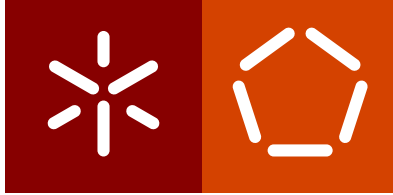


Universidade do Minho
Escola de Engenharia
Departamento de Informática

Bruno José Infante de Sousa

ROS-based Data Acquisition System

May 2022



Universidade do Minho
Escola de Engenharia
Departamento de Informática

Bruno José Infante de Sousa

ROS-based Data Acquisition System

Master dissertation
Integrated Master's in Informatics Engineering

Dissertation supervised by
Professor Doutor Paulo Cardoso
Doutor Adriano Carvalho

May 2022

COPYRIGHT AND TERMS OF USE FOR THIRD PARTY WORK

This dissertation reports on academic work that can be used by third parties as long as the internationally accepted standards and good practices are respected concerning copyright and related rights.

This work can thereafter be used under the terms established in the license below.

Readers needing authorization conditions not provided for in the indicated licensing should contact the author through the RepositóriUM of the University of Minho.

LICENSE GRANTED TO USERS OF THIS WORK:



CC BY-NC-ND

<https://creativecommons.org/licenses/by-nc-nd/4.0/>

STATEMENT OF INTEGRITY

I hereby declare having conducted this academic work with integrity.

I confirm that I have not used plagiarism or any form of undue use of information or falsification of results along the process leading to its elaboration.

I further declare that I have fully acknowledged the Code of Ethical Conduct of the University of Minho.

ACKNOWLEDGEMENTS

This work is supported by: European Structural and Investment Funds in the FEDER component, through the Operational Competitiveness and Internationalization Programme (COMPETE 2020) [Project n° 039334; Funding Reference: POCI-01-0247-FEDER-039334].

ABSTRACT

Nowadays, we are living in a time where sensors and applications that take advantage of them are increasingly taking part of our daily lives. Thus, it is increasingly common to be surrounded by sensors, like image, sound or even luminosity or motion sensors, among several other types. In this context arises the challenge of how to connect sensors and applications that use them. The basic approach is to have each sensor bound to an application with its own private interface. The desirable approach is the oposite: a sensor should serve any application that requires its data using a well known interface. For this purpose a middleware solution is needed.

Nowadays, two trends are emerging in automotive industry: electrification and autonomous driving. Both cases means more sensors and software to deal with these sensors. Also in this context the concept of a middleware makes sense to connect sensors and applications. More specifically, in this dissertation it is shown how Robot Operating System (ROS) can be used to bridge the gap between the in-vehicle sensors and the applications that process the data from those sensors. Through a distributed architecture, remote interaction between different components is possible, thus facilitating resources allocation and management.

The project on which the work developed during this dissertation focuses, is part of the *EasyRide Program*, the result of a partnership between the University of Minho and Bosch.

KEYWORDS Middleware, Robot Operating System, Data Acquisition System.

RESUMO

Atualmente, vive-se uma época em que os sensores e as aplicações que tiram partido deles são cada vez mais parte integrante do nosso quotidiano. Tornando-se natural contactar com uma vasta gama de dispositivos no nosso dia-a-dia, o crescimento da área aplicacional facilita o processo de interação entre os diferentes dispositivos. Assim, é cada vez mais comum estar-se rodeado por vários sensores, como sensores de imagem, som ou ainda sensores de luminosidade ou movimento, entre diversos outros tipos. É neste contexto que surge o desafio de criar soluções distribuídas para lidar com esta grande variedade de dispositivos, tecnologias e aplicações.

No caso concreto do setor automóvel, o processo de monitorização dos passageiros no veículo é muito similar. Então, o objetivo deste projeto consiste em criar um sistema de aquisição de dados, para o setor automóvel, para monitorizar o interior do veículo, desde o seu estado de conservação até ao estado e comportamento dos passageiros.

Mais concretamente, o objetivo é mostrar como o Robot Operating System (ROS) pode ser utilizado para fazer a ponte entre os sensores dentro do veículo e as aplicações que processam os dados desses mesmos sensores. Através de uma arquitectura distribuída, é possível uma interação remota entre diferentes componentes, facilitando assim a alocação e a gestão de recursos.

O projeto sobre o qual incide o trabalho desenvolvido durante esta dissertação, faz parte do Programa *EasyRide*, fruto de uma parceria entre a Universidade do Minho e a Bosch.

PALAVRAS-CHAVE Middleware, Robot Operating System, Sistema de Aquisição de Dados.

CONTENTS

1	INTRODUCTION	2
1.1	Context and Motivation	2
1.2	Objectives	3
1.3	Dissertation Structure	4
2	STATE OF ART	6
2.1	Middleware	6
2.2	Apache Kafka	7
2.3	Main Concepts	10
2.3.1	Scalability	10
2.3.2	Component-based Software	11
2.3.3	Distributed Systems	11
2.3.4	Message Passing	12
2.3.5	Real-time Systems	14
3	SOFTWARE DEVELOPMENT METHODOLOGY	15
4	ROBOT OPERATING SYSTEM 2	17
4.1	Design and Architecture	17
4.2	Main Concepts and Communication	18
4.2.1	Nodes	18
4.2.2	Topics	19
4.2.3	Parameter Server	20
4.2.4	Bags	20
4.2.5	ROS Middleware Interface	20
4.3	Package Structure	21
5	PACKAGES MIGRATION	22
5.1	Audio - audio_common	22
5.2	Image/Video - avt_vimba_camera	24
5.3	Package Migration Guidelines	25
6	HEALTH MONITORING	26
6.1	Requirements Specification	26
6.2	Architecture	28
6.2.1	Health Monitoring Node Machine	29
6.2.2	Health Monitoring Node Sensor	29
6.2.3	Health Monitoring Messages	30

6.3	Results	31
7	SENSORS SYNCHRONIZATION STRATEGY	35
7.1	Architecture	35
7.2	Cameras Synchronization – Precision Time Protocol	36
7.3	AVSync Generator	37
7.4	Blob Detector	38
7.5	Amplitude Getter	39
7.6	Results	41
7.7	Improvements	44
8	USER INTERFACE	47
8.1	Init	48
8.2	Preview	49
8.3	Record	50
8.4	Review	53
8.5	Reboot/Shutdown	54
9	RESULTS AND DISCUSSION	56
9.1	System Performance	56
9.1.1	Sensors Configuration	57
9.1.2	Test Scenarios and Setups	60
9.1.3	Performance Tests Results	63
9.1.4	Performance Improvements	73
9.2	Data Campaigns	74
9.2.1	Software Architecture	74
9.2.2	Hardware Architecture	75
9.2.3	Final Results	77
10	CONCLUSION	78

LIST OF FIGURES

Figure 1	Different messaging patterns: (a) request-response and (b) publish-subscribe.	13
Figure 2	Design Science Research Methodology model (adapted from [1]).	15
Figure 3	Communication structure of Robot Operating System.	19
Figure 4	Code changes after the audio package migration process from <i>ROS1</i> (top image) to <i>ROS2</i> (bottom image).	23
Figure 5	The general architecture of the Health Monitoring subsystem.	28
Figure 6	The architecture of the Health Monitoring Node Machine.	29
Figure 7	The architecture of the Health Monitoring Node Sensor.	30
Figure 8	Results from the HM Node Machine subsystem.	32
Figure 9	Results from the HM Node Machine subsystem with <i>plotjuggler</i> tool.	33
Figure 10	Usage of <i>ROS2</i> Node List command.	33
Figure 11	Usage of <i>ROS2</i> Topic List command.	33
Figure 12	Results from the HM Node Sensor subsystem.	34
Figure 13	The architecture of the synchronization strategy.	35
Figure 14	Precision Time Protocol Synchronization (from Allied Vision ¹).	36
Figure 15	Time diagram of the cameras synchronization process.	37
Figure 16	Frames with the pendulum at different coordinates.	37
Figure 17	Illustration of blob detection.	38
Figure 18	The pendulum's coordinates over the time.	39
Figure 19	Audio amplitude over the time, with two different amplitudes/frequencies magnified.	40
Figure 20	Excerpt from the audio graph, with the identification of amplitude/frequency change points.	41
Figure 21	Lag over time between two cameras synchronized using <i>PTP</i> .	42
Figure 22	A result from the Blob Detector.	42
Figure 23	Synchronization data from one camera and the microphone.	43
Figure 24	Astable Timer 55 circuit diagram (from Circuit Digest.	45
Figure 25	LED frames: (a) LED off and (b) LED on.	45
Figure 26	Initial screen.	47
Figure 27	System ready to initialize confirmation.	48
Figure 28	Skip initialize confirmation.	48

Figure 29	Wait 2 minutes until the system is ready.	48
Figure 30	Screen when the system is running.	49
Figure 31	Image preview of fisheye camera.	50
Figure 32	Insert ID for record.	51
Figure 33	Invalid ID confirmation.	51
Figure 34	Calibration record ID confirmation.	51
Figure 35	Test record ID confirmation.	52
Figure 36	Record screen, with Stop button unlocked.	52
Figure 37	Record performance statistics and confirmation to keep or delete record.	53
Figure 38	Insert the record ID for review.	53
Figure 39	Front camera review.	54
Figure 40	System reboot confirmation.	55
Figure 41	System shutdown confirmation.	55
Figure 42	Illustrative image of the Allied Vision MAKO G-234c camera.	56
Figure 43	Illustrative image of the miniDSP UMA-8 microphone-array.	57
Figure 44	Illustration of different audio sampling frequencies [2].	59
Figure 45	Camera's FPS_{error} under normal conditions, when using ROS2 and only the Vimba SDK.	64
Figure 46	Camera's FPS_{error} under lightweight conditions, when using ROS2 and only the Vimba SDK with CPU load.	66
Figure 47	Microphone's CPS_{error} under normal conditions.	68
Figure 48	Microphone's CPS_{error} under lightweight conditions.	69
Figure 49	Results from the minimal setup under normal conditions, with recording.	71
Figure 50	Results from the minimal setup under lightweight conditions, with recording.	72
Figure 51	Illustration of software operation.	75
Figure 52	Layout of sensors in the van.	76
Figure 53	Van trunk with hardware components.	77

LIST OF TABLES

Table 1	List of status data required per hardware type.	27
Table 2	Health monitoring machine message structure.	31
Table 3	Health monitoring sensor message structure.	31
Table 4	Excerpt from a CSV file with the coordinates of pendulum.	43
Table 5	Sensor configuration values used during the performance tests.	60

ACRONYMS

A

API Application Programming Interface.

C

CPU Central Processing Unit.

CSV Comma-separated values.

D

DCE Distributed Computing Environment.

DDS Data Distribution Service.

DNS Domain Name System.

DSR Design Science Research.

F

FPS Frames per Second.

G

GPS Global Positioning System.

GPU Graphics Processing Unit.

I

IDL Interface Definition Language.

M

MOM Message Oriented Middleware.

MTU Maximum Transmission Unit.

N

NTP Network Time Protocol.

O

OMG Object Management Group.

OSF Open Software Foundation.

P

PTP Precision Time Protocol.

Q

QoS Quality of Service.

R

RAM Random-access Memory.

ROM Read-only Memory.

ROS Robot Operating System.

ROS₁ Robot Operating System - version 1.

ROS₂ Robot Operating System - version 2.

RPC Remote Procedure Call.

S

SSH Secure Socket Shell.

T

TCP Transmission Control Protocol.

U

UDP User Datagram Protocol.

INTRODUCTION

This first chapter begins with a contextualization of the project and with the reasons that led to the realization of this dissertation. Next, the objectives of this study are presented. This chapter finishes with the presentation of the structure of the entire document.

1.1 CONTEXT AND MOTIVATION

With the constant advances in technology, it is increasingly common for us to live our daily lives surrounded by sensors of various types, even if we do not have this perception. Using as an example a home automation system we can find from a simple light sensor, to motion sensors, video cameras, among many others.

This development in technology is also present in the automotive domain, where numerous sensors are used for different purposes. This project focuses on monitoring the interior of the vehicle, where, soon, in a context of shared autonomous vehicles, passengers unknown to each other will share the same vehicle at the same time and throughout the day. Thus, it is important to have a set of sensors that will monitor not only the state of the interior of the vehicle, but also the state (i.e., alive, very sick, death...) and actions (e.g., aggression) of the passengers.

This dissertation focuses on the creation of an infrastructure that allows the creation of interfaces and mechanisms that facilitate the integration of data producers and data consumers, thus enabling a scenario where sensors make their data available to the applications that require them. This distributed context of great heterogeneity of devices, technologies and applications, is a challenge for the creation of solutions, dealing also with scalability of the context as well with the scalability of the solution. In this sense, the use of a middleware, is one way to tackle these requirements, providing an infrastructure that offers a set of services for the creation of distributed and heterogeneous solutions.

Although the *Robot Operating System (ROS)* framework [3] was created for the field of robotics it is also being used for research purposes by the automotive industry, particularly for research on autonomous driving, which concerns itself with what is outside of the vehicle. As an example, there is the case of Autoware [4] that provides open-source software, based

on *ROS2*, and intended for the development of autonomous driving technologies. Similarly, Apex.Ai [5] operates in the same area, also taking advantage of the *ROS* framework to develop new autonomous driving solutions. This dissertation, however, concerns itself with what is inside of the vehicle, where *ROS* act as middleware between the in-vehicle sensors (more specifically, their data) and the applications that need them.

ROS provides an infrastructure for robot control software, as well as a set of support tools and libraries for the development of robotic systems and applications. That is, it includes: a middleware layer for message/data exchange through a publish/subscribe messaging pattern; a set of tools to support the configuration, testing, debugging, visualization and start-up and stop of distributed computing systems; and a set of libraries/packages to support the implementation of a robotic systems, such as mobility, handling ability, perception, and many others.

There are two versions of *ROS*, *ROS1* and *ROS2*, and *ROS2* makes a modular redeployment, introducing, among other improvements, the notion of modularization of middleware and of quality of service. This framework is organized in packages and, although the two versions of *ROS* start from very similar principles, the packages are incompatible, so it is necessary to migrate packages from *ROS1* to *ROS2* (when a *ROS2* version of the package is not available).

1.2 OBJECTIVES

The main objective of this dissertation is the creation of a data acquisition system, based on *ROS2*. This data acquisition system is based on a set of sensors installed inside a vehicle. The sensors mentioned can be of various types, including RGB cameras, NIR and thermal cameras, microphones and microphone arrays, etc., depending on the particular application(s).

The following tasks were defined to be the main necessary steps to achieve the desired goals:

- Detailed study on *ROS2*, focusing on middleware, and the structure and creation of packages.
- Implementation/Migration of a *ROS2* package for the RGB camera;
- Implementation/Migration of a *ROS2* package for the microphone array;
- Development of health monitoring mechanisms in order to monitor the active sensors and machines in the system;
- Development of a mechanism capable of ensuring synchronization between image and sound data;

- Development of a graphical user interface;
- Analysis and improvement (if necessary) of the performance of the developed system.

1.3 DISSERTATION STRUCTURE

Regarding the structure of this document, it is divided into ten chapters.

In the first chapter (1), the context and motivation of this dissertation is presented, followed by its objectives. At the end of this chapter, there is this section, where the structure of this document is presented.

In chapter two (2) the state of the art is presented, with the main concepts that are the basis of this dissertation. In this way, the concept of middleware is explained, presenting different types of middleware and a concrete example of a framework that could have been used in the context of this project.

The third chapter is Software Development Methodology (3). Here is presented the research/development methodology adopted in this project and that served as support to all the options taken throughout the development of this dissertation.

Next comes the chapter four (4) entirely dedicated to the framework that serves as the basis for the entire system, the Robot Operating System 2 (*ROS2*). Here is presented its architecture, along with the main concepts to understand the operation of this software.

In chapter five (5) begins to be presented the work developed in this project. The migration processes of audio and image packages from *ROS1* to *ROS2* are described. This chapter ends with a set of guidelines that should be used in the process of migrating packages.

In chapter six (6), the system monitoring mechanism is presented. This mechanism is responsible for monitoring all system components, from machine resource consumption to the performance of sensors responsible for data acquisition.

Since one of the objectives of the system is to capture various types of data (e.g., audio and video) it is necessary to ensure the synchronization of the same. Thus, in chapter seven (7), the synchronization strategy of the various sensors, such as cameras and microphones, is presented.

In order to make the system developed user-friendly, a graphical interface was developed that exports all the functionalities of the system, in an intuitive way for the user to interact. The development of this interface is presented in chapter eight (8).

In chapter nine (9) the results section appears. Here are presented and analyzed the initial results of the system, in relation to its performance, followed by improvements that were implemented to improve the system performance. This chapter ends with the presentation of the architecture that was used during the Data Campaigns that were carried out, since this project is inserted in the *EasyRide Program*, the result of the partnership between the University of Minho and Bosch.

The document ends with the chapter of the conclusion (10), where a critical analysis of the work presented in this dissertation is made, focusing on points that may constitute improvements for the system in the future.

STATE OF ART

In this chapter is made a more presentation of the state of the art, resulting from the study of the main concepts of this project. Thus, the first step is to deepen the concept of middleware, exploring the different existing types, characteristics of each, as well as strengths of each type.

Next, is dedicated a section to Apache Kafka, with a more in-depth analysis, highlighting the pros and cons of using this software as middleware in the context in which this dissertation is included.

To end this chapter, are presented the main concepts of the project behind this dissertation with an explanation and the importance of each one of them.

2.1 MIDDLEWARE

Formally, Middleware is the software that resides between the operating system and the applications that run on it [6]. The concept of middleware has abroad scope. Here we use it as a software layer that resides between the operating system and the applications to support the development of distributed applications, providing to them an uniform interface, and mechanisms to their execution. There are distinct taxonomies of middleware as its categorisation is difficult. For the purpose of this brief presentation we will refer just *Remote Procedure Call (RPC)* and messaging *Message Oriented Middleware (MOM)*, not only because they have distinct communication paradigms but also because *RPC*-like middleware is the oldest, and *MOM* is one of the most popular types of middleware. *RPC* extends to a distributed environment the concept a local function call: an invocation is made and the function waits, synchronously for an answer in the context of the same transaction. *Distributed Computing Environment (DCE)* from *Open Software Foundation (OSF)*/Open Group, was the first implementation, being *DCOM* from Microsoft and its main competitor *CORBA* from *Object Management Group (OMG)*, and Java RMI other implementations.

Message-oriented middleware supports two different communication models, message queuing and publish/subscribe messaging and is based on an asynchronous paradigm and loose coupled components.

The context of this dissertation, as mentioned above, aims at the creation of a distributed and heterogeneous data acquisition system. In this sense, in the following sections will be presented two examples of almost antagonistic technology in terms of the scale of data with which they have to deal and which are solutions for the heterogeneity and distribution inherent to these architectures. Therefore, these tools can be used in the context of the theme of this dissertation but at different scales, that is, the Kafka used as a platform for massive data collection in an industrial context and the *ROS2* used in a much narrower context. For example, a robotic/car vehicle.

2.2 APACHE KAFKA

In a classic industrial context, you can find a significant set of machines producing maintenance and production data massively, each with its own format and with its application that understands/uses them. This existence of data islands is incompatible with an integrated view of the production system and with the use of this data for the most diverse purposes, from production analysis and maintenance strategies, to machine learning-based predictive systems for the most diverse purposes.

In this context, event streaming technologies, which allow real-time data capture of sensors and other sources, are backbone of a distributed and heterogeneous environment allowing different applications to have access to data for different purposes.

As has been seen currently there are several methods to establish communication between different processes or applications. This section addresses the topic of message queues, along with Apache Kafka software [7], a middleware that takes advantage of this message queue processing system [8] to process large amounts of data.

In this system, there is a process responsible for sending the message (producer) and another process responsible for its processing (consumer). The message queue is what allows these two processes to communicate with each other: the producer puts your message in the queue, so that it can be processed later by the consumer. An important point in this architecture is the ability to share a message queue between multiple producers and consumers, thus enabling the implementation of communication under broadcast and unicast paradigms. This solution is also characterized by the ability to store messages, since, unlike traditional question/answer systems, in which communication between producer/consumer is made through sockets with *TCP* and *UDP* protocols, here the messages are stored in a kind of buffer until they are processed by the consumer or removed. This scenario also causes an asynchronous process, allowing multiple messages to be stored consecutively without the previous ones being processed by the recipient. This is another point where it differs from that of question/answer systems, since the processes of production and consumption of messages can work independently, without the need for synchronized

execution. These characteristics become particularly interesting when certain scenarios are studied in which there are failures in communication between origin and destination. Because messages remain stored in the buffer and processes work asynchronously, the consumer can reprocess the message, even if the previous attempt failed, without interfering to the normal implementation of the producer process.

The message queue system has the ability to distribute work. With this, you can use several smaller programs, all independent, to divide the load of the full execution of the process. This allows you to prevent a single program from performing all tasks sequentially, taking advantage of the parallel execution of multiple programs that, at the end of performing their function, send the result in the form of messages. This division of tasks is facilitated by the fact that there is no need for direct connections between the various programs, whether producers or consumers. Because all messages are buffered, programs can continue to run without having to make interruptions to send or receive messages from other stakeholders. Thus, the program does not need to process the message at the exact moment of its receipt, but at the most appropriate time, depending on the task you are performing. The fact that messages remain in the queue until they are removed by another program, enhances the exploitation of this form of asynchronous execution.

