACK] Seq=5 Ac

▸ Frame 8: 144 bytes on wire (1152 bits), 144 bytes captured (1152 bits) on interface 0
▸ Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst: 00:00:00_00:00:00 (00:00:00:00:00:00)
▸ Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
▸ Transmission Control Protocol, Src Port: 51228 (51228), Dst Port: 1883 (1883), Seq: 22, Ack: 5, Len: 78
▾ MQ Telemetry Transport Protocol
  ▾ Publish Message
    ▸ 0011 0000 = Header Flags: 0x30 (Publish Message)
      Msg Len: 76
      Topic: CIN
      Message: {"type":0,"speed":-1,"location_x":-1,"location_y":-1,"acceleration":-1}

Message (mqtt.msg), 71 bytes        Packets: 9 · Displayed: 9 (100.0%)    Profile: Default
```

Figure A.5: Output of Wireshark for Context Information (CIN) message



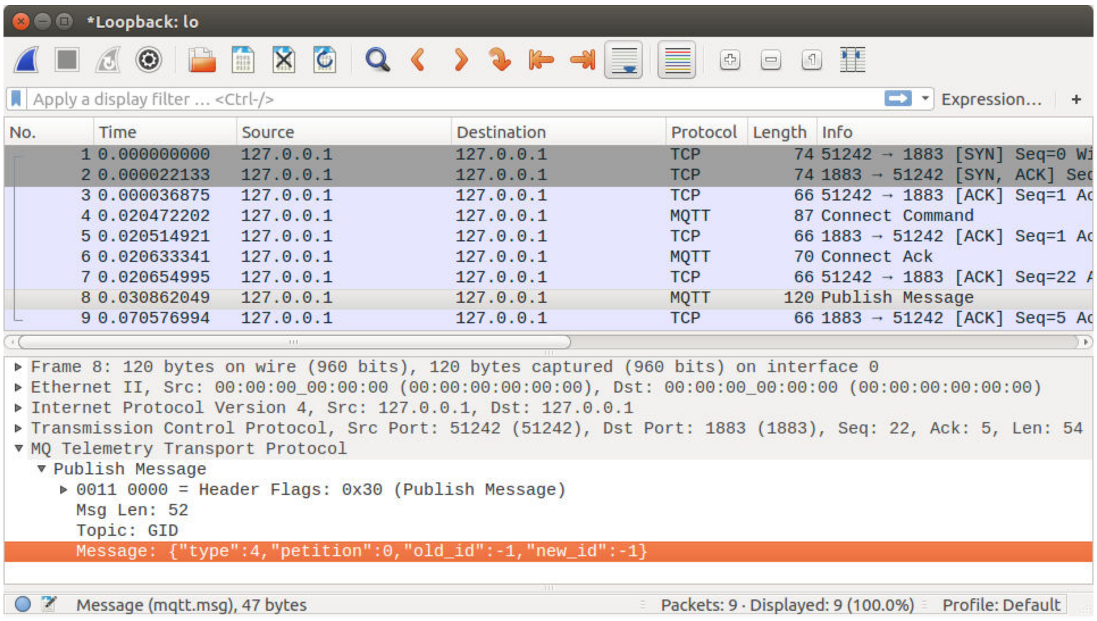
```
No.   Time          Source      Destination  Protocol Length Info
    1 0.000000000   127.0.0.1   127.0.0.1    TCP        74 51234 → 1883 [SYN] Seq=0 Wi
    2 0.000022560   127.0.0.1   127.0.0.1    TCP        74 1883 → 51234 [SYN, ACK] Sec
    3 0.000037240   127.0.0.1   127.0.0.1    TCP        66 51234 → 1883 [ACK] Seq=1 Ac
    4 0.020409731   127.0.0.1   127.0.0.1    MQTT       87 Connect Command
    5 0.020439703   127.0.0.1   127.0.0.1    TCP        66 1883 → 51234 [ACK] Seq=1 Ac
    6 0.020505360   127.0.0.1   127.0.0.1    MQTT       70 Connect Ack
    7 0.020517586   127.0.0.1   127.0.0.1    TCP        66 51234 → 1883 [ACK] Seq=22 A
    8 0.030722023   127.0.0.1   127.0.0.1    MQTT      142 Publish Message
    9 0.068708331   127.0.0.1   127.0.0.1    TCP        66 1883 → 51234 [ACK] Seq=5 Ac

▸ Frame 8: 142 bytes on wire (1136 bits), 142 bytes captured (1136 bits) on interface 0
▸ Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst: 00:00:00_00:00:00 (00:00:00:00:00:00)
▸ Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
▸ Transmission Control Protocol, Src Port: 51234 (51234), Dst Port: 1883 (1883), Seq: 22, Ack: 5, Len: 76
▾ MQ Telemetry Transport Protocol
  ▾ Publish Message
    ▸ 0011 0000 = Header Flags: 0x30 (Publish Message)
      Msg Len: 74
      Topic: EVM
      Message: {"type":1,"id_sender":-1,"id_receiver":-1,"petition":0,"position":-1}

