TÉCNICO
LISBOA

Formula Student Championship: an Integrated Data Processing Module Adapted for Pit-Stop Scenarios

João Pedro Passos Figueiras

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisors: Prof. Nuno Filipe Valentim Roma
Prof. Ricardo Jorge Fernandes Chaves

Examination Committee

Chairperson: Prof. João Emílio Segurado Pavão Martins
Supervisor: Prof. Nuno Filipe Valentim Roma
Member of the Committee: Prof. Renato Jorge Caleira Nunes

May 2014

Acknowledgments

Quero agradecer a todas as pessoas que de alguma forma contribuiram para que o sucesso deste trabalho.

À equipa da Formula Student do Instituto Superior Técnico, com especial atenção ao Pedro Oliveira, Bruno Santos, e Daniel Pinho, pelas orientações e contribuições dadas ao longo do desenvolvimento do trabalho.

Ao Diogo Carvalho, por me ter disponibilizado o seu trabalho juntamente com os trabalhos do Paulo Mendes e do David Copeto, e pela disponibilidade de auxiliar na minha integração no projecto e esclarecer dúvidas.

Aos professores Nuno Roma e Ricardo Chaves pela oportunidade de trabalhar nesta tese, orientação dada ao longo do trabalho, e grau de exigência tido que muito contribuiu para o meu desenvolvimento.

À minha familia e amigos, que ao longo do curso sempre me apoiaram e motivaram, e claro, à Joana, a minha maior fonte de motivação.

Abstract

This document describes the proposed solution for a software framework and integrated user interface to be deployed in the existing telemetry system installed in the vehicle prototype of Instituto Superior Técnico (IST) team that competes in the Formula Student Championship. The solution has two deployment devices, one located in the racing vehicle (referred to as mobile station), and the other located in the pit stop (referred to as pit station). The ultimate goal is to enable the data sent from the mobile station to be stored and displayed to users in the pit station either in real-time or at a later time. Data is acquired from several sensors installed in the car and interconnected by a Controller Area Network (CAN) bus network. This project consists essentially on the evolution of a previous developed software framework to better cater the end user needs. The work comprises important improvements on several components, such as the communication protocol used between the two stations, the database used to store all the information gathered, the required processing over the gathered data, and the adaptation of a graphical user interface to better fit the user profiles and requisites.

Keywords

Formula Student Championship, Graphical User Interface, Database Management System, Communication Protocol, Management/Processing of sensor data

Resumo

Este documento descreve a solução proposta para uma aplicação de software e correspondente interface do utilizador, com vista a sua integração no sistema de telemetria instalado no veiculo da equipa do IST que compete no campeonato Formula Student. A solução é constituída por dois módulos autónomos, um localizado no veiculo (referido como módulo móvel), e outro localizado na linha de meta ou boxes (referido como módulo fixo). O principal objectivo é permitir que os dados enviados da módulo móvel sejam guardados e exibidos aos utilizadores no módulo fixo, seja em tempo real ou numa altura posterior. Os dados são adquiridos de vários sensores instalados no carro e interligados por uma rede baseada no barramento CAN. Este projecto consiste essencialmente numa evolução de uma aplicação de software desenvolvida anteriormente, para melhor atender ás necessidades dos utilizadores. O trabalho inclui melhoramentos em diversos componentes, como o protocolo de comunicação usado entre os dois módulos, a base de dados usada para guardar a informação recebida, o processamento requerido sobre os dados recebidos, e a adaptação da interface gráfica do utilizador para melhor se ajustar aos perfis de utilizadores e seus requisitos.

Palavras Chave

Campeonato Formula Student, Interface Gráfica do Utilizador, Sistema de Gestão da Base de Dados, Protocolo de Comunicação, Gestão/Processamento de dados de sensores

Contents

List of Figures

List of Tables

List of Acronyms

ALMS American Le Mans Series

API Application Programming Interface

CAN Controller Area Network

CSV Comma-Separated Values

DBMS Database Management System

DLL Dynamic-link Library

ECU Electronic Control Unit

FIA Federation Internationale de l'Automobile

FST Formula Student Team

GPS Global Positioning System

GUI Graphical User Interface

GCC GNU Compiler Collection

IDE Integrated Development Environment

IST Instituto Superior Técnico

JIT Just-In-Time

JPEG Joint Photographic Experts Group

KML Keyhole Markup Language

MSVC Microsoft Visual C++

MTU Maximum Transmission Unit

NASCAR National Association for Stock Car Auto Racing

PDF Portable Document Format

POSIX Portable Operating System Interface

RPM Revolutions Per Minute

SQL Structured Query Language

TCP Transmission Control Protocol

UDP User Datagram Protocol

USB Universal Serial Bus

XML eXtensible Markup Language

1

Introduction

Contents

1.1 Scope

The origins of the student motor-sport competition dates back to 1981 when the Formula SAE programme started in the United states. In order for a team to participate, it had to develop a formula-style race car, which would be evaluated according to its potential as a production item [14]. Later, in 1998, the Formula Student Championship [4] was born in the UK, under the jurisdiction of the Institution of Mechanical Engineers. The goal of this competition is to give the chance for engineering students to demonstrate their technical, engineering design, and manufacturing skills, while developing other capabilities such as team working, time management, among others. Each team is challenged to conceive, design, build, cost, present and compete with a prototype of a single-seat racing car in a series of static and dynamic events. These challenges are used for students to demonstrate their understanding and test the performance of the vehicle, while being judged by experienced industry specialists. Just like any other competition, there are factors that influences the overall result of a team. Among these are the evaluation of the built prototype in terms of its cost (which must be low), ease to be maintained, reliability, and performance in terms of acceleration, braking and handling qualities.

In 2001, a project (denominated by Formula Student Team (FST)) [5] was initiated by a group of students from Instituto Superior Técnico (IST) with the aim of joining the Formula Student Competition. They started to compete a year later with their first produced prototype. Since its foundation, the team has already built five prototype vehicles. The first three have traditional combustion engines. The fourth started to follow the trends for electric vehicles, therefore having an electrical engine, and the fifth continued to follow this trends, thus also being equipped with an electrical engine. The team is composed of several students from diverse IST courses such as Mechanical, Electronics, Aerospace and Informatics Engineering, and is supported by several professors. The latest prototype produced by the team, called FST05e, is depicted in Figure 1.1.

Figure 1.1: The fifth prototype produced by the FST team (FST05e) [5].

To provide students with the means for an early and real-time problem detection on the vehicle, and consequent ability to make improvements or adjustments during testing sessions, a series of projects have taken place. These projects mainly consisted in creating the hardware infrastructure and the software application that would make this type of analysis possible. As such, the existent work counts with a fully featured telemetry system installed in the vehicle, which is composed by an embedded-computer connected to a Controller Area Network (CAN) bus and a Global Positioning System (GPS) receiver. This system collects data from several sensors such as: suspension displacement, engine Revolutions Per Minute (RPM), tire pressure, among others. This set of sensors is interconnected to the embedded-computer by the CAN, used to collect the data. This data is subsequently used for analysis by a software application in the pits, which receives it by means of a wireless network connection. The work presented in this thesis, is a follow-up to a series of previous works related to the software that enables the analysis of sensor data, gathered by the telemetry system installed in the vehicle and sent over the wireless network connection to the pits [6–8].

1.2 Motivation

It is very important to have well defined system requirements when developing any system. The degree of satisfaction of the users with a system, is directly related to the correct catering of their needs. The previous framework was the result of a series of independent works focused on some particular characteristic to be added to the system, and not an attempt to build a single tool for racing analysis tailored to the users. Therefore, it lacked a formal definition of an integrated set of system requirements which was very important to enable a useful data analysis from the users. As a consequence, it was necessary to gather these requirements and determine the user needs and conditions to meet. The result of this analysis was the integration of some previous existing modules improved and adapted for new needs, with new ones that were defined to meet new functionalities not yet introduced in the system. To cope with the collected requirements, several enhancements had to be introduced. In the communication module to make it independent of the adopted operating system and to change the structure of the transmitted network messages, in order to give more flexibility in changing their content. A new configuration module had to be developed to allow the users to manage the sensors used in their analysis. It was agreed that this module would let the users define sensors that were installed in the car and respective signal processing formulas, but also define a type of sensor that would represent a mathematical manipulation over the sensors installed in the car. It was also decided to introduce improvements on the Database Management System (DBMS), namely by changing it, in order to increase the speed performance for storing the acquired data, in a multi-transactional environment. Besides this, it was also necessary to define the interface between the system and the users, to promote

a familiar working environment where they can work at their best. Therefore, a redefinition of the Graphical User Interface (GUI) was also required, in order to provide the users with a comfortable interface for them to perform any data analysis.

Hence, the presented work constitutes the definition of an integrated framework, which defines a new distributed system that caters the user needs for a racing analysis tool. The new framework integrates some of the existing functionalities, with the new created modules in a single tool, together with a dedicated GUI. As a consequence, the solution implies deployment in both the mobile and pit station. The former is responsible for acquiring sensor data and sending it to the pit station, so the necessary modifications lied in the adaptation of the communication protocol, to be used when transmitting messages. The latest is responsible for receiving the sensor data, process it and store it. Therefore the modifications in the pit station are more emphasized, with adaptations in the communication module, the change of the DBMS, definition of the new information management module, and the redefinition of the GUI.

In the whole, this work represents a significant added-value to the work performed by the IST FST team, in the sense that it will enable the team to detect failures in the car and perform the necessary adjustments, in an easier, faster and more flexible way. It also represents an added-value to the data gathered by the car sensors, since the integrated management, manipulation and processing of data, enables the users to have their analysis environment configured to match their preferences. As an example, the users may have ten sensors installed in the car, but at a specific moment they are only interested in seeing information from five in an analysis session. The system enables the users to set up the five needed sensors in the application, while the information from the other sensors is disregarded and not displayed. Another example is for instance, manipulate data coming from the electric current sensor and from the battery voltage sensor, particularly by multiplying one for the other, and get the value corresponding to the power requested by the motor to the battery at a particular instant.

1.3 Objectives

As referred before, this project constitutes an evolution of a previous version of the framework already developed for the IST FST team, which can only be used for a limited and static set of analysis. Therefore, the project main goal is to overcome the existent limitations and make the software more useful for the users. To accomplish this, it is mainly focused on the development of a new and more sophisticated framework to be used in the pit station. This new version offers a more convenient and user-friendlier graphical user interface, new data processing and managing tools, and an improved reimplementation of the existent software modules. Hence, the end result of this thesis is a software framework that fully integrates the data acquisition and communication modules of the FST05e prototype vehicle with the user interface, while allowing a more

sophisticated manipulation of the gathered data. These improvements will provide the team with an easier way of detecting early signs of problems, providing the ability to warn the driver or to prepare the next adjustments to be made during the next pit stop. The main goals for this project can be enumerated as follows:

1. In order to allow the usage of the system in multiple operating systems, the implementation of the communication module will be adapted to be operating system independent.
2. Change the DBMS, in order to achieve better performances in terms of speed in a multi-transactional environment.
3. Development of a flexible sensor management system, providing the user with a friendly interface for adding or removing sensors from the desired analysis layout.
4. Integration of a mathematical engine, allowing users to introduce their own signal processing routines.
5. Development of new types of graphs and gauges, to allow the analysis of data in several kinds of views suited for specific purposes.
6. Integrate all components into a unique software framework, with a dedicated graphical user interface.

Regarding the two last goals, the requirements defined by the end users for the graphical user interface are:

- A single and unified graphical interface. In the previous version of this framework, there were two separated interfaces for two types of projects, one for the real-time data acquisition, and one for the offline data analysis. The end users required for the new version to have one single interface, applicable to both types of projects.
- User profile adaptable graphical layout. The layout of the graphical application should be adaptable to various types of user profiles since there are users concerned with the anaylisis of different areas of the vehicle (e.g: breaks, engine, suspension, etc.). This should be accomplished by having manageable tabs, in which several graphs can be introduced to display data from any sensor.
- Various types of graphs. The previous version lacks of graph veriety, giving only the possibility to have a time graph with information relating to single sensors, GPS track map, and a painting simulation of the team's vehicle. The end users required to have new set of graphs: time graphs, histograms and scatter plots (X-Y graphs where both axis are configurable by the user), and GPS track map;
- Various types of gauges. The previous version has three types of gauges, namely dials, bars, and steering wheels. The users required to extend this set of gauges to accomodate a greater variety. These include numeric lists, numeric gauges, and status lights;

- Graphical timeline: in the previous version there is no possible way to select a specific time span to be shown in the graph, neither is it possible to browse through time or distance in a graph. In the new version the end users required to have a graphical time/distance timeline mechanism where they would be able to select the time/distance span to be shown in a graph and browse through the graph data.

1.4 Main contributions

This section describes the main contributions of this work, as well as the most relevant characteristics for the end users.

1.4.1 Data Management

In order to have a working environment in which only the needed information by a specific analysis is taken into account, a flexible management context is required that allow the user to select what information should be loaded and presented. In this case, this means the currently relevant sensors for the users, together with the graphical elements in which their information is displayed. Hence, it is very important for the project to provide a context in which the user can select which information he wants to see. To accomplish this type of flexibility, dedicated graphical tools are provided for the users. Since many users, with possibly different profiles will be using the system, it is important to be able to load the information associated to each type of user when needed. The considered method to account for this aspect is to store this type of information in eXtensible Markup Language (XML) files:

- For the sensor information, when the user changes the list of sensors he is currently working with, a dedicated XML file containing this information is updated with the new list chosen by the user. This information, subsequently stored in a file, contains data such as the sensor name, id, type, and the number of bytes that must be read from network messages when receiving data for that particular sensor.
- For the information regarding the graphical elements used to display sensor data, another dedicated XML is used. The user can have several elements placed on the GUI, being responsible for deciding which ones are displayed. Unlike what is done for the information referred in the point above, when the user changes the graphical elements the information is not immediately stored. Instead, the user has to explicitly save the GUI state. For this particular type of information, there are dedicated files for each session in which the user has done some data processing. The data stored in these files contains information regarding the type of graphical element to be displayed, as well as the sensor or sensors associated with it.

1.4.2 Data Processing

It is very useful to be able to efficiently manipulate data from sensors. As an example, one may need to see several data records received for some set of sensors, which have to be all associated and displayed according to some mathematical equation. For this purpose, it is necessary to obtain an equal set of points for all sensors, composed by the intersection of all timestamps belonging to each sensor, and then interpolate the corresponding value for each timestamps. The mathematical equation should be evaluated after all of the interpolated values are obtained, and the result stored and displayed to the user.

To provide this functionality, the interface has a dedicated tool which allows to configure the mathematical manipulations over the existing set of sensors. The information about each new manipulation is then stored in the same XML file as the real sensors data, that are installed in the car (as mentioned in the previous section). With this approach, each data manipulation can be treated as a conventional sensor, but with the particularity that it is not installed in the vehicle, since it is the result of a combination of more than one sensors installed in the vehicle.

Another situation in which this type of processing is required is for one specific kind of plot, namely the *X-Y graph*, which associate a value of one sensor to a value of other sensor based on timestamps. For this type of plot, the same process of obtaining the set of timestamps described above is performed, as well as the interpolation of the obtained timestamps to get their corresponding values. Unlike the process described above, the evaluation of a mathematical equation is not needed in this case, as there are no such association between the sensors.

1.4.3 Time Line

Besides being useful for the users to see the overall information of a given analysis session, it is more useful for them if they can zoom in and out the time range of a session being displayed and browse through its overall time line. It is also helpful to select a specific point in time and check for the associated sensor information. The users should be able to select specific points in time by clicking on a graph and also select which information to be displayed based on shrinking and expanding the time range. When the time range is inferior to the overall time of a session, the user should be able to browse through its overall time. In this case the displayed information corresponds to the time range adjusted for the fraction of the session associated with the browse point selected.

1.4.4 Data Presentation

The management and processing of data is regarded as a very useful feature, but only if there is a GUI that makes it possible for the users to visualise the information in the most suitable way. For this purpose, some new graphical widgets were developed, each one being helpful for specific

purposes. There are two types of widgets that should be provided to the users: the *Graphs* and the *Gauges*. The *Graphs* provide the means for the user to visualize information in time (e.g to compare data from two sensors, check the frequencies of samples for a specific sensor, etc). The *Gauges* are similar to vehicle dash boards, and act as indicators for sensors which display information for a specific point in time. Besides this, the user also needs to know what is the time span of the information displayed in the widgets, as well as the data corresponding to a specific point in time that he has selected. This type of information is also displayed in the GUI, in order to assist the user to locate himself in time.

1.5 Dissertation outline

The content of this thesis is divided into six chapters, which are briefly described as:

- **Chapter 2 - Related Work and Supporting Technologies:** Starts by introducing both industrial and academic works, which are somehow related to this work, followed by describing all the software and hardware components that are part of the overall solution.
- **Chapter 3 - System Architecture:** Gives an overview of the proposed solution architecture, as well as a description of the structure of each deployment station present in the solution.
- **Chapter 4 - System Implementation:** Describes all software components that are part of the overall solution.
- **Chapter 5 - Results:** Presents the testing and evaluation environment, the tests that were performed and consequent results used to guarantee the accomplishment of the defined objectives.
- **Chapter 6 - Conclusions:** Presents a small reflection of the work done and the obtained results, as well as some ideas for future improvements on the new framework.

2

Related Work and Supporting Technologies

Contents

2.1 Summary

Telemetry has become an imperative technology in today's motorsports, acting as a competitive factor for racing teams to perform remote measurement on vehicles, but also to transmit instructions for drivers to change the way they are driving in order to achieve better performances. These remote measurements are transmitted via a wireless data communication link. Given the impact of this telemetry, some restrictions have been set for some competitions, such as Formula 1, to prohibit telemetry communication from pits to the cars [15].

This chapter starts by presenting the commercial and academic state of the art software solutions, which share the goal of providing the monitoring teams with the best service, in order for them to analyse data coming from the vehicles according to their needs. Following this, it introduces a set of technologies which support the implementation of these systems in some way.

2.2 Industrial/Commercial And Academic Solutions

Among the several existent commercial solutions for telemetry and data analysis, three are considered as the most important, and correspond to the ones developed by McLaren [12], by Cosworth [13], and by MoTeC [9]. This consideration comes from their usage in some of the most widely known competitions. Specifically, McLaren software is used on Formula 1, Cosworth software is used on MotoGP, and MoTeC software is used on Le Mans [9–11]. These three applications gather the most influencing technological aspects in this type of telemetry systems.

Among these three solutions, two are presented here with more detail, as they have more affinity with this project, namely the McLaren's software (used in Formula 1), and the MoTeC's software (used in Le Mans and also by the IST FST team).

2.2.1 McLaren

As the standard Electronic Control Unit (ECU) supplier chosen by the Federation Internationale de l'Automobile (FIA), McLaren plays a key role to all teams in Formula 1 today. Alongside the ECUs, a software system called Atlas [11] was developed with the goal of enabling teams to obtain, display, and analyse nearly real time data that is received from the car or to perform posterior analysis on logged data. Atlas allows users to access and visualise data in a diverse set of views, including Waveform, Circuit, Bar, Numeric, Scatter, Loadmap, Histogram, Summary, FFT, Map, and InPlace. Data is received, displayed and analysed, and the user is able to select the data to be presented from a browser, or drag from another display. Furthermore, data can be easily navigated using a specialised scroll bar, which provides users with a graphical timeline that allows them to select and see information of a specific time or distance in a specific lap. This software also allows the users to create and manipulate their own functions, based on the type of received data; to display the result in different views provided by the system; and to

automatically check the state of the car and engine. Session data can be exported and imported in various formats, including Matlab. Additionally, by using a specialised Dynamic-link Library (DLL), it is possible to write specific drivers to access other formats of data [11]. A screenshot of this application is presented in Figure 2.1.

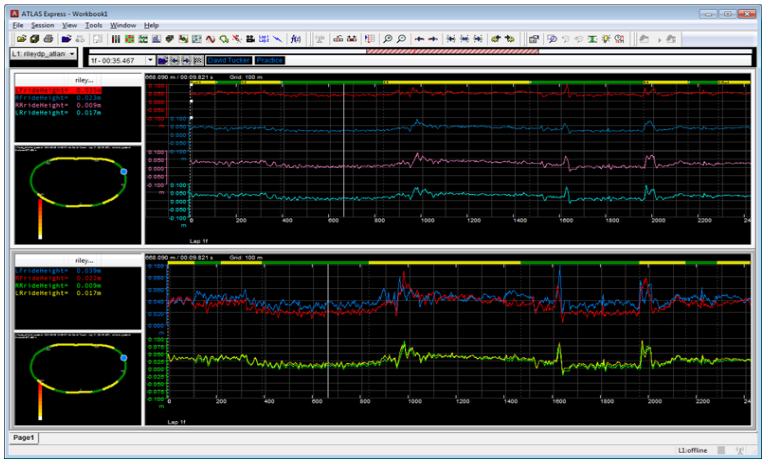


Figure 2.1: Screenshot of McLaren Atlas software application.

In order to store and manage the large quantity of data gathered at the track or by a track simulation, Structured Query Language (SQL) Race, an Application Programming Interface (API) for Microsoft SQL Server 2008 is also offered by McLaren. By using this, racing teams are provided with the means to keep data synchronized across multiple databases, allowing them to have databases located in different locations, such as in the racing track and at their factory. An example of a typical data distribution based on this system is depicted in Figure 2.2.

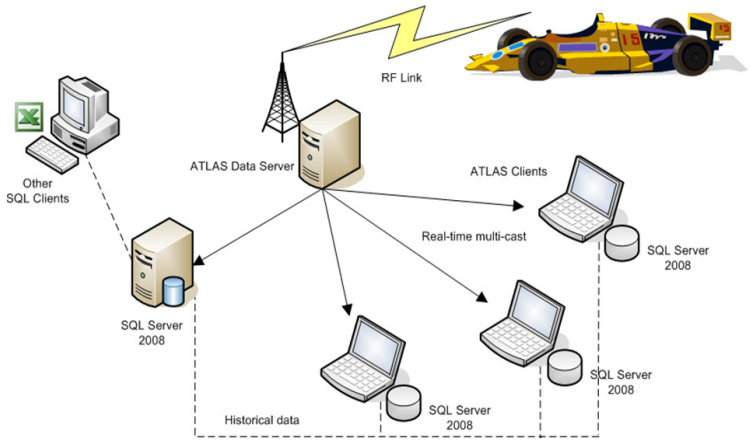


Figure 2.2: SQL Race data distribution.

2.2.2 MoTeC

MoTeC [9] is a top level motorsport technology systems manufacturer founded in 1988, dedicated to the design of efficient, reliable and versatile engine management and data acquisition systems. As a global leader, MoTec has its systems present on the military, commercial and motorsport championship industry, the latest counting with series such as 24 Heures Le Mans, American Le Mans Series (ALMS), National Association for Stock Car Auto Racing (NASCAR), FIA GT, Australian V8 Supercars, Dakar Rally and World Superbikes. Among its products, the Data Analysis Tools are the ones most relevant to this project. These tools, are divided into two subcategories, namely the *Analysing Logged Data* category and *Managing Telemetry Data* category. The three most relevant software applications in these subcategories are the following:

- The *MoTec i2 Standard* and *MoTec i2 Pro*, belonging to the Analysing Logged Data subcategory, are used to organise, interpret and manipulate large amounts of data in order to identify trends, problems and improvements;
- The *MoTeC Telemetry Monitor*, belonging to the Managing Telemetry Data category, is used to monitor data in real time and provide the means for decisions to be made quickly.