As already started above, programs don't need to process messages the instant they receive them. This producer/consumer communication can be event-oriented and controlled by the current state of the message queue. Thus, one program may be set to process the message at the time of its receipt, and another, fully independent, has indication to start running when there are more than N messages in the queue. With this, you can introduce another very important feature: the notion of priority between messages. In a scenario where there are multiple queued messages, programs can consume messages according to their priority and not on a first-come, first-served basis. This priority is set by the program that queues the message, allowing it to have preference over others at the time of processing.

Another important factor is that some systems, such as Kafka, allow several message queues to exist. These systems also allow easy scalability when we talk about increasing message load, since message queues can be scaled horizontally, also taking advantage of the fact that communication between the various entities occurs indirectly and asynchronously through the message queue.

After the general characterization of these messaging queue systems, highlighting their main advantages, follows the presentation of the concrete case of Apache Kafka software, a concrete example of the use of this type of communication between processes.

Apache Kafka software was developed from the first moment to be able to handle a high load of messages, having as main characteristic its strong scalability. This software is responsible for establishing communication between the set of programs that produce the messages and consumers thereof. In Kafka there are topics where producers publish

their messages, and consumers can subscribe and listen to these topics in order to receive the desired message. It is important to note that each consumer can read messages from various Kafka topics. These topics can be constituted by multiple partitions and, internally, a partitioned commit record (log) is made. From a more practical point of view, a partition is no more than an ordered sequence of messages that is continually attached to the commit log. In order to preserve the order of messages, exclusively within each partition, an identification number is assigned to each message in the partition, called by offset. This type of internal topic partitioning is important to preserve system scalability. Each producer can, in addition to the topic, choose which partition sends the message to, thus taking advantage of parallelism, being able to divide the messages by different processes. Each message is kept in the Kafka cluster (Kafka topic set) for an arbitrary period and is later deleted to free up space for new messages. Thus, taking advantage of the offset variable, consumer programs can read older or newer messages, varying only the value of this parameter. This offset value allows you to ensure that messages will be attached to the log in the order in which they were sent, thus causing consumers to see messages in the correct sending order.

In Kafka software, the servers responsible for ensuring the transfer of messages between producers and consumers are called brokers. These brokers are responsible for the persistence and replication of messages. The brokers are organized in a decentralized way and a distribution of all partitions is made between the brokers, each being responsible for the storage of one or more partitions. To build a fault-tolerant solution, partitions are replicated in different brokers, one of which is designated as the leader of a certain partition. This leading broker saves the record of all reads and writes to the partition assigned to it. Because the log of each partition is replicated by multiple brokers, a message can only be processed by the consumer when it is committed by all brokers that contain the replica of their partition. During this process, producers can choose to block until the message is committed by all replicas of the log or, in return, continue to transmit messages without blocking their normal execution.

In this architecture, producers are aware of the topics and their existing partitions, thus taking advantage of the Kafka API to send their messages. Through the *API* the producers can know which brokers are responsible for each partition and thus direct the message to the broker responsible for the partition to where the producer wants to send the message. Producers are also allowed to send batch messages. This is how an accumulation of messages is sent simultaneously.

For consumers, they request blocks of messages from brokers regarding the partitions from which they will read the messages and can read all the messages present in the log of that partition through the offset parameter, as mentioned earlier.

2.3 MAIN CONCEPTS

In this chapter are presented some important concepts related to the scope of this dissertation. Here are covered from development concepts, like scalability (2.3.1), to software concepts, like distributed(2.3.3) or real-time systems(2.3.5).

2.3.1 Scalability

One of the objectives of this project is that the developed data acquisition system follows a distributed architecture, and the number of components can grow easily. It is in this sense that the concept of scalability arises.

To better explain the meaning of this concept, three definitions of different authors (Bondi's [9] first and followed by El-Rewini and Abd-El-Barr [10]) are presented:

- The ability to handle more work by adding more resources to the system (e.g., two servers can handle more clients than one server).
- The ability to handle more work without changing the system (e.g., an oversized server may handle uncommon but very heavy workloads).
- The effort required for an existing system to handle more work or provide more functionality (e.g., a distributed system is often easier to scale than a monolithic system).

Therefore, considering these three definitions, the work rate that a system can handle in a given period of time can be increased in three different ways:

- Improving software
- Improving hardware (vertical scalability)
- Adding more hardware (horizontal scalability)

Sometimes the only chance to increase this rate is even adding more hardware to the system. However, it is important to understand the impact of this decision. To be an option with a high cost-benefit it is important that the system is dividing the workload, acting in a mostly parallel and non-sequential manner.

In this project, scalability is horizontal, being achieved through a distributed peer-to-peer architecture using *ROS2*.

2.3.2 *Component-based Software*

The system developed during this dissertation follows a component-based architecture. Thus, in this type of architectures, each component is responsible only for a portion of the total functionality of the system, communicating and exchanging data with the remaining components through well-defined interfaces. In this way, it has a supplied interface and an interface required.

Component-based software has the following advantages:

- Separation of concerns: each component is responsible for only a part of the total system functionality. Thus, the system is split into smaller problems/components that are more manageable.
- Loose coupling and encapsulation: components are defined by their interface, not their implementation. Thus, components can be developed and tested independently.
- Facilitates reuse: effective reuse, however, depends on well-defined interfaces. It is well known that a significant effort and expertise is needed to define widely reusable interfaces.
- Facilitates modification and evolution: one component can be replaced with a different version on the conditions that the provided interface is the same and the required interfaces are available.
- Increased test coverage and reliability: reusable components can be evaluated and tested by more parties.

To take advantage of this concept, this project takes advantage of *ROS2*'s publish-subscribe message pattern and the package management system also made available by *ROS2*. The first point allows separation of concepts, loose coupling, and encapsulation. The second facilitate the modification and evolution.

2.3.3 *Distributed Systems*

As seen before, the developed system follows a distributed architecture. This type of systems are characterized by the fact components are distributed among different networked computers, or nodes, which communicate with each other through message passing (2.3.4) [11]. As examples of distributed systems, there are the world wide web, most multiplayer online games, aircraft and car control systems, industrial control systems, among many others.

Distributed systems have the following characteristics:

- Concurrency and high performance: different computers run concurrently to each other. However, synchronization is required between the different computers, which, if not done appropriately will result in very low system performance.
- There is no global clock: each computer has its own particular clock and clock synchronization is required to ensure that all clocks remain synchronized.
- Fault tolerance: the failure of a component does not necessarily mean the failure of the entire system (as seen before (2.3.3), each component is only responsible for only a part of total functionality of the system).

This project use *ROS2*, following a distributed peer-to-peer architecture, providing high performance, concurrency and fault tolerance.

2.3.4 Message Passing

Message passing is one of several techniques for invoking behaviour; other techniques include: direct and indirect invocation and CPU interrupts. Message passing enables, in particular: (1) objects¹ to communicate with each other within the same or different computers (intra- and inter-process communication); and (2) remote procedure calls. In message passing:

- The invoking object (the caller) sends a message to another object (the callee).
- The invoked object, based on the message received, selects and runs a function(s).

In synchronous messaging the invoking object waits (or blocks) until the function completes. In asynchronous messaging, conversely, the invoked object doesn't wait for the function to complete and continues execution (completion can be signalled by, for example, a callback). Synchronous messaging can be built on top of asynchronous messaging²; asynchronous messaging, however, requires buffering as well as a strategy to handle, among other things, a full buffer. In message passing an object model is typically used, distinguishing the functions (or services) in abstract from their concrete implementation. Objects can invoke functions with no concern about how those functions are actually implemented (encapsulation). An object model relies on an intermediate layer to select and run the appropriate function's implementation. This intermediate layer (a message broker or middleware) can distribute messages between objects, not only within the same computer but also across different computers³ (distribution). The performance of message passing is much lower than direct

¹ In this context, objects can also stand for: programs, processes, agents, actors, etc.

² With additional infrastructure (e.g., threads), it is also possible that asynchronous messaging is built on top of synchronous messaging, but this is rare.

³ Different operating systems, different programming languages, etc

invocation by name and this should be considered during the design and implementation phases. Message passing is best suited for small-size messages. Two of the most common messaging patterns are: request-response and publish-subscribe. The simplest messaging pattern, request-response, is illustrated in Figure 3(a). In this pattern, the server issues a response to a client request. A single entity can act both as a server and as a client, meaning that it can issue requests as well as responses.

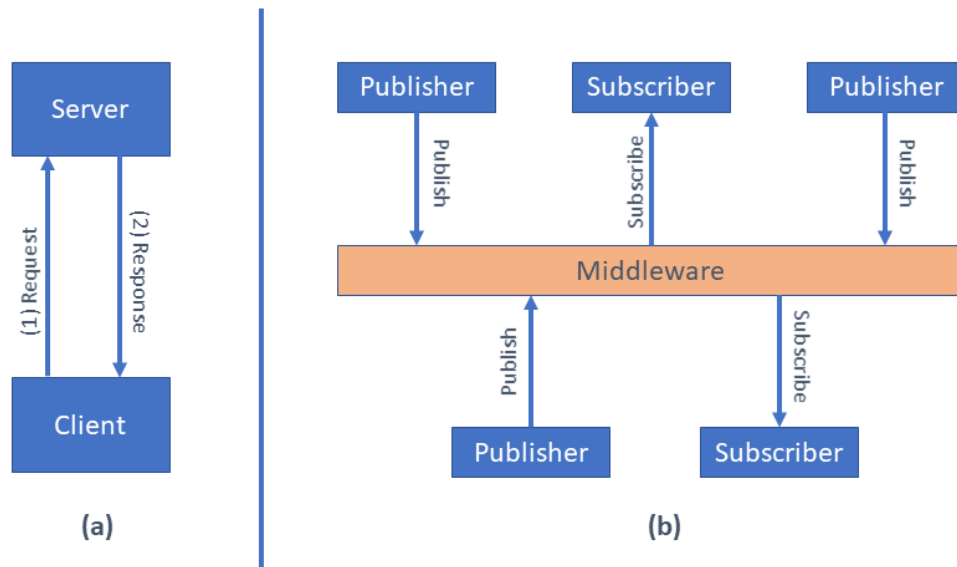


Figure 1: Different messaging patterns: (a) request-response and (b) publish-subscribe.

In a publish-subscribe messaging pattern, illustrated in Figure 3(b) publishers send messages, each message belonging to a specific class (or topic). Subscribers register interest in one or more classes of messages, and only receive messages of that class. Publishers don't explicitly specify the receivers of messages nor do receivers specify the senders of messages (loose coupling); instead, the middleware is responsible, based on the class of messages, to direct messages from the publishers to the appropriate subscribers. In this pattern, it is easy to add more publishers and subscribers, and thus, scalability is good. It is also possible to change the network topology with only a few modifications (e.g., moving a publisher or subscriber to a dedicated computer). One of the downsides is that, after a publisher has been deployed, it may be hard to modify that publisher and particularly the structure of the published data (many subscribers will have to be updated as well). Moreover, this pattern scales well for small networks and low message volume; however, as the size of the system increases the likelihood of instability increases (e.g., load surges, slowdowns).

This project follows a publish-subscribe messaging pattern, taking advantage of *ROS2* middleware capabilities'.

2.3.5 Real-time Systems

Real-time systems are useful when it is necessary to ensure that, in addition to logically correct operation, the system responds to certain events within a specific time period [12]. If the real-time system is unable to provide these guarantees, the system fails. In a real-time system predictability is more important than throughput or low latency. According to Shin and Ramanathan [13], real-time systems are usually distinguished between three classes, depending on the importance that a failure can have in the normal functioning of the system:

- Hard: failing to guarantee a deadline (a deadline miss) is a system failure (e.g., heart pacemaker, drive/fly-by-wire systems, anti-lock brakes).
- Firm: occasional deadline misses are acceptable (the quality of service may decrease). Results after the deadline must be discarded (no longer useful) (e.g., financial forecast systems).
- Soft: occasional deadline misses are acceptable (the quality of service may decrease). Results after the deadline are still useful (e.g., music/video player).

The difference between soft and firm systems lies in the usefulness of the result obtained after the deadline. The real-time system designed for this project takes advantage of *ROS2* real-time capabilities and fits into the soft category. Some occasional failures are acceptable, decreasing the quality of service but the results remain useful even after the deadline.

 SOFTWARE DEVELOPMENT METHODOLOGY

The research methodology represents the approach that is made to the problem that is intended to be studied. This methodology varies depending on the type of problem in question, and there are already several approaches to be taken depending on the type of case being investigated. Thus, this chapter describes the various stages of the *Design Science Research (DSR)* methodology, a methodology adopted in this project. The following figure 2 was taken from the article "A Design Science Research Methodology for Information Systems Research" [1] and represents all stages of this approach.

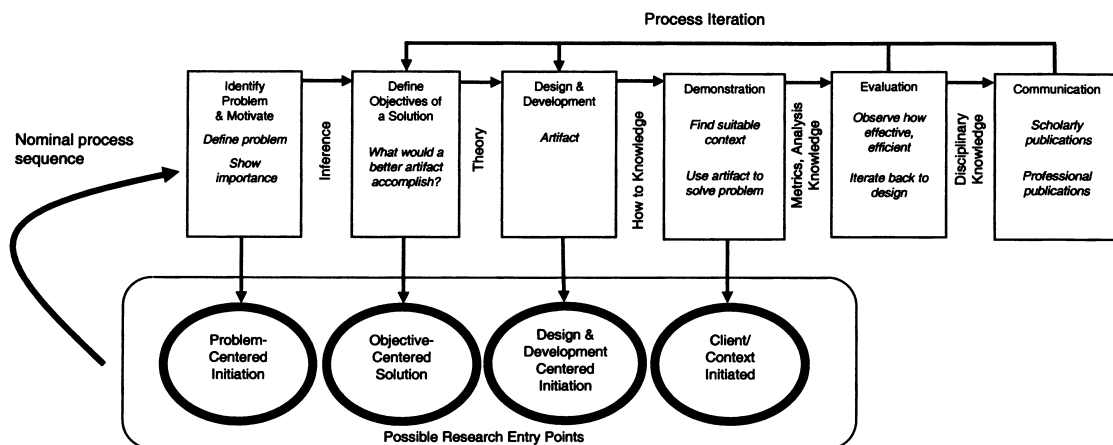


Figure 2: Design Science Research Methodology model (adapted from [1]).

As shown in the figure 2, the *Design Science Research (DSR)* methodology is based on a set of six steps that help to obtain a well-structured and reasoned solution for the case study to which it applies. Therefore, in order to allow a better understanding of this methodology of investigation, a brief presentation of each step is made:

1. **Problem identification:** this point is the basis of all development. It consists in the clear identification of the problem that is to be addressed and the reasons for doing so. It is important to present the case study in order to demonstrate the importance of

it and the relevance of the solution to be developed in the environment in which the problem is included.

2. **Definition of the objectives of the solution:** follows the first stage. Once the problem has been identified, it is necessary to understand what objectives are intended to be achieved with the solution that will be developed. Thus, at this stage it is important to study the possible solutions already existing to understand how one can develop something better or more relevant than the existing alternatives. This research also serves as a support for the motivation of the case study that is being addressed.
3. **Design and development:** in this stage begins the development of the solution. Ideally it begins with the idealization of the architecture to be followed until the development of a functional solution, according to the objectives defined in the previous point and that can be redefined.
4. **Demonstration:** a time when the solution developed to solve the problem proposed in whole or in a first phase is used in a partitioned form. As a first approach, test cases or simulation scenarios can be used to understand the effectiveness of the solution before going into the actual scenario. Important part where you can get a point of some points to improve in the architecture itself and in the development of the solution.
5. **Evaluation:** consists of analyzing the performance of the solution based on the results obtained in real environment. Whenever possible, it is important to have a comparison term with pre-existing solutions or fit-in alternatives. Point where the consistency of the solution is evaluated, from the proposed objectives to the designed architecture. At the end of this step, you can return to steps two or three to readjust the objectives you set earlier or restructure the architecture developed for the solution.
6. **Communication:** if the result of the previous stage is satisfactory, the problem is disclosed, highlighting its importance and the usefulness of the solution developed, supported by the results obtained in the previous points.

Since this is the natural structure of this methodology, it is important to note that, as the figure shows, it is not necessary to follow the methodology from the first point to the end. To exemplify, if it is supposed to develop a solution for a case study where the problem is already well identified and whose objectives have already been defined, there is no need to repeat the first two steps, and can immediately jump to the third stage of the development chain. According to the figure, this would be a startup centered on design and development.

ROBOT OPERATING SYSTEM 2

The *ROS* aims to create an integrative environment in a context of data acquisition. Unlike an event streaming processor like Kafka, *ROS* is not aimed at massive data retrieval, but rather a relatively modest environment in terms of data producers and consumers.

ROS consists of a set of tools, libraries and a middleware layer that aims to create robotic systems. As part of autonomous driving research programs, *ROS* is seen as an alternative given its past in the creation of mobile autonomous robotics. The context of this dissertation is based on the principle of the application of *ROS* in autonomous driving and takes advantage of its existence as a middleware system for an environment for data acquisition [14].

4.1 DESIGN AND ARCHITECTURE

For starters, *ROS* is advantageous because it is not limited by a specific language. Its developers designed it with the aim of being neutral in the language chapter. With this, it is possible to develop code in several languages, taking advantage of an *Interface Definition Language (IDL)* that makes a kind of "translation" of code, so that excerpts from programs written in several languages can interact with each other. Taking advantage of code generators for supported languages are also generated native implementations of objects that are realized and deployed by *ROS*, and then sent as messages (concept discussed later). This process allows to greatly reduce the writing time of code and consequently the number of programming errors. *ROS* still supports cross-platform, which can be an advantage for system integration.

For its architecture, *ROS* follows a distributed and component-based implementation, using a micro kernel design, with various modules, where several small tools are used to build and run software components. This effort to separate the various tasks by different modules, rather than concentrating everything on a master module, allows a significant gain in stability and complexity in program management, which ultimately compensates for the possible loss of efficiency when compared to a centralized architecture. The *ROS* package management's facilitates management and distribution of software.

Focusing attention on *ROS2* on its real-time support and security mechanism (namely, *DDS*-security support), are important points for choosing this version, rather than *ROS1*.

In addition to being free and open source, both for personal and commercial use, this software offers a great facility regarding code reuse. This is because the libraries used are developed independently and without *ROS* dependence. This puts all the complexity in the libraries, and then creates small executables in order to export the functionality of each library to *ROS*. Thus, the process of extracting and reusing code, outside its original context, is greatly simplified. This is also the reason *ROS* reuse code from various open source projects, such as drivers or OpenVC library algorithms, for example.

4.2 MAIN CONCEPTS AND COMMUNICATION

In order to better understand the functioning of *ROS*, it is necessary to know its main concepts.

4.2.1 Nodes

The main entity of *ROS* is node. These are responsible for the execution of the system and, as was seen earlier, a system is composed of several nodes, in order to maintain the distributed modular architecture presented.

In order to communicate with each other, nodes send messages with the desired information. By default, these messages adopt primitive types such as integers, floats or strings. However, you can also create custom structures to form a message, through arrays or matrices organized in the desired way. The communication is made via a publish-subscribe messaging pattern, enabling nodes to be developed independently and so to be also loosely coupled. At runtime, nodes are identified by a unique name (within a namespace).

In *ROS2* specifically, nodes can opt-in for a managed lifecycle, providing users with more control over the state of a *ROS2* system (i.e., nodes start executing only after all nodes have initialized). A managed node provides an interface which is called based on a known life cycle state machine. *ROS* services (e.g., publish, subscribe, the parameter server) are accessible through *ROS* client libraries. In *ROS1* there are *roscpp* (C++), *rospy* (Python), etc. And, in *ROS2* there are *rclcpp* (C++), *rclpy* (Python), etc. *ROS1* and *ROS2* interfaces are not compatible. *ROS1* interfaces were designed before February 2009 and some design decisions at the time are now known to not be the best. *ROS2* was an opportunity to improve on these interfaces. This meant breaking compatibility. There are, nevertheless, ways for *ROS1* nodes to communicate with *ROS2* nodes, and vice-versa; more specifically, using the *ros1_bridge* (a *ROS2* package) which translates topic/message data across the two (incompatible) systems.

4.2.2 Topics

To make communication between nodes possible, another entity emerges: the topics. These topics are a kind of "channel" where nodes publish messages so that other nodes can read them. Thus, each topic can have one or more nodes to publish or read messages. Just as a topic can be associated with one or more publishers, a node may also be publishing messages in more than one topic. This allows you to publish multiple messages simultaneously with different information, thus establishing communication with multiple nodes at the same time. For subscribers the process is identical, being possible to subscribe to several topics, thus receiving several messages simultaneously. The following figure (3) exemplifies this process of communication between nodes, through the respective topics.



Figure 3: Communication structure of Robot Operating System.

Topics are identified by a name (within a namespace) and each topic / name corresponds to one or more data type. Available data types include primitive data types (e.g., integers, booleans, strings), structures and arrays of primitive data types, and arbitrarily nested structures. There are also several types of topics: message (one-way), service (response-reply, remote procedure call), and action (response-reply-feedback, remote procedure calls). Examples of topics: data from sensors (e.g., images from cameras, audio from microphones), results of data processing, control signals; etc.

