Universitat de les Illes Balears

escola politècnica superior

Final Degree Report

GRAU D'ENGINYERIA ELECTRÒNICA INDUSTRIAL I AUTOMÀTICA

Design and Implementation of a Domotics System based on the Sentilo IoT Platform

ANDREU MARC SERVERA LAUDER

Tutor
Alberto Ballesteros Varela

Escola Politècnica Superior
Universitat de les Illes Balears
Palma, juliol de 2019

GRAU D'ENGINYERIA ELECTRÒNICA
INDUSTRIAL I AUTOMÀTICA

Design and Implementation of a Domotics System based on the Sentilo IoT Platform

ANDREU MARC SERVERA LAUDER

Tutor
Alberto Ballesteros Varela

Escola Politècnica Superior
Universitat de les Illes Balears
Palma, juliol de 2019

Dedicat als meus companys de classe i a la meva família.

Agraesc la dedicació, ajuda i interès per part del meu tutor durant la realització del projecte.

"*Keep it simple.*"

Contents

List of Figures

List of Tables

ABBREVIATIONS

IP Internet Protocol

IoT Internet of Things

UIB Universitat de les Illes Balears

FTT Flexible Time-Triggered

DES Distributed Embedded System

EC Elementary Cycle

TM Trigger Message

HaRTES Hard Real-Time Ethernet Switching

HTTP Hypertext Transfer Protocol

PC Personal Computer

GPIO General Purpose Input/Output

SG Sentilo Gateway

REST Representational State Transfer

API Application Programming Interface

HIL Hardware-in-the-Loop

IPC Inter-Process Communication

UDP User Datagram Protocol

IP Internet Protocol

JSON JavaScript Object Notation

FPS Frames Per Second

URL Uniform Resource Locator

SSH Secure SHell

IDE Integrated Development Environment

SUMMARY

Nowadays the automation of a building is implemented in the form of a Distributed Embedded System (DES), that is, as a set of nodes interconnected by means of a communication network that cooperate to achieve some objective. In this context the objective is to monitor and control the physical processes of the building. Moreover, these processes typically have specific operational requirements like, for instance, real-time requirements. That is why DESs are usually implemented as closed systems using specialized technologies.

In recent years, the concept of Internet of Things (IoT) has grown in popularity. Basically, it consists in creating a connection between a *thing* and the Internet, where this *thing* can be any physical device susceptible of being monitored and/or controlled. Obviously, there is a huge intersection among the concept of a DES and IoT. Actually, it would be interesting to merge them in order to get the main advantages of each one.

In this project we will describe the development of a distributed embedded system that executes various tasks demanding specific real-time communication requirements. More specifically, the system will execute two tasks; a domotics task and a control task. Also, the integration of an IoT platform into this system will be implemented in order to create new possibilities and applications.

The Hard Real-Time Ethernet Switching HaRTES is used as the communication network of the DES. This network makes it possible for the nodes of the DES to exchange real-time and non-real-time traffic. Additionally, Sentilo will be used as the IoT platform. This solution makes it possible for applications to interact with the sensors and actuators in an easy manner.

Finally, the creation of a user interface will be carried out to reflect the applications that the IoT platform offers. It will provide the user a way to monitor and control the system without having to be directly connected to it.

CHAPTER 1

INTRODUCTION

1.1 Background and Motivation

In the University of the Balearic Islands (*Universitat de les Illes Balears (UIB)*) there is an on-going research project called *Smart UIB* ([2]) with the purpose of: studying and developing new technologies to improve the operational efficiency of the UIB and its environment, developing and testing smart technologies to later transfer to the society and creating synergy points for the students and the researchers. One of its initiatives is the rehabilitation and renovation of *Ca Ses Llúcies*, a building that will serve as an example of a sustainable, smart and healthy space.

One of the projects that will take place in this building is the installation of a demonstrator consisting of a distributed embedded system running various applications with different real-time requirements. Some of these applications will automate certain internal processes of the building, that is, they will obtain information from the building itself or the environment and will compute a certain actuation that will modify the state of the building.

Ethernet is a good candidate for implementing the communications of such a system due to its maturity and huge bandwidth. Actually, Ethernet is nowadays being considered as a good substitute for the field busses used in the industrial domain. Its main downside in this domain is that it does not provide the required real-time response. In this regard, *Hart Real-Time Ethernet Switching (HaRTES)* is an Ethernet-based communication network that offers the main advantages of Ethernet, but also ensures a real-time behaviour in the communications. Moreover, HaRTES supports, in parallel, multiple communications, each one with different real-time requirements.

Consequently, the communication network used to interconnect the nodes of the system will be *HaRTES*. By doing this, we can demonstrate that it is possible to integrate the communications of various applications with different real-time requirements within the same network.

Another project within the *SmartUIB* initiative that is already operating is the monitoring of different environmental values of the University. This project has been imple-

mented using an IoT platform called *Sentilo*, which makes it possible for applications to interact with distributed sensors and actuators in an easy manner. Another example of a Sentilo deployment can be found in Barcelona. Specifically, the Barcelona city council uses this tool to monitor several aspects of the city, as a first step for converting Barcelona into a smart city.

Extending the HaRTES system by means of Sentilo is an interesting approach for several reasons. First, it makes it possible for entities outside the building to monitor its state, as well as to actuate on it. Moreover, this can be done using well-known generic Internet network protocols and technologies. Second, externalizing some decisions can make the management of the building more flexible. This is because these decisions can be taken considering, not only information coming from the building itself, but from other sources related, or not, to the building. Finally, with Sentilo it is possible to take decisions based on sensor readings from the past, and not only from the present as happens with HaRTES.

1.2 Project Objectives

The objective of this final degree project is to construct a distributed embedded system based on HaRTES implementing various tasks with different real-time requirements and, then, integrate the sensors and actuators in a Sentilo platform in order to create new applications that extend the capabilities of the system.

1.3 Tasks carried out

To make clear what was done and what was not done during this project, here we indicate the tasks that were carried out:

1. Study how a Sentilo platform works and become familiar with such an IoT technology.
2. Create an instance of this platform on which the project will be developed, integrating the corresponding sensors and actuators that the distributed system will use.
3. Study how a HaRTES network works to be able to interconnect the applications that are going to be implemented.
4. Implement the tasks:
    a) A domotics task consisting of the control over a light switch.
    b) A control task based on an inverted pendulum using a Hardware-in-the-Loop (HIL) simulation.
5. Integrate the functionalities of the IoT platform into the real-time network.
6. Create a user interface to manage and control the system, taking advantage of the functionalities provided by Sentilo.

1.4 Tasks not carried out

Here we indicate the tasks that were not carried out during the project but were used in its execution:

1. The implementation of the HaRTES network.
2. The simulation of the pendulum as well as the calculations of the control ([3]).
3. C libraries:
    a) *JSMN* [4]: used to facilitate the management of the messages received form Sentilo in terms of the format used.
    b) *libsentilo_mq*: Library provided by the tutor of this project used to enable communications between processes executed within the same operating system.

Other tools and technologies have been used and they are explained further on in section 4.6.

CHAPTER 2

PREVIOUS WORK

In this chapter we describe the concepts and technologies that are the basis for the system herein developed. We first describe FTT, the communication paradigm used by HaRTES. Then we explain the particular characteristics of the HaRTES network. Finally we describe in more detail the Sentilo platorm.

2.1 Flexible Time-Triggered (FTT) basics

FTT is a communication paradigm proposed at the university of Aveiro in Portugal that makes it possible for DES to exchange real-time data. Additionally, it enables these message exchanges to be *flexible*, in this case flexible meaning that the parameters of the real-time messages (periods, dead-lines, etc.) can be changed at run-time if the changes result in a schedulable message set.

The general FTT architecture applied to a DES is shown in figure 2.1. As observed, several nodes are interconnected through a network, one of them being the master and the rest being slaves. The master is responsible for controlling the communications between the slaves, which is done by broadcasting a special message called Trigger Message (TM). This message tells each slave *when* and *what* are allowed to transmit.

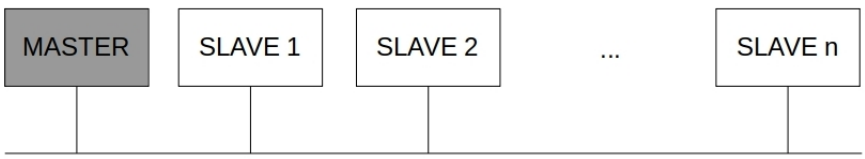


Figure 2.1: FTT's architecture.

2.1.1 Elementary Cycle (EC)

An *Elementary Cycle (EC)* is a slot of fixed duration responsible for structuring the message and it is used by the master node to organize the communication. Each time the TM is broadcast, it indicates the nodes that a new EC is starting, thus it synchronizes them. The EC also ends when the next TM is transmitted. These cycles are numbered incrementally as shown in figure 2.2, and are divided into two windows, the *synchronous* and the *asynchronous* windows. In the first window, time-triggered traffic is transmitted and in the second one, event-triggered traffic is transmitted. The TM contains the scheduling that the EC has to follow, specifying the set of messages that the slaves must transmit in the specified window.

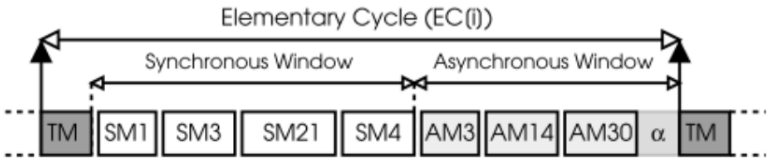
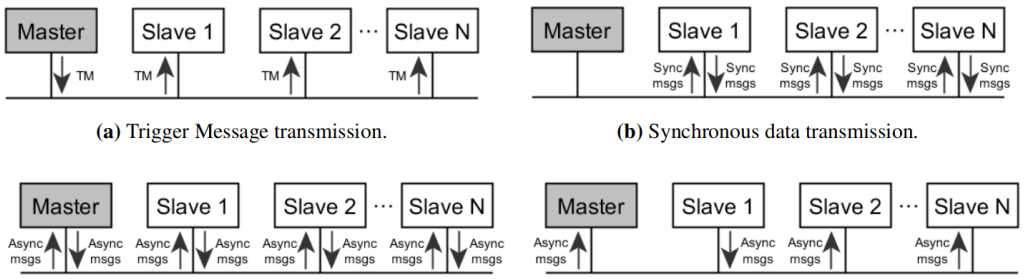


Figure 2.2: EC structure (from [1]).

The communication pattern defined by FTT for each EC is shown in figure 2.3. The first step begins with the master marking the opening of an EC by sending the TM (a), which is received and decoded by the slaves. The second step is the exchange of synchronous messages during the synchronous window according to the EC-schedule contained in the TM (b). These messages contain the output data from the applications executed inside the slaves. The third step is the exchange of asynchronous messages during the asynchronous window between the master and the slaves without any type of scheduling (c). The forth step finishes the pattern by adding spare time at the end of each EC to prevent any transmissions from slaves being done before the current EC ends.