MoTeC's technology systems are herein considered as the reference tools in this work, since these are the tools that the team uses and feels most comfortable working in. This package of tools offers the majority of the functionalities desired by the FST users. The following sections discuss the three software applications in more detail.

2.2.2.A MoTec i2 Standard and MoTeC i2 Pro

The *Motec i2 Standard* and *Motec i2 Pro* are the two MoTeC's data analysis tools herein considered. The first one is freely available to all customers, and offers the read of logged data facility from a MoTec Data Logger or ECU. The second tool requires an optional Pro Analysis upgrade or Feature License, and provides the additional functionalities of advanced mathematics, multiple overlay laps, and unlimited components, workbooks, and worksheets. Examples of various screen layouts from MoTeC graphical user interface can be seen in Figure 2.3. The highlights of the i2 data analysis software includes [9–11]:

- An analysis component to accurately access and visualize data, including Graphs, Multiple Overlay Support (only available on the i2 Pro version), Synchronised Video, Dual Cursor Support (only available on the i2 Pro version) and Reports;
- A data component to manage the displaying of data to the user, including Cursor and Zoom Linking, Data Gating (only available on the i2 Pro version), Graphical Overlay Alignment (only available on the i2 Pro version);
- A math component for mathematical processing and equation management, data export and parameter recording log sheets (only available on the i2 Pro version);

- A user-defined track sections component for the user to define his own sections on a circuit;
- A miscellaneous component, which include features like workbooks and worksheets management, global channels settings, video generation (only available on the i2 Pro version), and drag racing template.

Figure 2.3: MoTeC i2 graphical user interface.

2.2.2.B MoTec Telemetry Monitor

The MoTeC Telemetry Monitor tool enables real time monitoring of the vehicle conditions. The data received in the pits originated from the vehicle allows the engineers to monitor its condition on the fly, enabling a faster problem detection and preparation of changes to improve the racing performance. A key aspect of this tool is that it needs to be clear and highly intuitive, so problems can be easily identified and decisions taken quickly. To accomplish this, the software accommodates the received data in a grid of widgets, selected and placed in the screen by the user. Among these widgets, there are dial gauges, bar graphs, virtual steering wheels, track maps and warnings. Furthermore, there is the possibility to create a log file from the received data to be analysed with i2 Pro software [9, 10]. A personalised screen layout from this software can be seen in figure 2.4.

2.2.3 Academic Solution

Similarly to the presented commercial solutions, the previous academic solution that was developed at IST also provides the means to acquire and analyse data on the vehicle, either in real time or after a running session. This solution is a result of an on-going academic work performed by several students in the context of their master thesis. However, it is mostly an aggregation of various independent and not integrated software solutions, each one developed with a specific goal in mind, to provide or demonstrate some particular functionalities of the system. The existing solution has the capability to read data gathered in the mobile station, either coming from a wireless communication channel transmitting real time data, or from a data file containing previously

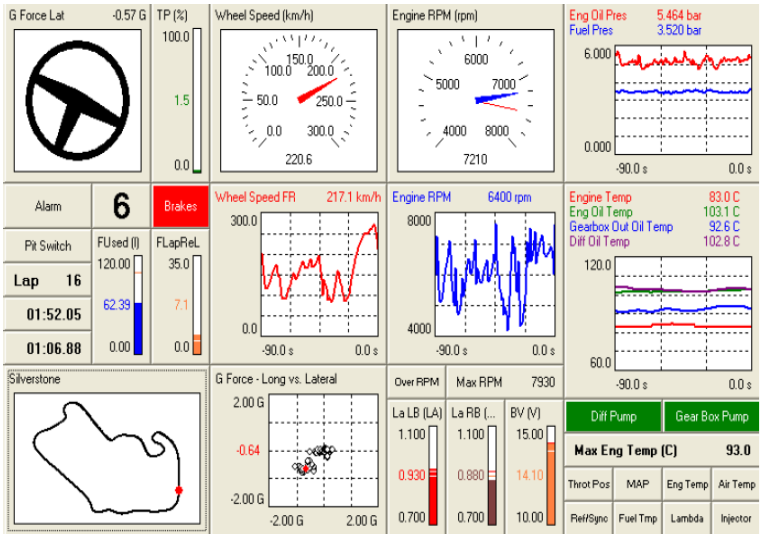


Figure 2.4: MoTeC telemetry monitor screen layout.

logged data. This tool also provides the means for a subsequent simple data manipulations, since the data gathered by the several existing sensors is sent in different formats. Data received from the mobile station (or uploaded from a file), is stored in a SQLite [3] database management system after being received and interpreted. This data includes the sessions date and time, and the whole set of collected values received or logged by the mobile station. The necessary transformations over the data are done regarding each specific type of sensor. However the set of possible transformations is static and the user is not able to change it, which forces the set of sensors in the car to be also static.

The existing application has two main parts: the *Wizard*, and the *Main Window* with its widgets. The *Wizard* is what the user sees when the application is launched, and is responsible for inquiring the user about the data needed to configure a session. The *Main Window* allows the user to visualise data from various sensors in a set of different widgets and graphs, including: a car widget, for quick observation of the state of the car; dial gauge widgets; bar gauge widget and time graph widgets. The *Wizard* and *Main Window* screen layouts are depicted in Figures 2.5 and 2.6 respectively.

Additionally, there is still the possibility to analyse the data without running the application. To accomplish this, the tool allows exporting the gathered data to a Portable Document Format (PDF) file with the information of a given session. The user is able to select which sensors and laps should be included in the file. Besides the PDF report, there is also the possibility to export the acquired positioning data to a Keyhole Markup Language (KML) file with the purpose of integrating this data into the Google Earth Application. This functionality allows the georeferencing of the acquired data using satellite images [7, 8].

Given the absence of a coherent integration of the several functionalities and the weak usability level that is offered by this tool, the racing team rarely uses this solution. However, it served as a

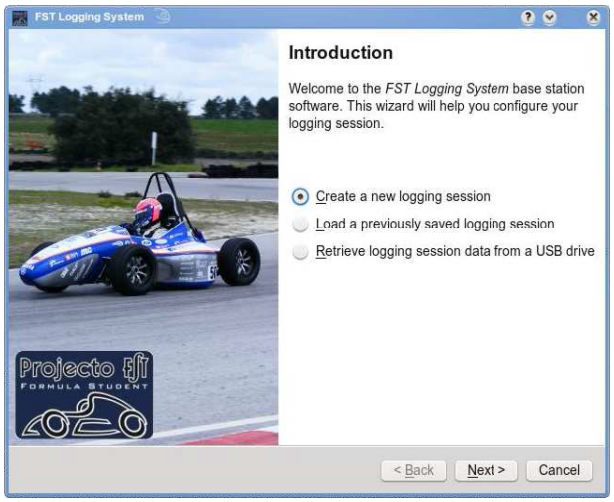


Figure 2.5: Previous FST academic solution: Starting wizard screen [7].

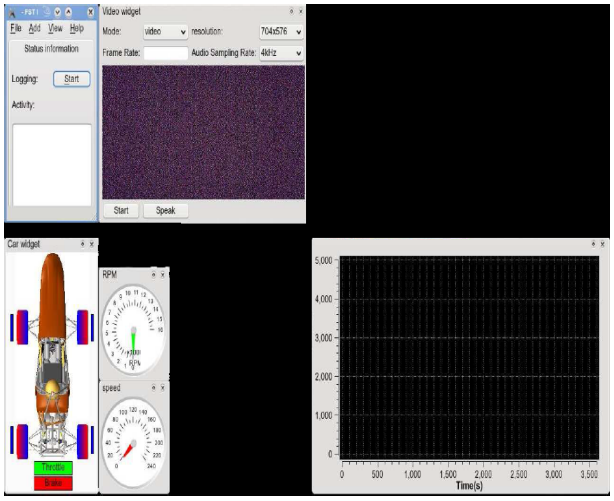


Figure 2.6: Previous FST academic solution: Main window with the real-time project graphical layout [7].

starting point for the development of this project. Important improvements needed to be introduced in order to make sure the next generation of this software is more user friendly and with the set of functionalities desired by the FST Team.

2.3 Supporting Technologies

This section provides a description of the technologies that allow for the implementation of the system. Starting with *Qt Framework* [19], facilitating with the development of the graphical user interface and operating system portability. Then, it introduces the *Qwt Library* [20], which also helped in the development of the graphical user interface, particularly by providing GUI elements useful for presenting the information from the sensors. Next, it introduces the *PostgreSQL* [1], which was the DBMS used to store the data. And finally the last technology presented, the

muParser [27], is a library which was helpful in parsing and evaluating mathematical expressions.

2.3.1 Qt Framework

Qt [19] is an application framework that helps in the development of applications in C++. Particularly, it is mostly useful for creating platform independent applications, and it also comprehends a set of technologies offered to support the software development of intuitive, modern and fluid GUIs. It supports the development through its dedicated Integrated Development Environment (IDE), Qt Creator, which is also platform independent. It was designed for providing the necessary means for Qt developers to boost their productivity to create their applications faster. Qt provided all the necessary features in the development of this project, and its relevance was emphasized in all modules of the system. It enabled the development of develop a cross platform communication protocol, cross platform DBMS operations, faster means of creating the necessary XML files for storing information, and last but most important, a cross platform GUI. All of these are described in chapter *4*.

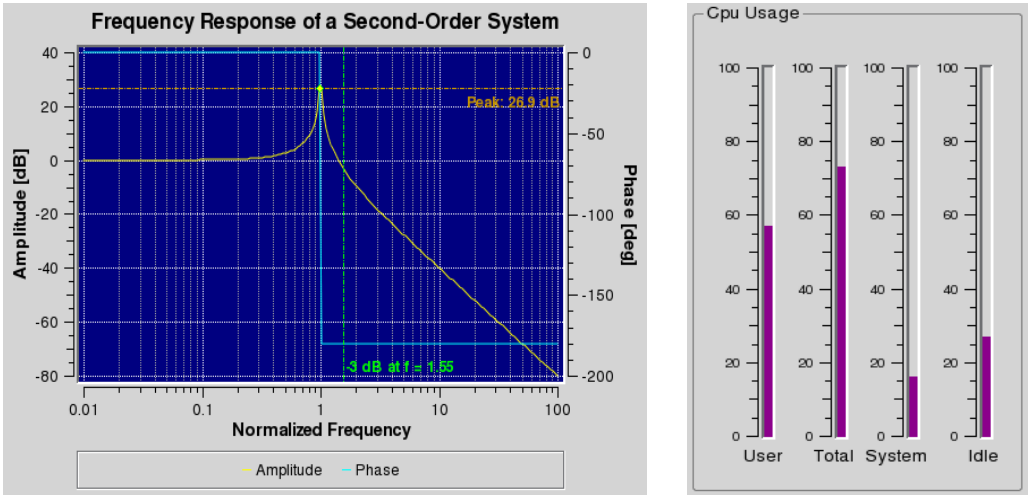
2.3.2 Qwt Library

Qwt Library [20] is a platform independent set of classes which helps in the development of graphical elements for a GUI. It can be integrated with Qt, and incorporated for faster developments in Qt Creator. It is tailored for applications with some technical context, because the elements it provides are specifically designed to present measurements associated to some scientific background. Therefore, this library supported the creation of the several types of graphs and gauges requested by the end users, as well as the time line. The most relevant components for this project that are provided by Qwt are the Curve Plots, Scatter Plots, Histograms, Dials, Compasses, Thermos and Sliders. An example with a curve plot and a thermo plot is shown in figure 2.7. Their usage in the project is detailed in chapter *4*.

2.3.3 PostgreSQL

PostgreSQL [1] is one of the most widely known open source DBMS in the world. It has been developed and maintained for more than fifteen years, and it has earned a strong reputation for reliability, data integrity, and correctness. It can be installed in almost all existing operating systems, including the ones in which the aimed framework must run. It is among the set of DBMS which can be used with Qt Framework, thanks to the existing SQL driver plugins that come along with the Qt Sources.

In the proposed work, it is used to store all the relevant information that are sent from the mobile to the pit station. This includes the data gathered from the sensors installed in the car, and information associated to the maps created from the GPS data. Details about the integration of the PostgreSQL DBMS with the application are described in more detail in chapter *4*.



(a) Curve Plot **(b)** Thermo

Figure 2.7: Example of plots and gauges provided by Qwt library [20].

2.3.4 muParser

The muParser [27] is a static library written in C++ that parses and evaluates mathematical equations. It enables the parsing and evaluation of expressions with an arbitrary number of variables. Besides the standard mathematical operators, namely addition, subtraction, division and multiplication, it provides several other built in functions, such as: min, max, avg, abs, sqrt, exp, log, log10, log2, sin, cos, and tan. This library transforms expressions into bytecode, precalculating their constant parts. It was designed with the aim to be portable and so it can be used in any operating system, working with the C++ compilers Microsoft Visual C++ (MSVC) and GNU Compiler Collection (GCC). Hence, it can be used in all major operating systems, including Windows, Linux and Mac Os. In the scope of the proposed project, it was used for the evaluation of user mathematical expressions, and its usage is detailed in chapter *4*.

2.4 Existing Hardware Infrastructure

Given the need to detect early signs of failures in the vehicle, make adjustments and improve the vehicle performance, it was decided by the FST team to create a system that would enable the transmission of information from the car to the pits for subsequent analysis. This system consists on two platforms, namely the set of components installed in the vehicle, called *Mobile Station Platform*, and a personal PC or laptop located in the pits, called *Pit Station Platform*. The communication between both stations is ensured by a IEEE 802.11 WiFi transceiver which is installed in the pits alongside the pit station platform. The two platforms the compose the system are described in the following subsections.

2.4.1 Mobile Station

The mobile station platform comprehends the aggregation of five independent parts that support the mobile station side of the presented framework. These are a single board computer which controls the operations in the car, a CAN bus used to acquire data from the sensors, a CAN to Universal Serial Bus (USB) transceiver used to convert data from the CAN bus to USB, a GPS device used to track the vehicle position, and a communication infrastructure, used to enable the transmission of data between both stations. The single board computer has four USB interfaces, which are used to connect the other platforms. The former runs a Knoppix 6.2 Linux operating system that is installed in a USB device. The framework is run through the operating system, acting as a bridge between all the other components and the framework. When needed, the USB device referred above, is also used to store data for subsequent analysis, namely data from the sensors. All of these devices were selected and integrated in the overall project in the context of previous master theses.

The single board computer that supports all the operations on the mobile station is depicted on Figure 2.8 and its technical specifications can be found on Table 2.1. More details are given on [7].

Figure 2.8: VIA EPIA-P700 Board equipping the car mobile station [7].

Processor:	VIA C7 x86 @1GHz
Board	VIA EPIA-P700 compact board with 1GB of DDR2 RAM memory
Chipset	VIA VX700 Unified Digital Media IGP
Graphics	VIA Unichrome Pro II IGP
Storage	Pen Drive USB 4GB
Interfaces	1xIDE 1xS-ATA 4xUSB 1xSerial RS232 1xVGA 1xGigabit Ethernet

Table 2.1: Mobile station specification [7].

The CAN bus installed in the vehicle and used to gather data from the several sensors is

depicted on Figure 2.9. This equipment is connected to the transceiver described below, to which it sends the gathered information. It was developed in the context of a previous thesis [6], where its specifications are described in detail. This equipment was specifically developed because the single board computer installed in the car does not support the CAN interface. Therefore, the CAN bus could not be directly connected to it.

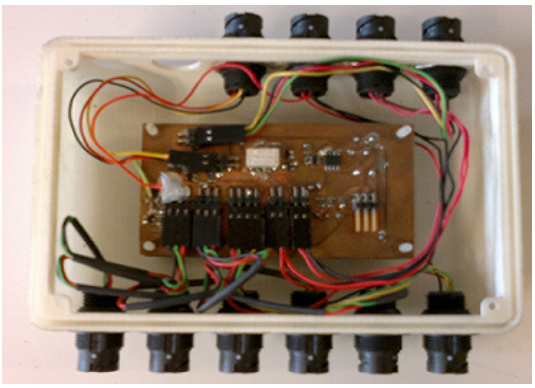


Figure 2.9: CAN bus installed in the vehicle [6].

The CAN to USB transceiver installed in the car is depicted on Figure 2.10. It collects data directly from the CAN bus and converts this data to the USB protocol, to be posteriorly read by the single board computer. It was also developed in the context of a previous thesis, being described in more detail in [7].

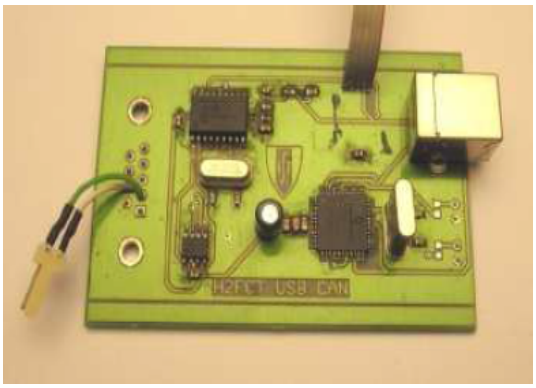


Figure 2.10: CAN to USB transceiver installed in the vehicle [7].

The GPS device that enables the acquisition of the vehicle's position, alongside other information such as time, is illustrated on Figure 2.11, and further detailed in [8].

The architecture of the mobile station side of the framework is described in Chapter *3*, and implementation details are presented in Chapter *4*.

Figure 2.11: Supporting GPS device installed in the mobile station [8].

2.4.2 Pit Station

Contrary to the mobile station, the pit station does not represent any specific part of hardware, although it must fulfil some minimum requirements to run the pit station side of the framework. In most situations, this will be a computer belonging to some member of the IST FST team, which means a recent PC or laptop. The minimum requirements that are necessary for the installation and correct execution of the framework in the pit station are the following:

- Have the PostgreSQL [1] DBMS (version 9.3 or later) installed, in order to enable the storage of the information gathered form the sensors.
- Have the Qt Framework [19] (version 5.2 or later) installed, to give cross-platform support.
- Have Qwt Library version [20] (version 6.1 or later) installed, to support platform independent design of the graphical elements that display the received information, namely the data gathered from the sensors or from user defined formulas, at runtime.
- Have the Matlab Compiler Runtime R2012a [25] installed. This enables the execution of the compiled Matlab modules associated to the GPS data processing.
- The supporting operating system can be one of the following: Windows, Mac or Linux. These three operating systems are supported by all the above mentioned software components.

The installation guide of the pit station software, covering the necessary external components for the correct operation of the framework can be found in Appendix C. The architecture of the pit station side of the framework is described in Chapter *3*, and its implementation details presented on Chapter *4*.

2.4.3 Communication Infrastructure

In order to support the communication between both stations, a USB Wi-Fi adapter and a wireless router are used [7]. The USB adapter is connected to the single board computer installed in the car. It enables the mobile station to transmit and receive messages through a private network. Figure 2.12 illustrates this hardware and Table 2.2 states its main characteristics.

Figure 2.12: Wi-Fi USB device installed on the mobile station [7].

Indoor Range	75m at 1Mbps / 40m at 54Mbps
Outdoor Range	350m at 1Mbps / 60m at 54Mbps
Transmit Power Output	16dBm at 1Mbps / 12 dBm at 54Mbps
Max. Data Rate	54 Mbps
Receiver Sensivity	-92dBm at 1Mbps / -65 dBm at 54Mbps

Table 2.2: Wi-Fi USB adapter specifications [7].

The wireless router acts as a manager of the wireless connection between the two stations. The other alternative would be to have the mobile and pit stations work in ad-hoc mode. However, this has bad support under Linux operating systems, so it was disregarded (see [7]). The router is placed in the pits, next to the computer that acts as pit station. Its most relevant characteristics can be found in [28].

3

System Architecture

Contents

3.1 Overview

This chapter describes the proposed solution and the resulting architecture for the considered framework. As it was already mentioned, it comprises two important units: a *mobile station*, where data is acquired and transmitted, and a *pit station*, where the data sent from the mobile station is stored and analysed. In this section an overview of the overall system architecture is presented, composed by the aggregation of both stations. The two following sections discuss each of the stations, detailing the architecture of its components. In this architecture, communication is performed in both directions, both from the mobile station to the pit station and vice-versa. However, the most important communication flow is from the mobile station to the pit station since data is acquired by the former and stored in the latest, creating a client-server style architecture. Figure 3.1 illustrates a layered view of the overall system architecture for the proposed system, identifying a boundary which represents the subset of modules that are affected by this project, and additionally the bridge of communication between both stations. It also represents a deployment view of the framework. This is because each of the represented components, are allocated to the corresponding platform in which it is assigned in the figure.

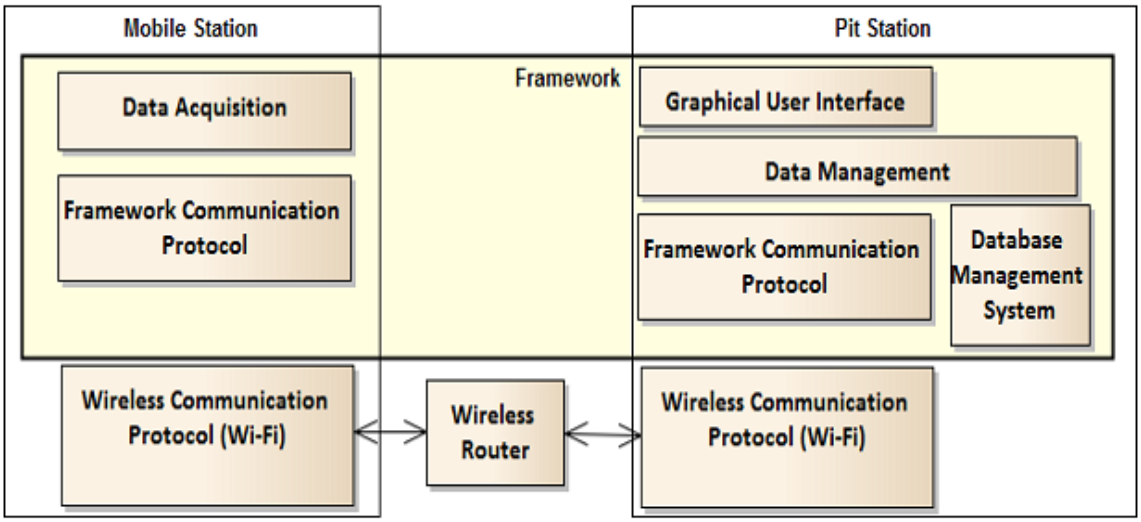


Figure 3.1: Overall system architecture.

The *Framework Communication Protocol* module is the only one that is common to both stations. This is where all the communications between the two stations are implemented. In the mobile station side, this is the only software module affected by this project. This module is further discussed on the Communication section. The presented architecture and belonging modules were devised with several objectives in mind. The main aims were to:

- Facilitate software maintenance, as the separation of the offered functionalities is done across the several modules, facilitating the correction of faults, improve the performance, or enhance other attributes.