ROS2 has control over the quality-of-service. More specifically, it provides control over:

- History, Depth: how many samples to keep;
- Reliability: "best effort" or "reliable";
- Durability: "transient" or "volatile".

In *ROS*, after name lookup (similar to *DNS*), nodes communicate directly with each other. In *ROS1*, name registration and lookup are provided by the *ROS* master (a special-purpose node). In *ROS2*, conversely, the *ROS* master is not necessary; name lookup and registration are done via multicast (*DDS*-specific). In *ROS*, the *ROS* master is implemented using

XMLRPC (a stateless, HTTP-based protocol); nodes, however, exchange messages using a different protocol (i.e., TCPROS or UDPROS).

4.2.3 *Parameter Server*

In *ROS*, the parameter server stores data (e.g., configuration parameters, state, etc.); data is stored by key (similar to a dictionary). In *ROS1*, the parameter server is part of the *ROS* master and, like the *ROS* master, implemented using XMLRPC. *ROS2*, on the other end, relies on the global parameter server (a dedicated node).

4.2.4 *Bags*

In *ROS*, a bag is an actual file or file format. It supports recording/capture and playback of data, and thus, facilitates development and testing. In *ROS2*, the default storage plugin is *sqlite3*, and the SQLite settings are optimized for performance, by default.

4.2.5 *ROS Middleware Interface*

ROS services are accessible via *ROS* client libraries. In *ROS1*, services are implemented on top of custom protocols: TCPROS/UDPROS (persistent, stateful TCP and UDP socket connections). These were built almost entirely from scratch and the *ROS* master is required (a single point of failure)

ROS2 services, conversely, are not tied to a specific protocol. *ROS2* services are built on top of existing *DDS*-based middleware solutions (though other solutions can be used). This approach was chosen in order to leverage existing, mature solutions, each with its own advantages and disadvantages. Users are thus free to choose the solution which better fits their needs (cost, footprint, dependencies, legal requirements, etc.). Moreover, the project's overall maintenance effort is lower. Existing solutions include:

- eProsima Fast RTPS (Real Time Publish Subscribe)¹;
- RTI Connex²;
- Eclipse Cyclone³.

By default, *ROS2* uses Fast RTPS, a *DDS* implementation and this project follows that implementation.

¹ <http://www.eprosima.com/index.php/products-all/eprosima-fast-rtps>

² RTI Connex <https://www.rti.com/products/>

³ <https://projects.eclipse.org/projects/iot.cyclonedds>

DDS is a machine-to-machine standard whose goal is to be dependable, high-performance, interoperable, real-time, and scalable. *DDS* follows a publish-subscribe messaging pattern, and provides support for discovery (in other words, name registration and lookup), serialization, and transport. *DDS* also provides control over the Quality of Service: configuration, however, is complex and so *ROS2* provides some predefined *QoS* settings to simplify development. Unlike *ROS1*, a *ROS* master is not required in *DDS* (no single point of failure).

Despite the advantages of building *ROS2* services on top of existing *DDS*-based middleware solutions, different solutions expose slightly different interfaces. To address this, an abstract middleware interface was introduced. Furthermore, this abstract interface, hides the complexity and implementation-specific details of *DDS*. This interface is a function-based interface and thus facilitates interoperability with different languages.

4.3 PACKAGE STRUCTURE

To better understand the *ROS* architecture, it is also important to know the base structure of your packages. In general, a *ROS* package is composed of the following (depending on the case, there may be more, as we will see later on section 5):

- *package.xml*: this file consists of information such as the name, version, author, license, and dependencies of the package.
- *CMakeLists.txt*: is the input to a CMake build system for building software packages, which describes all its dependencies, how to compile the code and how to install the necessary files.
- *src* folder: contains code of the package's nodes and related programs which enable their execution.
- *launch* folder: contains files that allow the execution of several nodes with a single command.

PACKAGES MIGRATION

The system developed under this project is responsible for the acquisition of data from sensors such as cameras and microphones, through *ROS2*. The sensors used already had compatible *ROS* packages implementations, but only for *ROS1*. Thus, to take advantage of these implementations, it was necessary to migrate the *ROS1* packages to *ROS2*. The migration was made to the *audio_common* and *avt_vimba_camera* packages, thus managing to acquire data from the UMA-8 microphone [15] and Mako G cameras [16], respectively.

To start the package migration, it is important to bear in mind the structure of a *ROS* package, explained earlier in section 4.3.

5.1 AUDIO - AUDIO_COMMON

The *audio_common* package contains two main sub-packages: *audio_capture* and *audio_play*. As their names imply, the first contains the features necessary for capturing audio through the microphone and the second for playing audio.

The first step of the migration of this package was to create a *ROS2* package with the default files and folders mentioned in section 4.3. It was then necessary to check the libraries used in *ROS1* and analyze whether they exist in *ROS2*. If they do not exist, *ROS2*-compatible libraries that perform the same function have to be found. In this package, it was necessary to replace the library *roscpp* with *rclcpp*, which is the corresponding in *ROS2*. All the others were supported in both *ROS1* and *ROS2*. With this, it was necessary to update the *CMakeLists.txt* file by adding the new libraries used, and removing the old ones. Finally, an analysis was made to the code implemented in the nodes, replacing the references to *ROS1* libraries with the corresponding references to *ROS2* libraries, as can be seen in Figure 4, with an excerpt of the code from the original package in *ROS1* (top image) and the corresponding code after the migration to *ROS2* (bottom image), with the respective changes highlighted.

```

6  #include <ros/ros.h>
7
8  #include "audio_common_msgs/AudioData.h"
9  #include "audio_common_msgs/AudioInfo.h"
10
11 namespace audio_transport
12 {
13     class RosGstCapture
14     {
15     public:
16         RosGstCapture()
17         {
18             _bitrate = 192;
19
20             std::string dst_type;
21
22             // Need to encoding or publish raw wave data
23             ros::param::param<std::string>("~format", _format, "mp3");
24             ros::param::param<std::string>("~sample_format", _sample_format, "S16LE");

```

```

6  #include <rclcpp/rclcpp.hpp>
7
8  #include "audio_common_msgs/msg/audio_data.hpp"
9
10
11 namespace audio_transport
12 {
13     class RosGstCapture : public rclcpp::Node
14     {
15     public:
16         RosGstCapture() : Node("audio_capture")
17         {
18             _bitrate = 192;
19
20             std::string dst_type;
21
22             // Need to encoding or publish raw wave data
23             _format = this->declare_parameter("format", "wave");
24             _sample_format = this->declare_parameter("sample format", "S16LE");

```

Figure 4: Code changes after the audio package migration process from *ROS1* (top image) to *ROS2* (bottom image).

5.2 IMAGE/VIDEO - AVT_VIMBA_CAMERA

The package responsible for support image/video capture is more complex than the audio's, presented earlier (5.1). It is important to understand its structure before going to the next steps of migration. Thus, in addition to the parts mentioned in section 4.3, this package also includes:

- *calibrations* folder: contains the files with the calibrations required for the cameras to use.
- *cfg* folder: integrates camera configuration files.
- *cmake*: contains the files needed to detect the type of architecture that is being used by the user.
- *include* folder: contains header files where data structures and methods are created to assist the nodes implementation in the *src* folder.
- *lib* folder: contains files that help the installation of the package depending on the architecture of the target machine.

Since this package is complex, only the essential points for migrating the package to *ROS2* will be addressed, referencing the respective files:

- *AvtVimbaCameraConfig*: The *ROS* library that required more attention was the *dynamic_reconfigure*. In the *ROS1* environment, this library aims to enable the update, at run-time, of the parameters of a given node, without having to restart it. Therefore, this library is an important point in the process of migrating the package to *ROS2*, so that it is possible to reconfigure the camera at run time, without having to restart the node.

To do this migration it was necessary to develop some new functionality in order to adapt the code to the *ROS2* architecture and keep the *dynamic_reconfigure* features active in *ROS2*.

The solution addressed, for the fact that the *dynamic_reconfigure* is not present in *ROS2*, was to take advantage of the *ROS2* equivalent built-in, the *ROS2* parameters, in order to read and update the camera parameters at run-time, without the need to interrupt the execution of the node.

- *Mono Camera*: In this class, is where the methods developed in the class "AvtVimba-CameraConfig" are called. This way the values of the parameters of the node that is running can be read and updated, without having to restart it.

- *Mono Camera Node*: In this class, the nodes that serve as the reference for the remaining classes are created, thus exporting all the functionalities contained in the package.

5.3 PACKAGE MIGRATION GUIDELINES

Regarding the process of migrating *ROS1* packages to *ROS2*, as seen in the previous sections (5.1 and 5.2), there are steps common to any migration. Thus, these steps are listed here.

Therefore, to migrate a *ROS1* package to *ROS2*, the following steps should be followed:

- Create the *ROS2* package with the files and default folders (see section 4.3).
- Check the libraries used in *ROS1* and analyze if they exist in *ROS2*. If they do not exist, it is necessary to find the libraries that perform the same (or equivalent) function in *ROS2*.
- After this analysis is done, it is needed to add the new dependencies of the libraries found to the *CMakeLists.txt* file and remove the old ones.
- Modify the paths of the header files that are included. The installation of *ROS1* and *ROS2* are different and, in some cases, the relative path of the files are also different.
- Finally, it is necessary to analyse the code implemented in the nodes, where references to *ROS1* libraries should be replaced by references to *ROS2* libraries and, if needed, modify the code implementation.

HEALTH MONITORING

Health monitoring aims real-time monitoring of software and hardware components, that make up the data acquisition solution.

In this context, it is essential to gather the requirements of the health Monitoring subsystem to organize all the work necessary for its implementation. Thus, according to the type of hardware that has been considered to be part of the data acquisition solution, the components that need to be monitored are the machines (or computers), where the software supporting data acquisition is running, and the sensors, whose performance and data quality is a key element for the data acquisition to be successful.

In the following sections it is presented the requirements defined for the health Monitoring subsystem, the architecture developed, and the results obtained after its implementation.

6.1 REQUIREMENTS SPECIFICATION

The first step in requirements gathering was to divide the requirements between monitoring requirements of the machines and monitoring the sensors. Thus, for machines, the requirements relate to the hardware resources used. Thus, the following features are monitored:

- CPU;
- Memory (RAM);
- Swap memory;
- Disk (ROM);
- Network.

As far as sensors are concerned, it is important to have access to:

- List of active nodes: to be able to identify which sensors are active.

- List of topics for each sensor: to identify the channels to which sensor data is being sent.
- Frequency of publication of each topic (i.e. frames per second or frame rate): To have access to the performance/quality of the data being published.

Table 1 summarizes the requirements described above, and you can observe in the second column the data for each requirement, with the respective units in which they are measured.

Table 1: List of status data required per hardware type.

Hardware	Status Data
CPU	Usage per CPU (%)
Memory (RAM)	Usage (%)
	Total (MB)
	Available (MB)
	Used (MB)
	Free (MB)
Swap Memory	Usage (%)
	Total (MB)
	Used (MB)
	Free (MB)
	Sin/Sout (MB)
Disk (ROM)	Usage (%)
	Total (MB)
	Used (MB)
	Read/Write (MB/s)
Network	Sent/Received (MB/s)
Sensors	Active Sensors
	Active Topics
	Publish Rate (frames per second/Hz)

For data regarding memory (RAM), it is important to note the difference between the following concepts:

- available memory: the memory that can be given instantly to processes without the system going into swap. This is calculated by summing different memory values depending on the platform and it is supposed to be used to monitor actual memory usage in a cross platform fashion.
- free memory: memory not being used at all, and that is readily available; note that this does not reflect the actual memory available (use available instead), i.e. $total - used$ does not necessarily match free.

6.2 ARCHITECTURE

Before presenting the architecture designed for the monitoring solution, it is important to note that this system was built from scratch and consists of a *ROS2* package, developed using the Python¹ and C++² programming languages. It also took advantage of some features present in *ROS2* to meet the proposed requirements, as will be seen later.

Development of the Health Monitoring subsystem starts with defining the general architecture. Thus, according to the architecture presented in Figure 5, it is possible to perceive the different layers that make up the Health Monitoring subsystem, more specifically:

- **ROS Health Monitoring Layer:** supports two separate stacks, one for machine monitoring and another for sensor monitoring, described in detail in the following subsections.
- **ROS Layer:** represents the *ROS* packages that supports the data acquisition from the sensor(s).
- **OS Layer:** denotes the operating system that supports the entire system.

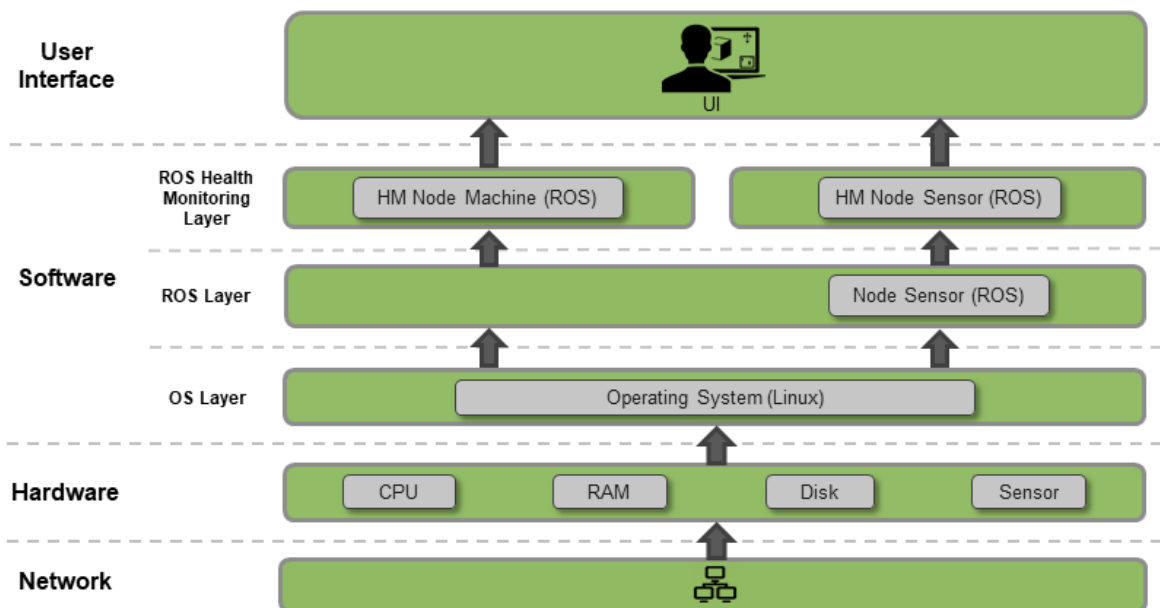


Figure 5: The general architecture of the Health Monitoring subsystem.

For the User Interface, refer to section 8.

As for the hardware layer, the machine components and sensors that are part of the data acquisition solution are represented, such as computers, cameras or microphones.

Finally, there is the network layer enabling different machines to interact with each other, allowing data to be exchanged between the machines and sensors connected to the network.

¹ <https://www.python.org/>

² <https://www.cplusplus.com/>

6.2.1 Health Monitoring Node Machine

Figure 6 illustrates the architecture designed to extract information about the resources used by the machine, namely CPU, Disk, Memory, Swap Memory, and Network. The implementation of this node was made using the Python language. For its implementation, there are two essential components:

- **Machine Monitor:** For the development of this node, a library called *psutils*³ was used, to retrieve the machine's status data from the operating system. With this, a publisher (i.e., a *ROS* node) was developed to retrieve the machine's status data and publish that under the topic */machine_state*.
- **Get Current State:** To view the information from the node described above, another *ROS* node was developed that subscribes to the */machine_state* topic and, when run by the user pretty-prints the status for the machine.

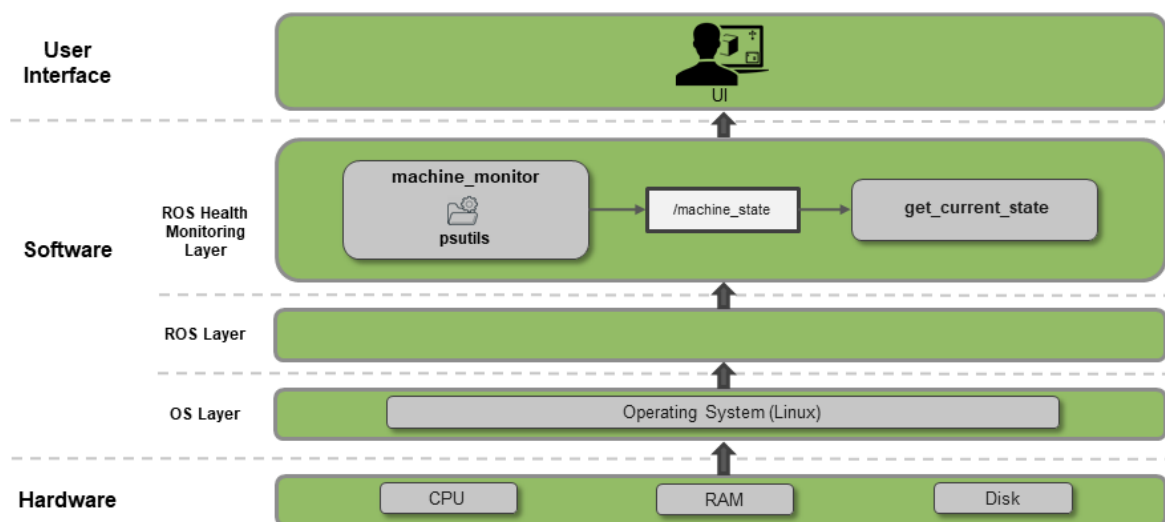


Figure 6: The architecture of the Health Monitoring Node Machine.

6.2.2 Health Monitoring Node Sensor

Figure 7 illustrates the architecture designed to monitor the status and data quality of a sensor, providing real-time access to a sensor's frame rate. These features were developed in a C++ subpackage. For its implementation, there are two essential components:

- **Sensor Monitor:** This *ROS2* node receives the name of the sensor to be monitored; then, it subscribes to the sensor's respective topic(s), calculates the frame rate at which

³ <https://psutil.readthedocs.io/en/latest/>

the sensor is publishing, and publishes that information, also adding the size of each frame, and a delta which corresponds to the time difference between each frame.

- **Get Current State:** This *ROS2* node displays the frame rate of the sensor specified by the user. To do this, it subscribes to the topic that the Sensor Monitor node described above and pretty-prints that information to the user.

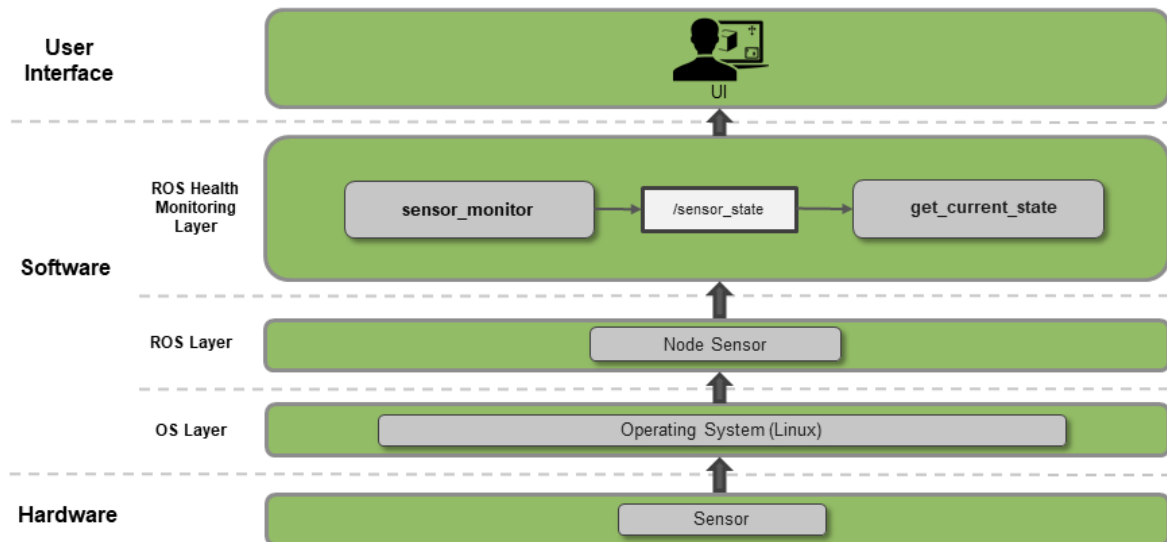


Figure 7: The architecture of the Health Monitoring Node Sensor.

To achieve the remaining requirements, standard *ROS2* tools⁴ are used to obtain the list of active sensors and their respective topics, namely:

- **ros2 node list:** presents a list with all active nodes, including the sensors' respective nodes.
- **ros2 topic list:** presents a list of all topics, including the topics published by the sensors' respective nodes.

6.2.3 Health Monitoring Messages

Since this package has been built from scratch, it has taken advantage of the capacity that *ROS* offers over custom messages. In this way, another subpackage was created to declare two data structures (one for the machines and one for the sensors) completely customized and adapted to this context. Each structure represents a custom message, and Table 2 and Table 3, show the variables present in each structure, as well as their data type.