(a) Trigger Message transmission.

(b) Synchronous data transmission.

(c) Asynchronous data transmission.

(d) Spare time at the end of the EC for slower slaves.

Figure 2.3: FTT communication scheme (as it appears in [1]).

ECs can change their size in order to meet the requirements set by the system. Depending on their size, the communication will have different characteristics. Mainly, longer ECs enable more data to be carried, decreasing the amount of information necessary to control the communication, that is, the overhead. The shorter the EC,

the better the responsiveness of the communication subsystem, given that the time between cycles is also reduced.

2.1.2 FTT Messages

In FTT messages can be classified in two main groups. Figure 2.4 shows the different types of messages common to all version of FTT.

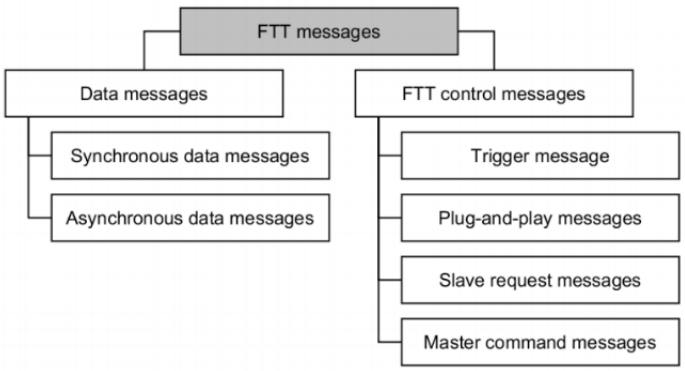


Figure 2.4: Types of messages in FTT (also from [1])

.

As depicted, the two main groups are the *data messages* and the *FTT control messages*. Data messages are used to exchange application data between slaves. On the other hand, FTT control messages are used to manage the communications which always involve the master, either as the receiving or transmitting node.

As mentioned before, depending on the temporal behaviour of the messages, they are classified in synchronous or asynchronous. Synchronous messages become ready for transmission periodically, where this period of time is defined by a value multiple of the EC's length. Asynchronous messages are transmitted irregularly.

Regarding the FTT control messages, we won't go into much detail about them as they are not of much importance for this project. Basically, the *slave request messages* and the *master command messages* negotiate changes in the real-time parameters between the master and the slaves. Finally, the plug-and-play messages are used when a slave wants to join the FTT network at run-time.

2.1.3 FTT Streams

In FTT, the messages are contained in virtual communication channels so called *streams*, which are managed by the master, but all the operations are triggered distributively be the slaves. Slave nodes that want to communicate with others will connect to a stream as a publisher or a subscriber, depending if they want to transmit or receive data. Only one node can be connected as a publisher, but various can be connected as subscribers. This way, FTT can transmit data without having to worry about point-to-point transmissions, that is, without having to know the destination of the message. The same situation happens from the receiver's point of view. Furthermore,

the creation, modification or destruction of FTT streams, as well as the attachment and unattachment of a node from a stream are also requested by slaves.

The following list explains the different attributes that streams have in order to suit the type of communication between the nodes. These attributes are specified in an application executed by a slave when creating a stream or having to operate with it:

1. *type*: the message type as synchronous or asynchronous.
2. *period*: the interval or time of the stream, that is, the number of ECs it takes between transmissions.
3. *size*: specifies the size of the message.
4. *offset*: delay in ECs with reference to the activation time.

2.2 Hard Real-Time Ethernet Switching (HaRTES)

HaRTES is an implementation of the FTT communications paradigm. More specifically, HaRTES is a switched-Ethernet communication network in which the switch itself is used as the master node where the slaves are connected to transmit in *star* form. Since all the communications pass through the switch, the master can manage them.

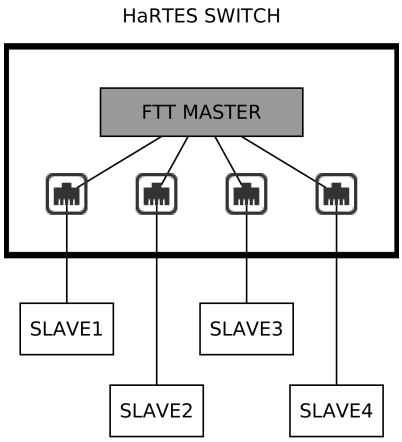


Figure 2.5: A switch in HaRTES.

These are the most important aspects of HaRTES:

1. Asynchronous traffic is sent automatically by slave nodes without having to pass it through the master. The switch can save these messages into a memory slot and send them later to the receiver, even if it doesn't use FTT.
2. Non-authorized or failed transmissions can be easily handled by the switch because it can block one or more of its ports if a node is not behaving as expected. For example, if a node fails, the switch can contain the information transmitted by this node to protect the others from creating more unwanted behaviour.

3. There are no collisions in the communication process since every node has a dedicated connection with the switch and they don't need to share the medium. The switch is in charge of organizing the communications with each node.

4. Integration of the traffic that doesn't use FTT, without affecting real-time services.

2.3 *Sentilo*

As mentioned earlier, this project entails using an IoT platform to control and monitor a system through the *cloud*. This is where *Sentilo* comes into play. We refer to the *cloud* as an online internet service that provides data storage and a way to manage it. In this case, *Sentilo* is going to be this *cloud* service, providing us a storage service for our sensors and actuators and a way to manage these values for further processing.

More specifically, this platform will act as a union between an application and the sensors and actuators of the system we want to monitor and/or automate. The platform's architecture is shown in figure 2.6. As observed, the architecture is divided in three layers. The bottom layer corresponds to the physical devices such as sensors and actuators that upload the information to the second layer. This second layer, placed in the middle of the image, is where Sentilo is in charge of managing all the incoming requests like publishing or retrieving data. Finally, the layer placed on top represents the applications that use the data from the sensors/actuators stored in the middle layer.

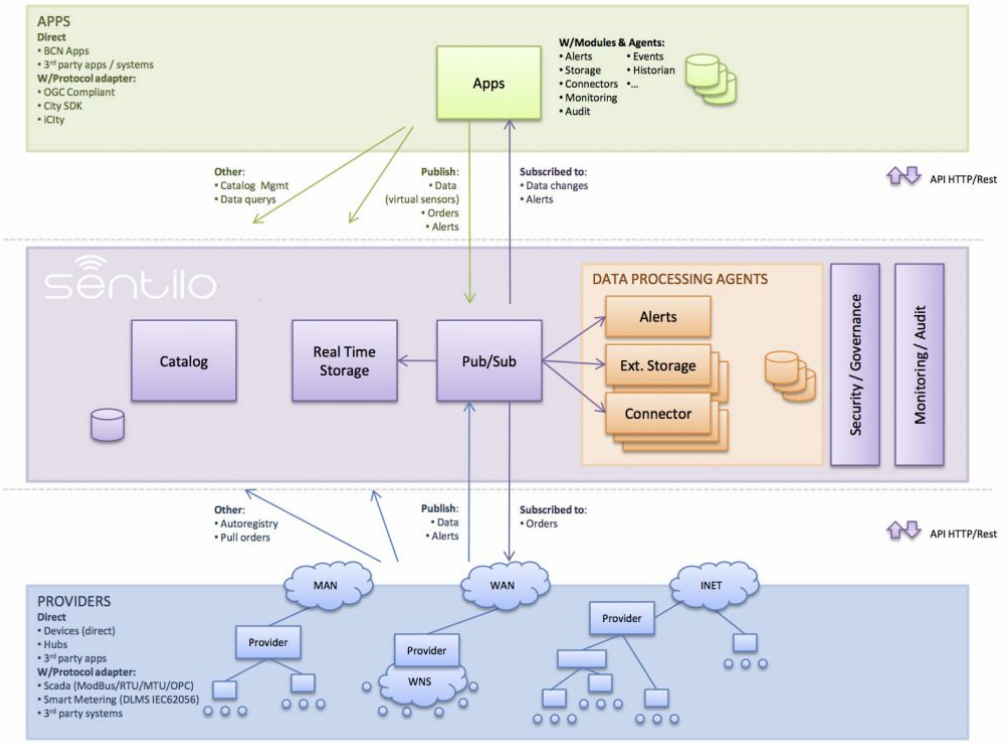


Figure 2.6: *Sentilo*'s architecture

From the user's point of view, in order to create a virtual instance of a sensor in the platform and enable the publication of its retrieved data, Sentilo uses different entities to structure the internal architecture. These entities are *components*, *providers* and *sensors*.

To enable the publication of sensor data, the first entity needed is the provider, which is the entity that manages devices (sensors) and is in charge of sending the data. Next, a component is a device that contains one or more sensors, such as a Raspberry Pi. They are not required in order to publish or read data but they are used to group together sensors that share a set of properties such as location, connectivity, power, etc. Moreover, components enable the representation of the sensors in the built-in map view provided in the platform. When you select a component, a pop-up window is opened and displays the list of sensors related to it with the last activity for each one of them. Finally, the sensors from Sentilo organise the different sensors that the component contains.

2.3.1 The API

Sentilo provides the user with a Representational State Transfer (REST) Application Programming Interface (API). An API is, in computing, a set of functions and procedures allowing the creation of applications that access the features or data of an operating system, application, or other service. REST is the technology used to implement this API, which is the most commonly used nowadays for simple applications. Simply put, the API will allow us to manage the data between our system and Sentilo. For more information about the API check [5].

The basic services that this API offers are the following:

1. **Data:** allows the client to read, write or delete the observations of the registered sensors.
    a) Publish observations of a sensor.
    b) Read observations from a sensor.
    c) Delete observations from a sensor.
2. **Order:** allows the client to send or retrieve orders to sensor/actuators.
    a) Publish orders.
    b) Retrieve orders.

These are the very basics of the service that the API offers, and will be the only commands used during the project. An example of another service it provides that hasn't been used is subscriptions. This allows the platform clients to subscribe to system events like orders, data or alarms related to the sensors. To do so, it is necessary to specify an endpoint to *Sentilo*'s platform, meaning that a server is needed. On this occasion this wasn't implemented, but it would be an interesting point to focus on for future projects.

2.3.2 HTTP

To be able to use all the services explained previously, it is necessary to use Hypertext Transfer Protocol (HTTP), a communication protocol massively used to transfer information through the World Wide Web. More specifically, we will make use of HTTP requests, where an HTTP client sends an HTTP request to a server in the form of a request message. A simple example of an HTTP request is the simple fact of writing the Uniform Resource Locator (URL) of a page we want to access in the navigation bar of a web browser. This request is processed by a web server which replies sending back the content of the web page.

CHAPTER 3

DESIGN

This chapter contains a description of the design of the project. To have a better understanding of the project we will make mention of the diagram shown in figure 3.1.