- Provide a good debugging ability, as the several modules can be used to identify particular faults related to different sources of activity. This can be, for example, a problem in the operations on the DBMS, or in the communication protocol.
- Allow the possibility of software reusability, since the implemented modules offer particular characteristics. These count with, data management, data processing, network communication, and display of data to the user. With this in mind, this functionalities are separated into different modules to easily allow them to be integrate, and thus used, into other developments without much offering the need for further efforts.
- Make it extensible without much effort. This can be done either by extending an existing module or by creating a new module to incorporate new functionalities.

3.2 Communication

While the car is running, data is gathered from sensors and sent from the mobile station to the pit station. This section describes the communication protocol used for network transmissions of this data between the two stations, with a brief explanation of what was already implemented in the previous version, followed by the improvements introduced in the new version. The devised solution is based on an improved version of the protocol developed for the previous version of the framework [7], detailed in Appendix A.

The communication between the two stations is performed by using a Wi-Fi connection. This is accomplished by having a Wi-Fi USB adaptor connected to the mobile station and by having a wireless router next to the pit. The communication flows from the mobile station to the pit station through the router [7]. Both stations send and receive packets. This means that while the team is performing a test session, the vehicle will be sending messages to the pit station and vice-versa. However, during testing sessions the mobile station will mainly act as client, while the pit station will mainly act as server. This is justified by the fact that relevant information about sensor's measurements is only sent from the client (the mobile station), to the server (the pit station). The structure of the network packets used in the previous software framework version to transmit data between the two stations is depicted on Table 3.1.

16 bits	16 bits	16 bits	Number of bytes indicated by the size	8 bits
Packet Number	Size	ID	timestamp and measurement	Checksum

Table 3.1: Previous packet structure used in the communication between the two ends [7].

- The *packet number* field is used to assure the order of the packets and to detect lost ones. It is incremented wherever each new packet is sent;

- The *size* header field indicates the length (in bytes) of the data field;
- The *ID* header field is used to identify the sensor from which the data were generated from. This field has a fixed length, and it is also used for control messages, as described later;
- The *data* field is where the measurements read from the sensors are transported. This field has a variable length, which is indicated by the *size* header field;
- The *checksum* field is used to ensure the integrity of the packet. This is accomplished by having the sender computing a checksum function over the content of the *data* field, and inserting it in the end of the network packet. Upon receiving a message, the receiver computes the same checksum function over the content received in the data portion, and checks it against the received checksum. If both match, no corruption of the message exists and the packet is stored to be processed. Otherwise, the packet is discarded.

The protocol was designed to be reliable, and to work in the presence of network failures. These failures can go from low to high severity. The lower severity ones include packet losses and small network failures. For these situations, the protocol maintains a structure with already transmitted packets, which is used to retransmit in case a loss occurs. The packet numbers associated with each packet are used to deliver the packets in the corresponding order, and to request a retransmission in case any packet is lost. High severity failures include severe network failures, in which case it is not possible to reconnect. For these situations, the data read from the sensors is stored in a dedicated file in the USB storage device that is connected to the on-board computer, to be subsequently used in the pit station to load information corresponding to the session.

Although the new version of the software framework also adopts the existing communication protocol, some improvements and modifications have been introduced. These relate not only to the design of the protocol, but also to some specific implementation details. The first tend to ease the modification of the content that is sent in each network packet, presented later in this section, while the second enables the software to be used on multiple operating systems. The implementation details are further described on Chapter *4*.

Regarding the design of the protocol, it was agreed with the FST team members to change the structure of the used network packets. This modification was necessary in order to make the communication protocol as independent as possible from the content transmitted between the two stations. This way, it is possible to change what is transmitted in the packets without changing their structure. Besides this, the network packet structure was also modified to cope with the required flexibility degree in what regards to the sensors to be supported by the system, and therefore their management. In fact, in the previous version of the framework it was not possible to send information from two sensors in the same network packet because its structure was strictly designed to accommodate data from one sensor only. With this modification, it is now possible to send data from an arbitrary number of sensors in just one single network packet, avoid network

delays introduced by transmitting a bigger number of packets. The main difference in the new packet structure, illustrated in Table 3.2, lies in the removal of the header field with the identifier of the sensor, which is now sent in the data field. The latest becomes indeed a structural field, where the identifier of the sensor is sent alongside the relevant content that it originated. Since the ID has a fixed length, it can always be read from the data field, without the need to have its size being given. The sensor data is placed next to the identifier. Hence, the size of the sensor data is variable and depends on the considered sensor. Consequently, the receiver will have to infer, from the received identifier, the length of data that should be read from that packet. For such purpose, a parametrization table will be filled in the pit station that sets the payload size associated to each sensor.

16 bits	16 bits	The number of bytes indicated by the size field	8 bits
Packet Number	size	Data (sensor id, timestamp and measurement)	Checksum

Table 3.2: New packet structure.

Besides its conventional usage for sensor data transmission, the identifier portion of this field can also be used for control messages. These can be: *acknowledgements*, to signal the reception of some packet; *requests*, to request the transmission of some packet; *sync*, to synchronize the current packet number in both stations. Other two types of messages exists that are not included in the control messages category and have the purpose of starting and finishing the execution of the communication between both stations. These are: *start*, to start the communication between both stations, and *stop*, to finish the ongoing communication process. Each of this messages have a dedicated identifier, in order to be clearly identified.

3.3 Mobile Station Architecture

This section describes the architecture of the mobile station in more detail. As it was already mentioned, this is the part that is responsible for acquiring data from the vast set of sensors connected to the CAN bus and from the on-board GPS device, as well as to send this data to the pit station. Hence, the mobile station architecture of the framework was structured according to the needed functionalities. These are: reading data from the CAN bus, reading data from the GPS receiver, transmitting the data over the network. This structure can be seen in Figure 3.2, which also includes the external components used for data acquisition and transmission over the network.

The depicted elements are described as:

- **Wireless USB Device:** Used to ensure a network connection in the mobile station.

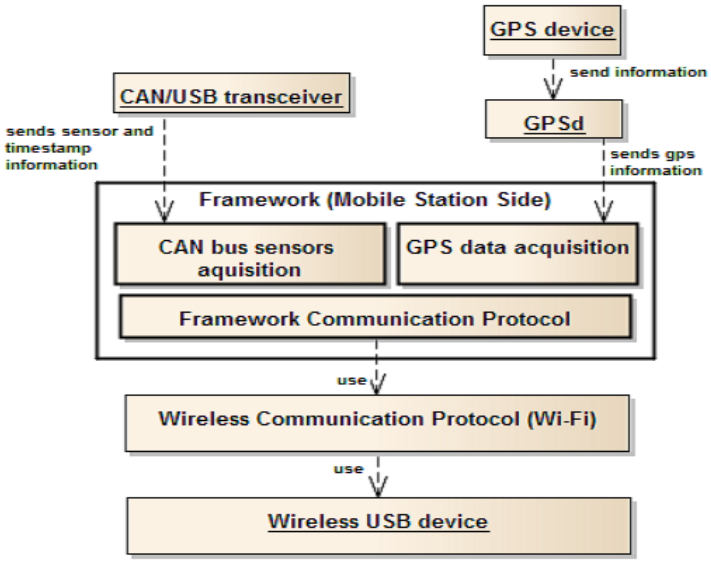


Figure 3.2: Architecture of the mobile station side of the framework.

- **Wireless Communication Protocol:** Used by the framework for the transmission of messages through the network. This directly uses the device mentioned above, for transmitting and receiving messages over the network.
- **Framework Communication Protocol:** Deals with all communications between both stations at the framework layer, thus being responsible for ensuring the reliable transmission of data over the network. Since the mobile station is the place where sensor data is produced, this module will be mainly concerned with sending the data gathered from the sensors. Besides receiving messages to start and stop a *real-time*, it will only receive control messages from the pit station, used to ensure the ordering and reliability of each transmitted data packet. This module was already implemented in the previous version of the framework, ensuring the protocol further described in Appendix A. The new version counts with the already described adaptation to include the improvements outlined in the previous section. It works together with the *Wireless Communication Protocol* and the *Wireless USB Device*. In practice, this was the only module of the mobile station that was affected by this thesis.
- **GPS Receiver:** Performs the communication with the GPS satellites, calculating the absolute position of the mobile station on Earth [8].
- **GPSd:** Independent daemon and device driver that runs in the operating system layer, which makes the interface between the GPS receiver connected to the mobile station and the framework. It reads coordinates data from the receiver and converts it to convenient data structures, which are passed to the framework [8].
- **GPS data acquisition:** Receives the coordinates data from the GPSd daemon and formats it accordingly, in order to be subsequently used by the *Framework Communication Protocol*, before being sent to the pit station [8].

- **CAN/USB transceiver:** Connected to the CAN bus, it allows the sensors installed in the car to have their measurements transmitted to the single board computer. This is done by transferring the data received from the CAN Bus into the USB bus.
- **CAN sensors data acquisition:** Responsible for receiving the sensors data from the above described transceiver. This information contains not only each value that is read from a specified sensor, but also a unique timestamp generated by the CAN bus. Just like the GPS data acquisition module, this module also formats the received data accordingly, by using the *Framework Communication Protocol* to send it to the pit station.

The above described modules compose the structure of the mobile station and are deployed in the mobile station infrastructure described in section *2.4.1*. However, only one, the *Framework Communication Protocol*, is affected by this work. In the previous version of the framework, this module used the network packet structure depicted in table 3.1. It forced sending one network packet per measurement gathered from the sensors. To overcome this, the design of the *Framework Communication Protocol* was changed, particularly to accommodate a new network packet structure, which would allow data from an arbitrary number of sensors to be sent at once.

3.4 Pit Station Architecture

This section describes the architecture of the pit station in more detail. As mentioned above, this is the part of the system that is responsible for receiving, processing and displaying the data from the sensors and GPS receiver to the users. Its architecture was also structured based on the needed functionalities. These are: displaying information to the users, store the sensor data on the database, managing the data stored in XML files, processing data from the sensors and from the GPS device. This structure can be seen in Figure 3.3, which also includes the external software components used for database operations and data transmission over the network.

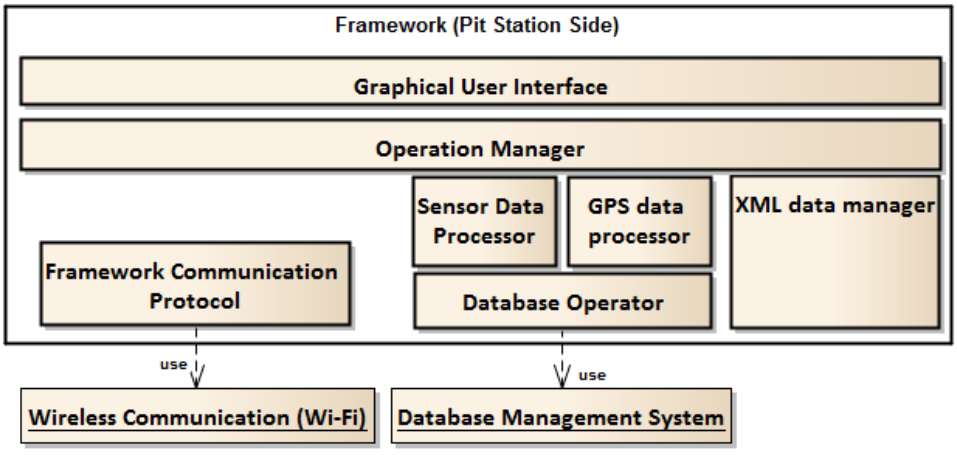


Figure 3.3: Architecture of the pit station side of the framework.

The elements contained in the architecture of the pit station are described as follows:

- **Wireless Communication:** Communication infrastructure contained in the pit station, which enables the transmission of messages to the mobile station.
- **Framework Communication Protocol:** Deals with all communications between both stations at the framework layer, thus being responsible for ensuring the reliable transmission of data over the network, performed when data is being acquired in real time. Regarding the main tasks of the pit station in what refers to network communication, it will mainly act as a receiver, receiving all the data gathered from the sensors at the mobile station. Additionally, it will send back control messages to the mobile station to ensure the ordering and reliability of the transmitted packets. This module was already implemented in the previous version of the framework, ensuring the protocol described in Appendix A. This updated version also counts with its adaptation to include not only the improvements outlined in Section *3.2*, but also with the introduced modifications to provide an operating system independent application. As such it makes use of a cross-platform interface which can be used in any of the required operating systems. It works together with the *Wireless Communication* module.
- **Database Management System:** This is the module that represents the actual database. The framework uses this module to make all data related to the sensors and the GPS receiver persistent. It provides easy means for accessing previously logged data and analyse it at any time. This module is cross platform, hence available for the operating systems required by the end users.
- **Database Operator:** Since the data gathered by the sensors installed in the car need to be made persistent to enable posterior analysis, this module is responsible for all operations related the manipulation of the sensor data over the database at the framework level. Further more, it makes use of the *Database Management System* to achieve this purpose, and enable the data to be available at any time. The data to be stored belongs to the sensors installed in the car as well as sensors composed of mathematical manipulations over the formers. This module provides support for storing data when receiving new sensor information, and for querying for data when analysing previously stored information.
- **XML Data Manager:** This is the module mainly related to the flexibility provided to the users in what regards to their preferences. These preferences may include sensor information, information related to the GUI, or user personal preferences (such as loading the last project at start-up). This is mainly useful, for instance, to enable the GUI to adapt to each user profile and display only the required information. This allows the user to chose which information should be loaded, corresponding to the sensor data, and hence avoid loading unnecessary data. It uses information gathered from user input, to manipulate data in the corresponding XML files. It also provides interfaces for all processing operations related to read, write and update user preferences into these files.

- **Sensor Data Processor:** The information related to the sensors has to go through some processing before being able to be displayed to the users, or stored in the database. The aim of this module is to process all the data related to the sensors connected to the CAN bus at the mobile station as well as the data representing posterior manipulations over these sensors, before this can be stored and displayed. This module uses the *XML Data Manager* module to get any information related to the sensors that it needs to perform the operations over their data, and the *Database Operator* module to store any data in the database.
  The main functionality of this module consists on applying mathematical formulas to sensor data. When creating a new session, either *real-time* or *offline*, these formulas are applied to the data received and then this is stored in the database or displayed. For the sensors composed of a mathematical manipulation over other sensors, it first implements a specific interpolation over the set of data points in the sensors present in the manipulation, followed by applying the corresponding mathematical formula for interpolation set.
- **GPS Data Processor:** It is responsible for processing and storing all the data related to the positioning of the car, gathered from the GPS receiver in the mobile station [8].
- **Operation Manager:** This module has two main purposes: one consists on being interface between the *Graphical User Interface* module, and the remaining modules (except the *Framework Communication Protocol*), when some user initiated operation needs to be performed (e.g. load previous session), or to fetch data to display to the users; the other is to work as an interface between the *Framework Communication Protocol* and the *Sensor Data Processor* modules, when data is being acquired in real-time. Basically, it coordinates all the operations performed in the framework, either initiated by the user or caused by new data arriving from the mobile station, by issuing the corresponding operation to be performed on the module responsible for the task when an external event occurs.
- **Graphical User Interface:** This module is the direct point of interaction with the end users, where they request the operations to be performed, and analyse the data that is displayed. It manages all the data associated to the user interface, provides the users with means to adapt the interface to their needs, and works alongside the *Operation Manager* to adjust to this set of needs, by requesting the necessary information to generate a representation to be displayed to the users. This module makes use of several widgets to provide the user with all of its functionalities, which can be divided into three categories: manage information related to a particular session (e.g. sensor information, user preferences, mathematical manipulations); display of graphs and gauges showing sensor data; display of sensor data values alongside other session parameter values defined by the user.

This station is where the improvements are more relevant. In the previous version of the framework, it lacked key features that prevented the team from using the application. Namely, the team needed a cross-platform framework, since not all team members use the same operating

system; the set of features available did not fulfil all the users requisites, since they were not able to manage the information related to the sensors through the application; the graphical user interface, was not designed to accommodate the user working styles.

For the new version, these difficulties were overcome by using a supporting application framework to help making the pit station implementation operating system independent. The implementation counts with new modules that allow the users to manage all the information related to sensors through the application, and a new graphical user interface that is designed to adapt to each user profile and to allow the users to perform the needed operations for the type of analysis that is hereby considered.

4

Pit-Station Implementation

Contents

4.1 Summary

In this chapter, a detailed description of the *pit-station* implementation is provided. It is based on the architectural structure of the *pit-station*, described in the previous chapter. It starts by providing a brief depiction of the cross-platform framework that was used to support the development of this side of the framework, followed by a detailed description of the particularities of each of the comprised modules in the earlier presented pit station architecture.

It is important to note that only the modules that were implemented or modified in the scope of this thesis will be considered by the following sections. Hence, all the *pit-station* modules are described except for the GPS data processing module, since it was developed in the context of a previous work [8] and it was not changed in the context of this thesis.

The programming language that was chosen for the implementation of the pit station software was C++, because the goal was to have a an efficient language while being object oriented, in order to be easy to maintain and modify the code. Additionally, some development tools were also used to support in the development of the system to best fulfil the user requirements. These tools, presented in Chapter *2* and herein detailed, in the sections describing modules the in which they were used. It should be reminded that the main objectives of the pit station are concerned with providing the users with a comfortable working environment, while providing the required set of functionalities. This, alongside the desired operating system abstraction, were the most important factors that were taken into account during the development process.

As described in the previous chapters, this station is responsible for processing, storing, and displaying to the users the data gathered at the mobile station. This means that it comprises the most fundamental elements to enable users to perform a useful analysis of the vehicle conditions. This analysis may be performed in two different modes, namely, when data is being acquired in real time, referred to as *real-time*, and when loading previously logged data from the database, referred to as *offline*. It is easy to identify that, with these requirements, there is a need for an application to provide the functionalities suited for the type of analysis hereby considered, with an integrated user-friendly graphical interface enabling the users to perform their tasks. Thus, the end result of this station is an application capable of performing the types of operations needed by the users, with a dedicated graphical user interface to display the data in the way that it improves the quality and productivity of their analysis sessions.

The previous version of the framework had three threads executing in the application. These were:

1. A thread to deal with the presentation of information to the user. This thread also dealt with the processing of loaded data and operations over the database when in *offline* mode. It also issued the starting and stopping session operations, for *real-time* mode.
2. A thread to deal with the processing of data and the operations over the database, when

in *real-time* mode. This thread processes the data and sends the information to the above thread, which at this point will only display it in the graphical elements.

3. A thread dedicated to the reception of network messages when in *real-time* mode. This thread implements the earlier described communication protocol, although without the improvements made in the context of this thesis.

Although the new version of the framework also comprises the execution of three threads, the required tasks will be split in a different manner. The new task division across these threads is the following:

1. A dedicated thread for operations related to the network communication between both stations. This thread implements the earlier described communication protocol.
2. A thread for coordinating all the operations over the data. This encapsulates all type of operations to be performed over the information, such as database operations, process sensor data and GPS data, perform mathematical manipulations over the sensor data, etc. This thread receives information from the communication thread, presented above, when data is being acquired in real time. It also provides information to the thread described below.
3. A dedicated thread for the user interface. This is responsible for providing the users with the graphical means to allow them performing a useful analysis. It deals directly with user input, and it requests information that it needs for building the correct views to be displayed to the users from the above thread.

As it can be inferred from the above description, the main changes from the previous to the current version of the pit station implementation lie in the thread which presents information to the user. In the current version, this thread no longer performs any processing over the data or issues directly any network operation. The processing over data is made by a dedicated thread, concerned with the processing and the storage of data. The network operations are also performed on a dedicated thread. The allocation of the several modules comprised in the architecture of the pit station to each of the above presented threads is depicted in Figure 4.1.

4.2 Cross-Platform Application Framework

This subsection describes the adopted approach to assist in the development of the aimed application. A cross-platform application framework was considered for this purpose, in both the previous and the new versions of the application. In this new version, it was decided to use the same framework as adopted in the previous one. This subsection outlines the application framework adopted for the previous version of this project, followed by the reasons why the same framework was used for the development of the new version.

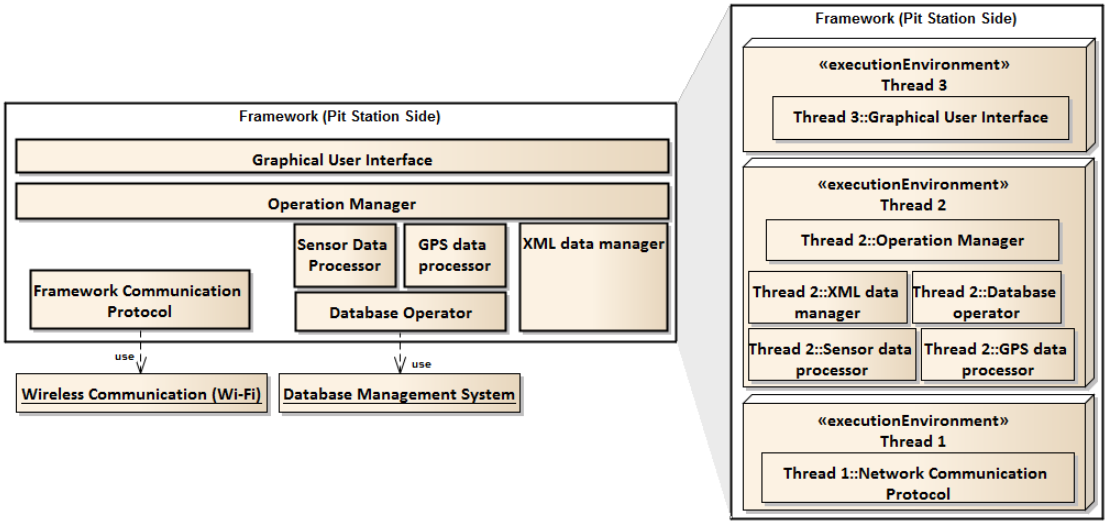


Figure 4.1: Allocation of the pit station modules to the corresponding threads.

In the previous version of the application, the Qt framework [19] was adopted for this purpose [7]. The necessary requisites for such a framework would be to provide operating system independence, the required means for displaying several types of technical widgets, offer an IDE for faster graphical user interface development, and have good documentation support. With this requisites in mind, the first choice would be to continue with the same framework that has been chosen for the previous version of the software, as it directly satisfied all requirements [7].

The other considered frameworks were GTK+ [18], and wxWidgets [17]. As for GTK+, the option was disregarded since it was designed primarily for applpication development using the C programming language, although it has the option of integrating several binds for many other languages, including C++. However, some useful higher level features that are provided by Qt (for example, signals and slots), are not provided at the same level by GTK+ or, in this particular case, by its C++ bind. As for wxWidgets, it was also disregarded. The reasons were that Qt is more matured in network programming which is also a requisite for this system. As an example, Qt offers matured dedicated features for User Datagram Protocol (UDP) communication, unlike wxWidgets. The other reason is concerned with the better support available for Qt than for wxWidgets. Besides this, Qt has its own distributed IDE, alongside with Qt's libraries, developed with the purpose of better providing the users with a more easy and intuitive use of Qt's features to improve their productivity.