⁴ <https://docs.ros.org/en/foxy/Concepts/About-Command-Line-Tools.html>

Table 2: Health monitoring machine message structure.

Variable	Data type
<i>mem_percent</i>	<i>float32</i>
<i>mem_total</i>	<i>float32</i>
<i>mem_available</i>	<i>float32</i>
<i>mem_used</i>	<i>float32</i>
<i>mem_free</i>	<i>float32</i>
<i>mem_active</i>	<i>float32</i>
<i>mem_inactive</i>	<i>float32</i>
<i>mem_cached</i>	<i>float32</i>
<i>swap_total</i>	<i>float32</i>
<i>swap_used</i>	<i>float32</i>
<i>swap_free</i>	<i>float32</i>
<i>swap_percent</i>	<i>float32</i>
<i>swap_sin</i>	<i>float32</i>
<i>swap_sout</i>	<i>float32</i>
<i>disk</i>	<i>float32</i>
<i>disk_used</i>	<i>float32</i>
<i>disk_read</i>	<i>float32</i>
<i>disk_write</i>	<i>float32</i>
<i>net_recv</i>	<i>float32</i>
<i>net_sent</i>	<i>float32</i>
<i>cpu</i>	<i>float32[]</i>

For the machine structure, the fields are those mentioned in Table 1, section 6.1, noting only the fact that a variable-sized array is created for the *CPU*, thus making this message universal for *CPU*'s with different number of cores.

Table 3: Health monitoring sensor message structure.

Variable	Data type
<i>fps</i>	<i>int64</i>
<i>size</i>	<i>int64</i>
<i>delta</i>	<i>float32</i>

For the sensor structure, there are three fields, as mentioned above (6.2.2). One for the frame rate, another for the size of each frame, and still a last one with the time delta between frames.

6.3 RESULTS

Regarding machine monitoring, to demonstrate the results obtained with this monitoring system, first, it is necessary to run the node *machine_monitor* of the package *machine_monitor*

(HM Node Machine). Then, running a listener/subscriber, the results shown in Figure 8 are obtained. In the figure, it is possible to verify that the machine that is being monitored has a *CPU* with 4 cores, through the size of the array present in the last field of the message.

```

Memory Percent: 54.200001
Memory Total: 8213393408.000000
Memory Available: 3762049024.000000
Memory Used: 3617955840.000000
Memory Free: 1640316928.000000
Memory Active: 4247437312.000000
Memory Inactive: 1523224576.000000
Memory Cached: 2739957760.000000
Swap Total: 2147479552.000000
Swap Used: 0.000000
Swap Free: 2147479552.000000
Swap Percent: 0.000000
Swap Sin: 0.000000
Swap Sout: 0.000000
Disk Percent: 75.599998
Disk Total: 490652.500000
Disk Used: 352252.906250
Disk Read: 0.000000
Disk Write: 0.000000
Network Received: 0.007571
Network Sent: 0.050761
CPU Usage:
  27.700001
  22.100000
  25.500000
  27.600000

```

Figure 8: Results from the HM Node Machine subsystem.

For a user-friendly interface, with the information provided by the above-mentioned package, it is possible to display the results in a graphical format using *plotjuggler*⁵, as shown in Figure 9. The graph on the left side, represents the *CPU* usage of each of the cores. On the right side, the percentage of disk usage, *RAM*, and swap memory is shown. This representation was enabled by creating a *ROS* message whose contents include several fields, each corresponding to each resource, as seen in section 6.2.3.

⁵ <https://plotjuggler.io/>

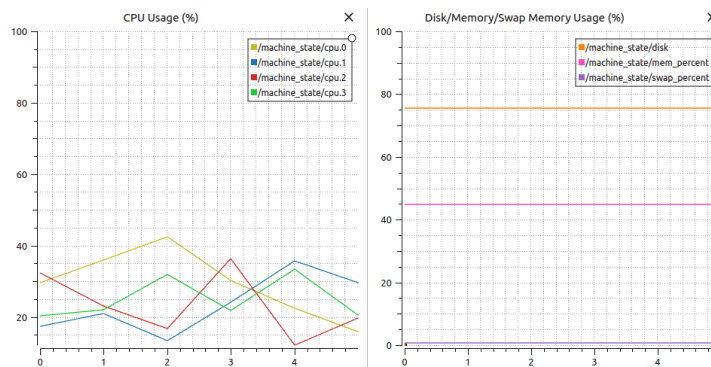


Figure 9: Results from the HM Node Machine subsystem with *plotjuggler* tool.

For sensor monitoring, at this stage, some requirements specified in section 6.1 can be fulfilled using *ROS2* tools, as shown in Figure 10 (the list of active nodes) and Figure 11 (the list of topics).

```
bruno@bruno-PU551JH:~/hm_sensor$ ros2 node list
/camera
/microphone_array
```

Figure 10: Usage of *ROS2* Node List command.

```
bruno@bruno-PU551JH:~/hm_sensor$ ros2 topic list
/camera_info
/image_raw
/image_raw/compressed
/image_raw/compressedDepth
/image_raw/theora
/parameter_events
/rosout
/sensor_state
```

Figure 11: Usage of *ROS2* Topic List command.

For continuous monitoring of the sensors, as described earlier, it is necessary to specify the sensor to be monitored to view its frame rate. As can be seen in Figure 12, using the *ROS2* topic echo command, the intended topic is subscribed and the frame rate is returned. A method was also developed to check when the frame rate drops below a given value, in relation to the previous one. This alerts the user when data acquisition from a given sensor begins to lose quality.

```
bruno@bruno-PU5513H:~/hm_sensor$ ros2 topic echo /sensor_state
data: 'Frame rate per second: 30'
---
data: 'ALERT - FRAME DROPS !! Less 5fps than previous measurement: 20'
---
data: 'Frame rate per second: 30'
---
data: 'Frame rate per second: 30'
---
data: 'Frame rate per second: 30'
---
data: 'Frame rate per second: 30'
---
data: 'Frame rate per second: 30'
---
```

Figure 12: Results from the HM Node Sensor subsystem.

SENSORS SYNCHRONIZATION STRATEGY

The developed Data Acquisition Solution has the ability to acquire different data types from different sensors simultaneously. Based on this functionality, it was necessary to develop a synchronization strategy of all acquired data, to reduce the lag that may exist between data coming from different sensors.

To better understand this, the designed architecture is explained first, with all components involved in the synchronization strategy. After that, an analysis is made of the results obtained with the application of this mechanism.

7.1 ARCHITECTURE

The developed synchronization strategy is divided in two distinct parts. The first concerns the synchronization of the cameras between each other and the second is related to the synchronization between the cameras and the microphone. In order to synchronize the cameras, the Precision Time Protocol (PTP) is used, as explained in more detail later (section 7.2). The architecture of the synchronization strategy between cameras and microphone is shown in Figure 13. It starts at least 2 sensors (1 camera and 1 microphone, for example) and the goal is to calculate the lag time between the data acquired by each sensor.

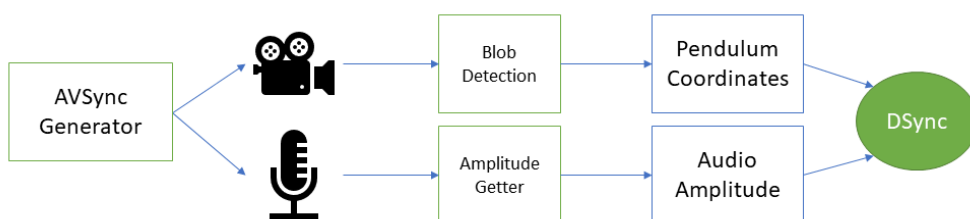


Figure 13: The architecture of the synchronization strategy.

7.2 CAMERAS SYNCHRONIZATION – PRECISION TIME PROTOCOL

The goal of the Precision Time Protocol¹ (*PTP*) is to manage and synchronize the clocks of multiple devices on the same Ethernet network. This protocol was created with the objective of improving upon existing methods. In comparison, this method offers more accuracy than the Network Time Protocol (*NTP*) and better accuracy/cost ratio than the Global Positioning System (*GPS*), which provides a fairly high accuracy, but at a high cost. Through *PTP*, it is possible to obtain clock synchronization at the microsecond level.

To obtain synchronization via *PTP*, whose protocol is illustrated in Figure 14, it is necessary to start by assigning roles to the cameras. One of the cameras must be classified as “Master” and the rest as “Slave”. After this assignment, the cameras classified as “Slave” will calculate the difference of their clocks to the clock of the “Master” camera and constantly adjust their clock. The cameras are then synchronized via *PTP* when the difference from the “Slave” camera clocks to the “Master” is within two microseconds.

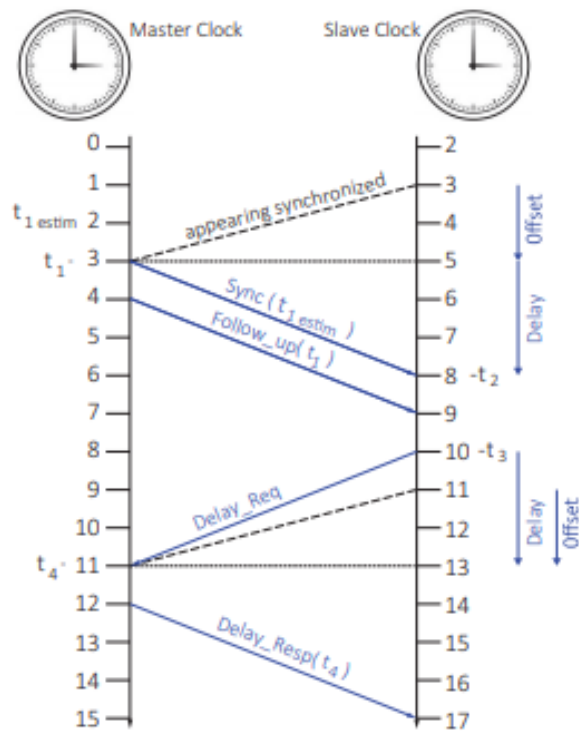


Figure 14: Precision Time Protocol Synchronization (from Allied Vision¹).

After reaching synchronization, the next step is to set the parameter “PtpAcquisitionGate-Time”, which represents the instant the cameras will start acquiring images. Please note that because the camera clocks are synchronized, this parameter must have the same value on

¹ https://cdn.alliedvision.com/fileadmin/content/documents/products/cameras/various/appnote/GigE/PTP_IEEE1588_with_ProSilica_GT_GC_Manta.pdf

all cameras, and it should be larger than the time required to reach synchronization, for the image acquisition to start at the same instant in all of them. The figure 15 shows a time diagram with all steps of this synchronization process.

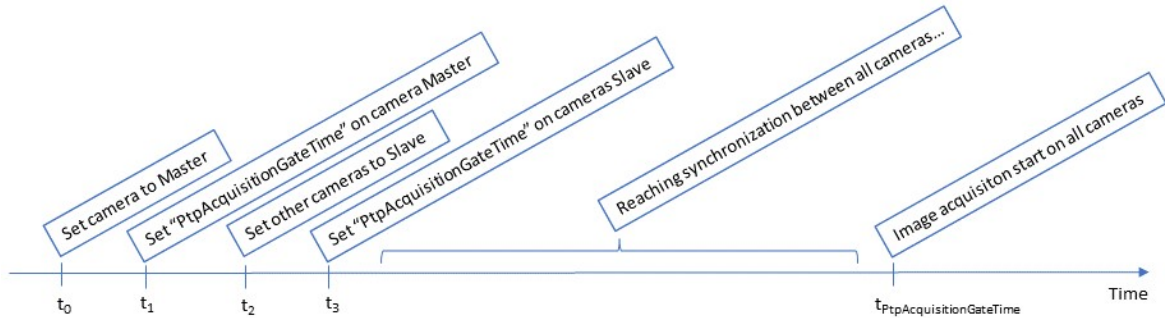


Figure 15: Time diagram of the cameras synchronization process.

7.3 AVSYNC GENERATOR

For the process of analysis and synchronization of the data acquired by the cameras and the microphone to be more intuitive, a test video with well-defined specifications was created in order to meet the desired objective.

For the video, a pendulum is used, with a controlled and repetitive movement, in order to allow the acquisition of a repetitive sequence of images by the camera. With the use of this pendulum, the goal is to get the exact coordinates of the object in each frame acquired by the camera. Figure 16 presents an excerpt with some frames of the video, where you can perceive the movement that the pendulum describes over time.

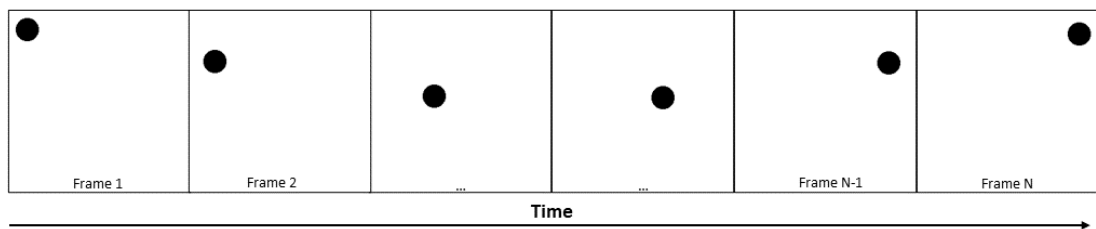


Figure 16: Frames with the pendulum at different coordinates.

As far as the audio is concerned, two audio samples with different frequencies and amplitudes were used. These audio samples were associated with the movement of the pendulum, one being reproduced when moving the object in the positive direction of the X

axis (left to right) and the other when the object moves in the opposite movement (right to left).

With this, it is possible to calculate the lag time between audio and video, taking into account the fact that the change in the direction of the movement of the pendulum and the change in the frequency/amplitude of the audio happen (or should happen) at exactly the same instant.

7.4 BLOB DETECTOR

Blob detection, in this context, is the process of analyzing the images acquired by the camera. This process consists of two parts: the first is the extraction of the images from the bag file in JPEG format, and the second is the analysis of the contents of each image.

For this, different tools based on the *OpenCV*² library were used.

First, after deserializing each message coming from the camera, the *ROS2's cv_bridge*³ package was used to convert it to *JPEG* format.

Second, for the analysis of each frame, the Blob Detector class, also provided by *OpenCV*, was used. The Blob Detector analyzes an image, depending on previously defined parameters and returns the coordinates of the objects or blobs that meet the criteria specified by those parameters. In this specific case, the parameters were defined for it to identify a black circle, and thus the coordinates of the pendulum in each image of the test video. In Figure 17 it is shown the result of this process. In each image, the pendulum is identified by a red circumference, which means that the Blob Detector was able to detect it.

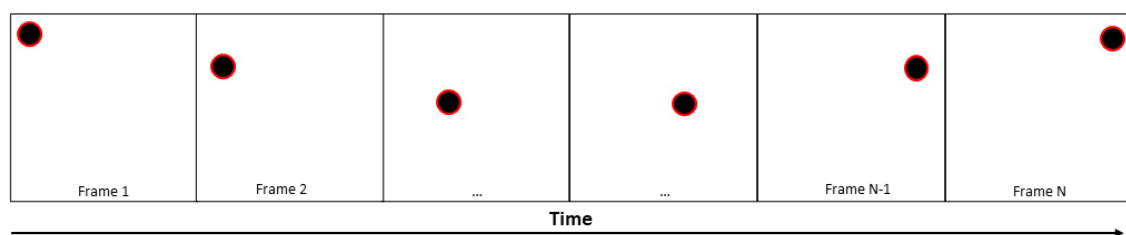


Figure 17: Illustration of blob detection.

With the coordinates of the pendulum in each image, it is possible to draw a chart representative of the pendulum's movement, as shown in Figure 18, where the red line corresponds to the coordinates of the X axis and the green line to the Y axis.

² <https://opencv.org/>

³ https://index.ros.org/p/cv_bridge/

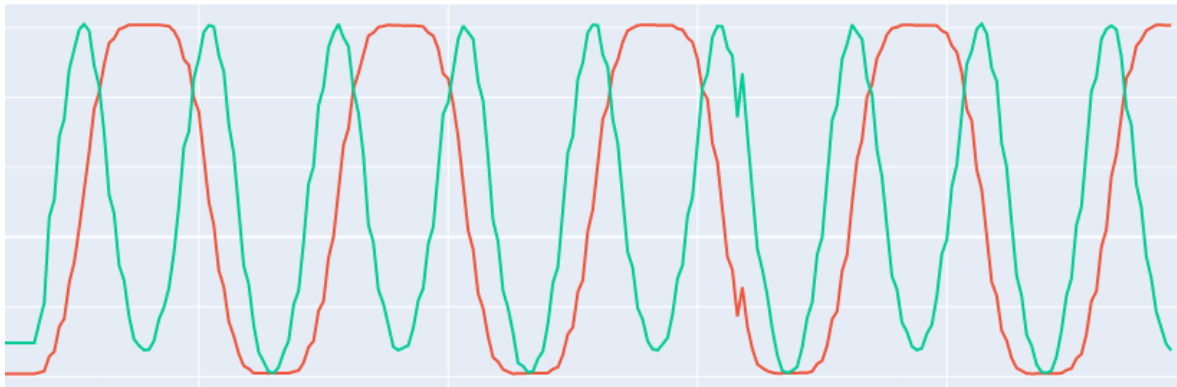


Figure 18: The pendulum's coordinates over the time.

As can be seen in section 7.6, to calculate the level of synchronization between the sensors, it is still necessary to identify the points of interest in the graph represented above. These points are identified on each curve of the graph by the (X_{Max}, Y_{Min}) and (X_{Min}, Y_{Min}) coordinates that correspond to the instants in which the pendulum movement direction changes and, consequently, the sound frequency/amplitude also changes, as seen in section 7.3.

7.5 AMPLITUDE GETTER

The Amplitude Getter is the tool that was developed to extract the audio data from the bag file, which contains the calibration video described in section 7.3, for post-processing. After identifying the messages corresponding to the audio topic, each message is deserialized through the libraries provided by ROS2. After this process, the content of each audio chunk present in each message is extracted, and, from this data, a sound wave is created where it is possible to observe the different amplitudes/frequencies acquired by the microphone, as shown in Figure 19.

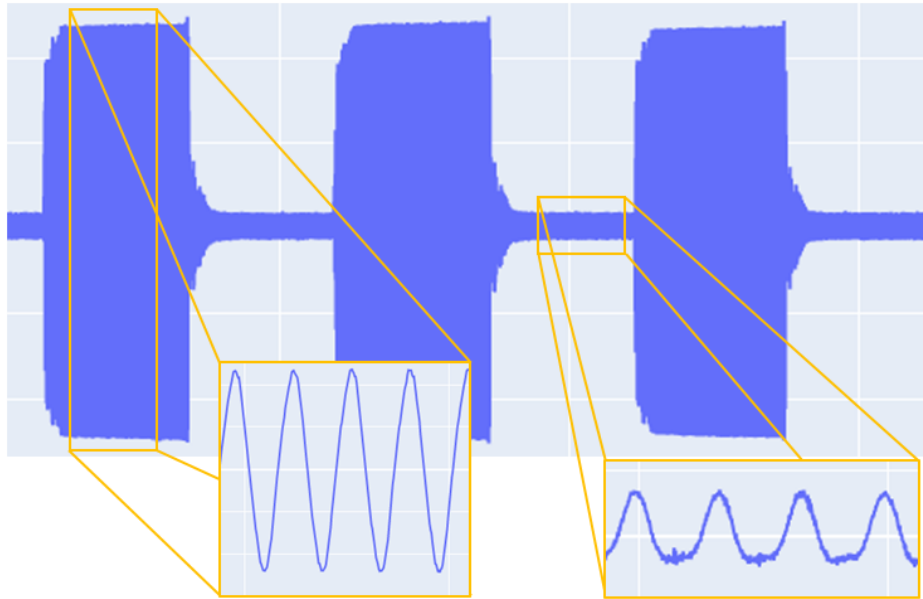


Figure 19: Audio amplitude over the time, with two different amplitudes/frequencies magnified.

With the pendulum's coordinates and the audio's frequency/amplitude in one chart, the level of (de)synchronization of the data present in the bag file can be seen. As mentioned in section 7.3, the change in the direction of motion of the pendulum occurs at the same instant as the change in audio frequency. Thus, in an ideal scenario, where the data is perfectly synchronized, these two shifts happen at exactly the same instant of time.

In order to automate the process of searching for the points at which the sound frequency changes, a low pass filter was applied to the audio amplitude graph (excerpt from Figure 19), generating a graph whose excerpt is shown in Figure 20.