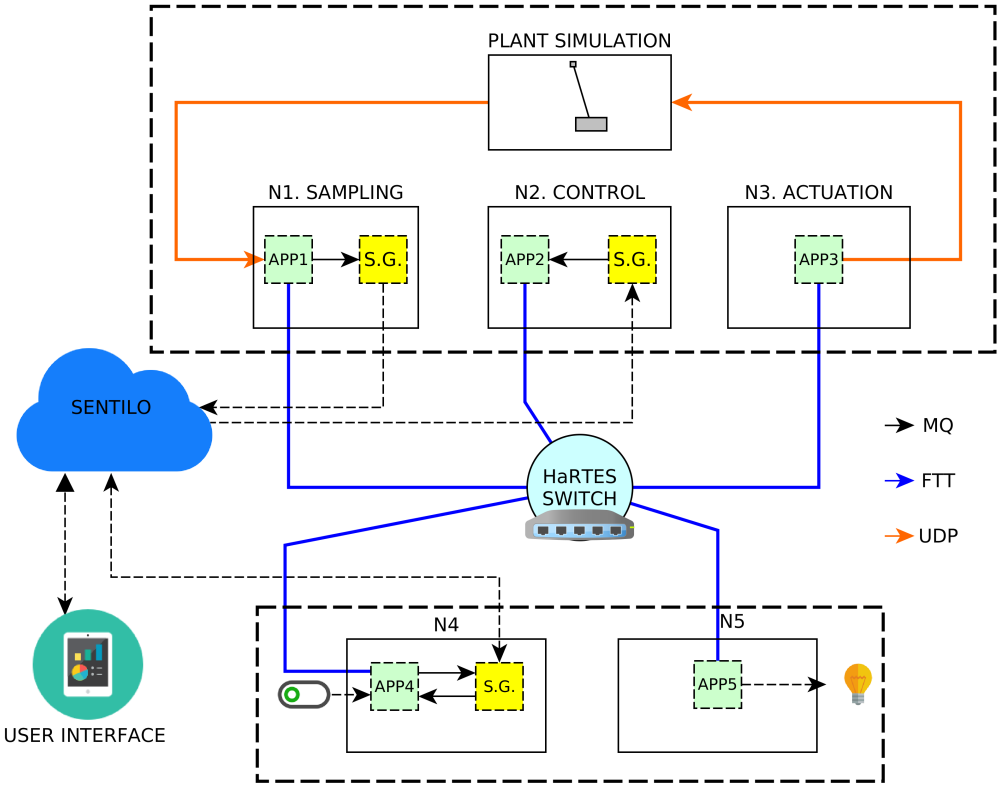


Figure 3.1: Diagram of the project design.

This diagram represents the structure of the system, indicating the way all the

hardware and software is interconnected. The set of tasks executed in the system can be divided into two different blocks, the domotics task and the control task. These blocks are depicted in the figure using rectangles with non-continuous borders. Note that we are talking about tasks and applications. To avoid any confusions, the criteria that has been used in this document is so: *tasks* make reference to the overall logic used to carry out the functionalities of the project, being the domotics task and the control task; *applications* (or apps) refer to the software programs used in HaRTES to implement the different processes needed to carry out the tasks.

Following on, the diagram represents the nodes that conform each task, which are physical devices that together carry out a task. Other hardware parts are the plant simulation, which is a node executing the simulation of the inverted pendulum needed for the control task and the HaRTES switch. Also, the switch connected to the fourth node *N4* is a physical switch. The software parts depicted in the diagram are the HaRTES apps executed in the nodes, the Sentilo platform and other processes (*SG*, 4.4) that will be explained further on. Moreover, a user interface will be implemented to enable us a way to control and monitor the system. This is a software application which is executed in any node provided with internet connection without having to be directly connected to the system.

Notice that the hardware parts of the system are depicted with blocks with continuous borders and, on the other hand, software components are indicated with non-continuous borders.

One point to clarify is that the system consists of two different networks, the first one being the HaRTES network and the other one being Wi-Fi. The first one is used by the apps to communicate among them, providing a real-time communication and all the properties that this protocol offers. The second network is mainly used for instrumentation purposes. That is, it makes it possible for us to remotely connect to all the nodes in order to monitor and control their operation during run-time so that we can test the implementation in a comfortable way. Moreover, this Wi-Fi network is also used to communicate the nodes with Sentilo and the nodes involved in the control task with the simulated plan.

As observed in the diagram, all parts of the system are inter-connected with a specific type of communication. These different types of communication are colour-coded to indicate which one is used in each node. Three different protocols have been used: *MQ*, which stands for *message queues*, has been used to communicate processes within the same device (or node); *FTT*, as explained in the previous section, is the protocol used by the HaRTES network; finally, *UDP* is a standard communication protocol that in this case was used to communicate the DES with the simulated plant. All three of these communication protocols will be explained further on in the implementation chapter (4).

The following sections explain in more detail the parts of the project. To begin with, the HaRTES system will be explained with the corresponding tasks that were implemented. Following on we will explain how the Sentilo platform was integrated. Finally, we will give a brief introduction to the user interface.

3.1 HaRTES

Based on the diagram of the design of the project (3.1), each node uses a program called *APP*. These are the applications used in HaRTES and they are executed by a node (a device). The switch, which also functions as the master node, manages the communications between the nodes. These communications are represented with blue lines in the diagram and correspond to the FTT protocol. In the following sections we will explain the design of both tasks, the domotics and the control tasks.

3.1.1 Domotics task

Starting with the simplest task, its main goal is to simulate a button that controls a light switch. We want to be able to control the light from both the physical and the virtual switch. The physical switch will be an input of a General Purpose Input/Output (GPIO) of a node and the virtual switch will be a virtual sensor from Sentilo's platform. If the system detects a change in the actual state from one of these two control points, the light will have to change accordingly.

3.1.2 Control task

This task is more complex than the previous one. It consists in managing the control of an inverted pendulum, which is a pendulum that has its center of mass above its pivot point, referred to as the cart. It is unstable and without additional help will fall over. In order to complete this control task we will implement three different stages; sampling, control and actuation, each one of these stages being executed in different nodes, and consequently, different HaRTES apps.

The importance of this task is to make a HIL simulation. This is a technique that is used in the development and testing of complex real-time embedded systems. It consists in connecting the system to be tested to a simulation of the real plant in a way that it thinks that it is connected to the real plant. This way, all of the designing process of the embedded system won't affect the real plant and all of the testings can be executed in unlimited types of scenarios. After the HIL simulation is complete and the functioning of the system is verified, the system is ready to be applied to the real plant.

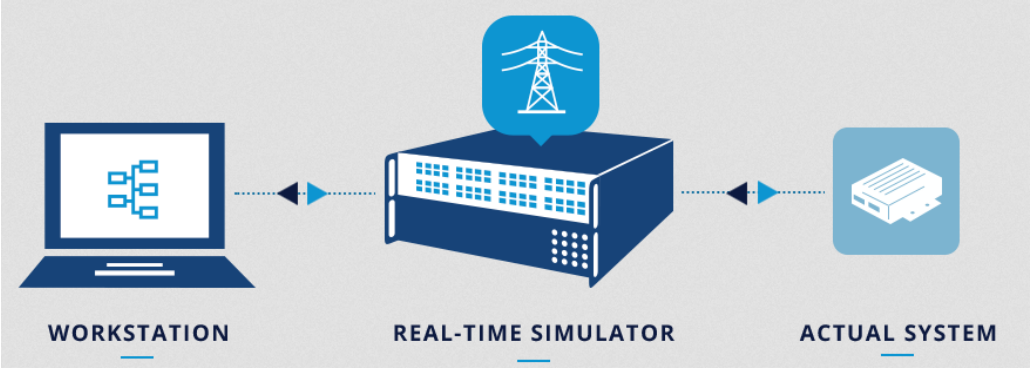


Figure 3.2: Hardware-in-the-Loop process.

In this case, the system under test is going to be the control of the inverted pendulum, the objective of which is to ensure that it doesn't fall to the sides after a force is applied to it. HaRTES will provide the system with real-time operation to ensure that the control process is executed in time, otherwise the pendulum would destabilize.

The simulation of the plant will be executed by another node running a graphical program to enable a visual real-time simulation of the inverted pendulum with working physics. This will be done with a program called *Processing*, as explained later in section (4.6.5). As observed in the diagram in figure 3.1, this node is separated from the HaRTES network. In fact, it belongs to the Wi-Fi network mentioned earlier, enabling the connection with the control system.

The first step that the embedded system needs to perform is the sampling of the variables of the simulated plant. This is done in the sampling stage, which obtains the values from the device executing the inverted pendulum via the Wi-Fi network. Once the sampling is done, the sampled values get passed to the control stage through the HaRTES network where the control operation will be executed. The control values calculated in the second stage are passed to the actuation stage, again via the HaRTES network, and finally, passed again to the inverted pendulum through Wi-Fi. The FTT communications, as explained in the HaRTES introduction, are managed by the master node (or switch). In the implementation section (4) we will go more in depth on how this system works.

3.2 Integration with *Sentilo*

The next step for this project is to enable a control through the *cloud*. We need to provide the system with a way to communicate with Sentilo's server to be able to upload the current state values and to read any changes that the user wants to apply to the system.

As mentioned before, we will make use of *curl* to perform HTTP operations between our system and *Sentilo*'s server using a library. The library has been written based on *Sentilo*'s API and it will enable us to do the basic operations. More details about this library are explained in section 4.4.3.

Each node in our system that needs to have a connection with Sentilo will be executing a program, separate from HaRTES, called Sentilo Gateway (SG), which will be in charge of all the processes that imply *Sentilo*. This way, the main program (HaRTES) will be less affected by the external processes needed to communicate with the *cloud*. It is also an advantage when it comes to integration, meaning that if we want to scale the system with different protocols or languages, the implementation of these will be easier. In section 4.4 we explain how the SGs work.

The following are the parts of our system that will need an SG. In the case of the domotics task, it will have one in the corresponding node and attached to one of the HaRTES apps that are needed to implement this function. In the case of the control task, there will be two; one in the sampling stage, used to upload data to Sentilo's server, and one on the control stage, used to download data from the server. The data that will be uploaded to Sentilo will be the position of the cart that sustains the inverted pendulum, meaning the horizontal axis. The data that will be downloaded into the system will be the position that we want the cart to go to, that is, a set-point.

3.2.1 *Thingtia*

To create an instance of this platform we have opted to use a cloud platform called *Thingtia*. It is a ready to use server with a free plan, providing all the functions from Sentilo with only a few limitations, but sufficient to carry out this project.

Thingtia enables the client to manage the sensors and actuators through their website, where the creation and configuration can be done in a graphical and intuitive way. It also offers a map and a graphical interface to view the latest data acquired from the sensors, which will be useful for the first tests.

3.3 The User Interface

The final part of the project consists of implementing a way to be able to monitor and control the system in a comfortable and intuitive way, enabling any type of user to interact with the system without any previous knowledge of how it works. The goal of the user interface is to provide information about the current state of both the domotics and the control tasks and to be able to change some variables, more specifically the position of the inverted pendulum and the switch of the button (on/off). To create this user interface we have opted to use a program called Node-red, which will be explained in a future section (4.6.6).