4.3 Framework Communication Implementation

This subsection describes the decisions that were considered for the implementation of the communication protocol in the pit station. It starts by briefly outlining the protocol implementation for the previous version of the framework, followed by the description of the improvements that

were introduced for the current version.

In the previous version of the software, the communication protocol was operating system dependent, namely it could only be used on an Unix based operating system. The communication was made possible through the usage of connectionless Portable Operating System Interface (POSIX) sockets, by using the UDP. The reason for choosing this protocol was mainly due to the imposed system requirement to deal not only with packet loss but also connection losses. The other alternative would be the usage of Transmission Control Protocol (TCP), which guarantees a reliable transmission of packets while a connection is set. However, the system would still have to deal with the mentioned connection losses and so it would still have to implement a reliable transmission protocol, duplicating this functionality. Besides this, the TCP protocol also introduces significant latencies that would make the real-time transmission of information more difficult.

The network communication tasks can be divided into two categories: sending and receiving network messages. Starting by the latest, the processing of newly received information was mainly split between two threads when working in *real-time*. One thread was blocked waiting for a network packet to arrive, while the other was waiting for an acknowledgement that new data is ready to be processed. In other words, the first thread was blocked waiting for new network packets, upon which it retrieved the relevant data from the packets, and stored it in a list of received messages. The second thread was blocked until any new data was put in the list of received messages to be further processed. Hence, a control mechanism had to exist to coordinate these two threads. The chosen method was based on the producer-consumer concurrency mechanism, with both threads using a shared *semaphore* to ensure the access control to the list of received messages. Hence, when relevant data was retrieved for each newly received packet and stored in the list, the semaphore was updated to provide access for the new resource. This way, it is ensured that the thread responsible for processing the relevant data on each network packet is blocked until there is at least one available. This is illustrated in Figure 4.2.

For the sending side of the implementation, there are two situations in which messages are sent. One is for starting and stopping a session, and the other is a retransmission of a network packet. For the first situation, the thread responsible for sending the messages was the same that displayed the content to the users. For the second situation, an independent thread was launched to take care of the retransmission task every time a retransmission was required.

Since the processing capabilities of the devices used for data analysis in the pit station are not as constrained as they are in the mobile station, the changes introduced for the new version were mainly related to two relevant requirements: one is the portability of the software across multiple operating systems, with a particular consideration to the Microsoft Windows operating system, as this is widely used by the end users; the other is to make better use of the processing resources.

Since the previous version of the framework was specifically tailored for Unix based operating systems [7], a new implementation of the communication layer had to be considered in order

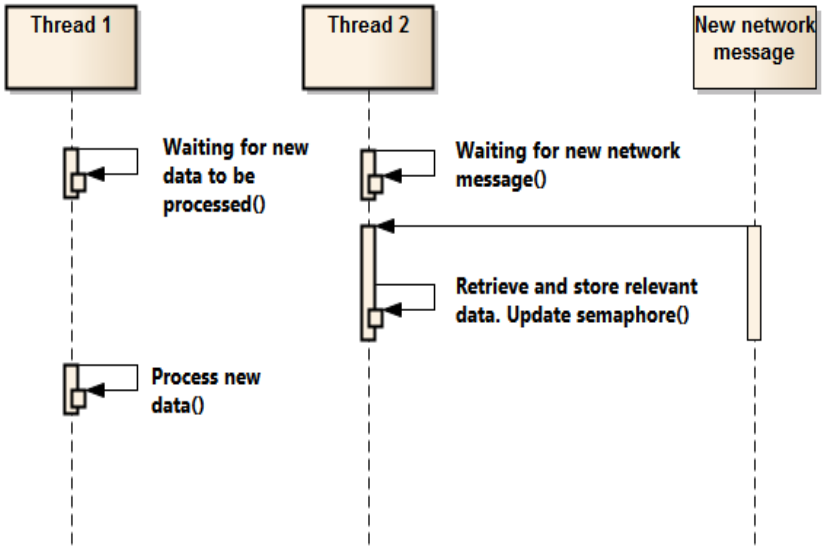


Figure 4.2: Receiving and processing new information in the previous framework version.

to satisfy the new requirement concerned with operating system independence. To solve this problem, the option of having one specific implementation for each operating system in which the framework should run was considered. This option was immediately discarded because, there is another approach that revealed to be much more effective for this purpose and avoids the need to write the same functionalities for different operating systems. The choice was the usage of Qt's sockets API [19]. With this approach, the network layer implementation is made independent from the operating system in use, allowing for a single version to run on every operating system on the pit station end. The only requirement is to have Qt libraries installed, which is possible for the target end user devices. Additionally, although Qt can be used with several programming languages, the C++ binding was chosen, since this was the programming language selected for the whole pit station implementation. For the second requirement, concerned with making better usage of the processing requirements, another Qt feature was taken into consideration: Qt signals and slots. This avoids the need to have any thread being explicitly blocked by locks or semaphores, waiting for some data structure to be updated by another thread, thus wasting processing resources. The threads communicate with each other through this signal/slot mechanism, provided by Qt API.

In the new version, both the receiving and transmitting messages procedures are performed by using the signal/slot mechanism mentioned above. For such purpose, there is a dedicated thread for network operations, which has particular methods (called slots) that are executed upon receiving specific signals for message transmission and reception. Hence, when a network message arrives to the pit station, a specific signal is emitted by Qt, and a slot in this thread is executed to read the new message. If a message contains sensor data, a signal is emitted in the communication thread, and a slot is executed in the thread dealing with the data processing, to process the new arrived data. As for transmitting messages, the thread that coordinates the operations

over data sends specific signals to the communication thread and another slot is executed in the latest, that transmits a message. This process is illustrated in Figure 4.3.

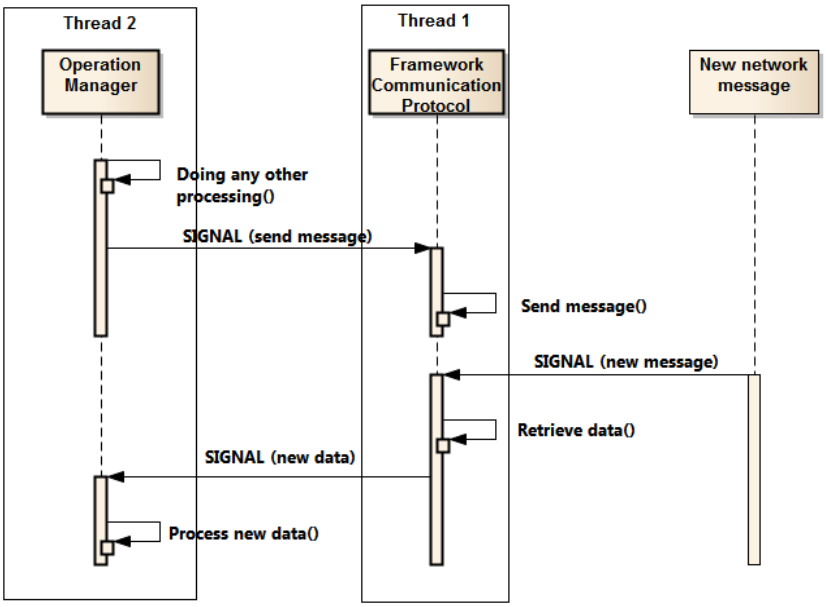


Figure 4.3: Network processing in the new framework version.

For the implementation of the protocol, two independent aspects had to be taken into account. One deals with the operations on the queues, (see the protocol design described in Appendix A), and another deals with network operations. The first has operations related to selections, insertions and removals from the queues used to manage the received and sent network messages. The second is an extension of the class *QUdpSocket* provided by Qt Framework. The class *QUdpSocket* provides platform independent methods for network operations.

4.4 Database Management and Implementation

This section describes the decisions that were taken with regards to the database, not only in terms of the used database management system, but also about the implementation details for accessing information in the database. All the data received from the mobile station is made persistent through the DBMS, to enable its posterior analysis. This provides a standardized mechanism of storing, accessing and changing data, without incurring into the overhead of implementing a less powerful storage system and yet, have to face problems arising from its implementation. This also provides the means for application and operating system independence, where data can be accessed from different applications without any particular protocol implementation, since this is made using SQL, a well known standardized language for database management. In the previous version of the software framework, the used database management system was SQLite [3]. However a new database management system had to be adopted for the following reasons:

- Having the whole database stored in a single file leads to concurrency issues, since multiple writes, and simultaneous reads and writes cannot both happen, even if they are performed on different data fields;
- Using a single storage file directly affects speed. On the bright side, the SQLite database allows the operations that are done inside a single transaction to discard the need for multiple lock acquisitions and releases on the same database file. However, since there are multiple transactions occurring in the system, this still results on many lock acquisitions and releases, which are costly operations and hence, forcing data to be stored in the database at a slower rate.
- SQLite does not provide some features needed for this project, since the set of data types is very restricted.

Hence, the considered improvements were mainly made to overstep the problems presented above. The set of most relevant improvements that were introduced in the new version of the software framework relate to the changing of the database management system in use and turn operations over the database into operating system independent. The most relevant requirements of a database management system in this project were:

- Cost - The academic nature of this project project imposed the choice for a free database management system.
- Speed - It needs to be fast, in order to cope with the rate at which data arrives at the system.
- Portability - It should be able to be installed in all the considered operating systems.
- Conform to SQL language - As a standardized language for database management, its use in this project forces the choice for the DBMS to implement most of the SQL features.
- Manageable through Qt - In order to have the platform and the database independent, an appropriate interface for accessing SQL databases should exist, so that the database can be managed using the QtSql module [19], which provides cross-platform database integration with the rest of the application.
- Easy administration - To facilitate maintenance, it is necessary to have a good administration tool.

With regard to these aspects, the choice was made between two database management systems: PostgreSQL [1], and MySQL [2]. The factors that uniquely influenced the final choice were related to speed in multi-transactional environment, as all the others were easily satisfied by the alternatives that were taken into account. Concerning the speed of *MySQL* and *PostgreSQL* DBMSs, there is an active discussion on which is faster. This, however, depends on indirect factors such as, the specific tuning of the database management system to the target application, the database structure, and the type of operations which will be performed on it. Accordingly, the decision that was taken for this new version of the framework is to adopt the database struc-

ture already designed and deployed in the previous version, as described in [7]. Additionally, the database should have to handle a significant number of transactions, and possibly concurrent reads and writes. Since the performance would not be significantly affected by choosing either MySQL or PostgreSQL, another factor was taken into consideration: the acquisition of Sun Microsystems by Oracle. With this acquisition, the future of MySQL became an uncertainty. Either Oracle can stop developing MySQL, which would require the database management system to be changed again since MySQL would not follow up with improvements made on the other database management systems, or even if Oracle continue to invest in MySQL, only paying customers will be able to take advantage of the best features. Based on this, the choice for the database management system fell directly to PostgreSQL.

With regards to the implementation of the operations over the database, this was entirely made by using the *QtSQL* module provided by Qt Framework. This not only makes the implementation operating system independent, but it also turns it into DBMS independent. This is because the set of operations provided by *QtSQL* module are already available to be used with any of the possible DBMS to be integrated with Qt. Hence, the development process of this module consisted in two tasks: the transition of the old DBMS (the SQLite) into the new one (the PostgreSQL); and make the implementation of the operations over the database operating system independent. Since Qt already provides full support to integrate PostgreSQL into an application, the main part of this task was to abstract the implementation of the framework operations over the database. In the previous version, these operations were directly performed over the database (*SQLite*), because the *QtSQL* module was not being used.

Additionally, the new implementation also had to consider a relocation of functionalities, because the loading of data from the database is now performed by the thread that coordinates all the operations over data, while in the previous version this was done by the thread associated to the GUI. This means that either the thread that handles the *Framework Communication Protocol* or the thread that takes care of the *GUI*, will send a signal to the *Operation Manager* thread whenever there is an operation to be performed over the database. The operation manager is then responsible for issuing the correct task to be performed on this module. An example of this is illustrated in Figure 4.4 for the case of loading a previous session from database.

As referred before, the main type of information that is stored by this module is concerned with the sensor information loaded in a working session. This means that it also contains the representation of a sensor that is used by the other modules, for example the sensor data processor module.

In what concerns the insertion of data in the database, a *batch* agglomeration mechanism was already in use in the previous version, and which continues for this version, in order to achieve better performance. Hence, instead of executing an insertion operation at a time, for each data sample originated by a sensor, these are grouped into a transaction which contains insertion

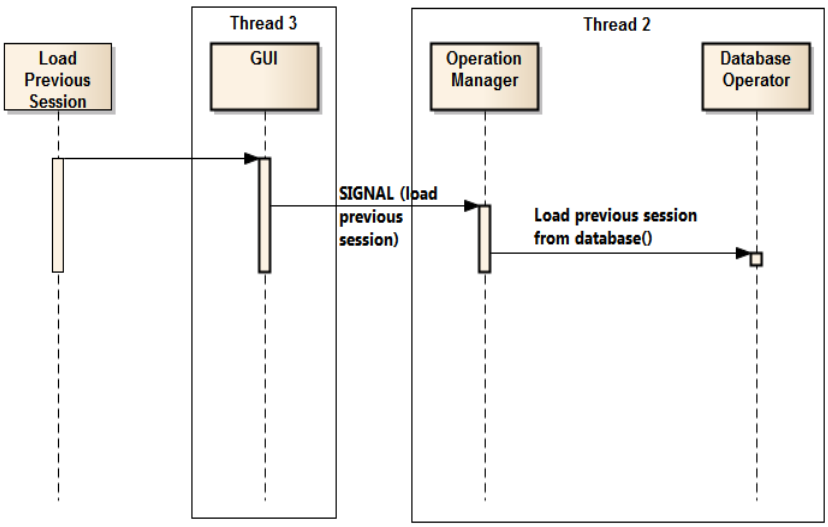


Figure 4.4: Example of database processing interactions between the GUI thread and the *Operation Manager* thread.

operations for several acquired samples. In the previous version of the framework, a timer is set to signal a time-out for every past second, which caused an operation to be launched to commits the current transaction. For the new version, this time-out was changed to happen every *5 ms*, to achieve better accuracy in storing the data in the database.

In summary, the benefits introduced to this module were concerned with changing the DBMS in use, restructuring the operations over the database to obtain platform and DBMS independence, and relocating its functionality both to relieve the GUI processing thread and to enforce a loose coupling and a high cohesion across the code.

4.5 XML Data Manager

In the previous version of the system, whenever the users wanted to change the number of stored sensors, they had to do it directly through the DBMS. Hence, there was the need for a system administrator (or anyone else who knew how to manage data in the database). Only this way it would be possible for the system to store data related to a new sensor or eliminate data of an existing one, since this was the only possible way to create the necessary information in the database to accommodate the desired changes.

In the new version that is hereby presented, and following the requirements that were stated by the team of end users, it was decided to have a mechanism that would allow the user to make some configurations without the need for any database knowledge. This would provide the team with the desired flexibility they require in a testing session for the sensor configuration. In accordance, it was decided to store this type information in XML files. These can contain several types of information such as: user preferences, namely if the last project should be loaded upon

system start-up or automatically saved on exit; the tuning parameters used in the car for a specific session, such as the camber angle for a wheel; the existing GUI elements associated to some session; the list of existing sensors, including those obtained from mathematical manipulations over other sensors; the implicit post-processing or data conversions of certain sensors, meaning the mathematical formulas that must be performed over the sensor data before storing in the database and display to the users. The latest only applies to sensors installed in the car. The ones composed of a mathematical manipulation do not go through this type of processing, as the mathematical formulas are applied to data that went already through this process.

The implementation of this module was performed using the Qt XML module. This module offers a set of operations to read and write data in the XML format. The choice for the XML representation was based on the desire for an easy and portable mechanism to transfer some kind of application definitions between the end users. That said, by using this approach the users can easily exchange any information contained in the files or even the files themselves between one another. Since XML is a user readable format, there is even the possibility for the user to interpret the content of some file and transmit to another user. In line with this, it is also easy to change the content of a file, even without the support of the framework. Assuming that the user understands the format chosen for storing the information, he can just open the file and edit it himself without much effort.

Besides this, it also abstracts the implementation for any operating system in which the framework should run, without imposing any restriction to its utilisation. All the XML files that are created to manage information are put inside a folder named *FstApp* which is created in the user directory. The creation, opening and closing of these files is managed by using Qt Framework *QFile* interface, a platform independent API to access files. When the user manipulates any of the configurations, with exception of changing the gauges/graphs (explained in subsection *Graphical Elements*), the system overwrites the corresponding XML file with the changes made. On start-up, the system is configured with the latest up to date configurations loaded from the corresponding files, with which the system is then prepared to work with.

Just as the previous described module, this module is based on the signal/slot mechanism provided by Qt. Thus, whenever the user enters some new configurations through the GUI, a signal is emitted from the GUI thread to the *Operation Manager* thread. The operation manager is then responsible for issuing the correct task to be performed on this module. This procedure is illustrated on Figure 4.5.

4.5.1 Sensors Management

The sensors are managed by means of an XML file containing information about all sensors. The name of this file is *sensors.xml*, and if it does not exist, the system will automatically create it on start-up. The file is loaded every time the system is launched, and the framework provides

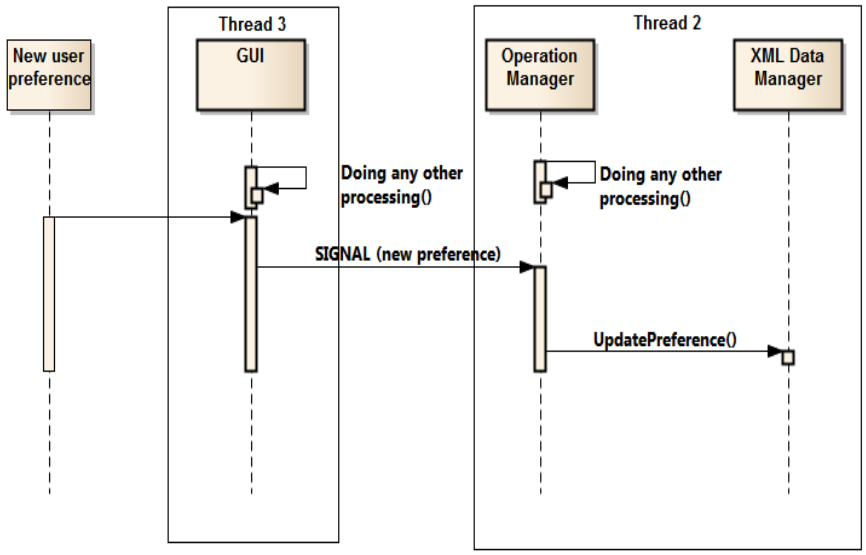


Figure 4.5: Sequence steps when setting a new user preference.

a feature, via its graphical interface, for the users to change any type of information related to the sensors. In this case, this means the sensor's name, type, identifier, and corresponding measurements and sample precision (in bytes), that will be received for each sensor.

As an example of the sensors management procedure that is carried out, the expected measurements included in a message from the GPS sensor, related to the position of the mobile station, are the latitude, longitude, altitude, speed, and a timestamp. These fields have to be configured by the user and included in the corresponding GPS sensor parametrization. An example of an XML file with a GPS sensor configuration is shown on Figure 4.6. The sensor management interface is illustrated on Figure 4.14(a) of Section *4.7*.

This file is also used to store the configuration of the sensors that represent mathematical manipulations of other sensors.

```
<?xml version="1.0" encoding="UTF-8"?>
<sensors>   Sensor ID |   Sensor Name          | Sensor Type
    <sensor id="70" name="coordinates_0" type="777">
        <latitude>1</latitude>
        <longitude>1</longitude>
        <altitude>1</altitude>
        <speed>1</speed>
        <timestamp>1</timestamp>
    </sensor>
</sensors>
```

Size (in bytes) of the sensor argument

Figure 4.6: XML file with a GPS sensor configuration.

4.5.2 User Preferences

For the user preferences, a simple XML file with the name *user_preferences.xml* is created the first time the application is launched, or every time the system cannot find it. At the moment, two types of personal preferences are allowed: automatically save changes made to the GUI when exiting the application; and load last working project, on start-up. The users are able to change these preferences through the GUI, and the preference's file is overridden with the new preferences values. The effects of the changes are instantaneous. The representation of these preferences, when loaded by the application, are two binary variables, which indicate if the user wants or not each of these preferences to be set. An example of an XML file containing a combination of these preferences is depicted on Figure 4.7. In the illustrated example, the user does not want the application to automatically save changes made to the GUI, neither wants the application to load the last working project on start-up. The session number field represents the identifier given to a session when one is created. This is generated serially and assigned to every new session. It is set to *0* in the first time the file is created, or when the user does not want a previous session to be automatically loaded, because a valid session number must be an integer greater than 0.

```
<?xml version='1.0' encoding='UTF-8'?>
<configurations>
 <configuration saveOnExit="false"/>
 <configuration loadPreviousProject="false" session="0"/>
</configurations>
```

Figure 4.7: User preferences configuration file.

4.5.3 Car Tuning Parameters

The car tuning used for a practice session, is a static list of parameters provided by the users that relate to the conditions of some properties of the vehicle upon starting a specific session. These parameters represent the state of some configurations in the car, such as the tire pressure and the angle that each wheel makes with the vertical or longitudinal axis of the vehicle, that are static during a practice session. These are also stored in a simple XML file, which is created every time it is missing. The file is composed of pairs of elements, where the first element is the name of the parameter and the second the corresponding value. The values stored in the file are always the last configuration set up by the user. To enable loading a previous logged session with the parameters associated to it when it was created, each session is also associated with a configuration of these parameters in the database, every time a new one is created. The corresponding parameter values are loaded each time a session is loaded. These parameters are displayed to the users, alongside the sensor information, in the GUI. The latest provides a mechanism for the users to change the parameters at they will. Upon the change, the corresponding file is updated

with the new information. In case the user changes a session parameter for a specific session after it has been created, the database is also updated with the new set of parameters.

An example of this is illustrated in the file depicted in Figure 4.8. The rear tag section, which is not stretched, contains a set of similar parameters as for the front tag, each corresponding to the front and rear sections of the car.

```
<?xml version="1.0" encoding="UTF-8"?>
<vehicle>
    <front>
        <default name="fArb" value="7"/>
        <default name="fHsc" value="22"/>
        <default name="fHsr" value="20"/>
        <default name="fLsc" value="20"/>
        <default name="fLsr" value="20"/>
        <default name="fCamberRight" value="0.5"/>
        <default name="fCasterRight" value="0.6"/>
        <default name="fKingpinRight" value="6"/>
        <default name="fToeRight" value="7"/>
        <default name="fTirePRight" value="1"/>
        <default name="fCamberLeft" value="0.5"/>
        <default name="fCasterLeft" value="0.6"/>
        <default name="fKingpinLeft" value="6"/>
        <default name="fToeLeft" value="7"/>
        <default name="fTirePLeft" value="1"/>
    </front>
    <rear>
</vehicle>
```

Arb = Anti-roll bar
Hsc = High speed compression
Hsr = High speed rebound
Lsc = Low speed compression
Lsr = Low speed rebound
Control the speed of compression/rebound of the damps
Camber = Angle between the vertical axis of the wheels used for steering, and the vertical axis of the vehicle
Caster = Angular displacement from the vertical axis of the suspension of a steered wheel car, measured in the longitudinal direction
Kingpin = Relative angle between the kingpin and the vertical axis
Toe = Angle that each wheel makes with the longitudinal axis of the vehicle
TireP = Tire pressure

Figure 4.8: Car tuning parameters configuration file.