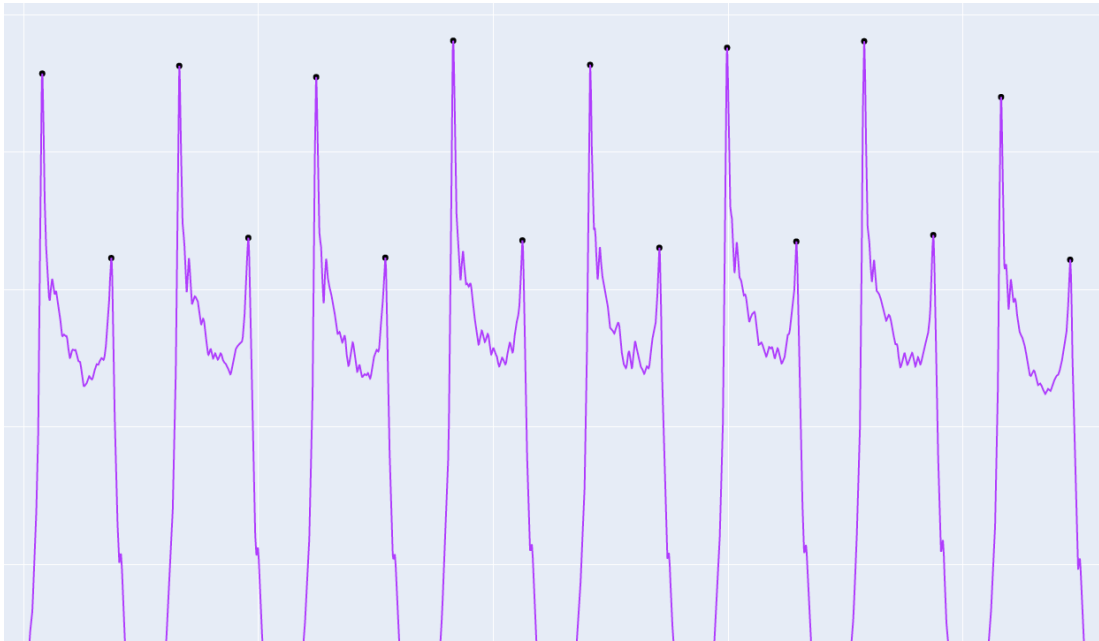


Figure 20: Excerpt from the audio graph, with the identification of amplitude/frequency change points.

The black points present in the figure indicate the instant in which the sound frequency changes and were identified through the *find_peaks* method. All this was possible taking advantage of the *scipy*⁴ library, specifically the signal module. After identifying these points, as shown in the next section, it is only necessary to subtract the timestamps from these with the points identified in the camera graph, in order to obtain the level of (de)synchronization between the sensors.

7.6 RESULTS

In this section, the results obtained with the synchronization strategy described earlier are presented.

As explained before, the cameras are synchronized with each other through the *PTP* protocol. Therefore, the process of calculating the lag time between audio and video is only applied to the camera designated as "Master". Since the other cameras are synchronized with it through *PTP*, if the "Master" is synchronized with the audio, then the other cameras are also synchronized. Figure 21 shows a chart with the variation of the lag/delay over time between two cameras (i.e., Master's timestamp minus Slave's timestamp), on a microsecond scale. In a two and a half hour-long test, the average lag was in the order of $-9 \mu\text{s}$, which means that the Slave camera was on average $9 \mu\text{s}$ behind of the Master camera. Still, the

⁴ <https://scipy.org/>

Slave camera for the same test had a maximum ahead lag of $15 \mu\text{s}$ (maximum in the chart) and a maximum behind lag of $30 \mu\text{s}$ (minimum in the chart).

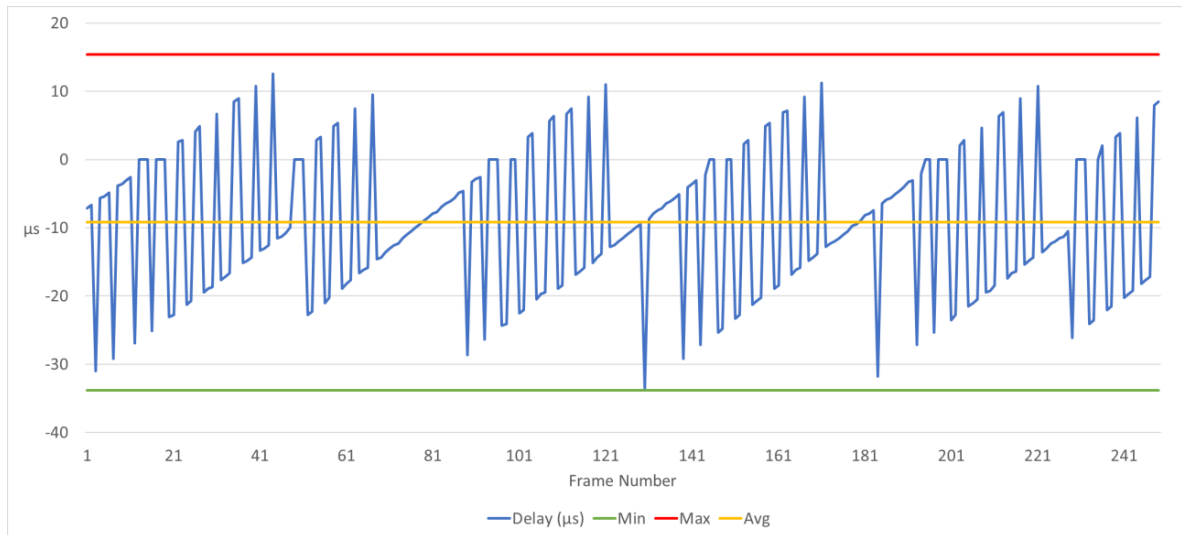


Figure 21: Lag over time between two cameras synchronized using *PTP*.

Regarding the Blob Detector (section 7.4), responsible for processing the data acquired by the camera, Figure 22 shows an example of a frame, where it is possible to verify the correct identification of the pendulum. Together with this, a *CSV* file is also obtained with the timestamps of each frame and the pendulum's coordinates at that moment, as can be seen from the excerpt shown in Table 4.

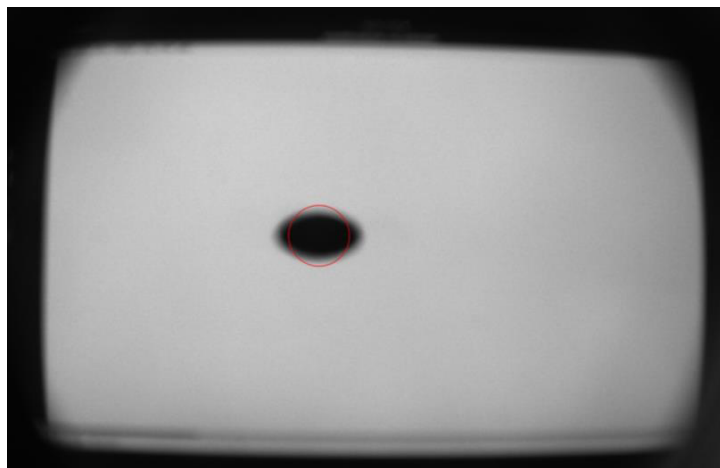


Figure 22: A result from the Blob Detector.

Table 4: Excerpt from a CSV file with the coordinates of pendulum.

Frame ID	Timestamp	Coordinate X	Coordinate Y
...
68	1619519659327756860	515,544	323,467
69	1619519659367819528	519,914	338,569
70	1619519659367819528	559,081	404,649
71	1619519659367819528	571,874	417,777
72	1619519659367819528	639,072	466,88
...

From the data in Table 4, it is possible to generate a chart representative of the movement of the pendulum, as seen in section 7.4. The chart shown in Figure 23 shows this, along with the result of the Amplitude Getter, already explained in section 7.5. The overlap of the camera data with that of the microphone is shown. The points of greatest interest are the points where the frequency/amplitude of the audio changes. In the case of the camera, it is important to focus on the coordinates (X_{Max}, Y_{Min}) and (X_{Min}, Y_{Min}), which correspond to the instant where the frequency/amplitude of the audio changed in the test video. With these points, the lag time, marked in the chart as "DSync", can be calculated by subtracting the timestamps.

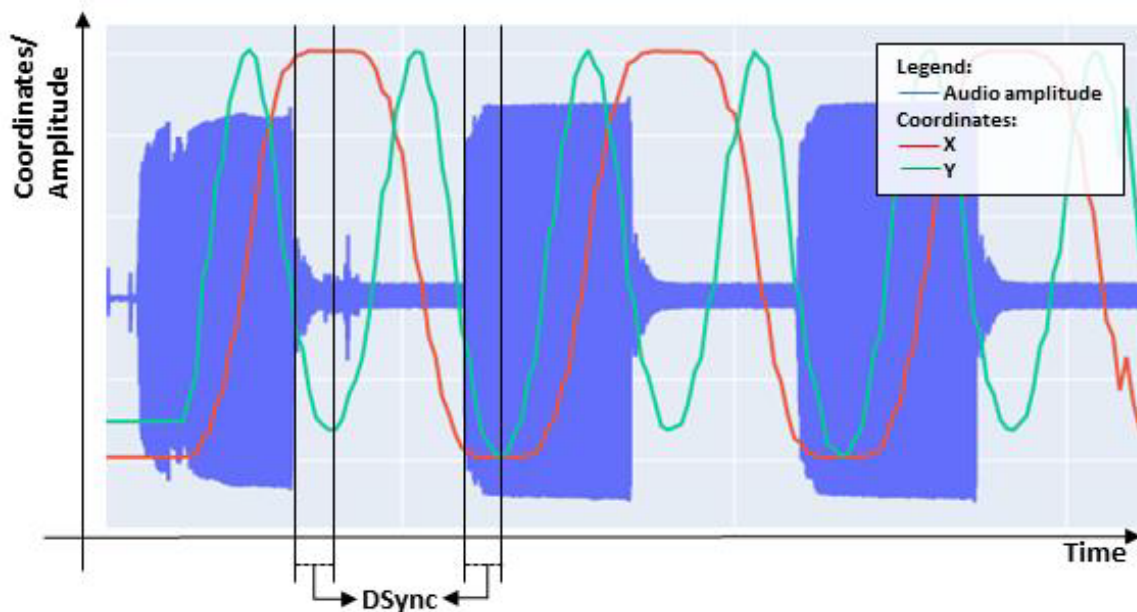


Figure 23: Synchronization data from one camera and the microphone.

After performing some synchronization tests, using the strategy presented in this dissertation, desynchronization values between audio and video in the order of milliseconds were obtained. In these tests, the minimum observed value was 20 ms, a maximum of 127 ms,

with an average of around 56 ms. Please note that all these values correspond to an audio delay in relation to the image. However, in one of the tests performed, a different behavior was observed, with the video delayed about 200 ms in relation to the captured audio. Since this scenario has only been observed once, it is considered an exception, but the system should be prepared if it happens again in the future.

Despite the delay verified between the sensors, this is not a problem, since during the post-processing it is possible to realign and synchronize the data, having as reference the obtained *DSync* value.

7.7 IMPROVEMENTS

Throughout the tests that were carried out, this strategy showed that some problems were needed to be solved.

The first had to do with the blob identification process. In laboratory environment, this question was not raised, because the conditions between the tests were identical and controlled. But when testing in a real environment, with different lighting conditions, it became difficult to automate the blob identification process, since the screen where the video was played was always subject to light reflections or other issues that could have influence.

The other point refers to the difficulty in ensuring that the computer playing the video was able to ensure that it was played with a fixed frequency. Some variations in the playback frequency of the image and audio of the video were found, which is a problem, since the video would have to be as accurate as possible, to be used as a reference for synchronization.

In order to solve this problem, the video option was discarded, finding a solution that would allow to maintain the developed strategy. An Astable Timer 555⁵ circuit was then used, such as the one that is diagramming in Figure 24.

⁵ <https://circuitdigest.com/calculators/555-timer-astable-circuit-calculator>

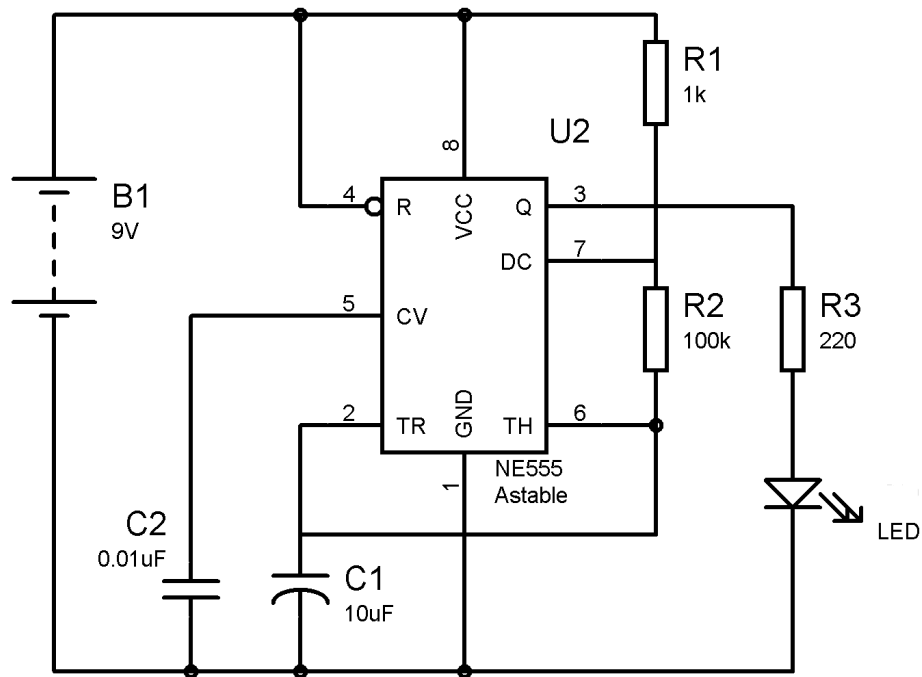


Figure 24: Astable Timer 55 circuit diagram (from Circuit Digest.⁵).

This circuit was connected to a speaker and an LED, and both components were activated at the same time and the frequency at which this occurred remained constant. Thus, the process presented in section 7.5 on the processing of the audio remained unchanged, and it was only necessary to change the image processing, thus solving the two problems identified.

For this, the first part concerning the extraction of the bag file frames continued to be done and was only replaced the blob identification part, which was generating problems. In order to understand the new process, it is important to analyze Figure 25 containing two frames taken from the bag file: the one on the left with the LED off and the one on the right with the LED on.

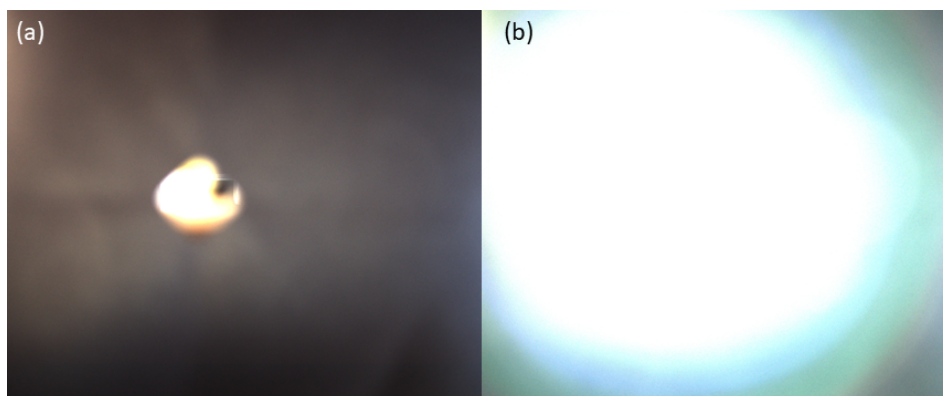


Figure 25: LED frames: (a) LED off and (b) LED on.

After analyzing the figure, it is easy to see the difference between one and the other. Thus, to process the frames was made the sum of the value of all pixels of the frame. As can be seen, when the LED is on, most pixels are clear, with the frame having a higher value (in terms of sum of pixels) than when it is turned off. Thus, by comparing the value of the sums of all frames consecutively, it is possible to identify the moments in which the LED turns on and match these instants to those identified by the Amplitude Getter (section 7.5).

To automate this process, a script was created that from the bag file, with the synchronization sequence, extracts all the necessary data and runs the entire process described in this strategy, returning the average value of *DSync*.

USER INTERFACE

The developed Data Acquisition Solution can obtain data from different sensors simultaneously. The challenge described in this chapter was to make this system more accessible to the common user, so a user-oriented graphical interface has been developed, where all the features offered by the system are available. The interface was developed in *Python*, by taking advantage of the PySimpleGUI¹ module, designed for the creation of graphical user interfaces. In each of the following sections, each of the features available are presented alongside a description of their implementation and one or more illustrative images.

As can be seen later in section 9, the final configuration of the Data Acquisition System consists of two computers and 6 sensors (5 cameras and 1 microphone array) distributed by the two machines (two cameras on one PC and one camera and the microphone array on another machine). So, it is important to take this into account when analyzing the present interface.

The graphical user interface presented here is heavily based on remote invocations, through *SSH*, for the two machines of the system, to start the whole system and interact with the sensors present in the two machines. The Figure 26, shows the interface state's before starting the system. As can be seen the only available functions are Init, Review, Reboot and Shutdown. After the system is initialized, it is also possible to use Record and Preview (Figure 30).

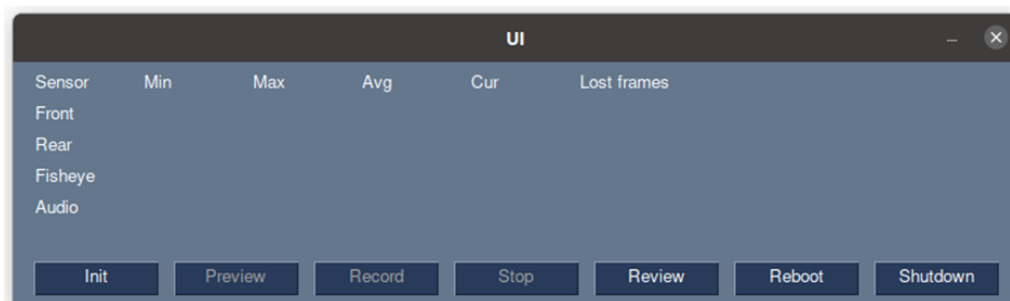


Figure 26: Initial screen.

¹ <https://pysimplegui.readthedocs.io/>

8.1 INIT

The Init button in Figure 26 initializes the Data Acquisition System. When pressed, the current state of the system is queried:

- If the system is turned off, a message with the information that the system is ready to initialize is displayed, asking users if they want to initialize the system (Figure 27).
- If the system is already in operation, users are informed about this and asked if they want to skip initialization (Figure 28). If the answer is yes, the interface is updated with the sensors' health monitoring data and the Preview and Record features are made available. Otherwise, the system initializes again.

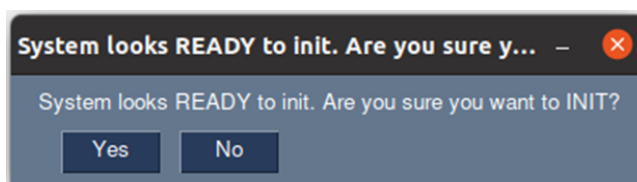


Figure 27: System ready to initialize confirmation.

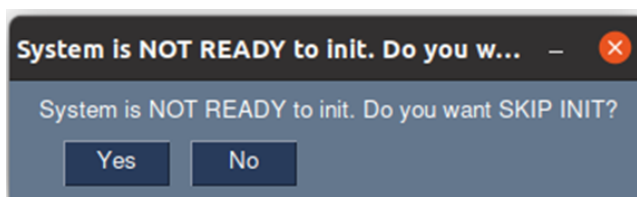


Figure 28: Skip initialize confirmation.

After this initial check, if the user wishes to proceed with init, a remote signal is sent to the two computers, starting the sensors contained in each of them, as well as the mechanisms for their health monitoring. After this signal is sent, a user message is displayed to the user, stating that there is the need to wait 2 minutes before the system is fully operational (Figure 29). During this wait time, the sensors are initialized, and the cameras are synchronized as described in the previous sections (7).

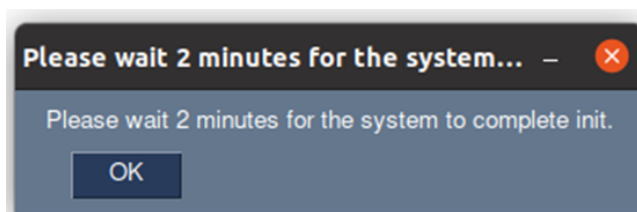
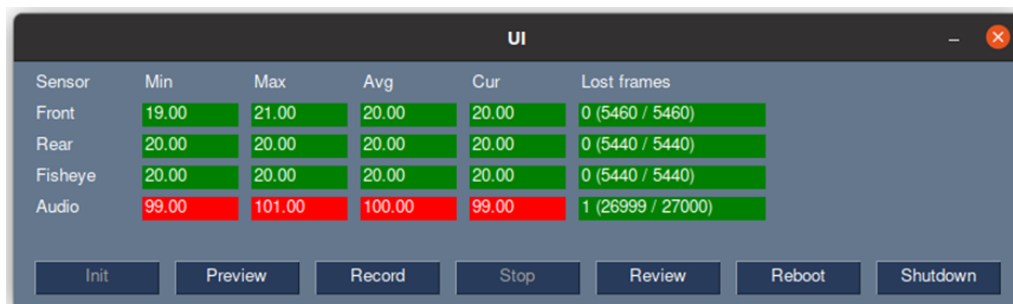


Figure 29: Wait 2 minutes until the system is ready.

After two minutes, the system should be operational, and the Preview and Record functions made available. As can be seen in Figure 30, the health monitoring data of each sensor are displayed, changing the colour from green to red in case there are differences between the number of received/recorded frames and the number of expected frames. For these, there are the minimum, the maximum and the average number of frames received so far as well as the calculated number of lost frames, considering the number of frames received so far and that which would be expected.