CHAPTER 4

IMPLEMENTATION

In this chapter we will go more in depth into how the project was completed, covering all the aspects explained in the previous chapter 3. To have a better understanding of how the project was implemented, this section is structured in a certain way; First, we explain the hardware used for the nodes. Secondly, we explain the implementation of both the domotics and the control tasks. Next, the carrying out of the SGs for each node is described. After that, we explain the user interface and finally we describe the tools and technologies used during the implementation of the project.

In order to implement the tasks in HaRTES we have used previous works that were also implemented with this network as a base. Nevertheless, all the work was redone given that the tasks where different, but it helped in order to understand the operation of such a network.

Although it is explained further on, before entering into detail about the implementation of the project, bear in mind that the major part was written in C (4.6.1) and the code that is referenced throughout the document has the syntax form of this language. The tools that don't use this language are Processing and Node-RED, which are also explained in section 4.6.

4.1 Hardware

The main focus on the hardware setup are the nodes. Each node consists of a physical device capable of executing the programs designed to carry out the project. Two types of nodes have been used and are explained below.

4.1.1 Raspberry Pi

For all the nodes except for the switch we have used Raspberry Pis, which are single-board computers, consisting of an ARM-based microprocessor, memory, GPIOs and other features required of a functional computer. The model used is the Raspberry Pi 3B+ and the specifications are shown in table 4.1.

Raspberry Pi 3B+	
Architecture	ARMv8-A (64/32-bit)
SoC (System-on-a-Chip)	Broadcom BCM2837B0
FPU (Floating-Point Unit)	VFPv4 + NEON
CPU	4× Cortex-A53 1.4 GHz
GPU	Broadcom VideoCore IV @ 250 MHz
	OpenGL ES 2.0, MPEG-2 and VC-1
Memory (SDRAM)	1 GB (shared with GPU)
USB 2.0 ports	4 (via on-board 5-port USB hub)
Video outputs	HDMI, DSI
On-board storage	MicroSDHC slot, USB Boot Mode
On-board network	10/100/1000 Mbit/s Ethernet
	(real speed max 300 Mbit/s)
	802.11b/g/n/ac dual band 2.4/5 GHz wireless
	Bluetooth 4.2 LS BLE
Low-level peripherals	17 x GPIO's
Power source	5 V via MicroUSB or GPIO header
Size	85.60 mm × 56.5 mm × 17 mm
	(3.370 in × 2.224 in × 0.669 in)

Table 4.1: Specifications of the Raspberry Pi used.

As you can see in the specifications, each Raspberry Pi has an on-board network Ethernet port that will enable us to communicate them with the switch. It also provides a Wi-Fi connection needed to communicate with the simulated plant and to be able to access Sentilo.

4.1.2 Mini-PC for the switch

For the switch we need a different piece of hardware, given that it needs to be able to interconnect the rest of the nodes and that it may need more computing capacity to manage all the communications. A *Computer Mini-PC 10 Lan* has been used from *Ibertronica* [6], and the main specification that it offers for our project is that it has 10 network interfaces, enough to connect all the nodes needed for this project. The table 4.2 shows the specifications of this model.

4.2 Domotic Task

In this section we will explain the process that was followed to implement the task of the button mentioned in section 3.1.1. As mentioned before, the objective is to control a switch from either the local system or from *Sentilo*. Looking at the diagram of the design (3.1) you can see that this task uses two nodes. The first one is attached to the local switch and the second one has a light output, to show an example of where it would connect to.

First of all, the first app located in the first node has to read the GPIO value of the switch. In *raspbian*, which is the operating system that we used in the Raspberry Pis,

Mini-pc Ibertronica	
Model	ORDMINI10LAN
CPU	Intel Celeron J1900 SoC Quad-core 2.0GHz
	2.42GHz Burst, 10W TDP, Bay Trail
RAM	SODIMM DDR3 4GB Crucial
Storage	SSD 120 Gb 2.5"
Network Interfaces	10x Intel 211-AT gigabit LAN
Back Panel	1x USB 3.0
	3x USB 2.0
	1x HDMI
	1x VGA
	2x RJ45
	1x Serial Port (RS232/422/485)
Power Input	12V DC

Table 4.2: Specifications of the Mini-PC.

the control over the GPIOs can be made with file descriptors, which are indicators used to access a file or other input/output resource. To initialize the GPIO we first need to indicate its direction, meaning if it is an input or an output. In this case we indicate it as an "in". Once that is done, we can proceed to read the value. In the following code you can see how this initialization is done:

```
FILE* fd = fopen("/sys/class/gpio/export", "w");
fprintf(fd, "%d", gpio);
flose(fd);

sprintf(buf, "/sys/class/gpio/gpio%d/direction", gpio);
fd = fopen(buf, "w");
fprintf(fd, "in");
fclose(fd);
```

The reading of the value is done by a thread; this way it can continuously read in parallel with the main program's execution and, whenever the button is pressed, the value of a local variable will change as well, so that the main program can access its current state.

The connection of the button with the GPIO is done with a pull-down resistor configuration. This is done because when the button isn't pressed, we want to make sure that the input value is a digital low, and this is done by enabling a direct connection to ground when the button isn't pressed (normally open switch). The following image 4.1 shows the schematic of a pull-down resistor configuration.

The next objective for this app is to send the value of the button to the second app in the second node, where the output will be used to control a peripheral. This is done through an FTT stream. The app first creates the stream and then it attaches to it as a publisher. From this point on, the HaRTES program will wait until another app is subscribed to the stream. If there is at least one subscriber, it will start sending the content of the stream in each transmission cycle, depending on the period value specified in the corresponding stream parameter.

The second app from the second node is in charge of receiving the value of the

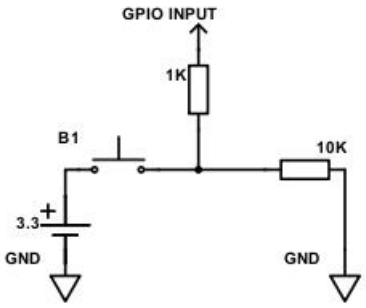


Figure 4.1: Schematic for the button connection with pull-down resistor.

button. We achieve this by attaching the app as a subscriber to the stream created by the first app and continuously read the value sent in each cycle.

4.3 Control Task

This section gives an explanation of the process implemented to carry out the control over an inverted pendulum as explained previously in section 3.1.2. First of all, we are going to explain the way the simulated plant works and next we will explain each HaRTES app that provide the control of the plant.

4.3.1 The Simulated Plant

As already stated, to carry out the simulation of the inverted pendulum we have used Processing, which is explained in more detail in section 4.6.5. The program used for the inverted pendulum has not been implemented in this project but has been obtained from a *github* page ([3]). To clarify, all the physics and control of the pendulum have not been part of this project. Our objective here is to extract the control of the pendulum that is already implemented in this program and execute it in another system, which we refer to as the control system.

Following on, we will explain how the program in Processing works, pointing out each part and its functionality. To do so, we will use pseudo-code to have an overall understanding without any previous knowledge of the programming language.

To have an understanding of how Processing works, it consists basically of two functions that are executed in different ways.

1. *setup()*: allows initializations needed for the correct execution of the program, for example, specifying the size of the display, initiating communications, etc. It is executed only once at the very beginning of the program.

2. *draw()*: this is the main function of this program and where all the graphics are drawn in the display. It is executed in a loop but it can also be controlled by

events. It also contains functions that don't imply any graphical modifications but need to be continuously executed.

```
// Pseudo-code of the Inverted Pendulum executed with Processing //

*Functions and variables*

void setup(){

   -> Initialize UDP communications.
   -> Initialize display.

}

void draw(){

   -> Manage mouse inputs used to apply a force to the pendulum.
   -> Send state variables through UDP to the control system.
   -> Update the state of the Inverted Pendulum.
        + Increment time variables.
        + Set Accelerations.
   -> Draw indicator of force applied to the system.
   -> Redraw the pendulum depending on the updated state.

}

void keyPressed(){

   -> Read any keyboard input values that control the simulation.

}

void receive(){

   -> Receive and manage incoming UDP messages from the control system.

}
```

To be able to use UDP with Processing we need to install a library. To do so, using the programs Integrated Development Environment (IDE), we can navigate to Sketch > Import library > Add Library... and search for UDP. The first library that shows up is the one used in this project. The image 4.2 shows the library used.

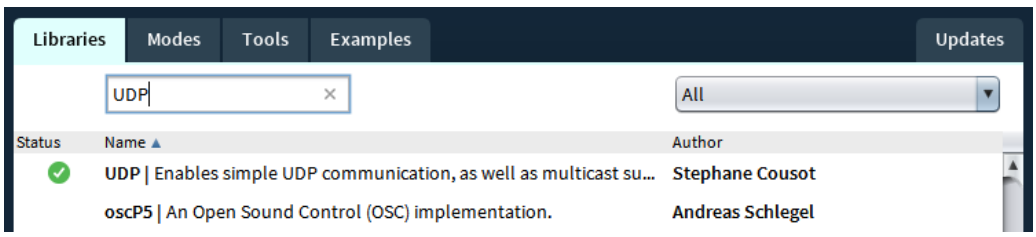


Figure 4.2: UDP library used in Processing.

As stated in the pseudo-code, each iteration of the draw function executes a function where the variables of the current state of the pendulum are sent through UDP. More

information on this communication protocol is found below in section 4.6.4. The rate that this loop is executed at, which is the same rate that the UDP message is sent, is periodic and dependent on the Frames Per Second (FPS) that Processing is able to operate at. This is because the draw function draws a new frame in the display every time the function is executed. Having said that, the control that we have over the FPSs in Processing is not absolute, meaning that the maximum rate will be achieved with no possibility to set a specific number for FPSs. This causes a problem in the simulation of the inverted pendulum because in each iteration of the draw function the state of the pendulum is updated with a time increment variable (dt). This time variable provokes the changes on the physics of the pendulum, since time is a parameter needed to simulate such process. Given that we cannot control the FPSs, to get a real-time simulation it comes down to modifying the value of the time increment variable dt.

Below, figure 4.3 shows an example of what the display looks like once the program is executed. The small vertical line indicates the set-point at which the pendulum has to stabilize. Below that, there is a visual representation in red lines of the current force that is applied to the cart in order to get to the set-point.