4.5.4 Graphical Elements

The GUI elements are divided in two categories: graphs and gauges. These are stored in separate XML files, dedicated to the corresponding category. The files are created every time the system notices they do not exist for a working session. The files are named as *sessionID_graphs.xml* and *sessionID_gauges.xml*, where sessionID is the identifier of the current working session. A pair of files is created and stored for each new existing working session.

Although the user is able to add and remove graphs and gauges through the GUI, the changes are not automatically written to the corresponding files. Instead, the user is providing with a saving functionality that he can use when he wants to save the setup corresponding to the current graphical display. This functionality is enabled or disabled based on changes made to the current graphical display. Thus, when the user adds or removes a graphical element from display, he can use this functionality to store the new changes.

4.5.5 Sensor Post-Processing

The last information that is managed using an XML file is concerned with the definition of the types of the sensors and of the corresponding post-processing manipulations. The *types* of

the sensors are integer values associated to mathematical operations that need to be performed in order to store the correct value of the sensor data in the database and display it to the users. Through this configuration, the user is able to define the units associated to the processing formula for a given sensor, which is always displayed in the GUI as part of the identification of the sensor, alongside the sensor's name. The user can change the type associated to each sensor when configuring such sensor in the system. Separately, the formulas associated to each type can also be configured through the GUI. The effects of the changes take place the next time new information for that sensor is received either through the network, or when loading data from a file.

4.6 Sensor Data Processor

This section focuses on the set of operations that are performed in the *pit station* over the sensor data. This data can either belong to a *single* sensor, meaning one installed in the car whose data is not dependent on other sensor data, or to a *combinational* sensor, whose data is defined as a composition of mathematical manipulations over *single* sensors data.

For the new version of the framework, one of the main requisites enunciated by the users was the possibility to perform user defined operations over the sensor data. Since the option of implementing a processing module capable of handling the needed mathematical manipulation was not viable, a library was considered for this purpose. The requirements that needed to be satisfied by the library were: the possibility to be executed under all the required operating systems; and offer enough computational performance to handle the rate at which data arrives at the system.

Based on this, some mathematical parsers were considered: *muParserSSE*, *muParser*, *fParser*, *MTParser*, and *MathPresso*. Among this set of parsers, and according to the benchmarks found in [26], the fastest considered parsers are *muParserSSE* [26] and *Math Presso* [24]. This is mainly due to an integrated built-in Just-In-Time (JIT) compiler, which transforms expressions into machine code at runtime. However, both are tailored for Windows, so they were disregarded. Among the others, *muParser* was the one which best combined the requirements mentioned above, together with a better support documentation. Hence, both processing types, for *single* and *combinational* sensors, make use of the *muParser* [27] library. This library is based on a convenient structure providing a set of methods for executing mathematical manipulations. The structure provided by *muParser* is the *Parser*. This handles the setup of the mathematical environment, and consequent evaluation of the expressions. Setting up this environment requires the definition of the necessary variables to be used as input to the formula, together with the constants, such as PI.

4.6.1 Single Sensor Manipulation

In the previous version of the software framework, all the data originated from the sensors had to go through a post-processing transformation, before it could be displayed to the users. Having this post-processing implemented by software allows a fast analysis and faster decisions concerning with the vehicle evolution to be made. This process was directly implemented in the pit station software, since it is where the most powerful computational power lies. These operations mainly consist on the application of mathematical formulas over the acquired data from the sensors. Since not every dataset requires the same operations to be performed, a distinction has to be individually taken for each single sensor. Unfortunately, in the previous version of the framework, these operation definitions were static. This means that the set of possible operations to be performed on the gathered data was limited and hard coded in the application. Hence, every time a new sensor was introduced in the car, someone had to go trough the code and add the new post-processing definition to application.

Since this is a very strict policy, when compared to the high degree of flexibility that is now required for adding and removing sensors from the application, a new approach was required to deal with this aspect for the current version of the software. This is based on the XML mechanism described earlier. Hence, the users may change the mathematical operations associated to the sensors, and these are stored in the corresponding configuration file. When the application is launched, the set of sensors that is present in the users' configured list is created. Upon creation, the sensors are then associated with the corresponding mathematical operation, being identified by a specific sensor type. If no type is defined for a given sensor, no operation will be performed over the gathered data for that sensor. Once the sensor data is received from the mobile station or read from a data file, it goes through the process of executing the corresponding operation for each data sample, whose result is subsequently stored in the database and displayed to the users. In the case of a working setup in *real-time* mode, a signal is emitted from the *Framework Communication Protocol* to the *Operation Manager*, which will issue the corresponding processing task to be performed in this module. After this process is completed (and if in *real-time* mode) a signal is emitted to the *Graphical User Interface*, with the post-processed data received from the *mobile station*.

4.6.2 Combinational Sensor Manipulation

The values corresponding to the *combinational* sensors are computed by evaluating the associated mathematical expression using the samples gathered from other sensors. Both the expression, and the associated list of input sensors are retrieved from the sensors' XML file. These set of sensors do not have any type associated with them. Instead, the values corresponding to the *combinational* sensors are computed either from the input values loaded, from the database or from a file. Alternatively, these can be also computed in real time when data is being received

from the mobile station through the data received for the *single* sensors that will compose the *combinational* one.

Starting with the case when the information is loaded from the database, the *combinational* sensors are created after the *single* sensors have been created and their information loaded. If data is missing for at least one sensor, the expression cannot be evaluated because no values for that sensor can be given as input for the formula evaluation, and no data is displayed for such *combinational* sensor. In a normal operation, data will be available for all the *single* sensors. Hence, data is loaded from the *single* sensors that are part of the *combinational* sensor, and the corresponding mathematical expression can be evaluated. This would be the simplest case, if all *single* sensors had the same sampling instants.

Unfortunately, this is not what happens in practice, when the values are gathered from real sensors. Therefore, it is necessary to create a single set of timestamps, composed of the intersection of the timestamps of the *single* sensors, and then interpolate the data of each sensor, in order to get the values corresponding to the missing timestamps.

For this purpose, an interpolation algorithm had to be implemented in order to interpret the data. The only requirement to be satisfied by the algorithm was to provide a reasonable approximation to the sensor signals. The considered algorithms for this purpose were the *Nearest-neighbour*[16, 23], the *Linear*[16, 23], and the *Lagrange Polynomial* [21, 22]. Among this set, the one providing the level of accuracy required is the *Lagrange Polynomial*, detailed in Appendix B, and hence it was the one that was chosen for the purpose. The first algorithm, the *Nearest-neighbour*, finds the nearest point in a given data set and assign the same value to the needed interpolation point. The second, *Linear*, connects the data points in a given set with straight lines, and finds the value of the needed interpolation point in these straight lines. Thus, it can be easily observed that both algorithms do not satisfy the required accuracy for sensor signal interpolation. The first approximates any point inside an interval range to the closest integer, while the second assumes that the values of the sensors varies constantly between two points.

Nevertheless, although the chosen algorithm satisfies the degree of precision needed for the project, it also has a drawback: It is not tailored to deal with a dynamic dataset. Hence, the adopted approach was an adaptation of the original *Lagrange Polynomial* algorithm to overcome this problem.

The *Lagrange Polynomial* algorithm is designed to create a polynomial of degree *N*, from a set of *N+1* samples available. The created polynomial goes through all the points that are part of the *N+1* samples, and approximates any point that is not part of this dataset. The only problem arises if this data set is dynamic, meaning that points can be added to it. The algorithm is not designed to deal with changes in the dataset, hence a new polynomial has to be created every time a new sample is added to the dataset. Based on this, the adopted implementation always considers *four* points, which are used to build a polynomial of at most degree *3*. After the polynomial is built, the

value corresponding to a point that is not in the dataset can be found, by evaluating the resulting polynomial using that point. The degree of *3* was chosen in agreement with the end users, to accomplish a reasonable degree of accuracy for the measurements that are being shown in the analysis.

After this algorithm is executed to obtain the values for the missing interpolated points, the overall set of data points (overlapped and interpolated) is evaluated using *muParser* to obtain the final result. At this point, the obtained values are ready to be stored in the database, as well as displayed to the user. From this moment on, every time the current session is loaded from the database, the data for the *combinational* sensors is loaded, the same way it is for any *single* sensor, without the need for this type of processing again.

The case for a real-time data acquisition mode is almost similar. Instead of interpolating and evaluating a static set of data points for the *single* sensors included in the *combinational*, this is performed with the data that is received from the mobile station. This data corresponds to measurements of single sensors, and hence it is necessary to check if there is sufficient data received from the *single* sensors to perform the interpolation of the data, as well as the evaluation of the mathematical operation. Hence, every time a new value is received for some sensor, a test is performed on the data points of the remaining *single* sensors included in the *combinational* one, to check if the interpolation can be performed. In this case, this means that the interpolation and the evaluation start as soon as there is a non-empty intersection in the time interval of points received from the mobile station. Just like the data from the *single* sensors, when the result of the evaluation of the mathematical expression is obtained for the *combinational* sensor, its data is immediately stored into the database, so that it can also be retrieved in a posterior loading of the current working session.

To display the information of a *combinational* sensor in *real-time* mode, the same mechanism that was used for the *single* sensors is adopted, as described in the previous section. To illustrate the workflow associated to an interpolation, Figure 4.9 illustrates the creation of the combinational data for sensor *C*, obtained from an interpolation of data from sensors *A* and *B*, received in real-time. The numbers in the figure, represent the timestamps received for each sensor, and the processing applied to the corresponding received sample. For sensor A and B this means the post-processing applied to their corresponding values. For sensor C, this means finding the intersection of the timestamps and applying the algorithm described to find the missing values, as well as the evaluation of the associated mathematical expression.

It is relevant to mention that the interpolation operation is also used for another feature provided by the framework. This is associated with the creation of the dataset to be displayed in one specific type of graph, namely the *Scatter Plot* presented on section *4.8.1*, which includes a sensor dataset for the *x* axis and another sensor dataset for the *y* axis.

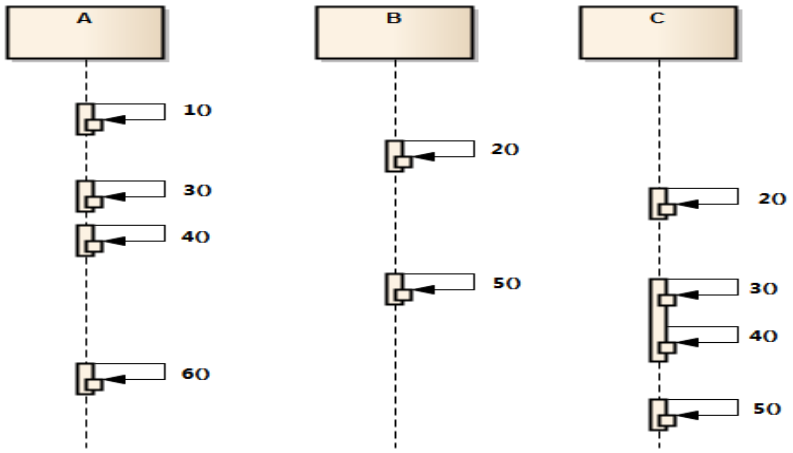


Figure 4.9: Interpolation of the data used to compute the values of a sensor C, composed of a mathematical manipulation of sensors A and B.

4.7 Operation Manager

This module coordinates all the operations over the data. It manages all the modules, with exception to the *GUI* and *Framework Communication Protocol* modules, which live in their dedicated threads. Its main task is to act as a broker, between these two modules and all the other modules.

This module implements a set of slots that are executed in response to user interaction over the GUI and to process newly received sensor data from the mobile station. The communication between the two above mentioned modules and the *Operation Manager* is made through the signal/slot mechanism provided by Qt Framework. The GUI and the communication thread emit signals whenever some operation must be performed, and a dedicated slot is executed in the *Operation Manager*.

On the other hand, since all the other modules are controlled by this module, and hence live in the same thread, the *Operation Manager* just issues the correct operation to be performed on some module, when a signal is received from one of the two above mentioned modules. In case any information needs to be displayed, as part of the requested operation signalled by the GUI, this module emits a signal to the GUI module, which contains the necessary information to be displayed.

Thus, from the point of view of the GUI module, this is the entity from which it gets the required information to be displayed to the users. From the point of view of the *Framework Communication Protocol*, this is the entity that starts or ends the communication between both stations and processes the relevant data that is received.

4.8 Graphical User Interface

The implementation of the pit station graphical user interface was conducted with permanent contact with the end users team, to best adapt the interface to their working methodologies and create the most comfortable environment tailored to their analysis. The graphical user interface, is the module that directly enables the users to perform a useful analysis and improve their productivity during the preparation for competitions. All the data is displayed through the GUI, which provides the users with several mechanisms to manipulate and visualise it. The previous version of this module lacked of the most relevant characteristics to provide these benefits to the extent of user analysis, and hence it was redesigned and improved to create better working conditions.

All the GUI elements are supported by the Qt Framework and the Qwt Library. The latest provides useful classes for the technical graphical elements. These count with graphs and gauges, which directly exhibits the information from the sensors and user manipulations, and a timeline mechanism to browse session data. Qt deals with the overall management of the resources for the whole application, while providing the rest of the elements used in the GUI, such as *Tab*, *Tree*, and *Button* widgets.

Just like the other modules, the GUI was required to have a platform independent implementation. As described earlier, the Qt Framework, as well as Qwt Library, fulfil these requirements, and hence provide cross-platform support.

4.8.1 Graphical Layout

Besides the significant improvements that were introduced in the back-end part of the application (as described in the previous sections), the main set of improvements can be also seen in the front-end, the part which the users directly interact with. When designing such a user oriented system, there are essential requirements that much be carefully gathered, such as the user needs in terms of functionality and visual appearance, and the user behaviour. In the previous version of the system, some key factors were missing, which inhibit the users from using the system.

In what concerns the user interaction, the main differences start at the launch phase of the system. In the previous version, the user is presented with a screen where he is able to chose among three types of projects: *real-time data acquisition*, *previous session data analysis*, and *upload data from USB pen drive*. According to the new requisites, there should be only two types of projects to chose: *real-time data acquisition* and *offline data analysis*. The *offline data analysis* should include both load data from file or from the database. Furthermore, the read of the data from a file, or the fetch of the data from the database should be more user friendly, just as in other everyday applications the users are used to work with: a simple file menu, followed by a load file or open previous session sub-menu entry, directing the user to the corresponding windows dialogue which enable the user to perform the desired task. The initial screen is illustrated on Figure 4.10,

which replaces the wizard approach previously implemented and depicted on Figure 2.5. The screen is divided into three main zones, as depicted in the picture, each of these described later in this chapter.

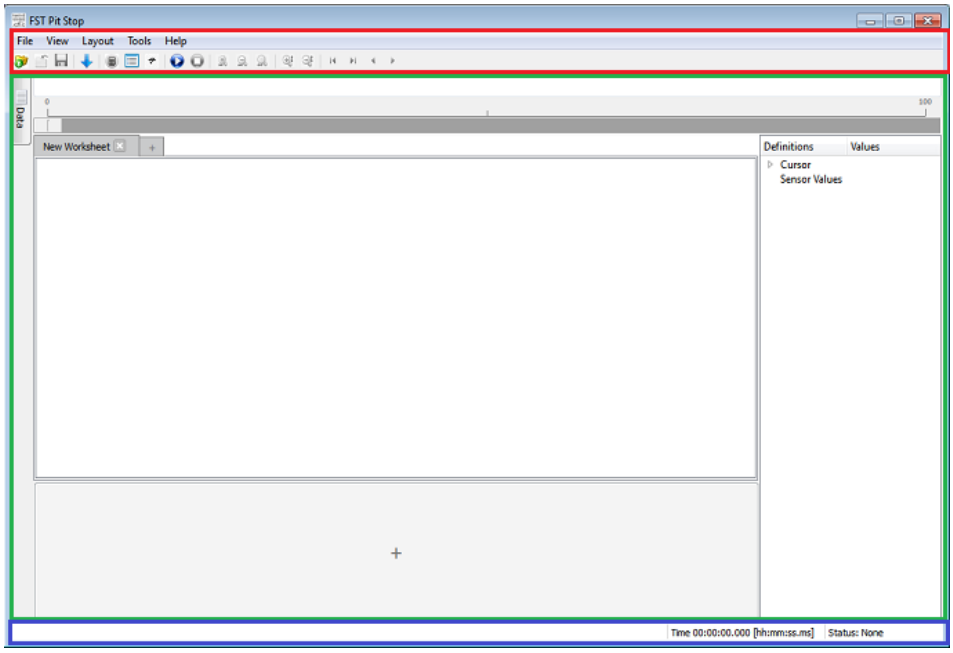


Figure 4.10: Main screen of the application, with no project initiated.

In the previous version of the system, the user is faced with the perception that each type of project originates a different application right from the beginning. There is a specific sequence of steps for each type of project, which the user must follow before reaching the main user interface, which is different for each type of project. The end users did not appreciate this initialisation method and stated their interest in that both (*real-time data acquisition* and *offline data analysis*) projects should have a unique and unified interface. This interface should also focus on flexibility to manage what users want to see on the screen. This means that the wizard approach, with the mandatory sequence of steps in the previous version was discontinued. The total control of the application should be given to the user through the single and unified GUI and the result is depicted in Figure 4.10.

Concerning the *real-time data acquisition* project of the previous version of the application, one point that more seriously demonstrates its lack of integration is the random disposal of the various types of graphical widgets. In one hand, these widgets can be visualised in different windows, as if they not even belong to the same application, requiring a lot of effort from the user to organise them himself and maintain a global view of the system. On the other hand the users may also view these widgets all placed in a single window, but with no possibility to organise them at their will. In this screen, the non-integrated and not flexible environment provided by the previous version is even more revealed since the widgets are displaced inside the same window, without being

adapted to it, both in terms of disposition and dimensions. Although there exists the possibility to have widgets displayed in tabs, the user is forced to have one tab for each widget, thus preventing one of the main goals, namely, to easily compare data on several graphs.

Hence, the criteria that were taken in this new verison of the project to overcome the drawbacks of its GUI were:

- Build a single windowed interface common to both types of projects;
- Design the graphical layout taking into account the best way to accomodate and organise the unified set of functionalities required by the end users;
- Provide an intuitive and quick way to compare different data;
- Provide the end users wht the desired flexibility and adaptation of the graphical user interface to various user profiles.

An example of the two types of screens of the previous software framework as described in this section are shown in Figure 4.11 and 4.12. These screens can be compared to the new user interface prototype for the new version, as illustrated in Figure 4.15.

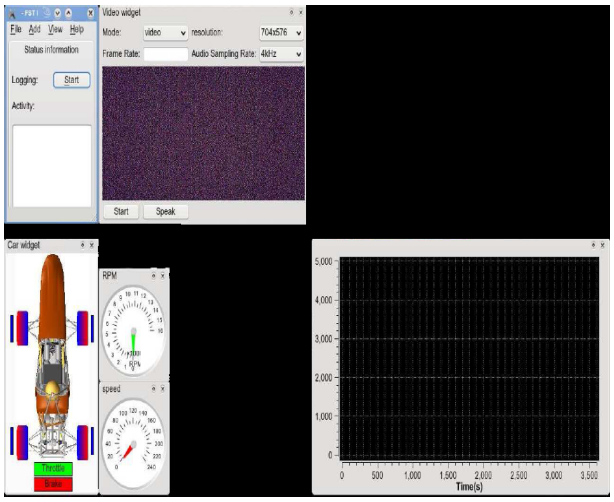


Figure 4.11: Previous FST academic solution: Main window with the real-time project graphical layout [7].

In what regards to the *offline data analysis* option of the previous application, there are also some relevant negative factors that the users did not like. First, during the execution of the application, the user has no option to change the session he is currently analysing. This means that if he wants to work on another previously stored session, he would have to close the application, start it again and chose the new session to work on. The users are also forced to have only one single type of graph, which clearly do not satisfy all their needs. Additionally, there is no direct way of identifying a sensor value on the graph. The only possibility is to look at the graph, look at the scale displayed by its side, and try to match the sensor value to one of the scale values. Furthermore, when showing multiple sensor data in a graph there is no direct way of identifying

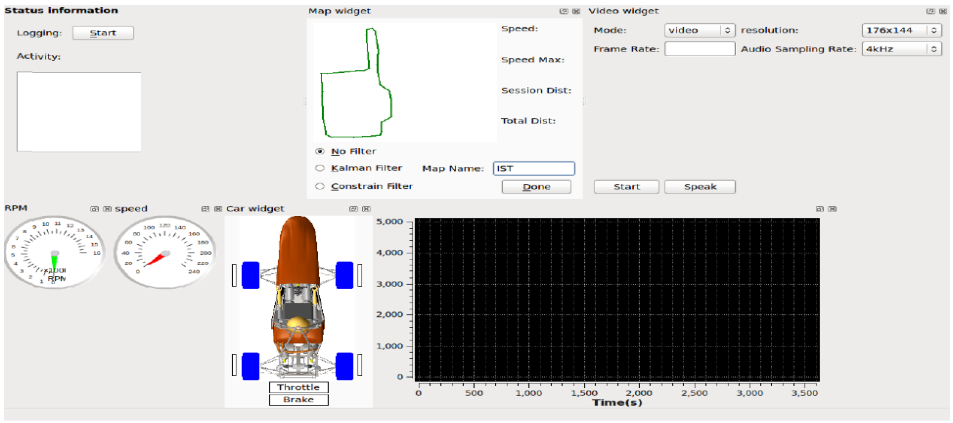


Figure 4.12: Global window of the previous version of the framework, with docked widgets.

each sensor line by looking only into the graph. This is because there is no sensor identification in the graph. To identify which sensors are displayed, the user has to check in the list of all sensors names displayed below the graph alongside their corresponding colours and try to match one with any of the lines in the graph. Figure 4.13 presents a screenshot where this problem is illustrated.

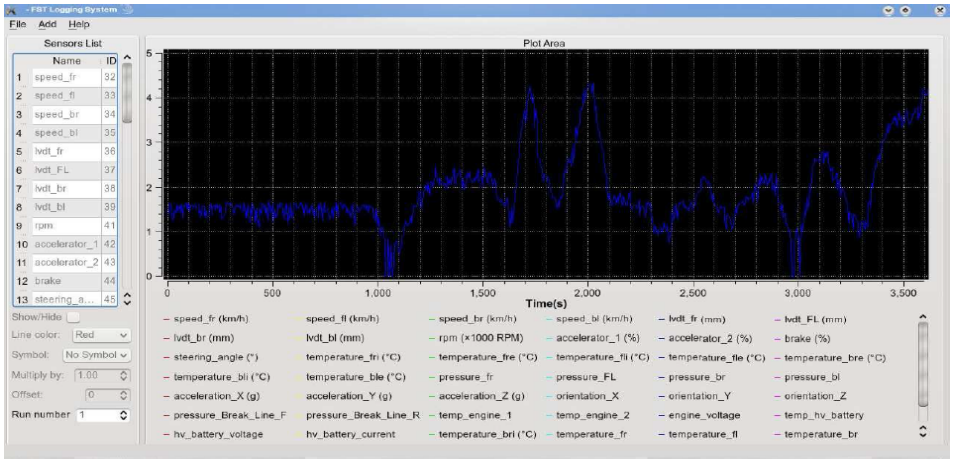


Figure 4.13: Offline data analysis GUI from the previous version of the framework.