Sensor	Min	Max	Avg	Cur	Lost frames
Front	19.00	21.00	20.00	20.00	0 (5460 / 5460)
Rear	20.00	20.00	20.00	20.00	0 (5440 / 5440)
Fisheye	20.00	20.00	20.00	20.00	0 (5440 / 5440)
Audio	99.00	101.00	100.00	99.00	1 (26999 / 27000)

Buttons: Init, Preview, Record, Stop, Review, Reboot, Shutdown

Figure 30: Screen when the system is running.

8.2 PREVIEW

The Preview feature allows the visualization of the image acquired by the cameras in real-time. This feature is only available after the system is initialized and acquiring data from the sensors, as illustrated in Figure 30. When the Preview button is pressed, a command is sent to one of the machines to run RQT². By taking advantage of the *image_view* plugin, the user only needs to select the camera he wants to preview, and as soon as the camera is selected, he has access to the image that is being acquired (Figure 31).

² <http://wiki.ros.org/rqt>

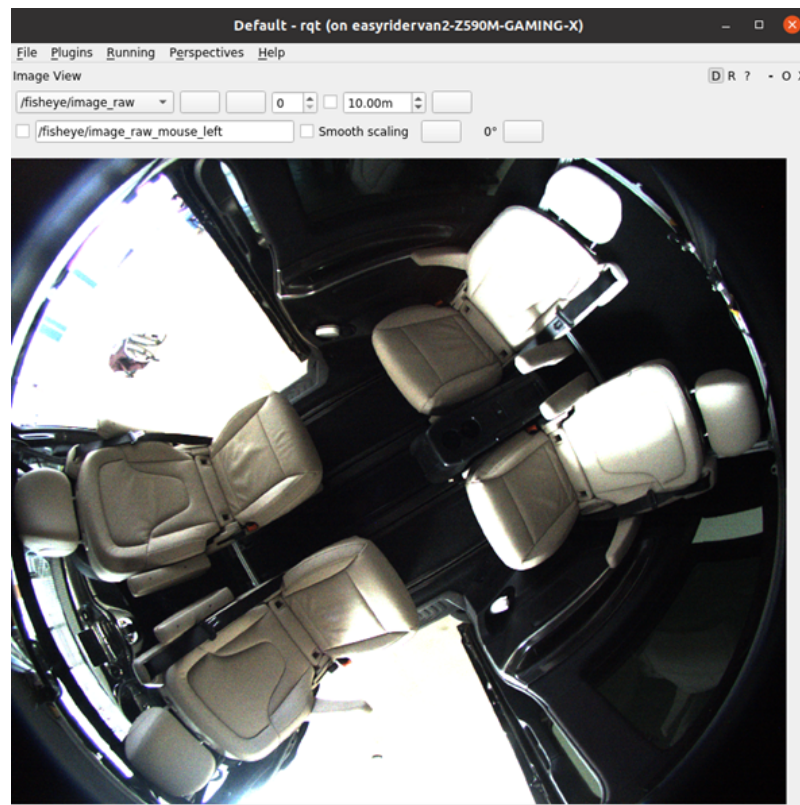


Figure 31: Image preview of fisheye camera.

8.3 RECORD

The Record feature records the data that is being acquired by the sensors so that it can be used later. For this the data of each sensor is recorded separately in different bag files, through the command `ros2 bag record`, made available by [ROS2](#).

Since this dissertation is framed within the scope of the *EasyRide Program*, as already mentioned, some features of this user interface were designed with data campaigns in mind that were carried out during the project. One of them is present in the record function, at the time of asking to enter an ID, as seen later.

This record feature is divided into 3 distinct scenarios, identified through the ID that is requested from the user before starting each recording (Figure 32). The first scenario is intended for the common user, and the remaining two were designed for the team responsible for managing the Data Acquisition System and the data campaigns.

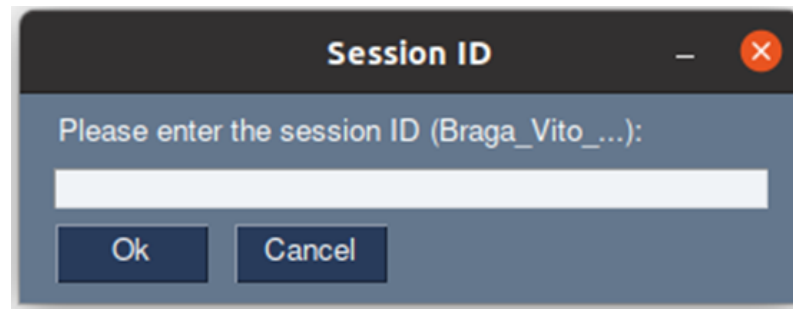


Figure 32: Insert ID for record.

In the case of a normal recording, the ID consists of 3 digits. This option was made to facilitate the process, since, as can be seen in the Figure 32, the complete ID of each scene was composed by the city and vehicle where it was being recorded. As these data campaigns took place in Braga, in a Mercedes Vito van, the first part of the ID was static (*Braga_Vito...*), being only necessary to enter the last 3 digits.

If the ID entered is not within the desired range (000-999), the user is warned (Figure 33).

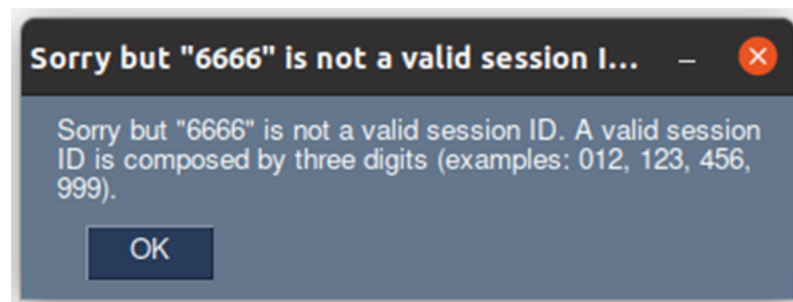


Figure 33: Invalid ID confirmation.

The two remaining recording scenarios concern calibration between sensors and test recordings. These scenarios are identified by another two types of IDs: the ID must start, in the case of calibration, with "cal." (Figure 34), and, in the case of test recordings, with "test." (Figure 35).

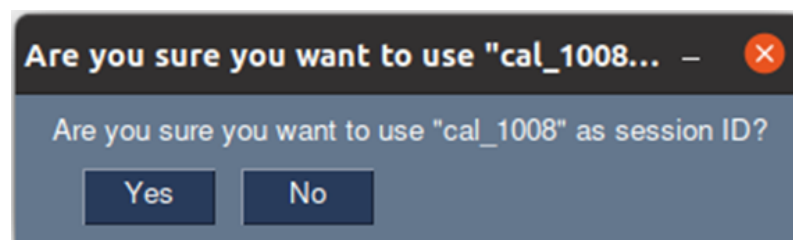


Figure 34: Calibration record ID confirmation.

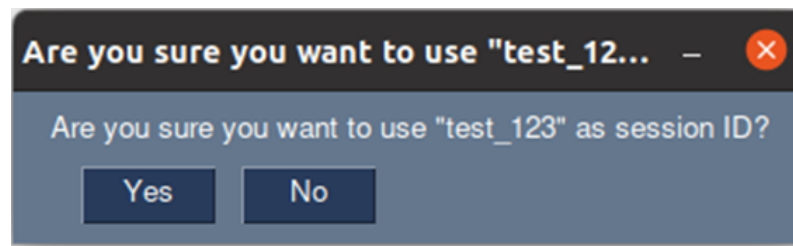


Figure 35: Test record ID confirmation.

As for the process behind each scenario, the calibration recording process corresponds to what has been presented in the section 7, regarding to the synchronization process. For this, only data from the master camera and microphone array are recorded (in separate bag files). When a calibration recording is finished, the script that calculates the desynchronization time between the audio and the video is run. The calculated value is shown to the user which can then save it for later (i.e., to synchronize the data recorded subsequently).

Finally, the test recording, as the name implies, was made available for testing and sanity checks. Test and normal recording are no different, except in their IDs. In the presence of a valid ID (000-999, *cal_XXX* or *test_XXX*), data recording begins until it is stopped, using the respective Stop button (Figure 36).



Figure 36: Record screen, with Stop button unlocked.

When a recording is stopped, the health monitoring data for each sensor during the recording is displayed (Figure 37), so that the user can decide whether to keep the recording or delete it (e.g., if the quality is not acceptable). If the user chooses to keep the recording, nothing is done; otherwise, the bag files are deleted.

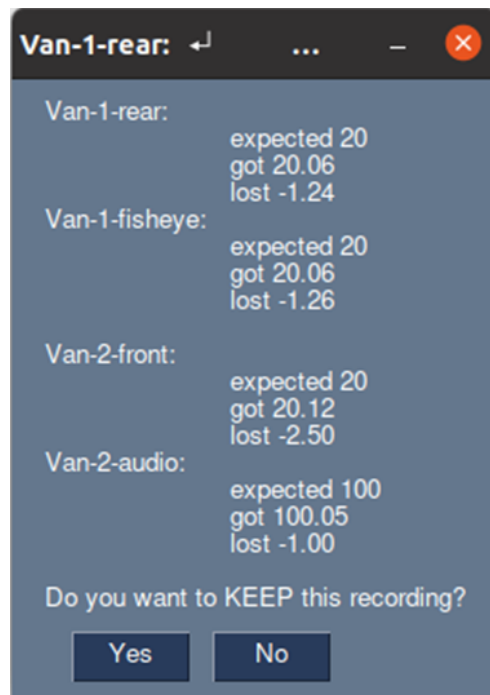


Figure 37: Record performance statistics and confirmation to keep or delete record.

In the Figure 37, it is possible to check the frame rate of three cameras (rear, fisheye and front) as well as the publish rate of the audio from microphone array.

8.4 REVIEW

The Review feature allows the user to review a recording by replaying the corresponding bag files. When the Review button is pressed, a text box is displayed (Figure 38) for the user to enter the ID of the recording that he wants to review.

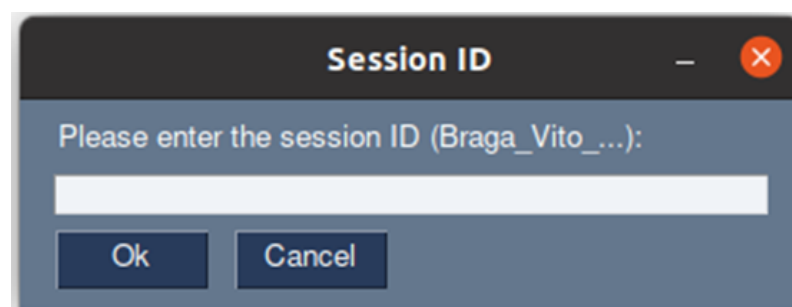


Figure 38: Insert the record ID for review.

The necessary commands are then sent to each machine to replay the bag files, through the command `ros2 bag play`, available in *ROS2*. To view the images from the recording, the process is identical to the one described in section 8.2 for Preview (Figure 39).

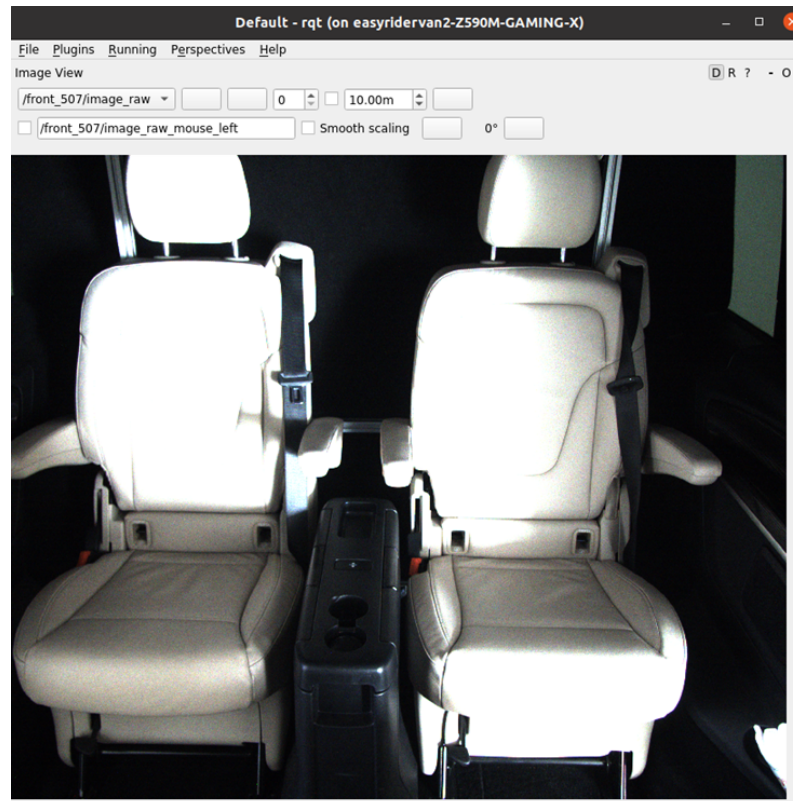


Figure 39: Front camera review.

8.5 REBOOT/SHUTDOWN

The processes behind the Reboot and Shutdown features are identical. If the respective button is pressed, a message is displayed to the user, asking for confirmation (Figure 40 and Figure 41). If there is confirmation, remote commands are sent to each machine to shutdown or restart, according to the button pressed. In the end, the graphical user interface is closed (it is running in one of the machines being shutdown/restarted).

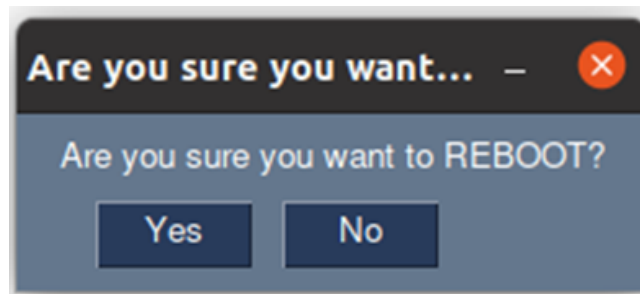


Figure 40: System reboot confirmation.

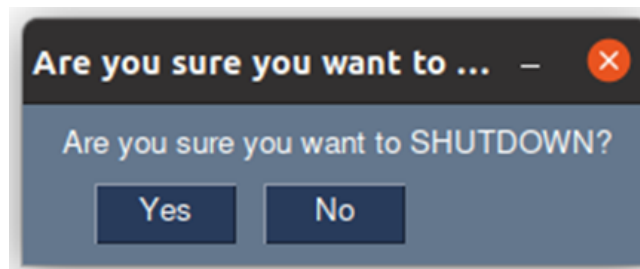


Figure 41: System shutdown confirmation.

RESULTS AND DISCUSSION

By taking advantage of the health monitoring system, described in section 6, some of the tests performed on the data acquisition solution to verify its ability to full the requirements are presented and discussed.

9.1 SYSTEM PERFORMANCE

Initially, performance tests were done with the aim of understanding what are the requirements in terms of hardware resources as well as improving the software (and the architecture) being developed.

These performance tests took place with three sensors:

- 2 x RGB cameras (*Allied Vision MAKO G-234c*): these cameras have a maximum resolution of 2,35MP (1936x1216), recording up to 41.5 fps. Power and data connection are supported through Power-over-Ethernet.



Figure 42: Illustrative image of the Allied Vision MAKO G-234c camera.¹

¹ <https://www.alliedvision.com/en/camera-selector/detail/mako/g-234/>

- 1 x microphone array (*miniDSP UMA-8*): this consists of an array of 7 michophones/channels, digital audio amplifier and noise reduction. Power and data connection are supported through USB.

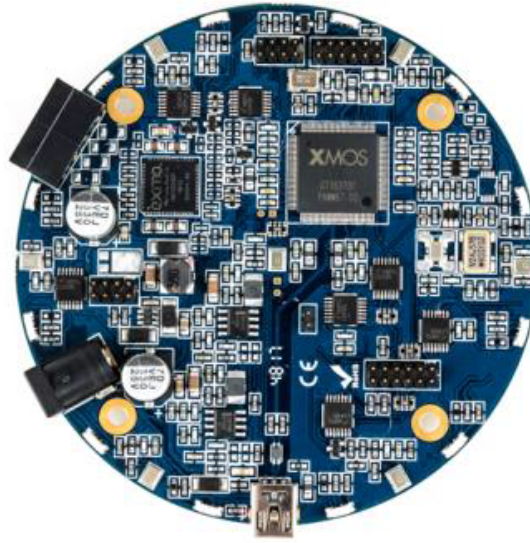


Figure 43: Illustrative image of the miniDSP UMA-8 microphone-array.²
² <https://www.minidsp.com/products/usb-audio-interface/uma-8-microphone-array>

Before the system performance tests, the sensors' configurations was analysed and tested in isolation to understand what exactly could be expected from them.

9.1.1 Sensors Configuration

From the analysis and tests performed on the sensors' synchronisation, a set of parameters were found to have a major impact on the sensors performance.

For the RGB camera used, the following set of parameters have an influence on the camera throughout, i.e., the number of frames per second:

- **Resolution:** parameter related to image size/quality. Several tests were performed, and this parameter varied between a minimum resolution of 640×480 and a maximum of 1936×1216 (maximum resolution supported by the camera). It directly influences the size of each frame received by *ROS2* and the maximum value supported for acquisition rate (i.e., higher resolutions reduce the maximum frame rate allowed by the cameras).
- **Stream Bytes per Second:** parameter that defines the maximum amount of data transmitted by the camera, in bytes per second. Varies between 45 and 115^3 MiB. This parameter also influences the maximum acquisition rate supported by the cameras (i.e., lower values of stream bytes per second, can reduce the maximum value of frames per second allowed).

³ In the final version this value was increased to 124, due to a firmware update.

- **Exposure/Exposure Auto:** corresponds to the camera's exposure time in microseconds. The value of "exposure_auto" has been set to "Off" so that multiple settings with different exposure values could be tested. This parameter also influences the maximum acquisition rate supported by the cameras (i.e., higher values of exposure, reduce the maximum value of frames per second allowed).
- **Acquisition Rate:** parameter that defines the frame rate of the camera, being affected by the settings of the remaining parameters presented above.

Regarding the microphone, it is necessary to have prior knowledge of some concepts about digital audio. The conversion of analog to digital audio, consists in sampling the original waveform according to a given sampling frequency, as illustrated in Figure 44. In this case, to perform this conversion process it is necessary to consider some parameters that determine the quality of the audio:

- **Sample Rate:** number of samples taken, per second. This feature determines the fidelity of the signal, i.e., the higher the frequency of the samples, typically, made the more percentage of signal is acquired.
- **Bit Depth:** determines how much information can be stored in each sample. The larger the number of bits, the more information each sample acquires, typically promoting audio quality.
- **Channels:** number of audio sources within the audio signal. Each channel contains a sample indicating the amplitude of the audio being produced by that source at a given moment in time.
- **Bit Rate:** bit transfer rate or flow that are converted or processed per second. It is commonly used to describe the audio stream. For uncompressed audio, it can be calculated.

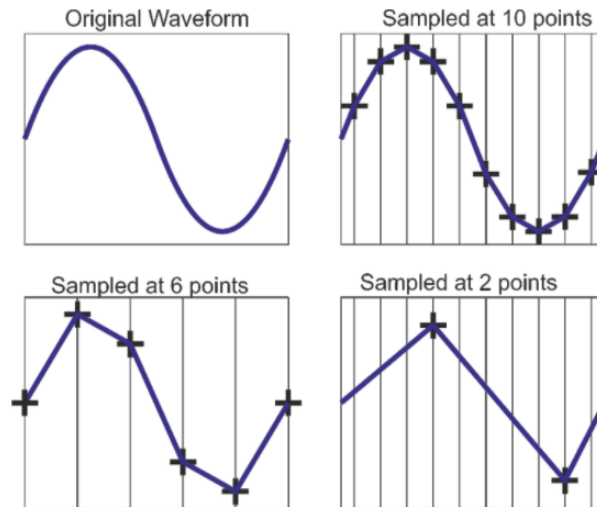


Figure 44: Illustration of different audio sampling frequencies [2].

In Table 5, the values used in the configuration of the sensors during the performance tests are presented.

Regarding the camera, the resolutions varied between 640x480 to 1936x1216 (maximum supported), the value of stream bytes per second alternated between 9000000 and 11500000, with the acquisition rate parameter varying between a minimum of 2 *FPS* and a maximum of 30 *FPS*. As for exposure, the *Auto* was set to *Off*, having always been used the value of 14988 for this parameter, which allowed to combine a good image quality, without compromising the acquisition rate values.

For the microphone, the sample rate varied between 8000 and a maximum of 48000 with the bit depth switching between 16 and 24. The number of channels also varied between 1 and 8, as can be confirmed in the table below.

Table 5: Sensor configuration values used during the performance tests.