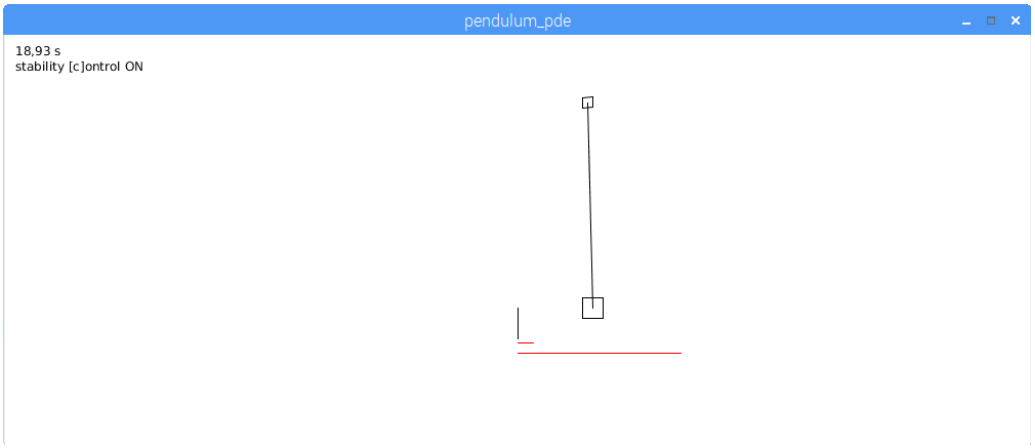


Figure 4.3: Display of the inverted pendulum simulation.

4.3.2 The Control System

In this section we will explain the process that was required to carry out the control of the inverted pendulum. As already mentioned, the control of the inverted pendulum was implemented in the code used for the simulation, and our goal is to separate it from Processing and implement our own control on a distributed embedded system using HaRTES. We will explain how each app works and how they communicate between each other and with the simulated plant.

Sampling

The first stage consists of obtaining the variables from the inverted pendulum needed to perform the control on it. To begin with, we need to have a connection between this stage and the plant. Given that the connection between the plant and the control

system is made through Wi-Fi, we use UDP (4.6.4) to transfer this data. The explanation of UDP from the point of view of Processing has been explained in the previous section (4.3.1). From this point of view, UDP is managed through a C library, given that the HaRTES program is written in this language.

To iniciate the UDP communications in this stage, we use a function called *init_server()* that was implemented in the HaRTES program prior to this project. It basically binds the socket to the specified port passed through as a parameter. The socket is returned by the function and it is then used for the function in charge of the reception. After this initialization, a thread called *sampling()* is created which will execute the reception of the UDP message. It basically uses a function from the library called *recvfrom()* and, after receiving the message, it assigns the content of the message to the corresponding variables. This is what the reception function looks like:

```
recvfrom(sckt, (char *)buffer, MAXLINE,
         MSG_WAITALL, ( struct sockaddr *) &cliaddr,
         &len);
```

As you can see, the socket is passed through as the first parameter. The second parameter is a char pointer that points where the message is stored. Next, MAXLINE defines the length in bytes of the buffer. The next parameter is a flag that specifies the type of message reception. In this case, MSG_WAITALL specifies the function to wait until all of the bytes are stored in the buffer defined by the length in the previous parameter. The last two parameters are the variables where the address of the sender and its length are stored.

One thing to point out is that each time the sampling stage is executed, the program reads the last UDP message sent by the simulated plant, thus the transmission and the reception are not synchronized. But this is not a problem because the plant is sending information at a higher rate than the control is processing the receptions and only the last message sent by Processing is received.

Once the reception of the message is carried out and the variables are correctly assigned in this program, we need to pass the necessary parameters to the control stage. To do so, an FTT stream (2.1.3) needs to be created and we need to indicate that this app will be a publisher in this stream. This is done using these two functions:

```
APP_STREAM_create( &app_stream1 );
APP_STREAM_attach_pub( &app_stream1 );
```

The next step is to create the message that will be sent to the control stage and to send it. To create the message we use a transmission buffer and we copy all the variables into it, which are of type double. To send this message we use:

```
APP_STREAM_send( &app_stream1, tx_buff, BLOCK );
```

This function waits until the next transmitting cycle (specified by the *BLOCK* flag in the parameters) and sends the message (*tx_buff*) through the stream previously created (*app_stream1*).

Control

The control stage is responsible for calculating the force that the pendulum needs to apply to its base so that it doesn't fall to the sides. Apart from keeping the pendulum stabilized, its goal is also to keep the cart on the horizontal set-point. Once the sampling stage has published in the stream the new values needed for the calculations, this app needs to read them, execute the calculations and finally, send the resulting force value to the actuation stage.

The first step is to receive the values published by the previous stage. To do so, this app needs to be subscribed to the stream that the sampling app created. In order to achieve this, a similar function as the one to make an app a publisher is used. Once the app has subscribed to the stream it can then use a receive function to read the message sent through the stream. These are the two functions explained:

```
APP_STREAM_attach_sub( &app_stream1 );
APP_STREAM_recv( &app_stream1, rx_buff, BLOCK );
```

As you can see, the receiving function also uses a *BLOCK* flag, meaning that the program will wait until the message is received before continuing with the execution of the program. Note that using this blocking mechanism during the communications with the apps facilitates the synchronization between them.

Once the values from the stream are read, the app proceeds to calculate the control function. As already stated, the control that is being used in this app is taken from the inverted pendulum simulation program stated earlier in 4.3.1. For more information about the control function, there is a mathematical explanation for obtaining the control function in the web page referenced ([3]). The control function breaks down to this line of code:

```
// APPLY CONTROL
F = -theta*100 - omega*50 + v*10 + (x-x_goal)*3.0;
```

As you can see, the function needs five variables that represent the state of the inverted pendulum to be able to calculate the force needed:

1. *theta*: angle of the pendulum from the vertical.
2. *omega*: the pendulum's angular velocity.
3. *v*: the cart's velocity.
4. *x*: the cart's position.
5. *x_goal*: the cart's goal position, which is set with the set-point.

The next step for this app is to send the calculated force value through another FTT stream to the next stage, which is the actuation stage. To do so, it uses the same process as the first app to create a stream and to attach to it as a publisher. In this case, the stream is called *app_stream2*.

Actuation

This is the final stage needed to complete the control over the inverted pendulum. Its task is to receive the force calculated in the previous stage and send it to the simulated plant.

As explained before, to receive the values from a stream, the app needs to attach itself to it by using the *APP_STREAM_attach_sub()* function provided by HaRTES. Once this is done, this app needs to send the calculated force value to the plant.

UDP is also used in order to send this message, which is formatted with the JavaScript Object Notation (JSON) format. This format will enable the simulated plant to easily parse the information which is explained in section 4.6.7. The following code shows how the message is constructed:

```
1 //MESSAGE SENT TO PROCESSING
2 sprintf ( message, "{\"F_control\":\"%f\",\"x_goal\":\"%f\"}", F_control, x_goal
      );
```

4.4 Sentilo Gateway

As explained in section 3.2, in order to integrate Sentilo into the system, each node that needs to communicate with the platform will make use of an SG, which is responsible for integrating this communication into the platform. In this section we will explain how each SG was implemented in the nodes. Looking back at figure 3.1, you can observe the nodes that have an SG depending on their function.

To communicate the HaRTES apps with the SGs we used the message queue mechanism explained in section 4.6.2 for which a library was provided from the tutor of this project to ease the process. These are the functions implemented in this library that we will use:

1. *int sentilo_mq_open ()*: creates a new message queue.
2. *void sentilo_mq_close ()*: deletes a message queue.
3. *void sentilo_mq_send_data()*: sends a new message through a specific message queue.
4. *void sentilo_mq_recv_data()*: receives the next message from a specific message queue.

Whenever HaRTES apps or SGs have a need to communicate with one or another, they will need to first open a new message queue if there isn't one opened. Then, depending if they have to send or receive, they will need to use the send function or the receive function.

Each one of the SGs has a similar structure. The following pseudo-code shows a general idea of how they are implemented:

```
1
2 -> Initialize message queue.
3 -> Initialize Sentilo.
4
```

```
-> Wait for first message in the message queue.

while(1){

    if the SG's objective is to upload data {
        -> Receive message from the app through message queue.
        -> Send value to Sentilo.
    }

    if the SG's objective is to read data {
        -> Read value from Sentilo.
        -> Send value to the app through message queue.
    }

}
```

As can be observed, in the beginning of the code we have initializations. Next, the program waits for the first message received through the message queue. We do this to ease the testing process because the SGs are executed before the apps. This way, the program will not execute any operations until the system is running. Once the first message in the queue is received, the program enters a loop where it continuously carries out operations with Sentilo, sending or receiving information to the corresponding HaRTES app.

Following on, we will explain the SGs used in each task.
[1]

4.4.1 SG for the Domotics Task

For this task, a single SG was implemented in the first node, the objective of which is to both upload and download data from Sentilo. In the case of the control task, which is explained in the following section (4.4.2), the upload and the download are done in separate SGs.

On the one hand, whenever there is a change in the button state, the app needs to send the new value through a message queue to the SG. On the other hand, when there is a change in the virtual sensor from Sentilo, the SG needs to send the new value to the app, again through a message queue. Note that, because of doing both operations in the same SG, two message queues are needed. That is because the downloaded data and the data to be uploaded can't be sent in the same message queue, otherwise they would get mixed.

Another thing to point out is that the reception of the message queues in both the app and the SG are made using a thread. This is done because the reception of a message from a message queue is blocking, meaning that when the receiving function is executed, the program waits until data is received. If this reception function was placed in the main loop of each program, the loop would be interrupted.

4.4.2 SG for the Control Task

For the control task, two SGs were implemented, one for the first node executing the sampling app, and the other for the second node executing the control app.

[1] explicar como se han insertado los sensores en sentilo

Sampling SG

The objective for the sampling SG is to upload the value of the actual position of the cart. Once the HaRTES app has sent the value to the SG via the message queue, using the functions provided from the library *libsentilo*, which are explained in section 4.4.3, it needs to upload it to the platform. To do so, a PUT operation is executed with the following *libsentilo* function:

```
1 sentilo_opt ( json_ptr, "data", "put", sensor, value_mq );
```

The *sensor* parameter contains the sensor name corresponding to the carts position and the *value_mq* parameter contains the value received from the message queue, which is the one that we want to upload. In this case, the return message stored in *json_ptr* is used as confirmation that the operation was executed correctly.

Control SG

The objective of the control SG is to receive any changes in the set-point of the cart by performing a polling to the server. To do so, in the *while(1)* loop it executes a GET operation as follows:

```
1 sentilo_opt ( json_ptr, "data", "get", sensor, NULL );
```

In this case, the *sensor* parameter refers to the set-point (*x_goal*) and the value received is stored in the *json_ptr*. Next, the returned string from Sentilo needs to be parsed:

```
1 sentilo_jparse ( json_ptr, "value", return_str, NULL );
```

The object from the JSON string that we are looking for is "*value*", and its contained value will be stored in the *return_str*. If the value returned from Sentilo is different to the last value obtained, the program will send this value to the control app from HaRTES and will change its value.

4.4.3 The library, *libsentilo*

In order to carry out the integration of Sentilo into the system, a library programmed in C was made to enable the interaction with the platform. In this project it was used in the SGs, but its sufficiently generic so that it can be used to implement other parts of the system or for future projects involving this platform.

This section explains the functions implemented in this library. It consists of three main functions to provide us with the basic tools to manage data operations between our system and Sentilo.