To overcome the referred problems of provided by the graphical user interface of the previous software framework, the following criteria were taken into account in the design of the new graphical user interface:

- Enable multiple session loading in a single application execution, provided by a user friendly mechanism such as: a file menu, followed by a load session menu which prompts the user to chose the session he wants to load. In case the user is working on an already opened session, a verification is performed to check if the user wants to save the work done upto the moment in the previous session. In case the new session to be loaded is the one the user is currently working on, nothing happens;

- Provide the user with several types of graphs displayed on the screen, to facilitate comparisions of data;
- Provide better information of the data that is presented in a graph, by displaying the correspondinf value when the user clicks in a point of a graph;
- Provide better identification of the data displayed in a graph, by labeling each function with the associated sensor and color;
- Possibility to zooming, both in and out, and browsing of session data.

The above criteria, alongside the others that were described for the *real-time* project, formed the main set of criteria that were taken into account in the development of a single and unified graphical user interface, which suits both types of projects. The proposed improvements can be observed in Figure 4.15, which represents the main screen of the application, common to both types of projects.

From this figure one can identify the main difference from this version to the previous one: the existence of one single interface for the two types of projects. This reveals the overall integration of all the available features, creating the mental idea of a unique global system. This unique interface also provides the users with much more flexibility than the previous one, allowing the users to manage the interface according to their needs.

The GUI is based on a set of operations based on the signal/slot mechanism provided by the Qt Framework, which deals with the necessary updates to the information displayed on screen. When the user requests an operation, a signal is emitted to the *Operation Manager*, so that it executes the required action and generates the information to be presented to the user by the GUI. Upon the generation of this information, the *Operation Manager* emits a signal to the GUI and a corresponding slot is activated, to handle the representation of the received information. Hence, the GUI module is structured as a set of slots, that respond to signals that are generated either by user input or by the *Operation Manager*, and a set of signals to request information from the *Operation Manager*.

The interface is divided into three main zones, as marked in Figure 4.10: an*upper zone* composed of the menu and a tool bar identified by the red rectangle in the figure, a *central zone* where the relevant information is displayed identified by the green rectangle, and a *lower zone* composed of a status bar to display state information for the data being presented, identified by the blue rectangle.

The most relevant elements that are displayed in the interface (namely the graphs, gauges and the slider) display information from the sensors and help to browse the data in a given session. The user is able to create and remove these elements from the interface according to his needs. The rest of this section describes, in more detail, each of these defined zones, specifying each of their components (see Figure 4.10):

- ***Upper zone*** **:** This is where the user makes actions related to the most common tasks

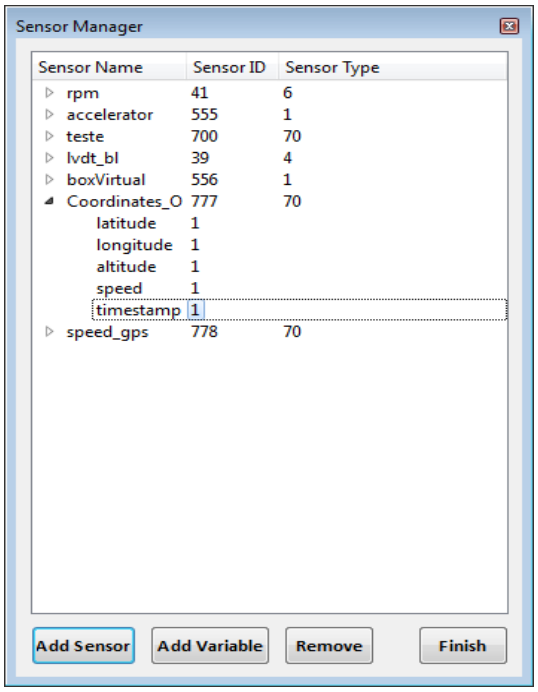
associated to system configurations, project and file management. This zone is composed by a *menu bar* and a *tool bar*. The *menu bar* is composed by a *menu* and a *submenu* elements, provided by Qt Framework. The operation of these elements is based on the signal/slot mechanism already described. When the user selects a specific menu, the selected element emits a signal to assert the user action. In accordance, a set of slots were implemented to define the corresponding application behaviour for each of the menu items. The same happens for the *tool bar*. This contains a set of buttons which emit a signal every time the users clicks. Again, a set of slots were implemented to define a specific behaviour for each of these buttons. Many of the elements of both the *menu bar* and *tool bar* have the same behaviour and thus share the same slot. Among the set of defined operations, the most relevant are: loading data from a file, loading previous logged data, saving data in a Comma-Separated Values (CSV) file, starting/stopping a live session, managing the sensors, configuring the user preferences, managing the sensor types, managing mathematical manipulations, saving the state of the GUI, etc. The implemented interface for configuring the sensors and their types is illustrated in Figure 4.14. This is one of the most relevant features introduced in the new framework.

- ***Central zone*** **:** This is the most important zone, since it is where the relevant information is presented to the users. This zone is responsible for providing the users with the necessary information they need during a working session. Figure 4.15 shows a division of this zone into five sub-zones. Each sub-zone has its specific purpose.

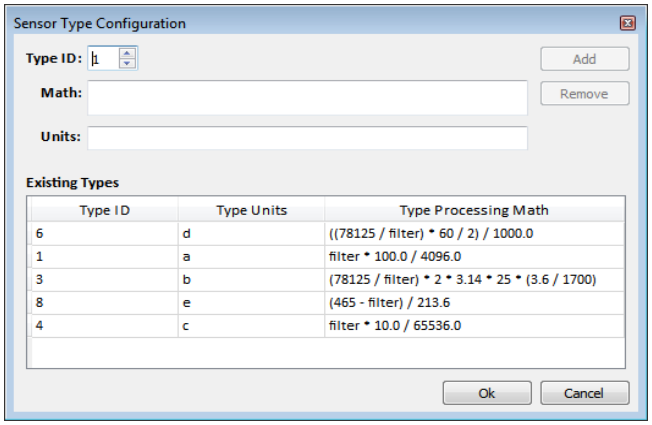
  The topmost sub-zone, shown in red, is responsible for displaying the session details to the user. This comprehends information, displayed as text, such as the name of the session, the identifier, date and time of creation, car driver, weather conditions and the location of the session. This text colour is set to black whenever the layout is saved. When changes occur, the text colour is set to red to signalise that there was an unsaved change.

  The second sub-zone, marked in green, is used for the purpose of browsing through the data of a given session. This is composed by a slider, which is one of the used elements from the Qwt Library. Since Qwt is tailored to be integrated with the Qt Framework, the operation of its elements is also based on the Qt signal/slot mechanism. Therefore, the slider emits a signal every time the user changes its value. Hence, this signal must be connected to a dedicated slot which, gathers essential information and performs the necessary updates in the relevant graphical elements.

  The zone in blue is one of the most relevant in the developed interface, as it displays the graphs and the GPS item. This zone is composed by a tab widget, provided by Qt Framework, where items can be added. The user is also able to change the name of the tabs. The user also controls the elements by means of a *context menu*, provided upon a mouse right-button clicking in this zone. This menu is shown in Figure 4.16(a). With this menu, the



(a) Management of the existing sensors. This is where the user can add and remove sensors from analysis.



(b) Management of the sensor types. This is where the user can define the mathematical post-processing formulas associated with the sensors.

Figure 4.14: Graphical interface of two functionalities provided by the menu and tool bar of the application screen.

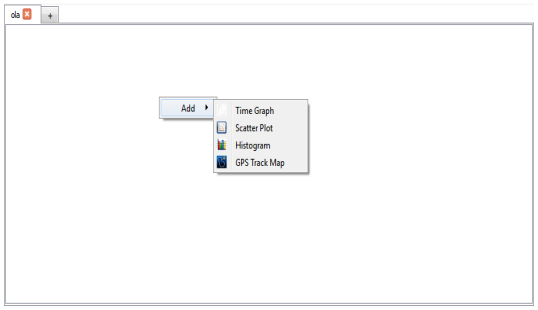
user is able to control what is displayed in each tab, by adding or removing elements. Each tab can accommodate one of three types of graphs, namely *time graph*, *scatter plot*, *histogram*, and it can also accommodate a *track map*. However, along the system specification the end-users stated the need to be have more than one time-graph per tab. As a consequence, this is the only type of graph that the user is able to add more than once in a given

Figure 4.15: Central zone of the interface, divided by its functionalities.

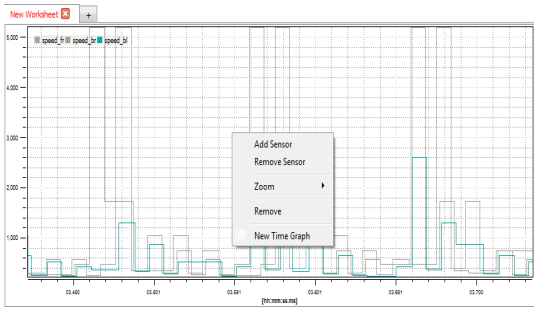
tab. Upon adding a specific type of graph (excluding the *track map*), the user is prompted with the list of sensors to chose which should be associated with the graph. As soon as the user adds a new graph, a set of new possible functionalities arise. These can be found through the graph context menu. In case of a *time graph* context menu, shown in Figure 4.16(b), this allows the user to zoom in and out the time range to be displayed, to add and remove sensors from the graph analysis, and to add or remove another graph in the same tab. For the other types of graphs, this menu only allows the user to remove the item from analysis. The graphs also have a legend, which identifies the set of sensors associated to the data displayed in it graph. From Figure 4.16(b), one can see the identification of each sensor with a square beside it filled with the colour associated to that particular sensor. The colours are associated with the sensors in a round-robin fashion by using the set of colours provided by Qt. At the moment, this set is composed by *148* colours, which fully satisfies the users requirements, since they have stated the need for at most *20* colours.

For the case of a GPS *track map*, a particular context menu as well as the new *GPS track map* graphical interface, illustrated in Figure 4.16(c), is displayed. This enables the user to perform operations on the map, such as applying filters to data, saving or loading a map, storing map information to a file (PDF, KML or CSV), removing the item from analysis, etc. The *track map* enables the user to analyse a sensor value when this is being associated to a position on the track. It was redesigned to conform to the new graphical user interface layout.

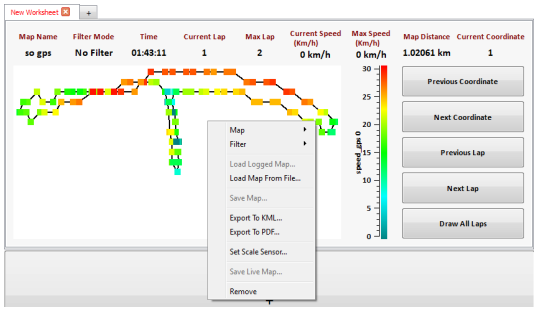
Whenever the user changes any information related to the tabs, the state of the GUI is marked as being changed. As it was already mentioned, upon a change, the topmost zone displaying the session information is set to red, and the user is able to save the state of



(a) Context menu when adding a graph.



(b) Context menu of a *time graph*.



(c) Context menu of a *track map*.

Figure 4.16: Tab, graph and track map context menus, allowing the user to manage the information displayed.

the GUI. In case of a saving operation, the information related to the content of the tab is stored in the session graphs XML file, as previously described. The information accommodates data related to the graphs, such as their type and their sensors, together with the identification of the tabs in which these are displayed.

When a *time graph* or *scatter plot* is displayed, the user is able to select a specific sample to further analyse information related to that particular sample. In the time-graph case, a vertical marker is displayed in the graph, as can be seen in Figure 4.15, with a cross-marker over the existing curves to identify the selected sample. For the scatter-plot, only a cross marker is used, which identifies the selected sample. This functionality, alongside with the zooming in or out functionality, are linked across tabs. This means that if one of these operations is performed in a graph on some specific tab, the results are propagated across the other remaining tabs. For the particular case of selecting a specific sample or point in time of a graph, the output is also propagated to the gauges (described later for the *brown zone*).

Again, all of these functionalities are performed based on the signal/slot mechanism. These elements, provided by the Qwt library, define dedicated signals that are emitted upon the user interaction with the graph. These signals are issued based on the user decisions

and are processed by dedicated slots, which request some information from the *Operation Manager* (if necessary), and update the state of the GUI.

The *brown zone* is where the other types of widgets (the *gauges*) are displayed. The *gauges* only show information for a specific point in time. This means that, unlike a *time graph* where the user is able to see all the values for the overall time of a session, these elements only show the value for a particular sample. Hence, when the user selects a given point in time in a graph, the gauge is updated according to the information for the selected sample. The gauges requested by the users (illustrated on figure 4.15) were:

1. *Bar*: displays the value corresponding to the measurement of a sensor for a given time instant. It is a scale which fills up and down, based on the measured values of a given sensor.
2. *Steering Wheel*: measures the value of a sensor in terms of the steering wheel angular measurement. Hence, it is most suitable for measuring the vehicle steering wheel angle during the driving.
3. *Dial*: another type of scale, witht the form of a conventional vehicle manometer. It is most tailored to measure the engine RPM and the speed of the vehicle.
4. *Numeric Gauge*: label with the value of a given sensor for a specific time instant.
5. *Status Light*: a simple coloured bar, displaying a colour based on a sensor value, used for the evaluation of good and bad state conditions, of a specific sensor. For this type of gauge, the user defines limits for good and bad values of the sensor, and the gauge displays green and red lights accordingly.
6. *Numeric List*: used to display values of several sensors, chosen by the user, for a specific instant.

The gauges can be added and removed by means of a button available on this zone. A context menu raises when the user clicks the button, which gives the option to add any type of these elements. When adding, the user is prompted with the list of sensors, in order to chose the one(s) he wants to be associated with the gauge. If a gauge has been added, the user can remove it by mouse right-clicking on the widget.

A already mentioned, these elements are updated when the user selects a particular point in time to analyse. Hence, the updates on these elements are performed through a slot connected to the signal which is emitted every time the user selects a point in a graph.

When the user changes the number of existing gauges, the state of the GUI changes. Upon a change, the red sub-zone content is set to red, and the user is able to save the state of the GUI. If the user chooses to save the current state of the GUI, the information related to the existing gauges is stored in the session gauges XML file, as it was previously described. The information accommodates all the data related to the gauges, such as their type and their sensors.

The last sub-zone, marked in yellow, is a zone specifically designed to display information about the sensors under observation. This zone is composed by a tree, which displays information related to a specific moment in time, together with the measurements of the sensors for that moment. For a sensor that has a colour already associated with it, its colour is also displayed as a small icon next to the identification of the sensor. The data in this zone is updated every time the user selects a specific sample from a graph, or in the case of a *real-time* project, once for each new data sample that is received. These updates are executed by a dedicated slot that defines the update behaviour when selecting a specific sample, thus updating the graphical display according to information related to that sample.

- ***Lower zone*** **:** This zone has the only purpose of displaying some information to the user. No user input occurs in this zone. It is used to display to the user, information about the current working session. Hence, the users can easily visualise some relevant information not related to the sensors but with the working project. It is populated with two types of information, required by the users to be always present during the analysis. This includes, the current selected *time* on a particular session, and the *type* of project the user is currently working on. The first displays the time associated to the sample the user has selected. When working in *real-time* mode, this field displays the most updated time instant that was received from the mobile station. The second, *type* of project, only displays if the current project is an *offline* or *real-time* project. This field can display *None*, if no project is loaded, *Offline* when users are working *offline*, and *Live* when working in *real-time*.
  Both these fields are changed after the user interaction has taken place. The *time* field is changed when the user clicks on a given point in time, to analyse information on that point. The *time* field can also change without any user interaction when the user is working in a *real-time* project. Upon reception of a network message, the *Operation Manager* processes the relevant received data, and emits the gathered information to the GUI, in order needed to update the interface. The *type* of project changes when the user requests data to be loaded from the database, start a live session, or when the user terminates working on the current project.
  Accordingly, the updates that are made to this zone are performed inside dedicated slots, which are executed every time the user performs any operation, or new data is received from the network, thus affecting the state of the information displayed in this zone.

In summary, the graphical user interface allows the users to perform the type of analysis hereby considered. It is composed of a single unified main screen for the two types of projects considered: *live* and *offline*. It is through this main screen that the users perform all the operations related to a session, such as the graphical elements displayed, or to configure information to be used across sessions, such as the sensors to be considered in the analysis.

The GUI enables the users to manipulate the layout according to its needs, by allowing them to

configure the information to be displayed, as well as the graphical items in which this information is displayed. The user is provided with an intuitive layout divided according to the needed functionalities, consisting of: an upper zone to perform configuration operations, or creation/loading session operations; a middle zone where the sensor data is displayed in the graphical widgets; and a lower zone to display the state of the current analysis.

5

Results

Contents

5.1 Summary

This chapter provides the conducted evaluation performed of the implemented framework, by taking into account the set of objectives defined in Chapter *1*. For this purpose, a set of tests were performed and the obtained results analysed. The following sections provide a description of the test procedures and the corresponding evaluation environments, together with a discussion of the obtained results.

5.2 Testing Environment

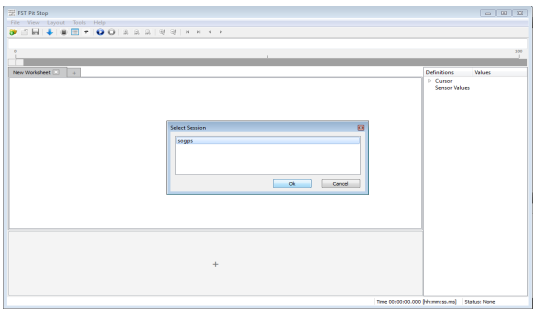
The winter season is usually used by the Formula Student teams to develop and modify their prototyping vehicles. As a result, the opportunities to test and evaluate the car, and the proposed framework in particular, in a real racing environment were very few. Consequently, most of the conducted evaluations had to be done in the FST laboratory at IST. The materials used for these test scenarios were:

- Mobile station platform, installed in the lab.
- A set of sensors of the car, used to evaluate the platform in *real-time* operation mode.
- Personal computer to act as pit station.
- Communication infrastructure composed by the wireless router that manages the communication between both stations, and the Wi-Fi USB adapter that enables communication to and from the mobile station. The first was installed in between both stations, while the second was connected to the mobile platform *mini-computer* through a USB interface.

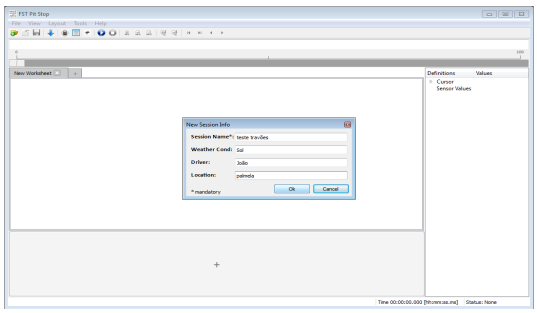
Section *5.4* describes the development test tools, the conducted tests, the evaluation methodology and the obtained results.

5.3 Session Management

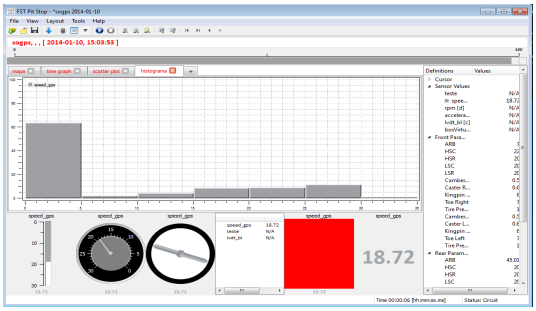
As required by the end users, the framework consists on an integrated unified GUI, in order to accommodate the available functionalities. The procedure that has to be followed to graphically manage a session's data is the same if the user is working in a *real-time* or *offline* project. This way, it is easier to apprehend the set of available features and the relationship between them, which makes the GUI more user-friendly. The screenshots depicted in Figure 5.1 illustrate the process of initiating any of these types of project, as well as some of the available features.



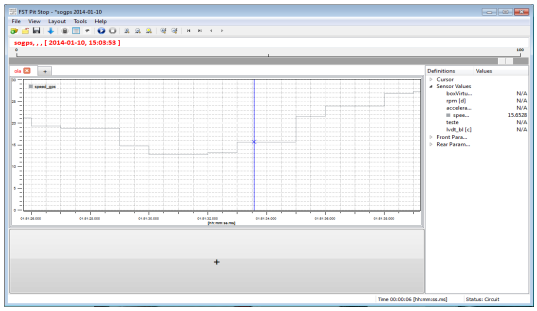
(a) Select the session to be opened.



(b) Sarting a *real-time* session.



(c) Multiple tabs created, with different graphs, as well as all the available gauges.



(d) Analysing a specific time sample, after zooming and adjusting the time slider.

Figure 5.1: Process of initiating each type of project, alongside the common process of managing graphical elements within the screen layout.

5.4 Experimental Tests And Results

5.4.1 Database Insertion Speed

As described earlier, the data gathered from the sensors is made persistent through the *PostgreSQL* database. However, to ensure an effective storage of the received data, the rate of insertion of sensor samples in the database must be enough to cover the rate of reception of data from the network. To measure the capability of the system to cope with the data reception rate from the network, a sample testing procedure was used to ensure the validity of the database insertion speed. This should be enough to cope with the maximum reception rate, which according to [7, 8]), is *13797.05 samples/second* from the CAN bus, and *1 samples/second* from the GPS receiver.

The performed test measures the database insertion speed for the new implementation by using the *QtSql* module. This test consists in evaluating the capacity of the system to store data received from the mobile station, using several sampling rates. Figure 5.2 illustrates the performance of the database, when there are only writing operations being performed, and when there write and read operations happening simultaneously. It can be observed from the *write only* test, that the database can easily cope with the maximum rate at which data is acquired, since the

number of writing operations that can be performed per second is between *33000* and *34000*. For the *read/write* test, a thread was used that constantly read data from the database, while other thread performed the storage operations.

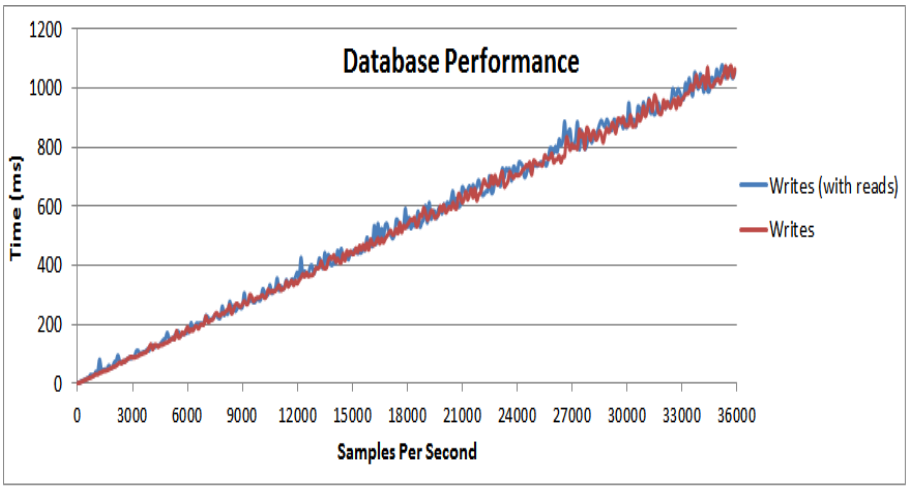


Figure 5.2: Max writing rate of *PostgreSQL*, when there are only write operations, and when there are write and read operations simultaneously.