Camera	
<i>Parameters</i>	<i>Values</i>
Resolution	640x480, 800x600, 1024x768, 1280x720, 1440x900, 1600x900, 1920x1080 and 1936x1216
Stream Bytes per Second	90000000 and 115000000
Acquisition Rate	2.0, 4.0, 8.0, 12.0, 14.0, 18.0, 22.0, 24.0, 26.0, 28.0 and 30.0
Exposure	14988 (Exposure Auto set to <i>Off</i>)
Microphone	
<i>Parameters</i>	<i>Values</i>
Sample Rate	8000, 16000, 44100 and 48000
Bit Depth	16 and 24
Channels	1, 2, 3, 4, 6 and 8

9.1.2 Test Scenarios and Setups

Besides the different sensor configurations (see 9.1.1, other factors that can compromise the system's performance have also been considered. Thus, for each one these, a test scenario and a test setup has been developed, all of which are described in this section. These scenarios/setups essentially have to do with the conditions of the machine on which the system is running (i.e., consumption or limitation of hardware resources).

Each test scenario goes through all the combinations of values of the different parameters of each sensor (see Table 5). In order to automate this process, a *bash* script was written to go through all those combinations. Moreover, tests were performed with and without storing/recording data, in order to better understand the impact of data acquisition from data storage.

All steps related to the script are listed below:

1. Drop cache.⁴
2. Set the sensor's configuration (depends on the sensor being tested).
3. Start Health Monitoring.
4. Start the sensor node (effectively starting data acquisition).
 - a) Test without recording.

⁴ The cache was cleaned up between each test so that the test of the previous configuration had no impact on the following.

- b) Test with recording.
- 5. Save file with results for analysis.

Below, the test setups that were developed to assess the system's performance under different scenarios are presented.

- ***Normal Conditions***

In this scenario, the sensors are used under "normal" conditions, that is, no changes to the hardware/software support system, or in other words, the tests running on a regular Ubuntu 20.04 operating system. This scenario is used as a reference for later drawing up conclusions about the results obtained in others scenarios. For this setup, it is only necessary to run the script for the camera and for the microphone, with no additional operations.

- ***Ubuntu 20.04 Lightweight Conditions***

Unlike the normal conditions' scenario, this scenario relies on a lightweight Ubuntu 20.04 operating system, limiting the number of drivers and services that are loaded (e.g., GUI). Under normal conditions, the Ubuntu 20.04 operating system activates services and drivers that are unnecessary for the application at hand, such as services related to the graphical user interface, network, and many more. For this setup, Ubuntu 20.04 is booted into recovery mode where only the most essential services and drivers are start. Because of this, the system has more resources available and its performance is expected to increase.

Moreover, the following setups are built on top of these two.

- ***Vimba SDK***

In this test setup, only the low-level image acquisition software, Vimba SDK⁵, is used (i.e., *ROS2* is not used). The goal is to remove *ROS2* from the acquisition process and so getting the highest possible performance from the camera. Besides, it enables the impact of *ROS2* on performance to be evaluated.

This setup was implemented on top of all the two setups presented so far.

- ***CPU Load***

This scenario assesses the influence of CPU load in the system. The performance of an application is often related to the resources made available by the operating

⁵ <https://www.alliedvision.com/en/products/software/>

system, being the CPU one of the most important. *CPU* (Central Processing Unit) can be idealized as the brain of a computer, responsible for processing all operations, as specified by instructions (i.e., by the software). In this way, it is essential to consider its analysis, since it directly influences the speed with which an application is running. Given this, to evaluate the influence of this resource on performance, *CPU* stress tests were performed, under different loads, using *stress-ng*⁶. More specifically, tests were performed for loads of 0, 20, 40, 60, and 80 percent.

This setup was implemented on top of all the three setups presented so far.

Here is a summary of the test setups/scenarios applied to each of the sensors that are reported in this dissertation:

- **Camera:**
 - Normal Conditions:
 - * Initial Tests (*ROS2*)
 - * Vimba SDK
 - Ubuntu 20.04 Lightweight Conditions:
 - * Initial Tests (*ROS2*)
 - * Vimba SDK (also with CPU load)
- **Microphone:**
 - Normal Conditions:
 - * Initial Tests (*ROS2*)
 - * CPU Load
 - Ubuntu 20.04 Lightweight Conditions:
 - * Initial Tests (*ROS2*)

To perform these tests, a *Workstation HP Z2 Tower G4*⁷ was used, with the following specifications:

- **Motherboard:** *HP Z2 Tower G4 Workstation*
- **CPU:** *Intel® Core™ i7-8700K 3.70GHz x 6*
- **RAM:** *16GB DDR4*
- **Storage:** *SSD 500GB M.2 NVMe*

⁶ <https://wiki.ubuntu.com/Kernel/Reference/stress-ng>

⁷ <https://support.hp.com/us-en/document/c06100744>

9.1.3 Performance Tests Results

In this section the results obtained from the tests performed are presented and analysed, for the different setups.

Each test for the RGB camera lasted 90 seconds, while for the microphone a duration of 60 seconds was used instead. For the entire execution of the test script, for the camera there is a total of 176 tests (8 resolutions x 2 stream bytes per second x 11 acquisition rates), due to the different combinations of parameters. For the microphone, for the reason, there is a total of 48 tests (4 sample rates x 2 bit depths x 6 number of channels).

For a clearer analysis of the results, the error value of the frames per second is presented. To obtain this value, the following formula is used:

$$FPS_{\text{error}} = FPS_{\text{expected}} - FPS_{\text{obtained}}$$

For the camera, FPS_{expected} corresponds to the acquisition rate. For the microphone, it is always 100.

In the following charts, FPS_{error} is on the Y axis, while the X axis corresponds to the different configuration for each of the sensors, namely:

- Camera: Resolution, Bytes per second, Acquisition Rate (e.g., "800x600,90000000,2.0" corresponds to a 800 by 600 resolution, 90000000 Bytes per second, and an Acquisition Rate of 2.0 Frames per Second).
- Microphone: Sample Rate, Bit Depth, Number of Channels (e.g., "16000,24,3" corresponds to a Sample Rate of 16000, a Bit Depth of 24 and 3 Channels).

Moreover, in the charts with the results from the microphone, their first half corresponds to tests without recording while the second corresponds to tests with recording. In the camera results' charts, the tests with and without recording are represented by different lines, as indicated by the labels in each chart.

Figure 45 presents results from the camera under normal conditions, when using ROS2 and only the Vimba SDK. The best results were obtained when using only the Vimba SDK (green line), where there are only very small deviations from what was expected. When using ROS2 (blue and orange lines), there was a higher FPS_{error} , especially with higher resolutions, in tests where the Stream Bytes per Second parameter was set to 90000000 bytes. This behaviour is understandable, since the camera may not hold with the amount of data being requested. Moreover, as expected, the FPS_{error} when recording (orange line) is higher than when not recording (blue line). All these results were obtained with a CPU load of 0%. Results using higher loads are not report because they were worse than what is already shown here.

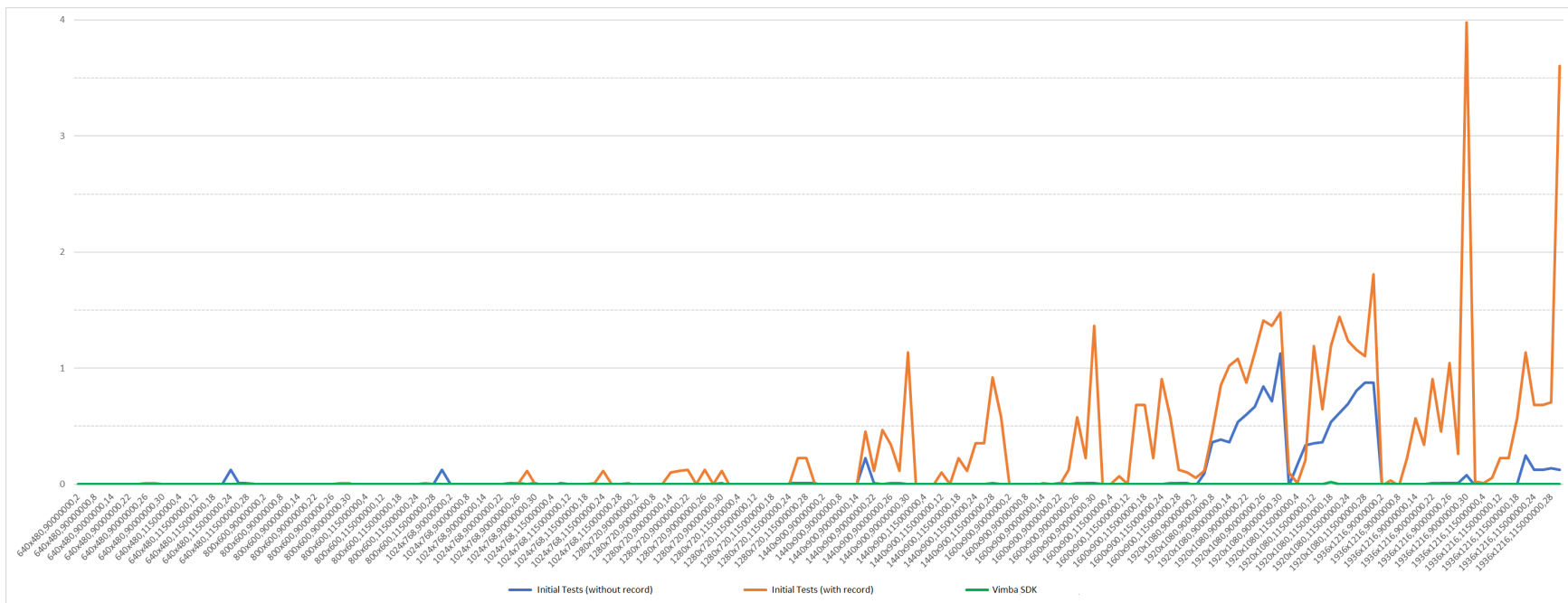


Figure 45: Camera's FPS_{error} under normal conditions, when using ROS2 and only the Vimba SDK.

Figure 46 presents results from the camera under lightweight conditions. These results are significantly better than under normal conditions. When using *ROS2*, there is no significant difference between recording or not, there are only very small deviations from what was expected. The highest errors occur with a resolution of 1920x1080, but still, always less than 1 frame per second, considerably better than under normal conditions. Results when using *ROS2* were only obtained with a CPU load of 0% because using higher loads the results were worse than what is already shown here. Conversely, when using the Vimba SDK even with a CPU load of 80% level (purple line), there are no significant errors.

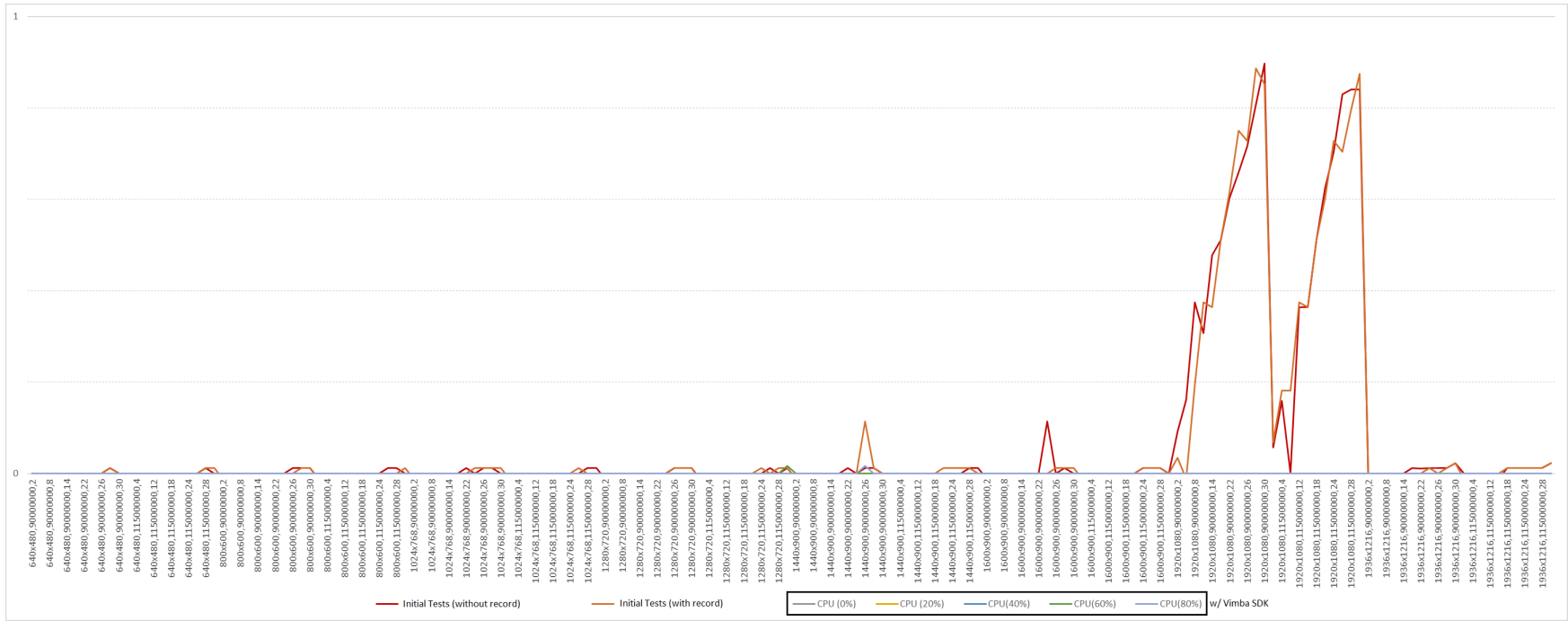


Figure 46: Camera's FPS_{error} under lightweight conditions, when using ROS2 and only the Vimba SDK with CPU load.

Figure 47 and Figure 48 present results from the microphone under normal and lightweight conditions, respectively. Under normal condition, there only a few significant errors; CPU load and recording seems to have negligible impact on the results. Under lightweight conditions, the results are even better, with CPS_{error} equal to zero on many occasions (i.e., the result obtained equals the expected).

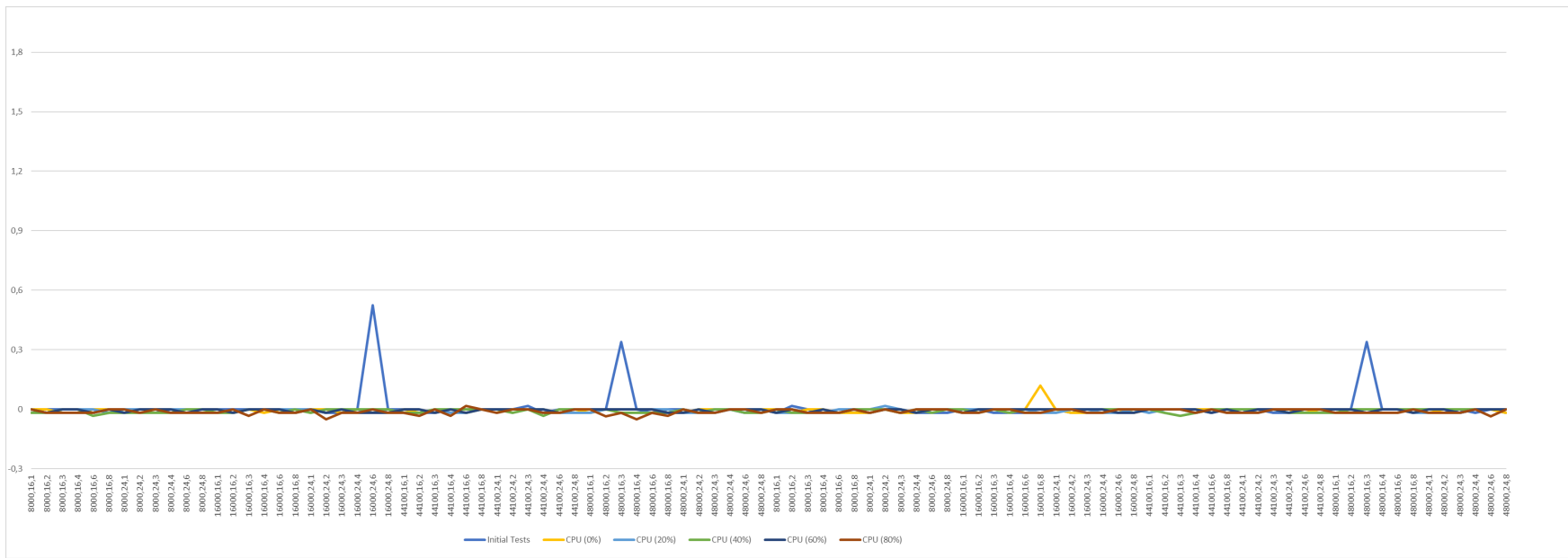


Figure 47: Microphone's CPS_{error} under normal conditions.

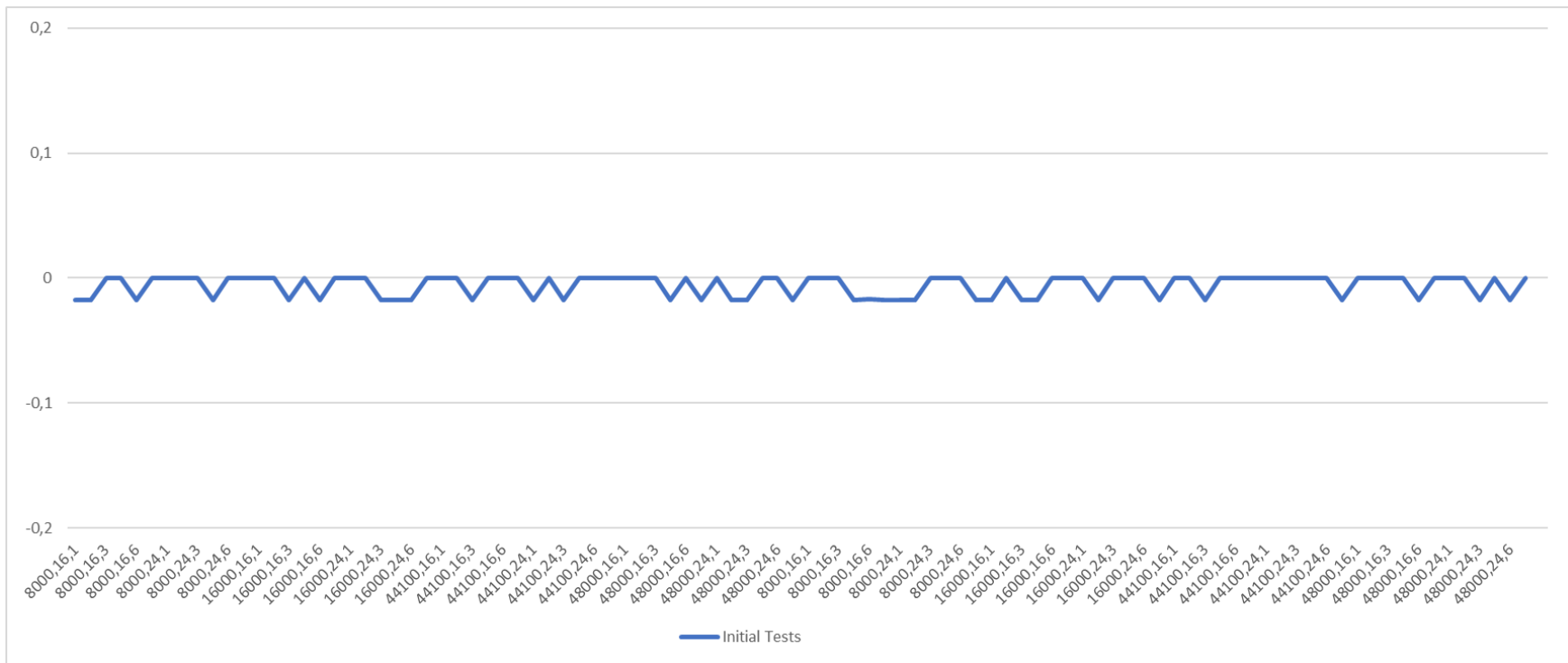


Figure 48: Microphone's CPS_{error} under lightweight conditions.

After testing the sensors separately, two tests of approximately one hour each, with recording, were performed. Using a "minimal setup", these tests served to understand the behavior of the system in its final version. Three sensors were used (i.e., 2 cameras and 1 microphone). The tests were performed, one under normal conditions and the other under lightweight conductions. The parameters used for the sensors were as follows:

- Cameras:
 - Resolution: 1936x1216 (2.35mpx)
 - Stream Bytes per Second: 11500000 bytes
 - Acquisition Rate: 20.0 frames per second
- Microphone:
 - Sample Rate: 44100 Hz
 - Bit Depth:16 bits
 - Number of Channels: 8

Figure 49 and Figure 50 present results from the minimal setup under normal and lightweight conditions, respectively. Regarding the microphone, the error is negligible. Regarding the cameras, on the other hand, the error starts high but decreases over time. Under normal conditions it reaches 0.4 but under lightweight conditions it get very close to zero.