4.4.4 *sentilo_init()*

As the name of the function indicates, it is a function with the objective to configure the initial state of the program enabling the correct execution of posterior uses of the other functions available in the library. It reads configuration parameters necessary to establish a connection with Sentilo's server. As explained previously in the design section (3.2.1), Thingtia is going to be used as a cloud server to create an instance of a Sentilo platform. When creating this instance, some parameters are created in order

to establish a connection with the cloud server. These configuration parameters are saved in a configuration file named "*conf*" that is found in the same directory as the library. The reason why these parameters are specified in a separate file is to provide a way to easily configure the server's parameters without having to edit and recompile the library, simplifying the work for possible future uses.

Basically, this function reads the text file and saves the values of each parameter in a variable. The main configuration parameter needed is the authentication token used by the server to establish a secure connection when performing HTTP requests. Another parameter is the name of the server's user and the provider we want to use. The provider needs to be created previously through Sentilo's server and it is always preceded by the user's id. For example, if the user's id is *foo_bar* and the provider's name is *test*, the way to reference this provider using this user's account would be "*foo_bar@test*". Finally, a parameter also needed is the URL of the server, which in this case is *http://api.thingtia.cloud/*.

4.4.5 *sentilo_opt()*

This is the general function needed to perform HTTP (4.6.3) requests to the server. It can perform *data* and *order* operations, and within these two types it can perform either a GET or a PUT operation. It uses the following parameters:

1. *char * json_ptr*: a char pointer used to store the result of the operation. It needs to have allocated memory prior to the execution of the function.
2. *char opt[]*: char array containing one of the two types of operations: *data* or *order*.
3. *char request[]*: char array containing one of the two types of requests: *GET* or *PUT*.
4. *char sensor[]*: char array containing the name of the sensor.
5. *char value []*: char array containing the value that wants to be published to the respective sensor. Only for *PUT* type requests.

To implement this function we first need to build the *url* depending on the type of request that the parameters of the function indicate, which is made with simple string concatenation. Once the URL is complete, we need to build up the headers for the HTTP request. HTTP headers allow the client and the server to pass additional information with the request or the response, and it's what we will use to indicate the authentication token to the server.

In order to append the HTTP headers to the URL we will make use of *curl*, an open source tool to transfer data through URLs that is explained in more detail in section 4.6.8. More specifically, we will use a library that implements curl called *Libcurl* ([7]. This library uses a function in which you specify the type of operation you want to perform, including the addition of custom headers. This function is called *curl_easy_setopt()* and all the different operations are documented in the API ([7]). To specify the headers we need to use CURLOPT_HEADER as a parameter and the next one will be a linked list of strings containing the information of the header or headers. This linked list is provided by the library as a *struct* type called *curl_slist* and a specific function is used

to add headers into it called *curl_slist_append()*. Having said this, after appending the authentication header to the linked list and specifying the type of request, we need to execute the operation with the function *curl_easy_perform()*.

If the operation specified in the parameters is of type *order*, another header needs to be specified to indicate the HTTP request that a *data* field will be sent. We also need to specify that this data message is using a *json* format. When performing this operation, the first parameter for this function (*char * json_ptr*) contains the message that wants to be sent as an order.

4.4.6 *sentilo_jparse()*

This function's name stands for *json* parse. Parsing consists of analysing a string of data, in this case a *json* string, with the objective of knowing the content referenced to each object that the string contains. Given that the libary is written in C and that JSON isn't oriented to use in this language, we made use of a library called *jsmn* ([4]) to ease the process. Basically, it offers predefined *struct*s and variables to organize the data contained in the JSON string.

The function is used with the following parameters:

1. *char * json_ptr*: a char pointer containing the string that wants to be parsed.
2. *char search_str*: a char containing the object that wants to be read.
3. *char * return_str*: a char pointer where the value of the object read is returned.
4. *char * timestamp*: a char pointer to store the timestamp read.

As you can see by the parameters given, the way this function has been implemented has the following idea. A *json* string is passed through the first parameter and the second parameter enables the user to specify from which object it should read the value. For example, if an HTTP request returns a *json* string containing data from a sensor and we wanted to parse its value, we would have to pass this string through the first parameter and then, with the second parameter, specify "value" as the object that wants to be read. This way, the function returns only the wanted information out of the *json* string.

4.5 The User Interface

Node-RED, a flow based programming tool explained in section 4.6.6, was used to implement the user interface. Its objective, as explained in the design section 3.3, is to provide the user with a way to monitor and control the system using the information stored in Sentilo.

Node-RED enables the installation of new blocks that provide new functions. In this case we have installed two additional plug-ins, one for Sentilo blocks and one for dashboard blocks. Sentilo blocks are used to perform HTTP operations to the server with ease, without having to manually set up the headers. Dashboard blocks permit the implementation of the user interface, managing inputs and outputs. Figure 4.4 hows how the final blocks are placed to achieve the desired results.

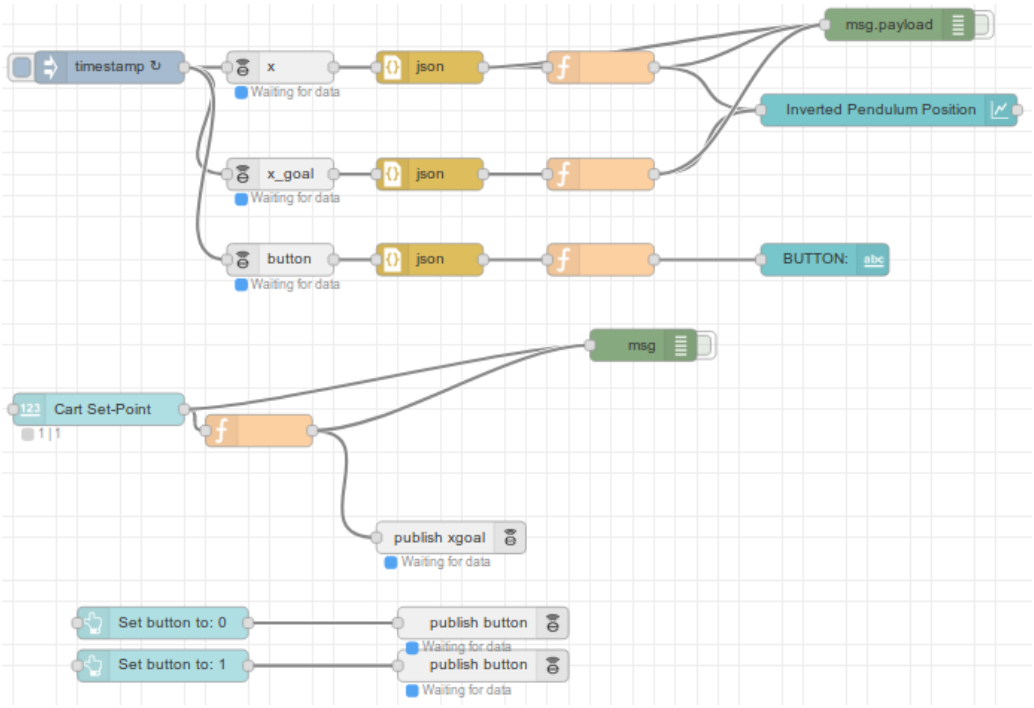


Figure 4.4: Node-RED's flow for the user interface.

4.6 Tools and Technologies

As previously mentioned, the majority of this project was written in C, the main reason being because the HaRTES program is already written in C. To carry out the project, Linux OS has been the main operating system used, as it's versatile and straight forward for these types of projects. The terminal simulator has been the main tool used for compiling, testing and running all the programs and libraries written through the making of the project. The following sections give a more in-depth description of the tools and technologies used.

4.6.1 C programming language

To program the apps for the nodes used in HaRTES and for the SGs, we have used the C programming language. This language is one of the most commonly used in general programming and is useful for this project for various reasons. The first is that it offers very good portability, meaning that a program can be run on different platforms without having to modify any changes or configurations. Taking into account that we are working with distributed embedded systems, this is helpful. Another important reason is because it's faster than the other typical programming languages, and, because we are working with real-time systems, it's an important point to consider.

To compile the programs, GNU Compiler Collection (GCC) has been used. It's a set of compilers for different languages, including one for C, and it's the standard for UNIX-like operating systems like Linux OS.

4.6.2 Inter-Process Communication (IPC)

The use of a separate program in a node to integrate Sentilo into the system, that is the SG, generates a need for a communication between the HaRTES app and the SG. To do so, we will make use of Inter-Process Communication (IPC)s. IPCs are one of the basic operating system mechanisms to allow the processes to manage shared data. One of the methods that this mechanism offers and the one we are going to use is *message queuing*. A message queue is a data stream similar to a socket, but which usually preserves message boundaries and allows multiple processes to read and write to the queue without being directly connected to each other.

An example of a system that could use a message queue would be in a web server. If this web server is receiving a lot of requests, to manage all the traffic, these requests could be stored in a message queue and multiple processes could repeatedly pick up messages once each one has finished the last request. This way the server could manage all the incoming traffic without collapsing.

The basic architecture of a message queue is simple. There are client applications called producers that create messages and deliver them to the message queue. Other applications, called consumers, connect to the queue and get the messages to be processed. Messages placed onto the queue are stored until the consumer retrieves them. The applications need to know which messages are addressed to them. To do so, messages entering the queue use an id provided from the sending applications.

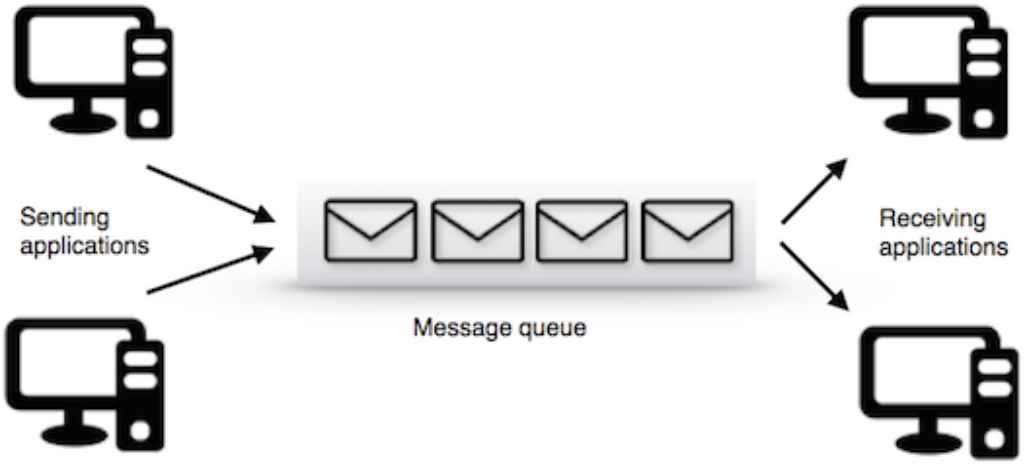


Figure 4.5: Architecture of message queues.