It can be observed that both cases have similar performances, and hence the database can cope with the writing and reading simultaneously.

Figure 5.3 illustrates a subset of the writing performance test illustrated in Figure.

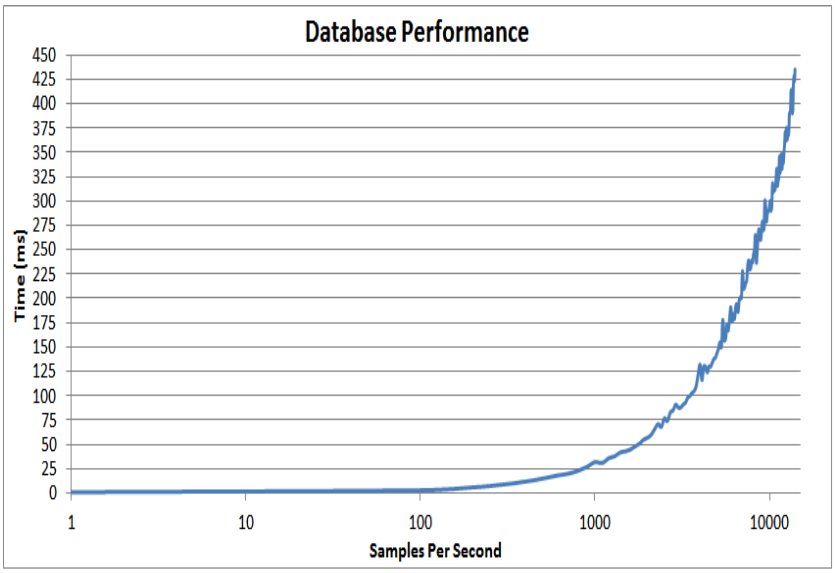


Figure 5.3: Subset of the write only performance of *PostgreSQL*.

From Figure 5.3, it can be observed that the new database management implementation can easily handle the maximum rate at which data arrives at the system.

In regards to the used batch mechanism, a modification is able to be performed due to the

better multi-transactional handling performed by the *PostgreSQL*. It can be observed from Figure 5.3 that the speed performance is constant, about *5 ms*, until the sampling rate is around *100 samples/second*. This would mean that, if the batch mechanism is executed at a rate at which *70 samples* or less are stored in the database, the insertion speed would not handle the maximum possible rate at which data can arrive at the system. However, this can be achieved if the batching mechanism is set to be executed for *70 samples* or more. Hence the timeout for executing the batch mechanism, which was every *1* second in the previous version of the framework, was set to *5 ms*, which is the time needed to received *70 samples* if the maximum sampling rate is considered. Thus, this makes the storage of data much more accurate, without affecting the performance of the application.

5.4.2 Network Communication Speed

In a *real-time* session, it is important that the network throughput is enough to deliver all the messages containing data from the car sensors. To evaluate the performance of the framework communication protocol, a sample test was performed in which data was transmitted from the mobile to the pit station. The test consisted in transmitting a set of data with *16 MB* using several network packet sizes, and evaluate the time taken to transmit the whole data. The results are depicted in Figure 5.4.

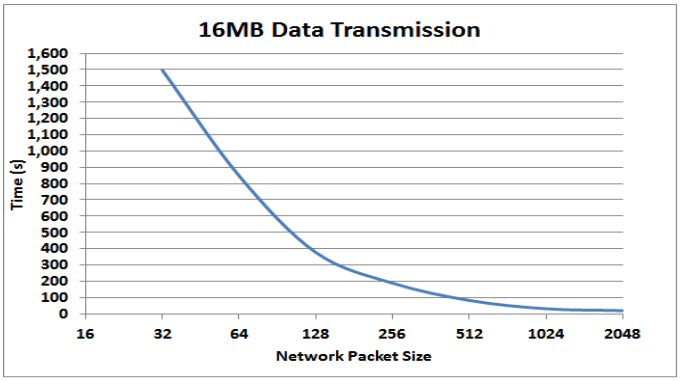


Figure 5.4: Transmission time required to send *16MB* to the pit station, by using several network packet sizes.

Due to limitations on the mobile station hardware, the maximum network packet size was *2048* bytes. This is because the hardware does not support a Maximum Transmission Unit (MTU) greater than *2274*. However from the above figure, it can be observed that there is a significant network overhead when the packet size is small, causing a larger number of packets to be transmitted. Thus, it can be concluded that there is a clear advantage of including more data within a single network packet.

5.4.3 Mathematical Engine Performance

The execution performance of the mathematical manipulations is crucial, particularly when gathering data in *real-time*. Therefore, the evaluation speed of the *muParser* had to be analysed, in order to make sure that it would be enough for the gathering rate of the sensor measurements. A set of benchmarks, illustrated in Figure 5.5, were used for this purpose, showing the performance of this library for some of the processing formulas used by the team for processing the received data. The test consisted in evaluating this set of formulas, considering different data reception rates. The library should be able to cope with the maximum data reception rate (according to [7, 8]), *13797.05 samples/second* from the CAN bus, and *1 samples/second* from the GPS receiver).

The set of post-processing formulas, where *filter* represent the raw value received from the mobile station, used in the conducted test is the following:

- RPM: *((78125/filter)*60/2)/1000.0*
- Acceleration: *(465-filter)/213.6*
- Tire temperature: *filter*0.02-273.15*
- Throttle: *filter * 100.0 / 4096.0*
- Displacement: *filter*10.0/65536.0*

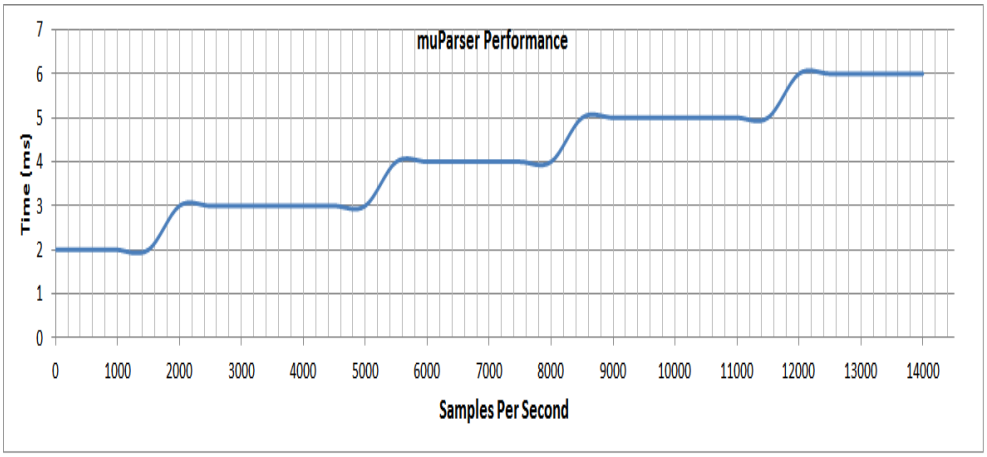


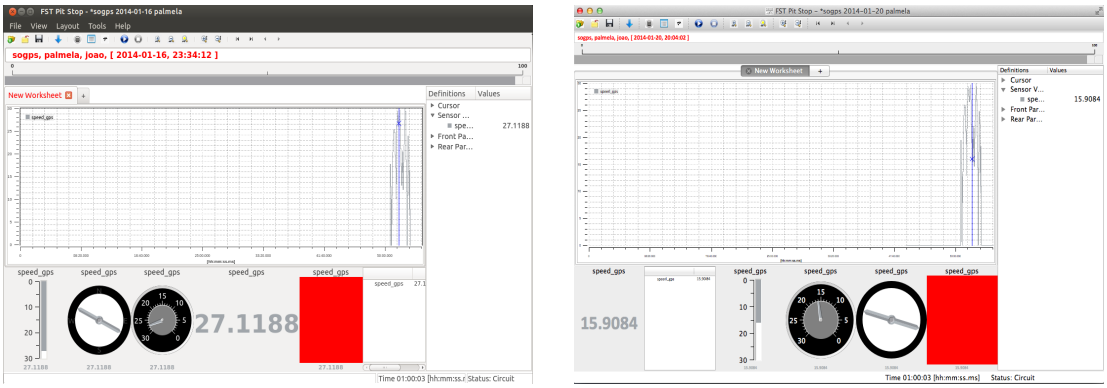
Figure 5.5: *muParser* speed performance, evaluated for several sampling rates.

As expected, there is an increase in the time taken to evaluate the expressions, as the sampling rate grows. Nevertheless, it can be observed that for the maximum possible data acquisition rate (see [7, 8]), the time needed for evaluating the expressions is not an issue. Hence, it can be concluded that for the type of post-processing analysis herein considered, the library can easily handle the data acquisition rate.

5.4.4 Multiple Operating System Support

One of the main objectives of this project was to turn the pit station side of the framework into an operating system independent application. As described earlier, this was obtained by using the Qt Framework in the development of the new version, which provided the necessary cross-platform API to implement the needed functionalities.

Hence, tests were conducted in all the targeted operating systems to prove the framework capability of being used in all of them. Figure 5.6 shows the framework look and feel for the different target operating systems. The cross-platform features are managed by Qt Framework in all of them.



(a) Framework main screen under Linux.

(b) Framework main screen under Mac.

Figure 5.6: Framework look and feel, for analysis performed under Mac and Linux.

5.4.5 Sensor Data Analysis

The most important characteristic of the framework is to provide the means to analyse the data gathered in a testing session, from the sensors installed in the car. This section demonstrates the capacity of the system in handling the data that is gathered in a testing session performed by the team. Since no test could be performed on track, this example represents an analysis session, using an *offline* project, loaded from a data file containing the sensor data. The working session is illustrated in Figure 5.7, Figure 5.8 and Figure 5.9.

It can be observed that the GUI can accommodate a diverse set of graphical widgets to display the information gathered from the sensors. Particularly, the framework includes all the required scientific widgets, in terms of graphs and gauges, displaying information in several formats. This set of widgets include the new widget created specifically for the GPS module, integrated into the overall application. For the graphs, the information can be displayed in a time based manner, trough *time-graphs*, as a relationship between two sets of data, as an historical frequency analysis of the data gathered in a session, or in a track map. For the gauges, the information is displayed

for a single instant in time, but through several graphical representations which are tailored for different situations, in order to facilitate their analysis. For example, to evaluate the angle of the steering wheel in an instant, a *Steering Wheel* gauge should be used, in order to graphically facilitate its study.

In summary, the users are able to manage the graphical layout, by choosing which widgets they want to display information and which data they want to be presented in these widgets, and hence, adapt it to their specific needs.

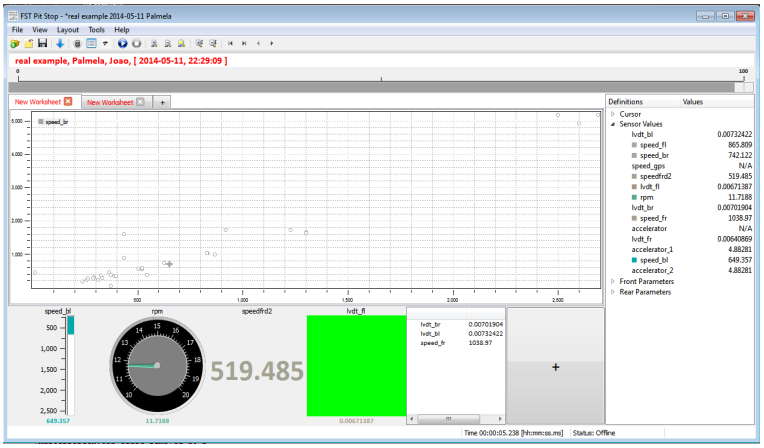


Figure 5.7: Offline session with scatter plot. This type of plot is an *X-Y* graph, in which each axis is filled with values from one sensor. This graph is used to evaluate the variation of one sensor in relation to another.

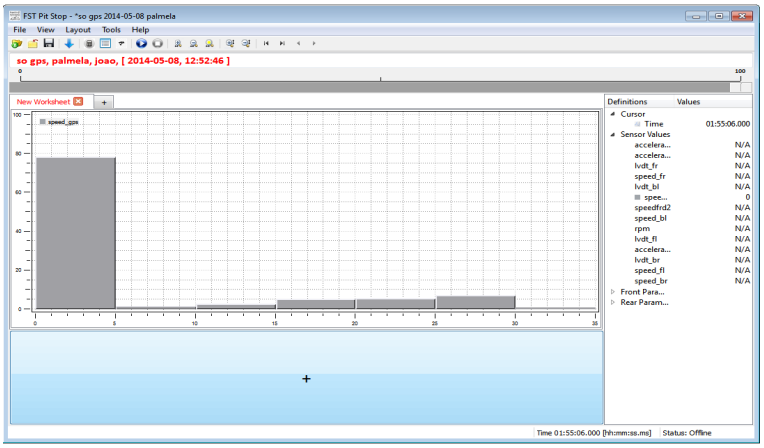


Figure 5.8: Offline working session with an histogram. This graph displays a set of bars, which represent the frequency of a range of values in a given session.

5.4.6 End-Users Feedback

The main motivation for this project was to develop a useful system, that enabled the FST IST team to perform vehicle condition analysis both in *real-time* and *offline*. The system should be used by the team in their working sessions, be helpful in their decisions and thus improve their

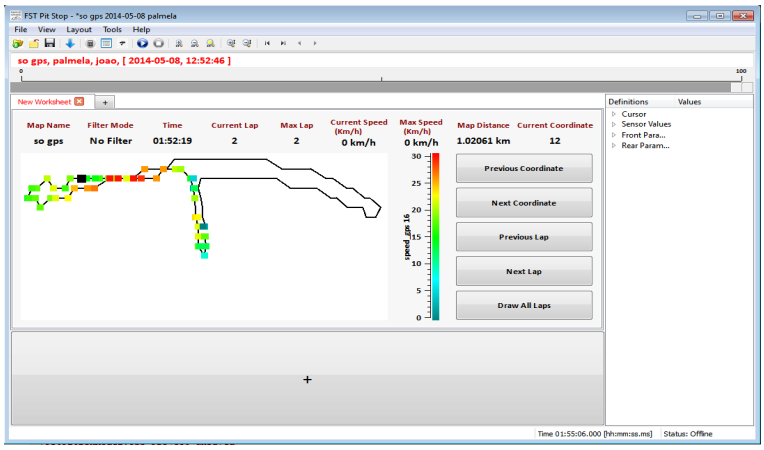


Figure 5.9: Offline working session with a gps track map. The interface for this widget is an adaptation of the interface used in the previous version of the framework. The user can chose any sensor, and browse through the acquired map coordinates to analyse the sensor values for any of them.

productivity. To evaluate their satisfaction with the end result, feedback was gathered to determine the success of the system. The following is a set of statements from two of the FST IST team leaders which were involved in the development process from the beginning:

- "Looking at João's final solution it does seem to be a good prototype for a data acquisition system's interface for the following reasons: It follows the main guidelines in regard to the layout of the menus, charts/graphs areas, etc. more or less specified by the suspension, vehicle dynamics and electronics teams which are undoubtedly the areas where reside most of the potential benefits from this tool. It has a reworked core implementation that will allow for a new level of customization, that formerly prevented previous interfaces designed for this system not to be adopted by the team. Namely it allows for complete freedom in choosing the number and meaning of each CAN ID being used, which means arbitrary number of sensors and IDs reassignment. I was demoed in lab conditions with some hardware and software designed for the car's side of the system, already surpassing the 'proof of concept' status. Compared to other teams systems, it seems to allow better modularity making it a tool, not for the previous car prototype, not for the current one, but for the future ones, putting IST's team on the right track not only to equal but even surpass competing teams." - Bruno Santos, Ex-Electronics & Propulsion team leader.
- "From the contact I had with the application and João's work in the lab, the application enables the evaluation of various parameters of the vehicle in real-time take decisions based on these. It also allows to run previous stored logs to evaluate the behavior of the vehicle and and get to know better what should be improved. It allows to chose how to treat the gathered measurements, which makes our work easier as it displays immediately the needed information. The fact that we can chose how the information is displayed is an added

> value since we can define the best output tailored to specific needs. When we manage to be familiarized with the application it is going to be easier to work on the several parameters of the car that are being observed, which will enable faster improvements on the vehicle. - Daniel Pinho, Electronics team leader.

From this, it can be concluded that the main goals for the framework in what regards to the satisfaction of the end users team were achieved, since the team are now comfortable to use it as a means to analyse the data gathered from the car during the practice sessions and hence, improve the quality of these.

6

Conclusions

Contents

6.1 Conclusions

This work aimed to offer an added value to the FST IST team analysis sessions tools, used to analyse the data that is gathered from the car sensors in order to prepare its vehicles for competitions. In particular, it aimed to obtain a great improvement on the quality of the data analysis and visualisation and a consequent reduction in the time needed to prepare the vehicle prototypes. Although it is a follow-up to a series of works that were previously performed, it was developed as a complete new system. This was done to avoid the undesirable effects suffered in the previous works, which were related to the lack of integration between all the system elements. Hence, designing the system from scratch, considering the full set of requirements desired by the users, would intrinsically integrate all the system components.

Since this is a user oriented work, the main goal that had to be accomplished in order to succeed was to create a software usable and useful for the team. Therefore, the major factor affecting the success of the project was the contact with the targeted users. This permanent contact with the users aimed the accomplishment of a system that pleases all the types of users within the team, both in terms of graphical presentation and system functionalities. This means that, in terms of graphical interface, users should be comfortable with the working environment that is provided. This is one of the most important element since this is the point of interaction between the users and the system, and therefore may or may not drive them to use it. Furthermore, in terms of that set of offered functionalities, the system should cater the user needs relating to the evaluation of vehicle conditions during a testing session, and provide the necessary means for the required physical analysis.

However, there was another important party to take into account during the system development: other possible developers that may arise to improve the system. Hence, the system was designed to provide easy means for future project evolutions based on the version herein presented.

In general, the development process can be divided into several phases that were conducted to accomplish the goals mentioned above. It can be described in the following:

1. Gather with the users, to define the system requirements.
2. Decide which modules from the previous version could be used in the new version. For instance, the GPS module implementation was kept in the new version. In regard to the communication protocol used in the previous version, its architecture and design were considered for the new version, but a new implementation had to be made.
3. Develop the architecture for the system.
4. Iterative process of system development, alongside the end-users, which comprised two main activities: designing and developing the graphical user interface; and implementation of the required functionalities.

5. Perform the necessary tests to evaluate the success of the project. These were evaluated according to the user requirements and objectives defined for the project. In general terms, this means the user acceptance of the graphical interface and correctness in performing the desired functionalities.

Summarizing, the improvements made to the previous version of the project, in terms of added value to the team, they can be divided into several categories, namely, one for each module present in the system:

- Regarding the communication protocol, improvements were introduced which enable the different users to use the system in several operating systems, namely, Windows, Linux and Mac. Another main benefit was introduced by restructuring the network packets used by the communication protocol, turning it independent from the data sent, and hence, enabling the customization of the content exchanged between the two stations.
- It was decided to change the database management system, aimed to significantly improve the following aspects of the database: speed performance in a multi-transactional environment. This enabled the storage of the data at a faster pace than the one in the previous version, and hence, have a more accurate database management implementation.
- The addition of the XML management module introduced one major improvement: it removed the need (for the users) to know how to work directly over the DBMS every time they wanted to change any information for the application, and thus it added a level of flexibility that was formerly missing. It enables the users to easily configure several types of information to adapt the analysis to their profiles. They can manage which, and how, the data is considered by the application and thus, manage all the information they need to analyse at a specific moment.
- The addition of the mathematical module significantly improved the quality of the analysis. It provides features to manipulate the information related to the sensors installed in the mobile station. These manipulations correspond to mathematical formulas that are applied to the sensor's data, in order to obtain a new set of data that is useful for the analysis.
- The graphical user interface provides a single and unified working environment for the end users. The design of the interface was performed with the involvement of the end users to better cater their needs and accommodate their working styles. The interface is adaptable to various users profiles, giving them the possibility to analyse and compare different data in multiple graphs and gauges and to browse a graph data based on a timeline mechanism.

Essentially, the points mentioned above were the ones that were mainly missing from the previous versions of the system, resulting in a incorrect attendance of the users' needs and behaviours. This changed with the new version of the system, since there was a great effort put into cater what were the requirements and working styles of the end users, in order to best provide

the means for them to perform their tasks in an easy and intuitive way. This was validates by the feedback given by the end users.

6.2 Future work

Since the technology is constantly evolving, new methods for improving the performance of real-time analysis and graphical presentation are always appearing. This project has been initiated some years ago, and since its inception it has been the target of several improvements. However, there is always room for extra improvements, and these do not mean improving only the system capabilities, such as performance, but also extending the set of provided features, by adding more relevant functionalities, or by improving system usability. Moreover, since this is an academic project, new needs arise in short time periods, as new students are always entering into the FST IST team, and the used technology in the cars continues to evolve. This section presents some of future enhancements that would likely be useful for future analysis provided by the users.

The new version of the framework does already cater the most relevant needs for the team to perform helpful analysis, either in real time or from data previously stored. The team can already take some benefits from the analysis made in the framework, and have its productivity improved. The following items describe possible improvements to be made to increase the advantages of the framework for the analysis sessions:

- Although the existing graphics that are made available to the user already fulfils the user's needs, having 3-D graphics would give the user additional means of analyzing data. 3-D graphics would be useful for instance as a standard time graphic, like the one presented in this thesis, but it could be used for comparing values from one sensor to another. The third dimension would be assigned to the time.
- Add a zoom-by-dragging mechanism. The user should be able to pick a time range in the graph which he wants to zoom in. This should be accomplished by a mouse drag mechanism associated to a plot which selects, and momentaneously highlights, the zone comprised in the user selection. When the user finishes his selection, the graph should be zoomed in to show the selected zone in more detail.
- Put more data in network messages sent from the mobile station. This means having the network messages comprise data from more than one sensor. This would reduce the number of messages sent from the mobile to pit station and vice versa. It would mean that instead of generating a new timestamp for each value gathered in the CAN bus, a value could be generated each time a set of values was gathered, thus increasing the precision of time of a given sensor value.
- It would be very helpful to be able to communicate with the driver when performing a testing session. Likewise, it would be very helpful to analyse how the driver is behaving on the

track, by view images from the race in real time. For these purposes, and to further add quality to the analysis session, the necessary adaptations in the framework should be made in order to add this functionalities.

Bibliography

[1] The PostgreSQL Global Development Group, PostgreSQL 9.3.1 Documentation, http://www.postgresql.org/files/documentation/pdf/9.3/postgresql-9.3-A4.pdf.