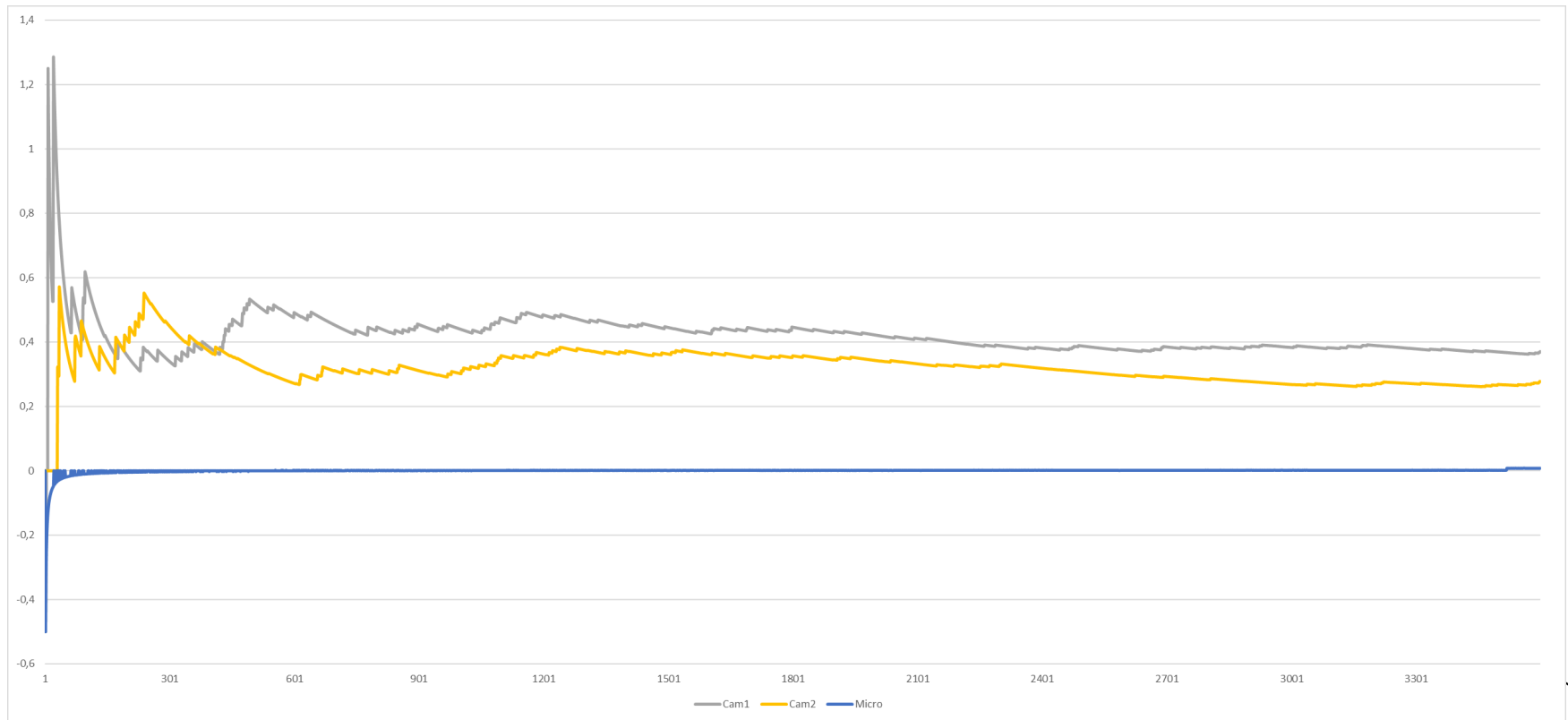


Figure 49: Results from the minimal setup under normal conditions, with recording.

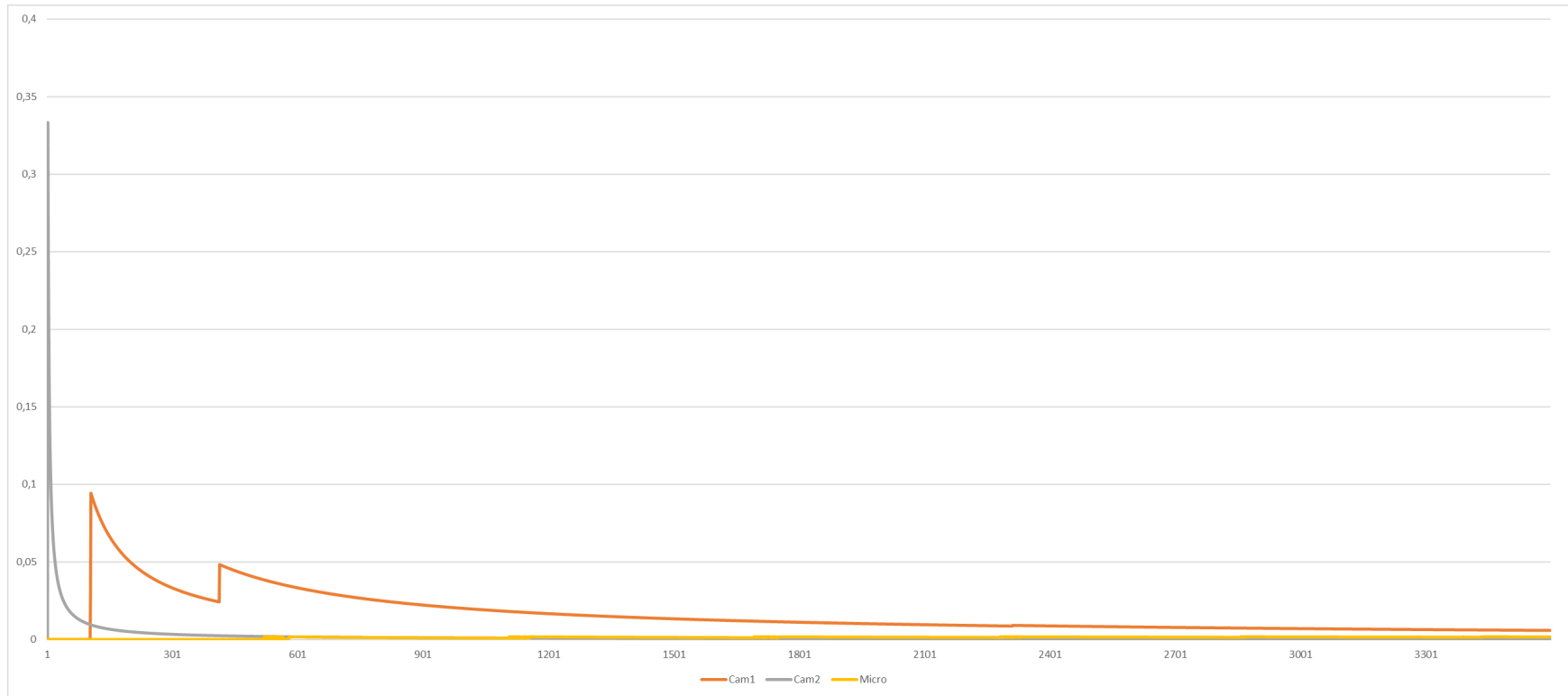


Figure 50: Results from the minimal setup under lightweight conditions, with recording.

From these results, it can be concluded that the system should run under lightweight conditions. Nevertheless, that is still not ideal, especially the performance of the cameras. The changes that have been implemented, after analyzing these results, are presented below, with the aim of improving the performance of the cameras.

9.1.4 Performance Improvements

After the analysis of the results presented in the previous section, and taking into account the issues with the performance of the cameras, the manufacturer was contacted and the documentation further analysed, from which small but significant changes were made, resulting in even better performance. These changes have to do with three specific parameters of the camera:

- **GVSPMaxRequests:** the maximum number of resend requests that the host attempts before marking a packet dropped.
- **GVSPPacketSize:** network packet size in bytes.
- **Stream Bytes per Second:** moderates the data rate of the camera.

The first parameter (*GVSPMaxRequests*) defines how many times packets can be resent before marking them lost and so it can be used to improve immunity against some network errors. The default value of this parameter is 3, but in the original *avt.vimba.camera* package, this parameter was set to 0 and no resend attempts were made. Having seen this, this value has been increased to 512 (maximum allowed by the camera), reducing the number of packets and frames lost, and thus, reducing FPS_{error} ⁸.

As for the second parameter (*GVSPPacketSize*), it defines the size of each packet that is transferred between the camera and the computer. This value is also related to the *MTU* settings of the computer's network card. At the time of the first tests this value was set to 1500 bytes. According to the documentation provided by Allied Vision⁹, the cameras being used support jumbo frames (in Ethernet terminology, jumbo frames are the name given to frames/packets whose size is greater than 1500 bytes¹⁰). Seen this, the packet size was increased to 9000 bytes (at the same time, the network card's *MTU* has to be configured with the same or higher value). As a result, network traffic decreased (larger packets result in fewer packets/headers being transmitted), thereby reducing the probability of a network error, packets or frames lost, and thus, reducing the probability of increased FPS_{error} .

⁸ This was confirmed during the Data Campaigns, through the graphical interface, therefore there are no graphics like those presented with the initial tests.

⁹ https://www.alliedvision.com/fileadmin/content/documents/products/cameras/various/installation-manual/GigE_Installation_Manual.pdf - How to minimize/eliminate dropped packets? (Page 50)

¹⁰ https://cdn.alliedvision.com/fileadmin/content/documents/products/cameras/various/features/Camera_and_Driver_Attributes.pdf - Packet Size (Page 54)

Finally, the maximum value supported for the Stream Bytes per Second parameter was changed. This parameter moderates the camera data rate and initially the maximum allowed was 115 MiB. After an update made to the firmware of the cameras, it was possible to increase this value to 124 MiB and thus increase the volume of data that can be transmitted, per second, from the cameras to the computers.

These three changes resulted in a performance improvement and were implemented before the Data Campaigns, the results of which are presented in section 9.2.3.

9.2 DATA CAMPAIGNS

As already mentioned, since the project of this dissertation is part of the *EasyRide Program*, the result of a partnership between Bosch and the University of Minho, data campaigns were carried out in this area. Thus, it was possible to test the system in an environment close to reality and perceive its behaviour, thus analyzing all the options that were taken throughout their development.

The system architecture used during the data campaigns is presented below, focusing first on the software organization and later on the hardware architecture.

9.2.1 *Software Architecture*

The final version of the system software architecture integrates all the work documented in this dissertation. As can be identified in the Figure 51, *ROS2* is the center of the entire operation, making the connection between the sensors, database or monitoring system, for example.

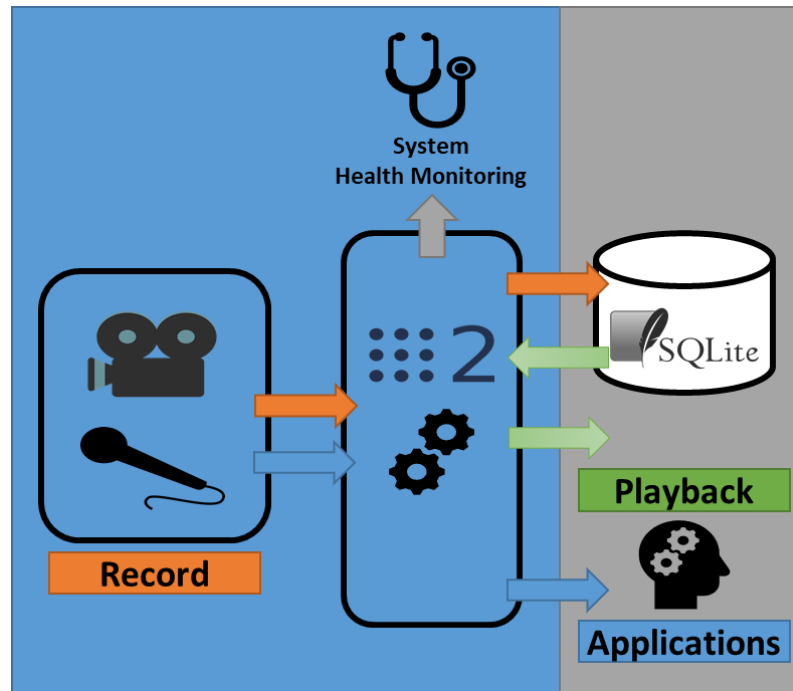


Figure 51: Illustration of software operation.

To clarify the information presented in the figure, the main concepts are explained:

- **Recording:** acquisition and storage of data from sensors.
- **Playback:** Playback and visualization of data stored during recording.
- **Monitoring:** analysis of the status of sensors and machines.
- **Applications:** being integrated in the *EasyRide* project, this system aims to provide data as each project needs, such as for the purpose of identifying violence, objects, among others.

9.2.2 Hardware Architecture

This data acquisition system was installed in a Mercedes Vito van, with the passenger part customized in a shuttle configuration (see Figure 31). The distribution of the sensors is shown in Figure 52.

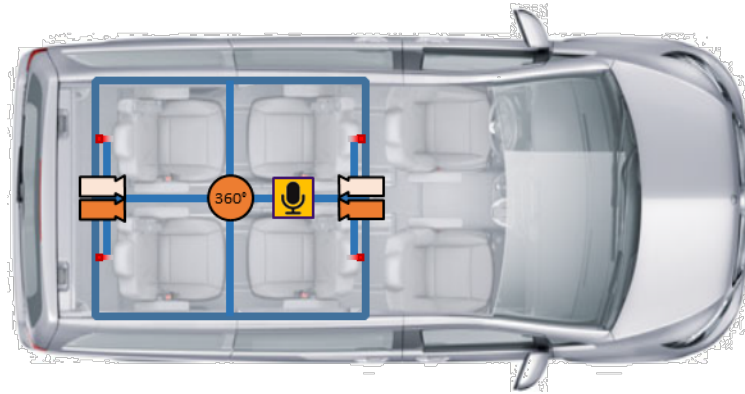


Figure 52: Layout of sensors in the van.

The system consists of six sensors:

- 3x RGB Camera Mako G-507C (a central with "fisheye" lens);
- 2x RGB Camera Mako G-234C;
- 1x UMA-8 Microphone Array.

The rest of the hardware is observable in Figure 53 and is installed in the trunk of the van. The system consists of two *PC Gigabyte Z950M Gaming X*¹¹ computers (with 10 Gigabit connection) to support the entire system, with the following features:

- **Motherboard:** *Gigabyte Z590M Gaming X*;
- **CPU:** *Intel® Core™ i9-11900KF Processor*;
- **RAM:** *32GB 3200MHz DDR4 CL16 DIMM (Kit of 2) HyperX FURY Black*;
- **ROM:** *Samsung M.2 PCI-E NVMe Gen4 980 PRO 2Tb*;
- **Network Card:** *Asus (1-Port) 10GBase-T PCIe Network Adapter*;
- **GPU:** *Geforce RTX 3080 Gaming 10GB*.

The sensors are distributed by the two machines, one responsible for the Graphical Interface (Section 8), one camera and microphone and the other with the other cameras. Attached to this, there is the switch (not visible in the Figure 53) that allows to follow this distributed architecture, responsible for connecting machines and sensors. There is also a *QNAP TES-3085U* with about 40TB of capacity to store data and a power system, to ensure power to the system while driving the van.

¹¹ <https://www.gigabyte.com/Motherboard/Z590M-GAMING-X-rev-10#kf>

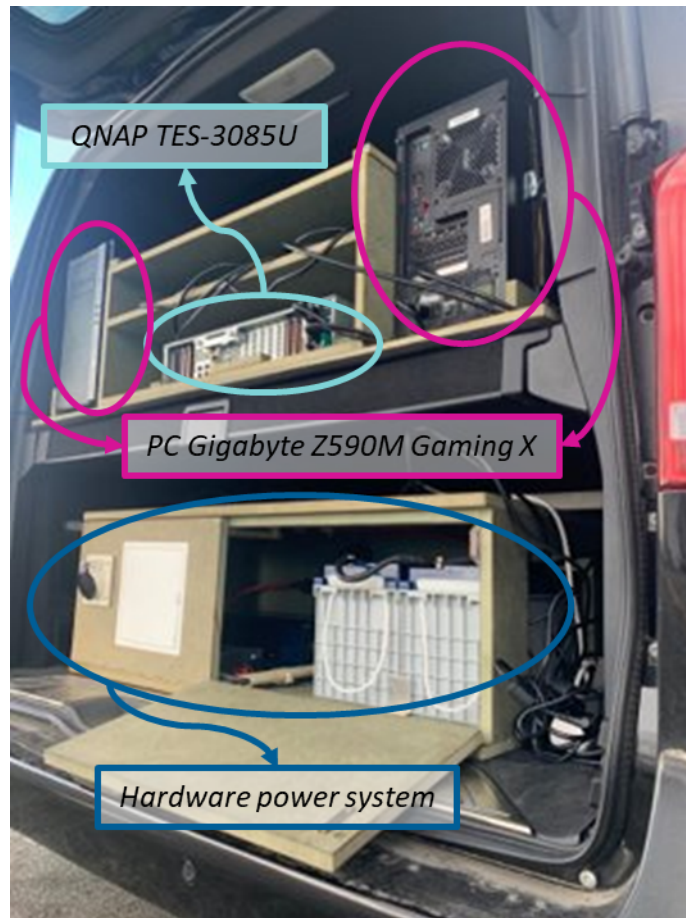


Figure 53: Van trunk with hardware components.

9.2.3 Final Results

In these data campaigns, more than 600 scenes were performed during almost 2 months, with close to 300 actors, and a volume of data in the order of 14TB was generated. The system performed very well, with no major performance breaks to be recorded, having fulfilled the project objective.

The dataset created during the data campaigns was important so that other projects within the *EasyRide Program* could test and train their algorithms. At the end of the Program, the objective was for the data acquired by the system presented in this dissertation to be made available in real time for the remaining projects.

In the next chapter, a complete analysis of all the work developed is made, also addressing the points that can be improved in the solution presented here.

CONCLUSION

The study of the state of the art permitted a better understanding of all the concepts important for the following stages of this work. With this, it was possible to create a solid foundation capable of sustaining the work, from the development of the architecture to the final solution. It was possible to understand, in more detail, some of the available tools and the main concepts that should underlie the development of this work.

In this way, the goal was on using *ROS2* to implement the audio and image packages, defining a set of guidelines for migrating packages from the first to the second version of *ROS*. After this, the focus was on developing the mechanisms of health monitoring, simultaneously with the mechanisms of synchronization between sound and image sensors. An graphical interface was also developed to facilitate the interaction between the system and the user. An analysis to the performance of the developed solution was also made, to understand if the objectives are being met or if there is any improvement that can be made in the built data acquisition system.

Thus, reaching the end of the project is made a positive evaluation of the work developed. The proposed objectives were met, developing a system capable of responding to requests for data acquisition from various sensors, such as cameras and microphones. The way in which some setbacks were resolved, such as performance issues or changes to the synchronization strategy, proved to be important points for the good performance of the system during the Data Campaigns, which constituted the final test of all the work presented in this dissertation.

Looking at the points that can be improved, it essentially goes through the issue of synchronization between sensors. The synchronization of the cameras was resolved in a very positive way, ensuring that the images of the various cameras were synchronized in real-time, without requiring any post-processing work. However, when this issue involves audio data, the strategy found requires post-processing data to synchronize and realign the different types of data. Thus, it would be interesting to get a solution identical to that used in cameras, where the different sensors would be synchronized in real-time, without forcing any data processing after their acquisition.

However, this less positive point does not hinder the functioning of the system, thus fulfilling the purpose for which it was developed throughout this dissertation project.

BIBLIOGRAPHY

- [1] K. Peffers, T. Tuunanen, M. Rothenberger, and S. Chatterjee, "A design science research methodology for information systems research," *Journal of Management Information Systems*, vol. 24, pp. 45–77, 01 2007.
- [2] "About oscilloscope sample rate — labtronix test and development solutions." <https://labtronix.co.uk/drupal/content/about-oscilloscope-sample-rate>. (Accessed on 2021).
- [3] "Ros.org — about ros." <https://www.ros.org/about-ros/>. (Accessed on 2021).
- [4] "Autoware.auto." <https://www.autoware.auto/>. (Accessed on 2021).
- [5] "Apex.ai – autonomous driving software." <https://www.apex.ai/>. (Accessed on 2021).
- [6] D. Linthicum, "Next generation application integration: From simple information to web services," 01 2004.
- [7] "Apache kafka." <https://kafka.apache.org/intro>. (Accessed on 2021).
- [8] "Kafka and message queues – microsoft docs." <https://docs.microsoft.com/en-us/learn/modules/cmu-message-queues-streams/1-message-queues>. (Accessed: 2021).
- [9] A. B. Bondi, "Characteristics of scalability and their impact on performance," *In Proceedings of the Second International Workshop on Software and Performance - WOSP '00*, 2000.
- [10] H. El-Rewini and M. Abd-El-Barr, "Advanced computer architecture and parallel processing," *John Wiley Sons*, vol. 42, 2005.
- [11] M. Van Steen and A. Tanenbaum, "Distributed systems principles and paradigms," *Network*, vol. 2, 2002.
- [12] M. Ben-Ari, "Principles of concurrent and distributed programming," *New York: Prentice Hall*, 1990.
- [13] K. Shin and P. Ramanathan, "Real-time computing: A new discipline of computer science and engineering," *Proceedings of the IEEE*, vol. 82, 1994.
- [14] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Ng, "Ros: an open-source robot operating system," vol. 3, 01 2009.

- [15] "UMA-8 USB microphone array." <https://www.minidsp.com/products/usb-audio-interface/uma-8-microphone-array>. (Accessed on 2021).
- [16] "Allied vision - MAKO G cameras." <https://www.alliedvision.com/en/products/camera-series/mako-g/>. (Accessed on 2021).

This work is supported by: European Structural and Investment Funds in the FEDER component, through the Operational Competitiveness and Internationalization Programme (COMPETE 2020) [Project nº 039334; Funding Reference: POCI-01-0247-FEDER-039334].