To make use of the message queue in our project, a library has been made to facilitate the implementation of each SG. This library provides the basic functions to send and receive messages from one process to another through the message queue. These messages will contain the information needed to upload to Sentilo's server, that is the sensor's id and the value that wants to be uploaded. The same happens when we want to download the data from Sentilo to our system.

4.6.3 HTTP

As explained in section 2.3.2, in order to enable a way to send data to Sentilo we use the HTTP protocol. HTTP requests are used to perform operations between a client and a server connected to the web.

In this project, the request will ask the Sentilo server to perform one of the operations that the API offers. After that, the platform will respond with an certain HTTP response depending on the type of request that was sent. The general form of a reply message in Sentilo consists of a *code* and a *message*. The code indicates if the request was accepted or not and if there were any errors. The numbers in this code give information about the type of error that has occurred, which are specified in the documentation. Also, if the message wasn't successful, the message field will contain information related to the error. On the other hand, if the operation was successful, the message will contain the data that was requested in the HTTP request.

4.6.4 UDP

As mentioned in the design section (3), the overall system uses two networks, the HaRTES network and a Wi-Fi network. Wi-Fi is used to communicate the simulated plant with the nodes, and to do so, UDP is used over it.

UDP is a protocol used in Internet Protocol (IP) networks and it's one of the core members of the Internet protocol suite situated in the transport layer. It consists of a connection-less communication, meaning that each data unit is individually addressed rather than prearranging a connection between the source and the destination. It doesn't provide much error detection mechanisms and thus there is no guarantee of delivery, ordering, or duplicate protection, but offers instead a direct and fast connection.

To establish a host-to-host connection, the applications need to bind a socket to the endpoint of data transmission. Sockets are a combination of an IP address and a port. A port is a software structure that is identified by the port number, which can vary from 0 to 65535 (16 bit integer value), and they are used to identify a specific process or a type of network service.

The fact that this protocol provides a fast connection is the main reason why it was chosen to use in this project, given that the data must arrive in time to prevent any delays in the control process of the pendulum. To be able to use this protocol we used libraries in both the simulated plant and the control system. The way it was implemented in each part will be explained in more detail below.

4.6.5 Processing

To simulate the plant we have used a program called *Processing*. Processing is an open-source graphical application built for programming visual designs with an easy-to-program language. It uses the Java language, with additional simplifications such as additional classes and aliased mathematical functions and operations. As well as this, it also has an IDE for simplifying the compilation and execution stage.

It also offers a built-in library installer which will be used to install the corresponding library for the UDP communications to enable the connection with the control system.

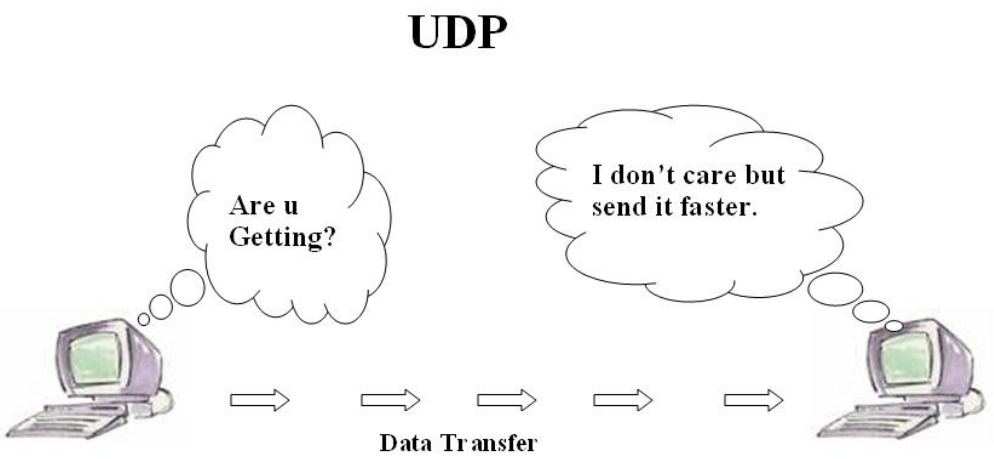


Figure 4.6: UDP's type of connection.

4.6.6 Node-RED

Node-RED is a flow-based programming tool for wiring together hardware devices, APIs and online services in certain ways. It works with nodes (also called blocks), and each one has a well-defined purpose. It's based in the run-time of Node.js, which is based on JavaScript.

This tool has been used to create the user interface, providing a real-time visual representation of the system's state and a tool to modify parameters.

4.6.7 JSON

JSON is a lightweight data-interchange format oriented for JavaScript objects. It is self-describing and easy to understand and it is language independent, that is, JSON uses JavaScript syntax, but the JSON format is text only. Text can be read and used as a data format by any programming language.

Here is an example of what a JSON string looks like:

```
{
  "firstName": "John",
  "lastName": "Smith",
  "isAlive": true,
  "age": 27,
  "address": {
    "streetAddress": "21 2nd Street",
    "city": "New York",
    "state": "NY",
    "postalCode": "10021-3100"
  },
  "phoneNumbers": [
    {
      "type": "home",
      "number": "212 555-1234"
    },
```

```
    {
      "type": "office",
      "number": "646 555-4567"
    },
    {
      "type": "mobile",
      "number": "123 456-7890"
    }
  ],
  "children": [],
  "spouse": null
}
```

Given that in this project Java was used on two occasions (with Processing and Node-RED), JSON has been used to ease the readings of data passed to these programs. But the main reason why it has been used is because Sentilo's HTTP responses also use the JSON format.

The downside of using this format based in Java is that, while using the C programming language, it's become tedious to read data strings that use this format given that it is oriented to Java objects. To ease this process, a library has been used called JSMN.

4.6.8 *Curl*

To be able to work with the HTTP protocol while programming in C, we will make use of *curl*, an open source tool used to transfer data with URLs and supports, among other protocols, HTTP. It can be used as a command line tool or as a library. In this case it will be used as a library, considering that a program will be written to make this project. The library that was chosen is *libcurl* [7], which is the most used in order to implement *curl* with C.

CHAPTER 5

FUNCTIONAL VERIFICATION

In order to verify that the system works correctly we performed tests with the hardware and software explained in this document. In this section the tests performed and the problems encountered during the development of the project will be explained.

5.1 Testing

The system needs to be able to execute both applications at the same time, given that the HaRTES system is developed with the intent of managing real-time communications in an efficient manner. The way this test was executed is by using the terminal emulator in Linux and executing in every node the corresponding programs required. The order in which the programs are executed is important, considering that some processes need information obtained by others in order to function properly. The important point to follow when initiating each program is to execute the master app in the switch last, the reason being all the programs wait until the program in the switch is executed. The user interface has also been tested and in some ways it has eased the testing of the system during the whole development process.

To execute the HaRTES apps in each node a connection to them is required, and in this case we used Secure SHell (SSH) to remotely access them. With SSH we can control each node through command lines, which is what we basically need to execute the programs. The image below (5.1) shows an example of how a connection to a node and the execution of a HaRTES app is carried out.

The way the apps are executed is by entering a command line specifying the app identifier that wants to be executed and the node identifier using the arguments *-a* and *-n* respectively. In image 5.1 you can see an example. In this case the app 1 is executed in the node 1. The command line to execute the HaRTES master app in the switch is similar but with different arguments. Here is an example of the command line needed for the execution of the master program in the switch:

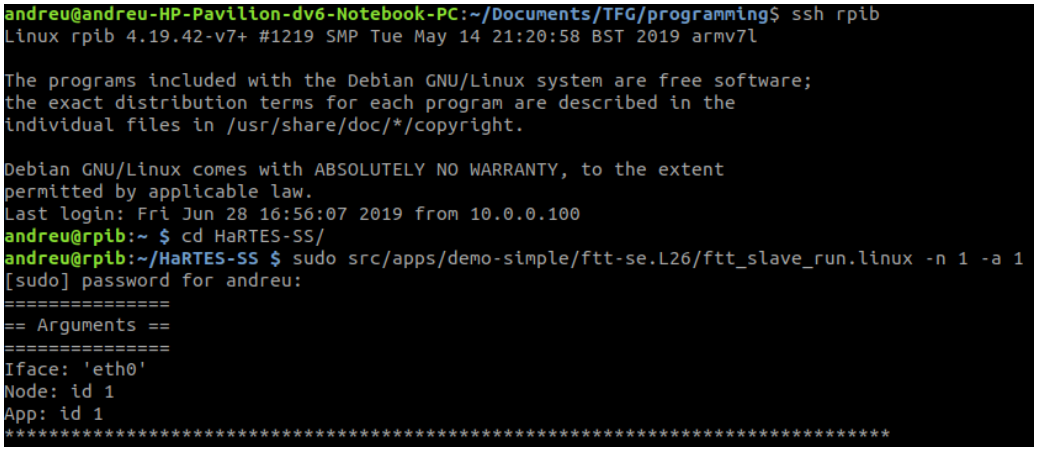


Figure 5.1: Example of an execution of a HaRTES app.

```
1 src/apps/demo-simple/ftt-se.L26/ftt_master_run.linux -S -i enp18s0 -i enp17s0 -e
    30
```

Argument *-S* indicates that the execution will be carried out with a standalone master. The following argument *-i* indicates the network interface where the apps are executed in incremental order. Finally, argument *-e* enables us to specify the time for each EC in milliseconds.

On the other hand, it is also necessary to execute the simulation of the plant before executing the master app. This is done with a Raspberry Pi that has an operating system with graphical user interface (GUI). Connected to this node there is a monitor to be able to see the display of the inverted pendulum. The execution of the simulation is simply done by executing the code with Processing. A mouse will need to be connected to this node to be able to apply external forces to the inverted pendulum by clicking and dragging the pointer in the direction that we want to apply the force. This way, we will be able to test that the control over the pendulum has been implemented correctly.

Once all the parts of the system are running, we can perform tests to verify that every task works. Figure 5.2 and 5.3 show an example of what the terminals look like once all the programs are executed and running in parallel.

To verify that the button task works, the physical button has been pressed and the change has been observed in the user interface. Changes to the value occur accordingly in the vast majority of cases. The reason why it can sometimes not detect the change is due to a bad connection of the button with the GPIO. On the other hand, if we try to change the button value from the user interface, which is equivalent to changing the value in Sentilo, we can observe the change happening in the reception of the value in the second node belonging to this task. That is, the node that will use the value of the button to control a peripheral. We can also observe the change of the value within the user interface, which is a direct representation of the value stored in the Sentilo platform after a *get* operation is performed by Node-RED.