[2] MySQL, MySQL Reference Manuals 5.6 Reference Manual, http://downloads.mysql.com/docs/refman-5.6-en.a4.pdf, 2013.

[3] SQLite Database, SQLite documentation, http://www.sqlite.org/docs.htm.

[4] Formula Student, http://www.formulastudent.com/.

[5] Projecto IST formula student, http://www.projectofst.com.

[6] David Rua Copeto, Automotive data acquisition system - FST, Master's thesis, IST, 2009.

[7] Paulo Fernandes Mendes, Formula student racing championship: design and implementation of the management and graphical interface of the telemetry system, Master's thesis, IST, 2011.

[8] Diogo Rafael Bento Carvalho, Formula Student Racing Championship: Design and implementation of an automatic localization and trajectory tracking system, Master's thesis, IST, 2012.

[9] MoTeC products specification, http://www.motec.com.au.

[10] MoTeC interpreter manual, http://www.motec.com/filedownload.php/Interpreter_\Manual_A5.pdf?docid=1084.

[11] MoTeC i2 software brochure, http://www.motec.com.au/filedownload.php/i2%20Brochure.pdf?docid=2388.

[12] McLaren ATLAS, http://www.mclarenelectronics.com/Content/Products/ATLAS/ATLAS.pdf.

[13] Cosworth Toolbox, http://cosworth.com/products/racing-electronics/software/toolbox/.

[14] Formula SAE, http://students.sae.org/competitions/formulaseries/.

[15] FIA Formula 1 2013 technical regulations, http://www.fia.com/sites/default/files/regulation/2013-F1-TECHNICAL-REGULATIONS-111212.pdf.

[16] Linear Interpolation, http://www3.nd.edu/~jjwteach/441/PdfNotes/lecture5.pdf.

[17] wxWidgets Cross-Platform GUI Library, http://docs.wxwidgets.org/3.0/.

[18] GTK+ Multi-Platform Toolkit, https://developer.gnome.org/gtk3/stable/.

[19] Qt Development Framework, Qt Reference Documentation, qt-project.org/doc/qt-5.1/.

[20] Qwt library, http://sourceforge.net/projects/qwt/files/qwt/6.1.0/qwt-6.1.0.pdf/download.

[21] Lagrange Polynomials, http://www3.nd.edu/~jjwteach/441/PdfNotes/lecture6.pdf.

[22] Lagrange Polynomials, https://ccrma.stanford.edu/~jos/pasp/Lagrange_Interpolation.html.

[23] Nearest-Neighbour Interpolation, http://sepwww.stanford.edu/public/docs/sep107/paper_html/node20.html.

[24] MathPresso Library, https://code.google.com/p/mathpresso/.

[25] Matlab Compiler Runtime, http://www.mathworks.co.uk/help/compiler/working-with-the-mcr.html

[26] muParserSSE Library, http://beltoforion.de/muparsersse/math_expression_compiler_en.html

[27] muParser Library, http://muparser.beltoforion.de/

[28] SMS Networks, SMSWEBT-G EZ Connect g 108 Mbps, http://www.smc-asia.com/files/MN_SMCWEBT-G_EN_0708_FW_B011.pdf

Appendix A

Contents

A.1 Communication Protocol

The communication protocol implemented in the framework was developed in the scope of a previous thesis [7] with the aim of offering some resilience to faults. This means that it is prepared to deal with packet, and connection losses. To accomplish this, a communication recovery mechanism was developed and implemented on both stations. When in sender role, the peer stores each sent packet in a circular list, only used for packets which were already sent, and where it will be temporarily available in case a retransmission is required. This circular list of already sent packets is statically allocated and has a fixed size. In line with this, when a packet is received, an acknowledgement packet is sent from the receiver. Only when the original sender of the packet receives the acknowledgement packet, will it remove the corresponding packet from the circular list. Retransmission of packets is required in two situations:

- One communication party explicitly requests a packet to be retransmited. This can occur when the receiver of that packet checks that the checksum computed for the packet received does not correspond to the one sent in the received packet. It can also occur if a packet is received out of order, in which case the receiver requests the packet with the number it is expecting to receive;
- A time out occurs for the maximum tolerated waiting period associated to an acknowledgement message of a particular packet. This means that either the message associated to the packet itself or the acknowledgement packet was lost.

As part of the sending procedure, an additional mechanism was developed to give some packets a higher priority than others when information is about to be transmitted. The reason is that some of the sensors are considered more relevant than others, and so the packets carrying their information should be sent ahead of the others. To accomplish this, three types of priority queues were developed, where packets are stored before being sent. These priority queues are used as temporary waiting queues for messages that are ready to be sent. The process of storing packets in these queues is described later in this section. The three types correspond to the priority level associated to each packet: high, medium, and low priority queues. The priority of each packet was assigned based on the identifier of the sensor being read. The range of identifiers assigned to each priority queue is depicted on Table A.1. This way, it is ensured that packets from sensors

Priority	ID
High	0 - 32768
Medium	32769 - 49152
Low	49153 - 65536

Table A.1: Mapping between transmission priority level and session identifiers [7].

connected to the CAN bus will have the highest priority, since their identifiers are always kept in the range of 0 - 2048. The medium and low priority queues are used for audio and video,

respectively. These two lower priority queues have a fixed size. This is justified by the fact that, for audio and video (sequence of Joint Photographic Experts Group (JPEG)s), it is only relevant to listen to or watch the most updated information being transmitted from the car. As a consequence, when they are full and a new packet is to be sent, the oldest one is discarded. Hence, regarding the process of transmitting a packet, the framework is structured according to these two logical sides: the sender and the receiver side. From the sender point of view, when a message is ready to be sent, the priority queues are first searched. The identifier of the sensor responsible for the data in the packet is then used to check the corresponding priority queue for emptiness. If the queue is not empty, the packet is temporarily stored, for later transmission. Otherwise, the packet is sent right away, and stored in the circular list of sent packets. In case the send process fails, the packet is stored back in its priority queue for later transmission. This process is depicted on Figure A.1.

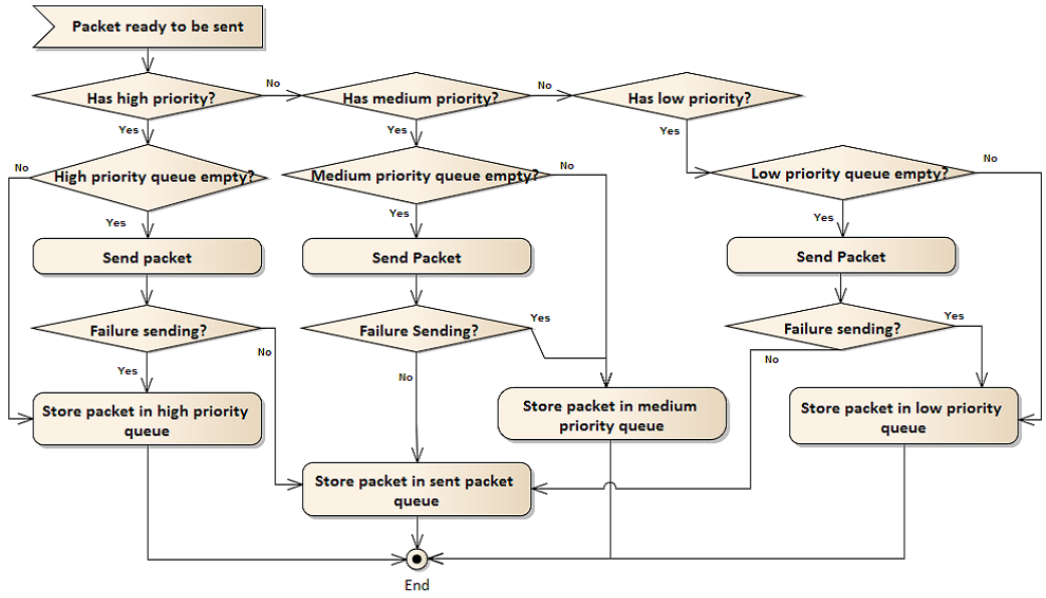


Figure A.1: Process of transmitting a packet.

As it was mentioned earlier, the already sent packets are temporarily stored in a circular list and they are kept there until the corresponding acknowledgement packet is received. Only when an already sent packet is removed from the sent packets list, can a new packet be removed from a priority queue. In this situation, the priority queues are searched, starting from the queue with the highest priority. If one non-empty queue is found on this process, the first packet from that queue is sent, removed from its priority queue, and stored in the sent packets circular list. In case there is a failure while sending this packet, it remains in its priority queue. This process is depicted on Figure A.2.

On the other side, there is the party who receives the packets. From the receiver point of view, two lists are maintained to store the received packets. One of these lists stores packets

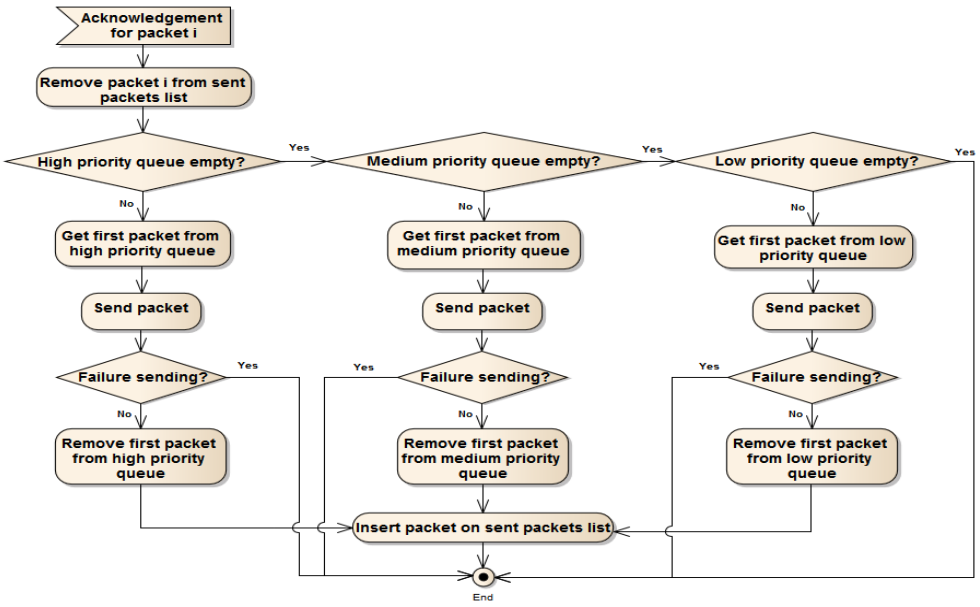


Figure A.2: Management of the priority queues in the packet transmission procedure.

that can already be processed. This is the case when all previous packets that were sent before them have already been received. This is checked based on the packet number sent in each network message. The other list stores packets received out of order, preventing unnecessary retransmission of packets. The list of out of order packets have a fixed size. Hence, packets are only stored in this list in case the packet number is within a boundary window, which has the same size as the circular list of sent packets mentioned earlier. In short, if a received packet has the expected packet number, it is stored directly on the list of ready to be processed packets. Otherwise, the packet is stored in the list of out of order packets and removed when all packets that were sent before it have been received. The flow of actions performed when a packet is received is illustrated on Figure A.3.

Both stations act as sender and receiver, although under normal circumstances, the mobile station is supposed to act more like a sender, because the relevant data is produced here, and the pit station more like a receiver, because relevant data is to be received and shown to users here. A detailed description of the protocol can be found in [7].

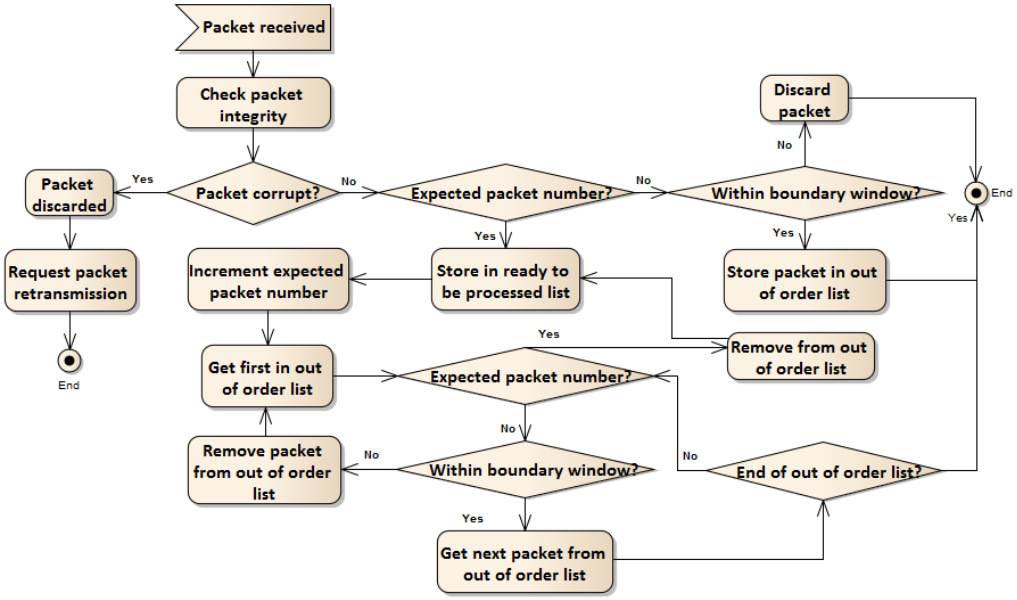


Figure A.3: Management of received packets.

B

Appendix B

Contents

B.1 Lagrange Polynomial

Lagrange Polynomial [21, 22] is an algorithm for creating sets of new data points within a set of known data points. It belongs to the set of polynomial interpolation algorithms, which means that for a set of given points, it finds a polynomial that goes through all of them. Technically speaking, given a set of N+1 known samples as follows:

$$(x_0, y_0), ..., (x_j, y_j), ..., (x_N, y_N) \tag{B.1}$$

with:

$$j = 0, 1, 2, ..., N$$

the problem is to find the *unique* order N polynomial $g(x)$ which interpolates the given samples. Figure B.1 illustrates the an example with $f(x)$ representing an exact function of which only the N+1 points are known, and $g(x)$ the interpolation function which will pass through all the known points.

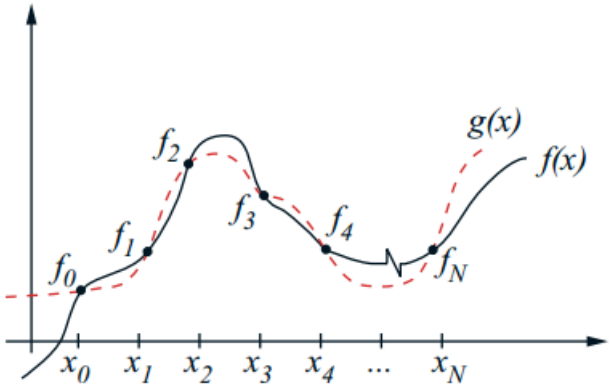


Figure B.1: Lagrange exact N+1 points function and correspondent N degree interpolation polynomial [21].

The solution to the problem, is expressed as the result of the following equation:

$$g(x) = \sum_{k=0}^{N} l_k(x) f(x_k) \tag{B.2}$$

with:

$$f(x_k) = f(x)\ evaluated\ at\ point x_k$$

and:

$$l_k(x_j) = \begin{cases} 1 & j = k \\ 0 & j \neq k \end{cases} \tag{B.3}$$

To exemplify, imagine that five samples were known, $(x_0, x_1, x_2, x_3, x_4)$, which means that $N = 4$. To evaluate the interpolated function in for example, sample x_2, the procedure is the following:

$$g(x_3) = f_0 l_0(x_3) + f_1 l_1(x_3) + f_2 l_2(x_3) + f_3 l_3(x_3) + f_4 l_4(x_3) \tag{B.4}$$

From the definition given above, the result is:

$$g(x_3) = f_0 \times 0 + f_1 \times 0 + f_2 \times 0 + f_3 \times 1 + f_4 \times 0 = f_3 \tag{B.5}$$

So, to construct the polynomial function $l_k(x)$, we start by defining the following polynomial:

$$w_k(x) = (x - x_0)(x - x_1)(x - x_2)\dots(x - x_{k-1})(x - x_{k+1})(x - x_N) \tag{B.6}$$

which satisfies the property stated above, with w_k having roots at all samples except x_k. The only drawback is that the polynomial $w_k(x)$ is not equal to one at x_k. So, in order to obtain a final result of one at x_k, the polynomial must be normalized, obtaining $l_k(x)$, yielding:

$$l_k(x) = \frac{(x - x_0)(x - x_1)(x - x_2)\dots(x - x_{k-1})(x - x_{k+1})(x - x_N)}{(x_i - x_0)(x_i - x_1)(x_i - x_2)\dots(x_i - x_{k-1})(x_i - x_{k+1})(x_i - x_N)} \tag{B.7}$$

With the equation defined above, the both criteria are satisfied, $l_k(x)$ evaluating to one at x_j for $j = k$ and evaluating to zero at any $j \neq k$. The above equation is more commonly defined as:

$$l_k(x) = \prod_{j=0, j\neq k}^{N} \frac{(x - x_j)}{(x_j - x_k)} \tag{B.8}$$

To illustrate the presented algorithm, consider an example for which $N = 2$, and the following set of points:

$x_0 = 3\ f_0 = 1$
$x_1 = 4\ f_1 = 2$
$x_2 = 5\ f_2 = 4$

To find the interpolation polynomial $g(x)$, the following $l_k(x)$ are define:

$$l_0(x) = \frac{(x-4)(x-5)}{(3-4)(3-5)}, l_1(x) = \frac{(x-3)(x-5)}{(4-3)(4-5)}, l_2(x) = \frac{(x-3)(x-4)}{(5-3)(5-4)}, \tag{B.9}$$

The resulting interpolation polynomial results in:

$$g(x) = 1 \times l_0(x) + 2 \times l_1(x) + 4 \times l_2(x) \tag{B.10}$$

Figure B.2 illustrate the process of obtaining the above interpolation polynomial.

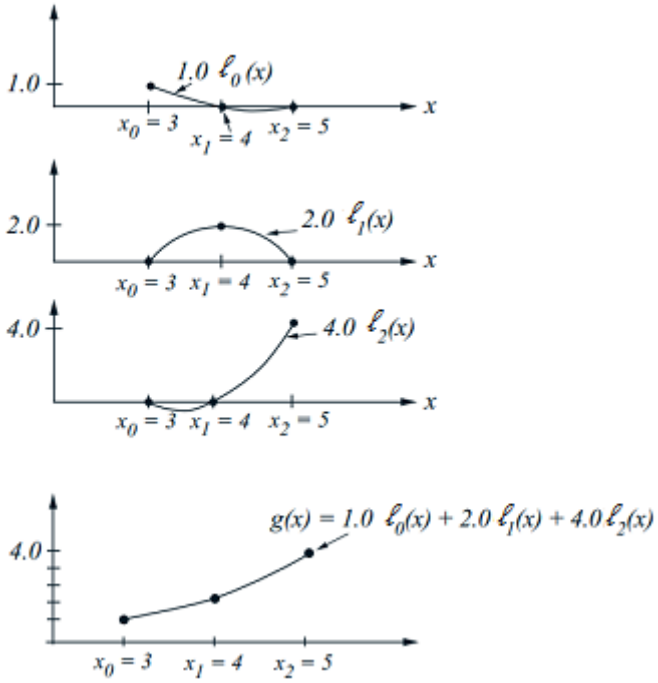


Figure B.2: Process of creating an interpolation polynomial of degree two [22].

C

Appendix C

Contents

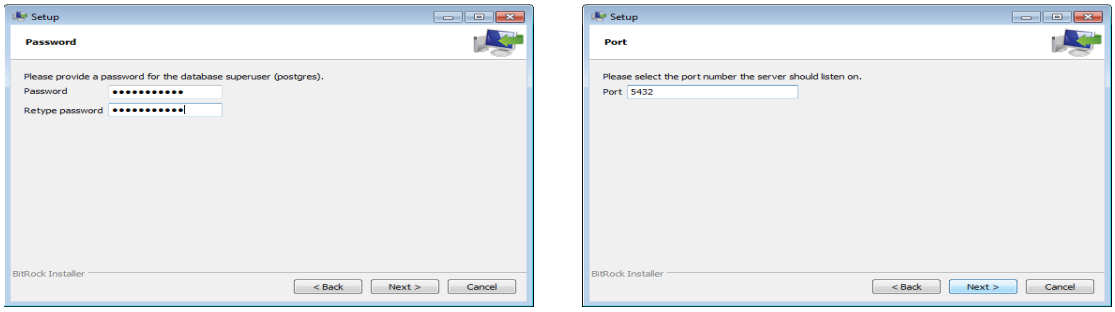
C.1 Installation Guide

In order to have the full features available to run the pit station side of the framework, the following instructions should be taken:

1. Download and install the *PostgreSQL 9.3* bynary packages for the operating system in use from `http://www.postgresql.org/download/`. During the installation several questions will arise, of which the following are preponderant for the hereby presented software:

   - Configure the superuser for the database server created, setting the password to *fst-database*.
   - Set the database server to listen on port *5432*.

   The figure below illustrates the steps above in *Windows*, but it should be the same for the others:



(a) Set the username and password as above.

(b) Create the server listening on the specified port as above.

Figure C.1: Installation steps for configuring *PostgreSQL*.

   In case of *Windows* these should be installed under the directory *C:\PostgreSQL*. In case of *Linux* this should be installed under the directory *etc/*. For last, in case of *Mac*, install PostgreSQL under the directory */Library/PostgreSQL*.

2. Download and install the *Matlab Compiler Runtime 2012a* binary packages for the target operating system from `http://www.mathworks.com/products/compiler/mcr/`. Install it under the default directory provided by the installer for all the operating systems.

3. Download and install *Qt Framework* binary packages in the target operating system from `http://qt-project.org/downloads`. Install it in the default directory provided by the installer during the installation for all operating systems. During the installation process, a dialogue will appear which prompts the user to select which components to include in the installation. The user should tick the box that says *Source Components* as in the figure below:

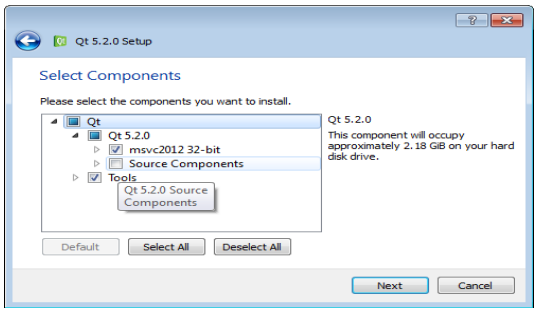


Figure C.2: Qt installation step where the user should tick the box corresponding to *Source Components*.

Once finished there is a couple of steps that need to be performed to integrate with the *PostgreSQL*. These are covered in detail in `http://qt-project.org/doc/qt-4.8/sql-driver.html#qpsql`, so just follow the instructions given for the target operating system.

4. The last step before being able to run the application is to install *Qwt Library*. First download version 6.1.0 from `http://sourceforge.net/projects/qwt/files/qwt/6.1.0/`. Then follow the *Build and installation* instructions on `http://qwt.sourceforge.net/qwtinstall.html` for the target operating system.

5. Install the framework application.