Message (mqtt.msg), 69 bytes        Packets: 9 · Displayed: 9 (100.0%)    Profile: Default
```

Figure A.6: Output of Wireshark for Event Maneuver (EVM) message of type JOIN



```
No.   Time          Source      Destination  Protocol Length Info
    1 0.000000000   127.0.0.1   127.0.0.1    TCP        74 51238 → 1883 [SYN] Seq=0 Wi
    2 0.000021553   127.0.0.1   127.0.0.1    TCP        74 1883 → 51238 [SYN, ACK] Sec
    3 0.000036150   127.0.0.1   127.0.0.1    TCP        66 51238 → 1883 [ACK] Seq=1 Ac
    4 0.020243787   127.0.0.1   127.0.0.1    MQTT       87 Connect Command
    5 0.020264656   127.0.0.1   127.0.0.1    TCP        66 1883 → 51238 [ACK] Seq=1 Ac
    6 0.020324995   127.0.0.1   127.0.0.1    MQTT       70 Connect Ack
    7 0.020336641   127.0.0.1   127.0.0.1    TCP        66 51238 → 1883 [ACK] Seq=22 A
    8 0.030457966   127.0.0.1   127.0.0.1    MQTT      128 Publish Message
    9 0.068290393   127.0.0.1   127.0.0.1    TCP        66 1883 → 51238 [ACK] Seq=5 Ac

▸ Frame 8: 128 bytes on wire (1024 bits), 128 bytes captured (1024 bits) on interface 0
▸ Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst: 00:00:00_00:00:00 (00:00:00:00:00:00)
▸ Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
▸ Transmission Control Protocol, Src Port: 51238 (51238), Dst Port: 1883 (1883), Seq: 22, Ack: 5, Len: 62
▾ MQ Telemetry Transport Protocol
  ▾ Publish Message
    ▸ 0011 0000 = Header Flags: 0x30 (Publish Message)
      Msg Len: 60
      Topic: EVM
      Message: {"type":2,"id_sender":-1,"id_receiver":-1,"petition":0}

Message (mqtt.msg), 55 bytes        Packets: 9 · Displayed: 9 (100.0%)    Profile: Default
```

Figure A.7: Output of Wireshark for Event Maneuver (EVM) message of type LEAVE



```
No.  Time         Source     Destination  Protocol Length Info
   1 0.000000000  127.0.0.1  127.0.0.1    TCP          74 51242 → 1883 [SYN] Seq=0 Wi
   2 0.000022133  127.0.0.1  127.0.0.1    TCP          74 1883 → 51242 [SYN, ACK] Seq
   3 0.000036875  127.0.0.1  127.0.0.1    TCP          66 51242 → 1883 [ACK] Seq=1 Ac
   4 0.020472202  127.0.0.1  127.0.0.1    MQTT         87 Connect Command
   5 0.020514921  127.0.0.1  127.0.0.1    TCP          66 1883 → 51242 [ACK] Seq=1 Ac
   6 0.020633341  127.0.0.1  127.0.0.1    MQTT         70 Connect Ack
   7 0.020654995  127.0.0.1  127.0.0.1    TCP          66 51242 → 1883 [ACK] Seq=22 A
   8 0.030862049  127.0.0.1  127.0.0.1    MQTT        120 Publish Message
   9 0.070576994  127.0.0.1  127.0.0.1    TCP          66 1883 → 51242 [ACK] Seq=5 Ac
```

```
▸ Frame 8: 120 bytes on wire (960 bits), 120 bytes captured (960 bits) on interface 0
▸ Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst: 00:00:00_00:00:00 (00:00:00:00:00:00)
▸ Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
▸ Transmission Control Protocol, Src Port: 51242 (51242), Dst Port: 1883 (1883), Seq: 22, Ack: 5, Len: 54
▾ MQ Telemetry Transport Protocol
  ▾ Publish Message
    ▸ 0011 0000 = Header Flags: 0x30 (Publish Message)
      Msg Len: 52
      Topic: GID
      Message: {"type":4,"petition":0,"old_id":-1,"new_id":-1}
```

```
Message (mqtt.msg), 47 bytes          Packets: 9 · Displayed: 9 (100.0%)     Profile: Default
```

Figure A.8: Output of Wireshark for Group Identifier (GID) message of type ID

Listing A.6: Code sending every type of message to analyse in wireshark

```
#include "v2v_comm.h"

char* mqtt_id = "robot_1";

int main(void){
    set_connection(mqtt_id); //Initialize MQTT communication

    Info m_inf = {-1, -1, -1, -1};
    create_info_msg(m_inf);
    publish_msg("CIN");

    Event m_join = {JOIN, -1, -1, REQ, -1, -1};
    create_event_msg(JOIN, m_join);
    publish_msg("EVM");

    Event m_leave = {LEAVE, -1, -1, REQ, -1, -1};
    create_event_msg(LEAVE, m_leave);
    publish_msg("EVM");

    Id m_i = {ID, REQ, -1, -1};
    create_id_msg(m_i);
    publish_msg("GID");

    sleep(60); // 1 minute
    close_connection(); //Close MQTT connection
    return 0;
}
```