The verification of the pendulum task can also be tested in different ways. First, the simulation of the pendulum can be tested without the control system by applying a force on it. The result is that the pendulum falls to one side and continuously moves because

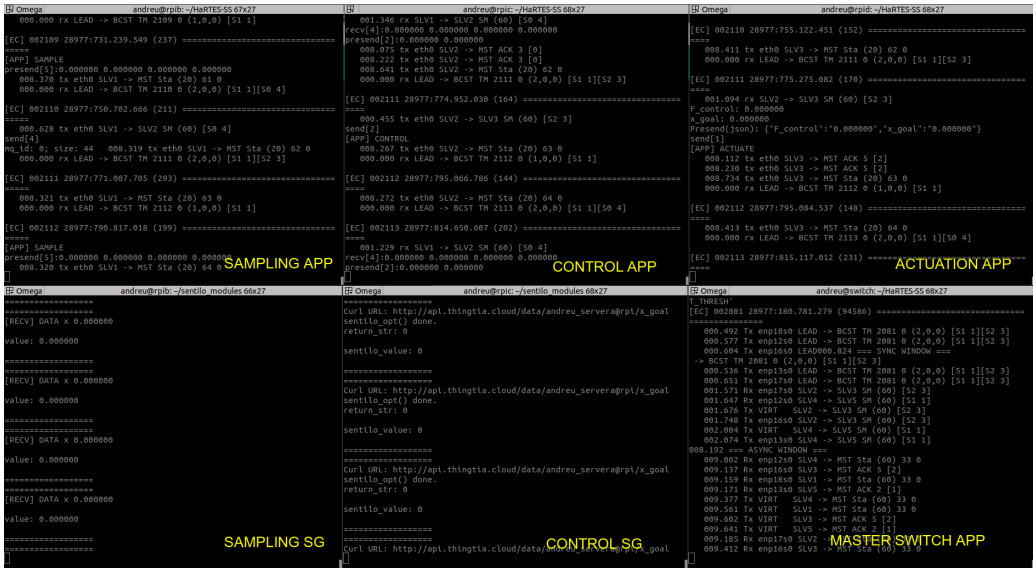


Figure 5.2: Example of the programs running in parallel in the terminal emulator (a).

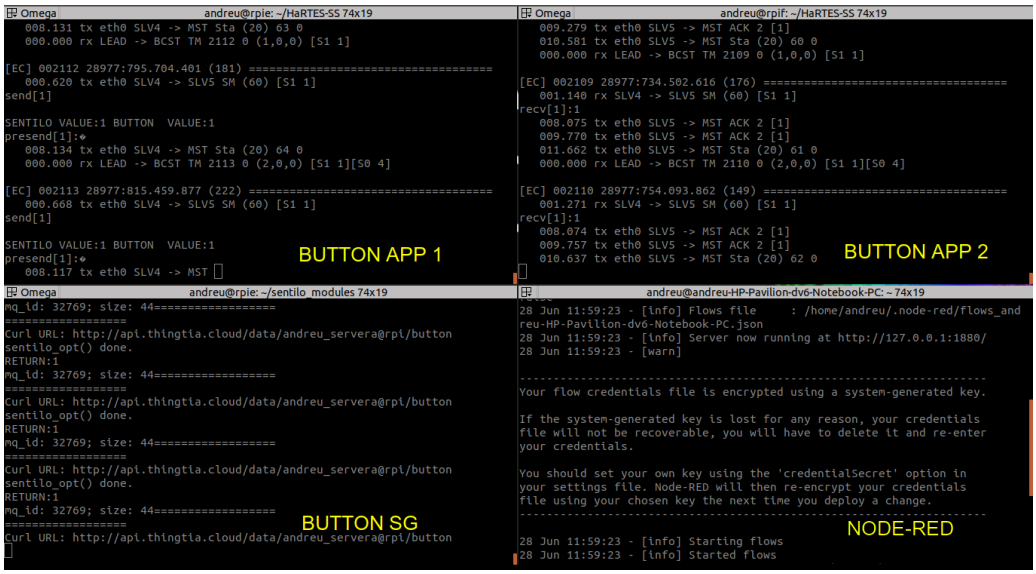


Figure 5.3: Example of the programs running in parallel in the terminal emulator (b).

of the momentum. This is behaving correctly because the physics are implemented in the simulation but the control over the pendulum is not. Also, the red horizontal lines that represent the force applied to the cart don't appear, because there is no value received from the control system.

Next, the control system needs to be tested. Once all the parts of the system have been executed properly and the control system is running, a force can be applied to the pendulum the same way as before, but in this case the pendulum receives the control actuation from the control system. The result is that the actuation is applied to the cart so that the pendulum doesn't fall and it slowly stabilizes to stay on the set-point.

The SG from the sampling node will upload the value corresponding to the position of the cart during this testing process and the result can be seen in the user interface. Figure 5.4 shows the user interface representing the result of the system after applying an external actuation to the pendulum with the mouse.

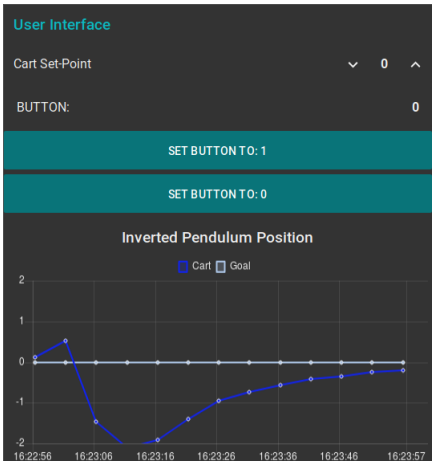


Figure 5.4: The user interface representing the results of an external actuation on the pendulum.

As observed in the chart, there are two lines, one representing the set-point of the cart and the other representing the position of the cart. After applying an external actuation to the pendulum, the cart changes position to prevent the pendulum from falling and then it slowly returns back to the position defined by the set-point.

Another test to be performed is to change the value of the set-point from the user interface. As expected, the cart will slowly make its way to the set-point without destabilising the inverted pendulum. Figure 5.5 shows the values of the position of the cart when changing the set-point from '0' to '1'.

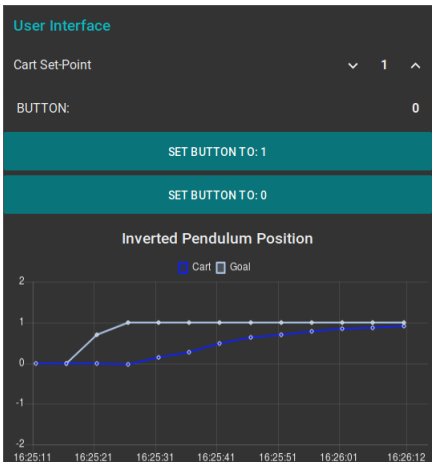


Figure 5.5: The user interface representing the results after changing the set-point value.

5.2 Problems encountered

During the testing of the system, the major problem encountered was in the control task. At first, the control over the pendulum failed after a period of time due to possible delays in the communications, considering that the communication between the simulated plant and the control system is made through Wi-Fi. To prevent this from happening we changed the value of dt which, as explained in section 4.3.1, is responsible to set the increments of time in each iteration. By lowering its value, the simulation has less changes in a period of time, thus giving more time to the control system to send the actuation value. A way to improve the performance in the system would be to change the Wi-Fi network for an Ethernet network, which has higher transmission speed.

CHAPTER 6

CONCLUSIONS

In this document we have described the design and implementation of a DES for the building *Ca Ses Llúcies* at the UIB. The objective is to demonstrate that is possible to execute several distributed tasks with different real-time requirements in the same network, as well as to monitor and control them through the Internet by means of an IoT platform. Finally, with the possibilities that this control offers, a user interface has been implemented in order to view the current state of the system and to perform changes in it.

In order to accomplish this, we have used HaRTES, a communication network based on Ethernet makes it possible for the nodes of a distributed embedded system to exchange traffic with different real-time requirements. Moreover, HaRTES can integrate traffic that is not FTT into the network, increasing its possibilities of use, for instance, in domotics, or in even more strict environments, such as the control of an industrial plant.

With reference to the IoT platform, we have used Sentilo a solution that is already been used in Ca Ses Llúcies and, thus, it is convenient for a complete integration in the building. Sentilo makes it possible for the applications to access to distributed sensors and actuators in a homogeneous manner without regard to the communication protocol or manufacturer.

The user interface developed in this project is an example of how an IoT platform can be useful in projects involving domotics, industrial plants or other fields like smart cities. The use of Node-RED as the programming tool to implement it also opens a whole new span of possibilities, while making use of Internet data to make new applications in a direct and intuitive way.

To sum up, this project has been a way to introduce someone with no experience at all to the concept of IoT and to get direct experience with DES that need specific communication requirements. Moreover, this has implied working with a multitude of different tools and technologies in order to integrate these two levels of the architecture.

Personally, to implement this project was very interesting, given that this new era of devices connected to the Internet is seemingly growing exponentially and is expected to

be an important field in the future. Moreover, the amount of programming carried out during this project has been a very good way to improve, as during the degree course, we only cover the basics and don't really go much in-depth. Also, to see how a relatively complex system gets to work for the first time is very satisfying.

CHAPTER 7

FUTURE WORK

In this final chapter we will make mention of possible future works that could be carried out to continue with this project in order to enhance it and provide more uses for it.

To begin with, prior to the implementation of this project, the UIB had carried out an implementation of a Sentilo server. As explained in this document, to create an instance of this platform we used a cloud server in order to carry out the project. Having said this, it would be interesting to implement this project using the server available at the University, given that it would offer more functionalities than the cloud server. One functionality that could be interesting is the ability to store data beyond the time span provided by the cloud server, in order to implement prediction algorithms.

As explained in the functional verification chapter 5, we used Wi-Fi to establish a connection between the simulated plant and the control system. While Wi-Fi is enough for a demonstrator, there are better solutions for implementing the communication in a distributed embedded real-time system. Having said that, a way to enhance the system would be to connect the simulated plant with a more suitable communication technology like, for instance Ethernet. Furthermore, the communication among the nodes and Sentilo also takes place through the Wi-Fi connection. Sending this traffic through the HaRTES network as non-real-time traffic would make the system less complex and easier to deploy.

BIBLIOGRAPHY

[1] A. Ballesteros and J. Proenza, "A description of the FTT-SE protocol," Tech. Rep., 2013. (document), 2.2, 2.3, 2.4

[2] UIB, "*SmartUIB*, an innovation project from the UIB." [Online]. Available: https://smart.uib.cat/ 1.1

[3] B. Martin-Anderson, "A minimal inverted pendulum simulation," 2016. [Online]. Available: https://gist.github.com/bmander/6cbaed2f9e686bb58553 2, 4.3.1, 4.3.2

[4] S. Zaitsev, "World fastest json parser/tokenizer." Updated 2019. [Online]. Available: https://github.com/zserge/jsmn 3a, 4.4.6

[5] Sentilo, "Sentilo's application programming interface." [Online]. Available: https://sentilo.readthedocs.io/en/latest/api_docs.html 2.3.1

[6] *Ibertronica*, "*Computer Mini-PC 10 LAN*." [Online]. Available: https://www.ibertronica.es/computer-mini-pc-10-lan.html 4.1.2

[7] *Open Source*, "*libcurl*, the multiprotocol file transfer library." [Online]. Available: https://curl.haxx.se/libcurl/c/ 4.4.5, 4.6.8